

# MICROPROCESSORS

*From Assembly Language  
to C Using the PIC18Fxx2™*

*Robert B. Reese*



DA VINCI ENGINEERING PRESS

# **MICROPROCESSORS**

## LIMITED WARRANTY AND DISCLAIMER OF LIABILITY

THE CD-ROM THAT ACCOMPANIES THE BOOK MAY BE USED ON A SINGLE PC ONLY. THE LICENSE DOES NOT PERMIT THE USE ON A NETWORK (OF ANY KIND). YOU FURTHER AGREE THAT THIS LICENSE GRANTS PERMISSION TO USE THE PRODUCTS CONTAINED HEREIN, BUT DOES NOT GIVE YOU RIGHT OF OWNERSHIP TO ANY OF THE CONTENT OR PRODUCT CONTAINED ON THIS CD-ROM. USE OF THIRD-PARTY SOFTWARE CONTAINED ON THIS CD-ROM IS LIMITED TO AND SUBJECT TO LICENSING TERMS FOR THE RESPECTIVE PRODUCTS.

CHARLES RIVER MEDIA, INC. ("CRM") AND/OR ANYONE WHO HAS BEEN INVOLVED IN THE WRITING, CREATION, OR PRODUCTION OF THE ACCOMPANYING CODE ("THE SOFTWARE") OR THE THIRD-PARTY PRODUCTS CONTAINED ON THE CD-ROM OR TEXTUAL MATERIAL IN THE BOOK, CANNOT AND DO NOT WARRANT THE PERFORMANCE OR RESULTS THAT MAY BE OBTAINED BY USING THE SOFTWARE OR CONTENTS OF THE BOOK. THE AUTHOR AND PUBLISHER HAVE USED THEIR BEST EFFORTS TO ENSURE THE ACCURACY AND FUNCTIONALITY OF THE TEXTUAL MATERIAL AND PROGRAMS CONTAINED HEREIN. WE HOWEVER, MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, REGARDING THE PERFORMANCE OF THESE PROGRAMS OR CONTENTS. THE SOFTWARE IS SOLD "AS IS" WITHOUT WARRANTY (EXCEPT FOR DEFECTIVE MATERIALS USED IN MANUFACTURING THE DISK OR DUE TO FAULTY WORKMANSHIP).

THE AUTHOR, THE PUBLISHER, DEVELOPERS OF THIRD-PARTY SOFTWARE, AND ANYONE INVOLVED IN THE PRODUCTION AND MANUFACTURING OF THIS WORK SHALL NOT BE LIABLE FOR DAMAGES OF ANY KIND ARISING OUT OF THE USE OF (OR THE INABILITY TO USE) THE PROGRAMS, SOURCE CODE, OR TEXTUAL MATERIAL CONTAINED IN THIS PUBLICATION. THIS INCLUDES, BUT IS NOT LIMITED TO, LOSS OF REVENUE OR PROFIT, OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THE PRODUCT.

THE SOLE REMEDY IN THE EVENT OF A CLAIM OF ANY KIND IS EXPRESSLY LIMITED TO REPLACEMENT OF THE BOOK AND/OR CD-ROM, AND ONLY AT THE DISCRETION OF CRM.

THE USE OF "IMPLIED WARRANTY" AND CERTAIN "EXCLUSIONS" VARIES FROM STATE TO STATE, AND MAY NOT APPLY TO THE PURCHASER OF THIS PRODUCT.

**MICROPROCESSORS**  
**FROM ASSEMBLY LANGUAGE TO C**  
**USING THE PIC18Fxx2<sup>®</sup>**

**ROBERT B. REESE**  
**MISSISSIPPI STATE UNIVERSTIY**



**DA VINCI ENGINEERING PRESS**  
Hingham, Massachusetts

Copyright 2005 by DA VINCI ENGINEERING PRESS, an imprint of CHARLES RIVER MEDIA, INC.  
All rights reserved.

Selected figures reprinted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Technology Inc.'s prior written consent.

No part of this publication may be reproduced in any way, stored in a retrieval system of any type, or transmitted by any means or media, electronic or mechanical, including, but not limited to, photocopy, recording, or scanning, without prior permission in writing from the publisher.

Editor: David Pallai

Cover Design: Tyler Creative

DA VINCI ENGINEERING PRESS  
CHARLES RIVER MEDIA, INC.  
10 Downer Avenue  
Hingham, Massachusetts 02043  
781-740-0400  
781-740-8816 (FAX)  
info@charlesriver.com  
www.charlesriver.com

This book is printed on acid-free paper.

Robert B. Reese. *Microprocessors: From Assembly Language to C Using the PIC18Fxx2*.

ISBN: 1-58450-378-5

eISBN: 1-58450-645-8

All brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks should not be regarded as intent to infringe on the property of others. The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products.

Library of Congress Cataloging-in-Publication Data

Reese, Robert Bryan, 1958-

Microprocessors : from assembly language to C using the PIC18Fxx2 / Robert B. Reese.

p. cm.

Includes bibliographical references and index.

ISBN 1-58450-378-5 (hardcover with cd-rom : alk. paper)

1. Microprocessors. 2. Microprocessors. I. Title.

TK7895.M5R44 2005

2005008835

Printed in the United States of America

05 7 6 5 4 3 2 First Edition

CHARLES RIVER MEDIA titles are available for site license or bulk purchase by institutions, user groups, corporations, etc. For additional information, please contact the Special Sales Department at 781-740-0400.

Requests for replacement of a defective CD-ROM must be accompanied by the original disc, your mailing address, telephone number, date of purchase, and purchase price. Please state the nature of the problem, and send the information to CHARLES RIVER MEDIA, INC., 10 Downer Avenue, Hingham, Massachusetts 02043. CRM's sole obligation to the purchaser is to replace the disc, based on defective materials or faulty workmanship, but not on the operation or functionality of the product.

*To Donna, who puts up with me*

*This page intentionally left blank*



# Contents

Acknowledgments	xv	
Preface	xvii	
<b>1</b>	<b>Number System and Digital Logic Review</b>	<b>1</b>
1.1	Learning Objectives	1
1.2	Binary Data	2
1.3	Unsigned Number Conversion	5
1.4	Binary and Hex Arithmetic	6
1.5	Combinational Logic Functions	10
1.6	Combinational Building Blocks	17
1.7	Sequential Logic	21
1.8	Sequential Building Blocks	25
1.9	Encoding Character Data	27
	Summary	29
	Review Exercises	29
<b>2</b>	<b>The Stored Program Machine</b>	<b>31</b>
2.1	Learning Objectives	31
2.2	Problem Solving the Digital Way	32
2.3	Finite State Machine Design	34
2.4	A Stored Program Machine	39
2.5	Modern Computers	47
	Summary	47
	Review Problems	47
<b>3</b>	<b>Introduction to the PIC18Fxx2</b>	<b>51</b>
3.1	Learning Objectives	51
3.2	Introduction to Microprocessors and Microcontrollers	52
3.3	The PIC18Fxx2 Microcontroller	53
3.4	Data Memory Organization and Data Transfer	55



3.5 Basic Arithmetic and Control Instructions	61
3.6 A PIC18 Assembly Language Program	64
3.7 The Clock and Instruction Execution	73
Summary	73
Review Problems	74
<b>4 Unsigned 8-Bit Arithmetic, Logical, Conditional Operations</b>	<b>77</b>
4.1 Learning Objectives	77
4.2 Bitwise Logical Operations, Bit Operations	78
4.3 The STATUS Register	83
4.4 Unsigned Conditional Tests	85
4.5 Looping	94
4.6 Shifts and Rotates	97
Summary	100
Review Problems	100
<b>5 Extended Precision and Signed Operations</b>	<b>103</b>
5.1 Learning Objectives	103
5.2 Extended Precision Integers	104
5.3 Extended Precision Operations	105
5.4 Signed Number Representation	114
5.5 Two's Complement Overflow	119
5.6 Operations on Signed Data	120
5.7 Branch Instruction Encoding	127
Summary	129
Review Problems	129
<b>6 Subroutines and Pointers</b>	<b>133</b>
6.1 Learning Objectives	133
6.2 Subroutines	134
6.3 The Stack and Call/Return	136
6.4 Implementing Subroutines in Assembly Language	141
6.5 Arrays and Pointers in C	146
6.6 Arrays and Pointers in Assembly Language	152
6.7 Accessing Table Data from Program Memory	160

6.8 Subroutines and Stack Frames: Dynamic Allocation	162
Summary	169
Review Problems	170
<b>7 Advanced Assembly Language: Higher Math</b>	<b>175</b>
7.1 Learning Objectives	175
7.2 Multiplication	176
7.3 Division	183
7.4 Fixed-Point and Saturating Arithmetic	188
7.5 Floating-Point Number Representation	192
7.6 BCD Arithmetic	197
7.7 ASCII Data Conversion	199
Summary	204
Review Problems	204
<b>8 The PIC18Fxx2: System Startup and Parallel Port IO</b>	<b>207</b>
8.1 Learning Objectives	207
8.2 High-Level Languages versus Assembly Language	208
8.3 C Compilation for the PIC18F242	210
8.4 PIC18F242 Startup Schematic	216
8.5 <i>ledflash.c</i> —The First C Program for PIC18F242 Startup	220
8.6 Datasheet Reading—A Critical Skill	223
8.7 PIC18Fxx2 Reset Sources	225
8.8 Experimenting with RESET, SLEEP, and the Watchdog Timer	228
8.9 Parallel Port Operation	231
8.10 LED/Switch IO and State Machine Programming	237
8.11 Interfacing to an LCD Module	242
Summary	249
Review Problems	249
<b>9 Asynchronous Serial IO</b>	<b>253</b>
9.1 Learning Objectives	253
9.2 IO Channel Basics	254
9.3 Synchronous Serial IO	257
9.4 Asynchronous Serial IO	259

9.5	The PIC18Fxx2 USART	263
9.6	The RS232 Standard	270
9.7	Serial IO Examples	273
	Summary	277
	Review Problems	278
<b>10</b>	<b>Interrupts and a First Look at Timers</b>	<b>281</b>
10.1	Learning Objectives	281
10.2	Interrupt Basics	282
10.3	PIC18 Interrupt Details	284
10.4	Interrupt-Driven Asynchronous Serial Data Input	287
10.5	Using a Software FIFO with Interrupt-Driven IO	291
10.6	Other Interrupt Sources, Sleep Mode	296
10.7	State Machine Programming for Interrupt-Driven IO	299
10.8	The Timer Subsystem: Timer2	304
10.9	Switch Debouncing Using a Timer	307
10.10	A Rotary Encoder Interface	309
10.11	A Numeric Keypad Interface	315
10.12	On Writing and Debugging ISRs	319
	Summary	321
	Review Problems	322
<b>11</b>	<b>Synchronous Serial IO</b>	<b>327</b>
11.1	Learning Objectives	327
11.2	The PIC18 and Synchronous Serial IO	328
11.3	USART Synchronous Mode	329
11.4	The Serial Peripheral Interface (SPI)	331
11.5	SPI Examples: A Digital Potentiometer and a Serial EEPROM	334
11.6	The I <sup>2</sup> C Bus	345
11.7	The I <sup>2</sup> C on the PIC18Fxx2	348
11.8	The 24LC515 Serial EEPROM	356
11.9	Double Buffering for Interrupt-Driven Writes	364
	Summary	366
	Review Problems	367

<b>12</b>	<b>Data Conversion</b>	<b>371</b>
12.1	Learning Objectives	371
12.2	Data Conversion Basics	372
12.3	Analog-to-Digital Conversion	373
12.4	PIC18Fxx2 Analog-to-Digital Converter	382
12.5	Digital-to-Analog Conversion	391
12.6	Digital-to-Analog Converter Example: The MAXIM 518	400
	Summary	406
	Review Problems	407
<b>13</b>	<b>Timers</b>	<b>411</b>
13.1	Learning Objectives	411
13.2	The Timer0 Subsystem	412
13.3	The Timer1 and Timer3 Subsystems	419
13.4	Pulse Width Measurement Using Capture Mode	422
13.5	Timer1/Timer3 Compare Mode	428
13.6	Using Capture Mode for Infrared Decoding	433
13.7	Timer2 and Pulse Width Modulation	442
13.8	Using Capture Mode for Frequency Measurement	447
	Summary	451
	Review Problems	452
<b>14</b>	<b>Capstone: Audio Sampling, Monitoring System, and Autonomous Robot</b>	<b>455</b>
14.1	Learning Objectives	456
14.2	Design of an Audio Record/Playback System	456
14.3	Implementation of an Audio Record/Playback System	459
14.4	Design of a Home Monitoring System	466
14.5	The DS1621 Digital Thermometer	469
14.6	Using the Nonvolatile Storage on the PIC18Fxx2	475
14.7	Implementation of a Home Monitoring System	483
14.8	Design and Implementation of an Autonomous Robot	494
	Summary	504
	Suggested Project Modifications	505

<b>15</b>	<b>Beyond the PIC18Fxx2</b>	<b>507</b>
15.1	Learning Objectives	507
15.2	External Memory Interfacing	508
15.3	Other PIC Family Members	513
15.4	Bus Arbitration in I <sup>2</sup> C	516
15.5	The Controller Area Network (CAN)	528
15.6	The Universal Serial Bus (USB)	523
15.7	A Brief Survey of Non-PIC Microcontrollers	527
15.8	Real-Time Operating Systems	531
	Summary	533
	Suggested Survey Topics	534
<b>Appendix A PIC18Fxx2 Architecture, Instruction Set, Register Summary</b>		<b>537</b>
<b>Appendix B Microchip MPLAB Quickstart</b>		<b>549</b>
<b>Appendix C HI-TECH PICC-18 C Compiler Demo for the PIC18F242</b>		<b>553</b>
<b>Appendix D Notes on the C Language</b>		<b>557</b>
D.1	Formatted IO ( <i>PRINTF</i> , <i>SCANF</i> , <i>SPRINTF</i> , <i>SSCANF</i> )	557
D.2	For C++ Programmers	559
D.3	For New Programmers	560
D.4	For Experienced C Programmers	560
<b>Appendix E Suggested Laboratory Exercises</b>		<b>563</b>
E.1	Lab Setup	563
E.2	Experiment 1: A Stored Program Machine (Chapters 1, 2)	566
E.3	Experiment 2: PIC18xx2 Introduction (Chapter 3)	567
E.4	Experiment 3: Unsigned 8-Bit Operations (Chapter 4)	569
E.5	Experiment 4: Extended Precision and Signed Operations (Chapter 5)	572
E.6	Experiment 5: Pointers and Subroutines (Chapter 6)	573
E.7	Experiment 6: Hardware Startup (Chapter 8)	575
E.8	Experiment 7: LED/Switch IO and Introduction to Asynchronous Serial IO (Chapters 8, 9)	578
E.9	Experiment 8: Interrupts (Chapter 10)	580

E.10 Experiment 9: More Interrupts, the I <sup>2</sup> C Bus, and a Serial EEPROM (Chapter 11)	583
E.11 Experiment 10: Introduction to Data Conversion (Chapter 12)	585
E.12 Experiment 11: Timer Introduction and Waveform Generation (Chapters 10, 13)	587
E.13 Experiment 12: Time Measurement and IR Waveform Decoding (Chapter 13)	593
E.14 Experiment 13: Audio Record/Playback (Chapter 14)	594
E.15 Hardware Debugging Checklist	595
E.16 Instrumentation and Prototyping Hints	598
<b>Appendix F The Jolt/Colt Serial Bootloaders</b>	<b>601</b>
F.1 Programming the Jolt/Colt Firmware	601
F.2 Jolt Installation	605
F.3 Running Jolt	608
F.4 Colt Bootloader Installation and Execution	610
<b>Appendix G Circuits 001</b>	<b>613</b>
G.1 Voltage, Current, Resistance	613
G.2 Capacitors	618
<b>Appendix H References</b>	<b>621</b>
<b>Appendix I Answers to Review Problems</b>	<b>625</b>
I.1 Chapter 1	625
I.2 Chapter 2	627
I.3 Chapter 3	629
I.4 Chapter 4	630
I.5 Chapter 5	631
I.6 Chapter 6	633
I.7 Chapter 7	638
I.8 Chapter 8	639
I.9 Chapter 9	643
I.10 Chapter 10	644
I.11 Chapter 11	646
I.12 Chapter 12	647

I.13 Chapter 13	649
<b>Appendix J About the CD-ROM</b>	<b>655</b>
J.1 General System Requirements	655
<b>Index</b>	<b>657</b>



# Acknowledgments

I would like to thank the following individuals for their assistance in preparing this book:

- J.W. Bruce, for authoring Chapter 12 and for providing valuable feedback and suggestions on the rest of the book.
- Jane Moorhead, for being a partner in teaching this material to Mississippi State University (MSU) electrical engineering, computer engineering, computer science, and software engineering students.
- Martin Dubuc, for providing the publicly available Jolt/Colt serial bootloaders for the PIC18. We make extensive use of these bootloaders in our microprocessors lab at MSU, and these bootloaders are included on this book's CD-ROM.
- HI-TECH Software for providing the 120-day demo version of the PICC-18™ C compiler that is included on this book's CD-ROM. I also want to thank Matt Luckman at HI-TECH Software for his efforts in putting that distribution together.
- Donna Reese, for her careful proofreading of the book material.





*This page intentionally left blank*



# Preface

**T**his book is intended as an introduction to microprocessors and microcontrollers for either the student or hobbyist. The book structure is:

**Chapter 1:** Review of digital logic concepts.

**Chapter 2:** Computer architecture fundamentals.

**Chapters 3 through 6:** Coverage of assembly language programming in a C language context using the PIC18Fxx2 family.

**Chapter 7:** Advanced assembly language programming structured around computer arithmetic topics.

**Chapters 8 through 13:** Fundamental microcontroller interfacing topics such as parallel IO, asynchronous serial IO, synchronous serial IO (I<sup>2</sup>C and SPI), interrupt-driven IO, timers, analog-to-digital conversion, and digital-to-analog conversion.

**Chapter 14:** Presents three capstone projects involving topics from Chapters 8 through 13.

**Chapter 15:** Topics beyond the PIC18Fxx2 family, such as a survey of other microprocessor families, the CAN bus, and memory technologies.

## **USING THIS BOOK IN AN ACADEMIC ENVIRONMENT**

---

At Mississippi State University, majors in Electrical Engineering (EE), Computer Engineering (CPE), Computer Science (CS), and Software Engineering (SE) take our first course in microprocessors. Previous to spring 2002, this course emphasized

X86 assembly language programming with the lab experience being 100-percent assembly language based and containing no hardware component. We found that students entering our senior design course who had the expectation of something “real” being built were unprepared for doing prototyping activities or for incorporating a microcontroller component into their designs. We did offer a course in microcontrollers, but it was an elective senior-level course and many students had not taken that course previous to senior design. In spring 2002, the Computer Engineering Steering committee reexamined our goals for the first course in microprocessors and the approach for this book was developed. This book is intended for use as a *first* course in microprocessors using the PIC18Fxx2 microcontroller with prerequisites of basic digital design and exposure to either C or C++ programming. The book begins with simple microprocessor architecture concepts, moves to assembly language programming in a C language context, and then covers fundamental hardware interfacing topics such as parallel IO, asynchronous serial IO, synchronous serial IO (I<sup>2</sup>C and SPI), interrupt-driven IO, timers, analog-to-digital conversion, and digital-to-analog conversion. Programming topics are discussed using both assembly language and C, while hardware interfacing examples use C to keep code complexity low and improve clarity. The book’s CD-ROM includes a 120-day demo version of the PICC-18 C compiler for the PIC18F242 from HI-TECH software. In addition to better preparing students for senior design, another goal of this book is to enable students to take courses in advanced embedded systems or computer architecture. As such, a broad coverage of software and hardware topics is included. The assembly language programming chapters emphasize the linkage between C language constructs and their assembly language equivalent so that students clearly understand the impact of C coding choices in terms of execution time and memory requirements. It is *not* a goal of this textbook to create students who are experts only in assembly language programming, with no understanding of high-level language programming techniques and limited hardware exposure. Most embedded software is written in C for portability and complexity reasons, which argues favorably for reduced emphasis on assembly language and increased emphasis on C. Embedded system hardware complexity is steadily increasing, which means a first course in microprocessors that reduces assembly language coverage (but does not eliminate it) in favor of hands-on experience with fundamental interfacing allows students to begin at a higher level in an advanced course in embedded systems. Hardware interface topics included in this book cover the fundamentals (parallel IO, serial IO, interrupts, timers, A/D, D/A) using devices that do not require extensive circuits knowledge because of the lack of a circuits course prerequisite. The microcontroller interfacing topics presented in this text-

book are sufficient for providing a skill set that is extremely useful to a student in a senior design capstone course or in an advanced embedded system course.

Thus, the principle motivation for this book is that microcontroller knowledge has become essential for successful completion of senior capstone design courses. These capstone courses are receiving increased emphasis under ABET 2000 guidelines. This places increased pressure on Computer Engineering and Electrical Engineering programs to include significant exposure to embedded systems topics as early in the curriculum as possible. A second motivation for this book is that the recently released ACM/IEEE Computer Engineering model curriculum recommends 17 hours of embedded system topics as part of the Computer Engineering curriculum core, which is easily satisfied by a course containing the topics in this book. A third motivating factor is the increased pressure on colleges and universities to reduce hours in engineering curriculums; this book shows how a single course can replace separate courses in assembly language programming and basic microprocessor interfacing.

The course sequence used at Mississippi State University that this book fits into is:

- Basic digital design (Boolean algebra, combinational and sequential logic), which is required by EE, CPE, CS, and SE majors.
- Introduction to microprocessors (this book), which is required for EE, CPE, CS, and SE majors.
- Computer architecture as represented by the topic coverage of the Hennessy and Patterson textbook “Computer Organization & Design: The Hardware/Software Interface.” This includes reinforcement of the assembly language programming taught in the microprocessor course via a general-purpose instruction set architecture (e.g., the MIPS) along with coverage of traditional high-performance computer architecture topics (pipelined CPU design, cache strategies, parallel bus I/O). Required for CPE, CS, and SE majors.
- Advanced embedded systems covering topics such as (a) real-time operating systems, (b) internet appliances, (c) advanced interfaces such as USB, CAN, Ethernet, FireWire, and (d) programming in alternate embedded languages such as Java. Required for CPE majors.

Chapter 1 provides a broad review of digital logic fundamentals. Chapters 2 through 6 and 8 through 13 cover the core topics of assembly language programming and microcontroller interfacing. Chapter 14 contains three capstone projects that integrate the material of the previous chapters. Chapters 7 and 15 have

optional topics on advanced assembly language programming and microprocessor interfacing, which can be used to supplement the core material. Appendix E, “Suggested Laboratory Exercises,” contains a sequence of 13 laboratory experiments that comprise an off-the-shelf lab experience: one experiment on fundamental computer architecture topics, four experiments on PIC18 assembly language, and eight hardware experiments. The hardware labs cover all major subsystems on the PIC18: A/D, timers, asynchronous serial interface, and the I<sup>2</sup>C interface. The hardware experiments are based on a protoboard/parts kit approach where the students incrementally build a PIC18F242 system that includes a serial EEPROM, an external 8-bit D/A, and a RS-232 port. A protoboard/part kits approach is used instead of a pre-assembled printed circuit board (PCB) for several important reasons:

- When handed a pre-assembled PCB, a student tends to view it as a monolithic element. A protoboard/parts kit approach forces a student to view each part individually and read datasheets to understand how parts connect to each other.
- Hardware debugging and prototyping skills are developed during the painful process of bringing the system to life. These hard-won lessons prove useful later when the student must do the same thing in a senior design context. This also provides students with the confidence that having done it one time they can do it again, this time outside of a fixed laboratory environment with guided instruction.
- A protoboard/parts kit approach gives the ultimate flexibility to modify experiments from semester to semester by simply changing a part or two; also when the inevitable part failures occur they are easily replaced.

In using this approach at Mississippi State University, I have seen a “Culture of Competence” develop in regard to microcontrollers and prototyping in general. Almost all senior design projects now routinely include a microcontroller component (not necessarily PIC-based). Students concentrate their efforts on design definition, development, and refinement instead of spending most of their time climbing the learning curve on prototyping and microcontroller usage.

This book’s *C* examples on hardware interfacing strive for educational value first and optimization second. Subsystem configuration code uses named bit fields and individual bit field assignments in *C* examples instead of whole-register assignments to emphasize bit field roles within those registers. Register values for controlling baud rate, I<sup>2</sup>C bus speed, and periodic interrupt rates are hard-coded for the clock frequency of the reference PIC18F242 system instead of hiding the calculations within *C* macros or functions that compute a register value for a desired

rate. This is done intentionally so students can be assigned different values within the lab and homework exercises, forcing them to use the PIC18F242 datasheet formulas for computing new register values.

## **FOR THE HOBBYIST**

---

This book assumes very little background, and thus is appropriate for readers with widely varying experience levels. It is suggested that you begin by examining the experiments in Appendix E and find the ones that interest you. Then, read the chapter that is referenced by the experiment. The suggested revisions for the capstone chapter projects (Chapter 14) are a good test of your knowledge once you have exhausted the experiments.

### **A Final Word**

Writing this book has been a rewarding experience, and many people have helped (see the Acknowledgments section). It has been akin to building a stone wall; each day a little more is added, each section covering more distance, with the satisfaction of seeing it grow as time and effort is expended. However, this book's lifespan will be a fraction of that of a sturdily built stone wall. But that is the fun of engineering—it is constantly changing, so you are constantly learning. I hope that you have fun while learning about microprocessors and the PIC18Fxx2!

Bob Reese  
Starkville, Mississippi

*This page intentionally left blank*

# 1

# Number System and Digital Logic Review

## In This Chapter

- Binary Data
- Unsigned Number Conversion
- Binary and Hex Arithmetic
- Combinational Logic Functions
- Combinational Building Blocks
- Sequential Logic
- Sequential Building Blocks
- Encoding Character Data

This chapter reviews number systems, Boolean algebra, logic gates, combinational logic gates, combinational building blocks, sequential storage elements, and sequential building blocks.

## 1.1 LEARNING OBJECTIVES

---

After reading this chapter, you will be able to:

- Create a binary encoding for object classification.
- Convert unsigned decimal numbers to binary and hex representations, and vice versa.



- Identify NOT, OR, AND, NOR, NAND, and XOR logic functions and their symbols.
- Evaluate simple Boolean functions.
- Describe the operation of CMOS P and N transistors.
- Identify the CMOS transistor level implementations of simple logic gates.
- Compute clock period, frequency, and duty cycle given appropriate parameters.
- Identify common combinational building blocks.
- Identify common sequential building blocks.
- Translate a character string into ASCII encoded data, and vice versa.

Binary number system representation and arithmetic is fundamental to all computer system operation. Basic logic gates, CMOS transistor operation, and combinational/sequential building block knowledge will help your comprehension of the diagrams found in datasheets that describe microprocessor subsystem functionality. A solid grounding in these subjects ensures better understanding of the microprocessor topics that follow in later chapters.

## 1.2 BINARY DATA

---

Binary logic or digital logic is the basis for all computer systems built today. *Binary* means two, and many concepts can be represented by two values: true/false, hot/cold, on/off, 1/0, to name a few. A single binary datum whose values are “1” and “0” is referred to as a *bit*. Groups of bits are used to represent concepts that have more than two values. For example, to represent the concepts hot/warm/cool/cold, two or more bits can be used as shown in Table 1.1

**TABLE 1.1** Digital Encoding Examples

Value	Encoding A	Encoding B	Encoding C
Cold	0 0	0 0	0 0 0 1
Cool	0 1	1 0	0 0 1 0
Warm	1 0	1 1	0 1 0 0
Hot	1 1	0 1	1 0 0 0

To encode  $n$  objects, the minimum number of bits required is  $k = \lceil \log_2 n \rceil$ , where  $\lceil \cdot \rceil$  is the ceiling function that takes the nearest integer  $\geq \log_2 n$ . For the four

values in Table 1.1, the minimum number of bits required is  $\lceil \log_2(4) \rceil = 2$ . Both encoding A and encoding B use the minimum number of bits, but differ in how codes are assigned to the values. Encoding B uses a special encoding scheme known as *Gray code*, in which adjacent table entries only differ by at most one bit position. Encoding C uses more than the minimum number of bits; this encoding scheme is known as *one-hot encoding*, as each code only has a single bit that is a “1” value.

Encoding A uses *binary counting order*, which means that the code progresses in numerical counting order if the code is interpreted as a *binary number* (base 2). In an unsigned binary number, each bit is weighted by a power of two. The rightmost bit, or *least significant bit* (LSb), has a weight of  $2^0$ , with each successive bit weight increasing by a power of two as one moves from right to left. The leftmost bit, the *most significant bit* (MSb), has a weight of  $2^{n-1}$ , for  $n$  bits in the binary number. A lowercase “b” is purposefully used in the LSb and MSb acronyms; the use of an uppercase “B” in LSB and MSB acronyms is discussed later.

The formal term for a number’s base is *radix*. If  $r$  is the radix, then a binary number has  $r = 2$ , a decimal number has  $r = 10$ , and a hexadecimal number has  $r = 16$ . In general, each digit of a number of radix  $r$  can take on the values 0 through  $r-1$ . The *least significant digit* (LSD) has a weight of  $r^0$ , with each successive digit increasing by a power of  $r$  as one moves from right to left. The leftmost digit, the *most significant digit* (MSD), has weight of  $r^{n-1}$ , where  $n$  is the number of digits in the number. For hexadecimal (hex) numbers, letters A through F represent the digits 10 through 15, respectively. Decimal, binary, and hexadecimal numbers are used exclusively in this book. If the base of the number cannot be determined by context, a “0x” is used as the radix identifier for hex numbers (i.e., 0x3A), and “0b” for binary numbers (i.e., 0b01101000). No radix identifier is used for decimal numbers. Table 1.2 lists the binary and hex values for the decimal values 0 through 15. Note that 4 bits are required to encode these 16 values since  $2^4 = 16$ . The binary and hex values in Table 1.2 are given without radix identifiers.

**TABLE 1.2** Binary Encoding for Decimal Numbers 0-15

Decimal	Binary	Binary to Decimal	Hex	Hex to Decimal
0	0000	$0*2^3 + 0*2^2 + 0*2^1 + 0*2^0$	0	$0*16^0$
1	0001	$0*2^3 + 0*2^2 + 0*2^1 + 1*2^0$	1	$1*16^0$
2	0010	$0*2^3 + 0*2^2 + 1*2^1 + 0*2^0$	2	$2*16^0$
3	0011	$0*2^3 + 0*2^2 + 1*2^1 + 1*2^0$	3	$3*16^0$
4	0100	$0*2^3 + 1*2^2 + 0*2^1 + 0*2^0$	4	$4*16^0$
5	0101	$0*2^3 + 1*2^2 + 0*2^1 + 1*2^0$	5	$5*16^0$
6	0110	$0*2^3 + 1*2^2 + 1*2^1 + 0*2^0$	6	$6*16^0$

→

7	0111	$0*2^3 + 1*2^2 + 1*2^1 + 1*2^0$	7	$7*16^0$
8	1000	$1*2^3 + 0*2^2 + 0*2^1 + 0*2^0$	8	$8*16^0$
9	1001	$1*2^3 + 0*2^2 + 0*2^1 + 1*2^0$	9	$9*16^0$
10	1010	$1*2^3 + 0*2^2 + 1*2^1 + 0*2^0$	A	$10*16^0$
11	1011	$1*2^3 + 0*2^2 + 1*2^1 + 1*2^0$	B	$11*16^0$
12	1100	$1*2^3 + 1*2^2 + 0*2^1 + 0*2^0$	C	$12*16^0$
13	1101	$1*2^3 + 1*2^2 + 0*2^1 + 1*2^0$	D	$13*16^0$
14	1110	$1*2^3 + 1*2^2 + 1*2^1 + 0*2^0$	E	$14*16^0$
15	1111	$1*2^3 + 1*2^2 + 1*2^1 + 1*2^0$	F	$15*16^0$

A binary number of N bits can represent the unsigned decimal values of 0 to  $2^N-1$ . A common size for binary data is a group of 8 bits, referred to as a *byte*. A byte can represent the unsigned decimal range of 0 to 255 (0x00 to 0xFF in hex). Groups of bytes are often used to represent larger numbers; this topic is explored in Chapter 5, “Extended Precision and Signed Operations.” Common powers of two are given in Table 1.3. Powers of two that are evenly divisible by  $2^{10}$  can be referred to by the suffixes K (Kilo,  $2^{10}$ ), M (Mega,  $2^{20}$ ), and G (Giga,  $2^{30}$ ). Thus, the value of 4096 can be written in the abbreviated form of 4 K ( $4 \times 1 \text{ K} = 2^2 \times 2^{10} = 2^{12}$ ).

**TABLE 1.3** Common Powers of 2

Power	Decimal	Hex	Power	Decimal	Hex
$2^0$	1	0x1	(K)... $2^{10}$	1024	0x400
$2^1$	2	0x2	$2^{11}$	2048	0x800
$2^2$	4	0x4	$2^{12}$	4096	0x1000
$2^3$	8	0x8	$2^{13}$	8192	0x2000
$2^4$	16	0x10	$2^{14}$	16384	0x4000
$2^5$	32	0x20	$2^{15}$	32768	0x8000
$2^6$	64	0x40	$2^{16}$	65536	0x10000
$2^7$	128	0x80	(M)... $2^{20}$	1,048,576	0x100000
$2^8$	256	0x100	(G)... $2^{30}$	1,073,741,824	0x40000000
$2^9$	512	0x200	$2^{32}$	4,294,967,296	0x100000000

**Sample Question: What is the largest unsigned decimal number that can be represented using a binary number with 16 bits?**

*Answer:* From Table 1.3, we see that  $2^{16} = 65536$ , so  $2^{16}-1 = 65535$ .

### 1.3 UNSIGNED NUMBER CONVERSION

To convert a number of any radix to decimal, simply multiply each digit by its corresponding weight and sum the result. The example that follows shows binary-to-decimal, and hex-to-decimal conversion:

$$\begin{aligned}
 \text{(binary to decimal) } 0b0101\ 0010 &= 0*2^7 + 1*2^6 + 0*2^5 + 1*2^4 + 0*2^3 + \\
 &\quad 0*2^2 + 1*2^1 + 0*2^0 \\
 &= 0 + 64 + 0 + 16 + 0 + 0 + 2 + 0 = 82 \\
 \text{(hex to decimal) } 0x52 &= 5*16^1 + 2*16^0 = 80 + 2 = 82.
 \end{aligned}$$

To convert a decimal number to a different radix, perform successive division of the decimal number by the radix; at each step the remainder is a digit in the converted number, and the quotient is the starting value for the next step. The successive division ends when the quotient becomes less than the radix. The digits of the converted number are determined rightmost to leftmost; with the last quotient being the leftmost digit of the converted number. The following sample problem illustrates the successive division algorithm.

**Sample Question: Convert 435 to hex**

*Answer:*

Step 1:	$435/16 = 27$ , remainder = 3 (rightmost digit).
Step 2:	$27/16 = 1$ , remainder = 11 = 0xB (next digit).
Step 3:	$1 < 16$ , so leftmost digit = 1.
Final answer:	$435 = 0x1B3$ .

To check your work, perform the reverse conversion:

$$0x1B3 = 1 * 16^2 + 11 * 16^1 + 3 * 16^0 = 1 * 256 + 11 * 16 + 3 * 1 = 256 + 176 + 3 = 435.$$

#### Hex to Binary, Binary to Hex

Hex can be viewed as a shorthand notation for binary. A quick method for performing binary-to-hex conversion is to convert each group of four binary digits (starting with the rightmost digit) to one hex digit. If the last (leftmost) group of binary digits does not contain 4 bits, then pad with leading zeros to reach four digits.

Converting hex to binary is the reverse procedure of replacing each hex digit with four binary digits. The easiest way to perform decimal-to-binary conversion is to first convert to hex, and then convert the hex number to binary. This involves less division operations, and hence less chance for careless error. Similarly, binary-to-decimal conversion is best done by converting the binary number to a hex value, and then converting the hex number to decimal. The following examples illustrate binary-to-hex, hex-to-binary, and decimal-to-binary conversion.

**Sample Question: Convert 0b010110 to hex.**

*Answer:* Starting with the rightmost digit, form groups of four: 01 0110. The leftmost group has only two digits, so pad this group with zeros as: 0001 0110. Now convert each group of four digits to hex digits (see Table 1.3):  
0b 0001 0110 = 0x 16.

**Sample Question: Convert 0xF3C to binary.**

*Answer:* Replace each hex digit with its binary equivalent:  
0x F3C = 0b 1111 0011 1100

**Sample Question: Convert 243 to binary.**

*Answer:* First, convert 243 to hex:  
Step 1:  $243/16 = 15$ , remainder 3 (rightmost digit)  
Step 2:  $15 < 16$ , so leftmost digit is 0xF (15). Hex result is 0xF3  
 $243 = 0xF3 = 0b 1111 0011$  (final answer, in binary)  
Check:  $0xF3 = 15 * 16 + 3 = 240 + 3 = 243$ .

## 1.4 BINARY AND HEX ARITHMETIC

---

Addition, subtraction, and shift operations are implemented in some form in most digital systems. The fundamentals of these operations are reviewed in this section, and revisited in Chapters 3 and 4 when discussing basic computer operations.

### Binary and Hex Addition

Addition of two numbers,  $i + j$ , in any base is accomplished by starting with the rightmost digit and adding each digit of  $i$  to each digit of  $j$ , moving right to left. If the digit sum is less than the radix, the result digit is the sum and a carry of 0 is used in the next digit addition. If the sum of the digits is greater than or equal to the radix, a carry of 1 is added to the next digit sum, and the result digit is computed by subtracting  $r$  from the digit sum. For binary addition, these rules can be stated as:

- $0+0 = 0$ , carry = 0
- $0+1 = 1$ , carry = 0
- $1+0 = 1$ , carry = 0
- $1+1 = 0$ , carry = 1

Figure 1.1 shows a digit-by-digit addition for the numbers  $0b110 + 0b011$ . Note that the result is  $0b001$  with a carry out of the most significant digit of “1”. A carry out of the most significant digit indicates that the sum produced *unsigned overflow*; the result could not fit in the number of available digits. A carry out of the most significant digit is an unsigned error indicator if the numbers represent unsigned integers. In this case, the sum  $0b110 + 0b011$  is  $6 + 3$  with the correct answer being 9. However, the largest unsigned integer that can be specified in 3 bits is  $2^3 - 1$ , or 7. The value of 9 is too large to be represented in 3 bits, and thus the result is incorrect from an arithmetic perspective, but is correct by the rules of binary addition. This is known as the limited precision problem; only increasing the number of bits used for binary encoding can increase the number range. We study this problem and the consequences of using more or less bits for number representation in more detail in Chapter 5.

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

**FIGURE 1.1** Binary addition example.

**Sample Question: Compute  $0x1A3 + 0x36F$ .**

*Answer:* A digit-by-digit addition for the operation  $0x1A3 + 0x36F$  is as follows. The rightmost result digit is formed by adding  $0x3$  (3) +  $0xF$  (15) = 18. Note the digit sum is  $\geq$  than 16, so a carry of 1 is produced and the rightmost result digit is computed by subtracting the radix, or  $18 - 16 = 2 = 0x2$ . The middle digit sum is then  $0xA$  (10) +  $0x6$  (6) + 1 (carry) = 17. This digit sum is  $\geq$  than 16, so this produces a carry of 1 with the middle digit computed as  $17 - 16 = 1 = 0x1$ . The leftmost digit sum is  $0x1 + 0x3 + 0x1$  (carry) =  $0x5$ . The result is then  $0x1A3 + 0x36F = 0x512$ . Converting each number to decimal before summing, or  $419 + 879 = 1298$ , checks this result. Verifying that  $0x36F - 0x512 = 0x1A3$  also checks this result, but this requires reading the next section on subtraction!

### Binary and Hex Subtraction

Subtraction of two numbers,  $i - j$ , in any base is accomplished by starting with the rightmost digit, and subtracting each digit of  $j$  from each digit of  $i$ , moving right to left. If the  $i$  digit is greater or equal to the  $j$  digit, then the result digit is the subtraction  $i - j$ , with a borrow of 0 used in the next digit subtraction. If the  $i$  digit is less than the  $j$  digit, then a borrow of 1 is used in the next digit subtraction, and the result digit is formed by  $i + r - j$  (the current  $i$  digit is increased by a weight of  $r$ ). For binary subtraction, these rules can be stated as:

- $0 - 0 = 0$ , borrow = 0
- $0 - 1 = 1$ , borrow = 1
- $1 - 0 = 1$ , borrow = 0
- $1 - 1 = 0$ , borrow = 0

Figure 1.2 shows a digit-by-digit subtraction for the value  $0b010 - 0b101$ . This operation produces a result of  $0b101$ , and a borrow out of the most significant digit of 1. If interpreted as unsigned numbers, the operation is  $2 - 5 = 5$ , which is incorrect. A borrow out of the most significant digit of 1 indicates an *unsigned underflow*; the correct result is a number less than zero. But in unsigned numbers, there is no number less than zero, so the result is incorrect in an arithmetic sense (the operation is perfectly valid, however). A binary representation for *signed integers* is needed to interpret the binary result correctly; this topic is saved for Chapter 5.

$$\begin{array}{r}
 \begin{array}{ccc}
 -1 \nearrow +2 & -1 \nearrow +2 & \\
 0 & 1 & 0 \\
 - & 1 & 0 & 1 \\
 \hline
 1 & 0 & 1
 \end{array}
 & \nearrow \text{Borrow}
 \end{array}$$

**FIGURE 1.2** Binary subtraction example.

The subtraction  $A - B$  can also be performed by the operation  $A + \sim B + 1$ , where the operation  $\sim B$  is called the *one's complement* of  $B$  and is formed by taking the complement of each bit of  $B$ . As an example, consider the previous operation of  $0b010 - 0b101$ . The one's complement of  $0b101$  is  $0b010$ . The subtraction can be rewritten as:

$$A + \sim B + 1 = 0b010 + (0b010 + 0b001) = 0b010 + 0b011 = 0b101$$

This is the same result obtained when binary subtraction rules were used. The value  $\sim B + 1$  is called the *two's complement* of B, and this is discussed in more detail in Chapter 5, “Extended Precision and Signed Operations,” when signed integer representation is covered.

**Sample Question: Compute  $0xA02 - 0x5C4$ .**

*Answer:* A digit-by-digit hex subtraction for the operation  $0xA02 - 0x5C4$  is as follows. The rightmost subtraction of  $0x2 - 0x4$  requires a borrow from the next digit, so the rightmost digit calculation becomes  $2 + 16 - 4 = 14 = 0xE$ . The middle digit calculation becomes  $0x0 - 0xC - 0x1$  (borrow). This requires a borrow from the next (leftmost) digit, so this calculation becomes:

$$16 + 0 - 12 - 1 = 3 = 0x3$$

The leftmost digit calculation is:

$$0xA - 0x5 - 0x1 \text{ (borrow)} = 10 - 5 - 1 = 4 = 0x4$$

Thus, the final result is  $0xA02 - 0x5C4 = 0x43E$ . As always, this result can be checked by verifying that  $0x5C4 + 0x43E = 0xA02$  (and yes, it is correct!).

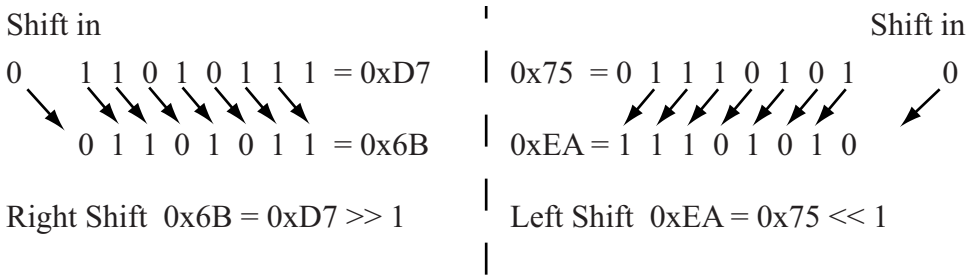
## Shift Operations

A right shift of a binary value moves all of the bits to the right by one position, and shifts a new bit value into the MSb. If the new shift value is a “0”, this is equivalent to dividing the binary value by two. For example, using a “0” value for the bit shifted into the MSb, the binary value  $0b1100$  (12) shifted to the right by one position is  $0b0110$  (6). If this value is shifted to the right once more, the new value is  $0b0011$  (3). In this book, operators from the C language are used for expressing numerical operations. The C language operator for a right shift is  $\gg$ , where  $A \gg 1$  reads “A shifted to the right by one bit.”

A left shift of an unsigned binary value moves all of the bits to the left by one position, and shifts a new bit value into the LSb. If the new bit shifted in is a “0”, this is equivalent to multiplying the binary value by two. For example, using a “0” value for the bit shifted into the LSb, the binary value  $0b0101$  (5) shifted to the left by one position is  $0b1010$  (10). If this value is shifted to the left once more, the new value is  $0b0100$  (4). The value 4 is not  $10 \times 2$ ; the correct result should be 20. However, the value 20 cannot fit in 4 bits; the largest unsigned value represented in 4 bits is  $2^4 - 1 = 15$ . In this case, the bit shifted out of the MSb is a “1”; when this happens, unsigned overflow occurs for the shift operation and the new value is incorrect in



an arithmetic sense. The C language operator for a left shift is  $\ll$ , where  $A \ll 1$  reads “A shifted to the left by one bit”. Figure 1.3 gives additional examples of left and right shift operations.



**FIGURE 1.3** Shift operation examples.

If an  $n$ -bit value is shifted to the left or right  $n$  times, with “0” used as the shift-in value, the result is zero, as all bits become “0”. When shifting a hex value, it is best to convert the hex number to binary, perform the shift, and then convert the binary number back to hex.

**Sample Question:** What is the new value of  $0xA7 \gg 2$  assuming the MSb is filled with a “0”?

Answer: The value  $0xA7 = 0b1010\ 0111$ , so  $0xA7 \gg 1 = 0b01010011$ . Shifting this value to the right by one more gives  $0b01010011 \gg 1 = 0b00101001 = 0x29$ . Therefore,  $0xA7 \gg 2 = 0x29$ .

## 1.5 COMBINATIONAL LOGIC FUNCTIONS

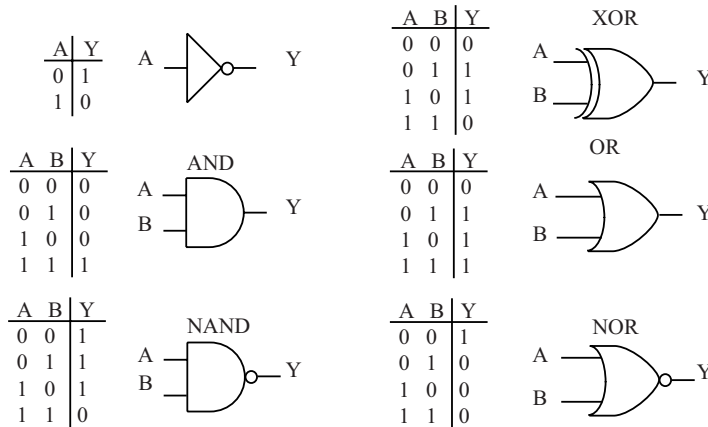
*Boolean algebra* defines properties and laws between variables that are binary-valued. The basic operations of Boolean algebra are NOT, OR, and AND whose definitions are:

**NOT(A):** Is “1” if  $A = 0$ ; NOT(A) is “0” if  $A = 1$  (the output is said to be the *complement* or *inverse* of the input).

**AND(A1, A2,...An):** Is “1” only if all inputs A1 through An have value = 1.

**OR (A1, A2, ...An):** Is “1” if any input A1 through An has value “1”.

The C language operators for bitwise complement (“~”), AND (“&”), OR (“|”) are used in this book for logic operations. Thus, NOT(A) = ~A, AND(A,B) = A & B, and OR(A,B) = A | B where the Boolean variables have values of either “0” or “1”. Logic operations are also defined by *truth tables* and shape distinctive symbols. A truth table has all input combinations listed in binary counting order on the left side, with the output value given on the right side of the table. Figure 1.4 lists the two-input truth tables and shape distinctive symbols for the NOT, AND, OR, NAND, NOR, and XOR (exclusive-OR) logic functions.



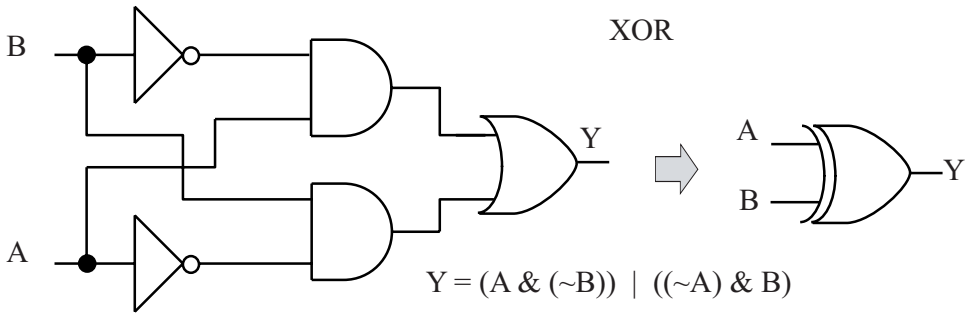
**FIGURE 1.4** Truth table, logic symbols for basic two-input logic gates.

A NAND function is an AND function whose output is complemented; similarly, a NOR function is an OR function whose output is complemented. An XOR function is defined by the truth table shown in Figure 1.4, or can be expressed using NOT, AND, OR operators as shown in Equation 1.1. The C language operator for XOR is ^, thus XOR(A,B) = A ^ B. Logically stated, XOR(A,B) is “1” if A is not equal to B, and “0” otherwise.

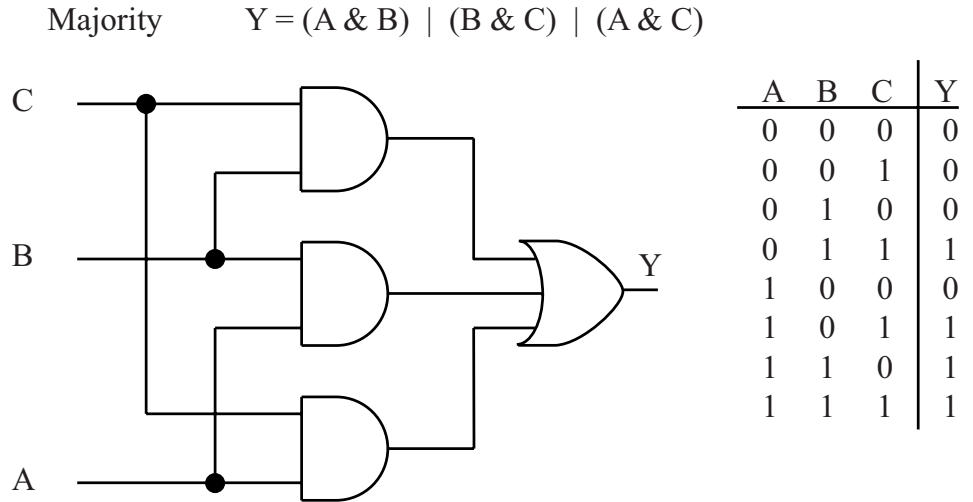
$$Y = (A \& (\sim B)) | ((\sim A) \& B) \quad (\text{Exclusive OR function}) \quad (1.1)$$

The shape distinctive symbol for a Boolean logic function is also referred to as the *logic gate* for the Boolean operation. A network of logic gates is an alternative representation of a Boolean equation. Figure 1.5 shows the Boolean equation of the XOR function drawn as a logic network using two-input gates.

Figure 1.6 gives the AND/OR network, Boolean equation, and truth table for a three-input majority function; so named because the output is a “1” only when a majority of the inputs are a “1”.



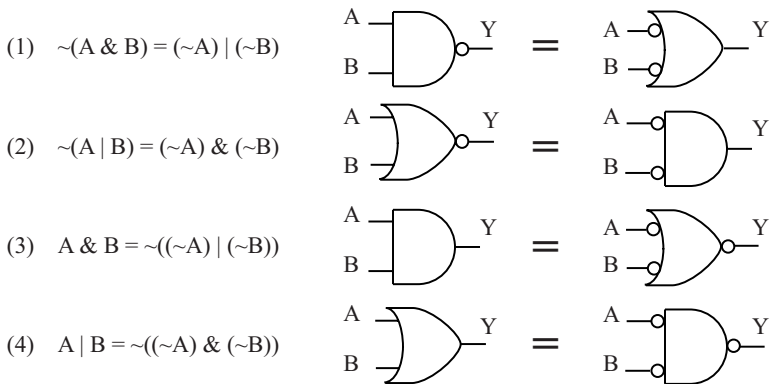
**FIGURE 1.5** AND/OR logic network for XOR function.



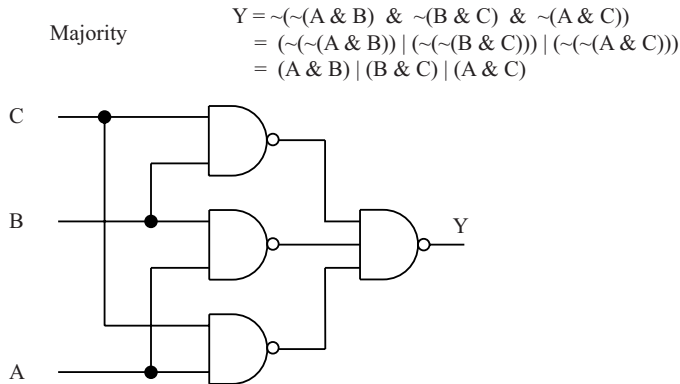
**FIGURE 1.6** AND/OR logic network for the three-input majority function.

An important law relating AND/OR/NOT relationships is known as *DeMorgan's Law*, with its forms shown in Figure 1.7. A "circle" or "bubble" on a gate input means that input is complemented. Note that a NAND function can be replaced by an OR function with complemented inputs (Form 1); while a NOR function can be replaced by an AND function with complemented inputs (Form 2). Forms 1 and 2 of DeMorgan's Law can be validated by comparing the truth tables of the left and right hand sides, while forms 3 and 4 follow from substitution of forms 1 and 2.

DeMorgan's law can be used to replace all of the AND/OR gates of Figure 1.6 with NAND gates as shown as in Figure 1.8. This is important as the physical implementation of a NAND gate using Complementary Metal Oxide Semiconductor (CMOS) transistors is faster and has fewer transistors than either an AND gate or an OR gate.



**FIGURE 1.7** DeMorgan’s Law.

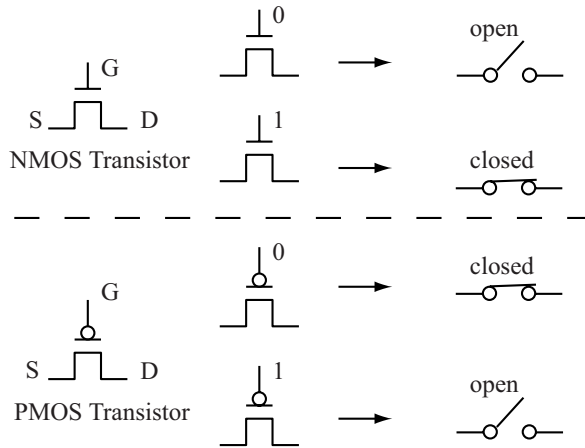


**FIGURE 1.8** NAND/NAND logic network for a three-input majority function.

### Logic Gate CMOS Implementations

CMOS transistors are the most common implementation method used today for logic gates, which form the building blocks for all digital computation methods. We review the basics of CMOS transistor operation here and revisit the topic in Chapter 8 when discussing computer input/output. The “C” in CMOS stands for *complementary*, which refers to the fact that there are two types of MOS transistors, N and P, whose operation is complementary to each other. Each MOS transistor type has three terminals: *Gate* (g), *Source* (s), and *Drain* (d). For our purposes, we will view a MOS transistor as an ideal switch whose operation is controlled by the gate terminal. The switch is either closed (connection exists between source and drain, so current flows between source and drain) or open (no connection between source

and drain, no current flow between source and drain). An N-type transistor is open when the gate has a logic “0”, and closed when the gate has a logic “1”. A P-type transistor has complementary operation; a “1” on the gate opens the switch, a “0” closes the switch. A logic “1” is physically represented by the power supply voltage of the logic gate, or *Vdd*. The power supply voltage used for a CMOS logic gate can vary widely, from 5 V (Volts) down to approximately 1.2 V. A logic “0” is physically represented by the system ground, or *Gnd*, which has a voltage value of 0 V. Figure 1.9 illustrates P and N transistor operation.

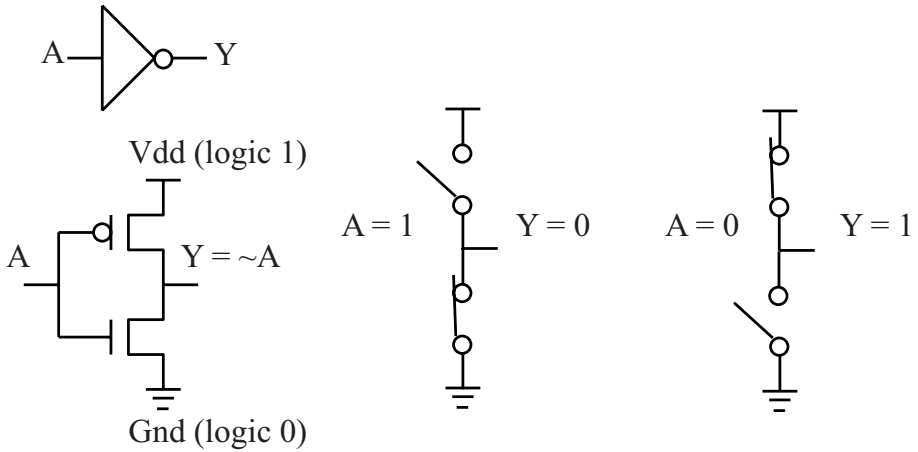


**FIGURE 1.9** CMOS transistor operation.

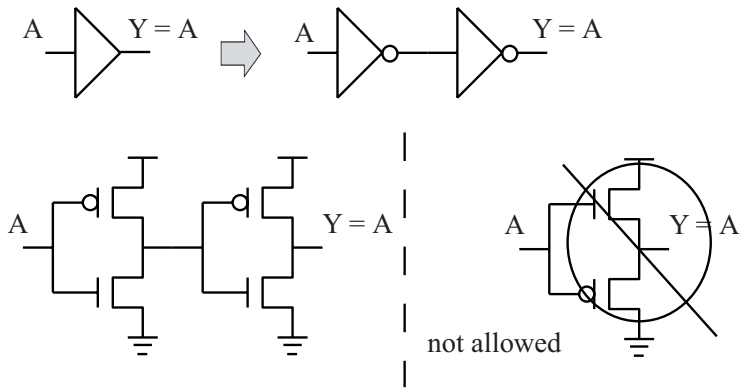
Multiple CMOS transistors can be connected to form logic gates. Figure 1.10 shows the simplest CMOS logic gate, which is the NOT function, or *inverter*. When the input value is “0”, the upper switch (the P transistor) is closed, while the lower switch (the N transistor) is open. This connects the output to *Vdd*, forcing the output to a “1”. When the input value is “1”, the upper switch is open, while the lower switch is closed. This connects the output to *Gnd*, forcing the output to a “0”. Thus, for an input of “0” the output is “1”; for an input of “1” the output is a “0”, which implements the NOT function.

Note that a buffer function  $Y = A$  is formed if two inverters are tied back to back as shown in Figure 1.11. It would seem that a better way to build a buffer is to switch the positions of the N and P transistors of the inverter; thus implementing the buffer with only two transistors instead of four. However, for physical reasons best left to an electronics book, a P transistor is always used to pass a “1” value, while an N transistor is always used to pass a “0” value. Thus, in digital logic, a P transistor is never tied to ground, and an N transistor is never tied to *Vdd*, so the

two-transistor buffer shown in Figure 1.11 is illegal. A noninverting CMOS logic function always takes two stages of inverting logic.



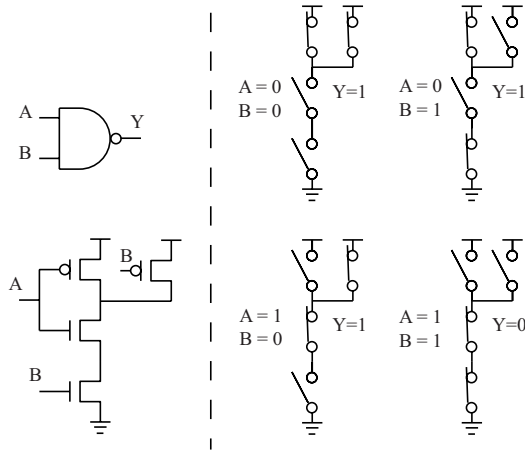
**FIGURE 1.10** CMOS inverter operation.



**FIGURE 1.11** CMOS buffer.

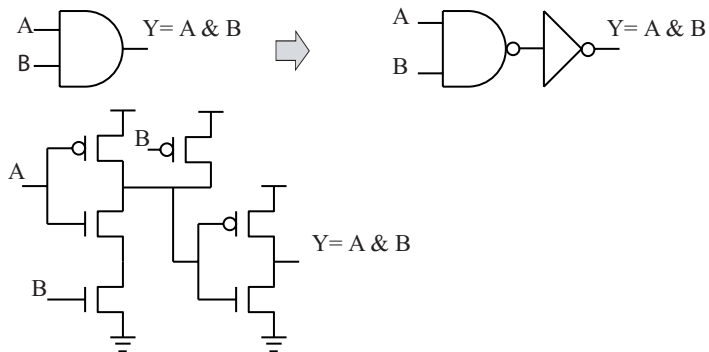
Figure 1.12 shows the transistor configuration and operation of a two-input CMOS NAND gate. Note that the output is connected to ground (both bottom transistors are closed) only when both inputs are a “1” value. Also observe that no combination of inputs provides a direct path between Vdd and Gnd; this would cause a *short* (low resistance path) between Vdd and Gnd resulting in excessive

current flow. The four-transistor configuration for a CMOS NOR gate is left as an exercise for the review problems.



**FIGURE 1.12** A CMOS NAND gate.

Figure 1.13 shows that a CMOS AND gate is actually built from a NAND gate followed by an inverter. Similarly, a CMOS OR gate is built from a NOR gate followed by an inverter. This clearly shows why replacing AND/OR logic with NAND gates via DeMorgan's Law is a good idea. The resulting circuit requires less transistors, meaning it is faster, consumes less power, and is cheaper to manufacture!



**FIGURE 1.13** A CMOS AND gate.

## 1.6 COMBINATIONAL BUILDING BLOCKS

---

Building logic circuits on a gate-by-gate basis is an exercise that can be quite fun, once. After that, one should look for shortcuts that reduce design time for complex circuits. One method for building complex combinational circuits quickly is to use combinational *building blocks*. The following sections describe some commonly used combinational building blocks; this list is by no means exhaustive. It should not be surprising that some of these building blocks (the adder and shifter) implement the arithmetic operations discussed earlier.

### The Multiplexer

A K-to-1 Multiplexer (or mux) steers one of K inputs to the output. The most common mux type is a 2-to-1 mux (two inputs, one output). A select control input *S* chooses the input that is passed to the output. The operation of 2-to-1 mux is written in *C* code as:

```
if (S) Y = A; else Y = B;
```

This *C* code description of a mux reads as “if *S* is non-zero, output *Y* is equal to *A*, else output *Y* is equal to *B*”. The Boolean equation for 2-to-1 1-bit mux is given in Equation 1.2.

$$Y = (S \& I1) \mid (\sim S \& I0) \quad (1.2)$$

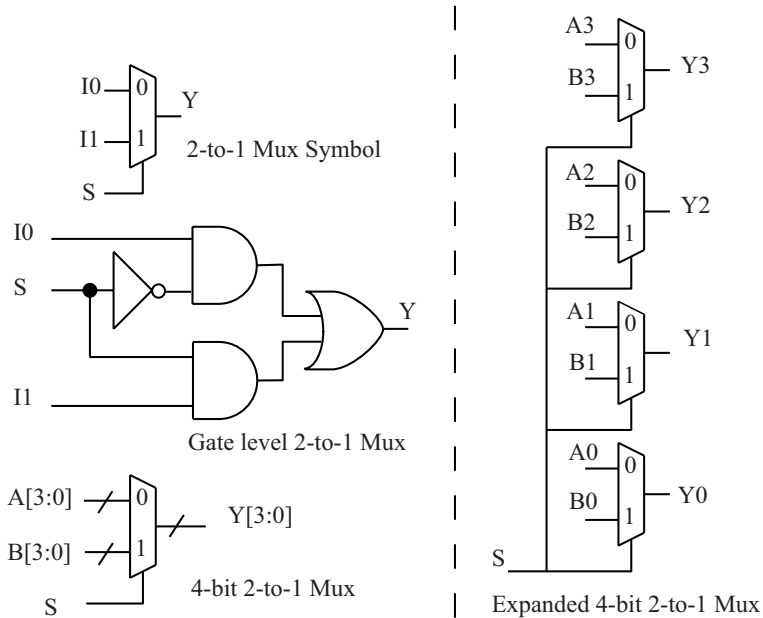
Figure 1.14 shows the gate equivalent for a 1-bit 2-to-1 mux and how a 4-bit 2-to-1 mux is built from four of these 1-bit building blocks. The 4-bit mux symbol in Figure 1.15 uses a *bus* labeling notation for the *A* and *B* inputs. In this context, a *bus* is simply a collection of parallel wires; a bus named *A* with *N* wires is designated as  $A[N-1:0]$ . The LSb and MSb of bus *A* are  $A[0]$  and  $A[N-1]$ , respectively. If  $N = 8$ , the entire bus *A* is labeled as  $A[7:0]$ , the LSb is  $A[0]$ , and the MSb is  $A[7]$ .

### The Adder

The adder takes two *N*-bit inputs (*A*, *B*) and computes the *N*-bit sum ( $A + B$ ). Most adders have a carry-in bit input for the LSb addition, and a carry-out bit output from the MSb addition. A *full adder* logic circuit that adds  $A + B + C_i$  (carry-in) and produces sum (*S*) and carry-out (*Co*) is a 1-bit building block of most adder circuits. Figure 1.15 shows the truth table, Boolean equations, and logic network for a full adder. The same figure shows how to build a 4-bit *ripple-carry* adder from four 1-bit full adders; the term *ripple-carry* is used because the carry ripples



from the rightmost bit to the leftmost bit. There are many other ways to build an adder; this is the simplest form of a binary adder.



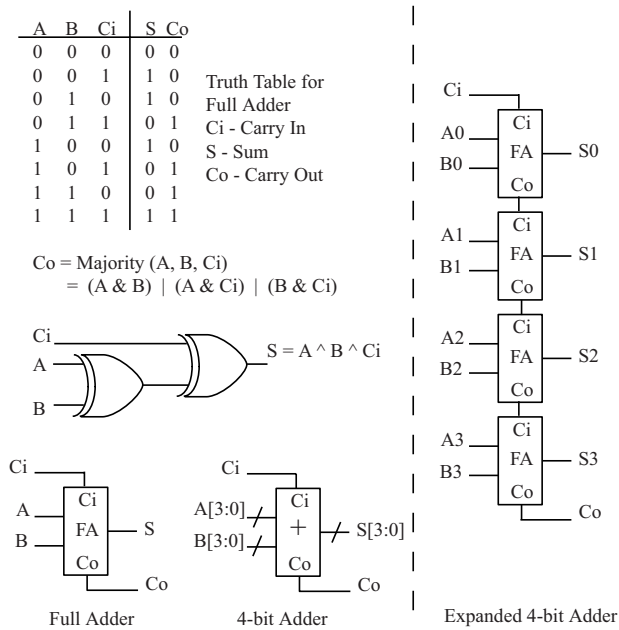
**FIGURE 1.14** One-bit 2-to-1 mux, 4-bit 2-to-1 mux.

### The Incrementer

The operation of an incrementer is described by the following C code:

```
if (INC) Y = A+1; else Y = A;
```

The INC (increment) input is a single bit input that determines if the N-bit output is  $A + 1$  or just  $A$ . An incrementer can be built from an adder by connecting all bits of one N-bit input to zero, and using the carry-in input as the INC input. This computes the sum  $Y = A + 0 + 1$  when  $INC = 1$  or the value  $Y = A + 0 + 0$  when  $INC = 0$ . There are more efficient methods in terms of logic gate count to implement an incrementer, but this illustrates the flexibility of combinational building blocks in implementing different functions.



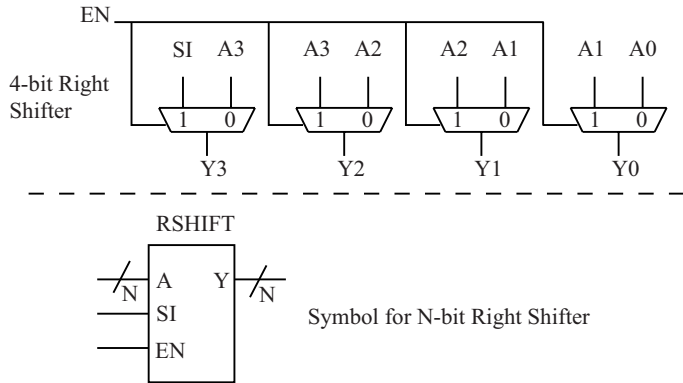
**FIGURE 1.15** One-bit adder, 4-bit ripple adder.

### The Shifter

There are many varieties of shifter combinational building blocks. The simplest type shifts by only one position, and in a fixed direction (either left or right). More complex types can shift multiple positions, and in either direction. Figure 1.16 shows the logic symbol for an N-bit right shifter, and the internal details of a 4-bit right shifter. When  $EN = 1$ , then  $Y = A \gg 1$  with the SI input providing the input bit for the MSb. When  $EN = 0$ , then  $Y = A$ , and the SI input has no effect. This is another example of simple combinational building blocks (2-to-1 muxes) being used to build a more complex combinational building block.

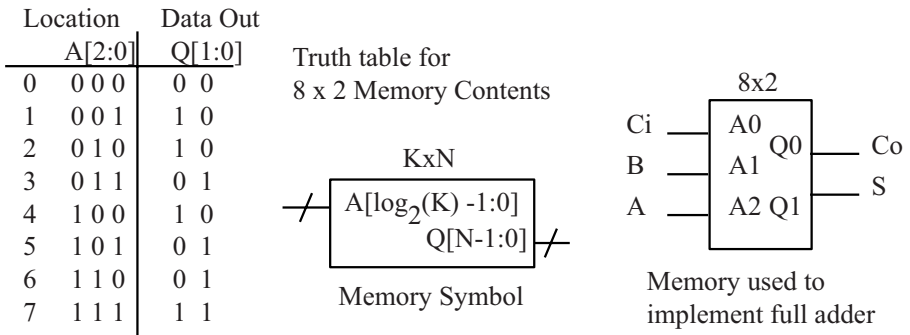
### Memory

A  $K \times N$  memory device has  $K$  locations, with each location containing  $N$  bits. Thus, a  $16 \times 1$  memory has 16 locations, with each location containing 1 bit. The *address* inputs specify the location whose contents appear on the data output. The number of bits required for the address is  $\lceil \log_2 K \rceil$ , a relationship seen previously as the address bits are used to uniquely specify 1 of  $K$  locations. The output databus has  $N$  bits as it is used to output the contents of a memory location. Following these rules, a  $16 \times 1$  memory has  $\lceil \log_2 16 \rceil = 4$  address inputs, and one data output. The



**FIGURE 1.16** N-bit right shift symbol and 4-bit right shift details.

idealistic view of memory presented here assumes a memory type with only these inputs and outputs, with memory contents loaded by some external means not discussed here. The most common usage of memory is to store data, but it can also be used to implement logic functions. Figure 1.17 shows an 8x2 memory used to implement the sum and carry-out equations of the full adder. The full adder inputs are connected to the 3-bit address bus as  $A = ADDR2$ ,  $B = ADDR1$ , and  $C_i = ADDR0$ . The 2-bit data output bus provides the outputs as  $C_o = Q0$  and  $S = Q1$ .



**FIGURE 1.17** Full adder implemented by an 8x2 memory.

**Sample Question:** How many address and data lines does a 4Kx16 memory have?

**Answer:** The number of address inputs is  $\lceil \log_2 4K \rceil = \lceil \log_2 2^2 \times 2^{10} \rceil = \lceil \log_2 2^{12} \rceil = 12$ . The number of data outputs is 16.

## 1.7 SEQUENTIAL LOGIC

The output of a combinational logic block is always uniquely defined by its current inputs. In contrast, the output of a *sequential* logic element is defined by its current inputs and also its *current state*, a value that is internal to the sequential logic element. A sequential logic element is a form of a memory device in that it retains internal state information between operations. In discussing sequential logic, the terms *asserted*, *negated*, *high-true*, and *low-true* are used in reference to inputs. When an input is *asserted*, it is said to contain a TRUE value; a *negated* input contains a FALSE value. A *high-true* input has a high voltage level for TRUE, and a low voltage level for FALSE. A *low-true* input has a low voltage level for TRUE, and a high voltage level for FALSE. The symbol for a sequential logic element uses a bubble on an input to indicate low-true.

### The Clock Signal

An important signal in a sequential logic circuit is the *clock* signal, whose waveform and associated definitions are shown in Figure 1.18. The following definitions are used in reference to clock waveforms:

- A *rising edge* is a transition from low to high; a *falling edge* is a transition from high to low.
- The *period* of a clock is the time in seconds (s) between two edges of the same type. A clock waveform typically has a fixed period; in other words, the period does not vary over time.
- The *frequency* of a clock is defined as  $1/(\text{period})$  measured in Hertz (Hz), where  $1 \text{ Hz} = 1/(1 \text{ s})$  (a 1 Hz clock has a period of 1 second).
- The *high pulse width* ( $PW_H$ ) is the amount of time the clock is high between a rising and falling edge, the *low pulse width* ( $PW_L$ ) is the amount of time the clock remains low between a falling and rising edge. The *duty cycle* is the percentage of time that the clock remains high.

Clock signal equations are summarized in Equations 1.3 through 1.7 as:

$$\text{Frequency} = \frac{1}{\text{Period}} \quad (1.3)$$

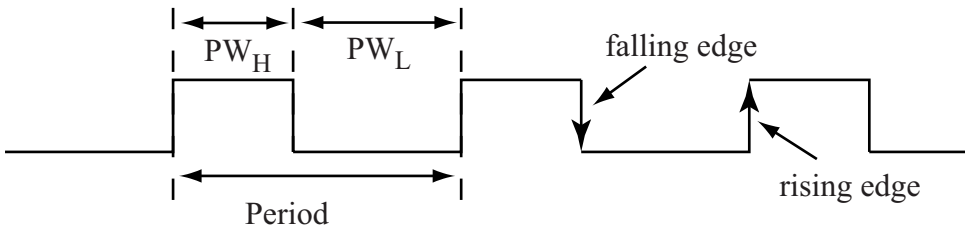
$$\text{Period} = \frac{1}{\text{Frequency}} \quad (1.4)$$

$$\text{Duty\_cycle} = \frac{PW_H}{\text{Period}} \times (100\%) \quad (1.5)$$

$$PW_H = \frac{(\text{Duty\_cycle} \times \text{Period})}{100} \quad (1.6)$$

$$PWL = \frac{((100-Duty\_cycle) \times Period)}{100} \tag{1.7}$$

Figure 1.18, the clock waveform, is an example of a *timing diagram*, which is a method for describing time-dependent behavior of digital systems. In a timing diagram, one or more waveforms can be displayed, and time is assumed to increase from left to right. A *waveform event* is a change in waveform value. If waveform event A occurs to the right of waveform event B, then A occurs after B in time. The clock waveform shown in Figure 1.18 is an idealized view of a clock signal; an oscilloscope trace of an actual clock signal would reveal visible rise and fall times for clock edges, and perhaps ringing (small oscillations) on the end of rising and falling edges.



**FIGURE 1.18** Clock signal definitions.

Table 1.4 lists commonly used units for time and frequency. A 1 kHz clock has a period of 1 ms, a 1 MHz clock has a period of 1 μs, and so forth. Observe that kHz has a lowercase “k”, where k = 1000; an uppercase K is reserved for the value 1024 (K = 2<sup>10</sup> = 1024). The suffixes M and G have values of 10<sup>6</sup> and 10<sup>9</sup>, respectively, when applied to time, frequency, and data transfer rate specifications (the values of M = 2<sup>20</sup> and G = 2<sup>30</sup> are used when referring to memory capacity). Timing and frequency specifications of digital circuits are contained in datasheets provided by the manufacturer. Time and frequency values are always specified using one of these units; in other words, a time is never specified as 1.05e-4; instead, it is specified as 105 μs.

**TABLE 1.4** Common Units for Time and Frequency

Time	Frequency
milliseconds = ms = 1e10 <sup>-3</sup> s	kilohertz = kHz = 1e10 <sup>3</sup> Hz
microseconds = μs = 1e10 <sup>-6</sup> s	megahertz = MHz = 1e10 <sup>6</sup> Hz
nanoseconds = ns = 1e10 <sup>-9</sup> s	gigahertz = GHz = 1e10 <sup>9</sup> Hz

**Sample Question: A clock has a duty cycle of 40%, and a frequency of 19.2 kHz. What is the period and low pulse width, in microseconds?**

*Answer:* The period is  $1/(19.2 \text{ kHz}) = 1/(19.2 \times 10^3) = 5.21 \times 10^{-5} \text{ s}$ . To convert this value to microseconds, do a unit conversion via:  $5.21 \times 10^{-5} \text{ s} \times 1 \mu\text{s}/1 \times 10^{-6} \text{ s} = 52.1 \mu\text{s}$ .  $PW_L = ((100 - \text{Duty\_cycle}) \times \text{Period})/100 = ((100 - 40) \times 52.1 \mu\text{s})/100 = 31.3 \mu\text{s}$ .

## The D Flip-Flop

There are many varieties of sequential logic elements. In this section, we review only the dominant type used in digital logic design, the *D Flip-Flop* (DFF). A DFF, as seen in Figure 1.19, can have the following input signals:

**CK (input):** The clock input; the arrival of the clock *active edge* sets the internal state of the DFF equal to the data input if the asynchronous inputs R, S are negated. The rising clock edge is the active clock edge for the DFF in Figure 1.19; it is said to be *rising-edge triggered*. A *falling-edge triggered* DFF has a bubble on its clock input.

**D (input):** The data input; the internal state of the DFF is set to this value on the arrival of an active clock edge if the asynchronous inputs R, S are negated. The D input is said to be a *synchronous input* as it can only affect the DFF on arrival of an active clock edge.

**S (input):** The set input; the internal state of the DFF becomes a “1” when this input is asserted. In Figure 1.19 this is a low-true input, so a low voltage on this input asserts set. This input is said to be *asynchronous* as its operation is independent of the clock input.

**R (input):** The reset input; the internal state of the DFF becomes a “0” when this input is asserted. In Figure 1.19 this is a low-true input, so a low voltage on this input asserts reset. This input is also an asynchronous input.

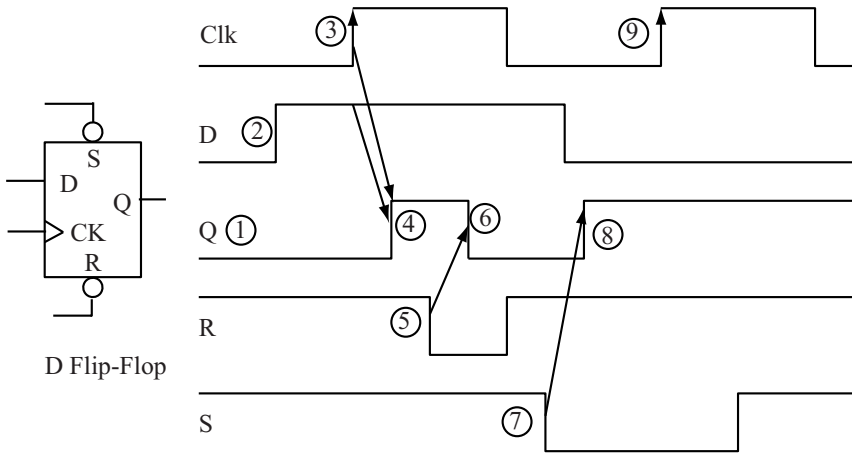
**Q (output):** The Q output is the value of the internal state bit.

Not all DFFs have S and R inputs; all DFFs have at least CK, D, and Q. The timing diagram in Figure 1.19 contains the following sequence of events:

1. The initial value of the DFF state bit is “0” as reflected by the Q output.
2. The D input becomes a “1”, but this has no effect on the Q output as a rising clock edge has not occurred.
3. A rising clock edge arrives.
4. The Q output changes to a “1” as the D value of “1” is clocked into the DFF by the rising clock edge. The time delay between the rising clock edge and

the Q output becoming a “1” is known as a *propagation delay*; changes cannot occur instantaneously in physical logic circuits. Propagation delay values are dependent upon the transistor topology of a logic gate, and have different values for different inputs and gate types. Timing diagrams in this book show propagation delay where appropriate.

5. The R input becomes a “0”, asserting this input.
6. The Q output becomes a “0” after a propagation delay; note that this occurs independent of the clock edge, as the R input is an asynchronous input.
7. The S input becomes a “0”, asserting this input.
8. The Q output becomes a “1” after a propagation delay; again, this occurs independent of the clock edge as the S input is an asynchronous input.
9. A rising clock edge arrives. The D input is a “0”, but this value is not clocked into the DFF as the S input is still asserted, which keeps the internal state at a “1”.



**FIGURE 1.19** D-flip-flop symbol and operation.

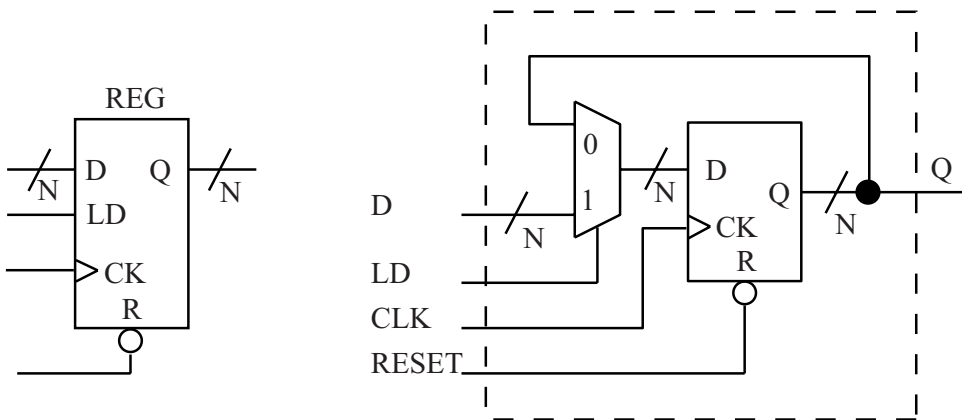
The DFF is the most commonly used edge-triggered sequential logic element in digital logic as it takes the fewest transistors to build. Other types of sequential logic elements that you may be familiar with are the JK Flip-Flop (JKFF) and T Flip-Flop (TFF); both of these can be built by placing logic gates around a DFF.

## 1.8 SEQUENTIAL BUILDING BLOCKS

Sequential building blocks are built using combinational building blocks and sequential logic elements. The following sections review some common sequential building blocks.

### The Register

A *register* is used to store an N-bit value over successive clock periods. One may think that paralleling N DFFs would suffice, but the problem with a DFF is that it samples its D input every active clock edge, potentially changing its value every active clock edge. Figure 1.20 shows an N-bit register built from an N-bit DFF and an N-bit 2-to-1 mux. When the load input (LD) is “1”, the DFF D input receives the value of the external D input by way of the mux, and thus the register is loaded with a new value on the next active clock edge. When LD is “0”, the DFF D input is connected to the DFF Q output; thus each active clock edge reloads the DFFs with their current output values. In this way, the register retains its current value over multiple clock cycles when the load input is negated (LD = 0). Registers are key components of all computers, and Figure 1.20 should be the physical element envisioned when the term *register* is used in future chapters on microprocessor operation.

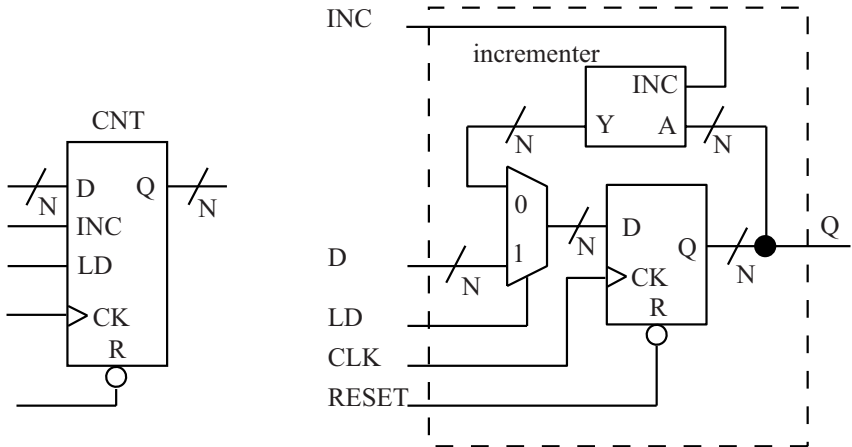


**FIGURE 1.20** N-bit register.



### The Counter

A counter is a register that has the additional capability of being able to increment (count up) or decrement (count down), or both. Figure 1.21 shows an N-bit counter that can count up. An N-bit incrementer has been added to the register design of Figure 1.21 to provide the counter functionality. The counter counts up by one when  $INC = 1$  and  $LD = 0$  on a rising clock edge as the DFF D input sees the value  $Q + 1$ . When  $INC = 0$  and  $LD = 1$ , the counter loads the external D input value on the active clock edge. This allows the counter to be initialized to a value other than zero. When  $INC = 0$  and  $LD = 0$ , the counter holds its current value. Counters are useful for producing addresses used to access memories, as sequential access of memory contents is a commonly needed operation in computer systems.



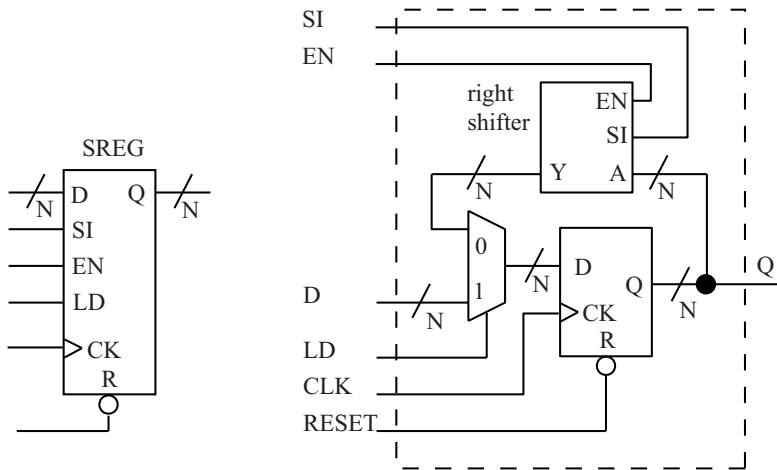
**FIGURE 1.21** N-bit up counter.

### The Shift Register

A shift register is a register that has the additional capability of being able to shift left, right, or both directions. Figure 1.22 shows a shift register that can shift right by 1 when  $SHIFT = 1$  and  $LD = 0$ . It uses the same design as the counter, except that a shift-right block has replaced the incrementer block. Shift registers are useful in computer input/output where an N-bit value must be communicated *serially*, in other words, bit by bit, to another device. The concept of *serial input/output* is covered in much detail in Chapters 9 and 11.

At this point, the usefulness of the concept of combinational and sequential building blocks should be apparent. One can easily envision other useful combinational building blocks such as a subtractor, decrementer, adder/subtractor, and

other sequential building blocks such as up/down counters, or combined counter/shift registers. These components form the basis for the logic circuits used within modern computers.



**FIGURE 1.22** N-bit shift register (right-shift by 1).

## 1.9 ENCODING CHARACTER DATA

Up to this point, the data encodings discussed have been for numerical representation of unsigned integers. Another common data type manipulated by computer systems is text, such as that printed on this page. The American Standard Code for Information Interchange (ASCII) is a 7-bit code used for encoding the Latin alphabet (the written form of the English language). The ASCII code contains uppercase letters, lowercase letters, punctuation marks, numerals, printer control codes, and special symbols. Table 1.5 shows the ASCII code; the top row specifies the most significant hex digit and the leftmost row the least significant hex digit of the 7-bit hex code. Thus, an “A” has the code 0x41, a “4” is 0x34, a “z” is 0x7A, and so on. The codes that are less than 0x20 are nonprintable characters that have various uses; some are printer control codes such as 0x0D (carriage return) and 0x0A (line feed). Eight bits are normally used to encode an ASCII character, with the eighth bit cleared to zero.

With the advent of the World Wide Web and the necessity to exchange binary-encoded text in other languages, the universal character-encoding standard, *Unicode*, was created (see [www.unicode.org](http://www.unicode.org) for more information). The Unicode goal is to provide a unique encoding for every character, numeral, punctuation mark, and

so forth, contained within every known language. The Unicode standard allows 8-bit (1 byte), 16-bit (2 byte), and 32-bit (4 byte) encodings. The 8-bit and 16-bit encodings are subsets of the 32-bit encodings; the first 128 codes (0x00 to 0x7F) are the same as the ASCII code for compatibility. Using 32 bits for each character allows for 4,294,967,296 unique characters, which is sufficient for the known character sets of the world. Individual character sets (Latin, Greek, Chinese, etc.) are assigned ranges within Unicode. Portions of the code are also reserved for use by private applications, so these codes are not assigned to any language. This book uses ASCII exclusively for character data encoding, but be aware that more sophisticated methods for text encoding exist.

**TABLE 1.5** ASCII Table

Most Significant Digit								
	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7
0x0	NUL	DLE	SPC	0	@	P	`	p
0x1	SOH	DC1	!	1	A	Q	a	q
0x2	STX	DC2	"	2	B	R	b	r
0x3	ETX	DC3	#	3	C	S	c	s
0x4	EOT	DC4	\$	4	D	T	d	t
0x5	ENQ	NAK	%	5	E	U	e	u
0x6	ACK	SYN	&	6	F	V	f	v
0x7	BEL	ETB	'	7	G	W	g	w
0x8	BS	CAN	(	8	H	X	h	x
0x9	TAB	EM	)	9	I	Y	i	y
0xA	LF	SUB	*	:	J	Z	j	z
0xB	VT	ESC	+	;	K	[	k	{
0xC	FF	FS	,	<	L	\	l	
0xD	CR	GS	-	=	M	]	m	}
0xE	SO	RS	.	>	N	^	n	~
0xF	SI	US	/	?	O	_	o	DEL

**Sample Question: What character string is represented by “0x48 0x65 0x6c 0x6c 0x6F 0x20 0x57 0x6F 0x72 0x6c 0x64 0x21”?**

*Answer:* Translating character by character yields the popular test message: “Hello World!”. Note that the string contains a space character (0x20) and an exclamation mark “!” (0x21).

## SUMMARY

---

In this chapter, we hopefully have refreshed some topics you have encountered previously concerning number systems, binary encoding, Boolean algebra, logic gates, combinational building blocks, and sequential building blocks. In the next chapter, we use these building blocks to introduce the concept of a stored program machine as a means of implementing a digital system.

## REVIEW PROBLEMS

---

1. How many bits does it take to represent 40 items?
2. What is the largest unsigned integer that can be represented in 7 bits?
3. Convert the value 120 to binary using 8 bits.
4. Convert 89 to hex using 8 bits.
5. Convert 0xF4 to binary.
6. Convert 0xF4 to decimal.
7. Convert the value 0b10110111 to decimal.
8. Compute  $0xB2 + 0x9F$ , give the result in hex.
9. Compute  $0xB2 - 0x9F$  and give the result in hex. Check your work by verifying that  $0xB2 + \sim(0x9F) + 0x1$  produces the same result. To compute  $\sim(0x9F)$ , complement each bit.
10. Draw the logic network and derive the truth table for the logic function  $F = (A \& B) | C$ .
11. Derive the CMOS transistor network that implements the NOR function.
12. Compute  $0xC3 \gg 2$ , give the value in hex (this is a right shift by two).
13. Compute  $0x2A \ll 1$ , give the value in hex (this is a left shift by one).
14. What is the period of a 400 kHz clock in microseconds?
15. Given a 30% duty cycle clock, with a high pulse width of 20  $\mu$ s, what is the clock frequency in kHz?
16. Design an N-bit subtractor using an adder with a carry-in input and the fact that  $A - B = A + \sim B + 1$ .

17. Design an N-bit adder/subtractor that has an external input called SUB that when “1”, performs a subtraction, when “0”, performs an addition (Hint: use an adder with a carry-in input, and a mux).
18. Design a left-shift by one combinational building block.
19. Design a counter that can count either up or down. Assume that you have incrementer and decremter building blocks available.
20. Write your first name as 8-bit hex values using ASCII encoding.

# 2

## The Stored Program Machine

### In This Chapter

- Problem Solving the Digital Way
- Finite State Machine Design
- A Stored Program Machine
- Modern Computers

This chapter introduces the fundamental concepts of computer operation by implementing a controller both as a finite state machine and as a stored program machine.

### 2.1 LEARNING OBJECTIVES

---

After reading this chapter, you will be able to:

- Compare and contrast controller designs using finite state machine and stored program machine approaches.
- Describe the operation of a finite state machine via an algorithmic state machine chart.

- Implement a finite state machine using a one-hot state encoding method.
- Discuss the basic elements of a stored program machine.
- Describe the meaning of the terms *opcode*, *machine word*, *instruction mnemonic*, *address bus*, *data bus*, *instruction pointer*, *assembly*, and *disassembly*.
- Describe the fetch and execute sequence of a stored program machine.
- Convert a simple assembly language program to machine code, and vice versa.
- Follow the execution of a simple assembly language program.
- Write a simple assembly language program to solve a specified problem.

The preceding tasks introduce you to the concept of stored program machines, of which the PIC18F242 microcontroller is a prime example, and is the principle focus of the rest of this book. The PIC18F242 microcontroller is simply a more complex version of the stored program machine discussed in the following sections. This chapter provides you with the first chance to dip your toes into the vast ocean of microprocessor operation, programming, and application.

## **2.2 PROBLEM SOLVING THE DIGITAL WAY**

---

Digital systems provide solutions for problems in which real-world inputs can be converted to a digital representation, which is then processed by a clever use of combinational and sequential building blocks to produce a digital output that is converted back to a useful quantity in the real world. As an example, consider a digital voice recorder:

- Pushbutton inputs on the recorder determine if the recorder is in record mode or playback mode. A microphone input converts voice that varies in amplitude over time to a continuously varying voltage between 0 V and the power supply voltage, where the voltage fluctuations are the sound wave variations of human speech.
- A building block called an *analog-to-digital converter* (ADC) produces a digital representation of the microphone output voltage at regular time intervals. Each converted digital value is called a *voice sample*.
- A digital building block called a *controller* monitors the button inputs on the recorder.
- In record mode, the controller reads the voice samples from the ADC and stores them in a memory device.
- In playback mode, the controller reads the memory contents sequentially, and sends each voice sample to a building block called a *digital-to-analog converter* (DAC) that converts a digital input value to a voltage value between 0 V and a

reference voltage VREF (which can be the power supply voltage). This voltage signal drives a speaker, allowing the recorded voice to be heard.

Analog-to-digital converters and digital-to-analog converters are essential parts of digital systems and are covered in Chapter 12, “Data Conversion.” In this chapter, we are concerned with the design of the controller that sequences the events within a digital system.

Two basic choices exist for building a controller: a *finite state machine* (FSM) or a *stored program machine* (also known as a Von Neumann machine after the scientist who first proposed this approach [1], [2]). In a FSM, dedicated logic implements the event sequence required for a task. In a stored program machine, data stored in a memory specifies the event sequence for a particular task. An advantage of a FSM is that it usually takes fewer clock cycles than a stored program machine to perform a particular task. The principle advantage of a stored program machine is flexibility; a different task can be implemented by simply changing memory contents. A FSM requires redesign of its internal logic to implement a different task, which is a much more difficult problem than memory modification.

To illustrate the differences between these approaches, consider a digital system that continuously outputs the digits of a phone number,  $Y_1Y_2Y_3-Z_1Z_2Z_3Z_4$ . In local mode, only  $Z_1Z_2Z_3Z_4$  are output, while in nonlocal mode all digits are output. In our controller designs for this problem, a common set of inputs and outputs is used for comparison purposes:

**LOC (input):** When “1” the system is in local mode; when “0”, in nonlocal mode.

**CLK (input):** This is obviously a sequential system, as the output cannot be solely determined by the LOC input, so the system requires DFFs, which implies that a clock input is needed.

**RESET# (input):** All sequential digital systems need an input signal that initializes the internal DFFs to a known state after power is applied, as the internal state of DFFs are indeterminate on power up. This signal is usually called *reset*; the “#” in the name indicates that this input is low true.

**DOUT[3:0] (output):** This 4-bit output bus is used to sequentially output the digits of the phone number. A digit has a value between 0 and 9; so 4 bits are sufficient for encoding purposes. Binary encoding is used for encoding these digits.

The following sections detail controller designs using both FSM and stored program machine approaches. Contrasting the two controller designs emphasizes the strengths and weaknesses of each approach.



## 2.3 FINITE STATE MACHINE DESIGN

---

An FSM can be thought of as a sequential building block that is custom designed to solve a particular problem. The problem solved by an FSM is described as a series of transitions between *states*, where a transition from the *present state* to the *next state* is determined by the present state and the current inputs. The outputs of the FSM are also determined either by only the present state or a combination of the present state and current inputs. In this example, we use an *Algorithmic State Machine* (ASM) chart to describe the state sequencing of an FSM. An ASM chart is similar to a software flow chart for those familiar with that notation. Symbols used in an ASM chart are:

**Rectangle:** This indicates a state. Outputs asserted during this state are written within the rectangle; these outputs are called *unconditional outputs*, as the assertion is only dependent upon the state, and not an external output. Any outputs that do not appear in a state are assumed negated.

**Diamond:** A decision point, with the input that the decision is based upon written within the diamond.

**Oval:** This can only appear after a decision point and is used for *conditional outputs*, which is an output dependent upon both a state and some set of inputs.

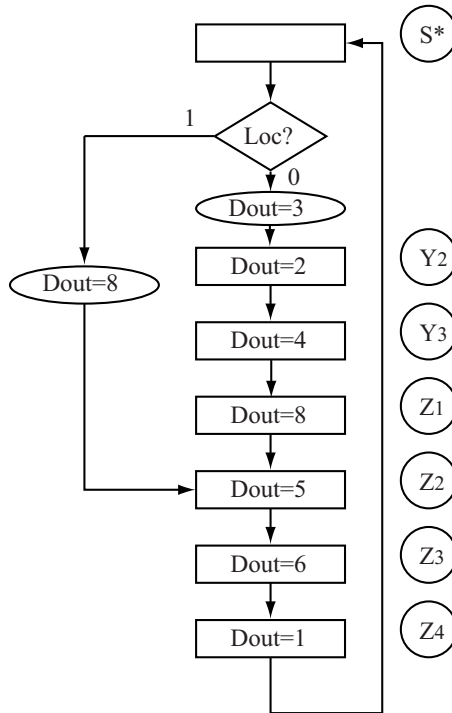
**Circle:** This appears next to each state and contains the state name.

The ASM chart in Figure 2.1 shows an FSM that outputs the number 324-8561 according to our problem specification. The ASM contains seven states with the  $S^*$  state being the state entered upon reset (the reset state is designated by the  $*$  symbol). The state following  $S^*$  is dependent upon the LOC input; if  $LOC = 1$ , then the next state is  $Z_2$ , else the next state is  $Y_2$ . The state progression for  $LOC = 0$  is  $S^*, Y_2, Y_3, Z_1, Z_2, Z_3, Z_4$ , which then loops back to  $S^*$ . Each state requires one clock cycle, so this state progression requires seven clock cycles. If  $LOC = 1$ , the state progression is  $S^*, Z_2, Z_3, Z_4$ , again looping back to state  $S^*$ . The LOC input is only checked in the  $S^*$  state, so the sequence cannot be altered once state  $S^*$  has been exited. The DOUT output of the  $S^*$  state is conditional upon the LOC input;  $DOUT = 8$  if  $LOC = 1$ , else  $DOUT = 3$ . The DOUT output in all other states is unconditional.

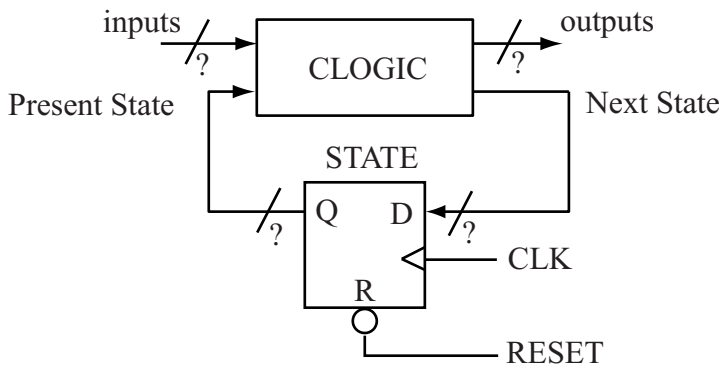
### Finite State Machine Implementation

A generic block diagram of an FSM implementation is shown in Figure 2.2. The state DFFs contain the present state of the FSM. The CLOGIC block is the combi-

national logic that produces the next state inputs to the state DFFs based upon the present state and external inputs. State changes occur on the active clock edge.



**FIGURE 2.1** ASM chart for the number sequence 324-8561.



**FIGURE 2.2** Finite state machine block diagram.

The first task in implementing this FSM is *state assignment*, which means assigning binary encodings for each state. The state assignment affects the number of state DFFs required and the combinational logic within the CLOGIC block. Table 2.1 lists two choices for state encoding; there are many others. Using binary encoding requires only three DFFs but more complex combinational logic; one-hot encoding requires six DFFs but simplifies the task of determining the required combinational logic equations. This is meant as an illustrative exercise, and not an exhaustive treatise on FSM design, so we will use one-hot encoding.

**TABLE 2.1** Two Possibilities for State Assignment

State	Binary Encoding	One-Hot Encoding
S*	000	0000001
Y2	001	0000010
Y3	010	0000100
Z1	011	0001000
Z2	100	0010000
Z3	101	0100000
Z4	110	1000000

Two sets of Boolean equations are required for the CLOGIC block: the set that controls the state sequencing, and the set that produces the required output values.

The set of Boolean equations that controls the state sequencing consists of seven equations, one for each D input of a DFF. An advantage of one-hot encoding is that these equations can be easily determined by inspection of the ASM chart. As an example, consider the equation for the D input of state S\*, designated by D\_S\*. This state is entered on the next clock cycle from state Z4, and we know that the Q output of the state Z4 DFF, denoted by Q\_Z4, will only be a “1” when the FSM is in state Z4. Thus, the D\_S\* Boolean equation has only one term as shown in Equation 2.1.

$$D_{S^*} = Q_{Z_4}; \tag{2.1}$$

The equation for the D input of the Y2 state is a bit more interesting, as this state is entered only if the current state is S\* and LOC is “0” as stated in Equation 2.2.

$$D\_Y_2 = (Q\_S^*) \& (\sim\text{LOC}); \quad (2.2)$$

The Boolean equations for the remaining DFF inputs are derived through similar reasoning as shown in Equations 2.3 through 2.7.

$$D\_Y_3 = Q\_Y_2; \quad (2.3)$$

$$D\_Z_1 = Q\_Y_3; \quad (2.4)$$

$$D\_Z_2 = ((Q\_S^*) \& (\text{LOC})) \mid (Q\_Z_1); \quad (2.5)$$

$$D\_Z_3 = Q\_Z_2; \quad (2.6)$$

$$D\_Z_4 = Q\_Z_3; \quad (2.7)$$

The previous Boolean equations for state sequencing do not include the reset behavior. The reset signal input is tied to the set (S) input of the DFF for state  $S^*$ , and to the reset (R) input of the remaining state DFFs. In this way, reset assertion forces the FSM to enter state  $S^*$  while reset negation allows normal state sequencing.

The Boolean equations for the DOUT outputs depend upon the current state and LOC input values. Table 2.2 lists the binary output values for DOUT given the current state and LOC values.

**TABLE 2.2** Output Values for DOUT Referenced to States

State	One-Hot Encoding	DOUT (LOC = 0)	DOUT (LOC = 1)
$S^*$	0000001	0011 (3)	1000 (8)
Y2	0000010	0010 (2)	0010 (2)
Y3	0000100	0100 (4)	0100 (4)
Z1	0001000	1000 (8)	1000 (8)
Z2	0010000	0101 (5)	0101 (5)
Z3	0100000	0110 (6)	0110 (6)
Z4	1000000	0001 (1)	0001 (1)

A Boolean logic equation is needed for each of the 4 bits of DOUT. We will write these equations by inspection and make no attempt at minimizing the amount of logic required for implementation. From Table 2.2, we see that the DOUT[0] output is a “1” for the following conditions: in state  $S^*$  and  $LOC = 0$ , or in states  $Z_2$  or  $Z_4$ . This is expressed as a Boolean equation for the DOUT[0] output as shown in Equation 2.8.

$$DOUT[0] = ((Q\_S^*) \& (\sim LOC)) \mid (Q\_Z_2) \mid (Q\_Z_4); \tag{2.8}$$

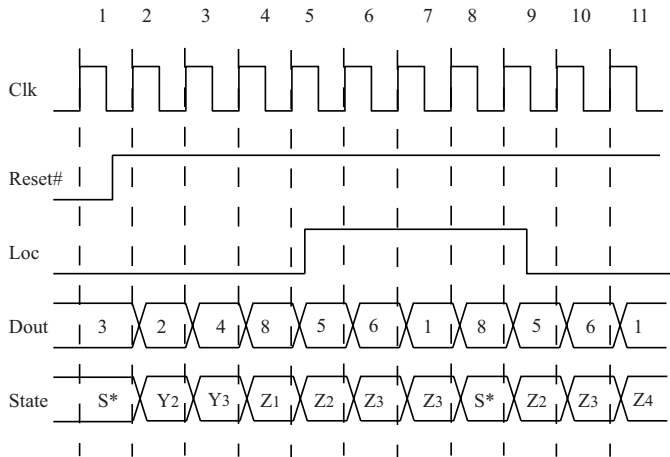
The equations for the other DOUT bits are derived similarly as shown in Equations 2.9 through 2.11.

$$DOUT[1] = ((Q\_S^*) \& (\sim LOC)) \mid (Q\_Y_2) \mid (Q\_Z_3); \tag{2.9}$$

$$DOUT[2] = (Q\_Y_3) \mid (Q\_Z_2) \mid (Q\_Z_3); \tag{2.10}$$

$$DOUT[3] = ((Q\_S^*) \& (LOC)) \mid (Q\_Z_1); \tag{2.11}$$

These Boolean equations can be mapped to logic gates for implementation; a simulation of the resulting gate level system is shown in Figure 2.3. Note that in clock cycle #1, the system is in state  $S^*$  with  $DOUT = 3$ , but when state  $S^*$  is reentered in clock cycle #8 the output is  $DOUT = 8$ , because now  $LOC = 1$  (the changing of the LOC input value is arbitrary as it is an external input; it is changed in clock cycle #5 to illustrate that the LOC input only affects the FSM behavior in state  $S^*$ ).



**FIGURE 2.3** Simulation of the FSM implementation.

Note that if the phone number is changed, only the Boolean equations for DOUT must be changed; the Boolean equations for the state sequencing remain the same. However, if the number of phone digits is changed, say to  $X_1X_2X_3-Y_1Y_2Y_3-Z_1Z_2Z_3Z_4$ , this requires a new state sequence and hence different logic for state sequencing.

## 2.4 A STORED PROGRAM MACHINE

---

A stored program machine is the formal term for a *computer*. Three components of every stored program machine are:

- **Input/Output (IO) signals** that are used for interfacing with the external world.
- **Memory** that stores the *instructions* that determine the sequence of events performed by the computer. An instruction is a binary datum that is usually a fixed width. Memory also stores data that instructions manipulate.
- **Control logic** that decodes the instructions and *executes* the actions specified by an instruction.

The common elements between a finite state machine and stored program machine are IO and control; the memory component is the differentiating factor. Memory provides flexibility for a stored program machine; a stored program machine can be programmed to perform a different task by changing the instructions stored in memory. A *program* is a sequence of instructions that implement a particular task. A stored program machine continuously performs a *fetch/execute* action, in which an instruction is fetched from memory, and then executed. Instruction fetches typically progress through memory in a sequential manner. Instructions are divided into different classes of instructions; some instructions perform arithmetic operations, some perform input/output, and some perform control. An example of arithmetic instruction execution is binary addition. An example of an input/output instruction execution is placing a data value on an output bus. An example of a control instruction execution is a `goto x` (jump) that fetches the next instruction from location  $x$  instead of the next sequential memory location.

### Instruction Set Design and Assembly Language

The design of our stored program machine begins with determining what type of instructions it should execute, and the format or binary encoding of these instructions. To determine this, we will describe the task as a sequence of statements in the C programming language, as seen in Listing 2.1.

**LISTING 2.1** C language program of the number sequencing task.

```
START:
  If (10c) goto LOCAL;
  dout = 3;
  dout = 2;
  dout = 4;
LOCAL:
  dout = 8;
  dout = 5;
  dout = 6;
  dout = 1;
  goto START;
```

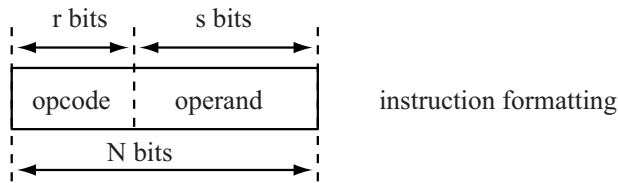
These statements represent three distinct operations:

- An output operation as specified by the statement `dout = 2`, which places the value `0b0010` on the DOUT data bus.
- An unconditional transfer of control, also known as a *goto* or *jump*, as specified by the statement `goto START`, which says to fetch the next instruction from the memory location represented by the label `START`.
- A conditional transfer of control, also known as a *branch* or *jump conditionally*, as specified by the statement `if(10c)goto LOCAL`. If `10c` is nonzero (i.e., true), the next instruction fetched is the one at label `LOCAL`. If `10c` is zero (i.e., false), the next sequential instruction in memory is fetched, which is `dout = 3`.

These instruction types must be assigned a binary encoding so that they can be stored in memory. The encoding of an instruction must specify the type of instruction, and the data required by the instruction (if the instruction requires data). The instruction bits that specify the instruction type form a bit field that is called the *opcode*. The data required by the instruction is more formally known as the *operand*. Figure 2.4 illustrates how instruction encoding is divided into opcode and operand fields.

Two bits are needed for the opcode to encode the three types of instructions in our computer. Table 2.3 lists the *instruction set* for our number sequencer computer (NSC). The leftmost column contains the instruction *mnemonic*, which is the human-readable form of an instruction. The middle column gives the opcode encoding, and the rightmost column the instruction operand.

Opcode encoding is usually chosen so that classes of instructions can be easily distinguished. In this case, the two control instructions `JC` and `JMP`, are distinguished from the IO instruction `OUT` by the most significant bit of the opcode. The `OUT` instruction operand is the 4-bit digit that appears on the DOUT data bus after the instruction is executed. The `JMP` and `JC` instructions have the same type of operand, which is the memory address of the target instruction of the jump.



**FIGURE 2.4** Instruction encoding split into opcode and operand fields.

The number of bits required for this memory address depends on the maximum number of memory locations allowed in our number sequencer computer. For now, a total of 4 bits is used, which limits the total number of memory locations to  $2^4 = 16$ . Each memory location contains one instruction, so the maximum number of

**TABLE 2.3** Instruction Encodings for the Number Sequencer Computer

Mnemonic	Opcode	Operand
JMP	00	instruction address
JC	01	instruction address
OUT	10	4-bit data

instructions in any program written for our computer is 16 instructions. Each of our instructions is 6 bits wide (2 bits opcode, 4 bits data), so a 16x6 memory device is required for storing the instructions of our program.

Listing 2.2 gives the *C* program of the number sequencing task translated into the instructions used by our computer. The translation of a high-level language to the native instructions of a computer is called *compilation*. A program specified using the instruction mnemonics of an instruction set is called an *assembly language* program. A computer program called a *compiler* is normally used to translate a high-level language program to assembly language, but in this book the process is done manually in many cases to illustrate the linkage between *C* statements and assembly language statements.

**LISTING 2.2** Assembly language program for the number sequencing task.

```
START:
  JC LOCAL
  OUT 3
```



```

    OUT 2
    OUT 4
LOCAL :
    OUT 8
    OUT 5
    OUT 6
    OUT 1
    JMP START

```

The next step is to translate our program into binary so that it can be stored in memory. This process is called *assembly*, and the resulting binary codes are called the *machine code* of the assembly language program. The reverse process of converting machine code to assembly language mnemonics is called *disassembly*. The assembly process is done in two passes; the first pass assembly does not assemble any `jmp/jc` instructions whose jump destination is ahead of the current instruction as the memory location of the jump destination is unknown. After the first pass assembly, the locations of all instructions are known so the second pass completes the assembly of any `jmp/jc` instructions that were not completed in the first pass. Table 2.4 gives the first-pass result of this assembly process. The leftmost column contains the memory location where the machine code for the instruction is placed. We will place our instructions starting at location `0x0` in memory for reasons that are explained later. Each assembly language statement is translated individually to its machine code representation. Observe that the operand field of the first instruction is left as “????” because the location of the instruction represented by the label `LOCAL` is unknown when this instruction is assembled during the first pass. The last instruction, `JMP START`, can be completely translated because the label `START` stands for the first instruction, which has already been translated and is located at location `0x0`.

**TABLE 2.4** First-Pass Assembly of the Number Sequencing Program

Memory Location	Machine Code	Instruction
0x0	01 ????	START: JC LOCAL
0x1	10 0011	OUT 3
0x2	10 0010	OUT 2
0x3	10 0100	OUT 4
0x4	10 1000	LOCAL: OUT 8
0x5	10 0101	OUT 5
0x6	10 0110	OUT 6
0x7	10 0001	OUT 1
0x8	00 0000	JMP START

Once the first pass assembly is complete, we see that the value for label `START` is the memory address `0x4`. Table 2.5 gives the second-pass assembly of the number sequencing program; the values of all labels are now known so the machine code translation is complete.

**TABLE 2.5** Second-Pass Assembly of the Number Sequencing Program

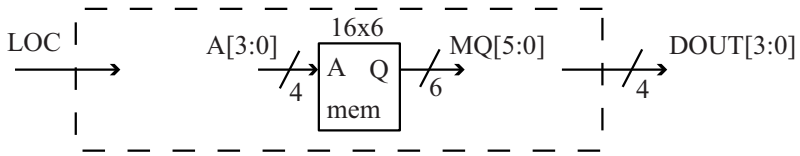
Memory Location	Machine Code	Instruction
0x0	01 0100	START: JC LOCAL
0x1	10 0011	OUT 3
0x2	10 0010	OUT 2
0x3	10 0100	OUT 4
0x4	10 1000	LOCAL: OUT 8
0x5	10 0101	OUT 5
0x6	10 0110	OUT 6
0x7	10 0001	OUT 1
0x8	00 0000	JMP START

Our program consists of nine assembly language instructions; the remaining seven locations of our  $16 \times 6$  memory are not needed for this program.

## Hardware Design

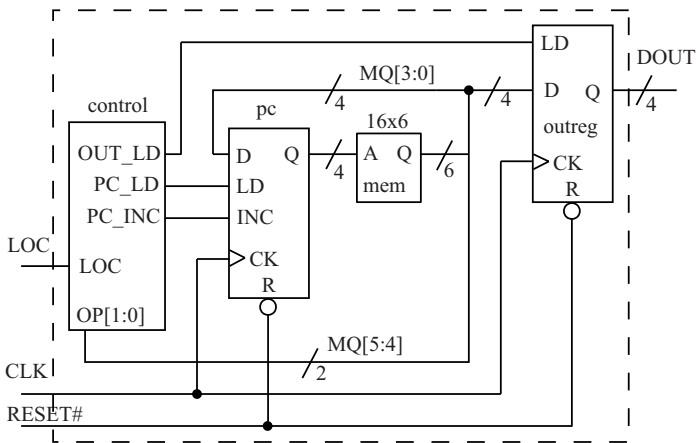
We must now design hardware that executes the machine code program of our number sequencing task. We will use the sequential blocks covered in the first chapter to make our job easier. Figure 2.5 shows what is currently known about the hardware components of our computer; it has a  $16 \times 6$  memory, a 1-bit input called `LOC`, and a 4-bit output called `DOUT`.

In looking at Figure 2.5, we see several busses (`DOUT`, Memory address bus, Memory Data bus) that connect to nothing, so this provides a logical place to begin adding components. Recall that a register is a sequential building block used to hold a value over one or more clock cycles. A 4-bit register is needed to drive the `DOUT` data bus; the register contents are modified by execution of the `OUT` instruction. Recall that a counter is a register that has the capability of counting up or down, or both. Instructions are usually fetched sequentially from memory, which means that the addresses provided to memory generally proceed in binary counting order. Thus, a 4-bit counter is the natural choice for providing the address to memory.



**FIGURE 2.5** Number sequencer computer initial components.

This counter is an integral part of any computer, and is known as the *Program Counter (PC)*, or *Instruction Pointer (IP)*. The PC register contains the address of the instruction currently being fetched from memory. Figure 2.6 shows the number sequencing computer hardware modified to contain the PC and output registers.



**FIGURE 2.6** Number sequencing computer with program counter, output register added.

The reset inputs of the PC and output register are tied to the external reset input. When reset is asserted, the PC register is cleared, which means the first instruction fetched from memory is at location 0, requiring the instructions in Table 2.5 to begin at location 0. Observe that the LD and INC control inputs of the PC, and the LD input of the output register now connect to a general block called *control*. Modification of the PC and output registers is controlled by the current instruction. Table 2.6 lists the PC and output register control input values for each instruction. The output register is loaded when the OUT instruction is executed. The PC register is incremented if an OUT instruction is executed, or when a JC instruction is executed and LOC = 0. The PC register is loaded when a JMP instruction is executed, or when a JC instruction is executed and LOC = 1.

**TABLE 2.6** PC, Output Register Inputs for Instruction Execution

<b>Instruction</b>	<b>PC Register</b>	<b>Output Register</b>
JC	LD = LOC, INC = ~LOC	LD = 0
JMP	LD = 1, INC = 0	LD = 0
OUT	LD = 0, INC = 1	LD = 1

The Boolean equations for the PC and output register control lines are shown in Equations 2.12 through 2.14 as determined from Table 2.6.

$$PC\_LD = (\sim MQ5 \& \sim MQ4) \mid (\sim MQ5 \& MQ4 \& LOC) \quad (2.12)$$

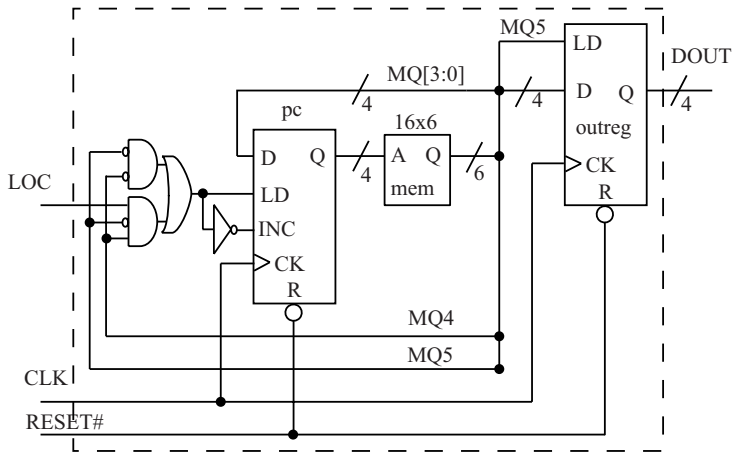
$$PC\_INC = \sim PC\_LD \quad (2.13)$$

$$OUT\_LD = MQ5 \quad (2.14)$$

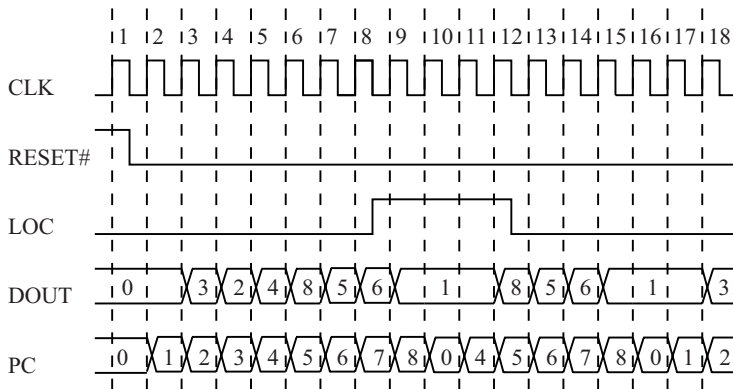
The values MQ5 and MQ4 are the two most significant bits of the instruction word fetched from memory; these are the opcode bits of the instruction. The Boolean equation for the PC\_INC signal is simply the complement of PC\_LD, as the PC is incremented if it not being loaded. The logic gates that implement the Boolean equations for PC\_LD, PC\_INC, and OUT\_LD are placed in the control logic block as shown in Figure 2.7, which completes our design.

Figure 2.8 shows a simulation of the number sequencing computer. The PC waveform is the program counter value, which can be considered the current state of this sequential system.

Table 2.7 compares the number of clock cycles required for each number sequence in the finite state machine and stored program machine implementations. The stored program machine requires two more clock cycles for each sequence because of the JC instruction at the beginning of the sequence, and the JMP at the end of the sequence. In general, a stored program machine will take more clock cycles to accomplish a task than a finite state machine, which is the penalty for increased flexibility and is the typical tradeoff when evaluating whether to use a general-purpose computer versus dedicated logic as a problem solution.



**FIGURE 2.7** Complete hardware design for the number sequencing computer.



**FIGURE 2.8** Simulation of the number sequencing computer.

**TABLE 2.7** FSM versus SPM Clock Cycles

	FSM Clock Cycles	SPM Clock Cycles
LOC = 0	7	9
LOC = 1	4	6

## 2.5 MODERN COMPUTERS

---

How does our number sequencing computer (NSC) compare against modern computers?

- Programs for the NSC reside in memory that has a maximum of 16 locations; modern computers have memory that contains billions of locations.
- The data register in the NSC is 4 bits wide; modern computers have data registers that are typically 32 or 64 bits wide, and are larger in some cases.
- The NSC has three different instructions; modern computers have tens to hundreds of different instructions.
- The NSC can be implemented in less than 100 gates; modern computers can require millions of logic gates.

Despite these differences, the NSC and modern computers share the basic components of all computers: input/output, memory, and control. These three components work together to fetch and execute instructions that are stored in memory.

### SUMMARY

---

In this chapter, we introduced the basic concepts of stored program machines. Stored program machines offer more flexibility than finite state machines, at the cost of lower performance. The next chapter introduces the stored program machine that is the main topic of this book, the PIC18F242 microcontroller.

### REVIEW PROBLEMS

---

1. Write an assembly language program for the number sequencing computer that outputs the four digit sequence 0, 2, 5, 7 if LOC = 0, else output the sequence 1, 3, 6, 8. After a sequence is finished, loop back to program start. Convert your assembly language program to machine code starting at location 0.
2. Write the assembly language for the NSC machine code program seen in Table 2.8.
3. For the NSC, assume that the LOC input is tied to the least significant bit of the DOUT bus. For the program in Table 2.9, give the location executed and the DOUT value for the first 10 clock cycles.
4. Repeat problem #3, except change the instruction at location #1 to OUT 4.

**TABLE 2.8** NSC Machine Code Program

Memory Location	Machine Code
0	100000
1	010000
2	100001
3	010000
4	100010
5	010000
6	101001
7	000000

5. Assume the number definition is changed to  $1-X_1X_2X_3-Y_1Y_2Y_3-Z_1Z_2Z_3Z_4$ , with the local number as  $Y_1Y_2Y_3-Z_1Z_2Z_3Z_4$ . How many instructions are required for the NSC to implement this program?

**TABLE 2.9** NSC Assembly Language Program

Memory Location	Machine Code
0	OUT 2
1	OUT 5
2	JC 5
3	OUT 4
4	JC 0
5	OUT 9
6	JC 2
7	JC 5
8	OUT 4
9	JC 0

6. Modify the schematic of the NSC (Figure 2.7) to add support for a new instruction called `INC` that increments the current contents of the output register. Assign this new instruction the binary opcode “11”; the data field is unused. Hint: Try replacing the output register with an up counter.
7. Modify the schematic of the NSC (Figure 2.7) so that it can access a memory with 32 instructions (Hint: Begin by extending the memory to 32 locations, then trace all of the changes required in the various components—you may be surprised at the number of modifications caused by this seemingly minor extension).
8. Assume the NSC has a new instruction called `INC` (opcode = “11”) that increments the contents of the `OUT` register; the `INC` instruction data field is unused. Also assume that the `LOC` input is tied to the complement of the `DOUT[3]` bit ( $LOC = \sim DOUT3$ ). For the program in Table 2.10, how many clock cycles does it take to reach location 3?

**TABLE 2.10** NSC Assembly Language Program

Memory Location	Machine Code
0	OUT 0
1	INC
2	JC 1
3	JMP 3

9. What changes have to be made to the NSC (Figure 2.7) to accommodate a maximum of eight instructions instead of four?
10. Assume the number definition is changed to  $1-X_1X_2X_3-Y_1Y_2Y_3-Z_1Z_2Z_3Z_4$ , with the local number as  $Y_1Y_2Y_3-Z_1Z_2Z_3Z_4$ . Draw the new ASM chart required to implement this number sequence. How many states are required? If binary encoding is used for the states, how many DFFs are required?



*This page intentionally left blank*

# 3

## Introduction to the PIC18Fxx2

### In This Chapter

- Introduction to Microprocessors and Microcontrollers
- The PIC18Fxx2 Microcontroller
- Data Memory Organization and Data Transfer
- Basic Arithmetic and Control Instructions
- A PIC18 Assembly Language Program
- The Clock and Instruction Execution

This chapter introduces the PIC18Fxx2 instruction set architecture by exploring the PIC18's banked data memory structure and data transfer instructions. The use of MPLAB® for assembly and simulation of PIC18 programs is also discussed.

### 3.1 LEARNING OBJECTIVES

---

After reading this chapter, you will be able to:

- Describe the basic data and program memory architecture of the PIC18.
- Convert PIC18 instruction mnemonics to machine code, and vice versa.

- Describe the operation of the `movwf`, `movlb`, `addwf`, `subwf`, `incf`, `decf`, and `goto` instructions.
- Write PIC18 instructions to perform data transfer between memory locations in the same bank, or in different banks.
- Translate (manually compile) a simple C program into PIC18 assembly language.
- Compute the number of clock cycles and the amount of time required to execute simple instruction sequences for the PIC18.

## 3.2 INTRODUCTION TO MICROPROCESSORS AND MICROCONTROLLERS

---

In the previous chapter, a computer was defined as a digital system composed of control, input/output, and memory components whose operation is controlled by instructions stored in memory. The first computers were designed in the early 1940s and filled entire rooms, with total processing capability that was less than a modern digital watch. Early computers used *vacuum tubes* (grossly, a current amplifier within a glass tube) to implement logic, and a logic gate could take up an entire board. Transistors were invented by Bell Labs in 1947 [1], allowing an order of magnitude size reduction in logic implementation. However, transistors were packaged individually, and computers still required a large number of circuit boards to implement. In 1958, Jack Kilby, a researcher at Texas Instruments™, created the first *integrated circuit* [1], which is a silicon substrate upon which circuits with multiple transistors can be fabricated (the slang term *chip* is now commonly applied to integrated circuits). As integrated circuit fabrication techniques evolved, the size of integrated circuit transistors steadily decreased, allowing increasing numbers of transistors to be placed on the same silicon substrate. In 1971, Intel® developed a set of four integrated circuits that implemented a 4-bit computer [1] (the data paths were 4 bits, much like the number sequencing computer of Chapter 2, “The Stored Program”). One chip, the 4004, implemented the instruction decode and execution (the *central processing unit*, or CPU), while the other chips implemented the memory and input/output. The term *microprocessor* ( $\mu\text{P}$ ) was applied to this chipset, as it was a very small (micro) processing engine. The 4004 chip is generally regarded as the world’s first microprocessor. Integrated circuit technology has continually improved since the 4004, producing two distinct paths of microprocessor evolution. One evolution path has stressed high performance, using the increasing number of transistors to build larger internal data paths (up to 64 bits) and registers, advanced numerical processing, and support for very large memory spaces. These microprocessors are referred to as *general purpose microprocessors*, and expect

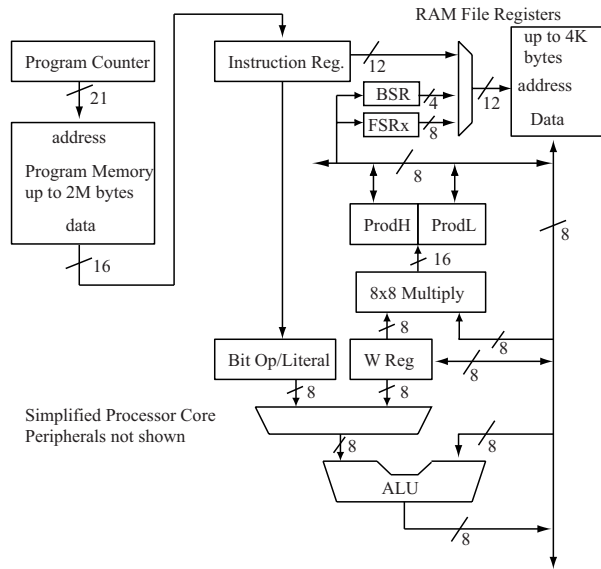
programs and data to be stored in memory external to the microprocessor. General-purpose microprocessors require external support chips, known as a *chipset*, that allow them to interface with memory and input/output devices. Examples of general-purpose microprocessors are the Intel Itanium<sup>®</sup>, Advanced Micro Devices Athlon<sup>®</sup>, and IBM PowerPC<sup>®</sup> families. The other microprocessor evolution path has stressed higher integration and lower cost, with the goal of producing a single chip solution to problems requiring a stored program machine approach. The term *microcontroller* ( $\mu C$ ) is generally applied to these devices. A microcontroller typically expects its program and data to be stored on-chip, with any logic required for external input/output devices also integrated into the same device. Thus, a microcontroller implements all of the components of a computer—control, memory, and input/output—in one chip. Microcontroller solutions are usually very cost sensitive, so applying exactly the right amount of processing power to a problem to minimize cost is important. As such, a large number of microcontroller families are available from 8-bit to 32-bit, with widely varying amounts of on-chip memory and different laundry lists of input/output interface options (whose number and variety grows each year). Microcontroller versions of general-purpose microprocessors have also been introduced over the years, so the distinction between a microcontroller and a microprocessor has become somewhat blurred, and in some cases, is an arbitrary labeling. In this book, the term *microcontroller* is applied to the PIC18Fxx2 device, but the term *microprocessor* is used any time a more general labeling is desired.

### 3.3 THE PIC18FXX2 MICROCONTROLLER

---

The PIC18Fxx2 microcontroller is the device used in this book to discuss microprocessor programming, architecture, and interfacing topics. Microchip Technology<sup>®</sup> makes the PIC18, an architecture variant within the PICmicro<sup>®</sup> microcontrollers family. There are several versions of the PIC18Fxx2 (hence, the “xx2” in the name); while they differ in the amount of on-chip memory and external I/O pins, the *Instruction Set Architecture* (ISA) is the same for all of them. A good place to begin discussing the PIC18Fxx2 is with a simplified block diagram of the processor core, shown in Figure 3.1 (a more complete architectural diagram is found in Appendix A).

The size of the internal data paths of the PIC18Fxx2 is 8 bits, so it is referred to as an 8-bit microcontroller. This means that the natural size for computations is 8 bits; arithmetic operations such as additions and subtractions operate on 8-bit data and can be specified with one instruction. Operations on data larger than 8 bits can be performed but require multiple instructions to accomplish them. The PIC18 instruction set defines 75 instructions, of which the majority require 16 bits (2 bytes) to encode. The term *instruction word* is used to refer to a 16-bit machine code. Four



**FIGURE 3.1** PIC18Fxx2 simplified architectural diagram.

instruction types require two instruction words (4 bytes) to encode. The instruction register in Figure 3.1 contains the instruction word that is currently being executed. The *arithmetic/logic unit* (ALU) in Figure 3.1 is the combinational logic that performs operations such as addition, subtraction, shift, bitwise AND/OR/XOR, and so forth. The left-hand ALU input is from a 2-to-1 mux that selects either the Working (W) register or data that is encoded in the current instruction word within the instruction register. The right-hand ALU input receives a value either from the W register or from a location in data memory. The result of the ALU operation is written either to the W register or a location in data memory.

Figure 3.1 shows that the PIC18Fxx2 has separate memories for program instructions and data. This type of arrangement is known as a Harvard architecture, as early electromechanical calculators such as the Harvard Mark I [1] read instructions from punched tape, with memory used only for storing data. Most microprocessors store programs and data in the same memory, which means that instructions can access memory that contains instructions as easily as locations that contain data. The majority of PIC18 instructions can only access data memory; a few special instructions are provided for accessing instruction memory. Program memory can be up to 2M bytes, or 1M instruction words (1 word = 2 bytes). Program memory is *nonvolatile*, meaning the memory contents are retained when power is removed. Some form of nonvolatile memory is required for any practical computer system, as this provides the instructions that are executed when power is

applied. The PIC18Fxx2 program memory is *flash programmable* (“F” for flash), meaning that it can be electrically erased and programmed. Other types of non-volatile memory are one-time-programmable (OTP), meaning they cannot be erased once programmed, and mask-programmed read-only-memory (ROM), which means the memory contents are determined at memory manufacture time and cannot be changed. Read operations on flash memory are fast, in the tens of nanoseconds, but write operations require much more time, a few milliseconds. Data memory can be up to 4K bytes, and is volatile, meaning memory contents are lost when power is removed. The term *random access memory* (RAM) is used for this data memory because reads and writes to this memory are fast (tens of nanoseconds), and write operations require the same amount of time as read operations. A small amount, 256 bytes, of nonvolatile data memory called EEPROM (electrically erasable programmable ROM) is included, which is in a separate memory from RAM data memory. Read and write times to EEPROM are similar to that of program memory. The sizes of program, data, and data EEPROM memories for the PIC18Fxx2 variants are listed in Table 3.1.

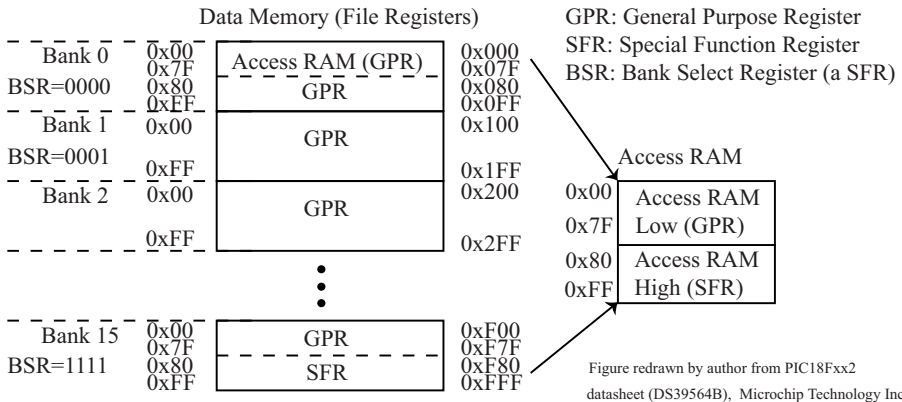
**TABLE 3.1** 18Fxx2 Memory Sizes

<b>Memory</b>	<b>18F242/18F442</b>	<b>18F252/18F452</b>
Program (bytes)	16 K	32 K
Program (instructions)	8192	16384
Data	768 (three banks of 256 locations each)	1536 (six banks of 256 locations each)
Data EEPROM	256	256

The 18F242/18F252 devices are available in 28-pin packages, while the 18F442/18F452 contain additional input/output ports and are available in 40- and 44-pin devices. Other members of the PIC18 family such as the PIC18F4550 have larger program and data memories.

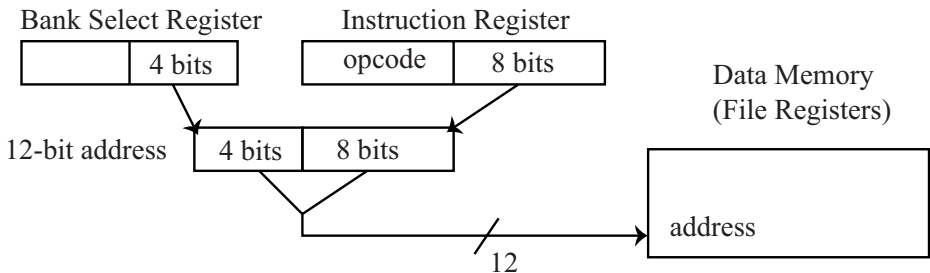
### **3.4 DATA MEMORY ORGANIZATION AND DATA TRANSFER**

The 4K-byte data memory of the PIC18xx2 is organized into 16 *banks* of 256 locations per bank as shown in Figure 3.2. Bank 0 contains locations 0x000 to 0x0FF, Bank 1 contains locations 0x100 to 0x1FF, and so forth.



**FIGURE 3.2** Data memory organization.<sup>1</sup>

A 12-bit address is required to specify a location in data memory. Most PIC instructions are encoded as one instruction word of 16 bits, of which only 8 bits specify a data memory location; the remaining 8 bits specify the instruction type. The 8 bits of the instruction word used for data addressing is the lower byte of the required 12-bit address, and specify the location within a bank. The remaining 4 bits of the 12-bit address determine the register bank, and are read from the Bank Select Register (BSR) as shown in Figure 3.3. The BSR value is 0 after processor reset. The BSR is an example of a *special function register* (SFR), of which there are many. SFRs are used as control and data registers for the on-chip peripherals such as the timers, asynchronous serial interface, analog-to-digital converter, and so forth. SFRs are contained in locations 0xF80 to 0xFFF within Bank 15. A *general-purpose register* (GPR) is any data memory location that is not a special function register.



**FIGURE 3.3** The Bank Select Register (BSR).

<sup>1</sup> Figure 3.2 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.'s prior written consent.

## The movf Instruction

One basic instruction class found within any microprocessor is the data transfer class. Data transfer instructions copy data between registers and locations in data memory, or write a value stored in the instruction word into data memory or registers. In most microprocessors, registers are a separate memory from data memory, and instructions contain addresses for both registers and memory. However, special function registers on the PIC18 are simply data memory locations with addresses of 0xF80-0xFFFF, making the terms *register* and *data memory location* interchangeable. Data memory locations are referred to as *file registers* in the Microchip PIC18 documentation, and this term is also used in this book.

The execution of data transfer between data memory locations is described as:

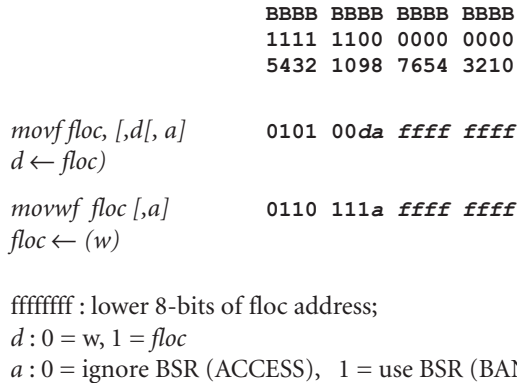
$dst \leftarrow (src)$  “copy the contents of source location *src* to destination location *dst*”

The  $\leftarrow$  symbol is a transfer symbol, and  $dst \leftarrow (src)$  is called *register transfer notation*, which is used to symbolically describe instruction execution. This is a common notation used to describe microprocessor instruction actions, and both register transfer notation and word descriptions are used in this book for discussing instruction execution.

Instructions that copy data between data memory locations are called *move* instructions in the PIC18. This is somewhat unfortunate, as the word *move* implies removing data from one location and placing it in another location, but *move* instructions do not affect the contents of the source memory location. Figure 3.4 shows the instruction mnemonic and machine code format for a commonly used data transfer instruction, *movf*, which copies the contents of a target data memory location to either the W register or back onto itself. The latter version is indeed useful, despite the fact that the target location contents are unchanged after execution, and its usage is discussed in the next chapter. The machine code format for all PIC18xx2 instructions can be found in Appendix A; many instruction types share common formats that differ only by the instruction opcode. Instruction word formats are discussed in detail if that particular format type is being covered for the first time.

In the *movf* instruction word, the *ffffff* bits specify the lower 8 bits of the source location, which is the *floc* operand of the instruction. The *d* bit specifies one of two destinations; “0” for the W register or “1” for *floc*. For improved code clarity, *w* or *f* is typically used for the destination register instead of “0” or “1”. The *a* bit is called the *data RAM access bit*. If the *a* bit is “1”, the 4 bits in the BSR are combined with the 8 bits of *floc* to provide the full 12-bit data memory address. If the *a* bit is “0”, the BSR is ignored, and the 8-bit *floc* value is an address in the *access bank*, a phantom bank composed of the first 128 locations of Bank 0 (0x000 to 0x07F) and the





**FIGURE 3.4** movf, movwf instructions, machine code format.

last 128 locations of Bank 15(0xF80-0xFFF), which are the special function registers. The *a* bit’s purpose is to provide fast access to the SFRs without having to modify the contents of the BSR. PIC18 programs access SFRs very frequently, so convenient access to SFRs is important. If the *a* bit did not exist, programs would need to frequently change the BSR value, increasing the number of instructions in PIC18 programs and slowing program execution. In the instruction operand, the symbols ACCESS or BANKED can be used instead of “0” or “1” to improve code clarity.

In this book, the access bit value *is not written* in instruction mnemonics. Instead, the access bit value is assumed using the following rules:

- If *floc* is in the range 0x00-0x7F or 0xF80-0xFFFF (the special function registers), then a=0 (ACCESS) is assumed.
- If *floc* in the range 0x080-0xF7F, then a=1 (BANKED) is assumed.

These are the same assumptions used by the MPLAB assembler, a program that converts PIC assembly language programs to machine code. Listing 3.1 shows four movf instructions and the machine code produced using these assumptions.

**LISTING 3.1** Machine code for movf instructions using access bit assumptions.

machine code	mnemonic	comment
0x5070	movf 0x070,w	;w ← (0x070), ACCESS (a=0)
0x5170	movf 0x170,w	;w ← (0x70), BANKED (a=1)
0x5170	movf 0x270,w	;w ← (0x70), BANKED (a=1)
0x5090	movf 0xF90,w	;w ← (0xF90), ACCESS (a=0)

**The mov1b Instruction**

Note that in Listing 3.1 the instructions `movf 0x170,w` and `movf 0x270,f` have the same machine code, despite the fact that the operands for the instruction mnemonics are different. This is because only the lower bytes of values 0x170 and 0x270 are used in the machine code, which have the same value of 0x70. For the `movf 0x170,w` instruction to operate as intended, the BSR value must be 0x1, selecting bank 1. Correspondingly, the BSR value must be 0x2 if the `movf 0x270,f` instruction is to read location 0x270. The `mov1b` instruction is used to modify the contents of the BSR. Figure 3.5 shows the instruction mnemonic and machine code format for the `mov1b` instruction.

	<b>BBBB BBBB BBBB BBBB</b>
	<b>1111 1100 0000 0000</b>
	<b>5432 1098 7654 3210</b>
<code>mov1b k</code>	<b>0000 0001 0000 <i>kkkk</i></b>
<code>BSR[3:0] ← k</code>	<b><i>k</i> : 4-bit literal</b>

**FIGURE 3.5** `mov1b` instruction machine code format.

The 4-bit *k* value in the `mov1b` instruction word is called a *literal*, which is a constant whose value is transferred to the BSR when the instruction is executed. A 4-bit literal allows the values 0 to 15 to be specified as BSR values. The instruction sequence in Listing 3.2 sets the BSR to 2, and then copies the contents of location 0x270 to the W register.

**LISTING 3.2** Using the BSR register.

<b>machine code</b>	<b>mnemonic</b>	<b>comment</b>
0x0102	<code>mov1b 2</code>	<code>;BSR ← 2</code>
0x5170	<code>movf 0x270,w</code>	<code>;w ← (0x270), BANKED</code>

If the `mov1b` instruction is not included, the current value of the BSR has to be known to determine what file register location is modified.

## Addressing Modes

The method by which the location of an instruction operand is specified is called the *addressing mode*. The `movf` instruction uses the *direct* addressing mode, so named because the address of the instruction operand is specified *directly* within the instruction machine code. Direct addressing implies a memory access to the location specified by the direct address, in order to obtain the operand for the instruction. The `movlb` instruction uses the *immediate* addressing mode, as the operand value is encoded *immediately* within the instruction word. Immediate addressing requires no further memory accesses for the operand value after the instruction word has been fetched.

### The `movwf`, `movff` Instructions

The `movwf` instruction copies the W register contents to a file register destination. The instruction mnemonic and machine code is given in Figure 3.4. The code fragment in Listing 3.3 copies the contents of location 0x1A0 to location 0x23F.

**LISTING 3.3** Copy location 0x1A0 to location 0x23F.

---

machine code	mnemonic	comment
0x0101	<code>movlb 1</code>	;BSR ← 1
0x51A0	<code>movf 0x1A0,w</code>	;w ← (0x1A0)
0x0102	<code>movlb 2</code>	;BSR ← 1
0x6F3F	<code>movwf 0x23F</code>	;0x23F ← (w)

The same functionality is accomplished by the single instruction in Listing 3.4, which uses the `movff` instruction.

**LISTING 3.4** Using the `movff` instruction.

---

machine code	mnemonic	comment
0xC1A0	<code>movff 0x1A0,0x23f</code>	;0x23f ← (0x1A0)
0xF23F		

The `movff` instruction requires two instruction words; the first word encodes the 12-bit address of the source, and the second word the 12-bit address of the destination. The BSR is not used by the `movff` instruction, as the complete addresses of

both operands are specified in the instruction words. Figure 3.6 gives the machine code format of the `movff` instruction.

	BBBB	BBBB	BBBB	BBBB
	1111	1100	0000	0000
	5432	1098	7654	3210
<code>movff <i>fsrc</i>, <i>fdst</i></code>	1100	<i>ffff</i>	<i>ffff</i>	<i>ffff</i> ( <i>fsrc</i> )
<code><i>fdst</i> ← (<i>fsrc</i>)</code>	1111	<i>ffff</i>	<i>ffff</i>	<i>ffff</i> ( <i>fdst</i> )

**FIGURE 3.6** `movff` instruction machine code format.

**Sample Question:** Using the assumptions for the access bit, how do the following three code segments differ in terms of the memory location that is accessed?

- (a) `movlb 3`  
`movf 0x345, w`
- (b) `movlb 3`  
`movf 0x245, w`
- (c) `movlb 3`  
`movf 0x045, w`

*Answer:* In all three code segments, the BSR is set to 3. In code segment (a), the address 0x345 is not in the access bank, so we assume BANKED ( $a = 1$ ) and location 0x345 is accessed as the BSR contains 0x3 and the instruction word contains 0x45. In code segment (b), the address 0x245 is not in the access bank, so we assume BANKED ( $a = 1$ ) and location 0x345 is accessed, as the BSR contains 0x3 and the instruction word contains 0x45. Location 0x245 is not accessed, as the “2” in the address contained within the instruction is only a hint to the assembler (and to us!) as to the access bit setting. The BSR register must have a value of 0x2 to access location 0x245. In code segment (c), the address 0x045 is in the access bank, so we assume ACCESS ( $a = 0$ ), causing the BSR to be ignored and location 0x045 to be accessed.

### 3.5 BASIC ARITHMETIC AND CONTROL INSTRUCTIONS

Two other classes of microprocessor instructions are arithmetic and control. All microprocessors have addition and subtraction instructions, and increment (+1)

and decrement ( $-1$ ) instructions. Some microprocessors also have instructions for multiplication and division; these can be implemented using addition, subtraction, and shifts if dedicated multiplication and division instructions are not present. The general form of all two-operand arithmetic operations for the PIC18 is:

$$\text{destination } floc \text{ or } w \leftarrow (w) \text{ op } (floc) \quad (\text{op is the operation performed})$$

Like the `movf` instruction, the destination is either  $W$  or  $floc$ . Observe that one of the source operands must be overwritten, but the user can choose which source operand to overwrite. Figure 3.7 gives the machine code format for the `addwf` (add  $W$  to file register) and `subwf` (subtract  $W$  from file register) instructions.

	BBBB	BBBB	BBBB	BBBB
	1111	1100	0000	0000
	5432	1098	7654	3210
<code>addwf floc, [d], a]</code>	0010	01da	ffff	ffff
$d \leftarrow (floc) + (w)$				
<code>subwf floc [d],a]</code>	0101	11da	ffff	ffff
$d \leftarrow (floc) - (w)$				

*ffffff*: lower 8-bits of  $floc$  address;  
*d*: 0 =  $w$ , 1 =  $floc$   
*a*: 0 = ignore BSR (ACCESS), 1 = use BSR (BANKED)

**FIGURE 3.7** `addwf,subwf` instructions, machine code format.

Increment (`incf`) and decrement (`decf`) instructions are one-operand arithmetic instructions, whose general form is:

$$\text{destination } floc \text{ or } w \leftarrow \text{op } (floc) \quad (\text{op is operation performed})$$

In this case, the user has a choice of overwriting the source operand if the destination is  $floc$ , or preserving it if the destination is  $W$ . The machine code and operand format of the `incf/decf` instructions are the same as the `movf` instruction, and only differ by the opcode value. To accomplish the operation  $+2$ , or  $+3$ , or  $+k$ , where  $k$  is an 8-bit value, examine the code sequence in Listing 3.5, which adds the value 5 to file register `0x040`.

**LISTING 3.5** Using the `movlw` instruction.

machine code	mnemonic	comment
0x0E05	<code>movlw 5</code>	;w ← 5
0x2640	<code>addwf 0x040,f</code>	;0x040 ← (w) + (0x040)

The instruction `movlw k` (move literal to W) is used to load a value of 5 into W; the `addwf` instruction then adds W to the file register 0x040. The machine code format of `movlw` differs from the previously discussed `movlb` instruction in the literal size, 8 bits versus 4 bits, and opcode value. The instruction `addlw k` (add literal to W) adds an 8-bit literal to the W register, which is useful in a succession of operations where the W register is being used to accumulate a value. The instruction `sublw k` (subtract W from literal) subtracts the W register from a literal value (useful for an unsigned comparison of a variable to a literal, discussed in Chapter 4, “Unsigned 8-Bit Arithmetic, Logical, Conditional Operations”). To subtract a literal from W, specify a negative value for *k* in the `addlw k` instruction; the binary representation of negative numbers is discussed in Chapter 5, “Extended Precision and Signed Operations.”

Control instructions affect the program counter contents, with the simplest form being a `goto`, an unconditional transfer of control. Figure 3.8 shows the machine code format for the `goto` instruction.

	<b>BBBB</b>	<b>BBBB</b>	<b>BBBB</b>	<b>BBBB</b>
	1111	1100	0000	0000
	5432	1098	7654	3210
<code>goto k</code>	1110	1111	<i>k</i> 7 <i>kkk</i>	<i>kkkk</i> 0
<code>PC[20:1] ← k</code>	1111	<i>k</i> 1 <i>9kkk</i>	<i>kkkk</i>	<i>kkkk</i>

**FIGURE 3.8** `goto` instruction machine code format.

The program memory location that control is transferred to is known as the *target address*, which is a 20-bit value as it specifies one of 1M possible instruction word locations. This 20-bit value specifies a word address, not a byte address, and is loaded into the upper 20 bits of the Program Counter, which contains a 21-bit byte address. The `goto` instruction requires two instruction words, with the second instruction word containing bits 19 through 8 of the target address, and the first instruction word containing bits 7 through 0 of the target address. Instruction words are located in program memory on even byte boundaries, beginning at location

0x0000. It is illegal to place an instruction word starting at an odd byte location, such as 0x0003. To determine the value of the 20-bit literal encoded in a `goto` instruction, take the program memory byte address and shift to the right by one position (divide by 2) to get the program memory word address.

**Sample Question: What is the machine code for the instruction `goto 0x0010A`?**

*Answer:* The target address specifies a byte location in program memory, so the word address is 0x0010A shifted to the right by 1 (divided by 2), or:

$$0x0010A \gg 1 = 0x00085 = 0b\ 0000\ 0000\ 0000\ 1000\ 0101 \quad (\text{target word address})$$

The first instruction word format is:  $0b\ 1110\ 1111\ k_7kkk\ kkkk_0$ , where the  $k$  bits are the lower 8 bits of the target word address. Hence, the first instruction word of `goto 0x0010A` is:

$$0b\ 1110\ 1111\ 1000\ 0101 = 0xEF85 \quad (\text{first instruction word})$$

The second instruction word format is:  $0b\ 1111\ k_{19}kkk\ kkkk\ kkkk_8$ , where the  $k$  bits are the upper 12 bits of the target word address. Hence, the second instruction word is:

$$0b\ 1111\ 0000\ 0000\ 0000 = 0xF000 \quad (\text{second instruction word})$$

## 3.6 A PIC18 ASSEMBLY LANGUAGE PROGRAM

---

At this point, we have enough instructions to write a simple PIC18 assembly language program. In this book, programs are first written in *C*, and then translated (*compiled*) to assembly language. This is done to improve the clarity of the program's functionality, as assembly language can be obtuse, especially for readers new to assembly language programming. If you are new to the *C* language, do not worry, as *C* language statements are introduced gradually and fully explained. A previous exposure to any modern programming language is all that is necessary to understand the *C* program examples used in this book. Example *C* programs only use those *C* language statements necessary to demonstrate PIC18Fxx2 capabilities, and do not attempt to cover the entire *C* language. A *C* program that uses the data transfer and arithmetic operations discussed so far is shown in Listing 3.6. Line numbers have been added for clarity, but would not be part of the actual *C* program source code.

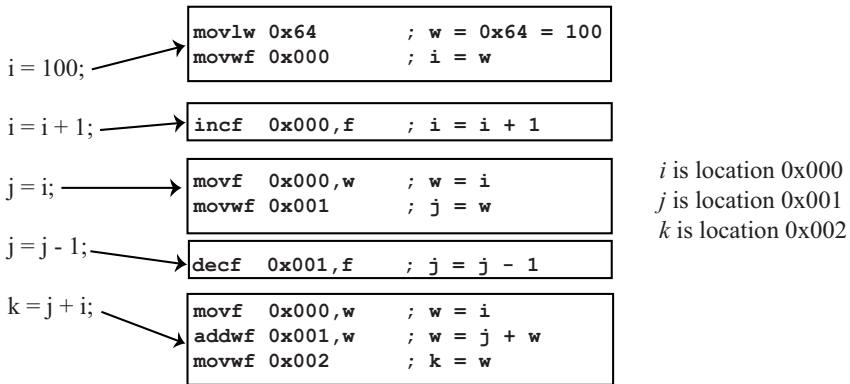
**LISTING 3.6** A “simple” C program.

```
(1)      #define avalue 100
(2)      unsigned char i,j,k;
(3)      main(void) {
(4)          i = avalue; // avalue = 100
(5)          i = i + 1;  // i++, i = 101
(6)          j = i;      // j is 101
(7)          j = j - 1;  // j--, j is 100
(8)          k = j + i;  // k = 201
(9)      }
```

The C language is case sensitive, with all reserved key words, such as `main` or `unsigned`, being lowercase. Comments begin with two `//` characters, and can start anywhere on a line. Simple C statements are terminated by a semicolon (“;”). Compound statements, which are composed of multiple simple C statements, are bracketed by `{}`. Line 1 contains a `define` statement, which is a method for assigning a symbolic name to a value. Use of `defines` for constant values usually improves code clarity. Line 2 defines three variables of type `unsigned char`, which means each variable is 8 bits, or 1 byte, and represents unsigned data. The `unsigned` modifier tag combined with the 8-bit data size gives a value range of 0 to 255 for each variable. Chapter 5 discusses the difference between `unsigned` and `signed` data types, and the effect this has on arithmetic operations. The type name `char` chosen by the original inventors of the C language is somewhat unfortunate, as this implies these variables contain character (ASCII) data, when in fact, they are simply 8-bit values. Line 3 defines the entry point for the C program, which must be named `main`. The `(void)` after the `main` label indicates that `main` receives no parameters, which will always be the case for C programs in this book. This can also be written as simply `main()`. The body of the `main()` code is a compound C statement, enclosed by `{}`. Line 4 assigns the constant value 100 to the variable `i`. Line 5 increments `i` by 1; `i` contains the value 101 after execution of this statement. The C statement `i++`, where `++` is the C increment operator, could be used instead of `i=i+1`. Line 6 copies the value of `i` to `j`. Line 7 decrements `j`, so `j` contains the value 100 after execution of this statement. The statement `j--` could be used instead of `j=j-1`.

The first step in *compiling* the program in Listing 3.6 to PIC18 assembly language is to choose locations for the variables `i`, `j`, and `k`. This can be any general-purpose register (a file register location not previously assigned to a special function register). For simplicity, we will use location 0x000 for `i`, 0x001 for `j`, and 0x002 for `k`. Figure 3.9 shows the program of Listing 3.6 translated to PIC18 assembly language.





**FIGURE 3.9** The “simple” C program compiled to PIC18 assembly language.

The compilation is straightforward when only one line is considered at a time. Optimizing C compilers (and expert assembly language programmers) consider multiple C language statements at a time during compilation in an effort to reduce the total number of instructions, and it may be difficult to correlate the final assembly code with the original C language statements. This book does not expect you to become an expert assembly language programmer; this only occurs after a considerable amount of time is spent crafting assembly language programs. Instead, this book strives for clarity and understanding, and will always perform C-to-PIC18 assembly language translation in the most straightforward manner possible. Of the C language statements in Figure 3.9, the statement  $k = j + i$  is the most difficult, and requires three PIC18 instructions to implement. In the resulting three instruction sequence, observe that the destination of `addwf 0x001, w` is W so that the value of 0x001 (*j*) is left undisturbed. The `movwf 0x002` instruction copies the result of the addition into the *k* variable location (0x002). This three-instruction sequence could be replaced by the instructions in Listing 3.7.

**LISTING 3.7** Alternate implementation of  $k = i + j$ .

```

movf 0x001, w    ; w ← (j)
addwf 0x000, w   ; w ← (i) + (w)
movwf 0x002     ; k ← (w)

```

This works because addition is a commutative operation, and  $i + j$  or  $j + i$  gives the same result. However, the C language statement  $k = j - i$  could only be implemented as shown in Listing 3.8 because  $j - i$  is not equal to  $i - j$ .

**LISTING 3.8** Implementation of  $k = j - i$ .

---

```

movf   0x000,w      ;w ← (i)
subwf  0x001,w      ;w ← (j) - (w)
movwf  0x002        ;k ← (w)

```

The PIC18 assembly language of Figure 3.9 is somewhat obtuse because memory location values (0x000, 0x001, 0x002) are used instead of the variable names  $i$ ,  $j$ ,  $k$ . Also, there is still the problem of translating the PIC instruction mnemonics to machine code, a process that is interesting the first time, boring the second time, and painful thereafter. A program called an *assembler* automatically converts instruction mnemonics to machine code. Microchip Technology provides an Integrated Design Environment (IDE) called MPLAB<sup>®</sup>, which contains an assembler and simulator for most Microchip microprocessors. Listing 3.9 gives the assembly language of Figure 3.9 written in a more readable form, and in a format compatible with the MPLAB assembler, MPASM<sup>™</sup> (the line numbers are not part of the source file).

**LISTING 3.9** MPLAB-compatible assembly source code for “simple” C example.

---

```

(1)   INCLUDE "p18f242.inc"

(2)           CBLOCK 0x000
(3)           i,j,k
(4)           ENDC

(5)   avalue equ D'100'

(6)           org    0
(7)           goto  main      ; reserve 0x0-0x1FF for interrupts

(8)           org 0x0200
(9)   main
(10)          ; i=avalue;
(11)          movlw  avalue ; w ← 100
(12)          movwf  i      ; i ← (w);

(13)          ; i = i + 1;
(14)          incf   i,f     ; i ← (i) + 1

```

```

(15)      ; j = i
(16)      movf   i,w      ; w ← (i)
(17)      movwf  j        ; j ← (w)

(18)      ; j = j - 1;
(19)      decf   j,f      ; j ← (j) - 1

(20)      ; k = j + i
(21)      movf   i,w      ; w ← (i)
(22)      addwf  j,w      ; w ← (w) + (j)
(23)      movwf  k        ; k ← (w)

(24)      here
(25)      goto   here     ; loop forever
(26)      end

```

The `INCLUDE` statement in line 1 is called an *assembler directive*, which is an instruction to the assembler and not a PIC18 assembly language statement. Lines 1, 2–4, 5, 6, 8, and 26 are all assembler directives. The `INCLUDE` statement causes the source file `p18f242.inc` to be included during assembly. When assembling a PIC18 program, the assembler must be told the target device, in this case the 18F242, and an appropriate definitions file, `p18f242.inc`. The target device is set from within the MPLAB program. The `p18f242.inc` file defines symbolic names for all SFRs and named bits within SFRs. For example, instead of using `0xFE0`, the symbol `BSR` can be used in a PIC18 instruction to refer to the bank select register.

The constant block (`CBLOCK`) assembler directive assigns constant values to a group of labels. Each label is assigned in order, counting up, so `i` is assigned the value `0x000`, `j` the value `0x001`, and `k` the value `0x002`. This allows the `i`, `j`, `k` variables to be referenced by name within PIC18 instructions instead of using absolute memory locations. This improves code clarity and makes it easy to change data memory assignments. To change the location of the `i`, `j`, `k` variables, simply change the `CBLOCK` statement; no editing of the instruction operands within the source code is required. A `CBLOCK` is terminated with an `ENDC` assembler directive.

The `equ` assembler directive assigns the value 100 to the label `ava1ue`. The default radix of numbers in MPLAB is base 16, so `D'100'` is the method for specifying 100 in decimal. The `org` (origin) assembler directive specifies the starting location in program memory for code that follows this statement. At power up, the program counter is reset to `0x0`. Thus, the first PIC18 instruction is fetched from location

0x0, which is called the *reset vector*. Typically, a `goto` statement that jumps to the program entry point is placed at the reset vector instead of locating the `main()` code itself at location 0x0. This is because there are *interrupt vectors* (discussed in Chapter 10) at locations 0x0008 and 0x0018. As with the reset vector, `goto` statements are typically placed at the interrupt vectors, so the `main()` code is then placed somewhere below these locations. Line 8 uses the `org 0x0200` directive, with line 9 containing the label `main`, which is the target of the `goto main` instruction at location 0x0. Thus, `goto main` is assembled as `goto 0x0200`. Placing the `main()` code at location 0x200 is somewhat arbitrary, as any location from 0x001C onward would do equally well (locations 0x0018, 0x001A will be used by the `goto` instruction placed at interrupt vector 0x0018). Labels such as `main` must start in column 1 and are case sensitive. Instruction mnemonics and assembler directives must begin after column 1 and are case insensitive. Note that the instructions in lines 10–23 are the same instructions from Figure 3.9, except that label names for `i`, `j`, and `k` are used. Lines 24 and 25 contain the infinite loop `goto here`, where the target address `here` is the location of the `goto` statement. A microcontroller program never really ends; it must always be doing something, as there is no place for the program to go when it finishes! When a program exits on a personal computer, control is returned to the operating system, which is in an infinite loop waiting for input from the keyboard, mouse, or some other input device. A microcontroller program is also typically an infinite loop that is waiting on input from some external device such as a car engine, sensor array, and so forth. In this simple example, the program execution is trapped when it falls into the `goto here` infinite loop. Another method to halt program execution is to stop the processor clock; this is discussed in Chapter 8.

Listing 3.10 gives the machine code listing produced by the MPLAB assembler for the assembly language program of Listing 3.9. The address column gives the program memory location in hex, and the machine code column the assembled code for the mnemonic to the right. The `ACCESS` label indicates the access bit value is a “0”, causing the BSR to be ignored. The assembler sets the access bit to a “0” because the memory locations for `i`, `j`, `k` are in the access bank range, and the access bit value is not explicitly specified for any instruction operands.

**LISTING 3.10** Machine code listing for simple C program.

---

address	machine code	mnemonic
0000	EF00	GOTO 0x0x200
0002	F001	NOP
0200	0E64	MOVLW 0x64
0202	6E00	MOVWF 0, ACCESS
0204	2A00	INCF 0, F, ACCESS

0206	5000	MOVF 0, W, ACCESS
0208	6E01	MOVWF 0x1, ACCESS
020A	0601	DECF 0x1, F, ACCESS
020C	5000	MOVF 0, W, ACCESS
020E	2401	ADDWF 0x1, W, ACCESS
0210	6E02	MOVWF 0x2, ACCESS
0212	EF09	GOTO 0x212
0214	F001	NOP
0216	FFFF	NOP

Assume the CBLOCK of Listing 3.9 is changed as shown in Listing 3.11.

---

**LISTING 3.11** Alternate CBLOCK location.

```
(2)    CBLOCK 0x100
(3)    i,j,k
(4)    ENDC
```

This assigns the labels *i*, *j*, *k* to values 0x100, 0x101, 0x102, respectively. For these locations to actually be used, the BSR must be set to the value 0x1 by modifying the `main()` code as seen in Listing 3.12.

---

**LISTING 3.12** Setting the BSR.

```
(8)    org 0x0200
(9)    main
(10)   movlb 0x1 ; set BSR = 1

(11)   ; i=avalue;
(12)   movlw avalue
```

etc..other program lines unchanged.

The `movlb 0x1` instruction is inserted at the beginning of the `main()` code. The access bits of any instructions that refer to locations 0x0100-0x102 must also be set to “1”, but this is done automatically by the MPLAB assembler because these

locations are not in the *access* bank. The machine code listing for this modified code as produced by MPLAB is given in Listing 3.13.

**LISTING 3.13** Machine code listing for simple C program, variables in bank 1.

---

address	machine code	mnemonic
0000	EF00	GOTO 0x0x200
0002	F001	
0200	0101	MOVLB 0x1
0202	0E64	MOVLW 0x64
0204	6F00	MOVWF 0, BANKED
0206	2B00	INCF 0, F, BANKED
0208	5100	MOVF 0, W, BANKED
020A	6F01	MOVWF 0x1, BANKED
020C	0701	DECF 0x1, F, BANKED
020E	5100	MOVF 0, W, BANKED
0210	2501	ADDWF 0x1, W, BANKED
0212	6F02	MOVWF 0x2, BANKED
0214	EF0A	GOTO 0x214
0216	F001	NOP
0218	FFFF	NOP

Notice that the access bits of instructions that reference locations 0x0100 – 0x0102 now have the word **BANKED** by their operands, indicating the access bit is set to “1”, which is the desired result.

### Using WREG in an Instruction

In the MPLAB assembler, the *W* register is specified as **WREG** when used as a file register in an instruction. For example, the instruction `incf WREG,w` increments the contents of the *W* register by 1, while the instruction `addwf WREG,w` adds the contents of the *W* register to itself.

### The nop Instruction

The second word of the `goto` instructions in Listings 3.10 and 3.13 displays as a `nop` instruction, which stands for “NO oPeration”. A `nop` simply causes the instruction word to be fetched, and the PC to be incremented to the next instruction word. One machine code encoding of a `nop` sets bits 16 to 12 (upper 4 bits) as “1”, and the

remaining bits as don't cares. All second instruction words are encoded such that bits 16 to 12 are "1". The instruction in location 218 of Listing 3.13 with machine code 0xFFFF is also a nop. The value 0xFFFF is the blank or erased state of program memory. An alternate coding for a nop is 0x0000 (all bits 0). The machine codes 0xFFFF and 0x0000 were chosen for the nop instruction because any erased location contains 0xFFFF, and also because any memory location in the 2M address range that is not physically implemented returns a 0x0000 when read (the PIC18F242 physical memory is 0x000000-0x003FFF, or 16 KBytes). In this way, if a program error causes a jump to the portion of memory that is erased, continuous nop instructions (0xFFFF) are fetched until the program counter exceeds physical memory. Then, 0x0000 values (nop instructions) are read until the PC wraps back to the reset location of 0x0, simulating a device reset. An internal register of the PIC can be checked by the startup code to determine if a physical reset actually occurred; if not, an error indicator could be displayed indicating that an anomalous reset condition occurred.

**Sample Question:** Write a PIC18 assembly language fragment that implements the C statement " $k = i + j + 20$ ;" where  $k$ ,  $i$ ,  $j$  are all char variables.

**Answer:** One solution is:

```
movf   i,w      ; w = i
addwf  j,w      ; w = w + j;
addlw  20       ; w = w + 20
movwf  k        ; k = w
```

Observe that a single C statement may require several PIC18 assembly language statements, as several operations can be written in one C statement. Translating the C statement to PIC18 statements requires that you decompose the C statement into steps that the PIC18 can accomplish.

**Sample Question:** A neophyte assembly language programmer translated the C statement:  $k = j + 1$  to the two statements `incf j, f ; movff j, k`. What is wrong with this?

**Answer:** The statement `incf j, f` modifies the variable  $j$ . The C statement  $k = j + 1$  only modifies  $k$ ; the variable  $j$  is not modified. A correct solution is `incf j, w ; movwf k`. The statement `incf j, w` places  $j+1$  in the W register and leaves the memory location  $j$  unmodified.

### 3.7 THE CLOCK AND INSTRUCTION EXECUTION

---

The clock signal that controls instruction execution on the PIC18Fxx2 can have a maximum frequency of 40 MHz for the devices available during the writing of this book. Methods for generating this clock signal and setting its frequency are discussed in Chapter 8. The symbol  $FOSC$  refers to the clock signal's frequency, and  $TOSC$  refers to its period. A finite state machine within the PIC18Fxx2 controls the fetching, decoding, and execution of instructions. The Program Counter (PC) register contains the address of the instruction that is fetched from program memory. The PC is 21 bits, and thus can access 2 Mbytes or 1M instruction words. Most instructions require one instruction cycle to execute, with one instruction cycle equal to four clock cycles (four  $Tosc$  periods). These four clock cycles are used in different ways depending on the instruction. For the `addwf` instruction, clock cycle 1 decodes the instruction, clock cycle 2 reads the register file, clock cycle 3 performs the addition, and clock cycle 4 writes the result to the destination. A 40 MHz clock has a 25 ns period, so one instruction cycle takes four clocks, a time of  $4 * 25 \text{ ns} = 100 \text{ ns}$ . This means that a PIC18Fxx2 with a 40 MHz clock can execute instructions at a rate of approximately 10 million instructions per second (MIPs). The actual instruction execution rate will be somewhat lower than this, as any instruction that causes the program counter to change value, such as a `goto` instruction, requires two instruction cycles (eight clocks) to execute. The instruction table contained in Appendix A, "PIC18Fxx2 Architecture, Instruction Set, Register Summary," gives the number of instruction cycles required for each instruction. Another instruction that takes two instruction cycles is `movff`, as it requires two instruction words and thus one instruction cycle is needed for each instruction word.

**Sample Question: How many clock cycles are required to execute through location 0x0210 of Listing 3.10? Assuming  $FOSC = 40 \text{ MHz}$ , how long do these instructions take to execute?**

*Answer:* There are 10 instructions total. Only the `goto` instruction requires two instruction cycles, so total clock cycles =  $9 * 4 + 1 * 8 = 36 + 8 = 44$ . Total execution time =  $1/(40 \text{ MHz}) * 44 = 1/(40e6) * 44 = 1.1e-6 \text{ s} = 1.1 \mu\text{s}$

### SUMMARY

---

In this chapter, we introduced the basic program and memory architecture of the PIC18xx2 microcontroller, and discussed a few PIC18 instructions from the data transfer, arithmetic, and control classes. A simple C program that operates on 8-bit unsigned data was converted to PIC18 assembly language. This prepares you for the



next chapter, in which additional arithmetic, shift, and logical operations on 8-bit data are covered, as well as conditional execution and loop structures.

## REVIEW PROBLEMS

---

For the following problems, if the access bit is not specified on an instruction operand, use the assumptions specified in this chapter.

1. Convert the instruction `addwf 0x030, f` to machine code.
2. Convert the instruction `addwf 0x230, f` to machine code.
3. Convert the instruction `goto 0x043E` to machine code.
4. Convert the machine code `0x010A` to a PIC18 instruction.
5. Convert the machine code `0xC2A5 0xF100` to a PIC18 instruction.
6. Convert the instruction word `0x0E09` to a PIC18 instruction.
7. What memory location is modified by the instruction `addwf 0x230, f, BANKED` (warning: this is a trick question)?
8. Write PIC18 assembly that will accomplish  $k = i - j$  where  $i$  is location `0x100`,  $j$  is location `0x240`, and  $k$  is location `0x030`. The variables  $i$ ,  $j$ ,  $k$  are all byte variables.
9. Given a 16 MHz clock, how long do the three instructions starting from location `0x0210` (`addwf`, `movwf`, `goto`) of Listing 3.13 take to execute? Give the answer in microseconds.
10. Write a PIC18 instruction sequence that accomplishes  $k = i + j + 5$ , where  $i$ ,  $j$ , and  $k$  are in the same locations as Listing 3.9.
11. Write a PIC18 instruction sequence that copies file register locations `0x100` through `0x103` to locations `0x200` through `0x203`.
12. How many `NOP` instructions are executed in one second assuming a 25 MHz `FOSC`?
13. What is the value of `W` after the instruction `subwf WREG, w` is executed?

For the remaining problems, give the affected registers after execution of each instruction, and assume the following file register contents at the beginning of EACH problem:

`W = 0x3D`, `BSR = 0x0` (see Table 3.2 for other memory contents)

**TABLE 3.2** Memory Contents

<b>Location</b>	<b>Contents</b>
0x05A	0x3B
0x05B	0xA2
0x05C	0xF4
0x05D	0x7D
0x250	0xF9
0x251	0xB2

14. Instruction: `subwf 0x5C,f`
15. Instruction: `movlw 0x5C`
16. Instruction: `addwf 0x5C,f`
17. Instruction: `addwf 0x5A,f`
18. Instruction: `incf 0x5B,w`
19. Write an instruction sequence that adds the contents of location 0x250 to location 0x251, and stores the result in 0x05A.
20. Instruction: `decf 0x5C,f`

*This page intentionally left blank*

# 4

## Unsigned 8-Bit Arithmetic, Logical, Conditional Operations

### In This Chapter

- Bitwise Logical Operations, Bit Operations
- The STATUS Register
- Unsigned Conditional Tests
- Looping
- Shifts and Rotates

This chapter examines additional features of the PIC18Fxx2 instruction set architecture in the context of unsigned 8-bit arithmetic, bitwise logical, and shift operations. These operations are used to implement equality, inequality, and comparison tests for conditional code execution and loop control.

### 4.1 LEARNING OBJECTIVES

---

After reading this chapter, you will be able to:

- Describe the operation of the bit manipulation instructions of the PIC18 instruction set.

- Translate C language statements that perform 8-bit addition, subtraction, bit-wise logical, and shift operations into PIC18 instruction sequences.
- Translate C language statements that perform 8-bit zero, nonzero, equality, inequality, and unsigned comparison operations into PIC18 instruction sequences.
- Translate C conditional statements and loop structures such as `do-while`, `while-do`, and `for{}` into PIC18 instruction sequences.

## 4.2 BITWISE LOGICAL OPERATIONS, BIT OPERATIONS

Table 4.1 lists the C language arithmetic and logical operators discussed in this book. As seen in the previous chapter, the arithmetic/logic unit (ALU) implements these operations in the processor data path. The previous chapter covered some of the arithmetic capabilities of the ALU such as addition, subtraction, increment, and decrement. The bitwise logical operators `&` (AND), `|` (OR), `^` (XOR), and `~` (complement) comprise the logical operations performed by the ALU.

**TABLE 4.1** C Language Arithmetic and Logical Operators

Operator	Description
<code>+</code> , <code>-</code>	( <code>+</code> ) addition, ( <code>-</code> ) subtraction
<code>++</code> , <code>--</code>	( <code>++</code> ) increment, ( <code>--</code> ) decrement
<code>*</code> , <code>/</code>	( <code>*</code> ) multiplication, ( <code>/</code> ) division
<code>&gt;&gt;</code> , <code>&lt;&lt;</code>	right shift ( <code>&gt;&gt;</code> ), left shift ( <code>&lt;&lt;</code> )
<code>&amp;</code> , <code> </code> , <code>^</code>	bitwise AND ( <code>&amp;</code> ), OR ( <code> </code> ), XOR ( <code>^</code> )
<code>~</code>	bitwise complement

Figure 4.1 shows the instruction format and machine code formats for the bitwise logical operations implemented by the PIC18. The `andwf` (AND), `iorwf` (OR), and `xorwf` (XOR) have the standard PIC18 two-operand instruction format where the source operands are *W* and *floc*, and the destination is either *W* or *floc*. The literal forms of these instructions are `and1w` (AND), `ior1w` (OR), and `xor1w` (XOR) where the source operands are *W* and an 8-bit literal encoded in the instruction word, and the destination is *W*. The `comf` (complement) operation complements each bit of its source operand, *floc*, with the result written to either *W* or *floc*.

		BBBB	BBBB	BBBB	BBBB
		1111	1100	0000	0000
		5432	1098	7654	3210
<code>andwf <i>floc</i>, [<i>d</i>], <i>a</i></code>	$d \leftarrow (floc) \& (w)$	0001	01da	ffff	ffff
<code>iorwf <i>floc</i>, [<i>d</i>], <i>a</i></code>	$d \leftarrow (floc)   (w)$	0001	00da	ffff	ffff
<code>xorwf <i>floc</i>, [<i>d</i>], <i>a</i></code>	$d \leftarrow (floc) \wedge (w)$	0001	10da	ffff	ffff
<code>comf <i>floc</i>, [<i>d</i>], <i>a</i></code>	$d \leftarrow \sim(floc)$	0001	11da	ffff	ffff
<code>andlw <i>k</i></code>	$w \leftarrow (w) \& k$	0000	1011	kkkk	kkkk
<code>iorlw <i>k</i></code>	$w \leftarrow (w)   k$	0000	1001	kkkk	kkkk
<code>xorlw <i>k</i></code>	$w \leftarrow (w) \wedge k$	0000	1010	kkkk	kkkk

*ffffff*: lower 8-bits of *floc* address; *kkkkkkkk*: 8-bit literal  
*d*: 0 = *w*, 1 = *floc*  
*a*: 0 = ignore BSR (ACCESS), 1 = use BSR (BANKED)

**FIGURE 4.1** Instruction and machine code formats for bitwise logical operations.

The term *bitwise* is applied to the  $\&$  (AND),  $|$  (OR),  $\wedge$  (XOR), and  $\sim$  (complement) operators because the logical operation is performed on a bit-by-bit basis on the operand(s). Bitwise logical operations are useful for clearing (AND), setting (OR), or complementing (XOR) groups of bits. Figure 4.2 shows a bitwise AND operation implementation of the C statement `i = i & 0x0F` where `i` is a `char` (byte) variable. The constant `0x0F` is called a *bit-mask* (or simply mask). Any bit in the mask that is a “0” clears the result bit, while a “1” in the mask leaves the result bit unchanged. Thus, the operation `i = i & 0x0F` leaves the lower 4 bits of `i` unchanged, and clears the upper 4 bits. An easy way to remember this is the rule that “0 ANDed with *anything* is 0; 1 ANDed with *anything* is *anything*”.

The OR bitwise logical operator is used to set groups of bits, as “1 ORed with *anything* is 1; 0 ORed with *anything* is *anything*.” Thus, the operation `j = j | 0x0F` sets the lower 4 bits of `j` to “1”s, but leaves the upper 4 bits unchanged as shown in Figure 4.2. The XOR bitwise operation is used to complement groups of bits, as “1 XORed with *anything* is NOT(*anything*); 0 XORed with *anything* is *anything*.” Thus, the operation `k = k ^ 0x0F` complements the lower 4 bits of `k`, but leaves the upper 4 bits unchanged.

Why are bitwise operations useful? One simple example is ASCII uppercase to lowercase conversion (or vice versa). The ASCII code for the uppercase character “A” is `0x41`; the ASCII code for the lowercase character “a” is `0x61`. Any ASCII uppercase character code OR’ed with the value `0x20` converts that ASCII code to the lowercase equivalent by setting bit 5 to a “1”. An ASCII lowercase character code AND’ed with the value `0xDF` converts that ASCII code to the uppercase equivalent by clearing bit 5 to a “0”. Using the bitwise logical operations to set or clear a

particular bit takes two instructions, as seen in Listing 4.1, which clears bit 5 of data memory location 0x004.

Location 0x020 is the variable *i*, which contains the value 0x2C

In C	In Assembly	Execution
<i>i</i> = <i>i</i> & 0x0F; bitwise-AND	<code>movf 0x020,w</code> <code>andlw 0x0F</code> <code>movwf 0x20</code>	<i>i</i> = 0x2C = 0010 1100 &&&& &&&& mask = 0x0F = 0000 1111 ----- result = 0000 1100 = 0x0C
<i>i</i> = <i>i</i>   0x0F; bitwise-OR	<code>movf 0x020,w</code> <code>iorlw 0x0F</code> <code>movwf 0x20</code>	<i>i</i> = 0x2C = 0010 1100                 mask = 0x0F = 0000 1111 ----- result = 0010 1111 = 0x2F
<i>i</i> = <i>i</i> ^ 0x0F; bitwise-XOR	<code>movf 0x020,w</code> <code>xorlw 0x0F</code> <code>movwf 0x20</code>	<i>i</i> = 0x2C = 0010 1100 ^^ ^^ ^^ ^^ mask = 0x0F = 0000 1111 ----- result = 0010 0011 = 0x23
<i>i</i> = ~ <i>i</i> ; bitwise complement	<code>comf 0x020,f</code>	<i>i</i> = 0x2C = 0010 1100 ~ ~ ~ ~ ~ ~ ~ ~ result = 1101 0011 = 0xD3

**FIGURE 4.2** Bitwise AND, OR, XOR, complement operation examples.

**LISTING 4.1** Clearing one bit using ANDWF.

---

machine code	mnemonic	comment
0x0EDF	<code>movlw 0xDF</code>	;w has mask value of 0xDF
0x1604	<code>andwf 0x004,f</code>	;0x004 ← (0x004)&0xDF, clears bit 5

Clearing, setting, or complementing a single bit in a memory location is a commonly performed operation. To improve the efficiency of these operations, the instructions *bsf* (bit set *f*), *bcf* (bit clear *f*), and *btg* (bit toggle *f*) are included in the instruction set. The instruction formats and machine codes of these instructions are shown in Figure 4.3.

The *f* bits in the machine code specify the lower 8 bits of the operand source data memory location as with other PIC instructions. However, there is no *d* bit in the instruction word, so the source operand is *always* affected by these instructions. The target bit that is cleared, set, or complemented is specified by the second operand, a value between 7 and 0, and is encoded by the 3-bit field *bbb* within the

instruction word. The bit numbering is as expected, where bit 0 is the least significant (rightmost) bit, while bit 7 is the most significant (leftmost) bit. The code fragment in Listing 4.2 clears bit 5 of data memory location 0x004.

			BBBB	BBBB	BBBB	BBBB
			1111	1100	0000	0000
			5432	1098	7654	3210
bcf	<i>floc, b [,a]</i>	bit clear ( <i>floc</i> )[ <i>b</i> ]	1001	<i>bbba</i>	<i>ffff</i>	<i>ffff</i>
bsf	<i>floc, b [,a]</i>	bit set ( <i>floc</i> )[ <i>b</i> ]	1000	<i>bbba</i>	<i>ffff</i>	<i>ffff</i>
btg	<i>floc, b [,a]</i>	bit toggle ( <i>floc</i> )[ <i>b</i> ]	0111	<i>bbba</i>	<i>ffff</i>	<i>ffff</i>
btfsc	<i>floc, b [,a]</i>	bit test f, skip if clear	1011	<i>bbba</i>	<i>ffff</i>	<i>ffff</i>
btfss	<i>floc, b [,a]</i>	bit test f, skip if set	1010	<i>bbba</i>	<i>ffff</i>	<i>ffff</i>

*ffffff*: lower 8-bits of *floc* address; *bbb*: bit #  
*a* : 0 = ignore BSR (ACCESS), 1 = use BSR (BANKED)

**FIGURE 4.3** Instruction formats and machine codes for bit-oriented operations.

**LISTING 4.2** Clearing 1 bit using BCF.

---

machine code	mnemonic	comment
0x9A04	bcf 0x004,5	;clear bit 5 of location 0x004

Figure 4.4 gives examples of *bcf*, *bsf*, and *btg* execution. When affecting only 1 bit in a target memory location is required, the *bcf*/*bsf*/*btg* instructions are more efficient than the bitwise logical instructions (*andwf*/*iorwf*/*xorwf*).

The *bsf*, *bcf*, and *btg* instructions are called *bit-oriented* file register operations. Two other instructions fall into this classification; the *btfsc* (bit test f, skip if clear) and *btfss* (bit test f, skip if set) instructions. These two instructions are used to implement conditional code execution, as the instruction following the *btfsc*/*btfss* instruction is skipped if the indicated bit test is true. Recall the program that was written for the number sequencer computer in Chapter 2, “The Stored Program Machine.” This program contained a conditional jump instruction (*jc*) that jumped to a target address if the LOC input was a “1”. The *jc* instruction was used in a program that output the digit sequence “8,5,6,1” to the DOUT databus if LOC was “1”; else the digit sequence “3,2,4,8,5,6,1” was output. Listing 4.3 shows a PIC18 program that implements the number sequencing program of Chapter 2 in PIC18 assembly language.



Location 0x062 is the variable *k*, which contains the value 0x8A

Location 0x061 is the variable *j*, which contains the value 0xB2

Location 0x060 is the variable *i*, which contains the value 0x2C

In C	In Assembly	Execution
<code>k = k &amp; 0x7F;</code>	<code>bcf 0x062,7</code> (bit clear)	<i>k</i> = 0x8A = 1000 1010 <code>bcf k,7</code> <i>new k</i> = 0x0A = 0000 1010
<code>j = j   0x04;</code>	<code>bsf 0x061,2</code> (bit set)	<i>j</i> = 0xB2 = 1011 0010 <code>bsf j,2</code> <i>new j</i> = 0xB6 = 1011 0110
<code>i = i ^ 0x20;</code>	<code>btg 0x060,5</code> (bit toggle)	<i>i</i> = 0x2C = 0010 1100 <code>btg i,5</code> <i>new i</i> = 0x0C = 0000 1100

**FIGURE 4.4** Code examples for `bsf`, `bcf`, `btg` instructions.



**LISTING 4.3** PIC18 assembly program for number sequencing task.

```

(1)      CBLOCK 0x0
(2)      loc,out          ;byte variables
(3)      ENDC
(4)      org      0
(5)      goto    main
(6)      org      0x0200
(7)      main
(8)      ;movlw  0      ;uncomment for loc=0
(9)      movlw   1      ;uncomment for loc=1
(10)     movwf   loc    ;initialize loc
(11)     Ltop
(12)     btfsc   loc,0  ; skip next if loc(0) is '0'
(13)     goto    loc_lsb_is_1
(14)     ;LSb of loc = 0 if reach here
(15)     movlw   3      ; W ← 3
(16)     movwf   out    ; out ← (W)
(17)     movlw   2      ; W ← 2
(18)     movwf   out    ; out ← (W)
(19)     movlw   4      ; W ← 4
(20)     movwf   out    ; out ← (W)
(21)     loc_lsb_is_1
(22)     movlw   8      ; W ← 8
(23)     movwf   out    ; out ← (W)
(24)     movlw   5      ; W ← 5
(25)     movwf   out    ; out ← (W)
(26)     movlw   6      ; W ← 6
(27)     movwf   out    ; out ← (W)
(28)     movlw   1      ; W ← 1
(29)     movwf   out    ; out ← (w)
(30)     goto    Ltop  ; loop forever
(31)     end

```

The single bit input LOC and output bus DOUT of the number sequencing computer are emulated in this program by the memory locations `loc` (0x000) and `out` (0x001) using the `CBLOCK` of lines 1 to 3. While this may seem like a poor replacement for external input/output pins, you will discover later that external pins on the PIC are actually accessed via data memory locations. Lines 8 through 10 initialize `loc` to either “0” (line 8 uncommented) or “1” (line 9 uncommented). The `btsfc loc,0` instruction in line 12 tests the least significant bit (i.e., bit 0) of `loc`; if this bit is “0”, the `goto loc_1sb_is_1` instruction at line 13 is skipped and the next instruction executed is at line 15. The `out` memory location is then written in succession with the values “3,2,4,8,5,6,1”. The `goto Ltop` instruction at line 30 causes a jump back to the `btsfc` instruction at line 12. If `loc` is given an initial value of “1”, the `btsfc loc,0` instruction of line 12 will not skip the `goto loc_1sb_is_1` instruction at line 13. This causes the program to jump to label `loc_1sb_is_1` (the label for the instruction of line 22), skipping the instructions between lines 14 to 20. Thus, for `loc = 1`, the number sequence “8,5,6,1” is written in succession into the `out` memory location. Observe that any odd number written to the `loc` memory location works equally well to select the sequence “8,5,6,1”, as the LSb of an odd number is “1”. Conversely, any even number written to the `loc` memory location selects the sequence “3,2,4,8,5,6,1”.

**Sample Question:** Write a C statement that clears the LSb of a char variable `j`. Implement this in PIC18 assembly language.

*Answer:* The C statement `j = j & 0xFE` clears the LSb of `j`, as `0xFE = 0b11111110`. Thus, bits 7 through 1 are unaffected, while bit 0 is cleared. This is implemented in PIC18 assembly as shown in Listing 4.4.

---

**LISTING 4.4** Sample question solution.

---

```

movf   j,w      ; w = j
andlw  0xFE     ; w = w & 0xFE
movwf  j        ; j = w

```

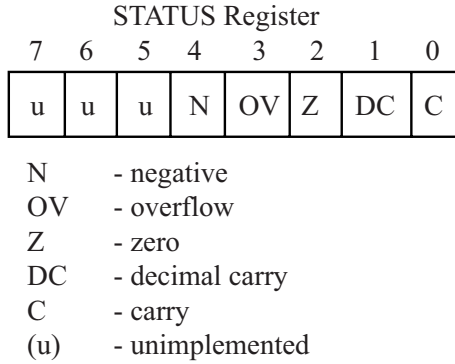
This can also be implemented using the single instruction `bcf j,0`.

## 4.3 THE STATUS REGISTER

---

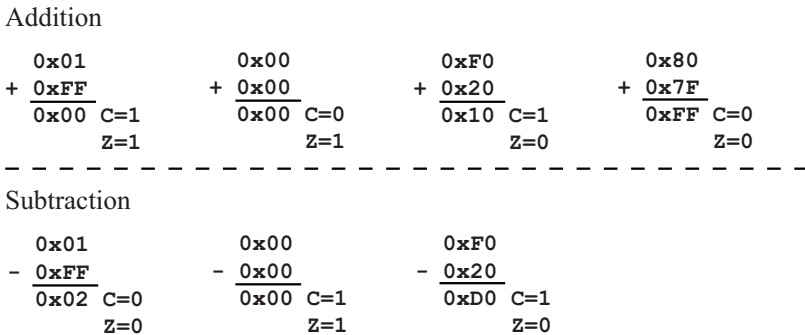
The STATUS register is a very important special function register that we have been ignoring up to this point. The STATUS, W, and BSR registers are the three most important 8-bit special function registers in the PIC18, as they are involved in the execution of most instructions. The lower 5 bits of the STATUS register contain

1-bit *flags*, which are set or cleared as a side effect of instruction execution. Figure 4.5 gives the names and positions of each of the flag bits within the STATUS register.



**FIGURE 4.5** The STATUS register.

The *zero* (Z) flag is set if the result of an instruction is zero, else it is cleared. The *Carry* (C) flag is set equal to the carry out of the most significant bit. The *Decimal Carry* (DC), *Overflow* (OV), and *Negative* (N) flags are discussed in Chapter 5 (OV, N) and Chapter 7 (DC). Different instructions affect different flags. The instruction summary for each PIC18 instruction given in Appendix A, “PIC18Fxx2 Architecture, Instruction Set, Register Summary,” lists the flags affected by each instruction. For example, the *addwf* instruction affects all five flags, the *movf* affects only the Z and N flags, and the *movff* instruction affects no flags. If a flag is unaffected by an instruction, it retains its current value. Figure 4.6 shows how the Z, C flags are affected by different addition and subtraction operations.



**FIGURE 4.6** C, Z flags for add/subtract operations.

An addition operation sets the C flag for a carry out of the MSb, while a subtraction operation clears the C flag if a borrow occurs out of the MSb. Note that the flag combination of  $Z = 1$  and  $C = 0$  cannot occur after a subtraction, as zero is produced only if the numbers are equal, which cannot produce a borrow out of the MSb. The reason behind the Carry flag condition after a subtraction is explained in Figure 4.7. The subtraction operation  $A - B$  is actually performed in hardware as  $A + (\sim B) + 1$ , where the value  $\sim B + 1$  is called the *two's complement* of B. Two's complement representation and its uses are explored in the next chapter when discussing signed number encoding. Note that the Carry flag produced by the addition operation  $A + (\sim B) + 1$  in Figure 4.7 matches the behavior of  $\text{Carry} = \sim \text{Borrow}$ ; in other words,  $C = 0$  if a borrow occurs, else  $C = 1$ .

$  \begin{array}{r}  A - B \\  0x\text{F0} \\  - 0x\text{20} \\  \hline  0x\text{D0} \quad C=1 \\  \quad Z=0  \end{array}  $	$  \begin{array}{r}  0x\text{20} = 0010 \ 0000 \\  \sim(0x\text{20}) = 1101 \ 1111 \\  = 0x\text{DF}  \end{array}  $	$  \begin{array}{r}  A + \sim B + 1 \\  0x\text{F0} \\  + 0x\text{DF} \\  + 0x\text{01} \\  \hline  0x\text{D0} \quad C=1 \\  \quad Z=0  \end{array}  $
--	--	---

**FIGURE 4.7** Subtraction of  $A - B$  performed as  $A + (\sim B) + 1$ .

**Sample Question:** What are the C, Z flag settings after the instruction `subwf WREG, w` is executed? After `movff i, j`? Use the instruction set table in Appendix A to determine the flags that are affected by an instruction execution.

**Answer:** The instruction `subwf WREG, w` affects all flags; the result is 0 since the operation is  $W = W - W$ . This produces no borrow, so the flag settings after execution are  $Z = 1, C = 1$ . The `movff` instruction affects no flags, so the C, Z flag settings after `movff` execution are left in the same state as before the instruction is executed.

## 4.4 UNSIGNED CONDITIONAL TESTS

Our first use of the Zero and Carry flags is to implement conditional code execution. This requires an overview of conditional tests in C, which are used principally in `if{}` statements and loop structures.

### Conditional Tests in C

Table 4.2 lists the conditional test operators for the C programming language. A conditional operator returns “1” if the test is true and “0” if the test is false.

**TABLE 4.2** Conditional Tests in C

Operator	Description
<code>==, !=</code>	equal, not equal
<code>&gt;, &gt;=</code>	greater than, greater than or equal
<code>&lt;, &lt;=</code>	less than, less than or equal
<code>&amp;&amp;,   </code>	logical AND, logical OR
<code>!</code>	logical negation

Listing 4.5 shows examples of the conditional operators; after execution, `a_lt_b` is “1” (true), `a_eq_b` is “0” (false), `a_gt_b` is “0” (false), and `a_ne_b` is “1” (true).

**LISTING 4.5** Examples of C equality and inequality tests.

```

unsigned char a,b,a_lt_b, a_eq_b, a_gt_b, a_ne_b;
a = 5; b = 10;
a_lt_b = (a < b);      // result is 1
a_eq_b = (a == b);    // result is 0
a_gt_b = (a > b);     // result is 0
a_ne_b = (a != b);    // result is 1

```

The logical AND (`&&`), OR (`||`), and negation (`!`) operators differ from the bitwise logical operators of AND (`&`), OR (`|`), and complement (`~`) in the fact that the logical operators treat their operand(s) as either zero or nonzero, and always returns a value of “0” or “1”. Care must be taken to remember this important difference; else your C code may behave in an unexpected manner. Listing 4.6 compares results produced by logical versus bitwise operators. Observe that the logical operator `&&` in line 3 gives the opposite result of the bitwise `&` operator. In line 2, a more verbose way to write the statement `(a && b)` is `((a != 0) && (b != 0))`, which returns a “1” only if both `a` and `b` are nonzero. The second form clearly illustrates how the operands `a`, `b` are treated by the logical `&&` operator. The statement `(!b)` in line 7 may be somewhat confusing at first, as it returns a “1” if `b` is zero. An alternate way to write `(!b)` is `(b == 0)`. Thus, a statement such as `(a && !b)` is equivalent to `((a != 0) && (b == 0))`, which returns a “1” only if `a` is nonzero, and `b` is zero.

**LISTING 4.6** Examples of C logical operators.

```

(1) unsigned char a,b,a_land_b, a_band_b;
    unsigned char a_lor_b, a_bor_b, a_lneg_b, a_bcom_b;
(2) a = 0xF0; b = 0x0F;
(3) a_land_b = (a && b);    //logical and, result is 1

```

```

(4) a_band_b = (a & b);      //bitwise and, result is 0
(5) a_lor_b = (a || b);     //logical or, result is 1
(6) a_bor_b = (a | b);     //bitwise or, result is 0xFF
(7) a_lneg_b = (!b);       //logical negation, result is 0
(8) a_bcom_b = (~b);       //bitwise negation, result is 0xF0

```

## Zero, Nonzero Conditional Tests

The use of the conditional operators in Listings 4.5 and 4.6 is atypical, as they are most often used in conditional tests for `if-else` statements or loop structures. The `C if-else` statement structure is shown in Figure 4.8. Observe that the `if_body` is executed if the conditional test is true (returns 1), while the `else_body` is executed if the conditional test is false (returns 0). Use of an `else_body` in an `if{}` statement is optional.

```

if (condition_test) {
    if_body ← Executed when condition_test is non-zero (true)
} else {
    else_body ← Executed when condition_test is zero (false)
}

```

**FIGURE 4.8** The `if-else` statement in C.

Figure 4.9 shows a nonzero test used as the conditional for an `if{}` statement. The test `if(i)` is equivalent to `if(i != 0)`; there is no advantage to either form. The nonzero test is accomplished by the `movf i, f` instruction, which copies the value of `i` back onto itself. While this leaves `i` unchanged, it does affect the Z, N flags. The flag of interest in this case is the Z flag, which is “0” if `i` is nonzero. The `btfs STATUS, Z` (bit test, skip if clear) skips the following instruction if `Z = 0`, causing the `if_body` to be executed. If `Z = 1`, the `goto end_if` instruction following the `btfs` instruction is executed, causing the `if_body` to be skipped.

An alternate method for accomplishing the same nonzero test is shown in Figure 4.10, where the `btfs/goto` combination is replaced by the single instruction `bz` (branch if zero; e.g., branch if `Z=1`). The `bz` instruction branches (jumps) around the `if_body` if `Z = 1`, which is true if `i` is zero. The `bz` instruction is one of several *branch* instructions that perform a conditional jump based on the setting of a `STATUS` flag. The branch instructions are:

```

bnz: (branch if not zero, Z = 0)
bz: (branch if zero, Z = 1)

```

In C	In Assembly
<pre> unsigned char i, j;  if (i) {     // do this if i is non-zero     j = i + j; } // ...rest of code... </pre>	<pre> movf i,f      ; i = i btfsc STATUS,Z ; skip if Z=0 goto end_if   ; Z=1, i is 0 → movf i,w    ; w = i   addwf j,f   ; j = j + i   end_if ←   ..rest of code.. </pre>

**FIGURE 4.9** Nonzero test using `movf/btfsc/goto`.

**bnc:** (branch if not carry, C = 0)  
**bc:** (branch if carry, C = 1)  
**bnn:** (branch if not negative, N = 0)  
**bn:** (branch if negative, N = 1)  
**bnov:** (branch if no overflow, V = 0)  
**bov:** (branch if overflow, V = 1)  
**bra:** (branch always, this is an unconditional branch)

Using branch instructions typically improves code clarity, and generally results in fewer instruction words. The examples in this book use branch instructions wherever possible. The machine code format of branches is discussed in the next chapter, after signed number representation is covered.

In C	In Assembly
<pre> unsigned char i, j;  if (i) {     // do this if i is non-zero     j = i + j; } // ...rest of code... </pre>	<pre> movf i,f      ; i = i bz end_if     ; skip if Z=1, i is 0 → movf i,w    ; w = i   addwf j,f   ; j = j + i   end_if ←   ..rest of code.. </pre>

**FIGURE 4.10** Nonzero test using `movf/bz`.

A zero test is written as either `if (!i)` or as `if (i == 0)` and is implemented in the same manner as Figure 4.10, except a `bnz` instruction replaces the `bz` instruction.

## Equality, Inequality Conditional Tests

Figure 4.11 shows the implementation in PIC18 assembly code of an `if{}`  statement that has the equality test `i == j` as its condition. The equality test is performed by the subtraction `i - j`, followed by a `bnz` that branches to the end of the `if{}`  statement if `Z = 0`, indicating that `i` is not equal to `j`. The use of the subtraction operation to affect status flags for conditional test purposes is a common theme that is useful for all types of comparison operations. It does not matter if `i - j` or `j - i` is performed, as both affect the `Z` flag in the same way. An inequality test `i != j` is performed in a similar manner, with a `bz` replacing the `bnz` instruction.

In C	In Assembly
<pre> unsigned char i, j;  if (i == j) {     // do this if i equal to j     j = i + j; } // ...rest of code... </pre>	<pre> movf j,w ; w = j subwf i,w ; w = i - j bnz end_if ; skip if Z=0, i != j movf i,w ; w = i addwf j,f ; j = j + i →end_if ..rest of code.. </pre>

**FIGURE 4.11** Equality test using `subwf/bnz`.

**Sample Question: Implement the C statement** `if (i && j){ k++; } in PIC18 assembly language.`

*Answer:* The conditional test `i && j` is the same as `i != 0 && j != 0`, the `incf k, f` statement in Listing 4.7 is only executed if both `i` and `j` are nonzero.

### LISTING 4.7 Sample question solution.

```

movf i,f ; test i
bz end_if ; skip if i is zero
movf j,f ; test j
bz end_if ; skip if j is zero
incf k, f ; do k++
end_if
.....rest of code.....

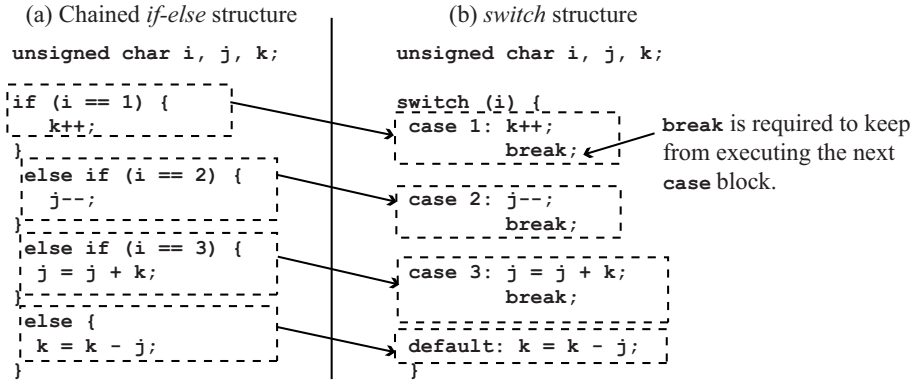
```

## The switch Statement

A common structure in C code is a chained if-then-else structure that compares a variable against several constant values to implement a choice selection. In fact, it



is so common that *C* has a shorthand method called a *switch* statement to implement this structure as shown in Figure 4.12.



**FIGURE 4.12** switch statement structure.

Each case block of the switch statement in Figure 4.12b corresponds to one of the `if(){}`  blocks in Figure 4.12a. The literal values 1, 2, 3 used for comparison in the switch statement do not have to be sequential; they can be any values. The `break` statement that ends each case block is very important; if a `break` statement is not included, the next case code is executed regardless of the value of the switch variable. The reason for this behavior is clear in Figure 4.13, which shows the assembly language implementation of the switch statement of Figure 4.12b.

The `break` statement translates to a `bra` statement that jumps to the end of the switch block; if the `break` statement is not included, execution falls through to the next case block.

**Greater-than (>), Greater-than-or-equal (>=) Conditional Tests**

Figure 4.14 illustrates how the test `i > j` is performed. If `i > j` is true, then the subtraction `j - i` produces a borrow as a larger number is subtracted from a smaller number. A borrow clears the Carry flag (`C = 0`), so the `bc` (branch if carry) skips the `if_body` if `C = 1`, indicating the condition `i > j` is false.

A common mistake for a `i > j` comparison is to perform the subtraction `i - j` instead of `j - i`, and to branch around the `if_body` on the condition `C = 0`, caused by a borrow because `j > i`. However, this means the `if_body` is executed for the case of `C = 1` (no borrow). The case `C = 1` is true for two conditions: `i > j` and `i == j`, and is thus the comparison `i >= j`. If the subtraction `i - j` is performed for the comparison `i > j`, the branch code must be structured such that the `if_body` is only

In C	In Assembly
unsigned char i, j, k; switch (i) {	movlw 1 ; w = 1 subwf i,w ; i == 1?
case 1: k++; break;	bnz case_2 incf k ; k++ bra end_switch → ; break statement
case 2: j--; break;	movlw 2 ; w = 2 subwf i,w ; i == 2? bnz case_3 decf j ; j-- bra end_switch → ; break statement
case 3: j = j + k; break;	movlw 3 ; w = 3 subwf i,w ; i == 3? bnz default movf k,w ; j = j + k addwf j,f ; break statement bra end_switch →
default: k = k - j;	movf j,w ; k = k - j subwf k,f
} // end switch	end_switch ← ..rest of code..

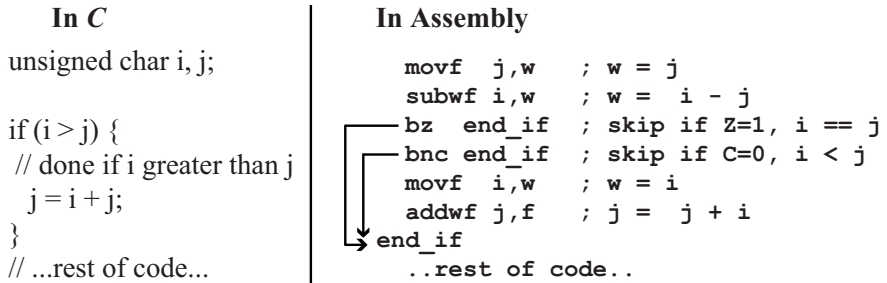
**FIGURE 4.13** Assembly language implementation of a switch statement.

In C	In Assembly
unsigned char i, j;	
if (i > j) {	movf i,w ; w = i subwf j,w ; w = j - i
// done if i greater than j	bc end_if ; skip if C=1, i <= j
j = i + j;	movf i,w ; w = i addwf j,f ; j = j + i
}	end_if →
// ...rest of code...	..rest of code..

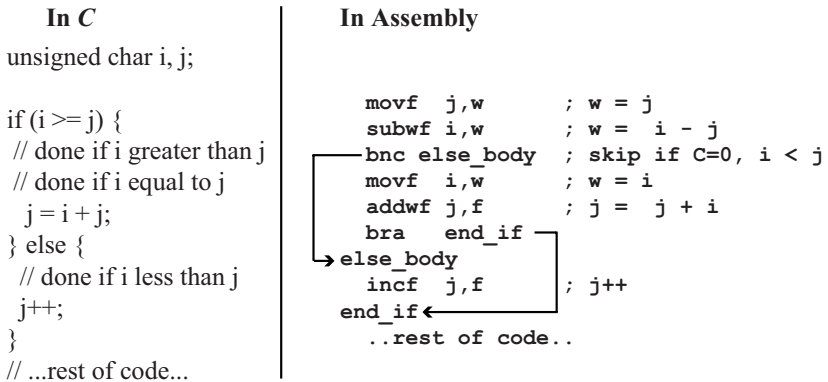
**FIGURE 4.14** Unsigned greater-than (>) test using subwf/bc.

executed for the case  $C = 1$  (no borrow,  $i > j$ ) and  $Z = 0$  ( $i \neq j$ ). Figure 4.15 illustrates this; obviously, requiring two flag tests increases code complexity, so the subtraction  $i - j$  should be avoided for the  $i > j$  comparison.

Figure 4.16 shows a  $i \geq j$  conditional test used within an if-else statement. The subtraction  $i - j$  is performed, and a branch to the else\_body is taken if  $C = 0$  (a borrow), which is true if  $j > i$ . Observe that a bra end\_if (branch always) is used as the last statement in the if\_body to jump to the end of the if-else statement, skipping around the else\_body. As you may have deduced, if the subtraction



**FIGURE 4.15** Unsigned greater-than (>) test variations.



**FIGURE 4.16** Unsigned greater-than-or-equal (>=) test using subwf/bnc.

$j - i$  is used for the comparison  $i \geq j$ , two flag tests are required; the code must be structured such that the `if_body` is executed for the case  $Z = 1$  ( $i == j$ ) or  $C = 0$  (a borrow,  $i > j$ ).

Table 4.3 summarizes the preferred operations and flag tests for unsigned comparison operations. Notice that the inequality  $i < j$  can be reversed as  $j > i$ , and  $i \leq j$  as  $j \geq i$ .

### Unsigned Comparisons using `cpfseq`, `cpfsgt`, `cpfslt`

The instructions `cpfseq` (compare `f` with `W`, skip if equal), `cpfsgt` (compare `f` with `W`, skip if greater than), and `cpfslt` (compare `f` with `W`, skip if less than) directly support 8-bit unsigned comparisons. The code in Listing 4.8 implements the  $i > j$  comparison of Figure 4.14 using the `cpfsgt` instruction. Observe that this takes the same number of instruction words as the `subwf/bc` code in Figure 4.14.

**TABLE 4.3** Comparison Summary

Comparison	Operation	If True, Then
if (i) {}	$i = i$ i.e., <code>movf i, f</code>	$Z = 0$
if (!i) {}	$i = i$ i.e., <code>movf i, f</code>	$Z = 1$
if (i == j) {}	$i - j$ OR $j - i$ ;	$Z = 1$
if (i != j) {}	$i - j$ OR $j - i$	$Z = 0$
if (i > j) {}	$j - i$	$C = 0$ (borrow)
if (i >= j) {}	$i - j$	$C = 1$ (no borrow)

**LISTING 4.8** Unsigned  $i > j$  comparison using `cpfsgt`.

```

movf    j, w        ;w ← (j)
cpfsgt  i           ;i > j?
bra     end_if      ;if branch taken, i <= j
movf    i, w        ;w ← (i)
addwf   j, f        ;j = j + i
end_if
;;rest of code

```

The advantage of these instructions is that they perform the subtraction and needed flag test with one instruction, thereby improving code clarity and sometimes reducing the number of instruction words. The disadvantage is that these instructions cannot be used for comparisons of extended precision numbers (16-bit, 32-bit, etc.), and the `cpfsgt/cpfls1t` instructions cannot be used for signed number comparisons (the reasons for both of these problems becomes clear in the next chapter). The subtraction/branch approach is a general method that is useful for comparing numbers of arbitrary precision, and for both unsigned and signed comparisons. Extended precision comparisons and signed number comparisons are covered in the next chapter, using the subtraction/branch approach.

## Unsigned Literal Comparisons

Figure 4.17 shows two different methods for comparing an unsigned `char` variable to a literal, using the comparison  $i > 0x40$  as the example. One version uses the `cpfsgt` instruction by loading `0x40` into `W`, then executing `cpfsgt i`, which does the comparison  $i > 0x40$ . If the comparison is true, the `bra end_if` instruction is skipped, causing the `if_body` to be executed. The second version uses the `sublw` instruction, which subtracts `W` from a literal. To perform the  $i > 0x40$  comparison, the operation  $0x40 - i$  is performed by loading `i` into `W`, and then executing `sublw`

0x40. If the Carry flag is set, then no borrow occurred, indicating  $0x40 \geq i$ , and the `bc end_if` instruction causes the `if_body` to be skipped.

In C	In Assembly Using <code>cpfsgt</code>	In Assembly Using <code>sublw</code>
unsigned char i,j;		
if (i > 0x40) {	<code>movlw 0x40 ; w = 0x40</code>	<code>movf i,w ; w = i</code>
i = i + j;	<code>cpfsgt i ; i &gt; 0x40?</code>	<code>sublw 0x40 ; w = 0x040 - i</code>
}	<code>bra end_if ; no, skip</code>	<code>bc end_if ; skip if 0x40 &gt;= i</code>
// ...rest of code...	<code>→movf j,w ; w = j</code>	<code>movf j,w ; w = j</code>
	<code>addwf i,f ; i = i + j</code>	<code>addwf i,f ; i = i + j</code>
	<code>end_if ←</code>	<code>end_if</code>
	<code>..rest of code..</code>	<code>..rest of code..</code>

**FIGURE 4.17** Unsigned literal comparisons.

**Sample Question: Implement the C statement `if (i >= 0x20 && i <= 0x41) {j++;}` in PIC18 assembly language.**

*Answer:* In Listing 4.9, the comparison `i <= 0x41` is reversed as `0x41 >= i` and is tested using the subtraction `0x41 - i`.

**LISTING 4.9** Sample question solution.

```

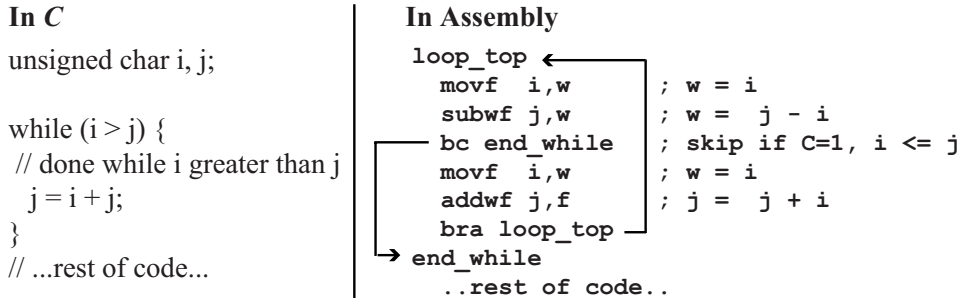
movlw    0x20        ; w = 0x20
subwf   i,w          ; i - 0x20 for i >= 0x20 test
bnc     end_if       ; if C = 0, test is false, go to end
movf    i,w          ; w = i
sublw   0x41         ; 0x41 - i for 0x41 >= i test
bnc     end_if       ; if C = 0, test is false, go to end
incf    j,f          ; only do this if both tests are true
end_if
....rest of code....

```

## 4.5 LOOPING

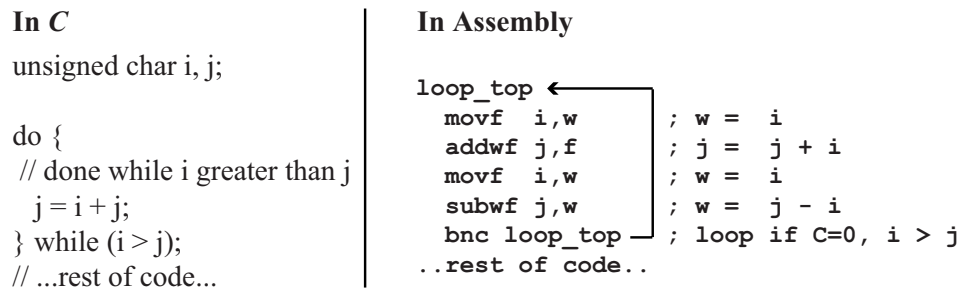
A `while{}` loop structure has the same form as an `if{}` structure, except the body is executed as long as the condition test is true. Figure 4.18 shows a `while{}` code example implemented in PIC18 assembly language. The condition test for the `while{}` loop is implemented in the same manner as for an `if{}` statement; if the condition test is false, a jump is made to the end of the loop body. In this case, the `subwf/bc` instructions implement the condition test `i > j`, which is the same test used in Figure 4.14. The only difference between the assembly code in Figure 4.14 (`if{}` statement) and that in Figure 4.18 is the `bra loop_top` instruction at the end of the loop

body that causes an unconditional jump back to the condition test at the beginning of the loop.



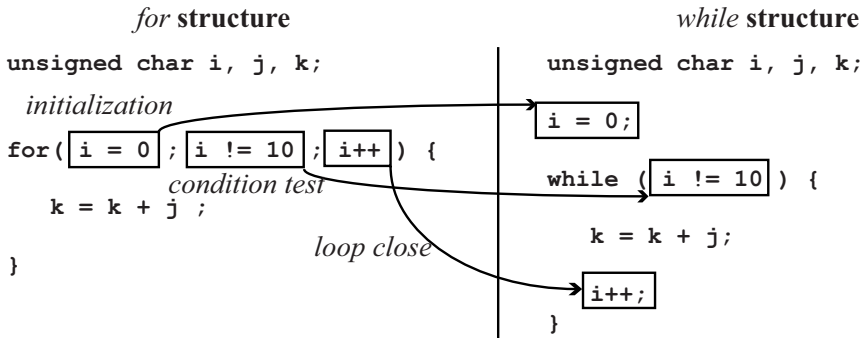
**FIGURE 4.18** Loop structure `while{}` example.

A `do-while{}` loop structure and example usage is given in Figure 4.19. Observe that the loop body is guaranteed to execute at least once. The condition test for  $i > j$  is implemented by the `subwf/bnc` instruction combination. Even though the same condition test  $i > j$  is used for both Figure 4.18 (`while{}`) and Figure 4.19 (`do-while{}`), the branches are different because the `bc` instruction of Figure 4.18 causes the loop body to be skipped, while the `bnc` instruction of Figure 4.19 causes the loop body to be re-executed by jumping back to the top of the loop.



**FIGURE 4.19** Loop structure `do-while{}` example.

A commonly used loop structure in the C language is the `for{}` loop. Figure 4.20 shows that a `for{}` loop is simply a shorthand notation for a `while{}` loop. As such, the use of a `for{}` loop structure is optional, as it can always be written as a `while{}` loop.



**FIGURE 4.20** Loop structure for{ }.

Many loops are counting loops, where a loop is executed a fixed number of times by incrementing or decrementing a counter within the loop body to track the number of times the loop body is executed. The variable that is incremented or decremented is called the *loop counter*. The PIC18 has four instructions that are useful for loop counters:

- decfsz f, d, a:** Decrement f, skip if zero
- dcfsnz f, d, a:** Decrement f, skip if not zero
- incfsz f, d, a:** Increment f, skip if zero
- infsnz f, d, a:** Increment f, skip if not zero

These instructions combine an increment/decrement operation with a zero/nonzero test of the operand. Figure 4.21 shows a counting loop that executes the statement  $k = k + j$  10 times. The loop counter is  $i$ , and the `decfsz i` instruction is used to decrement the counter and exit the loop when  $i$  reaches zero. Note that the `decfsz/bra` combination could be replaced by a `decf/bnz` combination that is equally effective (and probably clearer in its intent!). While in some cases, the `decfsz`, `dcfsnz`, `incfsz`, and `infsnz` instructions can reduce code size and/or decrease loop execution time, this book attempts to use instruction sequences that maximize understanding, and assembly code optimization is not emphasized in examples. There are other uses for these instructions aside from loop counters, as seen in the next chapter.

**In C**

```

unsigned char i, j, k;

i = 10;
do {
    k = k + j;
    i--;
} while (i != 0);
// ...rest of code...

```

**In Assembly**

```

movlw 0x0A    ; w = 10
movwf i      ; i = 10
loop_top ←
movf j,w     ; w = j
addwf k,f    ; k = k + j
decfsz i,f   ; i--, skip if zero
bra loop_top ; loop if i non-zero
← ..rest of code..

```

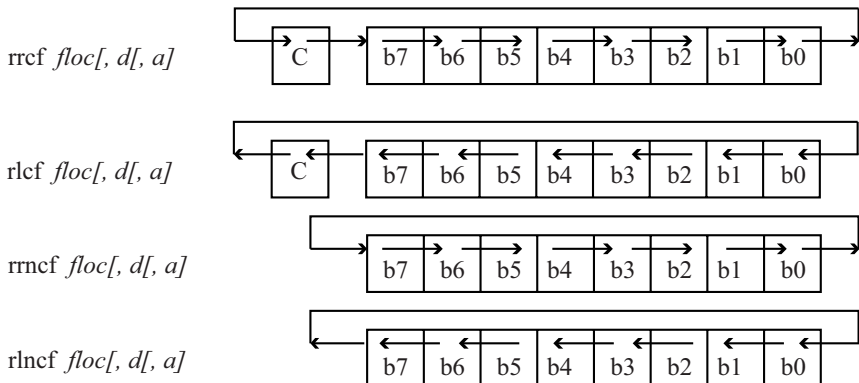
**FIGURE 4.21** Counting loop example.

**Sample Question:** In the code in Figure 4.19, replace the  $i > j$  test with  $i \neq j$  and modify the assembly code appropriately.

**Answer:** The `bnc loop_top` instruction is replaced with a `bnz loop_top` instruction, as  $Z = 0$  from the subtraction indicates that  $i$  is not equal to  $j$  so the loop execution should continue.

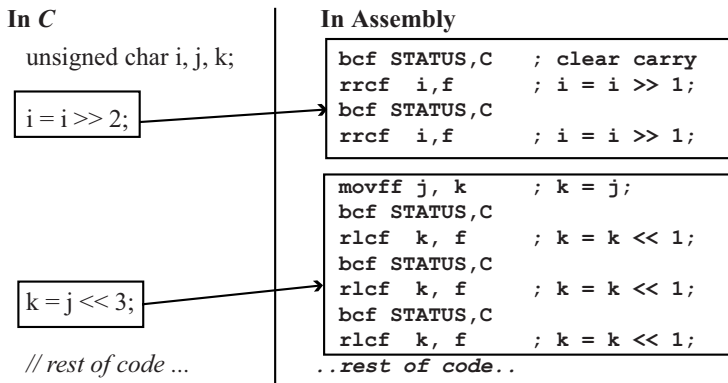
## 4.6 SHIFTS AND ROTATES

Shift operations are useful for either multiplication by two (shift left), division by two (shift right), or moving bits to new positions within a register. Figure 4.22 shows the four *rotate* instructions available in the PIC18 instruction set for implementing shifts (see Appendix A for the machine code format). The term *rotate* is applied to these instructions because instruction execution rotates the affected bits through a fixed set of positions; no bit is “lost” by the rotate operation.

**FIGURE 4.22** Rotate instructions.



The `rlcf` (rotate left  $f$ , no carry) instruction rotates the bits one position to the left, with the most significant bit rotating to the least significant bit position. The `rrcf` (rotate right  $f$ , no carry) instruction rotates the bits one position to the right, with the least significant bit rotating to the most significant bit position. The `rlcf` (rotate left  $f$  through carry) and `rrcf` (rotate right  $f$  through carry) includes the Carry flag in this rotation. Each rotate instruction only shifts the bits by one position; shifting bits  $n$  positions requires  $n$  rotate instructions. Figure 4.23 illustrates how the right shift (`>>`) and left shift (`<<`) operations of `C` are implemented using `rrcf` and `rlcf` instructions. Observe that before each `rrcf` or `rlcf` instruction, the Carry flag is cleared to ensure that a “0” is shifted into the most significant bit for right shift, or into the least significant bit for left shift. Observe that a multiple bit shift such as `i = i >> 2` is simply a 1-bit shift repeated two times. The first instruction for the operation `k = j << 3` copies the value of `j` to `k` (`movff j,k`), and then successive `bcf/rlcf` instruction pairs are used to shift `k`. A common mistake for new assembly language programmers is to use the instruction `rlcf j,f` in implementing `k = j << 3`. This changes the value of `j`, an incorrect side effect. More powerful microprocessors have shift instructions that can shift multiple bit positions with one instruction.



**FIGURE 4.23** C Shift operations using `rlcf/rrcf` instructions.

Listing 4.10 shows how using shift/subtract instructions can perform a constant multiplication. In this example, the computation `i = i * 7` is performed by computing `i = (i*8) - i` or `i = (i<<3) - i`. This raises the question of how the operation `k = i * j` is accomplished, which is a variable multiplication. Variable multiplication and division are covered in Chapter 7, “Advanced Assembly Language: Higher Math.”

**LISTING 4.10** Multiplication  $I * 7$  by  $I = (I \ll 3) - I$ .

```

;i = i * 7 = i * (8 - 1) = i*8 - i = i << 3 - i
;Could also be done as:
;i = i * 7 = i * (4 + 2 + 1) = i*4+i*2+i = i<<2 + i<<1 + 1
;In PIC18 assembly (temp is a temporary location):
movff  i,temp      ;save original i
bcf    STATUS,C    ;clear CARRY
rlcf   i,f         ;i = i << 1
bcf    STATUS,C    ;clear CARRY
rlcf   i,f         ;i = i << 1
bcf    STATUS,C    ;clear CARRY
rlcf   i,f         ;i = i << 1
movf   temp,w     ;w = original i
subwf  i,f         ;i = i<<3 - i = i*7

```

**Sample Question:** What is wrong with the following implementation of the statement  $i = i \ll 2$ ?

```

rlcf   i,f         ; i = i << 1
rlcf   i,f         ; i = i << 1

```

*Answer:* The error in this code is that the Carry flag state is unknown before each `rlcf` instruction, so the bit shifted into the LSb of `i` can be either 0 or 1. The Carry flag must be cleared before each `rlcf` instruction. It is not sufficient to only clear the Carry flag before the first `rlcf` instruction, as the Carry becomes equal to the old MSb of `i` after the first `rlcf` instruction is executed. A correct solution is shown in Listing 4.11.

**LISTING 4.11** Sample question solution.

```

bcf    STATUS,C    ; clear carry flag so 0 is shifted into LSb
rlcf   i,f         ; i = i << 1
bcf    STATUS,C    ; clear carry flag so 0 is shifted into LSb
rlcf   i,f         ; i = i << 1

```

**Sample Question:** Implement the C statement  $k = (i \ll 1) + (j \gg 1)$  in PIC18 assembly language.

*Answer:* When faced with a multipart computation, it is a good practice to break the computation into smaller steps that are easier to translate to assembly language. The computation can be rewritten as shown in Listing 4.12. Observe that these computations only modify `k`, and that `W` is used as a temporary register for holding the shifted value of `j`.

**LISTING 4.12** Breaking a computation into several steps.

---

```

// k = (i << 1) + (j >> 1)
k = i;           // copy i to k
k = k << 1       // first two statements implement k = i << 1
W = j >> 1;     // compute j shift
k = k + W;      // compute final value of k

```

These individual steps are now translated to PIC18 assembly language as shown in Listing 4.13.

**LISTING 4.13** Sample question solution.

---

```

movff  i,k        ; k = i
bcf    STATUS,C   ; clear carry before shift
rlcf   k,f        ; k = k << 1
bcf    STATUS,C   ; clear carry before shift
rrcf   j,w        ; W = j >> 1
addwf  k,f        ; k = k + W

```

**SUMMARY**


---

In this chapter, we explored 8-bit arithmetic, logical, shift, and unsigned comparison operations. The limitations of 8-bit unsigned integer data are quite apparent, and will be removed in the next chapter when we discuss extended precision operations on signed integer data.

**REVIEW PROBLEMS**


---

For the following problems, assume that all variables *i*, *j*, *k* are unsigned char data types, with *i*, *j*, *k* assigned to locations 0x000, 0x001, and 0x1A0, respectively.

For problems 1 through 11, give the Z, C flags and affected registers after execution of each instruction and assume the following register contents at the beginning of EACH problem:

W = 0xD7, BSR = 0x0, STATUS = 0x00 (see Table 4.4 for remaining initial memory contents)

**TABLE 4.4** Initial Memory Contents for Review Problems

Location	Contents
0x000 (i)	0xA0
0x001 (j)	0x7A
0x04E	0x00
0x04F	0xFF
0x1A0 (k)	0xFE

1. Instruction: `subwf j, f`
2. Instruction: `addwf i, f`
3. Instruction: `andwf i, w`
4. Instruction: `iorwf j, w`
5. Instruction: `movff 0x04E, 0x04F`
6. Instruction: `bsf i, 3`
7. Instruction: `bcf j, 4`
8. Instruction: `btg j, 7`
9. Instruction: `r1cf j, w`
10. Instruction: `rrcf i, w`
11. Instruction: `xorlw 0x4E`

Write PIC18 assembly language equivalents for the following C code fragments. Be aware that *k* is in a different bank from *j* and *i*! To check your work, compile and execute the C code using your favorite C compiler on your personal computer, and check against the results obtained by your code using MPLAB.

12. Code fragment:
 

```
k = i + (j << 1);
```
13. Code fragment:
 

```
i = (i >> 1) | k;
```
14. Code fragment:
 

```
if (i > k){
    k = i << 2;
} else {
    j = k ^ j;
}
```

15. Code fragment:

```
i = 0;
while (i != k) {
    k++; i = i << 1;
}
```

16. Code fragment:

```
do {
    i = i - j;
    k--;
}
while (i && j);
```

17. Code fragment:

```
if (!i || j) {
    k = i + 2;
} else {
    k = k << 1;
}
```

18. The following C code counts the number of “1” bits in  $j$  and returns the answer in  $k$ . Convert this to PIC18 assembly code. The operation  $j \& 0x01$  tests the value of the LSB of  $j$  while  $\{\}$  loop structure.

```
k = 0; // init bit count
for (i = 0; i != 8; i++) { // do for 8 bits
    if (j & 0x01) {
        k++; // LSB = 1, increment count
    }
    j = j >> 1; // look at next bit
}
```

19. Write a PIC18 assembly language program that performs the multiplication  $k = i * 12$ .

20. Convert the instruction `btfsc 0x56,3,BANKED` to machine code.

21. What instruction is represented by the machine code `0x9A04`?

# 5

## Extended Precision and Signed Operations

### In This Chapter

- Extended Precision Integers
- Extended Precision Operations
- Signed Number Representation
- Two's Complement Overflow
- Operations on Signed Data
- Branch Instruction Encoding

This chapter applies the arithmetic, logical, and shift operations discussed in the previous chapter to extended precision operands; that is, operands that are larger than 8 bits. Furthermore, signed number representation and its effect on shift and comparison operations are covered.

### 5.1 LEARNING OBJECTIVES

---

After reading this chapter, you will be able to:

- Translate C language statements that perform extended-precision addition, subtraction, bitwise logical, and shift operations into PIC18 instruction sequences.

- Compare and contrast signed magnitude, one's complement, and two's complement representations of signed integers.
- Translate C language statements that perform shift and comparison operations using signed operands into PIC18 instruction sequences.
- Translate PIC18 instructions, such as branches, that use PC-relative addressing into machine code.

## 5.2 EXTENDED PRECISION INTEGERS

Previous chapters have used only the `unsigned char` data type in C programs, which limits these variables to a 0 to 255 integer range. Obviously, there is a need to accommodate larger number ranges both in C and in PIC18 assembly language programs. The `short`, `int`, and `long` data types in C are used for extended precision integers.

Table 5.1 shows the sizes and ranges of the `char`, `short`, `int`, and `long` data types for the HI-TECH C compiler used in this book (other C compilers for the PIC18 may use different data sizes). While a `char` data type is always a byte, the `short`, `int`, and `long` sizes are compiler and target-processor dependent. Other common choices for `int` and `long` are 32 bits, with a `short` being 16 bits. The `long` data size for 64-bit processors such as the Intel Itanium and AMD Athlon64 is 64 bits, with an `int` being 32 bits and `short` using 16 bits. The data types `char`, `int`, and `long` are used in this book's C examples.

**TABLE 5.1** Unsigned Ranges for C Data Types of `char`, `int`, `short`, `long`

C Data Type	Size (PICC-18 Compiler)	Unsigned Range
<code>char</code>	1 byte (8 bits)	0 to 255 ( $2^8 - 1$ )
<code>short</code>	2 bytes (16 bits)	0 to 65535 ( $2^{16} - 1$ )
<code>int</code>	2 bytes (16 bits)	0 to 65535 ( $2^{16} - 1$ )
<code>long</code>	4 bytes (32 bits)	0 to 4,294,967,295 ( $2^{32} - 1$ )

For variables that are more than 1 byte in size, Figure 5.1 shows the two choices for storing these bytes in memory. Little-endian byte ordering arranges the bytes least significant to most significant, while big-endian stores the bytes most significant to least significant. The acronyms MSB (most significant byte) and LSB (least significant byte) are used for the rightmost and leftmost bytes, respectively, of a

multibyte number. In this book, an uppercase “B” in MSB (LSB) refers to a byte, while a lowercase “b” in MSb (LSb) refers to a bit.

```
int i;    i = 0xA457;
long k;  k = 0x38B83DF2;
```

Assume *i* is location 0x20, *k* is location 0x22

	Little Endian		Big Endian												
Location (hex) :	20 21 22 23 24 25		20 21 22 23 24 25												
Contents (hex) :	<table style="border-collapse: collapse; margin: auto;"> <tr> <td style="border: 1px solid black; padding: 2px;">57</td> <td style="border: 1px solid black; padding: 2px;">A4</td> <td style="border: 1px solid black; padding: 2px;">F2</td> <td style="border: 1px solid black; padding: 2px;">3D</td> <td style="border: 1px solid black; padding: 2px;">B8</td> <td style="border: 1px solid black; padding: 2px;">38</td> </tr> </table>	57	A4	F2	3D	B8	38		<table style="border-collapse: collapse; margin: auto;"> <tr> <td style="border: 1px solid black; padding: 2px;">A4</td> <td style="border: 1px solid black; padding: 2px;">57</td> <td style="border: 1px solid black; padding: 2px;">38</td> <td style="border: 1px solid black; padding: 2px;">B8</td> <td style="border: 1px solid black; padding: 2px;">3D</td> <td style="border: 1px solid black; padding: 2px;">F2</td> </tr> </table>	A4	57	38	B8	3D	F2
57	A4	F2	3D	B8	38										
A4	57	38	B8	3D	F2										
	i                    k		i                    k												

**FIGURE 5.1** Little-endian versus big-endian byte ordering.

There is no inherent advantage to little-endian or big-endian byte order. The architects of the microprocessor determine the choice during the design phase. The Intel x86 processors use little-endian, while Motorola (68xxx family) and IBM processors (PowerPC) use big-endian. For the PIC18, there are some multibyte special function registers that have not been discussed yet that are arranged in little-endian order (LSB to MSB order). This means that any multibyte values copied to these registers are arranged in little-endian order, and thus this byte ordering is used in this book’s assembly language examples concerning extended precision operations. Furthermore, the C compiler on the book’s CD-ROM uses little-endian byte ordering for the same reason.



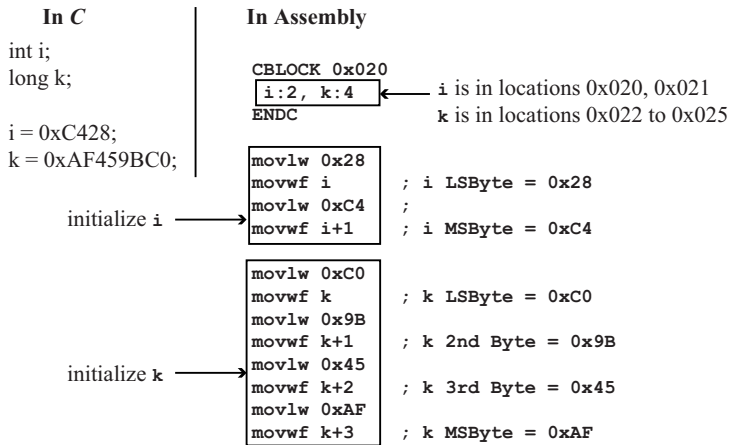
## 5.3 EXTENDED PRECISION OPERATIONS

The PIC18 is referred to as an 8-bit microcontroller because its registers, data paths, and ALU operands are 8-bits wide. As such, an operation such as value assignment, addition, subtraction, bitwise logical, or shift using extended precision (multibyte) operands is performed 1 byte at a time. This implies that an operation on an *int* variable will take approximately twice the number of instructions as for a *char* variable, while a *long* variable requires four times the number of instructions. As such, *int* and *long* variables should only be used in PIC18 applications if the extended range provided by those data types is actually needed by the application.



### Assignment of Extended Precision Variables

Figure 5.2 shows how a variable assignment is done for `int` and `long` operands. In the `CBLOCK`, the statement `i:2` reserves two consecutive locations for the label `i`, while `k:4` assigns four consecutive locations to `k`. The least significant byte of variable `i` is at location `0x20`, while the most significant byte of `i` is at location `0x21`. The instruction pair `movlw 0x28; movwf i` initializes the least significant byte of `i`; while `movlw 0xC4; movwf i+1` initializes the most significant byte. Note that the label `i + 1` refers to location `0x21`, the most significant byte of `i`. The C statement `k = 0xAF459BC0` requires four pairs of `movlw/movwf` statements, one pair for each byte of `k`. Location `0x22` (`k`) is the least significant byte of `k`, while `0x25` (`k + 3`) is the most significant byte. The order in which the `movlw/movwf` instruction pairs are written is not important, as long as the memory locations for `i`, `k` are initialized to the specified values in little-endian (least significant byte to most significant byte) order.



**FIGURE 5.2** Multibyte values in MPLAB.

### Bitwise Logical

Figure 5.3 shows a bitwise AND operation applied to `int` operands. Note that the `andwf` instruction is applied to each byte of the operands. It is immaterial as to the order in which a bitwise logical operation is applied to the bytes of an extended precision operand, as each byte operation is independent of each other, the same as was seen for the assignment operation.

<b>In C</b>	<b>In Assembly</b>
<pre>int i, j;  i = i &amp; j;</pre>	<pre>CBLOCK 0x020   i:2, j:2 ENDC  movf j,w andwf i,f      ; i = i &amp; j (LSByte)  movf j+1,w andwf i+1,f    ; i = i &amp; j (MSByte)</pre>

**FIGURE 5.3** Bitwise AND on int operands.

### Addition/Subtraction

Figure 5.4 shows addition and subtraction using two 16-bit operands. For the addition operation, the two least significant bytes are added first, followed by the addition of the two most significant bytes, which includes the carry produced by the least significant byte addition. The subtraction is done similarly, except the subtraction of the most significant bytes includes the borrow produced by the least significant byte subtraction. To accommodate the needs of extended precision addition and subtraction, the PIC18 instruction set includes the instructions *addwfc* (add W to f with carry) and *subwfb* (subtract W from f with borrow).

<p>addition</p> <div style="text-align: right; margin-right: 20px;">             C flag              1 ←         </div> <pre style="text-align: right; margin-right: 20px;"> 0x 34 F0 + 0x 22 40 ----- 0x 57 30</pre>		<p>subtraction</p> <div style="text-align: right; margin-right: 20px;">             ~C flag = Borrow              -1 ←         </div> <pre style="text-align: right; margin-right: 20px;"> 0x 34 10 - 0x 22 40 ----- 0x 11 D0</pre>
---	--	---

**FIGURE 5.4** Addition/subtraction on 16-bit operands.

Figure 5.5 illustrates how the *addwfc/subwfb* instructions are used to implement addition/subtraction on *int* variables. For addition, the least significant bytes are added first using the *addwf* instruction, and then the *addwfc* instruction is used for the most significant byte addition. For subtraction, the least significant bytes are subtracted first using the *subwf* instruction, and then the *subwfb* instruction is used for the most significant byte subtraction. For *long* operands, four addition/subtractions are required, using *addwf/subwf* for the least significant byte and

`addwfc/subwfb` for the remaining 3 bytes. Unlike assignment and bitwise logical operations, the order of the byte operations is critical, with addition/subtraction performed least significant byte to most significant byte.

In C	In Assembly
<code>int i, j;</code>	<code>CBLOCK 0x020</code>
<code>int p, q;</code>	<code>  i:2, j:2, p:2, q:2</code>
	<code>  ENDC</code>
 <code>i = i + j;</code>	 <code>  movf j, w</code>
<code>p = p - q;</code>	<code>  addwf i, f ; i = i + j (LSByte)</code>
 <code>addwfc</code> used for MSBbyte addition	 <code>  movf j+1, w</code>
	<code>  addwfc i+1, f ; i = i + j (MSByte)</code>
	 <code>  movf q, w</code>
	<code>  addwf p, f ; p = p - q (LSByte)</code>
 <code>subwfb</code> used for MSBbyte subtraction	 <code>  movf q+1, w</code>
	<code>  subwfb p+1, f ; p = p - q (MSByte)</code>

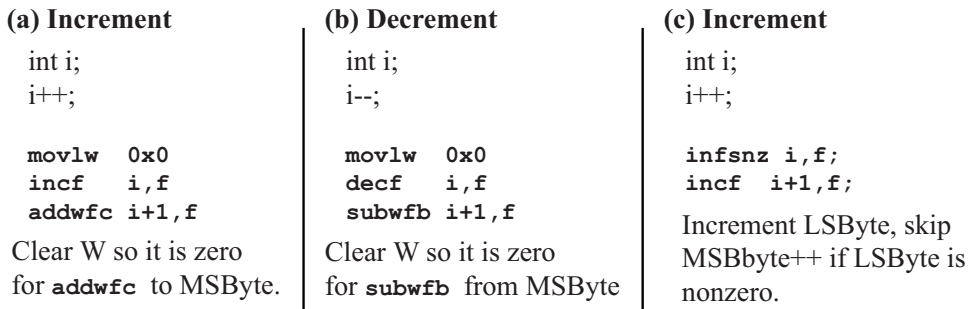
**FIGURE 5.5** Assembly language for 16-bit addition/subtraction.

Figure 5.6 shows three different methods for implementing increment/decrement on 16-bit operands (two for increment, one for decrement). Methods (a) and (b) initialize W to zero, and then use an `incf/decf` on the least significant byte followed by an `addwfc/subwfb` on the most significant byte. The W register is initialized to zero to ensure that only the Carry flag as produced by the first `incf/decf` instruction affects the result of the `addwfc/subwfb` operation. Methods (a) and (b) are general, and can be extended to any size operand. Method (c) is an optimization for 16-bit increment that uses the `infsnz` (increment f, skip if not zero) instruction on the first byte. Observe that the second byte only has to be incremented if a carry is produced by the increment operation, which only occurs if the result of the first increment is a zero. Method (c) takes one less instruction word (and hence, one less instruction cycle) to implement than method (a), but cannot be used for operands larger than two bytes.

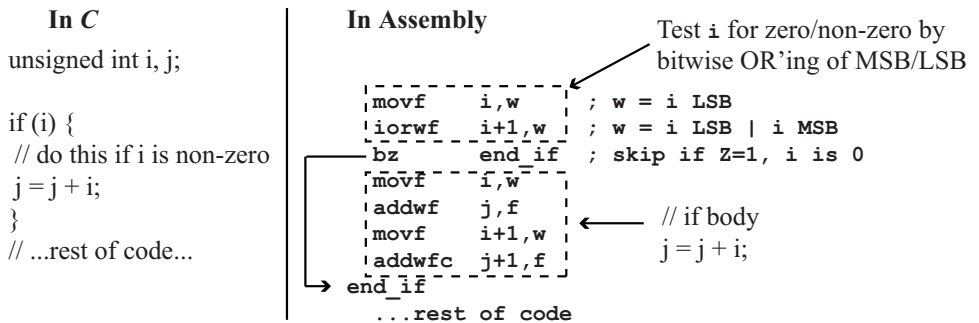
### Zero, Nonzero Conditional Tests

A 16-bit nonzero test is shown in Figure 5.7. This is the same example used in Chapter 4 with the exception that the data type has been changed from `unsigned char` to `unsigned int`. A zero/nonzero test is performed on an extended precision operand by bitwise-OR'ing of the bytes in the operand to each other. After the last

byte is bitwise-ORed, a  $Z = 1$  condition indicates an operand value of zero, as the final 8-bit result can only be zero if all bits in all bytes are zero. If the unsigned `int` operand of Figure 5.7 is changed to an unsigned `long`, the instructions `iorwf i+2,w` and `iorwf i+3,w` are required after the `iorwf i+1,w` instruction to perform the bitwise-OR of all 4 bytes of the `long` data type. If the `C` code of Figure 5.7 is changed to the zero test `if(!i){}`, the `bz` instruction is replaced with a `bnz` instruction to skip the loop body if the operand is nonzero.



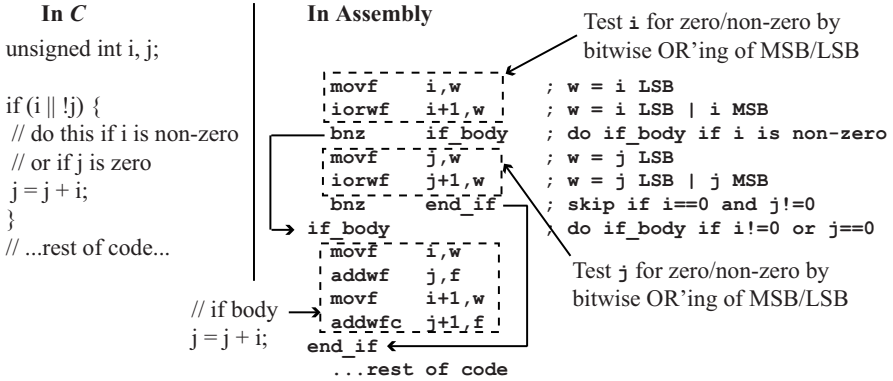
**FIGURE 5.6** Assembly language for 16-bit increment/decrement.



**FIGURE 5.7** Assembly language for 16-bit nonzero test.

A more complex zero/non-zero test is shown in Figure 5.8, in which the `if{}{}` body is executed if `i` is nonzero or `j` is zero (`i || !j`). Both the `i` and `j` variables are tested by bitwise-ORing of their most significant and least significant bytes together. The `if{}{}` body is skipped if `i` is zero and `j` is nonzero. Observe that the logical OR condition `||` in the `C` code is unrelated to the bitwise-OR used for testing each variable for zero/nonzero. If the condition in the `C` code is changed to `i && !j`

(execute the `if{}` body if `i` is nonzero and `j` is zero), a bitwise-OR operation is still used to test the `i`, `j` variables for zero/nonzero.



**FIGURE 5.8** Assembly language for 16-bit zero/nonzero test with two operands.

A common mistake is to test a 16-bit value for zero/nonzero as shown in Listing 5.1. Here, the zero/nonzero test is patterned after the method used for 8-bit operands in which the operand is copied to itself to affect the Z flag. However, this is an incorrect test, as the Z flag setting is only based on the value of the MSB, and not the combined MSB:LSB values.

**LISTING 5.1** A common mistake for zero/nonzero test.

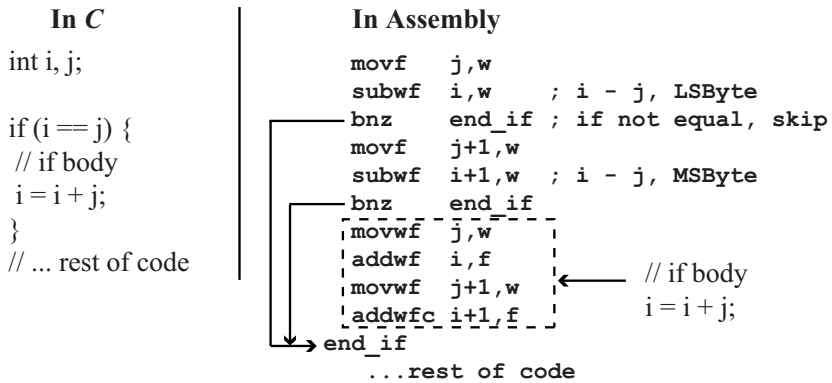
```

movf    i,f           ; test i LSB
movf    i+1,f         ; test i MSB
bz      i_is_zero     ; branch if i is zero

```

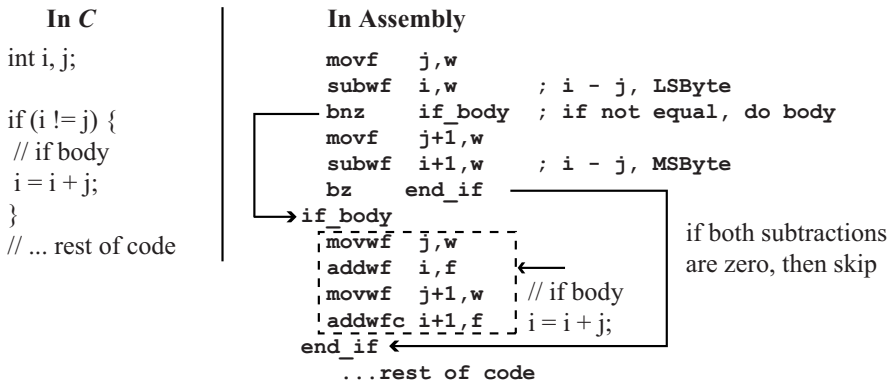
**Equality, Inequality**

Figure 5.9 shows assembly code for a 16-bit equality test. The `if_body` is executed if the test `i == j` is true. The test is performed by subtracting each byte of the two 16-bit operands, and executing the `if_body` if the Z flag is set for both subtractions, indicating that both the least significant and most significant bytes of the two operands are equal. This requires two `bnz` instructions, one for each subtraction operation. A common error is to perform the 16-bit subtraction, and only include the last `bnz` instruction. This is incorrect, as this only determines if the most significant bytes are equal, and not if the entire 16-bit operands are equal.



**FIGURE 5.9** Assembly language for 16-bit equality test.

An inequality test is shown in Figure 5.10, and the `if_body` is executed if either of the two subtractions yields a nonzero result. The first `bnz` instruction after the least significant byte subtraction branches to the `if_body`, while the second `bz` instruction branches around the `if_body`, causing the `if_body` to be skipped only for the case where both subtractions produce a zero result.



**FIGURE 5.10** Assembly language for 16-bit inequality test.

## Greater-than, Greater-than-or-equal

Figure 5.11 shows assembly code for a  $i > j$  comparison used in an `if{} statement`, where `i, j` are unsigned 16-bit operands. The only difference between this code and the 8-bit comparison code given in the previous chapter is that a 16-bit subtraction for  $j - i$  is performed for the comparison, and the `if_body` uses a 16-bit addition.

In C	In Assembly
<pre> unsigned int i, j;  if (i &gt; j) {     // done if i greater than j     j = i + j; } // ...rest of code... </pre>	<pre> movf    i,w    ; subwf   j,w    ; j - i LSByte movf    i+1,w  ; subwfb  j+1,w  ; j - i MSByte bc      end_if ; skip if C=1, i &lt;= j movf    i,w    ; addwf   j,f    ; j + i LSByte movf    i+1,w  ; addwf   j+1,f  ; j + i MSByte end_if ..rest of code.. </pre>

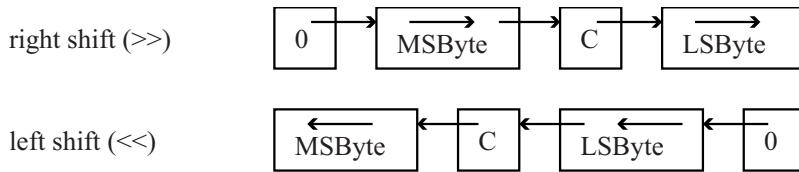
**FIGURE 5.11** Assembly language for 16-bit greater-than test using subtraction.

The Carry flag test is the same as was used for the 8-bit comparison. Similarly, if the variables are declared as `unsigned long i, j`, a 32-bit subtraction is needed for the  $j - i$  operation, while the flag test remains the same. This illustrates how the subtraction/flag test approach for comparison operations scales easily with operand size. The use of `cpfsgt` for extended precision compare can be performed by convoluted code that begins by performing a greater-than test on the most significant bytes, followed by an equality test of the most significant bytes, finally followed by a greater-than test on the least significant bytes. This approach requires more instructions, is slower, and is difficult to understand. This underscores the point that the `cpfsgt` instruction should be reserved only for comparisons of 8-bit, unsigned data.

## Shifts

Figure 5.12 illustrates how extended precision shift operations are accomplished. A left shift performs byte shift operations from least significant to most significant byte, while a right shift does the reverse, from the most significant byte to the least significant byte. In both left and right shift cases, the Carry flag is cleared for the first shift operation, but then is used as an intermediate storage bit for the shift operations on the remaining bytes.

Figure 5.13 shows examples of C left and right shift operations on `int` variables implemented in PIC18 assembly language. Extending these operations to `long` variables is an easy matter of including two additional shifts for the second and third bytes of the 4-byte operands. The use of an `unsigned int` data type for the right shift is intentional, as the right shift for signed operands is different as will be discussed in Section 5.6.



**FIGURE 5.12** Extended precision shift operations.

In C	In Assembly
int i;	
i = i << 1;	bcf STATUS,C ; clear carry rlcf i,f ; i << 1, LSByte rlcf i+1,f ; i << 1, MSByte
unsigned int i;	
i = i >> 1;	bcf STATUS,C ; clear carry rrcf i+1,f ; i >> 1, MSByte rrcf i,f ; i >> 1, LSByte

**FIGURE 5.13** Assembly language implementation of 16-bit shifts.

**Sample Question: Implement the C code fragment shown here in PIC18 assembly.**

```
unsigned int j, k;

do{
  j = j << 1;
}while(k >= j);
```

*Answer:* The comparison  $k \geq j$  is done by a 16-bit subtraction  $k - j$ . In Listing 5.2, the Carry flag is cleared before the shift of the LSByte of  $j$ , as we want a 0 shifted into the least significant bit of  $j$ . The Carry flag is set to the bit value that is shifted out of the most significant bit of  $j$ 's LSByte. The Carry flag is not cleared before the shift of  $j$ 's MSByte because we want the Carry flag to propagate this value into the least significant bit of  $j$ 's MSByte.

**LISTING 5.2** Sample question solution.

```
loop_top:
  bcf STATUS,C
  rlcf j,f ; left shift LSB of j
  rlcf j+1,f ; left shift MSB of j
  movf j,w
  subwf k,w ; w = k LSB - j LSB
  movf j+1,w
```



```

subwfb k+1,w          ; w = k MSB - j MSB
bc    loop_top       ; loop if C=1, no borrow, so k >= j
....rest of code....

```

**Sample Question: Implement the C code fragment shown here in PIC18 assembly. Remember that a long data type is 32 bits (4 bytes).**

```

long i, k;
i = i + k;

```

*Answer:* Because a long data type is 4 bytes, this requires four add operations. The LSByte add operation of Listing 5.3 uses an `addwf` instruction, while the last three add operations use an `addwfc` instruction to include the carry flag in the sum.

---

**LISTING 5.3** Sample question solution.

---

```

movf    i,w
addwf   k,f          ; first byte add (LSByte)
movf    i+1,w
addwfc  k+1,f        ; second byte add
movf    i+2,w
addwfc  k+2,f        ; third byte add
movf    i+3,w
addwfc  k+3,f        ; fourth byte add (MSByte)

```

---

## 5.4 SIGNED NUMBER REPRESENTATION

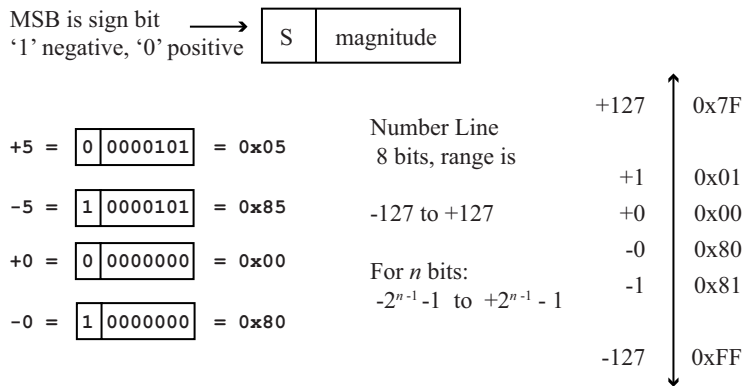
---

All examples up to this point have used unsigned data types. Obviously, we are also interested in performing operations on signed integers such as  $-100$  or  $+27$ , but to do this, we must have a binary encoding method that includes the sign (+/−) of a number and its magnitude. Three binary encoding methods for signed integers are *signed magnitude*, *one’s complement*, and *two’s complement*. These encodings share two common features, one of which is that a positive number in any of these encodings is the same, and is simply the binary representation of the number. The second common feature is that the most significant bit of a negative number is “1”.

### Signed Magnitude

Signed magnitude encoding is so named because the encoding is split into sign and magnitude, with the most significant bit used for the sign, and the remaining bits for the magnitude. Figure 5.14 shows examples of signed magnitude encoding. With  $n$  bits, the number range is  $-2^{(n-1)} - 1$  to  $+2^{(n-1)} - 1$ , or  $-127$  to  $+127$  for  $n = 8$ . Two encodings exist for zero, a positive zero and a negative zero. The advantage of

signed magnitude is that the sign and magnitude are immediately accessible for manipulation by hardware, and it is cheap from a logic gate viewpoint to produce the negative of a number (simply complement the sign bit). However, the disadvantage is that the same binary adder logic used for unsigned numbers cannot be used for signed magnitude numbers; signed magnitude arithmetic requires custom logic targeted for that encoding method. In a microprocessor, this would require special addition/subtraction instructions for use with signed magnitude integers, and instructions for converting between unsigned and signed integer representations. Fortunately, as will be seen shortly, two's complement encoding allows the same binary adder logic to be used for both unsigned and signed representations, and thus no separate instructions are needed for signed and unsigned addition/subtraction. However, there is one class of numbers, floating point numbers (i.e.,  $-10.235$ ), which do require their own dedicated arithmetic/logic units and instructions. Interestingly, a form of signed magnitude representation is used for floating point number encoding, which is discussed further in Chapter 7, "Advanced Assembly Language: Higher Math."

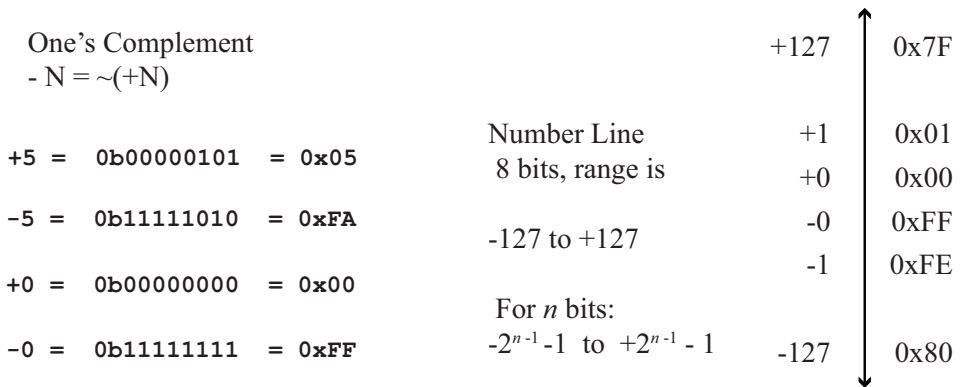


**FIGURE 5.14** Signed magnitude encoding.

## One's Complement

One's complement is so named because a  $-k$  is found by complementing each bit in the  $+k$  representation (i.e.,  $-k = \sim(+k)$ ). With  $n$  bits, the number range is  $-2^{(n-1)} - 1$  to  $+2^{(n-1)} - 1$ , or  $-127$  to  $+127$  for  $n = 8$ , the same as signed magnitude. Two encodings exist for zero, positive zero (all "0"s) and negative zero (all "1"s). The binary adder logic used for unsigned numbers can be used for one's complement numbers as long as an error of  $+1$  is acceptable in the result, which occurs if

adding two negative numbers or a positive and a negative number. For example, the correct result for the sum  $+5 + (-2)$  is  $+3$ . Written as 8-bit numbers in one's complement, the sum is  $0x05 + 0xFD = 0x02 (+2)$ , which is in error by  $+1$ . The advantage of one's complement is that the negative value of a number is cheap in terms of logic gates and fast in terms of delay; all that is needed is an inverter for each bit, producing one gate delay for the negation operation. One's complement is used within some graphics hardware accelerators for color operations on pixels, where speed is all-important and an error of 1 LSB is acceptable.

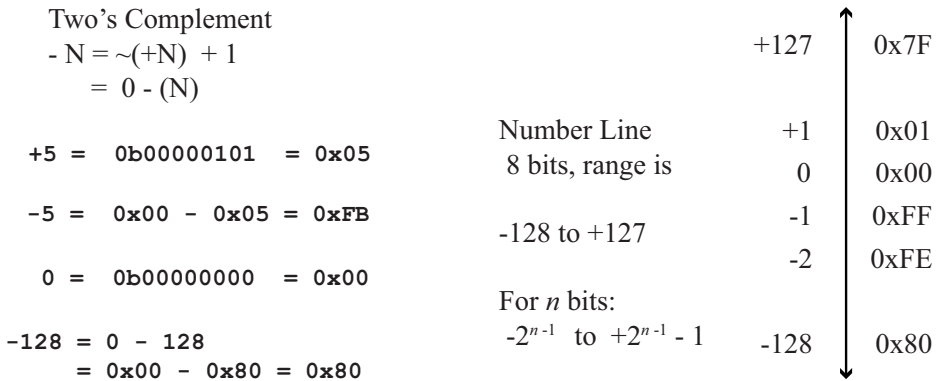


**FIGURE 5.15** One's complement encoding.

### Two's Complement

The  $+1$  error in one's complement addition using binary adder logic is corrected by encoding a  $-k$  as  $\sim(+k) + 1$ , the one's complement plus one. This is called two's complement representation, and is the standard method for encoding signed integers in microprocessors. With  $n$  bits, the number range is  $-2^{(n-1)}$  to  $+2^{(n-1)} - 1$ , or  $-128$  to  $+127$  for  $n = 8$ . There is only one representation for zero (digits are all 0s), with the negative zero (digits are all 1s) of one's complement now representing a negative one ( $-1$ ). The negative number range contains one more member than the positive range. Figure 5.16 gives examples of two's complement encodings.

As stated earlier, conversion of a positive decimal number  $+k$  to two's complement is easy, as it is simply the binary (hex) representation. The conversion of a negative decimal number to two's complement using the formula  $-k = \sim(+k) + 1$  is error prone, as it requires the number to be converted to binary, and then each bit complemented. An easier method is to use the fact that  $-k = 0 - (+k)$ , which computes  $-k$  by converting  $+k$  to hex, and then subtracting that number from zero. Figure 5.17 summarizes the rules for signed decimal to two's complement conversion and contains two sample conversions.



**FIGURE 5.16** Two's complement encoding.

If $n$ is positive	Convert $n$ to hex	+60 = 0x3C
If $n$ is negative	Ignore sign, convert $n$ to hex.  Then subtract from zero.	-60 = ?? 60 = 0x3C  -60 = 0x00 - 0x3C = 0xC4

**FIGURE 5.17** Signed decimal to two's complement conversion.

Converting a hexadecimal two's complement number to signed decimal requires first determination of the sign, and then the magnitude. If the most significant bit is one (hex digit is 8 or greater), the number is negative. To find the magnitude of this negative number, subtract the number from zero, as  $+k = 0 - (-k)$ . If the most significant bit is zero (hex digit is 7 or less), the number is positive, and converting it to decimal provides  $+k$ . Figure 5.18 summarizes these rules and shows two examples of two's complement hex to signed decimal conversion.

If MSb is 0 (hex digit < 8)	Number is positive, convert to decimal	0x4D = +77
If MSb is 1 (hex digit > 7)	Number is negative, subtract from zero, convert to decimal to find magnitude.	0xB3 = ?? 0x00 - 0xB3 = 0x4D 0x4D = 77
	Combine sign and magnitude	0xB3 = -77

**FIGURE 5.18** Two’s complement hex to signed decimal conversion.

### Sign Extension

To convert an 8-bit unsigned number to a 16-bit unsigned value, one simply adds extra zeros to the leftmost digits. As an example, the unsigned decimal number 128 in 8 bits is 0b10000000, or 0x80. In 16 bits, the same number is 0b0000000010000000, or 0x0080. For two’s complement numbers, extra precision is gained by *sign-extension*, which means the sign bit is replicated to produce the additional bits. Using the same example, the signed decimal number of -128 in 8 bits, two’s complement is 0b10000000, or 0x80. In 16 bits, the same number is 0b1111111110000000, or 0xFF80. Note that if zeros are used for padding instead of the sign bit, a negative number is changed to a positive number, an obviously incorrect result. For hex representation, extending the sign bit means padding with “F” digits for negative numbers, and “0” digits for positive numbers.

**Sample Question: Give the value of -3 as a 16-bit two’s complement number.**

*Answer:* The easiest way to accomplish this is to first write -3 as an 8-bit two’s complement number, and then sign extend. You know that +3 = 0x03, so -3 = 0 - (+3) = 0x00 - 0x03 = 0xFD. Sign extending (adding “1”s in binary or 0xF digits in hex to the left) the 8-bit value 0xFD to a 16-bit value produces 0xFFFFD. You can verify that this is correct by computing 0 - (-3) = +3, so 0x0000 - 0xFFFFD = 0x0003 = +3.

**Sample Question: The value 0xA0 is a two’s complement number; give its decimal value.**

*Answer:* The MSb of 0xA0 is a “1”, so this number is negative. We know that 0 - (-N) = +N, so 0x00 - 0xA0 = 0x60 = +96. Thus, the magnitude of the number is 96, the sign is negative, so 0xA0 = -96.

## 5.5 TWO'S COMPLEMENT OVERFLOW

In Chapter 1, “Number System and Digital Logic Review,” it was discussed that a carry out of the most significant bit is an indication of overflow for unsigned numbers. How is overflow detected for addition/subtraction of two's complement numbers? The sum  $-1 + (+1)$  written as 8-bit, two's complement numbers is  $0xFF + 0x01$ . The addition produces a carry out of the most significant bit and an 8-bit result of  $0x00$  (0), the correct result for the sum  $-1 + (+1)$ . This means that the Carry flag is not useful as an overflow indicator for signed arithmetic. Instead, the two's complement overflow flag (OV), bit 3 of the STATUS register, is the error indicator for two's complement arithmetic. In this book, the OV flag is shortened to the V flag and is referred to as the overflow flag, with two's complement understood. The negative (N) flag in the status register is set equal to the MSb of the operation result; thus, a value of “1” indicates a negative result if the result is interpreted as a two's complement number.

For addition, a two's complement overflow occurs when the sum of two negative numbers yields a positive number, or when the sum of two positive numbers yields a negative number. The sum of a positive and a negative number cannot produce overflow. Similar rules can be derived for subtraction. These rules are summarized in Equations 5.1 through 5.4. Observe that the subtraction rules 5.3, 5.4 are simply the addition rules 5.1, 5.2 stated in a different form:

$$\text{if } +p + (+q) = -r \quad \text{then } V = 1 \quad (5.1)$$

$$\text{if } (-p) + (-q) = +r \quad \text{then } V = 1 \quad (5.2)$$

$$\text{if } (+p) - (-q) = -r \quad \text{then } V = 1 \quad (5.3)$$

$$\text{if } (-p) - (+q) = +r \quad \text{then } V = 1 \quad (5.4)$$

The preceding rules aid the determination of the V flag when performing addition or subtraction manually. A method more suitable for logic gate implementation is shown by the Boolean test of Equation 5.5, where  $C_{MSb}$  is the carry out of the most significant bit,  $C_{MSb-1}$  is the carry out of the preceding bit as produced during binary addition, and  $\wedge$  is the XOR operation.

$$V = C_{MSb} \wedge C_{MSb-1} \quad (5.5)$$

Figure 5.19 illustrates the four possible cases of C, V flag settings for addition. The operands and results are shown interpreted as both unsigned numbers and signed numbers. Observe that  $C = 0$  if the unsigned result is correct. Similarly,

$V = 0$  if the signed result is correct. A natural question is “How do I know if the hex numbers in Figure 5.19 are supposed to represent signed or unsigned values?” The answer is, of course, that you do not know. There is nothing inherent in the 8-bit code  $0xFF$  that says it represents 255 or  $-1$ ; the application that uses the numbers determines if an unsigned or signed quantity is required. The binary logic performing the addition does not know if the numbers represent signed or unsigned quantities; the adder logic works equally well assuming either representation.

adder logic	unsigned	signed	adder logic	unsigned	signed
$0x01$	1	+1	$0xFF$	255	-1
+ $0xFF$	+ 255	+ -1	+ $0x80$	+ 128	+ -128
$0x00$	0	0	$0x7F$	127	+127
C=1, Z=1, V=0, N=0			C=1, Z=0, V=1, N=0		
-----					
adder logic	unsigned	signed	adder logic	unsigned	signed
$0x7F$	127	+127	$0x80$	128	-128
+ $0x01$	+ 1	+ +1	+ $0x20$	+ 32	+ +32
$0x80$	128	-128	$0xA0$	160	-96
C=0, Z=0, V=1, N=1			C=0, Z=0, V=0, N=1		

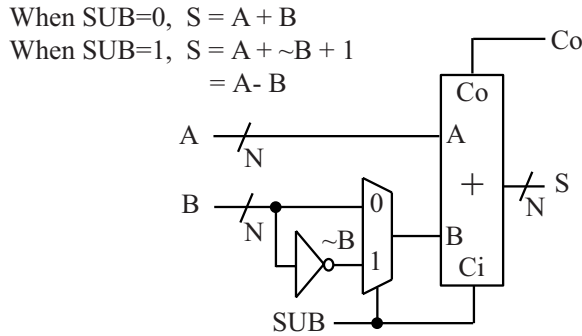
**FIGURE 5.19** Four cases of C, V flag settings.

**Sample Question:** Give the V, N flag settings after the operation  $0x60 + 0x40$ .

**Answer:**  $0x60 + 0x40 = 0xA0$ . The MSb of  $0xA0$  is 1, so  $N = 1$  (result is negative). Two positive numbers added together produce a negative number, so  $V = 1$ .

## 5.6 OPERATIONS ON SIGNED DATA

In Chapter 4, “Unsigned 8-bit Arithmetic, Logical, Conditional Operations,” you learned in the Carry flag discussion that the subtraction  $A - B$  is actually performed as  $A + (\sim B) + 1$ , for which the reasoning is now clear as  $-B = \sim B + 1$  by the definition of two’s complement. Figure 5.20 illustrates how a combinational building block capable of both addition and subtraction is built from an adder and a 2-to-1 mux. The SUB input connected to the mux select and the carry-in input of the adder determines if the operation  $A + B$  or  $A - B$  is performed. When  $SUB = 0$ , the output is  $Y = A + B + 0$ ; when  $SUB = 1$ , the output is  $A + (\sim B) + 1$ , which is actually  $A - B$ . An adder/subtractor building block is a key component of the arithmetic logic unit of any microprocessor. Figure 5.20 clearly shows the



**FIGURE 5.20** Adder/subtractor building block.

advantage of two's complement representation for signed numbers; the same binary adder logic used for addition/subtraction of unsigned numbers is used for signed numbers.

Unfortunately, some operations on signed numbers require different hardware, and thus, different assembly language instruction sequences. Table 5.2 lists the arithmetic operations discussed in this book, and whether different hardware is required for unsigned and signed operations.

Shift and comparison operations on signed data are discussed in this chapter, while multiplication and division are postponed until Chapter 7. In C code, the signed qualifier is used in front of data types; for example, signed int or signed char, to declare signed variables. If a signed or unsigned qualifier is not given for a data type, the normal assumption made by a compiler is that it is a signed data type.

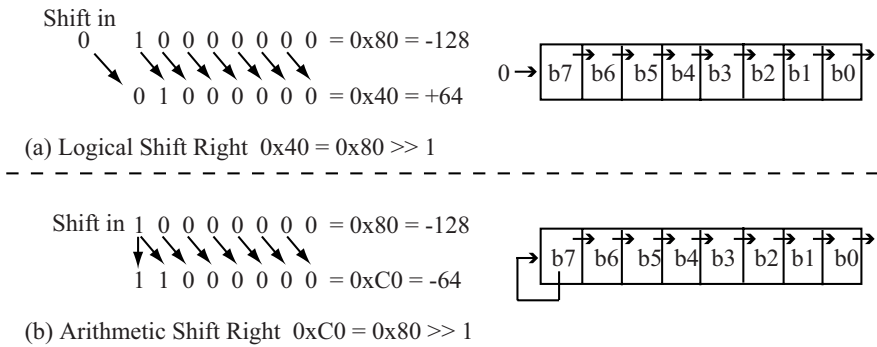
**TABLE 5.2** Unsigned versus Signed Arithmetic Operations

Operation	Same Operation for Unsigned/Signed Data?
+, -	yes
==, !=	yes
<< (left shift)	yes
>, >=, <, <=	no
>> (right shift)	no
*, /	no



### Shift Operations on Signed Data

In previous shift operation examples, a zero value has always been used as the shift-in value for the LSb (left shift) or MSb (right shift). Assume a right shift of the value  $-128$  is desired, which should yield  $-64$  as a right shift is a divide-by-two operation. The value  $-128$  is  $0x80$  as an 8-bit two's complement number, and Figure 5.21a shows that this becomes a  $0x40$ , or a  $+64$ , when a zero is used as the MSb input value. In Figure 5.21b, the sign bit is retained during the shift, keeping the MSb as one, which produces the correct value of  $-64$  or  $0xC0$ . For two's complement values, the sign bit must be retained during a right shift to obtain the correct arithmetic result of division-by-two. This is sometimes called an *arithmetic shift right*, while a right shift that always shifts in a zero is called a *logical shift right*.



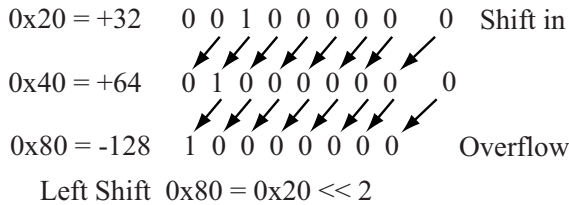
**FIGURE 5.21** Right shift, logical versus arithmetic.

Figure 5.22 shows assembly code that implements an arithmetic shift right on a 16-bit operand. Note that the Carry flag is set equal to the sign bit of the most significant byte before the shift of the most significant byte is performed, causing the sign bit to remain the same. The arithmetic shift right operation must be used in this case because the variable `i` is declared in the `C` code as a signed `int`, indicating that a signed shift right should be used.

It would seem logical that the same reasoning would apply to a left shift operation; that the sign bit has to be retained for a left shift of a two's complement operand. Figure 5.23 illustrates what happens when the value  $0x20$  ( $+32$ ) is shifted to the left two times. After the first shift, the new value is  $0x40$  ( $+64$ ), the correct value in an arithmetic sense as  $+64 = 2 * (+32)$ . After the second shift, the new value is  $0x80$  ( $-128$ ), which is obviously incorrect from an arithmetic sense as the sign changed. However, if the sign bit is kept during the shift, the result is  $0x00$ ,

In C		In Assembly
signed int i;		bcf STATUS,C ; clear carry
		btfsc i+1, 7 ; sign bit=0?
i = i >> 1;		bsf STATUS,C ; set carry
		rrcf i+1,f ; i >> 1, MSByte
		rrcf i,f ; i >> 1, LSByte

**FIGURE 5.22** Assembly code for arithmetic shift right operation.



**FIGURE 5.23** Left shift operation on signed data.

again an incorrect result in an arithmetic sense. This is because  $+64 * 2 = +128$ , and  $+128$  cannot be represented in 8 bits! If the sign bit changes after a left shift of a two's complement value, overflow has occurred. Keeping the sign bit does not help, as it is not possible to represent the correct arithmetic value using the available bits. As such, there is no need to distinguish left shift operations based on unsigned versus signed data.

### Greater-than, Greater-than-or-equal on Signed Data

Recall that the unsigned comparison tests of  $>$  and  $\geq$  use the subtraction operation on the two operands being compared, followed by a test of the Carry flag. Figure 5.24 illustrates what happens if two signed 8-bit operands are treated as unsigned operands for comparison purposes. Only if the signs of the operands are the same, does the unsigned comparison operation yield the correct result. One could envision writing code to check the signs of the operands before doing the comparison, setting a temporary flag to indicate sign equality or inequality, and then performing the appropriate comparison based on the flag setting. However, this is overly complex and not required, as there is an easier way involving subtraction and flags.

Table 5.3 lists the operation and flag tests for  $>$  and  $\geq$  comparisons for signed data. The subtraction operation used is the same as for unsigned data; the difference is that the V (overflow) and N (negative) flags are used instead of the C (carry) flag. The flag tests look complicated, but reasoning about them makes the conditions easier to remember. For the greater-than ( $>$ ) test, if  $i$  is indeed greater than  $j$ ,

the subtraction  $j - i$  should be a negative number ( $N = 1$ ), with  $V = 0$  indicating the correctness of the result. However, if overflow occurs ( $V = 1$ ), a positive number ( $N = 0$ ) is obtained, which is an obviously incorrect result caused by the overflow. The astute reader will recognize this flag test as  $V \wedge N$  (the exclusive-OR of the  $V$  and  $N$  flags), which is how this test is implemented in logic gates for microprocessors that have signed comparison instructions.

Numbers	As Unsigned	$i > j?$	As Signed	$i > j?$
$i = 0x7F,$ $j = 0x01$	$i = 127,$ $j = 1$	True	$i = +127,$ $j = +1$	True
$i = 0x80,$ $j = 0xFF$	$i = 128,$ $j = 255$	False	$i = -128,$ $j = -1$	False
$i = 0x80,$ $j = 0x7F$	$i = 128,$ $j = 127$	True	$i = -128,$ $j = +127$	False
$i = 0x01,$ $j = 0xFF$	$i = 1,$ $j = 255$	False	$i = +1,$ $j = -1$	True

**FIGURE 5.24** Unsigned versus signed comparisons.

**TABLE 5.3** Greater-than, Greater-than-or-equal Comparisons Using  $V, N$  Flags

Comparison	Operation	If True, Then
$\text{if } (i > j) \{ \}$	$j - i$	$V = 0$ and $N = 1$ OR $V = 1$ and $N = 0$
$\text{if } (i \geq j) \{ \}$	$i - j$	$V = 0$ and $N = 0$ OR $V = 1$ and $N = 1$

For the greater-than-or-equal ( $\geq$ ) test, if  $i \geq j$  is true, the subtraction  $i - j$  should be a positive number ( $N = 0$ ), with  $V = 0$  indicating the correctness of the result. However, if overflow occurs ( $V = 1$ ), a negative number ( $N = 1$ ) is obtained, which is an obviously incorrect result caused by the overflow. This flag test is implemented in digital logic as  $\sim(V \wedge N)$  for microprocessors that have signed comparison operations.

Figure 5.25 shows the assembly code for a 16-bit comparison operation  $i > j$  used as a conditional test within an `if{}` statement. A 16-bit subtraction of  $j - i$  is performed first, followed by branch code that implements the test of  $(\sim V \ \& \ N) \mid (V \ \& \ \sim N)$ . The multiple branch paths are unavoidable given the flag conditions that must be checked. Instructions that directly test the condition  $V \ \wedge \ N$ , and similar conditions for other types of signed comparisons, are often included in microprocessor instruction sets and are typically called *signed branches*. The assembly code for a greater-than-or-equal comparison is left as an exercise in the review problems.

In C	In Assembly
<code>signed int i,j;</code>	<code>movf i,w ;</code>
	<code>subwf j,w ; j-i LSByte</code>
<code>if (i &gt; j) {</code>	<code>movf i+1,w</code>
<code>  // if_body</code>	<code>subwfb j+w,w ; j-i MSByte</code>
<code>  i = i + j;</code>	<code>bov v_1</code>
<code>}</code>	<code>bnn end_if ; skip if V=0,N=0</code>
<code>  // rest of code</code>	<code>bra if_body ; V=0, N=1</code>
	<code>bn end_if ; skip if V=1,N=1</code>
	<code>if_body ←</code>
	<code>movf j,w</code>
	<code>addwf i,f ; i=i+j, LSByte</code>
	<code>movf j+1,w</code>
	<code>addwfc i+1,w ; i=i+j, MSByte</code>
	<code>end_if</code>
	<code>...rest of code...</code>

**FIGURE 5.25** Assembly for 16-bit greater-than comparison.

## Sign Extension

Even for operations like addition and subtraction that work the same for both unsigned and signed operands, care must be taken when dealing with operands of different precisions. The precision of the smaller operand must be extended to match the precision of the larger operand. This means padding with zeros for unsigned operands, and extending the sign bit for signed operands as discussed in Section 5.4. Figure 5.26 shows an example of an addition with 16-bit ( $i$ ) and 8-bit ( $j$ ) operands, for both unsigned and signed cases. In the unsigned case, the most significant byte of the 8-bit operand is set to zero by the `movlw 0` instruction. In the signed case, the `btfscc j,7` instruction is used to test the sign bit of the 8-bit operand. If the 8-bit operand is positive, the most significant byte is set to zero; else the most significant byte is set to `0xFF` to sign extend the negative 8-bit operand.

In C	In Assembly
unsigned int i; unsigned char j;  i = i + j;	<pre> movf    j,w        ; clear carry addwfc  i,f        ; i=i+j, LSByte movlw   0          ; MSB of j = 0 addwfc  i+1,f      ; i=i+j, MSByte                     </pre>
signed int i; signed char j;  i = i + j;	<pre> movf    j,w        ; clear carry addwfc  i,f        ; i=i+j, LSByte movlw   0          ; MSB of j = 0 btsfc   j,7        ; check sign of j movlw   0xFF       ; MSB of j= 0xFF addwfc  i+1,f      ; i=i+j, MSByte                     </pre> <p style="margin-left: 40px;">j is negative, so sign extend MSB with 0xFF</p>

**FIGURE 5.26** Addition operation for operands of unequal precisions.

**Sample Question: Implement the C code fragment shown here in PIC18 assembly.**

```

signed int j, k;
do{
    j = j << 1;
}while(k >= j);
    
```

*Answer:* This is the same as a sample question from Section 5.3, except the data type has been changed from unsigned int to signed int. This does not change the code used for the << shift operation, but it does change the code for the >= comparison. In Listing 5.4, the subtraction k - j is still performed, but the branch is done based on the V, N flags. If k >= j is true, k - j produces a positive number (N = 0, V = 0) unless overflow occurs, in which case a negative result is produced (N = 1, V = 1).

**LISTING 5.4** Sample problem solution.

```

loop_top:
    bcf    STATUS,C
    rlcf   j,f          ; left shift LSB of j
    rlcf   j+1,f        ; left shift MSB of j
    movf   j,w
    subwfb k,w          ; w = k LSB - j LSB
    movf   j+1,w
    subwfb k+1,w        ; w = k MSB - j MSB
    bov    L1
    bnn   loop_top ;true loop top
    bra   loop_exit ;exit
L1
    
```

```

bn    loop_top ;true loop top
loop_exit
....rest of code....

```

**Sample Question:** In the following code, what is the value of *i* when the loop is exited?

```

signed char i;

i = 0x80;
while (i < -32) {
    i = i >> 1;
}

```

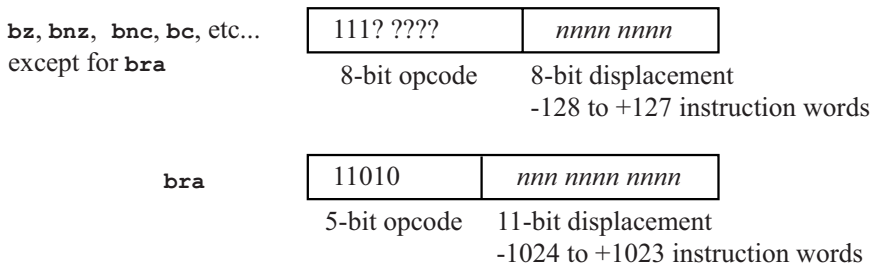
*Answer:* The variable *i* is declared as a signed char, so the assignment *i* = 0x80 initializes *i* to -128. Each time through the loop, *i* is divided by 2 by the right shift. Therefore, *i* is -128, then -64, then -32, at which point the comparison is no longer true and the loop is exited, with *i* = -32 (0xE0).

## 5.7 BRANCH INSTRUCTION ENCODING

The machine code format of branch instructions uses an addressing mode known as *Program Counter Relative*, in which a two’s complement offset is added to the program counter to determine the target branch address. The target address, *Taddr*, is computed by Equation 5.6.

$$Taddr = PC + 2 + 2 * n \tag{5.6}$$

The PC value in Equation 5.6 is the branch instruction location, and *n* is the displacement encoded in the branch instruction word. The +2 is added to the PC value because the PC is already incremented to the next instruction word when the branch target address is computed. See Figure 5.27.



**FIGURE 5.27** Machine code format of branch instructions.

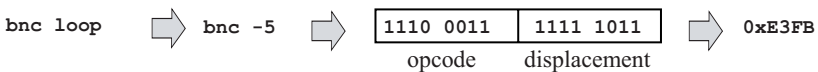
The displacement  $n$  is multiplied by 2 because the displacement is the number of instruction words, and each instruction word is 2 bytes. When determining the machine word of a branch, the locations of the target address and the branch are known, and the displacement is calculated using Equation 5.7.

$$n = \frac{(Taddr - (PC + 2))}{2} \tag{5.7}$$

Figure 5.28 shows how to calculate the displacement of a branch instruction. The `loop_top` label (`0x0100`) is the target address, while the branch location is `0x0108`. The branch displacement is calculated as a  $-5$ , or `0xFB` in 8-bit, two's complement format. This example shows why the displacement is a signed number; backward branch targets require a negative displacement, while forward branches use a positive displacement. It is also evident that a branch target must be within the range of the displacement that can be encoded in the branch instruction word. A `bra` has an 11-bit displacement ( $-1024$  to  $+1023$  instruction words), while all other branches have an 8-bit displacement ( $-128$  to  $+127$  instruction words). The advantage of branch instructions over a `goto` instruction is that a branch instruction takes only one instruction word, while a `goto` takes two instruction words. The disadvantage of a branch instruction is its limited range; a `goto` instruction can jump anywhere in program memory. Fortunately, a branch's limited range is typically not a problem, as most loops tend to be short. Program counter relative addressing is found in almost all microprocessor instruction sets.

location (hex)	machine code (hex)	instruction
0100	5000	<code>loop movf i,w</code>
0102	2601	<code>addwf j,f ; j = j+1</code>
0104	5000	<code>movf i,w</code>
0106	5C01	<code>subwf j,w ; j-i</code>
0108	E3FB	<code>bnc loop ; branch to top</code>

$$\begin{aligned}
 \text{branch\_target} &= \text{PC} + 2 + 2 * n && \quad 0x0100 \quad (\text{branch\_target}) \\
 n &= [\text{branch\_target} - (\text{PC} + 2)] / 2 && \quad - 0x010A \quad (\text{PC} + 2) \\
 &= [0x0100 - (0x0108 + 2)] / 2 && \quad 0xFFFF6 \\
 &= 0xFFFF6 / 2 = 0xFFFF6 \gg 1 \\
 &= 0xFFFFB = 0xFB \text{ (8 bits)} = -5 \text{ (displacement)}
 \end{aligned}$$



**FIGURE 5.28** Branch displacement calculation example.

## SUMMARY

---

Extended precision operations allow manipulation of arbitrarily sized data, 1 byte at a time. It is clear that variables of type `int` and `long` should not be used unless the extra precision offered by these data types is needed, as calculations on these data types require more instructions and longer execution time. The standard method of signed integer representation for microprocessors is two's complement format, which uses the same binary adder logic for addition and subtraction as used for unsigned numbers. Some operations, like right shift, greater-than, and greater-than-or-equal, require different instruction sequences for signed integers than what is used for unsigned integers. The V (overflow) and N (negative) flags are used for greater-than and greater-than-or-equal comparisons of two's complement integers. Branch instructions use program counter relative addressing, which means that a displacement value is added to the current PC value to determine the target location of a branch.

## REVIEW PROBLEMS

---

Convert the following C code segments to PIC18 instruction sequences. Assume that `i`, `j`, `k` are all unsigned `int` data types and assigned somewhere in locations 0x000 to 0x07F (the lower half of the access bank, so you do not need to be concerned with the BSR value). If you need to use other temporary memory locations in your solution, also place these in locations 0x000 to 0x07F.

1. Code fragment:

```
do {
    i = i - k;
}
while (i < (j + k));
```

2. Code fragment:

```
if (i && j) {
    k = k & 0xFF00;
}
```

3. Code fragment:

```
k = j | i;
```



4. Code fragment:

```
while (i != j) {
    k = k >> 1;
    j--;
}
```

Perform the indicated conversions:

5. The value  $-42$  to 8-bit two's complement.
6. The 8-bit two's complement value  $0xDC$  to decimal.
7. The 12-bit two's complement value  $0xBA3$  to decimal.
8. The value  $-390$  to 16-bit two's complement.
9. Sign extend the 8-bit value  $0x85$  to 16 bits.

Do the following calculations:

10. Give the value of the operation  $0x73 + 0x65$ , and the Z, N, V, C flag settings.
11. Give the value of the operation  $0x90 - 0x8A$ , and the Z, N, V, C flag settings.
12. Give the value of the operation  $0xF0 + 0xCA$ , and the Z, N, V, C flag settings.
13. Give the value of the operation  $0x2A - 0x81$ , and the Z, N, V, C flag settings.
14. In the following code segment, what is the value of  $i$  when the loop is exited?

```
signed char i, j;
i = 0x01; j = 0x80;
while (i > j) i++;
```

15. In the following code segment, what is the final value of  $k$ ?

```
signed char i, j;
i = 0xA0; j = 0x70;
k = (i > j);
```

16. In the following code segment, what is the final value of  $k$ ?

```
unsigned char i, j;
i = 0xA0; j = 0x70;
k = (i > j);
```

17. In the following code segment, what is the final value of `i`?

```
signed char i;
i = 0xA0 >> 2;
```

For the following problems, assume that `i`, `j`, `k` are all `signed char` data types and assigned somewhere in locations `0x000` to `0x07F`. Convert the following C code segments to PIC18 instruction sequences.

18. Code fragment:

```
do {
    i = i - k;
}
while (i < (j + k));
```

19. Code fragment:

```
if (k >= j) {
    i = i >> 2;
}
```

Answer the following questions:

20. What is the machine code for the instruction `bnc 0x0300` if the PC of the `bnc` instruction is `0x340`?
21. Assume you want to do a `bc` there, but the location of there causes the branch displacement to exceed the range of the `bc` instruction. What equivalent instruction sequence can be used that works regardless of the location of there?

*This page intentionally left blank*

# 6

# Subroutines and Pointers

## In This Chapter

- Subroutines
- The Stack and Call/Return
- Implementing Subroutines in Assembly Language
- Arrays and Pointers in C
- Arrays and Pointers in Assembly Language
- Accessing Table Data from Program Memory
- Subroutines and Stack Frames: Dynamic Allocation

This chapter examines the architectural features of the PIC18 that support subroutines and pointers, which are important capabilities of high-level programming languages.

## 6.1 LEARNING OBJECTIVES

---

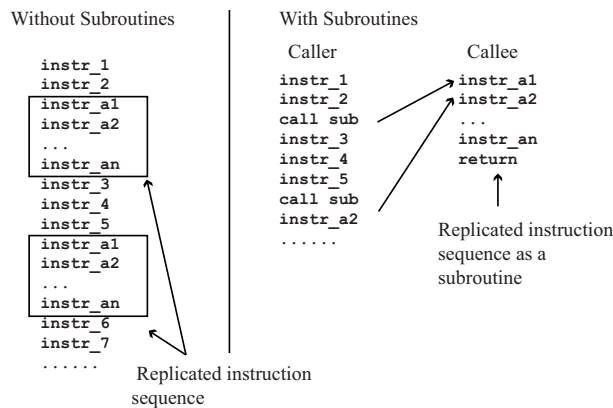
After reading this chapter, you will be able to:

- Implement subroutines in the C programming language and discuss features such as parameter lists, local variables, and return values.
- Discuss the operation of a stack data structure and its role in implementing subroutine call and return.

- Translate C subroutines with parameter lists, local variables, and return values into PIC18 instruction sequences.
- Compare and contrast static allocation versus dynamic allocation for subroutine memory requirements and implement PIC18 subroutines using either approach.
- Discuss the implementation of pointers in the C programming language.
- Use the FSRn/INDFn special function registers of the PIC18 to implement C pointers.
- Translate C code with array indexing into PIC18 instruction sequences.
- Use PIC18 table read instructions to transfer data from program memory to data memory.

## 6.2 SUBROUTINES

A *subroutine* is a block of code that is *called* from different locations within the main program or other subroutines. Instead of duplicating a commonly used instruction sequence in multiple locations, the instruction sequence can be encapsulated as a subroutine and then called from the main program. Using subroutines reduces code size, as the subroutine resides in only one place in program memory instead of multiple locations. This also improves code clarity, and produces code that is easier to maintain, as any code modifications are only performed in the subroutine body instead of in each of the duplicated code sections. Figure 6.1 illustrates this concept. The main program or other subroutine that calls a subroutine is known as the *caller*, while the subroutine being called is known as the *callee*.



**FIGURE 6.1** Use of subroutines saves code space.

The basic form of a C subroutine and a specific example is seen in Figure 6.2. In C, the preferred name for a subroutine is *function*, and these two terms are used interchangeably in this book. The example subroutine name is `vlshift()`, and it computes a variable left shift operation described as `v << amt`. It is legal in C to write this as a single statement instead of as a subroutine, but a subroutine is used here for illustrative purposes. Subroutines have distinct components that are defined as follows (it is not necessary for a subroutine to have all of these components):

**Parameter list:** Some subroutines are a fixed set of instructions that performs the exact same operation each time they are called. However, a subroutine can also have parameters that alter the subroutine behavior based on their values. The `vlshift()` subroutine in Figure 6.2 has two parameters named `v` and `amt`, both of type `unsigned char`.

**Local variables:** A subroutine may need additional variables that are used internally to perform its function. In C, these variables are declared within the subroutine, and are only visible to the subroutine itself. The `vlshift()` subroutine in Figure 6.2 does not have local variables.

**Return value:** In C, a subroutine may return a single value to the caller, by means of the *return* statement. The `vlshift()` subroutine returns a value of type `unsigned char` to the caller. It is not required that C subroutines contain an explicit return statement; an implicit return is done when the end of the subroutine body is reached.

General form of a C  
subroutine is:

```
(return_type) subname (parm list)
{
    local_variable_decl;
    subroutine_body;
    return(return_value);
}
```

```
vlshift Subroutine
// variable left shift
unsigned char vlshift(unsigned char v,
unsigned char amt)
{
    while (amt) {
        v = v << 1;
        amt--;
    }
    return(v);
}

main(void) {
    unsigned char i,j,k;

    i=0x24; j = 2;
    k = vlshift(i,j);
    printf(
        "i=0x%x, shift amount: %d,result: 0x%x\n",
        i,j,k);
}
```

Annotations in the original image:

- parameter list: gives types and names (points to `unsigned char v, unsigned char amt`)
- subroutine body (points to the `while` loop)
- subroutine return (points to `return(v);`)
- main program (points to `main(void) {`)
- subroutine call (points to `k = vlshift(i,j);`)



**FIGURE 6.2** C subroutine example.

The `v1shift()` subroutine uses a counting loop that is executed `amt` number of times, where `v` is shifted to the left by one ( $v = v \ll 1$ ) each time through the loop. The new value of `v` is returned to the caller by the statement `return(v)`. The `main()` code in Figure 6.2 uses the local variables `i`, `j` as the parameter values for `v`, `amt` in the statement `k = v1shift(i,j)` that calls the `v1shift()` subroutine. The assignment operator of the subroutine call copies the return value of `v1shift()` to the local variable `k`. The C language semantics define that parameters declared in this manner are not modified by the subroutine; the variables `i` and `j` in `main()` are unaffected by the subroutine call. The `printf()` statement in `main()` is a formatted print statement included for example purposes so that you can compile this program with a C compiler on a personal computer, and observe the input parameters and return value. The `printf()` statement is not implemented when we translate this to PIC18 assembly code; see Appendix D, “Notes on the C Language,” for more details on `printf()` syntax.

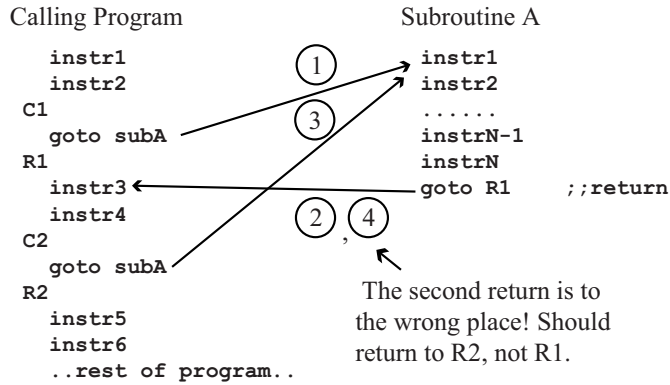
### 6.3 THE STACK AND CALL/RETURN

---

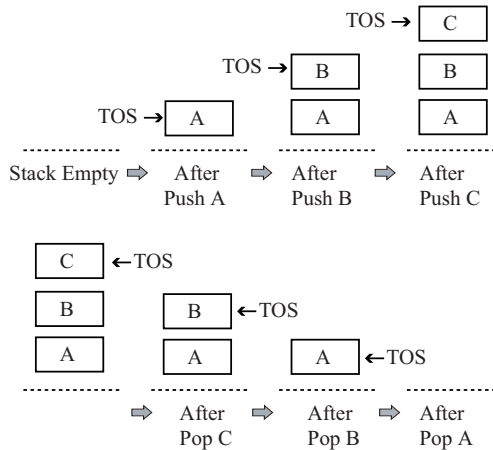
A subroutine *call* is a jump to the first instruction of the subroutine, while a subroutine *return* is a jump back to the instruction in the caller following the subroutine call. The location returned to by a subroutine return is known as the *return address*. Figure 6.3 shows the problem with implementing subroutine call and return by use of `goto` instructions. Subroutine A is called twice from within the calling program, once from label `c1` and once from label `c2`. Labels `R1` and `R2` mark the return addresses. The first call (1) and return (2) to subroutine A work as intended. However, while the second call (3) also works correctly, the return (4) is incorrect, as location `R1` is the return address instead of `R2`. Clearly, a mechanism other than a `goto` instruction is needed to implement call and return, as the return address depends on the call location. On the PIC18, the `call` and `rcall` instructions implement subroutine call, while the `return` and `retlw` instructions implement subroutine return.

What is needed is a method for saving the return address for later use by the subroutine return statement. A *stack* data structure is a commonly used mechanism in microprocessors for saving return addresses of subroutine calls. One way to visualize stack operation is by a stack of boxes, in which boxes are placed (stacked) sequentially on top of each other. Figure 6.4 illustrates a three-box stack, in which box A is placed first, then box B, and finally box C. Observe that at each step, only the box at the *top of the stack* (TOS) is accessible. Removing boxes from the stack is done in reverse order; first box C, then box B, and finally box A. Placing an item on the stack is referred to as a *push* operation, while removing an item is known as a *pop* operation. A stack is *empty* if it contains no items; a stack is *full* if another item

cannot be pushed onto the stack. Another common name for a stack is a *last-in, first-out* (LIFO) data structure, as the name describes the order in which data is accessed.



**FIGURE 6.3** Implementing call/return with goto.

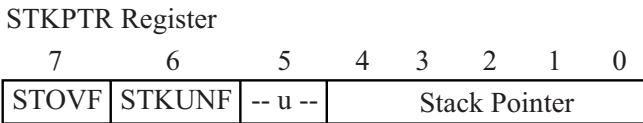


**FIGURE 6.4** Stack example.

A stack data structure needs a set of memory locations for storing the items of the stack, and a *stack pointer* (SP) that contains the address of the memory location that is the current top of the stack. Most microprocessors have a special register dedicated for the stack pointer; the PIC18 uses a special function register named STKPTR. Many microprocessors use data memory for stack storage. While this can be done on the PIC18 for creating custom stacks (see Section 6.8), the locations



accessed by STKPTR are in a special memory known as the *return address stack*. The return address stack has 31 locations, with each location containing 21 bits (the return address stack is a 31 x 21 memory). The STKPTR register is an 8-bit register, with the lower 5 bits (the stack pointer) used to reference the 31 locations of the return address stack. The upper 2 bits of the STKPTR register are status bits, named STKOVF (stack overflow, bit 7) and STKUNF (stack underflow, bit 6). The two status bits are discussed later in this section. Figure 6.5 shows the STKPTR register.



- STOVF : stack overflow (set to 1 on overflow)
- STKUNF: stack underflow (set to 1 on underflow)
- u -- : unimplemented
- Stack Pointer (SP): 5-bit stack pointer, cleared to 0 on reset

STOVF, STKUNF can only be cleared by user or by a power on reset

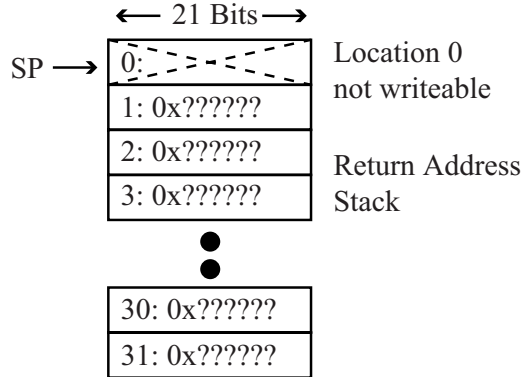
**FIGURE 6.5** STKPTR register.

As the name indicates, the return address stack is used by a subroutine call to save the return address. Each data location in the stack is 21 bits wide because the PC register is 21 bits, and the PC contains the return address when a call to a subroutine is made. A call to a subroutine pushes the return address on the stack, and then loads the PC with the starting address of the subroutine. A subroutine return pops the return address from the stack and places it in the PC. Figure 6.6 shows the PIC18 return address stack, and the mechanics of stack push and pop operations. On processor reset, the stack pointer bits (SP) of the STKPTR register are set to zero. A stack push first increments SP, and then stores the data at the stack location referenced by SP (the top-of-the-stack). This means that location 0 of the stack is never actually used for storing data. A stack pop first accesses the data in the stack location referenced by SP, and then decrements SP. The register transfer language  $PC \leftarrow ((SP))$  for pop uses two levels of parentheses around SP because the contents of the SP bits (STKPTR[4:0]) are not transferred to the PC, but rather the contents of the return address stack location *pointed to* by the SP bits are transferred to the PC. This is a form of *indirect addressing* and SP is known as a *pointer* because it is used to point to a memory location. Indirect addressing and pointers are discussed in more detail later in this chapter.

SP = STKPTR[4:0]  
 push:  
 SP++, (SP) ← nPC  
 (nPC is PC of instruction following  
 call or rcall instruction)

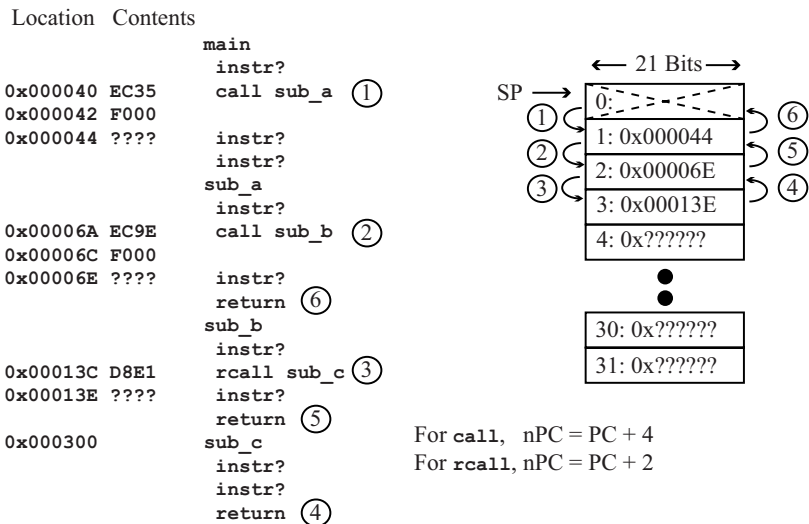
pop:  
 PC ← ((SP)), SP--

SP points to top-of-stack



**FIGURE 6.6** The PIC18 return address stack.

Figure 6.7 shows how the return address stack is modified during subroutine call and return. The call sub\_a instruction pushes the return address 0x000044 (PC + 4, or 0x000040 + 4) on the stack. The return address for a call instruction is PC + 4 because the call instruction requires two instruction words to encode the full 21-bit target address and the opcode in the same manner as the goto instruction. The subsequent call sub\_b instruction pushes the return address 0x00006E on the stack; this is an example of a *nested* subroutine call in which one subroutine calls another. The final subroutine call implemented by the rcall sub\_c instruction



For call, nPC = PC + 4  
 For rcall, nPC = PC + 2

**FIGURE 6.7** Example of return address stack usage during subroutine call/return.

pushes the return address 0x00013E on the stack, or PC+2. The `rcall` instruction uses PC relative addressing, the same addressing mode used by branch instructions, and requires only one instruction word. The first `return` instruction executed is the one in subroutine C, which pops the value 0x00013E from the stack, returning to subroutine B. Subsequent `return` instructions in subroutine B, then subroutine A are executed, finally returning to location 0x000044 in the `main()` code. At this point, the SP value is at its original value of zero, and the stack is in the empty condition. Observe that a stack pop does not actually remove or modify the contents of the return address stack; the stack locations 1 through 3 still contain the values 0x000044, 0x00006E, and 0x00013E. A stack pop only modifies the SP and PC values. However, future subroutine calls in the `main()` code after the initial `call sub_a` instruction will overwrite the stack contents with the new return addresses pushed by the `call/rcall` instructions.

After 31 `call/rcall` instructions without a `return` instruction, the SP value is 31, and the stack is full as there are no locations remaining to store new return addresses. If another subroutine call is made, stack *overflow* occurs, setting the STKOVF bit. Depending on the processor configuration, either the processor resets itself, clearing the lower 5 bits of STKPTR but leaving STKOVF set, or the processor continues operation but does not allow further pushes onto the stack. Configuration bits stored in program memory control processor configuration; this is covered in more detail in Chapter 8, “The PIC18Fxx2: System Startup and Parallel Port IO.” Stack *underflow* occurs if an attempt is made to pop a value from the stack via a subroutine return when the SP value is zero. Stack underflow sets the STKUNF bit, and clears the PC to zero, causing the reset code to be executed. The STKOVF and STKUNF status bits can be checked by the reset code to determine if reset was caused by one of these error conditions. Once a STKOVF or STKUNF status bit is set, it can only be cleared by either a user instruction, such as a `bcf`, or by a power-on reset.

There are instructions named `push` and `pop` in the PIC18 instruction set that allow modification of the return address stack outside of the normal subroutine call and return instructions. This is advanced usage of the return address stack, and is not covered in this book. However, creating user-defined data stacks in the file registers can be useful, and is covered in Section 6.8.

**Sample Question:** What value is pushed on the stack by the `call 0x0200` in this code fragment?

Location:	Contents	Instruction
0x03A4	EC80 F001	<code>call 0x200</code>
0x03A8	2A40	<code>incf 0x040,f</code>

**Answer:** The return address is the address of the instruction following the `call` instruction, so the value 0x03A8 is pushed on the return address stack.

## 6.4 IMPLEMENTING SUBROUTINES IN ASSEMBLY LANGUAGE

The `call`, `return`, `rcall`, and `retlw` instruction formats are shown in Figure 6.8. As noted earlier, the `call` instruction requires two instruction words to encode the opcode and 21-bit target address. Both the `call` and `return` instructions contain a special bit named *s* that is known as the *fast call/return mode select bit*. An *s* bit value of “1” causes the *W*, *BSR*, and *STATUS* registers to be stored in *shadow registers* when the call is made (a *fast call*). The shadow registers are also known as the *fast register stack*, and are useful for preserving the values of these registers during the subroutine call. A fast return is normally paired with a fast call to return the *W*, *BSR*, and *STATUS* to their original states on subroutine return. However, there is only one set of shadow registers, so the subroutine being called cannot make a nested fast call. Furthermore, an interrupt (discussed in Chapter 10, “Interrupts and a First Look at Timers”) cannot occur during the execution of the subroutine

		BBBB BBBB	BBBB BBBB
		1111 1100	0000 0000
		5432 1098	7654 3210
SP = STKPTR[4:0]			
<code>call k, [s]</code>	SP++; (SP) ← (PC)+4; PC[20:1] ← k	1101 110s	kkkk kkkk <sub>o</sub>
<i>if s=1, push W,BSR,STATUS into shadow regs</i>		1111	k <sub>19</sub> kkk kkkk kkkk
<code>rcall k</code>	SP++; (SP) ← (PC) + 2; PC ← (PC) + 2 + 2n	1101 1nnn	nnnn nnnn
<code>return [s]</code>	PC ← ((SP)); SP--	0000 0000	0001 001s
<i>if s=1, restore W,BSR,STATUS from shadow regs</i>			
<code>retlw k</code>	W ← k, PC ← ((SP)); SP--	0000 1100	kkkk kkkk

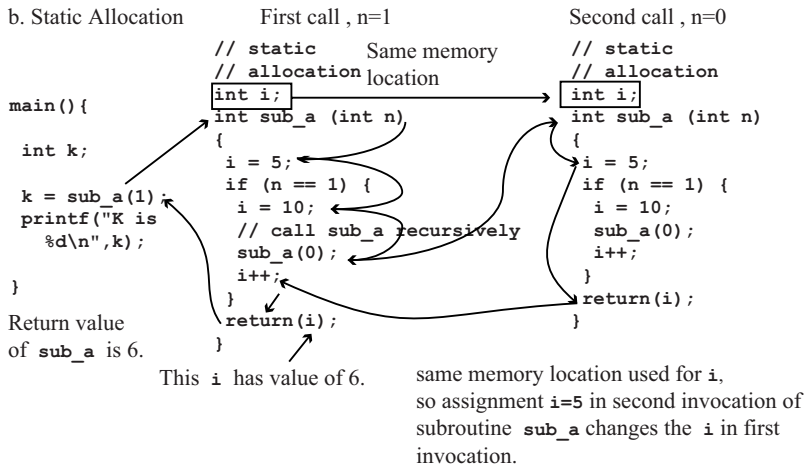
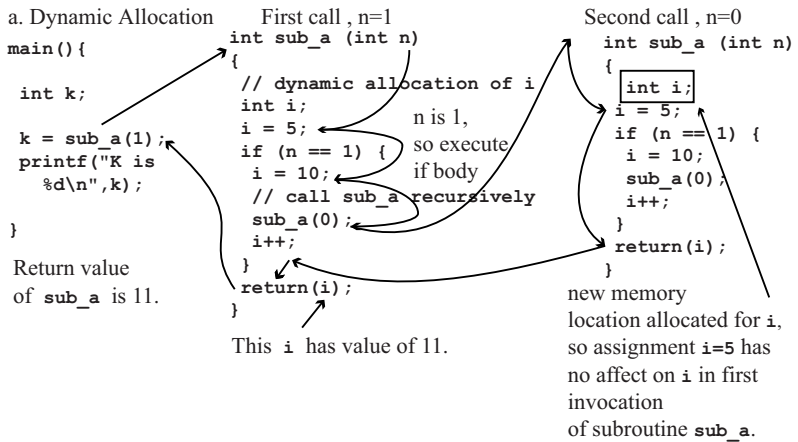
**FIGURE 6.8** Machine code formats for `call`, `return`, `rcall`, and `retlw` instructions.

that was invoked by the fast call, as interrupts automatically use the shadow registers. In practice, the fast register stack is primarily useful for interrupt service routines, which are discussed in detail in Chapter 10. Our use of subroutine call and return always assumes an *s* bit value of “0”, which is the default setting. The `rca11` (relative call) instruction uses program counter relative addressing with an 11-bit displacement, the same as the `bra` instruction. The `retlw` (return with literal in W) instruction encodes an 8-bit literal in the machine word, with the literal loaded into the W register upon return. This is useful for subroutines that return a 1-byte status code as their return value.

In translating a C function to a subroutine in PIC18 assembly language, the first decision is how to allocate the data locations needed for parameters, local variables, and the return value. One method is *static* memory allocation, in which the same data locations are used each time the subroutine is called. The advantage of static memory allocation is that it has low instruction overhead, thus reducing code size and improving execution time. The disadvantage of static allocation is that subroutine recursion is not allowed; that is, the subroutine cannot call itself (or call another subroutine that eventually calls the original subroutine). Subroutine recursion cannot be used with a static allocation strategy because the data locations for parameters and local variables are still in use when the subroutine is re-entered by the nested call to itself. The recursive call overwrites the subroutine data memory area with new values, losing the values still in use by the first call to the subroutine. Please note that any variables declared outside of a C function (*global* variables) are always statically allocated.

*Dynamic* memory allocation uses a new set of memory locations for each subroutine call, so clashes between data memory locations in recursive subroutine calls are avoided. One method to implement dynamic allocation is with a stack located in data memory, which is discussed in Section 6.8.

Figure 6.9 illustrates the problem with using static allocation for recursive subroutines. In this example, the C function `sub_a` uses a local variable named `i` and recursively calls itself if its input parameter `n` has value 1. Figure 6.9a has the `i` internal variable of `sub_a` declared locally to `sub_a`, causing a new memory location to be allocated for `i` each time `sub_a` is called assuming dynamic allocation. The first invocation of `sub_a` executes the *if\_body* because `n == 1`, causing `sub_a` to be called a second time, this time with `n = 0`. In the second invocation of `sub_a`, the *if\_body* is skipped as `n` is zero. The assignment `i = 5` in the second invocation of `sub_a` has no effect on the `i` value in the first invocation, as each version of `i` has a different memory location. Thus, the `i` value in the first invocation of `sub_a` remains at 10, is incremented to 11, and then is returned to `main()`.

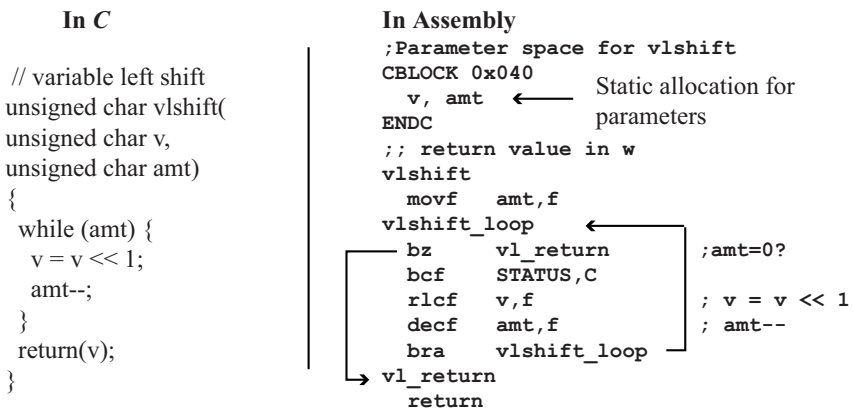


**FIGURE 6.9** Dynamic versus static allocation with recursive function calls.

Figure 6.9b moves the declaration `int i` out of the function, making it a statically allocated variable, causing each call to `sub_a` to use the same memory location for `i`. This time, the assignment `i = 5` in the second invocation of `sub_a` overwrites the previous value 10 assigned to `i` in the first invocation of `sub_a`. When a return is made to the first invocation of `sub_a`, `i` has the value of 5, which is incremented to 6 and returned to `main()`. If static allocation is used for subroutine parameters and local variables, subroutine recursion is not allowed. Whether subroutine recursion is needed is heavily dependent upon the particular application. The compiler used for translating a C program to assembly language is responsible for the

allocation strategy. Most C compilers use dynamic allocation, except for those targeting low-end microcontrollers where code and memory space is limited and the overhead of dynamic allocation is too costly. Outside of the section detailing dynamic allocation, the example subroutines in this chapter use static allocation for any required data memory.

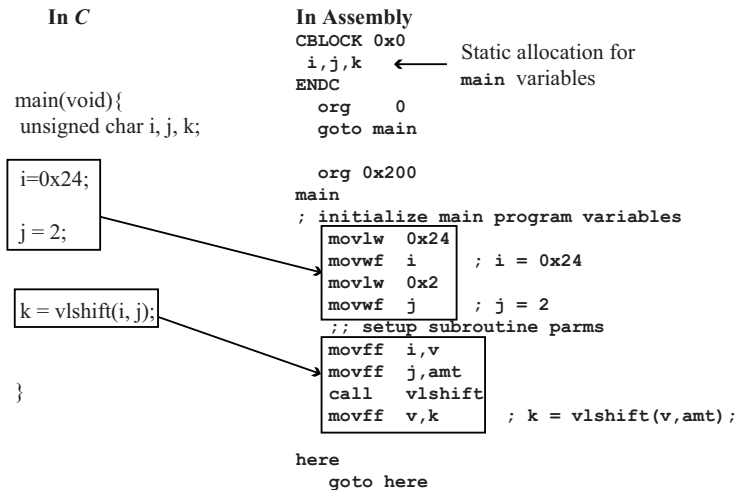
Figure 6.10 shows the `vlshift()` C function implemented as a subroutine in PIC18 assembly language. The implementation is a straightforward implementation of the `while{} loop` of the C function. Note that a `return` statement is used to exit the subroutine; every assembly language subroutine must use a `return` or `retlw` instruction to return to the caller. The `CBLOCK` statement assigns the `unsigned char` parameters `v` and `amt` to locations `0x040` and `0x041`, respectively (these locations were arbitrarily selected for this example). A separate location is not assigned for the return value, as the `v` parameter contains the subroutine return value after execution. In this case, the W register could have been used to pass the return value of the subroutine back to the caller, as the return value is an 8-bit value. Similarly, the W register could be used to pass one of the two parameters, either `v` or `amt`, into the subroutine. This approach would save one memory data location and is an example of a *code optimization*. Code optimizations are done to reduce code size, reduce data memory requirements, and speed execution. Compilers and expert assembly language programmers routinely perform code optimizations such as this. On microprocessors that have separate register memories and data memories, use of registers for parameter passing is important, as access to registers typically costs fewer clock cycles than data memory accesses. However, on the PIC18, data memory locations are the file registers (or vice versa, however one wishes to view it), so there are few optimizations available for parameter passing. Using the W register for parameter passing will save a data memory location, but can only be done if the



**FIGURE 6.10** The `vlshift()` C function in assembly language.

parameter is a byte variable. In the next section, we will discover a set of special function registers that are useful for passing parameters that are pointers; that is, they contain addresses of file registers.

Figure 6.11 shows the `main()` code that calls `vlshift()` implemented in assembly language. Observe that the `i` and `j` values are copied into the `vlshift()` parameter locations `v` and `amt`, respectively, before the `call vlshift` instruction. One may be tempted in this example to simply write the `vlshift()` subroutine to access the `i` and `j` variable locations directly, and to dispense with parameters altogether. However, this simply turns `i` and `j` into the de-facto parameters of the `vlshift()` subroutine, as any other calls to `vlshift()` must modify the values of `i`, `j` before invoking `vlshift()`. Having separate locations for parameters `v` and `amt`, and copying the values of `i`, `j` to these locations before the subroutine call preserves the semantics of the original C program, which does not modify the values of either `i` or `j` via the subroutine call. The `movff v,k` instruction after the subroutine call copies the return value of the subroutine into `k`, as required by the C statement `k = vlshift(i,j)`.



**FIGURE 6.11** Calling the `vlshift()` subroutine from `main()`.

In general, the responsibilities of the caller in a subroutine call are: (1) initialize parameter values, (2) call the subroutine, and (3) copy any return value to a variable within the caller. A caller should not expect the STATUS, BSR, or W registers to remain unchanged by the callee; if these register values are to be preserved during the subroutine call, either a fast call/return should be used or these register values should be copied to temporary memory locations.



## 6.5 ARRAYS AND POINTERS IN C

---

Until now, we have been using variables to store data of varying precisions—char (1 byte), int (2 bytes), long (4 bytes)—each of which can either be unsigned or signed. However, another important class of variables is that of *pointers*, variables that contain *memory addresses* of other variables. We have already used two special function registers that are actually pointers, namely the program counter and stack pointer. The program counter contains the address of a location in program memory, while the stack pointer contains the address of a location in the return address memory. The sizes of these two pointer registers vary widely, as the size is dependent upon the *maximum number of locations in the referenced memory*, and not the size of the data stored in the memory location. The PC requires 21 bits as it addresses 2M bytes of program memory; the SP only requires 5 bits to access the 31 locations of return address memory.

How wide does a pointer register need to be to specify a location in the PIC18 file registers? The answer is 12 bits, because the maximum number of file registers in the PIC18 is 4K bytes, or  $2^{12}$  bytes. The PIC18 has three 12-bit registers named FSR0, FSR1, and FSR2 that are used to implement pointer variables in assembly language. The functionality of these registers is discussed in the next section; first, we explore how pointer variables are implemented in C.

A C pointer variable contains the address of another variable. Pointer variables are typically used to pass blocks of data, or *arrays* of data, to functions. An array is a contiguous block of memory that contains multiple data items of the same type. Listing 6.1 gives examples of arrays and pointers in C.

### LISTING 6.1 Arrays and pointers in C.

---

```
(1) char sa[10], sb[] = "Hello";
(2) char *ptrA, *ptrB;

(3) ptrA = sa;           // ptrA is assigned starting address of sa
(4) ptrA = &sa[0];      // same as previous line
(5) sa[0] = 0x20;        // assign the value 0x20 to element 0
(6) *ptrA = 0x20;        // same as previous line
(7) sa[2] = 0x45;        // assign the value 0x20 to element 2
(8) *(ptrA+2) = 0x45     // same as previous line
(9) sb[0] = sa[0];       // copy sa[0] to element sb[0]
(10) sb[0] = *ptrA;      // same as previous line
(11) ptrA++; ptrA++;     // ptrA now points to sa[2]
(12) sb[0] = *ptrA;      // sa[2] copied into sb[0]
(13) ptrB = sb;          // ptrB is assigned starting address of sb
(14) sa[2] = sb[1]       // copy element 1 of sb to element 2 of sa
(15) *(ptrA) = *(ptrB+1) // same as previous line
(16) ptrA = ptrB+1;      // ptrA now points to element sb[1]
(17) do_mysub(sb);       // pass address of sb to subroutine
(18) do_mysub(ptrB);     // same as previous line
```

The statement `char sa[10]` in line 1 declares an array of 10 `char` elements (or 10 bytes); this occupies 10 consecutive byte memory locations. The array contents are initially cleared to zero if the array is a *global* variable; that is, if it is declared outside of `main()` or a subroutine. In general, the memory location corresponding to a global variable that is not given an initial value is guaranteed by C language semantics to be cleared to zero before the first instruction in `main()` is executed. The `char sb` array is given the initial contents of “Hello”; that is, the array elements contain the ASCII codes for each character of the string “Hello”. The `char sb` array is actually six elements in size instead of five, as any `char` array initialized in this manner is given a null byte (a `0x00` value) as the last element in the array. This is done so that the end of the `char` array can be determined by searching for a null byte. Line 2 declares two pointer variables. The `*` (asterisk) in the declaration `char *ptrA` is what distinguishes `ptrA` as a pointer variable from a normal variable. The declaration `char *ptrA` is read as “`ptrA` is a pointer to data of type `char`.” Line 3 shows how to initialize `ptrA` so that it contains the address of the first member of array `sa`. If an array name, such as `sa`, is used without brackets to identify a particular array member, it is assumed to be the *address* of the first member of the array. Thus, the value `sa` returns the address of the first member of the array, while `sa[0]` returns the value of the first member of the array (in this case, a value of zero). The compiler determines the placement of arrays in memory; when manually converting these statements to PIC18 assembly language you have the choice of placing them wherever there is available data memory. Line 4 shows the use of the “`&`” (*address of operator*) applied to an array member. The statement `&sa[0]` is read as “the address of array element `sa[0]`.” Thus, lines 3 and 4 accomplish the same action; namely, `ptrA` is assigned the address of the first element of array `sa`. In C, if the array contains  $n$  elements, elements are referenced from 0 to  $n-1$ . Line 5 copies the constant value `0x20` to array element `sa[0]`. Line 6 shows the use of the “`*`” (*pointer dereference*) operator applied to a pointer. Line 6 accomplishes the same result as line 5; because when `*ptrA` is used on the left-hand side of an assignment, the assignment value is copied into the memory location referenced by `ptrA` (the memory location referred to by `ptrA` is modified as a result of the assignment). The symbol `ptrA` refers to the memory location containing the pointer value; the operation `*ptrA` refers to the memory location referenced by the pointer value. The operation `*ptrA` is a replacement for `sa[0]` because of the assignment in line 4 that assigned `ptrA` to the address of the first element of `sa`.

Line 7 assigns the value `0x45` to array element `sa[2]`. Line 8 accomplishes the same result as line 7 since `*(ptrA+2)` is a direct replacement for `sa[2]` because `ptrA` contains the starting address of the `sa` array. The computation `ptrA+2` is an example of *pointer arithmetic*; the computation returns the address of element `ptrA[2]` when `ptrA` is viewed as an array. The actual value added to the `ptrA` pointer depends on the type of data that `ptrA` is referencing. Because `ptrA` is a pointer to data of type

char, which is 1 byte in size, the value  $2*1$  is added to `ptrA` to form the address of `ptrA[2]`. If `ptrA` was changed to a pointer of type `int`, where each `int` element occupies 2 bytes of memory, the value  $2*2$ , or 4 is added to `ptrA` to form the address of `ptrA[2]`. Similarly, if `ptrA` was a pointer to type `long`, the value added to `ptrA` for the pointer computation `ptrA+2` is  $2*4$ , as each element that `ptrA` references is 4 bytes in size. In general, the offset added to a pointer for the computation `ptr + i` is computed as  $i*\text{sizeof}(\text{datatype})$ , where `sizeof(datatype)` is the number of bytes required by the data type that `ptr` references. For example, `sizeof(int)` is the value 2, as 2 bytes are required for an `int` data type. The type of data referenced by a pointer affects how pointer arithmetic is done; it does not affect the size of the pointer itself. Pointers to data types `char`, `int`, or `long` are all the same size, as the size of the pointer (number of bytes required to hold the pointer) is dependent upon the maximum number of memory locations that can be referenced by the pointer. Thus, a pointer to data memory in the PIC18 is always 12 bits wide because of the 4K byte limit on file registers, regardless of the data type referenced by the pointer.

Line 9 copies array element `sa[0]` into array element `sb[0]`. Line 10 accomplishes the same result as line 9, as `*ptrA` is a replacement for `sa[0]` as noted previously. When `*ptrA` is used on the right-hand side of an assignment operator, this means to read the contents of the memory location referenced by `ptrA`. Each `ptrA++` operation of line 11 increments `ptrA` to the location of the next `char` that `ptrA` is referencing, after which `ptrA` points at `sa[2]`. Line 12 could then be replaced by the statement `sb[0] = sa[2]`, as `*ptrA` is now equivalent to `sa[2]`. Line 13 assigns `ptrB` to the address of element `sb[0]`. Line 14 copies element `sb[1]` to element `sa[2]`. Line 15 accomplishes the same result as Line 14, as `ptrA` now references element `sa[2]` (because of the previous two `ptrA++` operations), and `ptrB+1` references element `sb[1]` because `ptrB` contains the address of element `sb[0]`. Line 16 removes the `*` operator from the pointers, causing `ptrA` to be assigned the value `ptrB+1`, or the address of element `sb[1]`. Line 17 calls a function named `do_mySub()`, and passes the starting address of array `sb` as the parameter value. Line 18 accomplishes the same result as Line 17, as `ptrB` contains the starting address of array `sb`.

Figure 6.12 uses memory assignments for the locations of the arrays and pointers of Listing 6.1 and illustrates how each `C` statement affects memory contents. Sometimes, seeing numerical values for pointer values can aid in understanding pointer functionality. Array `sa` begins at location `0x150` and occupies 10 bytes of memory (locations `0x150` through `0x159`). Array `sb` requires 6 bytes of memory, locations `0x15A` through `0x15F`. Each pointer variable requires 2 bytes of memory in order to contain the 12-bit addresses that reference data memory. The starting location of `0x150` for these variables is arbitrary; any other free locations in the file registers could have been chosen. After the execution of the statement `ptrA = sa`, the value of `ptrA` is now `0x150`, the starting address of array `sa`. The address of `ptrA`

is 0x0160 (&ptrb), while the value that ptrb is referencing (\*ptrb) is the contents of location 0x150, or 0. The use of numerical values for pointers in Figure 6.12 should make clear the difference between such statements as \*ptrb = \*(ptrb+1) and ptrb = ptrb+1. When \*ptrb = \*(ptrb+1) is executed, ptrb contains the value 0x015A, the starting address of sb. The value of ptrb is 0x0152, the location of sa[2]. Thus, ptrb+1 is 0x015B, and \*(ptrb+1) is the contents of location 0x15B, which is copied to location 0x0152. The statement ptrb = ptrb+1 changes the ptrb value to 0x15B. Because ptrb resides at location 0x0160, this statement modifies the memory locations 0x0160 and 0x161.

C Code	Location	Contents (values in Hex)
char sa[10];	0150	00 00 00 00 00 00 00 00 00 00 sa
char sb[]="Hello";	015A	48 65 6C 6C 6F 00 sb
char *ptrb, *ptrb;	0160	00 00 ptrb
	0162	00 00 ptrb
-----		
ptrb = sa;		
ptrb = &sa[0];	0160	50 01 ptrb = 0x150, value of ptrb is stored at 0x160 in little endian order
sa[0] = 0x20;		
*ptrb = 0x20;	0150	20 ptrb points at location 0x150, *ptrb modifies contents of 0x150
sa[2] = 0x45;		
*(ptrb+2) = 0x45;	0150	20 00 45 ptrb+2 is location 0x152, new value is 0x45
sb[0] = sa[0];		
sb[0] = *ptrb;	015A	20 contents of location 0x150 (0x20) copied to location 0x15A (sb[0])
ptrb++;	0160	51 01 ptrb is now 0x151
ptrb++;	0160	52 01 ptrb is now 0x152
sb[0] = *ptrb;	015A	45 contents of location 0x152 (0x45) copied to location 0x15A (sb[0])
ptrb = sb;	0162	5A 01 ptrb is now 0x15A
sa[2] = sb[1];		
*(ptrb) = *(ptrb+1);	0150	20 00 65 contents of location 0x15B (0x65) copied to location 0x152
ptrb = ptrb+1;	0160	5B 01 ptrb is now 0x15B
do_mysub(sb);		
do_mysub(ptrb);		The value of ptrb, 0x15A, is passed to do_mysub subroutine.

**FIGURE 6.12** C pointer/array code with memory assignments.

Figure 6.13 shows examples of arrays and pointers using type `int` instead of `char`. The difference in data types means that each element of arrays `sa`, `sb` now occupy 2 bytes of memory instead of only 1 byte. Thus, array `sa` occupies  $10 * 2 = 20$  bytes of memory from location 0x150 through location 0x163. The starting address of array `sb` is 0x164, and occupies  $4 \text{ int} * 2 \text{ bytes} = 8$  bytes of memory. Even though `ptrb` and `ptrb` are now pointers to elements that are double the size of the `char` elements of Listing 6.1, the pointers themselves have not changed size, they still each occupy 2 bytes of memory. The first two lines of code initialize `ptrb` and `ptrb` to the starting locations of arrays `sa` and `sb`, respectively. Observe that the op-

eration `ptrb++` increases the value of the pointer by 2, from 0x164 to 0x166. As noted earlier, `ptrb++` is a pointer to type `int`, and by the rules of pointer arithmetic, `ptrb` is increased by  $1 * \text{sizeof}(\text{int})$ , or  $1 * 2 = 2$ . The operation `*(ptrb+2) = *(ptrb+2)` copies the `int` at the location referenced by `ptrb+2` to the location referenced by `ptrb+2`. Because `ptrb` was first initialized to the starting location of `sb`, then incremented, the value `ptrb+2` now references `sb[3]` or location 0x16A. The variable `ptrb` references `sa[0]`, or location 0x150, so the address `ptrb+3` references `sa[3]`, or location  $0x150 + 2 * 3 = 0x156$ .

C Code	Location	Contents (values in Hex)								
<code>signed int sa[10];</code>	0150	00 00 00 00 00 00 00 00								
<code>signed int sb[]={</code>	0158	00 00 00 00 00 00 00 00								
<code>  -20,1000,-546,</code>	0160	00 00 00 00								
<code>  23444};</code>	0164	EC FF E8 03 DE FD 94 5B								
<code>signed int *ptrb;</code>	016C	00 00 ptrb								
<code>signed int *ptrb;</code>	016E	00 00 ptrb								
-----										
<code>ptrb = sa;</code>	→ 016C	50 01 ptrb is now 0x150								
<code>ptrb = sb;</code>	→ 016E	64 01 ptrb is now 0x164								
<code>ptrb++;</code>	→ 016E	66 01 ptrb is now 0x166, note that it increased by 2 because each element size is 2 bytes; ptrb points at sb[1]								
<code>*ptrb = *ptrb;</code>	→ 0150	E8 03 copy integer at location 0x166 (ptrb) to location 0x150 (*ptrb). This is the same as sa[0] = sb[1]								
<code>*(ptrb+3) = *(ptrb+2);</code>	→ 0150	<table border="0"> <tr> <td>sa[0]</td> <td>sa[1]</td> <td>sa[2]</td> <td>sa[3]</td> </tr> <tr> <td>E8 03</td> <td>00 00</td> <td>00 00</td> <td>94 5B</td> </tr> </table> ptrb+3 refers to location sa[3] (0x156) as ptrb is sa (0x150) ptrb+2 refers to location sb[3] (0x16A) as ptrb is &sb[1] (0x166)	sa[0]	sa[1]	sa[2]	sa[3]	E8 03	00 00	00 00	94 5B
sa[0]	sa[1]	sa[2]	sa[3]							
E8 03	00 00	00 00	94 5B							

**FIGURE 6.13** C pointer/array of type `int` with memory assignments.

**Sample question:** For the following C code fragment, assume the variables are stored in memory starting at location 0x0100. What is the starting address of each variable assuming this code is compiled for the PIC18? What is the final value of `ptr`? What array element in `s` is modified?

```
char s[8],*ptr, a;

a = 5;

ptr = s;
```

→

```
ptr = ptr + 3;
*ptr = a;
```

*Answer:* The starting memory location for `s` is `0x100`. The starting memory location for `ptr` is  $0x100 + 8 = 0x108$  because `s` occupies 8 bytes of memory. The starting memory location for `a` is  $0x108 + 2 = 0x10A$  because `ptr` occupies 2 bytes of memory (a pointer variable always occupies 2 bytes of memory irregardless of the data type it references). The `ptr` variable is first initialized to the starting address of `s` (`0x100`). The pointer arithmetic `ptr = ptr + 3` is calculated as  $0x100 + 3*1 = 0x103$  since `ptr` is a `char *` pointer, so each element is 1 byte in size. Thus, the final value of `ptr` is `0x103`. The statement `*ptr = a` modifies array element `s[3]` because `ptr` is pointing to `s[3]`.

**Sample Question:** For the previous question, change the data type from `char` to `int` and answer the same questions; change `char` to `long` and answer the same questions.

*Answer:* When the data type is changed from `char` to `int`, the starting memory location for `s` is still `0x100`. The starting memory location for `ptr` is  $0x100 + 0x10 = 0x110$  because `s` occupies  $8*2 = 16 = 0x10$  bytes of memory. The starting memory location for `a` is  $0x110 + 2 = 0x112$  because `ptr` occupies 2 bytes of memory (a pointer variable always occupies 2 bytes of memory irregardless of the data type it references). The `ptr` variable is first initialized to the starting address of `s` (`0x100`). The pointer arithmetic `ptr = ptr + 3` is calculated as  $0x100 + 3*2 = 0x106$  since `ptr` is an `int *` pointer, so each element is 2 bytes in size. Thus, the final value of `ptr` is `0x106`. The statement `*ptr = a` modifies array element `s[3]` because `ptr` is pointing to `s[3]`.

When the data type is changed from `char` to `long`, the starting memory location for `s` is still `0x100`. The starting memory location for `ptr` is  $0x100 + 0x20 = 0x120$  because `s` occupies  $8*4 = 32 = 0x20$  bytes of memory. The starting memory location for `a` is  $0x120 + 2 = 0x122$  because `ptr` occupies 2 bytes of memory (a pointer variable always occupies 2 bytes of memory irregardless of the data type it references). The `ptr` variable is first initialized to the starting address of `s` (`0x100`). The pointer arithmetic `ptr = ptr + 3` is calculated as  $0x100 + 3*4 = 0x10C$  since `ptr` is a `long *` pointer, so each element is 4 bytes in size. Thus, the final value of `ptr` is `0x10C`. The statement `*ptr = a` modifies array element `s[3]` because `ptr` is pointing to `s[3]`.

## 6.6 ARRAYS AND POINTERS IN ASSEMBLY LANGUAGE

The PIC18 has three registers named FSR0, FSR1, and FSR2 for implementing pointers. These are collectively referred to as FSRn, and all operations affecting an FSRn register work identically regardless of the particular register being used. The FSRn registers are 12 bits wide, and are used to contain addresses of file register locations. The individual bytes of an FSRn register are named FSRLn (FSRn low byte) and FSRHn (FSRn high byte). To initialize an FSRn register, each individual byte of an FSRn register can be loaded separately to form a 12-bit address. Alternately, the `lfsr FSRn, k` instruction can be used, where *k* specifies a 12-bit literal that is loaded into the FSRn register. The FSRn registers contain pointer values, which are addresses of data memory locations. To affect the memory location that the pointer references, the special register INDFn is used as the file register for an instruction. Each INDFn register is paired with its associated FSRn register; FSR0 with INDF0, FSR1 with INDF1, and FSR2 with INDF2.

Figure 6.14 shows an example of FSRn/INDFn usage to implement C pointer referencing. The statement `ptr a = sa` initializes `ptr a` to point to the starting memory location of `sa`. The variable `ptr a` is implemented using FSR0, with each byte of FSR0 initialized separately. The construct `low sa` instructs the assembler to form a literal using the low byte of symbol `sa`. The symbol `sa` represents location `0x0100` as per the `CBLOCK` statement, so `movlw low sa` is equivalent to `movlw 0x00`. Similarly, `high sa` refers to the high byte of symbol `sa`, so `movlw high sa` is equivalent to `movlw 0x01`. The instruction pair `movlw 0x30; movwf INDF0` implements the C statement `*ptr a = 0x30`. The instruction `movwf INDF0` accomplishes the operation `movwf`

In C	In Assembly
<code>char sa[5];</code>	<code>CBLOCK 0x100</code>
<code>char sb[10];</code>	<code>sa:5, sb:10 ;sa is 0x100, sb is 0x105</code>
<code>char *ptr a, *ptr b;</code>	<code>ENDC</code>
	<code>; FSR0 used for ptr a, FSR1 for ptr b</code>
<code>ptr a = sa;</code>	<div style="border: 1px solid black; padding: 2px; display: inline-block;"> <code>movlw low sa</code>  <code>movwf FSR0L</code>  <code>movlw high sa</code>  <code>movwf FSR0H</code> </div> <span style="margin-left: 20px;">;init FSR0L</span> <span style="margin-left: 20px;">;FSR0 = 0x0100</span>
<code>*ptr a = 0x30;</code>	<div style="border: 1px solid black; padding: 2px; display: inline-block;"> <code>movlw 0x30</code>  <code>movwf INDF0</code> </div> <span style="margin-left: 20px;">;(FSR0) ← 0x30</span>
<code>ptr b = sb;</code>	<code>lfsr FSR1, sb</code> <span style="margin-left: 20px;">;FSR1 = 0x105</span>
<code>*ptr b = *ptr a;</code>	<code>movff INDF0, INDF1</code> <span style="margin-left: 20px;">;(FSR1) ← ((FSR0))</span>

**FIGURE 6.14** FSRn/INDFn usage.

0x0100, as FSR0 contains the value 0x0100. When INDF0 is used in an instruction, it *indirectly* specifies the actual file register location being referenced through the FSR0 register. This type of addressing is called *indirect addressing*, where a register contains the address of a memory location being referenced. As mentioned in Chapter 3, the instruction `movwf 0x100` uses *direct addressing*, because the address 0x100 is encoded directly in the machine code of the instruction. The instruction `movwf 0x100` always refers to location 0x100, and cannot be changed without physically changing the instruction word. The instruction `movwf INDF0` refers to whatever memory location is contained in FSR0, and is thus more flexible than `movwf 0x100`. Indirect addressing allows subroutines that operate on arrays to be written generically; the starting address of an array used by the subroutine is passed as a parameter value. The register transfer language description of `movwf INDF0` is  $(FSR0) \leftarrow W$ , with the parentheses used to indicate the indirection (W is copied to the memory location referenced by FSR0). If FSR0 contains 0x100, then  $(FSR0)$  is replaced by 0x100, and the equivalent action becomes  $0x100 \leftarrow W$  (W is copied to memory location 0x100).

In Figure 6.14, FSR1 is used to point to array `sb`, and is initialized with the single statement `1fsr FSR1, sb`, which copies the 12-bit literal represented by the symbol `sb` into FSR1. The `1fsr` instruction is explicitly provided for loading a 12-bit literal into an FSRn register, and is more efficient than using two pairs of `movlw/movwf` instructions. The assignment `*ptrb = *ptr` is accomplished by the single instruction `movff INDF0, INDF1`. Because `ptr` contains the starting address of `sa` (0x100), and `ptrb` the starting address of `sb` (0x105), the statement `*ptrb = *ptr` copies the contents of location 0x100 to location 0x105. The FSR0, FSR1 registers contain the values 0x0100, 0x0105, respectively, so the action of instruction `movff INDF0, INDF1` is equivalent to `movff 0x100, 0x105`.

It is somewhat of a misnomer to refer to the INDFn file registers as registers, as they do not physically implement file registers even though they are assigned locations in the special function register range of 0xF80 to 0xFFF. Instead, these are essentially a form of opcode that specifies the addressing mode that is used with the corresponding FSRn register, which *does* represent a physical register. There are four addressing modes other than INDFn available for use with the FSRn registers, and they are defined as follows:

**POSTINCn:** Use the address in FSRn, and then increment FSRn. In C, this is equivalent to `*ptr, ptr++`, if `ptr` is a pointer to type `char`.

**POSTDECn:** Use the address in FSRn, and then decrement FSRn. In C, this is equivalent to `*ptr, ptr--`, if `ptr` is a pointer to type `char`.

**PREINCn:** Increment FSRn, and then use the address in FSRn. In C, this is equivalent to `ptr++, *ptr`, if `ptr` is a pointer to type `char`.



**PLUWn:** Uses the value of  $W$  as an offset added to  $FSR_n$  to form the address. In  $C$ , this is equivalent to  $*(ptr+k)$ , where  $k$  is the offset in the  $W$  register, and  $ptr$  is a pointer to `type char`. If  $ptr$  references a type other than `char`, the  $W$  register value has to be set equal to  $k * \text{sizeof}(\text{datatype})$ , where *datatype* is the type referenced by  $ptr$ .

Observe that  $FSR_n$  is incremented by both the  $POSTINC_n$  and  $PREINC_n$  addressing modes. However, in the  $POSTINC_n$  mode,  $FSR_n$  is incremented *after* (post) it is used to reference memory. In the  $PREINC_n$  mode,  $FSR_n$  is incremented *before* (pre) it is used to reference memory. Figure 6.15 shows how  $POSTINC_n$  is used within a `for{}`  loop that copies the first five values of array `sb` (`sb[0]` to `sb[4]`) into array `sa`. Notice that four  $C$  language statements are replaced by the single instruction `movff POSTINC1, POSTINC0`. This code reduction occurs because  $POSTINC_n$  combines both a memory reference and pointer increment in one operation. It would be an error to use the instruction `movff PREINC1, PREINC0`, as this would copy values `sb[1]` through `sb[5]` into `sa[1]` through `sa[5]`, because the increment of  $FSR_1/FSR_0$  happens before the memory reference. The instruction sequence `infsnz FSR0L, f ;incf FSR0H` can be used to increment the  $FSR_0$  register if use of  $POSTINC_n/PREINC_n$  modes is not possible. The `movlb 1` instruction is used to set the BSR to bank 1 because of the instructions that access the `i` variable; the BSR register is not used with operations involving indirect addressing, as an  $FSR_n$  register specifies the full 12-bit address required for a data memory location.

In C	In Assembly
<code>char sa[5];</code>	<code>CBLOCK 0x100</code>
<code>char sb[10];</code>	<code>sa:5, sb:0xa, i ;sa is 0x100, sb is 0x105</code>
<code>char *ptra, *ptrb;</code>	<code>ENDC</code>
<code>unsigned char i;</code>	<code>movlb 1 ;select bank 1</code>
<code>ptra = sa;</code>	<code>lfsr FSR0, sa ;init FSR0</code>
<code>ptrb = sb;</code>	<code>lfsr FSR1, sb; ;init FSR1</code>
<code>for (i=0; i &lt; 5; i++) {</code>	<code>clrf i, f ;i = 0</code>
<code>  *ptra = *ptrb;</code>	<code>loop_top</code>
<code>  ptra++;</code>	<code>movlw 5</code>
<code>  ptrb++;</code>	<code>cpfslt i ;i &lt; 5?</code>
<code>}</code>	<code>bra loop_end ;exit loop</code>
	<code>movff POSTINC1, POSTINC0</code>
	<code>incf i, f ;i++</code>
	<code>bra loop_top</code>
	<code>loop_end</code>
	<code>...rest of code...</code>



**FIGURE 6.15**  $POSTINC_n$  usage.

Figure 6.16 shows the C code of Figure 6.15 rewritten in a more natural manner to use array references instead of pointer references. Note that FSR0/FSR1 are still used to access the elements of arrays *sa* and *sb*. The PLUSWn addressing mode is used in this example to implement the statement *sa[i] = sb[i]*. Observe that before the statement `movff PLUSW1, PLUSW0` is executed, the value of *i* is loaded into the W register by `movf i,w`. The use of the PLUSWn addressing mode is limited by the size of offsets that can be stored in the W register, namely 0 through 255. If more than 256 elements were being copied, this code would have to be rewritten, as W is not large enough to contain the required offsets.

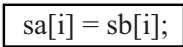
In C	In Assembly
<code>char sa[5];</code>	<code>CBLOCK 0x100</code>
<code>char sb[10];</code>	<code>sa:5,sb:0x10,i ;sa is 0x100,sb is 0x105</code>
<code>unsigned char i;</code>	<code>ENDC</code>
	<code>movlb 1 ;select bank 1</code>
	<code>lfsr FSR0, sa ;init FSR0</code>
	<code>lfsr FSR1, sb ;init FSR1</code>
	<code>clrf i,f ;i = 0</code>
	<code>loop_top</code>
	<code>movlw 5</code>
	<code>cpfslt i ;i &lt; 5?</code>
	<code>bra loop_end ;exit loop</code>
<code>for (i=0; i &lt; 5; i++) {</code>	<code>movf i,w ;get i</code>
<code>sa[i] = sb[i];</code>	<code>movff PLUSW1, PLUSW0</code>
	<code>incf i,f ;i++</code>
	<code>bra loop_top</code>
<code>}</code>	<code>loop_end</code>
	<code>...rest of code...</code>



**FIGURE 6.16** PLUSWn usage with `char` arrays.

Figure 6.17 shows the C code of Figure 6.16 changed to use `int` data instead of `char` data. The *i* value loaded into the W register for use by PLUSWn has to be multiplied by 2 because each array element is 2 bytes in size. The first `movff PLUSW1, PLUSW0` instruction copies the least significant byte of *sb[i]* to *sa[i]*. Then, W is incremented by 1 so that the subsequent `movff PLUSW1, PLUSW0` instruction copies the most significant byte of *sb[i]* to *sa[i]*.

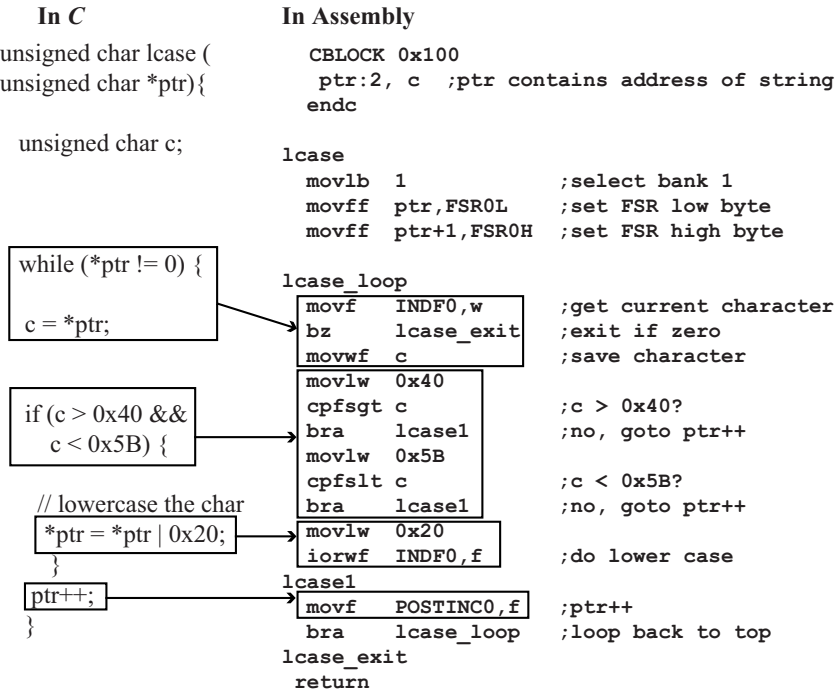
In C	In Assembly
<pre>int sa[5]; int sb[10]; unsigned char i;  for (i=0; i &lt; 5; i++) {     sa[i] = sb[i]; }</pre>	<pre>CBLOCK 0x100     ;sa is 0x100, sb is 0x10A     sa:5*2, sb:0x0a*2,i ENDC  movlb 1 ;select bank 1 lfsr FSR0, sa ;init FSR0 lfsr FSR1, sb; ;init FSR1 clrf i,f ;i = 0 loop_top movlw 5 cpfslt i ;i &lt; 5? bra loop_end ;exit loop bcf STATUS,C rlcf i,w ;w = i * 2 movff PLUSW1, PLUSW0 ;LSByte copy addlw 1 ;w = w+1 movff PLUSW1, PLUSW0 ;MSBbyte copy incf i,f ;i++ bra loop_top loop_end ...rest of code...</pre>



**FIGURE 6.17** PLUSWn usage with int arrays.

### A Subroutine with Pointers

Figure 6.18 shows a function named `lcase()` that converts any uppercase characters in a string to lowercase. The amount of ASCII data manipulation performed by a microcontroller depends on the application; string data is used in this example, as it is easy to see the effects of the ASCII data manipulation within the MPLAB simulation environment. The function loops through the characters in the string, and tests for an uppercase character by checking if the character is greater than 0x40 or less than 0x5B. If an uppercase character is found, the character is converted to lowercase by OR'ing it with 0x20, which sets bit 5 of the character. If the character is zero, the end of the string has been reached, and the function is exited. The starting address of the string is passed in the `ptr` parameter, which requires 2 bytes to hold the 12-bit address. The assembly code moves the contents of `ptr` into FSR0, and uses this register to access the string contents. The assembly code is a straightforward conversion of the C code. The instruction `movf POSTINC0, f` is used to accomplish `ptr++`; the `PREINC0` addressing mode works equally well in this case.



**FIGURE 6.18** String lowercase function.

The `main()` code that calls the `lcase()` function is shown in Figure 6.19. The `s1` string is declared as a global variable and is given the initial value “Upper/LOWER0123”. The `main()` code consists of one statement, `lcase(s1)`, that calls the `lcase()` function with the starting address of the `s1` string. After `lcase()` execution, the `s1` string is modified to contain “upper/lower0123”. The assembly code reserves 16 bytes of space for `s1` in the file registers at location `0x280` via the `CBLOCK` statement. The `call init_s1` instruction executes a subroutine that copies the initial contents of the `s1` string from program memory to data memory; details of this subroutine are discussed in the next section. The starting location of `s1` is copied into the `ptr` parameter for `lcase()`, and then `call lcase` executes the `lcase()` subroutine. A code optimization would be to drop the use of the `ptr` memory locations and pass the starting address of `s1` directly in the `FSR0` register.

In C	In Assembly
<pre>char s1[] =     "Upper/LOWER0123";  main(void) {     lcase (s1); }</pre>	<pre>CBLOCK 0x280     s1:0x10 ;reserve 16 bytes of space endc  org 0     goto main  org 0x0200 main     ;copy string in program mem     ;to s1 data memory     call init_s1      ;set up call for strtnt     movlb 1 ;select bank1     movlw low s1     movwf ptr     movlw high s1     movwf ptr+1 ;set ptr = s1 value     call lcase ;do lower case      here ;end program with infinite loop     goto here</pre>

**FIGURE 6.19** main() code that calls lcase() subroutine.

**Sample Question: Implement the following C function in PIC18 assembly language.**

```
mysub (char *s){
    (*s)++;
}

char a_val;
main() {
    mysub(&a_val);
}
```

*Answer:* This function performs an increment operation on the value pointed to by s. Recall that the operation &a\_val returns the address of variable a\_val. In Listing 6.2, separate CBLOCKS are used for the memory space required by mysub() and by main(). An optimization could be to just use the FSR0 register to pass in the parameter required by mysub().

**LISTING 6.2** Sample question solution.

```
CBLOCK 0x100 ;space for mysub
    s:2
ENDC
```

```

mysub
    movff    s,FSR0L    ;
    movff    s+1,FSR0H ;FSR0 holds s pointer
    incf    INDF0,f    ;(*s)++
    return

    CBLOCK 0x000    ;space for main
        a_val
    ENDC

main
    movlw    low a_val ;
    movwf    s        ;initialize s LSByte to LSByte address of a_val
    movlw    high a_val ;initialize s MSByte to MSByte address of a_val
    movwf    s+1
    call    mysub     ;mysub(&a_val)

here
    goto    here     ;stop

```

**Sample Question: Implement the following C function in PIC18 assembly language.**

```

mysub (int *s){
    (*s)++;
}

int a_val;
main() {
    mysub(&a_val);
}

```

*Answer:* This is the same as the previous sample question, except the data types have been changed from char to int. In this solution (Listing 6.3), a CBLOCK is not used for mysub(). Instead, the main() code uses the FSR0 register to pass the s value to the mysub() function. This reduces the number of instructions required in both mysub() and main(). Note that the (\*s)++ operation in mysub() is performed on an int value, so a 16-bit increment is performed. The use of POSTINC0 in the LSByte increment operation incf POSTINC0,f increments FSR0 to point to the MSByte of \*s after the incf operation is performed. This means that the addwfc INDF0,f instruction affects the MSByte of \*s.

**LISTING 6.3** Sample question solution.

```

mysub                                ;on entry, assume FSR0 contains s value
    movlw    0                        ;W = 0, needed when W is added to MSByte
    incf    POSTINC0,f                ;(*s)++ of LSByte, point FSR0 at MSByte
    addwfc  INDF0,f                    ;(*s)++ of MSByte, must include Carry flag
    return

```

```

        CBLOCK 0x000          ;space for main
        a_val
    ENDC

main
    lfsr    FSR0,a_val      ;load FSR0 with address of a_val
    call   mysub           ;mysub(&a_val)

here
    goto   here           ;stop

```

## 6.7 ACCESSING TABLE DATA FROM PROGRAM MEMORY

---

Initial values for *C* global variables present an interesting challenge in microcontrollers. The *C* language semantics guarantee that before `main()` is executed, a global variable is cleared to zero if no specific initial value is given, or is loaded with the initial value specified in the variable declaration. A *C* compiler generates *initialization code* that is executed on microprocessor reset, and before `main()` is called, that accomplishes global variable initialization. The `s1` string in Figure 6.19 is given an initial value of “Upper/LOWER0123”, which is later modified by the subroutine call to `lcase()`. The `s1` string must be stored in the file registers since its value is modified during code execution. However, the initial value “Upper/LOWER0123” must be stored in nonvolatile memory, as it must be available after power-on so it can be copied by initialization code to file registers. The nonvolatile memory used to store initial values for character strings, and global variables in general, is program memory. A special set of PIC18 instructions named *table reads* is used to read the contents of program memory, while *table writes* are used to write new values into program memory. This section discusses table read operations; table writes are discussed in Chapter 14, “Capstone: Audio Sampling, Monitoring System, and Autonomous Robot.”

A pointer register named TBLPTR is used to hold the address of the program memory location accessed by a table read instruction. The TBLPTR register is 21 bits wide, the same width as the program counter, as both registers access locations in program memory. A table read transfers the contents of the program memory location referenced by TBLPTR into an 8-bit register named TABLAT. There are four table read instructions, their names and operation are as follows:

**tblrd\*:** TABLAT ← (Program memory(TBLPTR)). Transfers the contents of the program memory location specified by TBLPTR to the TABLAT register.

**tblrd\*\*:** TABLAT ← (Program memory(TBLPTR)); TBLPTR++. Transfers the contents of the program memory location specified by TBLPTR to the

TABLAT register, and then increments TBLPTR (table read with post-increment).

**tblrd\*-:** TABLAT ← (Program memory(TBLPTR)); TBLPTR--. Transfers the contents of the program memory location specified by TBLPTR to the TABLAT register, and then decrements TBLPTR (table read with post-decrement).

**tblrd\*+:** TBLPTR++;TABLAT ← (Program memory(TBLPTR)). Increment TBLPTR, and then transfer the contents of the program memory location specified by TBLPTR to the TABLAT register (table read with pre-increment).

Note the similarities of these address modes for table reads with the addressing modes available for FSRn, and the use of \* in the instruction mnemonic to suggest pointer referencing in C.

Listing 6.4 shows the `init_s1` subroutine used in Figure 6.19 that copies the initial value for `s1` from program memory.

---

**LISTING 6.4** `init_s1` code that performs table reads.

---

```
(1)  s1const
(2)  da "Upper/LOWER0123",0 ;"da" packs two bytes to a word

(3)  init_s1
(4)  movlw upper s1const ; 'upper' is upper byte of 21-bit
address
(5)  movwf TBLPTRU
(6)  movlw high s1const
(7)  movwf TBLPTRH
(8)  movlw low s1const
(9)  movwf TBLPTRL
(10) lfsr FSR0,s1 ; point FSR at s1
(11) call init_str
(12) return

;; FSR must be pointing to where string is to be stored
(13) init_str

(14) tblrd*+ ; use table read to get byte
(15) movf TABLAT, w ; transfer TABLAT to W
(16) movwf POSTINC0 ; save to string location, FSR0++
(17) bnz init_str ; loop if byte not zero
(18) return
```

The initial value of the `s1` string is stored in program memory under the `s1const` label, using the `da` (store string in program memory) assembler directive. The `da` directive packs two ASCII characters into each 16-bit word of program memory. A null byte (0x0) is included in the declaration of the `s1const` string so

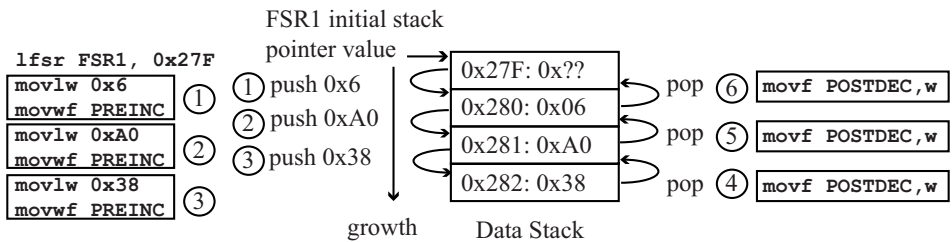


that the end of the string can be detected during copying from program memory to data memory. The `s1_init` subroutine initializes `TBLPTR` to the starting address of `s1const`, and `FSR0` to the starting address of `s1`, and then calls the `init_str` subroutine that performs the actual copying. The 21 bits of the `TBLPTR` register are initialized by writing to three separate registers: `TBLPTRL` (lower byte), `TBLPTRH` (middle byte), and `TBLPTRU` (upper byte). The upper `s1const` directive returns a 5-bit literal that is the upper 5 bits of the program memory address represented by `s1const`. The `init_str` subroutine is a loop that uses a table read with post-increment (`tblrd++`) to copy the contents of the program memory location referenced by `TBLPTR` to the `TABLAT` register. The `TABLAT` register is transferred to `W`, and then the `movwf POSTINC0` instruction copies `W` to data memory. The loop continues until the null byte of the string is copied, at which point the `init_str` subroutine exits. You may be tempted to replace the instruction pair `movf TABLAT,w; movwf POSTINC0` with the single instruction `movff TABLAT,POSTINC0`. However, the `movff` instruction does not affect the `Z` flag, so the following `bnz` instruction will not detect when the null byte is copied.

## **6.8 SUBROUTINES AND STACK FRAMES: DYNAMIC ALLOCATION**

Now that pointers have been discussed, we need to revisit the topic of memory allocation for subroutine parameter lists and local variables. Recall that static allocation uses a fixed set of memory locations for subroutine parameters and local variables. The problem with static allocation is that subroutine recursion is not supported, as a recursive call to a subroutine destroys the static variables in use by the current subroutine call. Dynamic allocation uses a potentially new set of memory locations for parameters and local variables each time a subroutine is called. One common method for implementing dynamic allocation is to reserve a section of memory for a *data stack*, which dynamically grows in size to accommodate subroutine memory requirements as subroutine calls are made. The caller pushes parameters onto the data stack before calling the subroutine, while the subroutine allocates space on the stack for local variables. To implement a data stack, a pointer register is reserved to keep track of the top of the stack. This register is referred to as the *stack pointer* (`SP`), and `FSR1` is used in these examples. We will define a *push* operation as incrementing the stack pointer, and then storing the data value at the location referenced by the stack pointer. Thus, a push of the `W` register onto the stack is accomplished by the instruction `movwf PREINC1`, which increments `FSR1` and then stores `W` at the location referenced by `FSR1`. By this definition, the stack pointer points at the last item pushed on the stack, and the memory location that the stack pointer initially references will not contain data. Conversely, a *pop* operation reads the value referenced by the stack pointer, and then decrements the stack

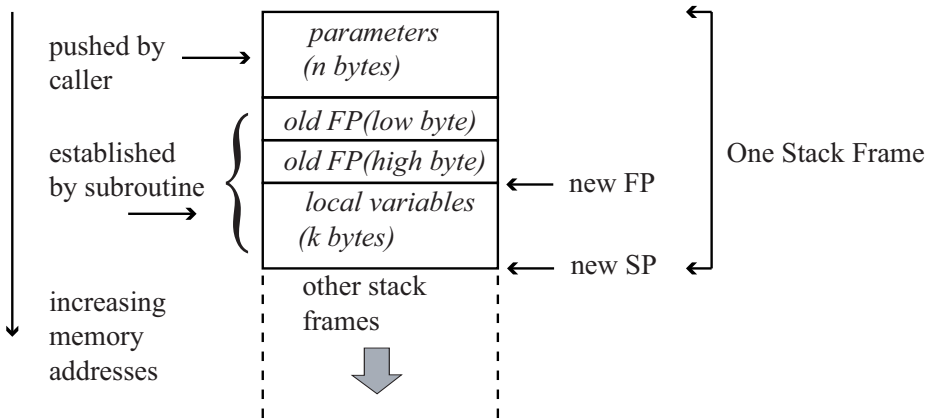
pointer. The instruction `movf POSTDEC0,w` thus pops a value off the stack into the W register. By these definitions, the stack grows toward increasing memory addresses, and shrinks toward decreasing addresses. These definitions of push, pop, and the use of FSR1 as the stack pointer are somewhat arbitrary choices; other definitions could be used. For example, many microprocessors have their stacks grow down in memory; that is, toward decreasing memory addresses. These push and pop definitions are the same as used for the return address stack and thus should be familiar. Figure 6.20 shows sample push and pop operations of literal values onto a data stack located at 0x27F. Observe that while the stack pointer (FSR1) is initialized to 0x27F, the first push operation actually writes data to location 0x280. While a data stack can be placed anywhere in memory, a typical strategy is to allocate space for static variables (global variables) in low memory, and stack space in high memory. Our data stack will have no automatic means of checking stack overflow or underflow. If this stack was implemented on the PIC18F242, which only has three banks physically implemented, the last data location is 0x2FF. A push to memory location 0x300 would be a stack overflow, but there is no hardware detection of this overflow. Extra code could be inserted in our push and pop implementations to check for stack overflow and underflow, but this costs program memory space and execution time. Our examples will not check for data stack underflow/overflow. In some compilers, data stack underflow/overflow checking is enabled or disabled via compiler options.



**FIGURE 6.20** Examples of push/pop operations to a data stack.

The space allocated on the stack for subroutine parameters and local variables is called a *stack frame*. Because a subroutine can call other subroutines, which changes the value of the stack pointer, a second pointer register called a *frame pointer* (FP) is used as a stable reference to the parameters and local variables of a subroutine. In our examples, we use FSR2 as the frame pointer. Figure 6.21 shows the format of the stack frame used in these code examples. The caller pushes the parameters onto the stack before calling the subroutine; the number of bytes required for parameters depends on the number of parameters and their types. The first ac-

tion of the subroutine is to push the current frame pointer on the stack to preserve its value, as this subroutine changes the value of the frame pointer to reference its parameters and local variables. Local variable space is allocated by incrementing the stack pointer by the number of required bytes. The frame pointer is left pointing to the first local variable. Subsequent subroutine calls allocate new space above this stack frame. Parameters are accessed from the frame pointer using negative offsets, while local variables are accessed from the frame pointer using positive offsets. The subroutine return value can be passed back in registers, or written to the space used by the parameters. The subroutine must restore the frame pointer to its value on entry before executing a return from subroutine. The parameters passed by the caller are also cleaned up before return, by either popping the parameters off the stack or simply subtracting  $n$  from the stack pointer, where  $n$  is the number of bytes required for the parameters.



**FIGURE 6.21** A stack frame.

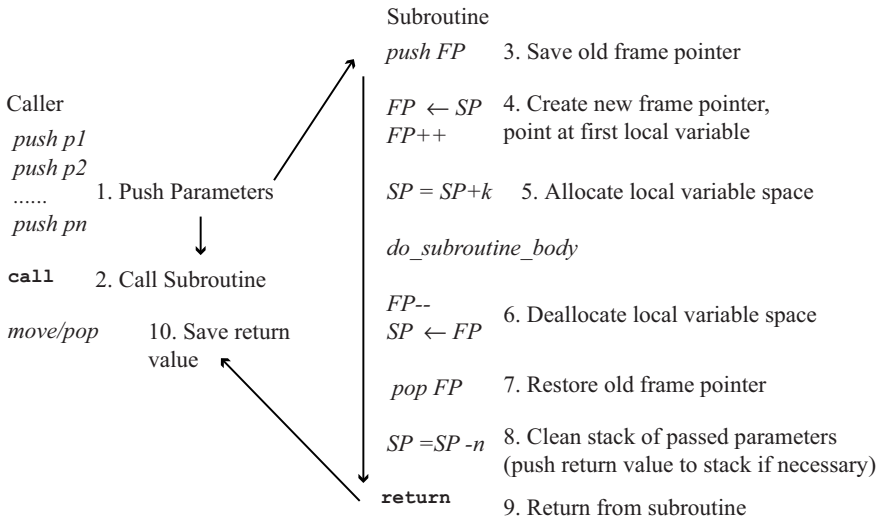
Table 6.1 summarizes how return values should be passed back to the caller. If the subroutine return value is an 8-bit value, it is passed back in the W register. If the return value is a 16-bit value, it is passed back in the PRODH:PRODL special function register pair (these registers are used by the multiply instructions but can also be used as general-purpose registers; these registers are discussed in Chapter 7, “Advanced Assembly Language: Higher Math”). A 32-bit or greater return value should be pushed on the data stack by the subroutine before returning to the caller; it is the responsibility of the caller to pop this return value off the stack.

The detailed steps in constructing a stack frame are given in Figure 6.22. The majority of the work is done by the subroutine; the caller only has to push the parameters on the stack before the call, and save the return value. In the subroutine,

after the old frame pointer is pushed on the stack, the actions  $FP \leftarrow SP$ ;  $FP++$  establish the new frame pointer and leave it pointing at the first location used for local variables. Local space is allocated by incrementing the stack pointer by  $k$ , the number of bytes required for local variables. After the subroutine body is executed, the stack is cleaned by first pointing the stack pointer at the location of the old frame pointer by the actions  $FP--$ ;  $SP \leftarrow FP$ . Popping  $FP$  off the stack restores the old frame pointer, then the subroutine parameter space is reclaimed by incrementing the stack pointer by  $n$ , which is the number of bytes required for the parameters. At this point, if the subroutine has a 32-bit or greater return value, it is pushed on the stack before returning to the caller. Having the subroutine clean the stack of passed parameters means that this code does not have to be repeated in the caller each time the subroutine is called.

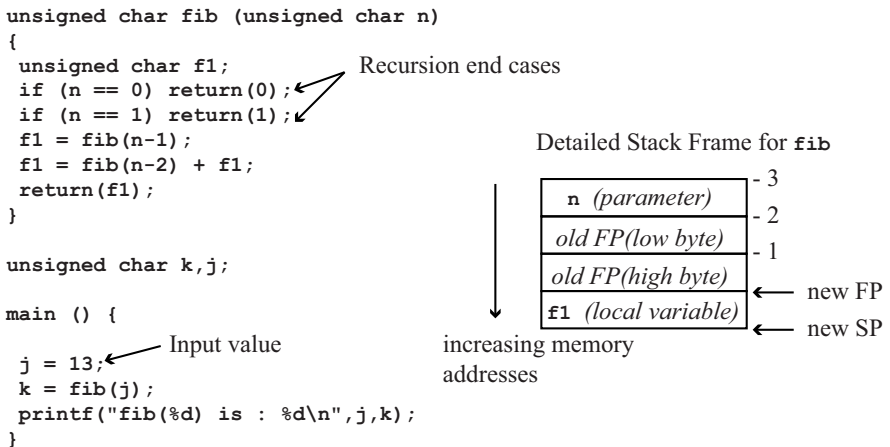
**TABLE 6.1** Subroutine Return Values

Return Value Size	Passed Back In
8-bit	W register
16-bit	PRODH:PRODL register pair
32-bit or greater	Pushed on stack by subroutine, popped by caller



**FIGURE 6.22** Steps in constructing a stack frame.

A subroutine that computes the *Fibonacci* number given an input integer is used to illustrate stack frames and recursion. The Fibonacci computation is most naturally performed by recursion, and thus is a classic example of how subroutine recursion operates. Figure 6.23 shows a C function implementation of the Fibonacci computation, and the detailed stack frame for the subroutine. A recursive subroutine must have a terminal condition where the subroutine does not call itself. If a terminal condition did not exist, the subroutine would become trapped in an infinite loop of repeatedly calling itself. For the `fib` subroutine, the terminal conditions are  $n = 0$  (returns 0) or  $n = 1$  (returns 1). For other values of  $n$ , the subroutine returns the value  $\text{fib}(n-1) + \text{fib}(n-2)$ . A local variable named `f1` is used to store the value returned by  $\text{fib}(n-1)$  before the  $\text{fib}(n-2)$  call is made.



**FIGURE 6.23** C Subroutine for Fibonacci computation and detailed stack frame.

Before converting the `fib` function to a PIC18 assembly language subroutine, a detailed stack frame is needed to determine the offsets required for accessing parameters and local variables from the frame pointer. From the stack frame in Figure 6.23, it is seen that `n` is accessed using a  $-3$  offset and `f1` by a  $0$  offset. This means that the instruction pair `movlw D' -3' ; movf PLUSW2` loads the value of parameter `n` into the W register assuming FSR2 contains the frame pointer. Table 6.2 gives the Fibonacci numbers from 0 to 17; it is seen that the use of `unsigned char` data types limits the maximum value of `n` for `fib` to a value of 13 in order to remain in the range 0 to 255.

Figure 6.24 shows the assembly code for `main()` of Figure 6.23. The stack pointer (FSR1) is initialized to location `0x27F`; the frame pointer (FSR2) value is initialized within the first subroutine call. The global variable `j` is initialized to the

**TABLE 6.2** Fibonacci Numbers 0 to 17

n	fib(n)	n	fib(n)
0	0	9	34
1	1	10	55
2	1	11	89
3	2	12	144
4	3	13	233
5	5	14	377
6	8	15	610
7	13	16	987
8	21	17	1597

**In C**  
 unsigned char k,j;

main () {

  j = 13;

  k = fib(j);

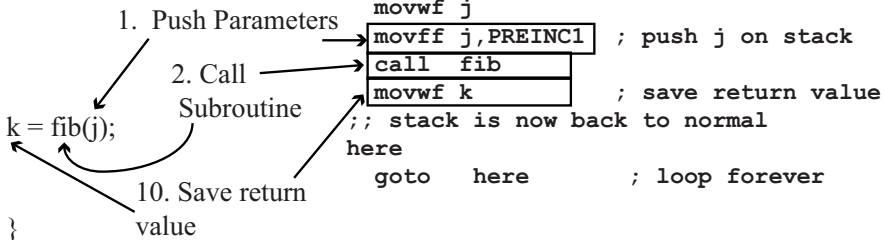
}

**In Assembly**

```
;; make stack last 128 locations
STKBASE EQU 0x27F
CBLOCK 0x00
  k,j
endc

  org 0
  goto main

  org 0x0100
main
  lfsr FSR1, STKBASE ; init stackpointer
  ;; set up subroutine call
  ;; reserve space for return value
  movlw D'13'
  movwf j
  movff j,PREINCL ; push j on stack
  call fib
  movwf k ; save return value
  ;; stack is now back to normal
  here
  goto here ; loop forever
```

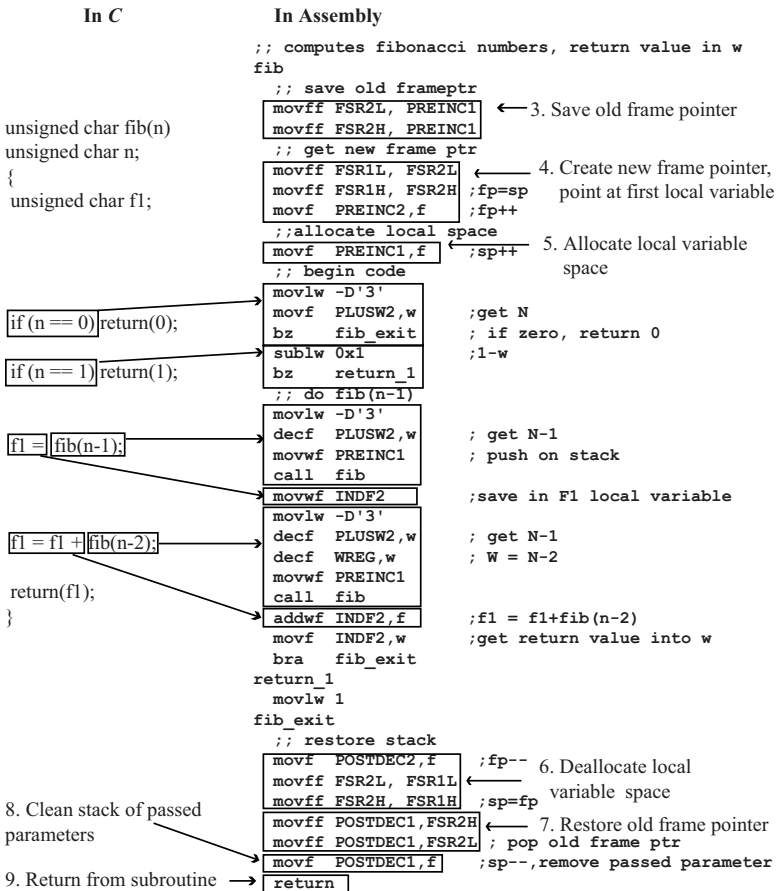


**FIGURE 6.24** Assembly implementation for main() Fibonacci C code.

value 13, and then `j` is pushed on the stack before the subroutine call (`call fib`). The `fib` return value is returned in `W` and saved to the global variable `k`. The numbering of the actions taken by `main()` corresponds to the actions found in Figure 6.22.

The assembly code for the `fib()` subroutine is seen in Figure 6.25. The numbered actions within the subroutine code correspond to those of Figure 6.22. Some observations are:

- In step 3, the bytes of the frame pointer are pushed as lower byte first, then upper byte. This orders the bytes in little-endian order on the stack because the stack grows toward increasing memory locations. In step 7, the bytes are popped off the stack in reverse order—upper byte first, then the lower byte.
- The space for the local variable `f1` is allocated on the stack in step 5 by incrementing the stack pointer by 1, since `f1` is a char variable and thus requires only 1 byte. This space is reclaimed in step 6 by transferring the frame pointer to the



**FIGURE 6.25** Assembly implementation for `fib()` Fibonacci C code.

stack pointer, as the frame pointer references the beginning of local variable space.

- The value  $n-1$  is pushed on the stack before the `fib(n-1)` subroutine call; the return value is saved to local variable `r1` by the instruction `movwf INDF2` because `r1` is at offset 0 from the frame pointer (FSR2).
- The instruction `movf POSTDEC1, f` before the subroutine return frees the space on the stack used by the input parameter  $n$ ; this increments the stack pointer (FSR1) by 1, freeing 1 byte of space.

It is obvious that dynamic memory allocation for parameters and local variables requires more complex code than static allocation. Some microprocessor instruction sets have specialized instructions for efficient creation of stack frames. Despite the complexity, use of stack frames for parameters and local variables is the best option if subroutine recursion must be supported.

**Sample Question:** *If the data type of parameter  $n$  to `fib` was changed to `int`, what is the parameter offset for  $n$  from the frame pointer?*

*Answer:* The change from `char` to `int` increases the parameter space from 1 byte to 2 bytes. The frame pointer will still point to the beginning of the local space, and the saved frame pointer will still occupy 2 bytes. Thus, the offset only changes by 1, from  $-3$  to  $-4$ . The instruction pair `movlw D'-4'; movf PLUSW2` will copy the value of  $n$  from the stack into the `W` register.

## SUMMARY

---

Subroutines improve code efficiency and clarity by encapsulating often-used code sequences as a single unit that can be called from multiple locations within a program. A stack is needed to save the return address so a subroutine can determine the return location within the calling function. Static allocation uses a fixed set of memory locations for subroutine parameter lists and local variables. While static allocation is easy to implement and requires few CPU resources, subroutine recursion is not supported, as subroutine data is overwritten during the recursive call. Dynamic allocation for parameters and local variables is required if subroutines are to support recursion. A data stack using FSR0, FSR1 as a stack pointer and frame pointer is one method to implement dynamic allocation for subroutines. A pointer register contains the address of a data memory location that is used by an instruction. Pointers allow references to data arrays to be passed to subroutines instead of the entire array, improving code execution speed. Data that needs to be stored between processor resets can be saved in program memory, which is nonvolatile, and retrieved when needed using table read instructions.



## REVIEW PROBLEMS

---

Problems that refer to initial memory contents use Table 6.3.

**TABLE 6.3** Memory Contents

Location	Contents
W	0x02
0x023	0x38
0x024	0xC7
0x025	0x9B
0x026	0xD0
0x027	0xFE

1. If the `call 0x0480` instruction is at program memory location 0x0100, what return address is pushed on the stack?
2. If the `rcall 0x0480` instruction is at program memory location 0x0100, what return address is pushed on the stack?
3. When would you have to use a `call` instruction instead of an `rcall` instruction?
4. Give the contents of any changed memory locations and/or registers after execution of the following instructions, assuming the initial memory contents of Table 6.3.

```
lfsr FSR0, 0x024
incf POSTDEC0, f
```

5. Give the contents of any changed memory locations and/or registers after execution of the following instructions, assuming the initial memory contents of Table 6.3.

```
lfsr FSR0, 0x024
decf INDF0, f
```

6. Give the contents of any changed memory locations and/or registers after execution of the following instructions, assuming the initial memory contents of Table 6.3.

```
lfsr FSR0, 0x024
incf PREINC0, f
```

7. Give the contents of any changed memory locations and/or registers after execution of the following instructions, assuming the initial memory contents of Table 6.3.

```
lfsr FSR0, 0x024
incf POSTINCO, f
```

8. Give the contents of any changed memory locations and/or registers after execution of the following instructions, assuming the initial memory contents of Table 6.3.

```
lfsr FSR0, 0x024
incf PLUSW0, f
```

9. For the following C code sequence, assume the variables begin at location 0x100. Give the contents of any changed memory locations after the code has been executed.

```
char a[] = {0x34, 0x24, 0x11, 0xFE};
char *ptr;

ptr = a;
*(ptr+1) = *(ptr+2);
```

10. Write a PIC18 assembly language sequence that implements the code of problem 9. Use pointer operations, and use FSR0 for the pointer register.
11. For the following C code sequence, assume the variables begin at location 0x100. Give the contents of any changed memory locations after the code has been executed.

```
int a[] = {-234, 120, 30000, -20000};
int *ptr;

ptr = a;
*(ptr+1) = *(ptr+2);
```

12. Write a PIC18 assembly language sequence that implements the code of problem 11. Use pointer operations, and use FSR0 for the pointer register.
13. Implement the following subroutine in PIC18 assembly language.

```
// this subroutine implements a string swap.
str_swap (char *s1, char *s2){
    char c;
    while (*s1 != 0) {
        c = *s1;
        *s1 = *s2;
        *s2 = c;
    }
}
```

```

        s1++;s2++;
    }
}

```

14. Implement the following subroutine in PIC18 assembly language.

```

// this subroutine implements an integer swap.
int_swap (int *ptr, unsigned char i, unsigned char j){
    int k;

    k = *(ptr+i);
    *(ptr+i) = *(ptr+j);
    *(ptr+j) = k;
}

```

15. Implement the following subroutine in PIC18 assembly language.

```

// this subroutine implements a long swap.
long_swap (long *ptr, unsigned char i, unsigned char j){
    long k;

    k = *(ptr+i);
    *(ptr+i) = *(ptr+j);
    *(ptr+j) = k;
}

```

16. Implement the following subroutine in PIC18 assembly language.

```

// this subroutine implements a max function.
int find_max (int *ia, unsigned char cnt){
    int k;

    k = 0;
    while (cnt != 0) {
        if (*ia > k) k = *ia;
        ia++; cnt--;
    }
    return(k);
}

```

17. Implement the following subroutine in PIC18 assembly language.

```

// this subroutine adds the contents of two integer arrays
// the number of elements to add is given by cnt
ivec_add (int *ia, int *ib, unsigned char cnt){
    while (cnt != 0) {
        *ia = *ia + *ib;
        ia++; ib++;
        cnt--;
    }
}

```

```

    }
}

```

18. Implement the `putstr()` function of the following C code as PIC18 assembly language. Assume the `putch()` function expects its input parameter to be passed in the W register.

```

putch (unsigned char c){
putstr (unsigned char *s){
}

// print string
putstr (unsigned char *s){
    while (*s != 0) {
        putch(*s);
        s++;
    }
}

```

19. Implement the `getstr()` function of the following C code as PIC18 assembly language. Assume the return value of the `getch` function is passed back via the W register.

```

unsigned char getch()
{ // not shown
}

// get string
getstr (unsigned char *s){
    unsigned char c;
    do{
        c = getch();
        *s = c;
        s++;
    }
    while (c != 0)
}

```

20. Write a PIC18 subroutine that will initialize the contents of an integer array stored in data memory with the contents of an integer array stored in program memory. Assume `FSR0` points to the integer array in data memory, `TBLPTR` to the integer array in program memory, and the W register contains the number of integers to be copied. You will probably need to use another temporary memory location to track the number of integers that have been copied from program memory to data memory.
21. For the `fib` assembly language implementation, what is the maximum value of `n` that can be computed before stack overflow occurs (exceeds

0x2FF) given the stack pointer is initialized to 0x27F? Or will the return address stack overflow before the data stack overflows?

22. Modify the `fib()` C code of Figure 6.23 to use a `long` data type instead of a `char` data type. Implement this in PIC18 assembly language in the same way as was done in Figures 6.24 and 6.25.
23. Give the detailed stack frame required for the following subroutine:

```
// this subroutine adds the contents of two integer arrays
// the number of elements to add is given by cnt
ivec_add (int *ia, int *ib, unsigned char cnt){
    while (cnt != 0) {
        *ia = *ia + *ib;
        ia++; ib++;
        cnt--;
    }
}
```

24. Implement problem #13 using dynamic parameter allocation as was done for the `fib` example.
25. Define the push and pop operations needed for a stack that grows toward decreasing memory locations.

# 7

## Advanced Assembly Language: Higher Math

### In This Chapter

- Multiplication
- Division
- Fixed-Point and Saturating Arithmetic
- Floating-Point Number Representation
- BCD Arithmetic
- ASCII Data Conversion

This chapter examines various higher math topics such as multiplication and division operations for unsigned and signed integers, floating-point number representation, saturating arithmetic, BCD arithmetic, and ASCII/binary conversions.

### 7.1 LEARNING OBJECTIVES

---

After reading this chapter, you will be able to:

- Implement signed and unsigned integer multiplication in PIC18 assembly language.
- Implement signed and unsigned integer division in PIC18 assembly language.

- Discuss the formatting and storage requirements of single and double precision floating-point numbers.
- Implement saturating addition and subtraction operations in PIC18 assembly language.
- Implement BCD addition and subtraction operations in PIC18 assembly language.
- Implement ASCII-to-binary and binary-to-ASCII for both hex and decimal number formats in PIC18 assembly language.

## 7.2 MULTIPLICATION

In C, the multiplication operation is written as  $product = multiplicand * multiplier$ . For integer multiply, the number of bits required for the product to prevent overflow is the sum of the bits in the multiplicand and multiplier. Typically, the two operands are the same size; so two  $n$ -bit operands produce a  $2n$ -bit result. Figure 7.1 shows a paper and pen multiply of two 3-bit operands that produces a 6-bit product. Starting with the rightmost bit of the multiplier, a *partial product* is formed by multiplying the multiplier bit with the multiplicand, with the rightmost bit of the partial product aligned under the multiplier bit that produced it. Since this is binary multiplication, a “1” in the multiplier produces a partial product that is equal to the multiplier, while a “0” produces a partial product of all zero bits. The product is formed from the sum of all of the partial products.

multiplicand	$r_2$	$r_1$	$r_0$		Binary	Decimal
multiplier	X $s_2$	$s_1$	$s_0$		1 1 0	6
partial product	$s_0 * r_2$ $s_0 * r_1$ $s_0 * r_0$				X <u>1 0 1</u>	X <u>5</u>
	$s_1 * r_2$ $s_1 * r_1$ $s_1 * r_0$				1 1 0	30
+	$s_2 * r_2$ $s_2 * r_1$ $s_2 * r_0$				0 0 0	
	$P_5$	$P_4$	$P_3$	$P_2$	$P_1$	$P_0$
				product	+ <u>1 1 0</u>	
					0 1 1 1 1 0	= 30

**FIGURE 7.1** 3x3 Unsigned multiply.

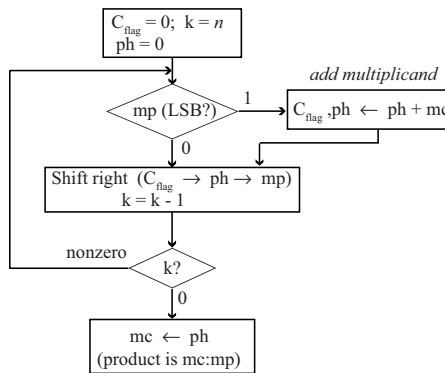
The multiplication operation can be implemented in numerous ways. The cheapest method in terms of logic gates is to not add any support for multiplication to the ALU, and to simply rely on the pre-existing microprocessor add and shift operations to perform a multiply. An algorithm for an unsigned integer multiply

using add/shifts is seen in Figure 7.2. The two  $n$ -bit operands are named  $mc$  (multiplicand) and  $mp$  (multiplier). The  $ph$  variable holds the accumulated sum of the partial products, and at algorithm termination,  $ph$  contains the upper  $n$  bits of the product. The algorithm loops  $n$  times, one for each bit of the multiplier ( $mp$ ). Each time through the loop, the LSB of  $mp$  is tested; if “1”, the multiplicand is added to the  $ph$  variable. If the LSB of  $mp$  is “0”, then addition is not performed as the partial product is zero in this case, making the addition superfluous. The shift right of the  $C_{flag}$ ,  $ph$ , and  $mp$  values accomplishes two things:

- The LSB of the  $ph$  variable is a final bit of the product; shifting  $ph$  right moves this bit into the MSb of  $mp$ , saving this bit for the result.
- The right shift moves the next bit of the multiplier ( $mp$ ) into the LSB for testing at the top of the loop. As the loop iteration proceeds, each bit of the multiplier is examined from LSB to MSb.

After  $n$  iterations, the multiplication is finished and the loop is exited. The  $ph$  variable contains the upper  $n$  bits of the product, and  $mp$  the lower  $n$  bits. The  $ph$  variable is copied to the  $mc$  variable so that the final  $2n$ -bit product returns in the original operands as  $mc:mp$ .

Unsigned Multiplication: Shift/Add Algorithm  
 $mc$  (multiplicand)  $n$ -bits,  $mp$  (multiplier)  $n$ -bits,  $ph$  (product high,  $n$ -bits),  $k$  (counter)  
 $2n$ -bit product returns in  $mc:mp$



**FIGURE 7.2** Unsigned add/shift integer multiply algorithm.

Table 7.1 shows the progress of the  $k$ ,  $C_{flag}$ ,  $ph$ ,  $mp$ ,  $mc$  values for the add/shift algorithm using the multiplication of Figure 7.1.



**TABLE 7.1** Numerical Example for Shift/Add Multiplication Algorithm

k	Cflag	ph	mp	mc	Comment
3	0	000	101	110	Initial values
3	0	110	101	110	mp(LSb)=1, so C <sub>flag</sub> ,ph = ph+mc
2	0	011	010	110	shift right C <sub>flag</sub> →ph→mp; k--
2	0	011	010	110	mp(LSb)=0, so no add
1	0	001	101	110	shift right C <sub>flag</sub> →ph→mp; k--
1	0	111	101	110	mp(LSb)=1, so C <sub>flag</sub> ,ph = ph+mc
0	0	011	110	110	shift right C <sub>flag</sub> →ph→mp; k-- Algorithm exit, product is ph:mp

An assembly language implementation of the add/shift algorithm for 8-bit operands is seen in Figure 7.3; the `mult8x8` subroutine performs the operation `mc*mp`, with the 16-bit product returning in `mc:mp`. Static memory allocation is used for the parameters and local variables of `mult8x8`.

```

org 0          CBLOCK 0x00
goto main     mp,mc,ph,k
              ENDC

org 0x0100
main
  movlw 0x2C
  movwf mc
  movlw 0xA5
  movwf mp
  call mult8x8
  ;infinite loop
  here
  goto here

; does mc * mp
; answer returns in mc:mp
mult8x8
  movlw 8      Execute loop 8 times, once for
  movwf k      each bit of multiplier
  clrf ph
  mult_loop ←
  bcf STATUS,C ; clear C flag, add affects
  btfss mp,0   ; test LSb of mp
  bra mult_1   ; LSb = 0, do shift
  movf mc,w    ; LSb = 1, add mc+ph
  addwf ph,f
  mult_1 ←
  rrcf ph,f    ; right shift Cflag,ph,mp
  rrcf mp,f
  decfsz k,f   ; k--
  bra mult_loop ; loop if k non-zero
  movff ph,mc
  return
  k is zero, exit

```

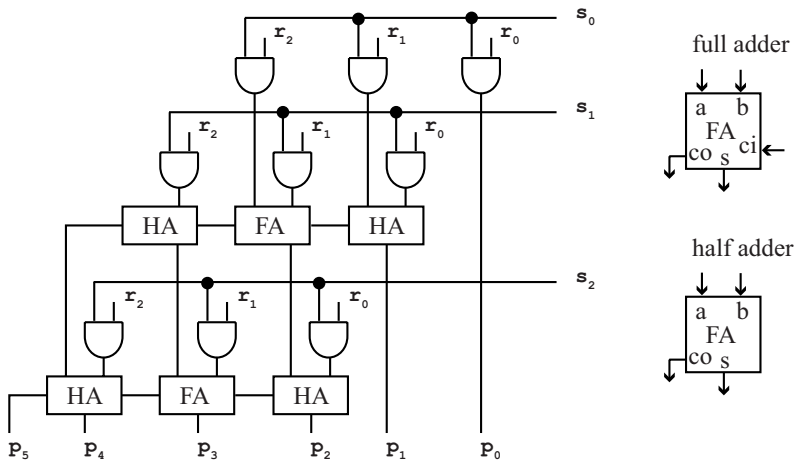
Initialize parameters for call to `mult8x8`

mp LSb is 1



**FIGURE 7.3** Assembly language implementation of add/shift multiply.

The disadvantage of the shift/add technique for multiplication is obvious—it is slow! Each bit of the multiplier requires iteration through the loop of Figure 7.3 between the `mult_loop` label and the `bra mult_loop` instruction. This takes 9 to 10 instruction cycles (36 to 40 clock cycles) depending on the LSb test, requiring 72 to 80 instruction cycles (288 to 320 clock cycles) for the 8x8 unsigned multiply, not counting the overhead for subroutine call/return and loop entry/exit. If hardware support for the shift/add iteration is added to the ALU in the form of a double-length shift register for the product and specialized control, this can be reduced to one clock cycle per loop iteration (eight clock cycles). While this would be an improvement, a faster method is to augment the ALU with a specially designed multiplier unit such as *array multiplier* that produces the result in one clock cycle. Figure 7.4 shows a naive implementation of a 3x3 array multiplier that performs the operation of Figure 7.1. There are more efficient methods for constructing array multipliers, but this conveys the key point of an array multiplier: the product is available a combinational delay after the inputs are applied. This means the multiplication is completed in one clock cycle (if the clock cycle is long enough). The origin of the term *array multiplier* is obvious from Figure 7.4, as it is built from an array of full-adders and half-adders that implements the addition of the partial products. Observe that binary multiplication for each partial product bit is simply an AND gate, as the Boolean multiply  $a * b$  is a “1” only if both inputs are “1”.



**FIGURE 7.4** Naive 3x3 array multiplier.

The PIC18 has an 8x8 array multiplier, whose 16-bit result is placed in the PRODH, PRODL register pair. Two instructions make use of the array multiplier:

**mulw f [,a]:** Multiply W with *f*, the 16-bit product returns in PRODH:PRODL.  
**mullw k:** Multiply W with 8-bit literal *k*, the 16-bit product returns in PRODH:PRODL.

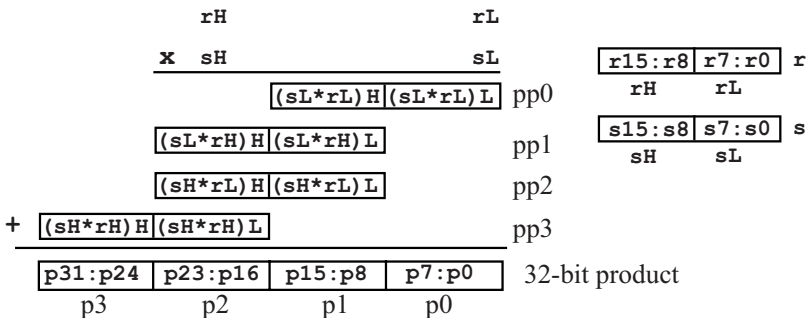
Neither instruction affects status flags, and both require only one instruction cycle. Listing 7.1 gives the `mult8x8` subroutine of Figure 7.2 implemented using the `mulwf` instruction. The advantages in terms of code size and speed in using the `mulwf` instruction versus a shift/add approach are obvious.



**LISTING 7.1** Multiplication using MULWF.

```
;; do 8x8 mult using hw instruction
mult8x8
    movf    mc,w
    mulwf   mp        ; PRODH:PRODL = mp * W
    movff   PRODL,mp
    movff   PRODH,mc
    return
```

The shift/add approach in Figure 7.2 is scalable, in that it can be applied to any size operands. It is straightforward to extend the code of Figure 7.2 to use `int` (16x16) or `long` (32x32) operands. Using the 8x8 hardware multiply of the `multw` instruction in larger operations requires more effort than scaling the algorithm of Figure 7.2. To illustrate, a 16x16 multiplication `s*r` is performed using 8x8 multiplication operations in Figure 7.5. Four 16-bit partial products are formed as `pp0=sL*rL`, `pp1=sL*rH`, `pp2=sH*rL`, and `pp3=sH*rH` where `{sL, rL}` and `{sH, rH}` are the lower and upper bytes of the 16-bit values `s, r`. Observe that the partial products `pp1, pp2` are shifted to the left such that the lower bytes of these partial products align with the upper byte of `pp0`; the lower byte of `pp3` is aligned with the upper bytes of `pp1, pp2`. When performing the byte additions of the partial products, care must be taken to propagate the carries during the summation.



**FIGURE 7.5** Unsigned 16x16 multiplication as 8x8 multiplication.

A subroutine that implements the 16x16 multiply of Figure 7.5 is seen in Listing 7.2. The partial products pp0, pp3 are computed first, and copied to product bytes p0:p1, and p2:p3, respectively. The partial product pp1 is computed next, and the sums p1+pp1L, p2+pp1H+C<sub>flag</sub>, p3+0+C<sub>flag</sub> are performed in order. The final mulwf instruction computes pp2, and the sums p1+pp2L, p2+pp2H+C<sub>flag</sub>, p3+0+C<sub>flag</sub> are computed to generate the final 32-bit product.



**LISTING 7.2** Assembly code for unsigned 16x16 using MULWF.

```

CBLOCK 0x00
    mp:2,mc:2,p:4
    endc

    ;; do p = mc * mp,
    ;; 16x16 mult using mulwf
mult16x16
    movf mp,w
    mulwf mc           ;pp0 = mpL * mcL
    movff PRODL,p
    movff PRODH,p+1   ; save pp0
    movf mp+1,w
    mulwf mc+1        ;pp3 = mpH * mcH
    movff PRODL,p+2
    movff PRODH,p+3   ;pp3

    movf mp,w
    mulwf mc+1        ; pp1 = mpL * mcH

    movf PRODL,w
    addwf p+1,f       ;p1 + pp1L
    movf PRODH,w
    addwfc p+2,f      ;p2 + pp1H + carry
    clrfs WREG
    addwfc p+3,f      ; p3 + zero + carry

    movf mp+1,w
    mulwf mc          ; pp2 = mpH *mcL

    movf PRODL,w
    addwf p+1,f       ;p1 + pp2L
    movf PRODH,w
    addwfc p+2,f      ;p2 + pp2H + carry
    clrfs WREG
    addwfc p+3,f      ;p3 + carry
    return

```

Signed multiplication requires different hardware or instruction sequences than unsigned multiplication does. Many microprocessors provide both unsigned and signed multiply instructions. If a microprocessor has no hardware support for multiply, a shift/add/subtract approach called *Booth's algorithm* can be used to

iteratively determine the product in a manner similar to that of Figure 7.2. A good discussion of Booth's algorithm can be found in [3]. Most modern processors have hardware support for at least unsigned multiply, and this can be used to create a signed multiply. One simple approach is to convert each operand to positive numbers, perform the unsigned multiply, and then negate the product by subtracting it from 0 if either of the operands was originally negative. However, there is a more efficient method. Consider the multiplication in Equation 7.1.

$$P = A * (-1) \quad (7.1)$$

Equation 7.2 shows the operation of Equation 7.1 as 8-bit numbers using an unsigned multiply.

$$P = A * 255 \quad (7.2)$$

The  $-1$  in Equation 7.1 is  $0xFF$  in 8 bits, or 255 as an unsigned number as seen in Equation 7.2. To reach the correct product of Equation 7.1, Equation 7.2 is rewritten as shown in Equation 7.3.

$$P = A * (255 - 256) = A * 255 - A * 256 = A * 255 - (A \ll 8) \quad (7.3)$$

Equation 7.3 indicates that the product of the unsigned multiply must be post-corrected by subtracting  $A * 256$  or  $A \ll 8$  from the product. For an  $8 \times 8$  unsigned multiply, this is the same as subtracting  $A$  from the upper byte of the 16-bit product, as the lower byte of  $A \ll 8$  is a zero value. Table 7.2 lists the steps for a signed  $n \times n$  multiply using an unsigned  $n \times n$  multiply.

**TABLE 7.2** Signed Multiply Algorithm Using Unsigned Multiply

#### Steps

1. Perform the  $A * B$  unsigned multiply, each operand is  $n$  bits.
2. If  $B$  is negative, subtract  $A \ll n$  from the product.
3. If  $A$  is negative, subtract  $B \ll n$  from the product.

Listing 7.3 shows the algorithm of Table 7.2 implemented in PIC18 assembly code using the `mulwf` instruction. This approach is efficient in that it requires no extra memory locations, but it is still double the number of instructions of the unsigned  $8 \times 8$  multiply of Listing 7.1. Adding support for signed operands to the  $16 \times 16$

unsigned multiply of Listing 7.1 requires 12 additional instructions, an increase of about one third.




---

**LISTING 7.3** Signed 8x8 multiply using `mulwf` instruction.
 

---

```

;; do 8x8 signed mult using mulwf instruction
mult8x8
  movf mc,w
  mulwf mp
  btfsc mp,7      ;; test sign of mp
  subwf PRODH,f  ;; mp negative, subtract mc << 8
  movf mp,w
  btfsc mc,7      ;; test sign of mc
  subwf PRODH,f  ;; mc negative, subtract mp << 8
  movff PRODL,mp
  movff PRODH,mc
  return
  
```

**Sample Question:** What does the product  $0x3A * 0xA8$  return if the numbers are unsigned? signed? (two's complement)

*Answer:* As unsigned numbers, the product is  $0x3A * 0xA8 = 58 * 168 = 9744 = 0x2610$ .

As signed numbers, the product is  $0x3A * 0xA8 = +58 * (-88) = -5104 = 0xEC10$ .

---

## 7.3 DIVISION

---

Equation 7.4 represents the division operation, where  $p$  is the dividend,  $q$  is the quotient,  $d$  is the divisor, and  $r$  is the remainder.

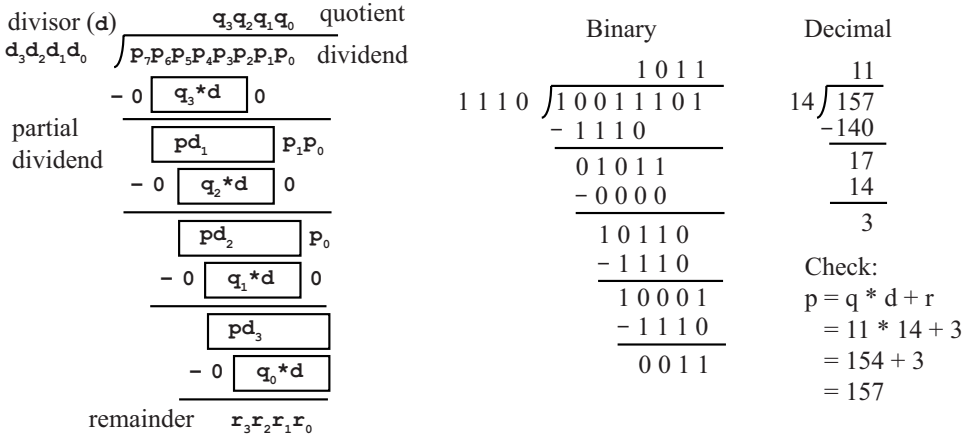
$$q, r = \frac{p}{d} \quad (7.4)$$

The relationship between  $q$ ,  $r$ ,  $p$ , and  $d$  is more clearly expressed by Equation 7.5.

$$p = q \times d + r \quad (7.5)$$

Implementations of the division operation typically use a  $2n$ -bit dividend, an  $n$ -bit divisor, and produce an  $n$ -bit quotient and  $n$ -bit remainder. Figure 7.6 shows a paper and pen division of an 8-bit dividend by a 4-bit quotient, producing a 4-bit quotient and a 4-bit remainder. The subtraction performed at each step produces a *partial dividend*, which forms the dividend for the next stage. The last subtraction produces the remainder, which is guaranteed to be in the range 0 to  $d-1$ . Unlike

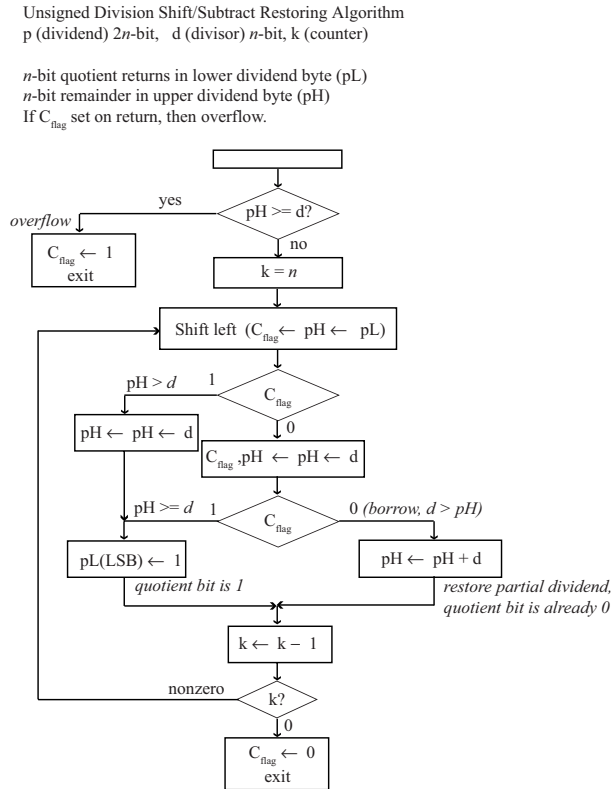
multiplication, overflow can occur if the quotient requires more than  $n$  bits, which is true if the value formed by the upper  $n$  bits of the dividend is greater than or equal to the divisor.



**FIGURE 7.6** Unsigned division (8-bit dividend, 4-bit quotient).

Several iterative division algorithms use shift/subtract operations to produce the quotient and remainder. Figure 7.7 shows the *restoring* division algorithm for a  $2n$ -bit dividend  $p$  and an  $n$ -bit divisor  $d$ . The high and low bytes of  $p$  are designated as  $p_H$  and  $p_L$ , respectively. On algorithm entry, the comparison  $p_H \geq d$  is performed to check for overflow; if true, the Carry flag is set to “1” and the algorithm terminates. Like the add/shift multiplication algorithm, the main loop performs  $n$  iterations, with each iteration determining a quotient bit. A quotient bit is “1” if the partial dividend  $C_{flag}, p_H \geq d$ , in which case the new partial dividend is  $p_H - d$ . A quotient bit is “0” if  $C_{flag}, p_H < d$ , and the partial dividend remains the same. The first operation of the loop performs the shift  $C_{flag} \leftarrow p_H \leftarrow p_L$ , moving the partial dividend into the  $C_{flag}$  and upper  $n$  bits of  $p_H$ . If the shift produces a carry, the partial dividend is greater than the divisor, so the new partial dividend is computed as  $p_H \leftarrow p_H - d$  and the quotient bit is set to “1” (the LSB of  $p_L$  is the current quotient bit). If no carry is produced by the shift, the subtraction  $p_H \leftarrow p_H - d$  is performed, which has two side effects: a) the comparison  $d > p_H$  is determined by the state of the  $C_{flag}$ , and b) the new partial dividend is computed. If  $C_{flag}$  is a “1”, this indicates that no borrow occurred, so  $p_H \geq d$ , the new partial dividend is valid, and the quotient bit is set to “1”. If  $C_{flag}$  is a “0”, a borrow occurred, so  $p_H < d$  meaning the partial dividend should have been left undisturbed. In this case, the operation  $p_H \leftarrow p_H + d$  is performed to *restore* the partial dividend to its original value and the quotient

bit is cleared to “0”. This action of performing the subtraction  $pH \leftarrow pH - d$  and then restoring the partial dividend, if necessary, is why this algorithm is called *restoring* division. When the loop terminates after  $n$  iterations, the quotient is contained in  $pL$ , and the remainder in  $pH$ .



**FIGURE 7.7** Unsigned restoring division algorithm.

Table 7.3 shows the restoring division algorithm steps for the binary division of Figure 7.6. The execution time of the algorithm is dependent upon the number of restoring steps needed, as each restore requires an extra addition operation.



**TABLE 7.3** Numerical Example for Restoring Division Algorithm

k	Cflag	pH	pL	d	Comment
4	0	1001	1101	1110	Initial values
4	1	0011	1010	1110	shift left Cflag← pH← pL
4	x	0101	1010	1110	Cflag =1 in previous step, do pH← pH-d
3	x	0101	1011	1110	pL(MSb) ← 1; k--
3	0	1011	0110	1110	shift left Cflag← pH← pL
3	0	0101	0110	1110	Cflag =0 in previous step, do Cflag ,pH← pH-d
2	x	1011	0110	1110	Cflag =0 in previous step, restore by pH← pH+d; k--
2	1	0110	1100	1110	shift left Cflag← pH← pL
2	x	1000	1100	1110	Cflag =1 in previous step, do pH← pH-d
1	x	1000	1101	1110	pL(MSb) ← 1; k--
1	1	0001	1010	1110	shift left Cflag← pH← pL
1	x	0011	1010	1110	Cflag =1 in previous step, do pH← pH-d
0	x	0011	1011	1110	pL(MSb) ← 1; k--
0	x	0011	1011	1110	Algorithm exit, pH is remainder, pL is quotient

An assembly language subroutine that implements restoring division for a 16-bit dividend and an 8-bit quotient is seen in Listing 7.4. The worst-case execution time through the loop is 13 instruction cycles (52 clock cycles), resulting in approximately 104 instruction cycles (416 clock cycles) worst-case execution time required for completion, not counting loop setup/exit and subroutine call/return. It is possible to augment the ALU with extra logic to support division operations, but it is a more difficult task to speed up division than multiplication. Even on high-performance microprocessors that have division hardware support, integer division is 5 to 30 times slower than multiplication [4].

**LISTING 7.4** Restoring division subroutine for 16-bit dividend, 8-bit divisor.

```

div16_8
    ;check for overflow
    movf    d,w
    subwf   p+1,w      ;p MSByte >= d?
    bc     div_exit    ;if C=1, then overflow
    movlw   8
    movwf   k          ;loop counter is 8
div_loop

```

```

movf    d,w          ;load divisor into W
bcf     STATUS,C
rlcf   p
rlcf   p+1          ;shift left of dividend
bc     div_dosub    ;if C=1, pd > d, do sub
subwf  p+1,f        ;create pd
bc     div_q_1      ;if C=1, pd>=d, quotient bit is 1
addwf  p+1,f        ;restore pd
bra    div_dec_k
div_dosub
subwf  p+1,f        ;create pd
div_q_1
bsf    p,0          ;quotient bit is '1'
div_dec_k
decfsz k,f          ;k--
bra    div_loop
bcf    STATUS,C     ;no overflow, clear carry
div_exit
return

```

Signed division can be accomplished by converting the dividend and divisor to positive numbers, performing the unsigned division, and then post-correcting the sign of the quotient and remainder based on the sign of the dividend and remainder. Table 7.4 gives the steps for performing signed division using an unsigned division operation.

**TABLE 7.4** Signed Division Algorithm Using Unsigned Division

#### Steps

1. Convert dividend and divisor to positive numbers by subtracting from 0 if negative.
2. Perform the unsigned division.
3. If either of the original operands is negative, make the quotient negative by subtracting from zero.
4. If the original dividend is negative, make the remainder negative by subtracting from zero.

Unlike multiplication, if a microprocessor has an explicit integer division instruction, such as a 16-bit/8-bit operation, it is not possible to use this instruction in the implementation of a larger operation such as a 32-bit/16-bit division. As such, some microprocessors offer different sized integer division operations such as 16-bit/8-bit, 32-bit/16-bit, and 64-bit/32-bit.

**Sample Question:** What does the operation  $0x2EF0 \div 0x8D$  return if the numbers are unsigned? signed (two's complement)?

**Answer:** As unsigned numbers,  $0x2EF0 \div 0x8D = 12016 \div 141 = 85$  (0x55) quotient, 31 (0x1F) is the remainder. As a check, quotient \* divisor + remainder = dividend, or  $85 * 141 + 31 = 12016$ .

As signed numbers,  $0x2EF0 \div 0x8D = 12016 \div (-115) = -104$  (0x98) quotient, 56 (0x38) is the remainder. As a check, quotient \* divisor + remainder = dividend, or  $(-104) * (-115) + 56 = 12016$ .

## 7.4 FIXED-POINT AND SATURATING ARITHMETIC

Up to this point, we have viewed binary integers as having the decimal point always located to the right of the least significant bit. The formal name for this type of representation is *fixed-point*, because the decimal point is *fixed* to a particular location. The decimal point can be positioned in any location within the binary number, as long as the application is consistent about where the decimal point is located. A fixed-point binary number is said to have the format  $x.y$ , where  $x$  is the number of digits to the left of the decimal point (integer portion) and  $y$  is the number of digits to the right of the decimal point (fractional portion). The integer portion of the number has range 0 to  $2^{x-1}$ , while the fractional range is 0 to  $(1 - 2^{-y})$ . The 8-bit unsigned integer representation used to this point has thus been 8.0 fixed-point numbers. A 0.8 fixed-point number has a number range of 0 to  $(1 - 2^{-8})$ , or 0 to approximately 0.9961. A 6.2 fixed-point number has a number range 0 to 63.75. Table 7.5 shows examples of different 8-bit fixed-point formats.

**TABLE 7.5** Sample Fixed-Point Formats

Format	Min	Max	Example
8.0	0	255	$0xA7 = 10100111 = 167$
6.2	0	63.75	$0xA7 = 101001.11 = 41.75$
4.4	0	15.9375	$0xA7 = 1010.0111 = 10.4375$
0.8	0	0.99609375	$0xA7 = 0.10100111 = 0.65234375$

### Decimal to $x.y$ Binary Format

An unsigned decimal number is converted to its fixed-point representation by converting the integer and fractional portions separately. The integer portion is

converted to binary following the procedures given in Chapter 1, “Number System and Digital Logic Review.” The fractional portion  $f$  is converted to binary through an iterative process of performing the comparison  $f \times 2 \geq 1$ ; if this is true, the new binary digit is “1” and the new fractional part is  $f = f \times 2 - 1$ . If  $f \times 2 < 1$ , the new binary digit is “0”, and the new fractional part is  $f = f \times 2$ . The binary digits of the fractional part are determined left to right (most significant to least significant). The process stops when  $y$  binary bits of the final  $x.y$  binary result have been computed. The fractional portion of the final  $x.y$  binary result may only be an approximation of the fractional decimal part, as there may not be enough  $y$  digits to accurately represent the decimal fraction portion. Obviously, the more bits used for  $y$  in the  $x.y$  format, the better the approximation.

**Sample Question: Convert 13.365 to a binary 8-bit number with 4.4 fixed-point format.**

*Answer:* The integer portion 13 has the binary value 1101. The following steps do the conversion of the fractional portion 0.365 to its 4-bit binary representation.

1.  $0.365 \times 2 = 0.73$ , which is  $< 1$ . The first (leftmost) binary digit is 0, and the new  $f$  is 0.73.
2.  $0.73 \times 2 = 1.46$ , which is  $> 1$ . The second binary digit is 1, and the new  $f$  is  $1.46 - 1 = 0.46$ .
3.  $0.46 \times 2 = 0.92$ , which is  $< 1$ . The third binary digit is 0; the new  $f$  is 0.92.
4.  $0.92 \times 2 = 1.84$ , which is  $> 1$ . The fourth and last binary digit is 1.

The value 13.365 converted to a 4.4 binary format is 0b11010101, or 0xD5.

### ***x.y* Binary Format to Decimal**

A fixed-point binary number is converted to decimal by multiplying each bit by its appropriate binary weight. The fractional bits have weights  $2^{-1}$ ,  $2^{-2}$ , ... to  $2^{-y}$  going from leftmost bit to rightmost bit. Another method is to view the  $n$ -bit number as an  $n.0$  fixed-point number and divide that number by  $2^y$  to get the  $x.y$  decimal value. Observe that dividing by  $2^y$  is the same as shifting the  $n.0$  fixed-point number to the right by  $y$  positions.

**Sample Question: Convert 0xD5, an unsigned 4.4 binary number, to its decimal value.**

*Answer:* The value 0xD5 is 0b11010101, so the integer portion is 1101, or 13. The fractional portion 0101 is (left to right):

$$0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-3} = 0 + 0.25 + 0 + 0.0625 = 0.3125. \quad \rightarrow$$

Thus, 0xD5, an unsigned 4.4 binary number, is 13.3125. Note that the value 0xD5 was the result obtained in the previous sample problem when converting 13.365 to a 4.4 binary format. This indicates the approximation that occurs in the decimal to fixed-point binary conversion because of the limited number of bits in the fractional portion. An alternate method is to note that 0xD5 is the value 213 as an 8.0 fixed-point number and compute  $213/(2^4) = 213/16 = 13.3125$ .

### 0.n Fixed-Point Format and Saturating Operations

In the coverage of the multiplication operation, you may have noticed a troubling problem: to prevent overflow, the size of the operands have to keep doubling! For example, an 8x8 multiplication produces a 16-bit product. If this value is then used in a subsequent 16x16 multiplication operation, a 32-bit product is produced. Note that the product size doubles again to 64 bits if the previous 32-bit product is used in a 32x32 multiplication. Obviously, it not possible to keep doubling the size of the operands in each multiplication, and so eventually an  $n$ -bit value must be used to hold the result of an  $n \times n$  bit multiplication. If the operands are viewed as unsigned integers between 0 and  $2^{n-1}$ , overflow occurs if the upper  $n$ -bit value of the actual  $2n$ -bit product is nonzero. When overflow does occur, either for multiplication, addition, or subtraction, what can be done about it? In some cases, it is sufficient to simply set an error flag and let the higher level application code deal with the problem. In other cases, such as real-time digital signal processing applications like audio or video data manipulation, there is no way to halt the system to “fix” the overflow problem. One approach to keep functioning in the presence of overflow is to produce a value that is a reasonable approximation of the correct answer. The 0.n fixed-point format is often used for data in digital signal processing applications, as it has advantages in regard to multiplication overflow and using the same sized operands for all operations. Numbers in 0.n fixed-point format have the range [0,1) (up to 1 but not including 1), where the maximum value gets closer to 1 as  $n$  increases. When two 0.n fixed-point numbers are multiplied, the *upper*  $n$  bits of the  $2n$ -bit product are kept, while the lower  $n$  bits are discarded to keep the resulting product size as  $n$  bits. The lower  $n$  bits of the  $0.2n$  product that are discarded are the least significant bits, which are the bits that one wants to be discarded if precision has to be limited. With the 0.n fixed-point representation, the multiplication operation cannot overflow, because the result is always in the range [0,1). Also, while the result is not the exact product since bits have been discarded, it is a good approximation of the correct product.

It would be nice to have addition and subtraction operations that performed in a similar manner with regard to overflow; that is, when overflow occurs, a value is returned that is a close approximation of the correct result. *Saturating* addition and

subtraction operations clip results to maximum and minimum values in the presence of overflow or underflow. Figure 7.8 shows examples of unsigned saturating addition and subtraction for 8-bit numbers. On unsigned overflow (carry out of the most significant bit), the result is saturated to all “1”s, which is the maximum unsigned value. On unsigned underflow (borrow from the most significant bit), the result is clipped to the minimum value of zero. It is clear that the unsaturated results are nonsensical when overflow occurs, while the saturated results return the closest possible approximation given the range limits.

unsaturating add	8.0 format	0.8 format	saturating add	8.0 format	0.8 format
$0x60$	96	0.375	$0x60$	96	0.375
+ $0xA7$	+ $\frac{167}{7}$	+0.65234375	+ $0xA7$	+ $\frac{167}{255}$	+0.65234375
$0x07$		$0.02734375$	$0xFF$		$0.99609375$
-----					
unsaturating subtraction	8.0 format	0.8 format	saturating subtraction	8.0 format	0.8 format
$0x60$	96	0.375	$0x60$	96	0.375
- $0xA7$	- $\frac{167}{185}$	-0.65234375	- $0xA7$	- $\frac{167}{0}$	-0.65234375
$0xB9$		$0.77265625$	$0x00$		$0.0$

**FIGURE 7.8** Unsigned saturating addition/subtraction examples.

Listing 7.5 shows assembly code for  $j = j+i$  implemented as unsigned, 8-bit saturating addition. If the carry flag is set after the `addwf` instruction, the `setf` instruction is used to set the `j` result to all “1”s. Signed saturating addition clips values to either the maximum positive value or maximum negative value on two’s complement overflow. Some microprocessors, especially those touted as being especially suited for digital signal processing applications, have specialized instructions that directly implement saturating arithmetic. The C language does not have saturating arithmetic operators or data types, and thus saturating arithmetic must be implemented as a specialized library of function calls.

**LISTING 7.5** Assembly code for unsigned 8-bit saturating addition.

```

satadd_8bit
    movf    i,w
    addwf   j,f
    bnc     skip_1
    setf    j,f ;saturate
skip_
    ....rest of code...
    
```

**Sample Question: What does the sum  $0xA0 + 0x90$  equal as a saturated unsigned addition? As a saturated signed addition?**

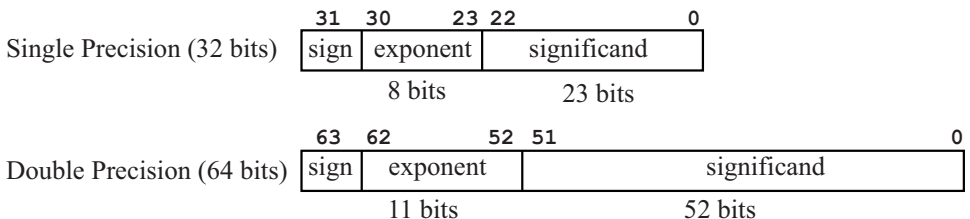
*Answer:* With binary addition, the sum  $0xA0 + 0x90 = 0x30$ , with  $C = 1$ ,  $V = 1$ . As a saturated unsigned addition, the result clips to the maximum unsigned value of  $0xFF$  because unsigned overflow occurred ( $C = 1$ ). As a saturated signed addition, the result clips to the maximum signed negative value of  $0x80$  because two negative numbers were added and the two's complement overflow occurred ( $V = 1$ ).

## 7.5 FLOATING-POINT NUMBER REPRESENTATION

Fixed-point representation forces an application to determine a priori the number of bits to devote to the integer and fractional parts. More bits used for the integer portion means less precision for the fractional part, and vice versa. *Floating-point* (FP) representation encodes an exponent field in the binary encoding, removing the need to allocate a fixed number of bits for the integer and fractional representation. This section gives a brief overview of floating-point number encoding and floating-point number operations in microprocessors; a more detailed discussion is found in [5].

### IEEE 754 Floating-Point Encoding

Many different encodings for floating-point numbers have been proposed and used over the years, but in 1985, after a long review process, the IEEE 754 Standard for Binary Floating-Point Arithmetic was approved. Figure 7.9 shows the formats for single and double precision floating-point numbers in IEEE 754 format. The single precision format is 32 bits, while the double precision format is 64 bits. Each encoding is divided into sign, exponent, and significand fields. The use of these fields to produce a floating-point number is given by Equation 7.6.



**FIGURE 7.9** Single precision and double precision FP formats.

$$(-1)^s \times 1.\textit{significand} \times 2^{(\textit{exponent}-\textit{bias})} \quad (7.6)$$

This is a signed magnitude encoding, so the most significant bit is the sign bit, which is “1” if the number is negative, and “0” if positive. The *significand* field determines the *precision* of the floating-point number. You can view the significand as encoding both the integer and fractional parts of the floating-point number. The *exponent* field determines the *range* of the floating-point number. For the single precision format, the exponent field is 8 bits and is encoded in *bias 127*, which means that 127 has to be subtracted from the field value to determine the actual exponent. For normal floating-point numbers, the exponent codes 0x01 through 0xFE are allowed. The exponent encodings 0x00 (all 0s) and 0xFF (all 1s) are reserved for so-called *special* numbers, discussed later in this section. Thus, the exponent range for single precision, IEEE 754 floating-point numbers is  $2^{+127}$  ( $10^{+38}$ ) to  $2^{-126}$  ( $10^{-38}$ ). The double precision format uses an 11-bit exponent field, with a bias value of 1023. The exponent range for double precision, IEEE 754 floating-point numbers is  $2^{+1023}$  ( $10^{+307}$ ) to  $2^{-1022}$  ( $10^{-308}$ ). In the C language, the `float` and `double` types are used for single precision and double precision floating-point variables, respectively. The MPLAB assembler does not support specification of floating-point values as a data type; you must convert decimal floating-point numbers to their equivalent byte encodings manually.

Figure 7.10 shows an example of converting a decimal number to its single-precision, floating-point number representation. First, the decimal number is converted to its binary representation by converting the integer and fractional parts to binary. The binary number is then *normalized* to the form of Equation 7.6 by shifting the number to the left or right. Each time the number is shifted to the left (multiplied by 2), the exponent is decremented by 1. Each time the number is shifted to the right (divided by 2), the exponent is incremented by 1. Observe that the “1” to the left of the decimal point in Equation 7.6 is *not* encoded in the significand; it is *understood* to be in the encoding. This is called a *phantom one bit*, and provides an extra bit of precision to the significand without having to provide space for it in the significand field.

Converting a binary value in single precision FP format to its decimal representation is done by simply converting each component of Equation 7.6 to its decimal representation and multiplying as seen in Figure 7.11. The most common error in this conversion is to forget to add the phantom one bit to the significand.

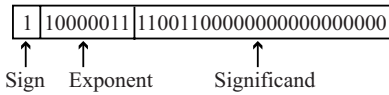
The all ones and all zero exponent encodings are reserved for *special number* encoding. Special numbers are zero, positive/negative infinity ( $\pm \infty$ ), and NaN (Not a Number). Table 7.6 gives the encodings for special numbers. Infinity is produced when anything is divided by zero. A NaN is produced by invalid operations, such as zero divided by zero, or infinity minus infinity.



Convert -28.75 to single precision floating-point format.

1. Number is negative, so sign bit is 1.
2. Convert 28.75 to binary.  
 Integer portion:  $28 = 0x1C = 0b11100$   
 Fractional portion:  $0.75 = 0b11$   
 So,  $28.75 = 11100.11 \times 2^0$
3. Normalize so that number is the form  $1.mmm \times 2^n$   
 Shift right  $11100.11 \times 2^0 \gg 1 = 1110.011 \times 2^1$   
 $11100.11 \times 2^0 \gg 2 = 111.0011 \times 2^2$   
 $11100.11 \times 2^0 \gg 3 = 11.10011 \times 2^3$   
 $11100.11 \times 2^0 \gg 4 = 1.110011 \times 2^4$  (this is normalized form)
3. Determine the bit fields.  
 Sign bit = 1  
 Exponent field =  $4 + 127 = 131 = 0x83 = 0b10000011$   
 Significand field =  $0b110011000\dots0$  (23 bits total)

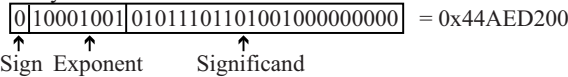
Final Result:



**FIGURE 7.10** Decimal to single precision FP format conversion.

The value 0x44AED200 is a single precision floating-point number, find the decimal value.

In Binary:



1. Sign bit is 0, so number is positive.
  2. Exponent field is  $0b10001001 = 0x89$ , so exponent is  
 $0x89 - 127 = 137 - 127 = 10$ .
  3. Number is:  $+1.01011101101001 \times 2^{10}$   
 $= +10101110110.1001 \times 2^0$   
← Do not forget phantom bit!!!
  4. Integer portion  $0b10101110110 = 0x576 = 1398$   
 Fractional portion  $0b0.1001 = 1 \times 2^{-1} + 1 \times 2^{-4} = 0.5 + 0.0625 = 0.5625$
- So,  $0x44AED200 = +1398.5625$

**FIGURE 7.11** Single precision FP format to decimal conversion.

**TABLE 7.6** Special Number Encodings

Special Number	Encoding
zero	All fields are zero
$\pm \infty$	Exponent field all ones, significand is zero
NaN	Exponent field all ones, significand is nonzero

**Sample Question:** What does the 32-bit value 0xFF800000 represent as a single precision floating-point number?

*Answer:* The value is 0xFF800000 = 0b1111 1111 1000 0000 0000 0000 0000 0000.

Grouping this into fields produces sign bit = 1, exponent = 1111 1111, significand is all zeros. By Table 7.6, this value is  $-\infty$  (negative infinity).

## Floating-Point Operations

A complete discussion of the implementation of floating-point arithmetic is beyond the scope of this book. To provide a glimpse at what is involved in performing floating-point arithmetic, the basic steps required to perform a floating-point addition are given in Table 7.7. The hardware required to support floating-point operations is much more complex than fixed-point arithmetic. Speed comparisons of floating-point instructions versus integer instructions depend greatly on the particular hardware floating-point implementation. In comparing single-precision floating-point operations versus integer operations on the Intel Pentium® IV processor, FP add/subtraction instructions are five times slower than integer operations, FP multiplication is about two times slower than integer multiplication, and FP division is actually faster than integer division [4].

**TABLE 7.7** Floating-Point Addition

Steps
1. Detect special numbers in operands and handle these boundary cases.
2. For nonspecial number operands, align the decimal points of the significands of the two operands (make the exponent fields equal) by shifting one of the operands to the left or right as many times as necessary. This process is called denormalization.
3. Add the significands.
4. Normalize the result.

For microprocessors like the PIC18 that do not have floating-point hardware support, library subroutines are used to implement floating-point operations. To provide a feel for the relative time differences between integer and floating-point operations on the PIC18, the C code in Listing 7.6 was used to test different operations (addition, multiplication, division) using `long` and `float` data types. Each loop iteration reads values from two arrays, performs an arithmetic operation on those values, and stores the result in a third array.

**LISTING 7.6** Simple C benchmark for `long` vs `float` operations.

```
for (i=0; i< 10; i++) {
    *ptrc = *ptra + *ptrb;    // add contents of two arrays
    ptra++; ptrb++; ptrc++;
}
```

Table 7.8 gives the instruction cycles per loop iteration of the code in Listing 7.6 for the three different operations tested with `long` and `float` data types. The code was compiled with the C compiler in this textbook using the optimization option, and executed in the MPLAB simulator. You may be surprised that the `float` to `long` ratio of the loop iteration times is not any higher than shown given the complexity of floating-point arithmetic. Each loop iteration contains significant memory access overhead in reading the arrays, updating pointers, and saving the result, which is the same overhead regardless of the operation that is used. This overhead is present in any realistic application and thus should be considered when comparing operation execution times. The data movement overhead is especially significant when the operand data size does not match the natural data size of the processor, which is true in this case as each operand is 32 bits, and the natural data size of the PIC18 is 8 bits. The message to be gleaned from the numbers in Table 7.8 is that you should be cognizant of the execution time differences between different arithmetic operators of the same data type, and between different data types (such as integer versus floating-point) for the same arithmetic operation.

**TABLE 7.8** PIC18 `long` versus `float` C Performance

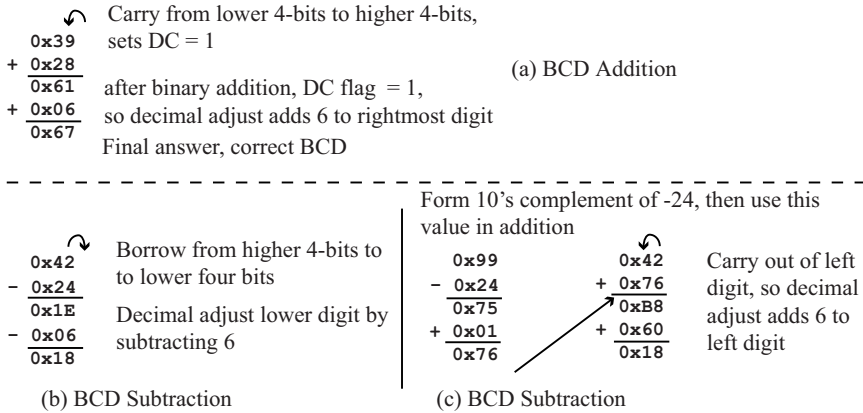
Operation	Instruction Cycles per Loop Iteration		
	Long	Float	Float/Long Ratio
Addition	55	279	5.1
Multiplication	134	494	3.7
Division	605	768	1.3

## 7.6 BCD ARITHMETIC

*Binary Coded Decimal* (BCD) encodes each digit of a decimal number as its 4-bit binary value. This means that the decimal value 83 is simply 0x83 as an 8-bit BCD number. Thus, an 8-bit BCD number can represent the number range 0 to 99. This is a less efficient coding scheme than binary representation, as the codes 0x9A to 0xFF are unused. Some rotary encoders that track the movement of a rotary shaft as it turns either clockwise or counter-clockwise use BCD outputs. Some rotary encoders output incremental codes that only describe the direction of shaft movement; others output absolute position information. If absolute position information is given in BCD, BCD subtraction must be performed to compute the distance between two absolute positions. Addition must be done to compute a finishing location given a starting location and a distance to travel. Adding the two numbers using binary arithmetic, and then post-correcting the sum to obtain the BCD value performs BCD addition.

A BCD digit is a 4-bit value. When two BCD digits are added using binary addition, if the result digit is greater than 9 or if a carry is produced, the result digit must be *decimal adjusted* by adding 6 to produce the correct BCD digit. Similarly, when two BCD digits are subtracted using binary subtraction, if the result digit is greater than 9 or if a borrow is produced, the result digit must be decimal adjusted by subtracting 6 to produce the correct digit. The DC (Decimal Carry) flag in the STATUS register is used for BCD post-correction after addition operations; the DC flag is set if there is a carry from bit 3 to bit 4 during a binary addition. The *daw* (Decimal Adjust W) instruction post-corrects the W register contents to the correct BCD value after any addition or increment instruction that affects the C and DC flags. The *daw* instruction adds 6 to the lower 4 bits (lower digit) of the W register if the DC flag is 1 or if the lower digit is greater than 9; the upper digit is corrected by +6 if the C flag is 1 or if the upper digit is greater than 9. Unfortunately, the *daw* instruction cannot be used after a subtraction operation, so BCD subtraction requires more effort. Recall the binary subtraction  $A - B$  is performed as  $A + \sim B + 1$ , where  $\sim B + 1$  is the two's complement of  $B$ . Similarly, the BCD subtraction  $A - B$  can be performed as  $A + (99 - B + 1)$ , where  $99 - B + 1$  is the *ten's complement* of  $B$ . Figure 7.12a shows the 8-bit BCD addition  $0x39 + 0x28$ . After the binary addition,  $DC_{\text{flag}} = 1$  and the result is  $0x61$ . Because  $DC_{\text{flag}} = 1$ , the lower digit must be corrected by adding 6 to reach the correct result of  $0x67$ . Figure 7.12b shows the BCD subtraction  $0x42 - 0x24$  using binary subtraction, which produces the value  $0x1E$  and a borrow from the upper 4 bits. Because there was a borrow from the upper 4 bits, the lower digit must be post-corrected by subtracting 6 to produce the correct result of  $0x18$ . Figure 7.12c shows the BCD subtraction  $0x42 - 0x24$  performed by adding the ten's complement,  $0x42 + (0x99 - 0x24 + 1)$ . The ten's complement of  $0x24$  is computed as  $0x99 - 0x24 + 1 = 0x76$ . Observe that  $0x24 + 0x76 = 0x00$  in

BCD arithmetic, or  $(+n) + (-n) = 0$ . The ten's complement of 0x24 is then added to 0x42, or  $0x42 + 0x76 = 0xB8$ . This sets  $C_{flag} = 1$ , so the upper digit is post-corrected by adding 6, producing the correct result of 0x18. Observe that both Figure 7.12b and Figure 7.12c produce the same result, but the method in Figure 7.12c is used on the PIC18 as this allows use of the `daw` instruction.



**FIGURE 7.12** BCD addition and subtraction.

Figure 7.13a shows assembly code for the BCD 8-bit addition  $j = j + i$ , using the numbers from Figure 7.12a, while Figure 7.13b gives assembly code for the BCD 8-bit subtraction  $j = j - i$  using the ten's complement approach of Figure 7.12c. Extended precision BCD arithmetic is done in the same manner as extended precision binary arithmetic using the `addwfc` instruction.

<p>(a) BCD Addition <math>j = j + i</math></p> <pre> movlw 0x39 movwf i movlw 0x28 movwf j movf i,w addwf j,w ; do add daw ; post correct W movwf j ; save result                 </pre>	<p>(b) BCD Subtraction <math>j = j - i</math></p> <pre> movlw 0x24 movwf i movlw 0x42 movwf j movf i,w sublw 0x99 ; subtract 99 incf WREG,w ; add 1 for 10's complement addwf j,w ; now do addition daw ; post correct W movwf j ; save result                 </pre>
--	---



**FIGURE 7.13** Assembly code for BCD 8-bit addition and subtraction.

**Sample Question: What does  $0x56 + 0x29$  produce as a BCD sum?**

*Answer:* Using binary addition,  $0x56 + 0x29 = 0x7F$ . Post-correcting for BCD produces  $0x7F + 0x06 = 0x85$ . So, as a BCD sum,  $0x56 + 0x29 = 0x85$ .

## 7.7 ASCII DATA CONVERSION

---

A common task in microprocessor programs is to convert numerical data in ASCII format to binary, or vice versa. This functionality is generally provided by formatted IO functions in high-level languages, such as the `printf` (ASCII output) and `scanf` (ASCII input) C library functions. While the amount of ASCII data manipulation required in typical microcontroller applications is limited, the need for ASCII numerical manipulation invariably arises and one should have some familiarity with the problem.

### Binary to ASCII-Hex

Suppose you wanted to see the bytes of a single or double precision floating-point number printed in ASCII format. The code in Figure 7.14 prints the individual bytes of a single precision and double precision floating-point number in ASCII-hex format. The C function `char2hex()` is the key piece of code in this discussion, as it converts a `char` variable into the ASCII-hex representation of that number. For example, the 8-bit value  $0xA3$  is converted to the two ASCII values  $0x41$ ,  $0x33$ , as these are the ASCII codes for the two hex digits “A” and “3”, respectively. The `char2hex()` function is called for each byte of the single-precision and double-precision floating-point numbers `f` and `d`, respectively. The pointer variable `ptr` is used to iterate over the bytes of the floating-point numbers from most significant to least significant, which are assumed to be stored in little-endian order in memory. The `char buf[2]` array is used as temporary storage for the two ASCII characters generated by `char2hex()`. The `char2hex()` function converts the upper 4 bits, then the lower 4 bits of the input variable `c` to its ASCII equivalent. The 4-bit value that represents one hex digit is first compared to 10. If it is greater or equal to 10, it is converted to its ASCII equivalent “A” ( $0x41$ ) through “F” ( $0x46$ ) by adding the value  $0x37$ . If it is less than 10,  $0x30$  is added to the 4-bit value to produce the appropriate ASCII digit “0” ( $0x30$ ) through “9” ( $0x39$ ). In C, an integer can be printed in ASCII-hex format using the `%x` format in `printf()`. An example usage is `printf(“c = %x”, c)`, which prints `c = 3A` if the binary value of `c` is  $0x3A$  (see Appendix D, “Notes on the C Language,” for more information on `printf()`).

```

    main() Code
float f; //single precision
double d; //double precision
char *ptr;
int i;
char buf[2]; //temp space

main(){
    f = 1398.5625;
    ptr = (char *) &f;
    printf("float: %6.2f bytes are: ",
        f);
    // print the four bytes
    for (i=0;i<4;i++){
        char2hex(*(ptr+3-i),buf);
        putchar(buf[0]); // print MS digit
        putchar(buf[1]); // print LS digit
    }
    printf("\n");
    d = -28.75;
    ptr = (char *) &d;
    printf("double: %6.2lf bytes are: ",
        d);
    for (i=0;i<8;i++){
        char2hex(*(ptr+7-i),buf);
        putchar(buf[0]); // print MS digit
        putchar(buf[1]); // print LS digit
    }
    printf("\n");
}

char2hex() Function
void char2hex(
unsigned char c,
unsigned char *s){
    unsigned char tmp;

    tmp = c >> 4;
    // first hex digit
    if (tmp >= 10)
        tmp = tmp + 0x37;
    else tmp = tmp + 0x30;
    *s = tmp;
    s++;
    // second hex digit
    tmp = c & 0x0F;
    if (tmp >= 10)
        tmp = tmp + 0x37;
    else tmp = tmp + 0x30;
    *s = tmp;
}

```



**FIGURE 7.14** C code for ASCII-hex display of floating-point numbers.

An assembly language version of the `char2hex()` function along with a test program is shown in Figure 7.15. The comparison `c >= 10` contained in the C code is performed in the assembly code by placing the 4-bit value to be tested in the lower 4 bits of `W`, and executing `addlw -D'10'`, which effectively performs `w-10`. The `-10` is added back later by either `addlw 0x37+D'10'` or `addlw 0x30+D'10'` to produce the ASCII codes for "A" to "F" or "0" to "9", respectively. This code makes use of the `swaph` instruction, which swaps the lower and upper nibbles of the source operand.

## Binary to ASCII-Decimal

Table 7.9 shows the steps necessary to convert a binary number to its unsigned ASCII-decimal representation. The successive division by 10 produces the digits from least significant digit to most significant digit. The C statement `printf("%d", i)` prints the value of the `i` variable in decimal; the `printf()` C library function implements the algorithm of Table 7.9 when formatting numbers in ASCII-decimal format.

<pre> Test program CBLOCK 0x00 buf:2 ENDC  org 0 goto main  org 0x0100 main movlw low buf movwf s movlw high buf movwf s+1 movlw 0x2F movwf c call char2hex here goto here         </pre>	<pre> char2hex() Implementation CBLOCK 0x60 c,s:2 ENDC  char2hex movff s,FSR0L movff s+1,FSR0H ;; do most significant digit movf c,w ;; move upper 4-bits to lower 4-bits swapf WREG,w andlw 0x0F ;mask off upper bits addlw -D'10' ; w-10 bnc c2hex_1 addlw 0x37+D'10' bra c2hex_2 c2hex_1 addlw 0x30+D'10' c2hex_2 movwf POSTINC0 ;store digit ;; do least significant digit movf c,w andlw 0x0F ;mask off upper bits addlw -D'10' bnc c2hex_3 addlw 0x37+D'10' bra c2hex_4 c2hex_3 addlw 0x30+D'10' c2hex_4 movwf INDF0 ;store digit return         </pre>
---	---

c >= 10 comparison done by W-10, the -10 is added back.



**FIGURE 7.15** Assembly code implementation of the char2hex() function.

**TABLE 7.9** Conversion of a Binary Number to Unsigned ASCII-Decimal

<b>Steps (digits are determined least significant to most significant)</b>
1. If the number is 9 or less, set the quotient to the number and go to step 3; else, divide the number by 10.
2. Add 0x30 to the remainder; this is the ASCII-decimal digit.
3. If the quotient is 9 or less, then this is the last nonzero digit, so add 0x30 to the quotient to get the ASCII value of the last digit, and exit. If the quotient is 10 or greater, set the number equal to the quotient and loop back to 1.

### ASCII-Hex to Binary

The hex2char() C function of Figure 7.16 does the reverse of the char2hex() C function, in that it converts two ASCII characters representing the hex value of an 8-bit number into the binary value of that number. The main() code of Figure 7.16 passes a two-element char buffer containing the ASCII-hex digits to the hex2char()



function, and saves the return value in *c*. The `hex2char()` result is checked using the *C* formatted input function `sscanf()` in the statement `sscanf(buf, "%x", &i)`. The `%x` format causes the string in *buf* to be scanned for an ASCII-hex value, which is converted to binary and returned in the `int` variable *i*. A failure message is printed if the result returned by `hex2char()` does not match the result returned by `sscanf()`. The `hex2char()` function converts the first ASCII-hex digit by comparing the ASCII value to `0x3A`; if greater than `0x3A`, the character must be in the range “A” (`0x41`) to “F” (`0x46`), so the value `0x37` is subtracted to get the binary value. If the character is less than `0x3A`, it must be in the range “0” (`0x30`) to “9” (`0x39`), so the value `0x30` is subtracted from the character to obtain the binary value. The resulting 4-bit value is placed in the upper half of the `char` variable *c*. The second ASCII-hex character is converted in the same way as the first, to produce the lower 4-bit binary value. Finally, the statement `c = c | tmp` combines the upper 4 bits with the lower 4 bits, and *c* is returned as the converted 8-bit binary value.

<pre> main() Code char buf[3]; // temp space unsigned char c; unsigned int i; main(){     buf[0] = '9'; buf[1] = 'A';     //terminate string     //for sscanf     buf[2]= 0x00;     c = hex2char(buf);     //use sscanf to check     sscanf(buf,"%x",&amp;i);     if (i != c)         printf("hex2char failed!\n");     else         printf("hex2char passed!\n"); } </pre>	<pre> hex2char() Function unsigned char hex2char(unsigned char *s){     unsigned char tmp,c;      tmp = *s;     s++;     // convert 1st char to upper 4-bits     if (tmp &gt;= 0x3A) tmp = tmp - 0x37;         else tmp = tmp - 0x30;     // move to upper four bits     c = tmp &lt;&lt; 4;     // convert 2nd char to lower 4-bits     tmp = *s;     if (tmp &gt;= 0x3A) tmp = tmp - 0x37;         else tmp = tmp - 0x30;     //combine lower 4-bits with upper 4-bits     c = c   tmp;     return(c); } </pre>
---	---



**FIGURE 7.16** C code for converting ASCII-hex to binary.

An assembly language version of the `hex2char()` *C* subroutine is seen in Figure 7.17. The address of the two-digit ASCII-hex number to be converted is passed in *s* of the static parameter block of the subroutine, and the 8-bit value is passed back in the *W* register. This is a straightforward conversion of the *C* function.

```

Test program
CBLOCK 0x00
buf:2
ENDC

    org    0
    goto  main

    org 0x0100
main
    movlw "9"
    movwf buf
    movlw "A"
    movwf buf+1
    movlw low buf
    movwf s
    movlw high buf
    movwf s+1
    call hex2char
here
    goto  here

hex2char () Implementation
CBLOCK 0x60
c,tmp,s:2
ENDC

hex2char
    movff s,FSR0L
    movff s+1,FSR0H
    ;; do most significant ASCII-hex char
    movff POSTINC0,tmp
    movf tmp,w
    addlw 0-0x3A ;w-0x3A
    bnc hex2c_1
    movlw 0x37
    bra hex2c_2 tmp >= 0x3A comparison
hex2c_1
    movlw 0x30
hex2c_2
    subwf tmp,f
    swapf tmp ;move lower 4-bits to upper
    movf tmp,w
    andlw 0xF0 ;clear lower 4-bits
    movwf c ;save in variable c
    ;; do least significant ASCII-hex char
    movff INDF0,tmp
    movf tmp,w
    addlw 0-0x3A ;w-0x3A
    bnc hex2c_3
    movlw 0x37
    bra hex2c_4
hex2c_3
    movlw 0x30
hex2c_4
    subwf tmp,w ; get lower 4-bit value
    iorwf c,w ; combine upper 4-bits with lower
    return ; W reg has 8-bit return value

```



**FIGURE 7.17** Assembly code implementation of the `hex2char()` function.

## ASCII-Decimal to Binary

Table 7.9 shows the steps necessary to convert an ASCII-decimal number to its binary value. The conversion proceeds from most significant digit to least significant digit, converting each digit  $d$  to its binary value, and forming the sum  $r = r*10 + d$  where  $r$  is the cumulative result. The number range that can be converted is dependent upon the size of  $r$ . In C, ASCII-decimal to binary conversion can be accomplished with `sscanf()` using the `%d` format. An example is `sscanf(buf, "%d", &i)`, which converts the first ASCII-decimal string found in `buf` to binary, and returns the result in `i`.

**TABLE 7.10** Conversion of an ASCII-Decimal Number to Binary**Steps**

1.  $r = 0$ .
2. For each digit  $d$  in the ASCII-decimal number starting with the most significant digit:
3. Convert  $d$  to its 4-bit binary value, and let  $r = r * 10 + d$ .
4. If  $d$  is the last digit, exit and return  $r$  as the result; else, advance to the next digit and go to 3.

**SUMMARY**

Multiplication operations in the PIC18 are enhanced by the availability of an 8x8 array multiplier, which can be used to implement higher precision multiplications. Hardware support for division does not exist on the PIC18, but can be implemented using the restoring division algorithm, which requires  $n$  loop iterations for a  $2n$ -bit quotient and an  $n$ -bit divisor. Saturating arithmetic is a method for dealing with overflow in addition and subtraction by clipping the result to either the maximum or minimum values of the number range in case of overflow or underflow, respectively. Floating-point representation encodes an exponent field in addition to magnitude and sign information, greatly expanding the number range that can be represented, at the cost of extra complexity in performing floating-point calculations. Binary Coded Decimal encodes each decimal digit as a 4-bit value, providing fast conversion from BCD to decimal, and vice versa. Support for BCD arithmetic is present in the PIC18 via the DC flag and the `daw` instruction. Conversion of ASCII numerical data in hex or decimal formats to binary, and vice versa, is required for input/output operations of ASCII numerical data and is usually implemented in the form of formatted IO subroutines.

**REVIEW PROBLEMS**

1. What is the 16-bit result for  $0x39 * 0xAD$  if these numbers represent unsigned 8-bit integers?
2. What is the 16-bit result for  $0x39 * 0xAD$  if these numbers represent signed integers using two's complement representation?
3. What data type in C is needed for a multiplication result of a char variable (8-bit) times an int variable (16-bit) if overflow is to be avoided?

4. Extend the 16x16 unsigned multiply of Listing 7.2 to a 16x16 signed multiply.
5. What is the value 0x93AD divided by 0xC5 if these numbers represent unsigned 8-bit integers? Give both quotient and remainder.
6. What is the value 0x93A9 divided by 0x3B if these numbers represent signed integers using two's complement representation? Give both quotient and remainder.
7. Divide by zero is an illegal operation; what does the code of Listing 7.4 do if the divisor is zero?
8. Extend the code of Listing 7.4 to perform signed division.
9. What is the value 0xC4 as a 0.8 fixed-point number?
10. What is the value 0xC4 as a 4.4 fixed-point number?
11. Saturating signed addition is defined as saturating to either the maximum positive value or maximum negative value in the case of overflow using two's complement encoding. What is the result for  $0x39 + 0x59$  using saturating signed addition?
12. Write a PIC18 instruction sequence that implements signed saturating addition as defined by the previous problem.
13. What is the value  $-0.15625$  in single precision floating-point format?
14. The value 0x42F18000 is a single-precision floating-point number; what is its decimal value?
15. Write the steps of an algorithm that compares two single-precision floating-point numbers. Assume both numbers are normalized before the compare is done, and that you do not have to handle special numbers. Hint: Think about comparing the numbers by comparing the individual sign, exponent, and significand fields.
16. What flag would you check to detect overflow in BCD addition of two 8-bit numbers?
17. What is the ten's complement of the BCD value 0x58?
18. Rewrite the `char2hex()` subroutine of Figure 7.21 such that a subroutine is called by `char2hex()` to convert each character to its appropriate 4-bit value.
19. Assuming the divide subroutine of Listing 7.4, what is the largest binary number that can be converted to decimal using the algorithm of Table 7.9?
20. Implement the algorithm of Table 7.10 in PIC18 assembly language for any ASCII-decimal string up to three digits (0 to 999).

*This page intentionally left blank*

# 8

## The PIC18Fxx2: System Startup and Parallel Port IO

### In This Chapter

- High-Level Languages versus Assembly Language
- C Compilation for the PIC18F242
- PIC18F242 Startup Schematic
- *ledflash.c* – The First C Program for PIC18F242 Startup
- Datasheet Reading – A Critical Skill
- PIC18Fxx2 Reset Sources
- Experimenting with RESET, SLEEP, and the Watchdog Timer
- Parallel Port Operation
- LED/Switch IO and State Machine Programming
- Interfacing to an LCD Module

This chapter introduces the hardware side of the PIC18Fxx2 by exploring reset behavior and parallel port IO. In addition, the nuances of writing C code for PIC18Fxx2 applications are examined.

### 8.1 LEARNING OBJECTIVES

---

After reading this chapter, you will be able to:

- Implement a simple PIC18F242 system that has an oscillator, power supply, and reset switch.
- Write C code for the PIC18 that implements IO via pushbutton switches and LEDs using a finite state machine approach.

- Discuss the different features of the PIC18 parallel ports such as bidirectional capability, weak pullups, and open-drain outputs.
- Describe the factors that affect dynamic power consumption in CMOS circuits.
- Identify common features of integrated circuit datasheets.
- Discuss the use of sleep mode in the PIC18 and its effect on power consumption.
- Describe the operation of the watchdog timer and its interaction with sleep mode.
- Implement a parallel interface between a PIC18 and a liquid crystal display module.

## **8.2 HIGH-LEVEL LANGUAGES VERSUS ASSEMBLY LANGUAGE**

---

Previous chapters explored the instruction set of the PIC18 and assembly language programming in the context of *C* programming. This was done so that the linkage from high-level language constructs such as data types, conditional statements, loop structures, subroutines, signed/unsigned arithmetic, and so forth to assembly language is clear. This understanding is needed, as most programming of microprocessors and microcontrollers is done in a high-level language such as *C*, not assembly language, and therefore one must be cognizant of the performance and memory usage repercussions when using features of a high-level language. For example, at this point you would not use floating-point data types for convenience (or out of ignorance), but rather, would carefully weigh whether the computations required by your application actually need the large number range available with floating-point representation. You now know that using floating-point data types requires more memory space for variables, more program memory for calculations, and more execution time for application code. The same tradeoffs apply when weighing the choice between `long` and `char` data types, but not on as dramatic a scale as floating-point versus integer types.

Why is most programming of microprocessors and microcontrollers done in a high-level language and not assembly language? One reason is programmer productivity, which is usually measured in the number of debugged code lines produced per day by a programmer. At this point, you know that it generally takes more assembly language statements than *C* statements to implement the same task, because *C* statements imply data movement or arithmetic operations that require multiple register transfer operations when mapped to a specific microprocessor architecture. Writing more statements takes more time; hence it generally takes longer to write applications in assembly language than in a high-level language.

Another reason is code clarity; code in a higher level language is typically easier to read and understand than assembly language because the fine-grain details of operator implementation are hidden. Another reason is portability; code written in a high-level language is easier to port to another microprocessor than assembly language because it is the compiler's task to translate the *C* to the target microprocessor instruction set. This is important, as code is often reused from application to application, and you do not want to lose the time and money invested in creating an application suite if the target microprocessor changes.

So, when is assembly language needed? One reason to write in assembly language is to implement special arithmetic functions that are not available in the high-level language of choice, such as implementing saturating integer arithmetic in *C*. Another reason is to write a performance-critical section of code in assembly language if the compiler cannot be trusted to produce code that meets required performance specifications. Yet another reason might be to use certain features of the processor that can only be accessed by special instructions within the instruction set. All of these reasons require an understanding of assembly language programming. Even when writing in a high-level language, one should be aware of the features of the instruction set and architecture of the target processor. For example, if the target processor is a 32-bit processor, using 32-bit data types versus 8-bit data types will probably not have much impact on the execution speed of integer operations.

The term *embedded system* is often applied to microcontroller applications because the microcontroller is hidden within the target system, with no visible external interface. A car typically has 10s of microcontrollers within it, yet this fact is not apparent to the car owner. What high-level languages are used to program embedded systems? The *C++* language is a popular choice for complex applications written for high-performance microprocessors. However, the *C* programming language is often the language of choice for an embedded system, as there is a fairly tight coupling between *C* statements and assembly language statements. In addition, most embedded system programs are control-intensive, and do not require complex data structures. Thus, the powerful data abstraction capabilities of an object-oriented programming language such as *C++* are often not required in embedded system applications. If a compiler is available for a microcontroller, in general, it will be a *C* compiler and not a *C++* compiler. This does not mean that there are no microcontroller applications programmed in *C++*, but rather, that *C* is the more popular choice, especially for lower performance microcontrollers.

This chapter begins the hardware topic coverage in this book. Over the next seven chapters, the details of the major hardware subsystems of the PIC18 are explored, and sample applications discussed. To exercise the features of these hardware subsystems, application programs that transfer data between the subsystems and memory, configure subsystems for different operating modes, and check



subsystem operation status are presented. These programs are written in C to promote code clarity. It is a difficult enough task to grasp the operational details of a hardware subsystem without the additional problem of struggling with long assembly language programs, where the details of memory transfers and arithmetic operator implementation mask the overall program functionality. The previous coverage of the PIC18 instruction set and assembly language programming techniques in the context of the C language has prepared you for moving beyond assembly language when discussing the hardware subsystems of the PIC18. In covering the PIC18 hardware subsystem details, it is expected that the reader will frequently reference the PIC18Fxx2 data sheet [6], available from the Microchip WWW site at [www.microchip.com](http://www.microchip.com). This book does not attempt to duplicate all of the information in the PIC18FXX2 data sheet, which is clearly an unnecessary task. Instead, this book presents key functionality of each subsystem in the context of application examples. In some cases, detailed descriptions of the registers associated with a subsystem and individual register bits are presented in this book; at other times, the reader is referred to the datasheet. The ability to read datasheets and extract key information is a necessary survival skill for any person interfacing microprocessors or microcontrollers to other devices. A section within this chapter is devoted to providing tips on datasheet reading for those readers who are encountering datasheets for the first time.

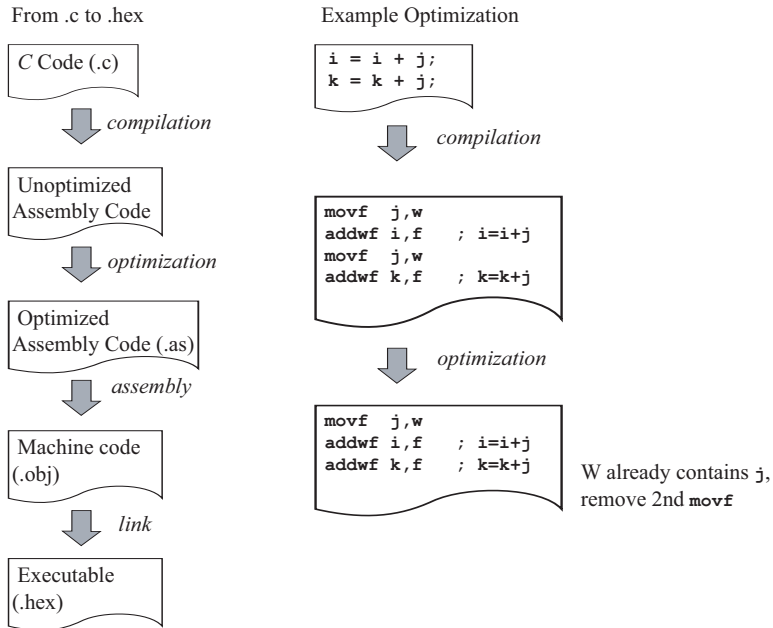
### 8.3 C COMPILATION FOR THE PIC18F242

---



In this book, the C programs for hardware application examples use the PICC-18 C compiler from HI-TECH Software. A demo version of this compiler is on the CD-ROM included with this book. Details on using the compiler within MPLAB are given in Appendix C, “HI-TECH PICC-18 C Compiler Demo for the PIC18F242.” In Chapter 2, “The Stored Program Machine,” a compiler is defined as a program that translates statements in a high-level language to assembly language. Figure 8.1 shows the steps of transforming a C program into machine code that can be programmed into the PIC18. The compiler first transforms the C code into *unoptimized* assembly language, which is done by looking at each C statement individually and implementing it as a sequence of instructions. The *optimization* stage then looks at groups of assembly language instructions, and attempts to reduce the number of instructions by considering data transfer requirements of the entire group. The right-hand side of Figure 8.1 shows an example compiler optimization, in which two C statements are translated into four assembly language instructions when each C statement is considered individually. The optimizer then considers the four assembly language instructions as a group, and notes that the W register already contains the value of j from the previous operation, and therefore

the second `movf j,w` instruction can be removed. This is only a small example, as there are many different types of compiler optimizations that are performed. Typically, code size is reduced and performance improved after optimization.



**FIGURE 8.1** The compilation process.

After optimization, an assembler internal to the compiler translates the assembly language to machine code that is then placed within an object code (*.obj*) file. If a program becomes large, it is regarded as good programming practice to split the source code among several files, where each file contains functions that are related to each other. This makes the source code easier to maintain, and allows a group of programmers to work on the same microcontroller application concurrently. Thus, an application may have several object files, and it is the job of the *linker* to combine these files into a single file that is executed by the microprocessor. In the case of the PIC18, this “executable” file is a *.hex* file, which contains an ASCII-hex representation of the program memory contents and is downloaded into the program memory of the PIC18 (see Appendix F, “The Jolt/Colt Serial Bootloaders,” for details on how this is accomplished).

## Special Function Registers and bit Variables

A C compiler for a microcontroller must provide access to the special function registers and individual bits of those registers. In the PICC-18 compiler, all special function registers have C `define` statements for them contained in a header file (*.h*) that is particular to the target device. For example, a special function register used for parallel port IO and discussed later in this chapter is named `PORTB`, whose location in the file registers is `0xF81`. In the PICC-18 installation directory, a file named `PIC18/include/pic18xx2.h` is the header file used for PIC18Fxx2 devices. Within this file is contained the following line that defines the special function register `0xF81` as the unsigned char variable `PORTB`.

```
static volatile near unsigned char   PORTB           @ 0xF81;
```

This means that `PORTB` can be used in the same manner as any C variable name; for example, the `PORTB` register contents can be assigned a value using a C assignment statement, such as `PORTB = 0xF0`.

In PIC18 assembly language, individual bits of file registers are set and cleared using the `bsf` and `bcf` instructions, respectively. In addition, the instructions `bitfsc` (bit test, skip if clear) and `bitfss` (bit test, skip if set) are used for conditional execution based on the status of an individual bit. One way to accomplish the same functionality in C is to use *macros* as seen in Listing 8.1, which define `bitset`, `bitclr`, and `bittst` macros using C bitwise logical operations. Listing 8.1 also contains examples uses of these macros. The code for a macro is replicated inline wherever it is used; the statement `bitset(PORTB,2)` may look like a function call but is replaced by the code `PORTB |= (1 << (2))` during compilation. The `|=` operator is a shorthand notation for `PORTB = PORTB | (1 << (2))`.

**LISTING 8.1** C Macros for `bitset/bitclr/bittst` and example uses.

```
#define bitset(var,bitno) ((var) |= (1 << (bitno)))
#define bitclr(var,bitno) ((var) &= ~(1 << (bitno)))
#define bittst(var,bitno) (var & (1 << (bitno)))

// example uses
bitset(PORTB, 2); //PORTB, bit 2 is set to 1
bitclr(PORTB, 7); //PORTB, bit 7 is cleared to 0
if (bittst(PORTB,6)) {
    //execute if_body if PORTB, bit 6 is 1
}
if (!bittst(PORTB,5)) {
    //execute if_body if PORTB, bit 5 is 0
}
```

In the PICC-18 compiler, the `bitset` and `bitclr` C macros compile directly to the corresponding `bsf` and `bcf` instructions, making these macros very efficient. Similarly, the `bitst` macro also compiles to either the `btfss` or `btfsc` instruction, depending on context. However, because bit set/clear/test operations are so prevalent in microcontroller applications, the PICC-18 compiler defines a `bit` data type that supports naming of individual register bits as separate variables, allowing them to be used any place a normal C variable is used. Most individual bits of special function registers are defined using `bit` data types in the `pic18xx2.h` header file. For example, the following line defines bit 5 of PORTB as a `bit` variable named RB5.

```
static volatile near bit RB5 @ ((unsigned)&PORTB*8)+5;
```

Listing 8.2 shows the bit set/clear/test operations of Listing 8.1 implemented using `bit` variables instead of C macros.

---

**LISTING 8.2** Bit set/clear/test using bit variables.

---

```
// example uses of bit variables
RB2 = 1; //PORTB, bit 2 is set to 1
RB7 = 0; //PORTB, bit 7 is cleared to 0
if (RB6) {
    //execute if_body if PORTB, bit 6 is 1
}
if (!RB5) {
    //execute if_body if PORTB, bit 5 is 0
}
```

One advantage to using `bit` variables is improved code clarity, as the statement `CARRY = 1` is clearer in its intent of setting the carry flag than `bitset(STATUS,0)`, as one has to remember that bit 0 of the STATUS register is the Carry flag. Another advantage is that it helps in porting code between PIC microcontroller families, as using `bit` variables can isolate code from changes such as status or control bits changing positions within a register, or being moved to a different register altogether. The header file for the target device contains the register and position information for a named bit, and thus code can be made portable between closely related PIC microcontrollers (not all porting issues can be solved this way, but this helps). The disadvantage to using `bit` variables is that the `bit` data type is not a standard data type in the C language, but is an extension to the language added by HI-TECH Software. This means any code using `bit` variables has to be changed if a different C compiler is used, such as the Microchip MCC18 compiler. Another possible problem is that the use of `bit` variables masks the fact that a bit is the target of an assignment instead of a register. For example, there is nothing inherent in the statements `PORTB = 1` and `RB6 = 1` that clues a reader unfamiliar with PIC18 details that PORTB is an 8-bit register whose contents is assigned the value 0x01,

while RB6 is an individual bit within PORTB. This lack of distinction between registers and bits is confusing only if the reader is unaware of PIC18 hardware details before reading PIC18 C code. In this book, all named bits of special function registers are introduced in proper context before showing their usage in C code. The C code examples in this book use both `bitst/bitset/bitclr` macros and `bit` variables, with one being favored over the other if code clarity is improved within a particular context.

## PICC-18 Runtime Code

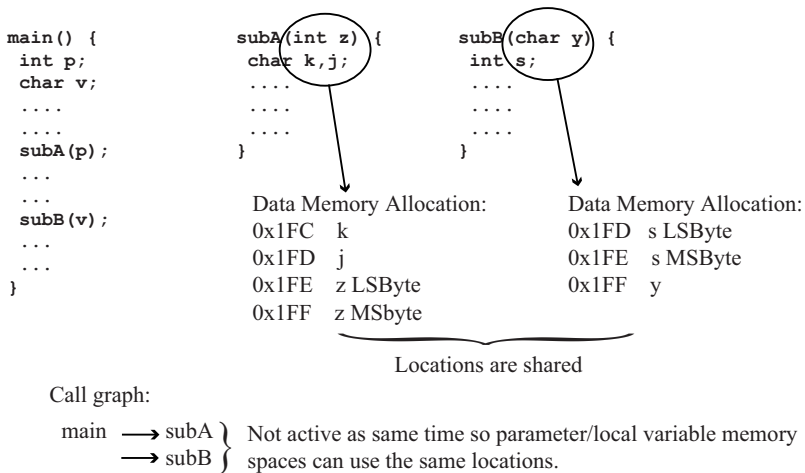
The runtime code produced by the PICC-18 compiler first initializes all global variables (to zero if they are not given an explicit initial value) and then jumps to the entry point of `main()`. Qualifiers can be used with variable declarations to control particulars about where the variable is located in memory, and how it is initialized. Table 8.1 lists the qualifiers used within examples in this book; see the PICC-18 compiler documentation for a complete description of all qualifiers. A variable declared outside of a function is a global variable; local variables within a function are called auto variables. An auto variable is only given an initial value if one is explicitly stated, unlike global variables that have a default initial value of zero. If an auto variable is given an initial value, the variable is initialized each time the function is called. Auto variables are not guaranteed to retain their values between function calls. A `static` qualifier applied to an auto variable means the auto variable is only initialized once, and causes it to retain its value between function calls. The `static` qualifier also causes the variable to have a nonshared memory address (explained later in this section), the same as a global variable. The `persistent` qualifier can be used on a global variable to prevent it from being initialized by the startup code that is executed before `main()` is called. This is useful for tracking events between processor resets, and its use is covered in more detail later within this chapter. The `volatile` qualifier tells the compiler that this variable may change its value between successive accesses in the code; that is, some other agent such as an external pin change or interrupt (see Chapter 10, “Interrupts and a First Look at Timers”) can change the variable’s value. This prevents certain compiler optimizations from being applied to the variable.

In the PICC-18 compiler, static allocation is used for function parameters and auto variables, so function recursion is not supported. During the compilation process, the PICC-18 compiler builds a *call graph* to determine the functions that are active at any given time. Those functions not active at the same time share the same data memory space for parameters and local (auto) variables as shown in Figure 8.2. This can be done because local variables in C are not guaranteed to retain their values between function calls. This provides efficient utilization of the available data memory space. If a local variable is declared `static`, its memory space is

not shared, as it must retain its value between function calls. The call graph also allows the compiler to determine if recursive calls are being made; if a recursive call is found, an error message is printed. The *map file* produced by the compiler gives the call graph and the locations of all functions and global variables (see Appendix C for more details on PICC-18 compiler usage).

**TABLE 8.1** Variable Qualifiers

Qualifier	Example	Comment
static	static char i =1;	Initialize only once, retains value between function calls.
persistent	persistent char j;	For global variables, do not touch with reset code.
volatile	volatile char k;	Can be changed between accesses.

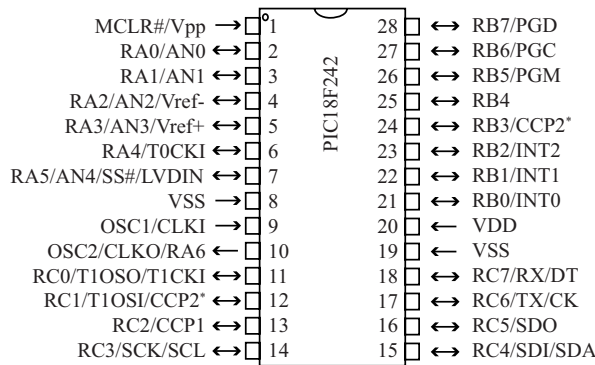


**FIGURE 8.2** PICC-18 compiler static allocation.

By default, variables are allocated anywhere there is free memory in the file registers. The storage for active auto variables is limited to one bank, or 256 bytes. Thus, a variable declaration such as `char buf[300]` is not allowed within a function (an auto variable), but could be declared as `static char buf[300]` within the function or declared as a global variable external to the function.

### 8.4 PIC18F242 STARTUP SCHEMATIC

The version of the PIC18xx2 used in the interfacing examples in this book is the PIC18F242 and its pin diagram is seen in Figure 8.3. This device is imminently suitable for experimentation on a protoboard, given its small footprint of 28 pins and the dual-inline package (DIP). An external pin usually has more than one internal function mapped to it because of the limited number of pins on the 28-pin package. Control bits within PIC18 special function registers determine the mapping of internal functions to external pins.



\*RB3 is the alternate pin for the CCP2 pin multiplexing  
 Figure redrawn by author from PIC18Fxx2 datasheet (DS39564B), Microchip Technology Inc.

**FIGURE 8.3** PIC18F242 pin diagram.

A brief summary of the pin functions used in this book is given here; more details are provided in the appropriate chapter covering that functionality. The arrows next to the pin in Figure 8.3 indicate the pin direction; an arrow pointing into the device indicates an input-only pin, an arrow pointing out of the device indicates an output-only pin, and arrows on both ends indicates a bidirectional pin (a pin that can function as either an input or output).

**Vdd, Vss:** These are power (Vdd) and ground (Vss) pins; observe that there is more than one ground pin. It is not unusual for an integrated circuit to have multiple power and ground pins, all of which must be connected.

**MCLR#:** This input pin resets the device when brought low. The # symbol in the name indicates a low true signal.

**Vpp, PGD, PGC, PGM:** These pins are used to download a program into the device via an external programmer (see Appendix F).

**OSC1, OSC2:** These pins are used to provide the main clock source for the device (details in this chapter).

**RAn, RBn, RCn:** These bidirectional pins are parallel port IO pins (details in this chapter). In this book, we principally use the RBn pins (PORTB) for parallel IO, as these pins share the least amount of functionality with other internal subsystems.

**ANn, Vref-, Vref+:** The ANn inputs are the analog inputs for the analog-to-digital converter subsystem (Chapter 12, “Data Conversion”). The Vref-/Vref+ pins are used to provide negative and positive voltage references for the analog-to-digital converter.

**TX, RX:** These pins are used for asynchronous serial transmit (TX) and receive (RX) (see Chapter 9, “Asynchronous Serial IO”).

**SCL, SDA:** These pins implement the I<sup>2</sup>C synchronous serial data interface (see Chapter 11, “Synchronous Serial IO”).

**SCK, SDI, SDO, SS#:** These pins implement the serial peripheral interface (SPI), which is a synchronous serial data transfer protocol (see Chapter 11).

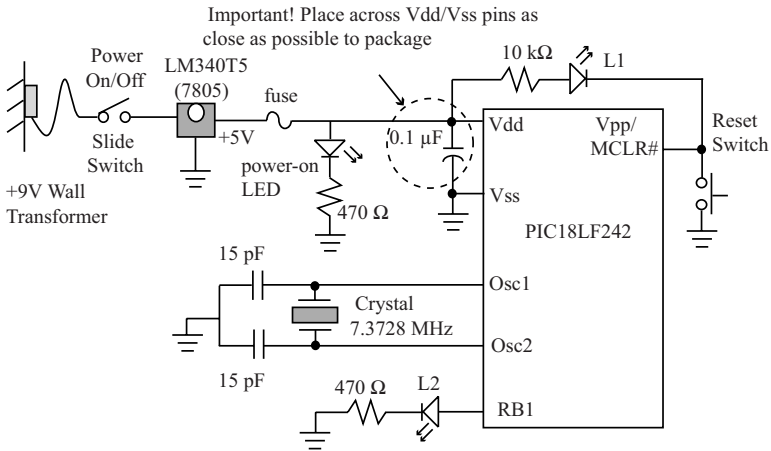
**CCP1, CCP2:** These pins are used by the capture and compare module, which is useful for measuring time between external events and also square wave generation (see Chapter 13, “Timers”).

**INTn:** These pins provide external interrupt sources to the PIC18 (see Chapter 10).

A schematic used to test PIC18F242 functionality by flashing a light emitting diode (LED) is seen in Figure 8.4. A PIC18F242 requires that Vdd be in the range 4.2 V to 5.5 V, while a PIC18LF242 supports a wider Vdd range of 2.0 V to 5.5 V. The “L” in the PIC18LF242 part number stands for low power; reducing Vdd reduces power consumption. Further details on power consumption of CMOS circuits are provided later in this chapter. An external AC-to-DC wall transformer provides power that outputs 9 V *unregulated*, which means that the output voltage can fluctuate depending upon the amount of current that is drawn from the transformer. A LM340T5 *voltage regulator* is used to provide a stable 5 V output as the Vdd for this system (Appendix E, “Suggested Laboratory Exercises,” contains a complete list of the components used in the interfacing examples of this book as well as a picture of a completed protoboard layout). The fuse in the power path is a safety precaution; it will *blow* (no longer conduct current) if excessive current is drawn from the power supply. The total current required for our PIC18 system after all peripheral chips have been added is under 40 mA (1 mA = milliAmperes



= 0.001 A), so a 200 mA fuse provides more than enough margin. Excessive current can be drawn if a *short* (a low resistance path) is created between Vdd and Vss by a wiring mistake during circuit hookup, or by component failure.



**FIGURE 8.4** PIC18F242 schematic for flashing an LED.

The 0.1 μf capacitor connected from Vdd to Vss is called a *decoupling* capacitor; it assists in supplying transient current needs caused by high-frequency digital switching. This capacitor should be placed as close as possible to the Vdd and Vss pins of the device for maximum effectiveness; having these two pins adjacent to each other on the package allows this capacitor to be placed directly across Vdd and Vss. This capacitor is important; the author has seen several examples in lab where a PIC18 system has acted erratically when a student has neglected to include the decoupling capacitor. Appendix G, “Circuits 001,” provides a brief introduction to elementary circuit concepts if you are unfamiliar with basic circuit elements such as resistors, capacitors, diodes, and so forth. You only need a hobbyist-level intuition about basic circuit concepts for the interfacing topics in this book; detailed circuit analysis background is not required.

When power is applied to the PIC18F242, the device does a *self-reset*, which means that the program counter is cleared to 0 and the first instruction is fetched from location 0. This is known as a *power-on reset* (POR). However, it is also convenient to have a manual reset capability during testing to restart a program without needing to cycle power. The momentary pushbutton connected to the MCLR# input applies a low voltage to MCLR# when pushed, causing the PIC18 to reset. The 10 kΩ resistor that connects the MCLR# pin to Vdd is called a *pullup* resistor, as it keeps the MCLR# input pulled up to near 5 V when the pushbutton is released. If

the pullup resistor is removed, and a direct connection is made from MCLR# to Vdd, a short is created when the pushbutton is pressed, causing excessive current flow and probably causing the fuse to blow. If the pullup resistor and Vdd connection is not made at all, the input floats between 0 V and Vdd when the pushbutton is not pressed. A floating input can read as either “0” or “1” depending on the switching activity of nearby pins, causing spurious circuit operation. A PIC18 with a floating MCLR# input can experience intermittent resets, a problem that is difficult to debug. The L1 LED and the 10K resistor connected from Vdd to MCLR# causes the LED to light dimly when the reset button is pressed, giving visual feedback that reset is being applied. More importantly, if the PIC program memory is modified by an external programmer (Appendix F), a high voltage (12 V, the programming voltage Vpp) is applied to the Vpp/MCLR# pin. The LED between MCLR# and Vdd prevents this high voltage from appearing on the Vdd bus, averting damage to other devices sharing the Vdd bus.

The crystal connected between the OSC1 and OSC2 pins provides an accurate clock source for the device. The 15 pF capacitors along with an internal amplifier circuit in the PIC18 causes the crystal to begin oscillation shortly after power is applied (startup time is typically less than 50 ms and can vary, see [7] for a complete discussion of PIC oscillator characteristics). The waveform produced by the crystal oscillator is a sinusoidal signal; this is converted to a square-wave clock signal within the PIC18. In the examples in this book, an internal PIC18 option called HSPLL (high speed crystal/resonator with Phase Locked Loop) is used that multiplies the crystal frequency by four to produce the internal clock frequency FOSC. Thus, the 7.3728 MHz crystal frequency results in an internal clock frequency of 29.4912 MHz. You will discover later that this “strange” frequency is useful for asynchronous serial communication. Two other methods of clock generation for the PIC18 are:

- An external resistor/capacitor can be connected to the OSC1 input; this is an inexpensive method of clock generation but the highest clock frequency is limited, and clock frequency accuracy is sacrificed.
- An external clock can be input directly into the OSC1 pin. Many varieties of external oscillator devices are available that provide an accurate, high frequency clock waveform but the cost is typically twice that of a crystal/capacitor network.

Details on oscillator options for the PIC18 are found in the PIC18Fxx2 datasheet [6].

## 8.5 LEDFLASH.C—THE FIRST C PROGRAM FOR PIC18F242 STARTUP

The C code of Figure 8.5 flashes the L2 LED connected to pin RB1 in Figure 8.4. The statement `#include <pic18.h>` includes the generic PIC18 header file; additional `#include` statements within the `pic18.h` file cause a device-specific header file, such as the `pic18xx2.h`, to be included based upon the target device specified within MPLAB and passed to the PICC-18 compiler.

```

#include <pic18.h> ← Standard PIC18 header file
#include "config.h" →
Contains configuration bits and bit macros.
See the file on the CDROM at
/code/common/config.h
- - - - -
CONFIG(1,HSPL);
CONFIG(2, BORDIS & PWRTDIS & WDTDIS);
CONFIG(4, DEBUGDIS & LVPDIS);
#define bitset(var,bitno) ((var) |= (1 << (bitno)))
#define bitclr(var,bitno) ((var) &= ~(1 << (bitno)))
#define bittst(var,bitno) (var & (1 << (bitno)))
- - - - -

void a_delay(void)
{
    unsigned int i,k;
    // change count values to alter delay
    for (k=1800; --k;) {
        for(i = 200 ; --i );
    }
}

// just flash LED on port RB1
// use this to test if PIC board is alive
main(void){
    TRISB1 = 0;    // configure RB1 as output
    RB1 = 0;      // set RB1 low initially

    while(1) {
        a_delay(); // call delay subroutine
        RB1 = 1;   // turn on RB1 (LED)
        a_delay();
        RB1 = 0;   // turn off RB1 (LED)
    }
}

```

Subroutine for software delay

Infinite loop that blinks LED.  
Only exit is through MCLR# reset or power cycle.



**FIGURE 8.5** C code for flashing an LED.

Figure 8.5 is the first C code listing for PIC18 hardware experiments presented in this book. Because of space considerations, the C source code given in figures is typically not complete—the figure source code will usually omit C functions previously covered or omit `include` statements for various header files. Most of the C source code used in book figures that illustrate PIC18 hardware features is included on the companion CD-ROM in their complete form; please use these source files when attempting to duplicate the experiments.

## Configuration Bits

The `CONFIG` lines are C macros that define PIC18 *configuration* bit settings for the compiler. Configuration bits select different operating modes for the PIC18 and are stored in configuration registers in program memory beginning at location 0x300001. Each `CONFIG` statement specifies bit settings in a different configuration register; for example, `__CONFIG(4, DEBUGDIS & LVPDIS)` specifies the configuration bits for configuration register 4. After compilation, configuration bits and the program machine code are found in the *.hex* file produced by the compiler. The `CONFIG` statements in Figure 8.5 select the following modes, which are used in the hardware examples in this book.

**HSPLL:** This controls oscillator selection. The HSPLL option creates the internal clock via an external crystal whose frequency is multiplied by 4.

**BORDIS:** This option disables brown-out reset. A *brown-out* refers to a failure in operation due to V<sub>dd</sub> droop, a common problem in battery-operated systems. The PIC18 has a brown-out detection circuit that generates a device reset when the V<sub>dd</sub> value drops below one of four selectable trigger levels.

**PWRTDIS:** This option disables the power-up timer. A *timer* on a microcontroller is simply a counter that is clocked at a particular frequency, and generates an action when a particular count is reached. The amount of time to trigger the event depends on the clock frequency of the counter, and the size of the counter. The power-up timer on the PIC18 implements a fixed delay (typical value is 72 ms) after power-up. Its intended use is to provide time for the power supply to stabilize before fetching of the first instruction.

**WDTDIS:** This option allows the Watchdog Timer (WDT) to be disabled in software via the SWDTEN bit (bit 0) of the WDT configuration register (WDTCON). If this option is not specified, the watchdog timer is always enabled and cannot be disabled in software. Additional information on the watchdog timer and its usage is included later in this chapter.

**DEBUGDIS:** This option disables the PIC18 in-circuit debug (ICD) capability, which provides simple debugging functions such as breakpoints and limited register examination while the microprocessor is *in-circuit* (i.e., on the proto-board or circuit board). When enabled, pins RB6 and RB7 are used to transfer debugging information in a serial manner to an external programmer that supports the ICD functionality. Furthermore, the in-circuit debugger uses some return address stack space (2 locations), data memory (10 bytes), and program memory (512 bytes) when enabled. While the examples in this book do not explicitly use the PIC18 ICD capability, you can experiment with ICD if you have the required external programmer support (see Appendix F for more details).

**LVPDIS:** This option disables the low-voltage programming mode of the PIC18. When enabled, the program memory is programmed using the normal V<sub>dd</sub> supply instead of using a higher programming voltage applied to the V<sub>pp</sub> pin. Pin RB5 is lost for parallel IO usage when this option is enabled, and must be held low during normal operation. The PIC18 system used in this book has an asynchronous serial port interface, allowing the PIC18 to program itself from bytes downloaded through the serial port using a program referred to as a *serial bootloader*, which negates the need for low-voltage programming. The asynchronous serial port is discussed in Chapter 9, and the serial bootloader in Appendix F.

Appendix A, “PIC18Fxx2 Architecture, Instruction Set, Register Summary,” summarizes the configuration registers and their bit definitions; a complete list of the configuration bits and their functions is found in the PIC18Fxx2 datasheet.

## Flashing the LED

Pin RB1 is a bidirectional, parallel port pin and can either be configured as an input or as an output. Pin RB1 is just 1 bit of the 8-bit parallel port called PORTB. The RB1 pin must be configured as an output to drive the LED, which is accomplished by the statement `TRISB1 = 0`. Each bit of the special function register TRISB controls the direction of the corresponding PORTB bit. A “1” in a TRISB bit configures the corresponding PORTB pin as an input, while a “0” configures the PORTB pin as an output (additional details on parallel port IO are given later in this chapter). The statement `RB1 = 1` assigns a zero to the RB1 data latch, thus driving the RB1 pin low and turning off the LED. The statement `while(1){}` creates an infinite loop, whose loop body alternately turns the LED on (`RB1 = 1`) and off (`RB1 = 0`). A time delay is created between each RB1 assignment by the function call `a_delay()`, which is composed of two nested `for{}` counting loops. This type of time delay is called a *software delay loop*, and the delay time is dependent upon the number of instructions in the loop and the clock frequency of the PIC18. The delay can be increased or decreased by changing the count values in the nested `for{}` loops of the `a_delay()` function. The delay must be long enough so that the LED can fully turn off or turn on between RB1 pin assignments. If the delay is too short, the LED will appear always on, even though an oscilloscope trace would reveal that RB1 is transitioning between low and high voltages (a square wave output). Software delays are easy to implement, but hardware timers are much better at creating accurate time delays. The timer subsystem of the PIC18 and its usage is first discussed in Chapter 10, and covered in more detail in Chapter 13. Observe that the only method of terminating the `while(1){}` loop in Figure 8.5 is by cycling power or reset via the pushbutton on MCLR#. This infinite loop nature is typical of microcontroller applications because if the loop is exited, there is nowhere to go!

Figure 8.6 gives a macro, `DelayUs(x)` (delay for  $x$  microseconds), and a function, `DelayMs(cnt)` (delay for  $cnt$  milliseconds), that provide a better implementation of software delay loops than the `a_delay()` function of Figure 8.6. This code is a variation of the sample delay-loop code provided with the HI-TECH PICC-18 compiler and works well for internal clock frequencies (FOSC) greater than 12 MHz. Under full compiler optimization, the `DelayUs(x)` macro compiles to a loop that contains three instruction cycles (12 clock cycles); hence the loop takes 1  $\mu$ s if FOSC is 12 MHz. The `DelayUs(x)` macro is less accurate for lower values of  $x$  and where FOSC is not evenly divisible by 12, but typically software delay loops are not used where high accuracy is needed. The result of the computation  $(x * FOSC) / 12\text{MHz}$  should not exceed 255, since the `_dcnt` loop variable is an 8-bit value; for FOSC = 40 MHz, this limits  $x$  to a maximum value of 76. The `DelayMs(cnt)` function contains two nested loops; the inner loop is a 1 ms delay that is executed  $cnt$  times by the outer loop to provide a  $cnt$  ms delay. The inner loop's 1 ms delay is performed by calling `DelayUs(50)` 20 times, as  $20 * 50 \mu\text{s} = 1000 \mu\text{s} = 1 \text{ ms}$ . Full compiler optimization should be enabled when using this delay code, else longer delays than expected are generated. Code examples in this book that use software delay loops make use of these functions.

```

#define FOSC    29491L    // Internal clock freq in KHz
#define MHZ     *1000L   // number of KHz in a MHz
#define KHZ     *1       // number of KHz in a KHz

#define DelayUs(x) { unsigned char _dcnt; \
                    _dcnt = ((x* FOSC) / (12MHZ)); \
                    while(--_dcnt != 0) \
                    continue; }

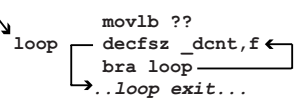
void DelayMs(unsigned char cnt)
{
    unsigned char i;
    do {
        Call DelayUs(50) i = 20;
        20 times for do {
        1 ms delay DelayUs(50);
                    } while(--i);
    } while(--cnt);
}

```

When using these functions, must use full optimization!

Be cautious of overflow in `_dcnt` variable. For 40 MHz,  $x$  max is 76.

Compiles to loop with 12 clock cycles, 1  $\mu$ s per iteration if FOSC = 12 MHz



**FIGURE 8.6** Delay loop code (see CD-ROM file `./code/common/delay.h`).

## 8.6 DATASHEET READING—A CRITICAL SKILL

At this point, the phrase “Topic  $x$  is discussed in more detail in the PIC18Fxx2 datasheet” has been used several times. It is impractical for this book to replicate all PIC18Fxx2 datasheet information, as this datasheet is over 300 pages in length! As

such, you *must* become comfortable with reading the PIC18Fxx2 datasheet, and the datasheets of other devices referenced in this book, to gain full understanding of the interfacing examples. The information detail in a component datasheet may initially seem overwhelming, but this can be countered by knowing how typical datasheets are organized, and where to look for certain types of information. A typical component datasheet is organized as follows:

**Initial summary and pinouts:** The first section contains a device functional summary, which includes pin diagrams and individual pin descriptions.

**Functional description:** Individual sections discuss the functional details of the device operation. In the PIC18Fxx2 datasheet, these sections correspond to the subsystems of the PIC18 such as the timers, the analog-to-digital converter, etc., with each section containing the special function registers used by the subsystem and the individual bit definitions of these SFRs. Step-by-step instructions for subsystem configuration and usage are provided. Each section ends with a summary that lists all of the special function registers used by a particular subsystem, which is very useful for quick reference.

**Electrical characteristics:** Electrical characteristics are divided into DC specifications (operating voltage, power consumption, output port drive capability, etc.) and AC specifications (timing characteristics such as propagation delay, maximum clock frequency, etc.). This section contains tables of values with minimum, typical, and maximum values; the typical values are used in this book whenever timing information is given. For the PIC18Fxx2, graphs such as current requirements versus voltage and frequency are provided in this section. The electrical characteristics section always contains a table labeled as *Absolute Maximum Ratings*, which are the maximum voltage/current values that can be experienced without damaging the device. These are not the typical operating voltage/current ratings of the device. For example, the maximum voltage rating of the V<sub>DD</sub> pin is -0.3 V to +7.5 V. However, the actual operating voltage of the PIC18Fxx2 is 4.2 V to 5.5 V.

For a microprocessor, the component datasheet such as [6] contains information specific to the features of that particular processor, but may only contain summaries if this information is common to many members of a microprocessor family. Expanded descriptions of features common to all microprocessor family members, such as the instruction set or hardware subsystems, are contained in *reference manuals* for that family [8]. *Application notes* such as [7] give detailed usage examples of particular microprocessor features. Datasheets, reference manuals, and application notes assume a general familiarity and previous background with similar components on the part of the reader. Books such as this one are useful for

readers who are new to these devices, or for experienced readers who are looking for a single source that combines and summarizes information from datasheets, application notes, and reference manuals. The ability to read a datasheet is a *critical* skill for any practicing engineer, engineering student, or hobbyist, and skills are obtained only through practice. So please, take the time to peruse the PIC18Fxx2 datasheet and the datasheets of other devices used in the hardware examples when working through the remaining chapters.

## 8.7 PIC18FXX2 RESET SOURCES

Methods of resetting the PIC18Fxx2 discussed to this point have been power-on, MCLR#, brown-out, stack underflow, and stack overflow. Figure 8.7 shows a simplified version of the reset circuitry for the PIC18Fxx2. The RESET instruction provides software reset capability, which is useful if some catastrophic error is detected and a clean start is desired. The power-up timer (PWRT) and oscillator start-up timer (OST) delay the release of reset after power is applied. The OST provides a 1024 cycle counter delay after the clock is applied, providing extra time for the oscillator to stabilize, while the PWRT provides a fixed delay (typically 72 ms) for power stabilization. Observe that the PWRT has a separate on-chip clock source and that its delay is placed in series with the OST if the power-up timer is enabled.

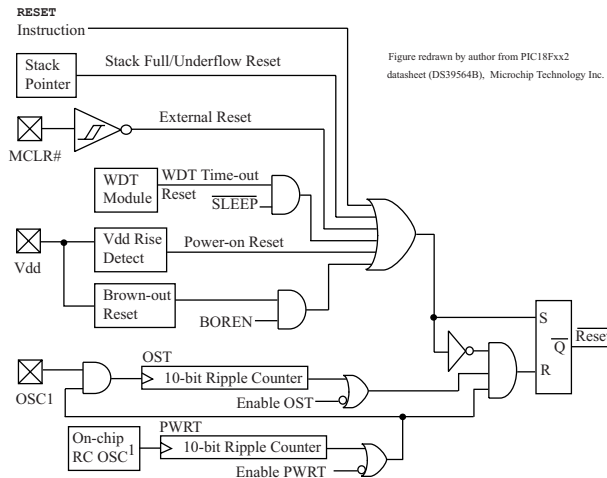


Figure redrawn by author from PIC18Fxx2 datasheet (DS39564B), Microchip Technology Inc.

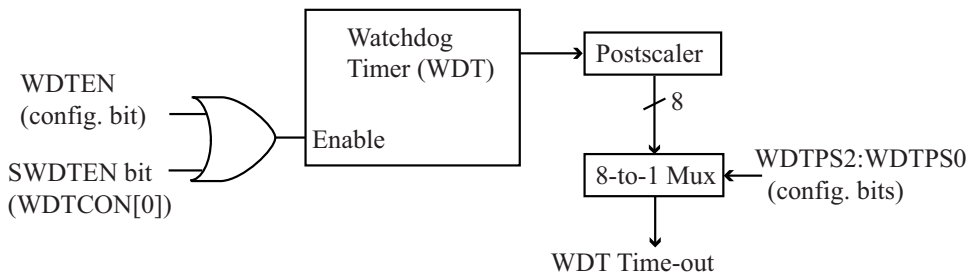
Note 1: This is a separate oscillator from the RC oscillator of the CLKI pin.

**FIGURE 8.7** Reset sources for the PIC18Fxx2.<sup>1</sup>

<sup>1</sup> Figure 8.7 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.'s prior written consent.



The watchdog timer (WDT) also has a separate on-chip clock source and can act as a reset source if enabled. Figure 8.8 shows a block diagram of the WDT. A *postscaler* on a timer is used to lengthen the timeout period of the timer. The WDT has a postscaler controlled by 3 bits WDTPS2:0 stored in the configuration registers of program memory. These bits select postscale values of 1:1 (bits = 000), 1:2 (bits = 001), 1:4 (bits = 010), and so on, up to a maximum of 1:128 (bits = 111). A typical WDT timeout value without postscaling is 18 ms. Using a maximum postscale value of 1:128 (WDTPS2:0 = 111) increases this timeout value to  $18 \text{ ms} * 128 = 2.3 \text{ s}$ .



**FIGURE 8.8** Watchdog timer block diagram.

One use of the watchdog timer is to place a maximum wait time for some external event to occur. For example, assume the PIC18 has sent a request for information to an external device, and is now waiting for a response. If the external device fails, or if the request is corrupted such that the external device never received the request, the processor is stuck in an infinite loop, waiting for a response that it will never receive. The WDT timer can act as an alarm clock, forcing a device reset upon expiration, allowing recovery from this infinite wait scenario. To prevent WDT timeout during normal operation, the PIC18 instruction `CLRWDT` (clear watchdog timer, has no arguments) must be executed periodically to reset the WDT before the WDT expires.

Another use of the WDT timer is to wake the PIC18 from *sleep* mode, a low-power standby mode entered by executing the PIC18 instruction `SLEEP` (has no arguments). Sleep mode stops the internal clock, thus freezing all register contents and dramatically lowering power consumption. One way to exit sleep mode is for the WDT to expire; even though the internal clock is stopped, the WDT continues running because it has an independent, internally generated clock source. Sleep mode exit caused by WDT expiration is called *WDT wake-up*. When WDT wake-up occurs, the PIC18 resumes at the instruction immediately following the `SLEEP` instruction. One can envision an application in which the PIC18 enters sleep mode

in-between reading data from external sensors, with the WDT used to wake the processor for the next sensor reading.

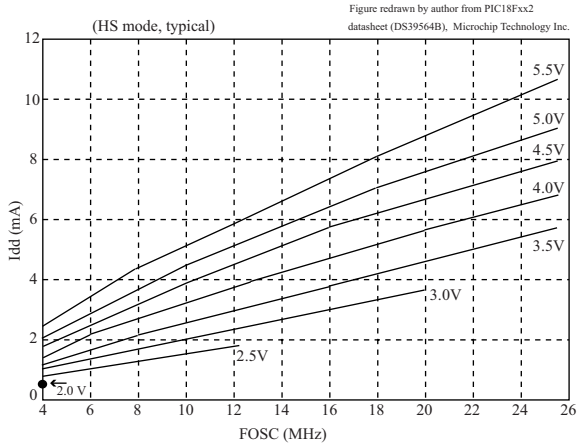
## CMOS Power Consumption

Why does entering sleep mode decrease power consumption? The answer is intuitive; if the clock is stopped, no transistors are switching, which means no energy is being dissipated, thus reducing power. Of course, *some* energy is being dissipated, even with the clock stopped, but the amount is much lower than with the clock running. Power dissipation in a CMOS circuit is divided into two categories: static ( $P_s$ ) and dynamic ( $P_d$ ). Static power is the power dissipated when no switching activity is occurring and is due to high-resistance leakage paths between Vdd and ground within CMOS transistors. Power dissipation is measured in watts (1 Watt = 1 Volt \* Amp), but in datasheets, power dissipation is typically given as power supply current for a particular operating condition. Typical sleep mode current for the PIC18Fxx2@4.2 V with the WDT enabled is 13  $\mu$ A. When transistors are switching, dynamic power is dissipated. The principle contribution to dynamic power dissipation  $P_d$  is given in Equation 8.1, where Vdd is the power supply voltage,  $f$  the switching frequency, and  $c$  the amount of capacitance being switched.

$$P_d = Vdd^2 * f * c \quad (8.1)$$

Either Vdd or the clock frequency can be reduced to lower  $P_d$ ; the capacitance that is switched each clock cycle depends on the circuit topology, which is fixed. It is obviously more effective to lower Vdd if possible due to the square relationship between Vdd and  $P_d$ .

Figure 8.9 shows curves of Idd versus FOOSC (clock frequency) for several different Vdd curves (data taken from the PIC18Fxx2 datasheet [6]). It is apparent that lower Vdd and/or lower clock frequency reduces power consumption. Also observe that for Vdd = 2.5 V, the maximum clock frequency shown is approximately 12 MHz. This is because as Vdd is lowered, transistors take longer to switch, causing CMOS gates to have longer propagation delays, thus reducing the maximum clock speed at which a sequential circuit can be switched. This is the tradeoff associated with lowering Vdd to reduce power dissipation—the maximum achievable performance is also reduced. Note that current draw at @ 4 V, 20 MHz is about 5.5 mA, so the sleep mode current of 13  $\mu$ A reduces power supply current by a factor greater than 400!



**FIGURE 8.9**  $I_{dd}$  versus FOSC.<sup>2</sup>

**Sample Question:** Assume a PIC18 is consuming 13 mA@5 V, 40 MHz. If the voltage, frequency is reduced to 4 V, 20 MHz, what is the new predicted current draw based on Equation 8.1?

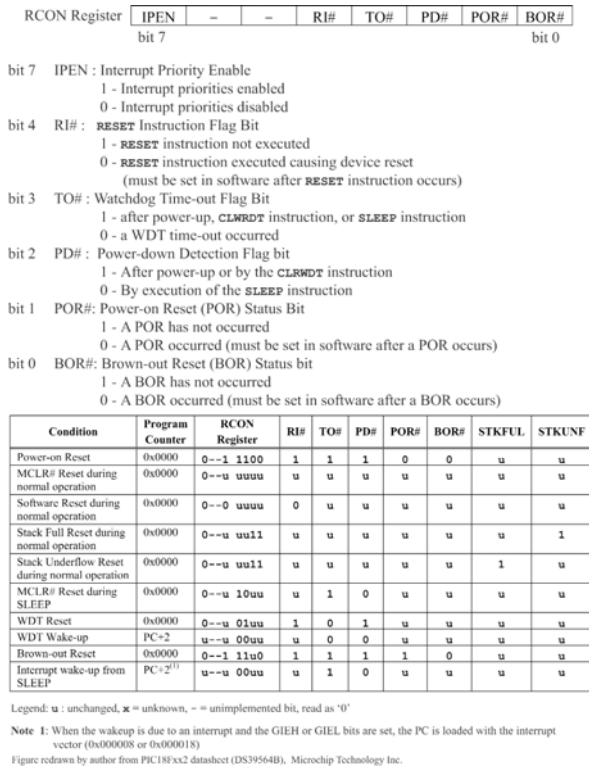
**Answer:** It is known that  $13 \text{ mA} \sim 5 \text{ V} * 5 \text{ V} * 40 \text{ MHz} * c$ , so  $c \sim 13 \text{ mA} / (5 \text{ V} * 5 \text{ V} * 40 \text{ MHz})$ .

We are looking for  $y \text{ mA} \sim 4 \text{ V} * 4 \text{ V} * 20 \text{ MHz} * c$ . Replacing  $c$  in this equation gives  $y \text{ mA} \sim 16 * 20 * 13 \text{ mA} / (25 * 40) = 4.2 \text{ mA}$ , expected current draw.

## 8.8 EXPERIMENTING WITH RESET, SLEEP, AND THE WATCHDOG TIMER

The RCON register contains status bits that are used for determining the reset type that occurred. Figure 8.10 defines the RCON register bits and their values under various reset conditions. It is important to be able to determine the reset source in order to execute different code segments based on reset type. For example, after a watchdog timer reset, if the WDT is being used as a timeout for some type of IO operation, status information can be displayed to help determine the reason for the timeout. The POR# bit (Power-on Reset Status bit) is cleared to “0” on power-on reset (POR). After power-on reset has occurred, this bit has to be manually set to “1” in software to distinguish future resets from a power-on reset.

<sup>2</sup> Figure 8.9 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.’s prior written consent.



**FIGURE 8.10** RCON register definition, bit values under different reset conditions.<sup>3</sup>

Figure 8.11 shows a C program useful for experimenting with the different PIC18 reset conditions. The code assumes the existence of a serial port connection for displaying the results of the printf() statements and reading serial port input, which is discussed in Chapter 9.

On main() entry, the program initializes the serial port, and then determines if a power-on reset or watchdog timer reset has occurred. If the test POR == 0 is true, power-on reset has occurred and the variable reset\_cnt is initialized to zero. Observe that POR = 1 is included, as this bit is unaffected by other reset types, so it must be set in order to detect other reset types. In addition, the persistent qualifier is used for the declaration of reset\_cnt, excluding it from being initialized by the start-up code executed before main() entry. This means that reset\_cnt is only initialized in the if{} body of the POR test, allowing this variable to keep a count of the number of non-POR resets experienced. If the test TO == 0 is true, the watchdog timer has expired, causing a reset. The watchdog timer is then disabled by clearing

<sup>3</sup> Figure 8.10 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.'s prior written consent.

```

void pcrLf (void) // print a newline to terminal
{
    putchar(0x0a);  putchar(0x0d);
}

persistent char reset_cnt;

main(void) {
    int i;
    char c;

    serial_init(95,1); // 19200 in HSPLL mode, crystal = 7.3728 MHz
    pcrLf();
    if (POR == 0) {
        printf("Power-on reset has occurred."); pcrLf();
        POR = 1; // setting to bit to 1 means that will
                // remain a '1' for other reset types
        reset_cnt = 0;
    }
    if (TO == 0) {
        SWDTEN = 0; // disable watchdog timer
        printf("Watchdog timer reset has occurred."); pcrLf();
    }
    i = reset_cnt;
    printf("Reset cnt is: %d",i);
    pcrLf();
    reset_cnt++;
    while(1) {
        printf("'1' to enable watchdog timer"); pcrLf();
        printf("'2' for sleep mode"); pcrLf();
        printf("'3 ' for both watchdog timer and sleep mode"); pcrLf();
        printf("Anything else does nothing, enter keypress: ");
        c = getch();
        putchar(c);
        pcrLf();
        if (c == '1') SWDTEN = 1; // enable watchdog timer
        else if (c == '2') asm("sleep");
        else if (c == '3') {
            SWDTEN = 1; // enable watchdog timer
            asm("sleep");
        }
    }
}

```

Annotations in the original image:

- Arrow from "persistent" to `reset_cnt`: persistent qualifier prevents variable from being initialized by start-up code prior to main
- Arrow from "serial\_init" to `serial_init(95,1)`: Initializes serial interface, discussed in Chapter 9.
- Arrow from "Detects power-on reset" to `if (POR == 0)`
- Arrow from "reset\_cnt only initialized here" to `reset_cnt = 0;`
- Arrow from "Detects watchdog timer expiration" to `if (TO == 0)`
- Arrow from "reset\_cnt increases in value for each non-POR reset" to `reset_cnt++;`
- Arrow from "Inline assembly used to insert SLEEP instruction" to `asm("sleep");`



**FIGURE 8.11** Program (*reset.c*) for experimenting with reset types.

the SWDTEN bit, which is bit 0 of the WDTCON register. After these two reset checks, the `reset_cnt` variable is incremented, and an infinite `while(1) {}` loop is entered. A choice menu is printed within the loop, giving the user the option of enabling the watchdog timer, entering sleep mode, or enabling the watchdog timer, and then entering sleep mode. Choice #1 enables the watchdog timer via the statement `SWDTEN = 1`, which sets the SWDTEN bit. If nothing else is done after this choice is made, the WDT expires after approximately two seconds assuming the WDT postscaler is set to 128:1. This causes a reset, and the message “Watchdog timer reset has occurred” is printed by the code that detects this condition. Choice #2 enters sleep mode by executing the SLEEP instruction, which is inserted in the C code by the statement `asm(“SLEEP”)`. This is known as *inline assembly*, and allows assembly language instructions to be specified directly within C code. After this choice is made, the only method of waking the processor is to press the MCLR# reset button, or to cycle power (assuming the schematic of Figure 8.4). Choice #3 first enables the watchdog timer, and then enters sleep mode. Two seconds later, the watchdog timer expires, waking the processor from sleep mode, and causing execution to resume at the instruction following the SLEEP instruction. Since this is at

the end of the loop, a jump is made back to the beginning of the loop, and the choice menu is reprinted. If no choices are made, the watchdog timer expires again after another two seconds, causing a device reset. Extending this code to check for other reset conditions, or generating other types of reset is left for the review problems at the end of the chapter. Figure 8.12 shows terminal output from testing the code of Figure 8.11.

```

Power-on reset has occurred.
Reset cnt is: 0 ←———— Power-on reset, so reset_cnt is 0
'1' to enable watchdog timer
'2' for sleep mode
'3' for both watchdog timer and sleep mode
Anything else does nothing, enter keypress: ←———— Pressed reset button to generate
Reset cnt is: 1 ←———— MCLR# reset, so
reset_cnt increments
'1' to enable watchdog timer
'2' for sleep mode
'3' for both watchdog timer and sleep mode
Anything else does nothing, enter keypress: 1 ←———— Typed "1", WDT enabled.
'1' to enable watchdog timer
'2' for sleep mode
'3' for both watchdog timer and sleep mode
Anything else does nothing, enter keypress: ←———— Menu reprinted,
but no input typed,
Watchdog timer reset has occurred. ←———— WDT reset occurs, WDT disabled.

Reset cnt is: 2 ←———— reset_cnt increments.
'1' to enable watchdog timer
'2' for sleep mode
'3' for both watchdog timer and sleep mode
Anything else does nothing, enter keypress: 2 ←———— Typed "2", so sleeps.
Reset cnt is: 3 ←———— Must press reset to wakeup,
incrementing reset_cnt
'1' to enable watchdog timer
'2' for sleep mode
'3' for both watchdog timer and sleep mode
Anything else does nothing, enter keypress: 3 ←———— Typed "3", so WDT enabled
and sleep.
'1' to enable watchdog timer ←———— WDT expires, waking PIC.
'2' for sleep mode ←———— Loop and re-display menu.
'3' for both watchdog timer and sleep mode
Anything else does nothing, enter keypress: ←———— No input typed, WDT reset occurs,
Watchdog timer reset has occurred. ←———— WDT disabled.

Reset cnt is: 4 ←———— reset_cnt increments.
'1' to enable watchdog timer
'2' for sleep mode
'3' for both watchdog timer and sleep mode
Anything else does nothing, enter keypress: ←———— Cycle Power
Power-on reset has occurred.
Reset cnt is: 0 ←———— Power-on reset, so reset_cnt is 0
'1' to enable watchdog timer
'2' for sleep mode
'3' for both watchdog timer and sleep mode
Anything else does nothing, enter keypress:

```

**FIGURE 8.12** Testing the *reset.c* program.

## 8.9 PARALLEL PORT OPERATION

Parallel port IO refers to groups of pins whose values can be read or written as a group via special function registers. On the PIC18F242, three parallel ports are available: PORTA, PORTB, and PORTC. Two additional ports, PORTD and

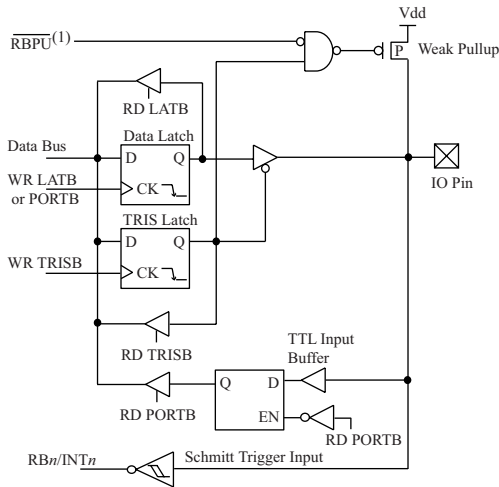
PORTE, are available on larger pin count versions of the PIC18Fxx2. Only PORTA, PORTB, and PORTC are discussed in this book, as the PIC18F242 is the target device for the interfacing examples. Each parallel port has three special function registers associated with it: PORT $x$ /TRIS $x$ /LAT $x$ , where  $x$  is A, B, or C. The TRIS $x$  register is used to configure each bidirectional port bit as either an input or output. A “1” in a TRIS $x$  register bit configures the associated PORT $x$  register bit to be an input, while a “0” configures the associated PORT $x$  register bit to be an output. The LAT $x$  register is the data latch used to drive the port pins when it is configured as an output. Reading PORT $x$  returns the values of the external pins, while reading LAT $x$  reads the data latch value. Writing to either LAT $x$  or PORT $x$  writes to the data latch of the associated port. Please note that reading LAT $x$  may not return the same value as reading PORT $x$ . If the port is configured as an input, reading LAT $x$  returns the last value written to LAT $x$  or PORT $x$ , while reading PORT $x$  returns the value of the external pin. If the port is configured as an output, reading LAT $x$  will normally return the same value as reading PORT $x$  because the data latch is driving the external pin. However, if there is another external driver that is clashing with the port driver, or if the port driver itself is a special case like an open drain output (explained later in this section), LAT $x$  and PORT $x$  may return different values when read. A write to a port bit configured as an input changes the value of the output data latch (LAT $x$ ), but does not change the value seen on the external pin whose value is set by whatever is driving that pin.

## PORTB

Figure 8.13 shows the internal logic of port pins RB[2:0]. Each port has slightly different features, and the PIC18Fxx2 datasheet should be consulted for a complete description of each port’s capabilities. This diagram clearly shows the differences between the PORTB, LATB, and TRISB special function registers. The TRISA register controls port direction, PORTB represents the state of the external pins, and LATB holds the data used to drive the port pins.

Figure 8.13 shows that the TRISB[ $y$ ] bit is connected to the enable input of the *tristate buffer* that is on the output data latch that drives the PORTB pin. If TRISB[ $y$ ] is “0” (the port bit is an output), the tristate buffer is enabled, and simply passes its input value to its output. If TRISB[ $y$ ] is “1” (the port bit is an input), the tristate buffer is disabled, and its output becomes high impedance, whose state is commonly designated as “Z”. Figure 8.14 shows that one can think of the tristate buffer enable as controlling a switch on the output of the buffer; the switch is closed when the enable is asserted, allowing the buffer to drive its output. The switch is open when the enable is negated, causing the output to float (also known as high impedance). Note that a port bit cannot be both an input and an output simultaneously; it is either one or the other based on the setting of the TRISB[ $y$ ] bit.

The lower part of Figure 8.14 shows data flow on a bidirectional data link using one wire; data is either flowing from CPU\_a to CPU\_b or vice versa, but never both directions at the same time over one wire if voltage mode signaling is used.



(1) To enable weak pull-ups, set the appropriate TRIS bit(s) and clear the RBPB bit (INTCON2[7])  
 Figure redrawn by author from PIC18Fxx2 datasheet (DS39564B), Microchip Technology Inc.

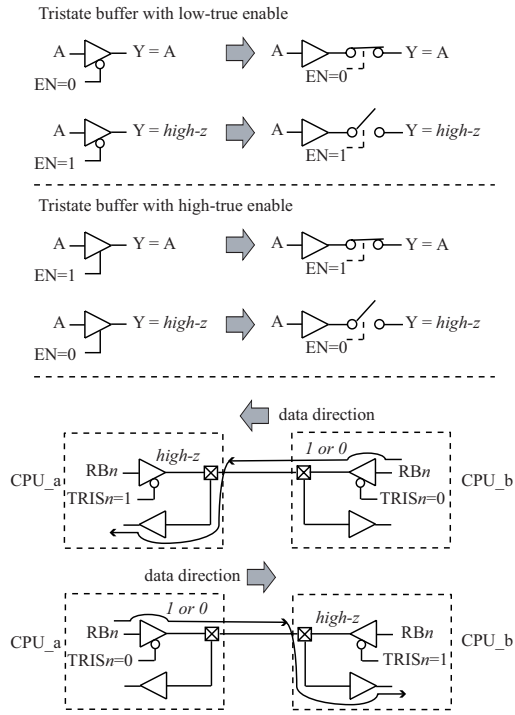
**FIGURE 8.13** Port RB2:RB0 pin internals.<sup>4</sup>

In Figure 8.13, if a port is configured as an input and the RBPB bit (INTCON2[7]) is “0”, the *weak pullup* is enabled; observe that this enables the weak pullup for all PORTB pins configured as inputs. The weak pullup is implemented as a high-resistance P-transistor; when enabled, the gate of this transistor is “0”, turning it on. The weak pullup is useful for eliminating the need for an external pullup resistor on an input switch (see Figure 8.15). The term *weak* is used because the resistance is high enough that an external driver can overpower the pullup resistor and pull the input to near ground, producing a “0” input. PORTB is the only port with internal weak pullups. Note that a pushbutton switch configured as a low-true input switch must have some form of pullup resistor, either internal or external, or the input floats when the pushbutton is not pressed, allowing the input to be read as either “1” or “0”.

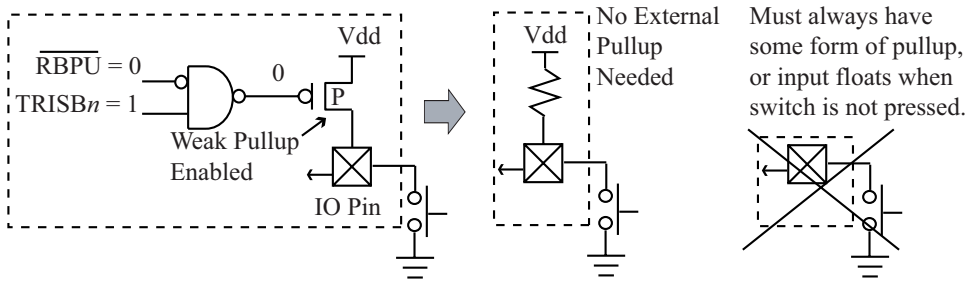
The PORTB pin logic of Figure 8.13 shows both a TTL buffer and a *Schmitt trigger* buffer used for driving internal signals from the external pin. Figure 8.16 shows the  $V_{in}/V_{out}$  characteristics of these two input buffer types. Observe that for a Schmitt trigger, a low-to-high transition must become close to  $V_{dd}$  before the

<sup>4</sup> Figure 8.13 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.’s prior written consent.





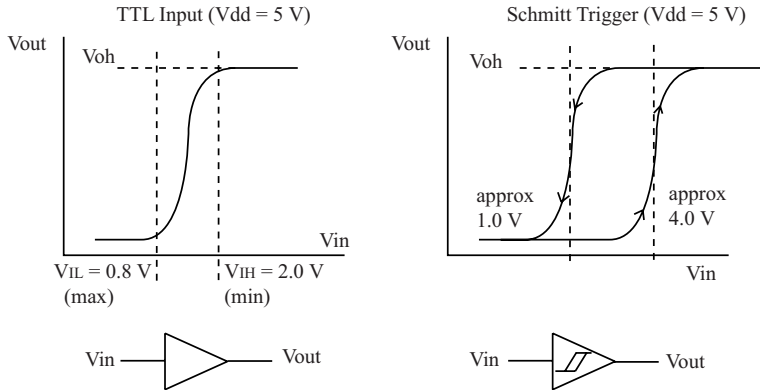
**FIGURE 8.14** Tristate buffer operation and bidirectional IO.



**FIGURE 8.15** Weak pullup operation.

buffer trips; conversely, a high-to-low transition must be close to ground before the buffer trips. This *hysteresis* in the buffer action provides extra immunity against noise on the input signal, and is also used for transforming slowly changing inputs into signals with fast rise and fall times. Schmitt triggers consume more power than

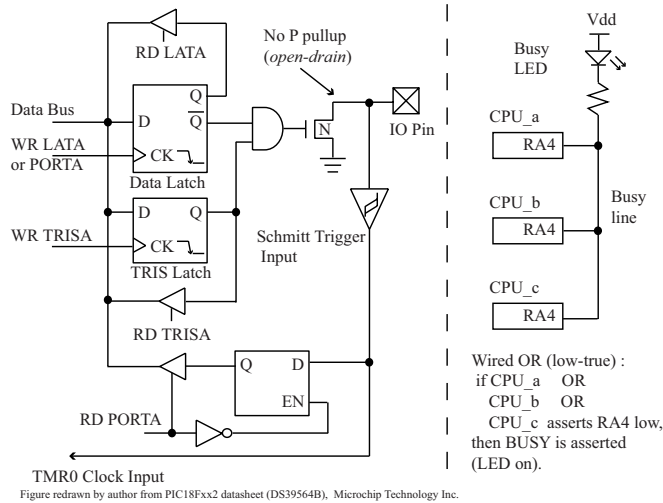
TTL inputs, and thus are only used on inputs that either provide internal interrupts (such as RB0, RB1, RB2, MCLR#) or drive the clock signal of an internal subsystem.



**FIGURE 8.16** TTL buffer versus Schmitt trigger buffer PORTA.

PORTA pins are shared with the analog-to-digital converter subsystem, and the default configuration is for these pins to be analog inputs, causing a “0” to always be read for those inputs when reading special function register PORTA. To configure all PORTA pins for digital operation, the statement `ADCON1 = 0x06` is required; other values written to the ADCON1 register provide different combinations of analog/digital inputs (see the PIC18Fxx2 datasheet and Chapter 12 for more details). Pins RA0:RA3 and RA5 provide bidirectional IO, CMOS logic levels for output, and use a TTL buffer for input. Pin RA4 is different; it is an *open-drain* output configuration, which can either drive its output low or leave it floating (see Figure 8.17). The term *open-drain* is used because the drain terminal of the N pull-down transistor is open; there is no P pull-up transistor. The right-hand side of Figure 8.17 shows a common use of open-drain outputs, which is to implement *wired logic*. The three RA4 outputs of the CPUs are wired directly together, and pulled up to Vdd through an external resistor. This is a low-true *wired-or* configuration because the BUSY line is asserted low, turning on the LED, whenever CPU\_a or CPU\_b or CPU\_c asserts its RA4 output low. If all RA4 outputs are floating, the BUSY line is high, and the LED is turned off. You cannot connect normal CMOS outputs directly together like this because there will be a clash when one output drives high, and another output drives low, resulting in an uncertain voltage level on the wire. A common mistake is to use RA4 as a normal output that must provide both high and low voltage levels. This is difficult to debug, as a “1” written to RA4 causes the output to float, which can be read by the receiving logic (if it is a CMOS input) as

either a “1” or “0”. Thus, the system will fail intermittently, which is one of the most difficult types of hardware problems to debug.



**FIGURE 8.17** Pin RA4, open-drain output.<sup>5</sup>

## PORTC

PORTC pins are bidirectional, with CMOS output drivers. In the PIC18F242, the PORTC output pins are shared with other subsystem functions that are used in this book’s interfacing examples. As such, the examples in this book use PORTB and PORTA pins for any parallel IO needs. Details on the PORTC parallel port logic, as well as PORTD and PORTE can be found in the PIC18Fxx2 datasheet.

**Sample Question: Write C code that configures pins RB0, RB2, RB4 as outputs, and the rest of PORTB as inputs.**

*Answer:* Recall that if a TRISx bit is a “1”, the port pin functions as an input; a “0” configures the port as an output. This configuration can be accomplished in two ways; either by a single assignment to the TRISB register or by individual TRISB bit assignments. The statement `TRISB = 0xEA (0b11101010)` works, as this clears bits TRISB4, TRISB2, TRISB0 to “0” and sets the rest as “1”. If the TRISB configuration occurs after power-on reset (POR), then the statements:

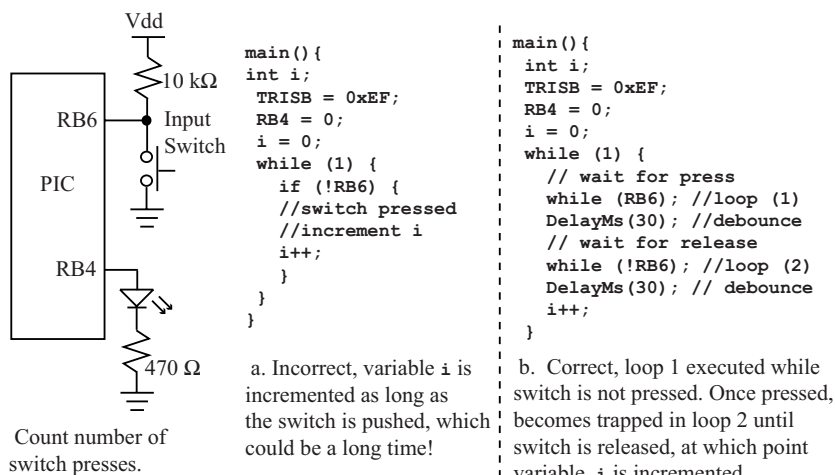
```
TRISB4 = 0; TRISB2 = 0; TRISB0 = 0;
```

<sup>5</sup> Figure 8.17 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.’s prior written consent.

also works, as all bits in the TRISB register are reset to “1” after power-on reset (see Appendix A for the settings of all special function register bits after POR). Obviously one statement is more efficient than multiple statements, but if you are new to microcontrollers it is suggested that you use the coding style that you understand the best.

## 8.10 LED/SWITCH IO AND STATE MACHINE PROGRAMMING

A common input device is a momentary pushbutton switch. Figure 8.18 shows a pushbutton switch connected to RB6. When the pushbutton is released (not pressed) the RB6 input reads as “1”; when the pushbutton is pressed the RB6 input reads as “0”.



**FIGURE 8.18** LED/switch IO example #1.

Assume we would like to count the number of pushbutton presses and releases; each press and release counts as one switch activation. A common mistake is shown in code segment (a), which increments a variable *i* when RB6 returns “0”. The problem with this code is that the variable *i* is not only incremented when the pushbutton is pressed, but is also incremented for as long as the pushbutton is held down. Human reaction times on pushbutton switches are measured in tens of milliseconds, so *i* is incremented many times for each pushbutton activation!

Code segment (b) shows a correct solution to this problem. When the `while(1){}` loop is entered, the code becomes trapped in the loop `while(RB6){}`,

which loops waiting for the pushbutton to be pressed. Once the pushbutton is pressed, the code is then trapped in the loop `while(!RB6){}`, waiting for the pushbutton to be released. Upon release, the variable `i` is incremented and the code becomes trapped in the loop `while(RB6){}` again. Thus, `i` is incremented only once for each press and release of the pushbutton. The `DelayMs(30)` function calls are included after each change in the input switch status to ensure that all switch bounce has settled before continuing. Mechanical switch bounce can produce multiple pulses when a pushbutton is activated. The required delay is a function of the mechanics of the switch bounce, which can only be seen by using an oscilloscope to capture the switch waveform or from a manufacturer data sheet. The value of 30 ms used here should be adequate for common momentary switches. This is a simple method for debouncing a switch with the drawback that the CPU cycles spent in the software delay loop are wasted. Alternate methods for switch debouncing are presented in Chapter 10.

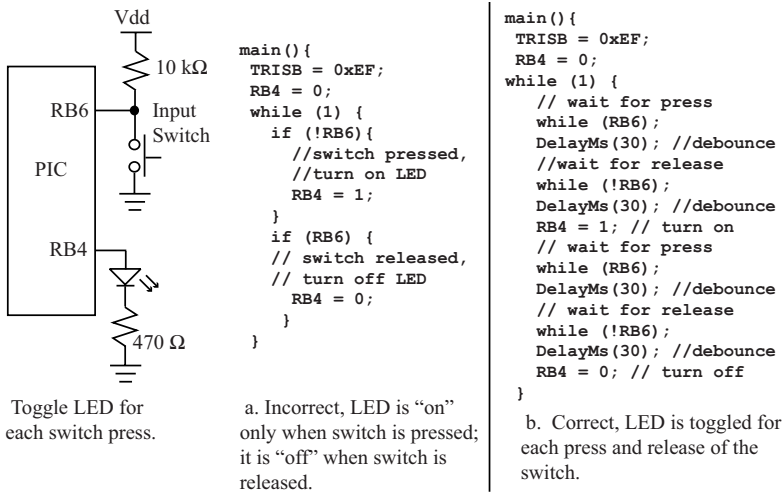
Another example of LED/switch IO is given in Figure 8.19, in which the goal is to toggle the LED each time the pushbutton is pressed and released. Code segment (a) is incorrect, as the `if(!RB6)` statement only turns on the LED as long as the pushbutton is held down. Code segment (b) is correct, as the two statements `while(RB6){}`, `while(!RB6){}` wait for a press and release before turning on the LED; then the next two statements `while(RB6){}`, `while(!RB6){}` wait for a subsequent press and release before turning off the LED. The bit assignments `LB4 = 1`, `LB4 = 0` can be used instead of `RB4 = 1`, `RB4 = 0`, as this also writes to the port data latch register. For the PICC-18 compiler, data latch bits can also be referenced using the *LAT* prefix (e.g., `LATB4`).

**Sample Question:** Assume the same LED/switch configuration of Figure 8.19. Write a `while(1){}` loop that blinks the LED twice for each switch press and release. Assume the port is already configured.

*Answer:* One solution is shown in Listing 8.3.

### LISTING 8.3 Solution A.

```
while(1){
    while(RB6); DelayMs(30); //wait for press
    while(!RB6); DelayMs(30); // wait for release
    //blink twice
    RB4 = 1; DelayMs(200); //turn on, delay for blink
    RB4 = 0; DelayMs(200); //turn off, delay for blink
    RB4 = 1; DelayMs(200); //turn on, delay for blink
    RB4 = 0; DelayMs(200); //turn off, delay for blink
}
```



**FIGURE 8.19** LED/switch IO example #2.

The `DelayMs(200)` software delay is necessary to actually see the LED turning off and on. A common mistake is to forget to include this delay; the RB4 pin still toggles but the LED will appear to be dimly “on,” as it cannot respond fast enough to the changes on the RB4 pin. An alternate solution is shown in Listing 8.4.

**LISTING 8.4** Solution B.

```

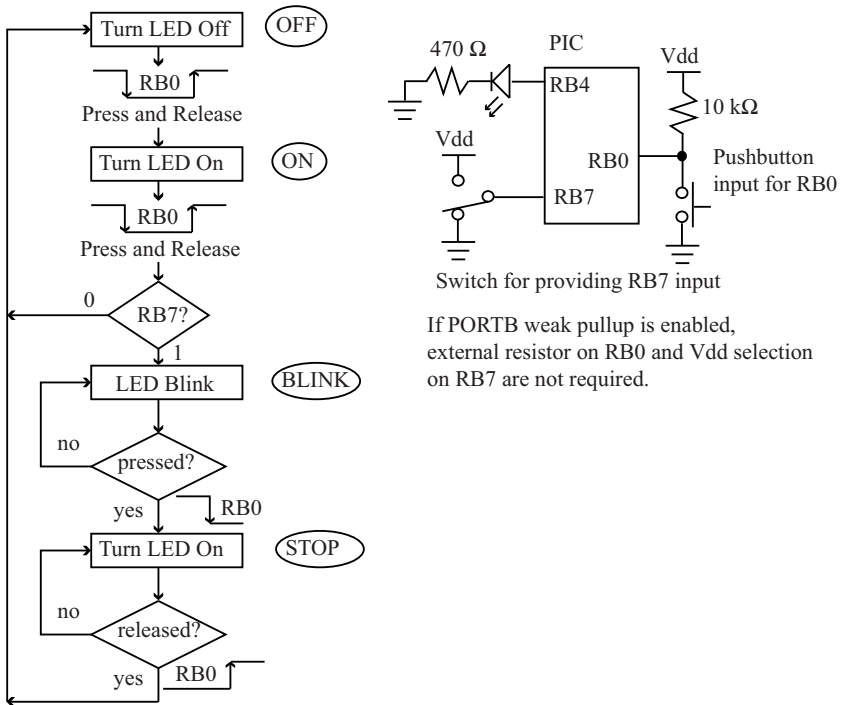
while(1){
    char i;
    while(RB6); DelayMs(30); //wait for press
    while(!RB6); DelayMs(30); // wait for release
    //blink twice
    for (i=0;i!=(2*2); i++) { //four times thru loop blinks twice
        if (LB4)RB4 = 0; else RB4 = 1; //toggle RB4
        DelayMs(200);
    } //end for
} //end while

```

The `for{} loop` iterates four times; each pass through the loop the LED is either turned off or on, so the LED is blinked for every two passes through the loop. The LED is toggled by reading the status of the data latch bit LB4; if it is “1”, it is cleared to “0”; else it is set to “1”. Other solutions are possible; it is suggested that you use the coding style that you understand the best.

### State Machine IO Programming

The loop structures of Figures 8.18 and 8.19 wait for an IO event (switch press and release) and then perform an action. A common task in microcontroller applications is to perform a sequence of events that span a series of IO actions. A finite state machine approach for code structure is useful for these types of problems. Figure 8.20 shows a state machine specification of an LED/switch IO problem. Each state accomplishes an action, such as turning the LED off, turning the LED on, or blinking the LED. Transitions between states are controlled by an event on the pushbutton, which is a press, a release, or both a press and release. State OFF turns the LED off and transitions to state ON by a press and release. State ON turns on the LED and transitions to the next state on a press and release. The next state from the ON state is state OFF if the RB7 input is 0; else the next state is the BLINK state. The BLINK state flashes the LED until the pushbutton is pressed, at which point it transitions to state STOP. The stop STATE freezes the LED on as long as the pushbutton is pressed. State STOP transitions to state OFF when the pushbutton is released.



**FIGURE 8.20** State machine specification of LED/switch IO.

Figure 8.21 gives a C code implementation of the LED/switch IO state machine of Figure 8.20. The `#define` statements define labels for each state with the state assignment arbitrarily chosen to start at 0. In a software state machine, the state assignments are usually unimportant, unlike a hardware finite state machine in which state assignments affect the logic generated for the state machine implementation. The unsigned char state variable is used to keep track of the current state.

```

// State definitions
#define LED_OFF    0 // turn off
#define LED_ON    1 // turn on
#define LED_BLINK 2 // start blinking
#define LED_STOP  3 // stop blinking
} State Definitions
unsigned char state; // Variable for tracking current state
main(void) {
    serial_init(95,1); // 19200 in HSPLL mode, crystal = 7.3728 MHz
    pcrflf(); // this subroutine prints a newline to the terminal
    printf("Led Switch/IO started"); pcrflf();
    // RB4 is the output, RB7, RB0 are inputs
    TRISB = 0xEF; LATB = 0x00; STATE = LED_OFF;
    // enable the weak pullup on port B
    RBPU = 0; // Enable weak pullup
    while(1) {
        switch (state) {
            case LED_OFF:
                printf("LED_OFF"); pcrflf(); // printf statements in each state are
                LATB4 = 0; // Could use RB4 here as well // included for debugging.
                while(RB0); DelayMs(30); // wait for press
                while(!RB0); DelayMs(30); // wait for release
                state = LED_ON; // Change state variable so next time through
                break; // loop will execute new case block.
            case LED_ON:
                printf("LED_ON"); pcrflf();
                LATB4 = 1;
                while(RB0); DelayMs(30); // wait for press
                while(!RB0); DelayMs(30); // wait for release
                if (RB7) state = LED_BLINK; // Chooses next state based on RB7 value
                else state = LED_OFF;
                break;
            case LED_BLINK:
                printf("LED_BLINK"); pcrflf();
                while (RB0) { // while not pressed
                    // toggle LED
                    if (LATB4) LATB4 = 0; // Toggles LED each time through the loop,
                    else LATB4 = 1; // delay so we can see LED blink
                    DelayMs(250);
                }
                DelayMs(30);
                state = LED_STOP;
                break; // Must have break at end of each case block
            case LED_STOP:
                printf("LED_STOP"); pcrflf();
                LATB4 = 1; // freeze on // or will execute next case block!!!!
                while(!RB0); DelayMs(30); // wait for release
                state = LED_OFF;
                break;
        }
    }
}

```

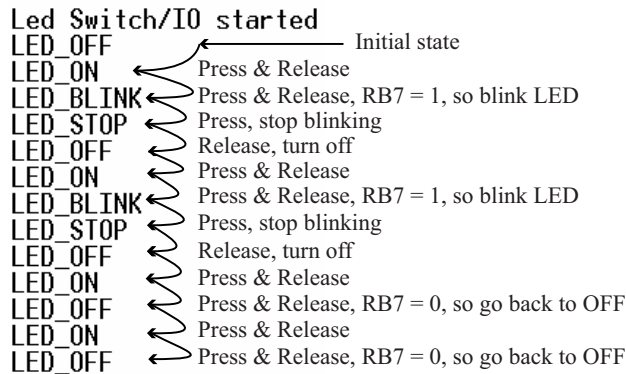


**FIGURE 8.21** C code for LED/switch IO.



The `main()` code performs initialization of the serial port and `PORTB`, and then enters a `while(1){}` loop that uses a `C` switch statement to execute different code segments based upon the `state` variable. A `printf()` statement that prints the state name is the first statement in each case block and is included for debugging purposes. Each case block performs its associated action and only changes the state variable to the next state once its specified pushbutton event is detected. It is very important to end each case block with a `break` statement, or else the next case block code is executed regardless of the `state` value. Reading the current state of `LB4` (data latch port B, bit 4) and complementing it toggles the LED in the `LED_BLINK` state. A read of `RB4` can be used here instead of `LB4`, because the external pin value will be the same as the data latch value because there are not multiple drivers on the `RB4` external pin, so no possibility of driver conflict exists. However, in general, if you need to read the last value written to an output port, the data latch register should be read instead of the port register.

Figure 8.22 shows console output while testing the `C` code of Figure 8.21. The `RB7` input was “1” for the first two times that the `LED_ON` state was exited, causing the next state to be `LED_BLINK`. After this, the `RB7` input was low the next two times that the `LED_ON` state was exited, causing the following state to be `LED_OFF`.

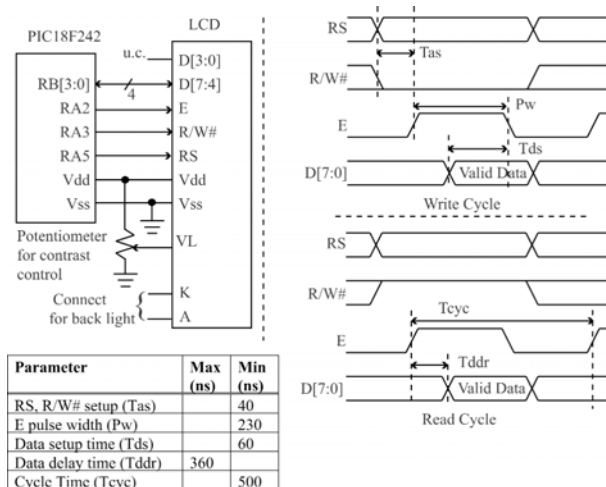


**FIGURE 8.22** Console output for LED/switch IO C code.

## 8.11 INTERFACING TO AN LCD MODULE

A liquid crystal display (LCD) is often used in microcontroller applications, as they are low power and can display both alphanumeric and graphics. Disadvantages of LCDs are that they have low viewing angles, are more expensive than LED displays, and must be lit by either ambient light or a back light. LCD *modules* display multiple characters; with part numbers using a  $k \times n$  designation where  $k$  is the number

of characters displayed on each of the  $n$  display lines. LCD modules have either a parallel or serial interface, with many LCD parallel interfaces standardized around the Hitachi HD44780 LCD controller. Figure 8.23 shows a PIC18 to LCD interface for a Hantronix 16x2 LCD module (part# HDM16216L-5). This interface is independent of the  $k \times n$  organization of the LCD module, and is applicable for most LCD modules based on the HD44780 LCD controller.



**FIGURE 8.23** PIC18 to LCD interface (4-bit mode).

The interface is divided into control lines (E, R/W#, RS), data lines (D7:D0), and power (Vdd, Vss, VL, K, A). The 4-bit interface mode is used to reduce the number of connections between the PIC18 and the LCD. In 4-bit mode, 8-bit data is sent in two 4-bit transfers on lines D7:D4, allowing D3:D0 to be unconnected. The K, A inputs are for the back light display (see datasheet [9]), while the Vdd – VL voltage difference determines the intensity of the displayed numerals (connecting VL to VSS provides maximum intensity but may cause Vdd – VL to exceed the maximum recommended VL value on some LCD modules). A logic high on the R/W# signal indicates a read operation; the LCD module provides data to the PIC18. A logic low on R/W# is a write operation; the PIC provides data to the LCD module. The E signal is a data strobe used to signal valid data on the  $D_n$ , RS, and R/W# lines. To perform a write operation, the PIC places data on the  $D_n$ /RS signals, R/W# is driven low, and E is brought high and then low. The LCD latches the input data on the falling edge of E, so the  $D_n$  lines must satisfy the setup time  $T_{ds}$ , while the control lines RS, R/W# are latched on the rising edge of E and must satisfy the setup time  $T_{as}$ . Not all timings are shown on the diagram; there are small

hold times about the E edges that must be satisfied as well (see [9], these are easily satisfied by typical microcontroller operation). To read data from the LCD, data is placed on the RS signal, R/W# is driven high, and E is brought high. After the data delay time  $T_{ddr}$  has elapsed, valid data is ready on the  $D_n$  lines from the LCD, which can then be read by the PIC. The E signal is brought low to finish the read operation. The use of the PIC18 RA2, RA3, and RA5 pins for the LCD control signals is an arbitrary choice; however, you must be careful to not use pin RA4 for any of these control signals, as the open drain structure of this port prevents it from providing a high output.

A subset of the available LCD commands [10] is shown in Table 8.2. If  $RS = 0$ , the D7:D0 bits represent an LCD command that affects mode, screen, or cursor position. If  $RS = 1$ , the D7:D0 bits contain data being written to or read from the LCD *data display RAM* (DD RAM) in the form of an ASCII character code.

The internal memory configuration of an LCD is dependent upon the particular module. The HDM16216L-5 is a 16x2 display, but its internal data display RAM has 80 total locations with 40 locations mapped to line 1 (addresses 0x00 to 0x27) and 40 locations mapped to line 2 (addresses 0x40 to 0x67). The 16x2 display is a window into these 80 locations, with only 16 characters of each line displayed at any given time as shown in Figure 8.24. By default, the display shows locations 0x00-0x0F of line 1, and locations 0x40-0x4F of line 2. A left shift moves the display to the right, causing locations 0x01-0x10 to be displayed in line 1, and locations 0x41-0x50 in line 2. This creates the appearance that the displayed line has shifted one position to the *left*, as the leftmost character disappears, and the character in column 1 now appears in column 0. Continual left shifting causes the lines to scroll marquee-fashion, moving right to left across the display.

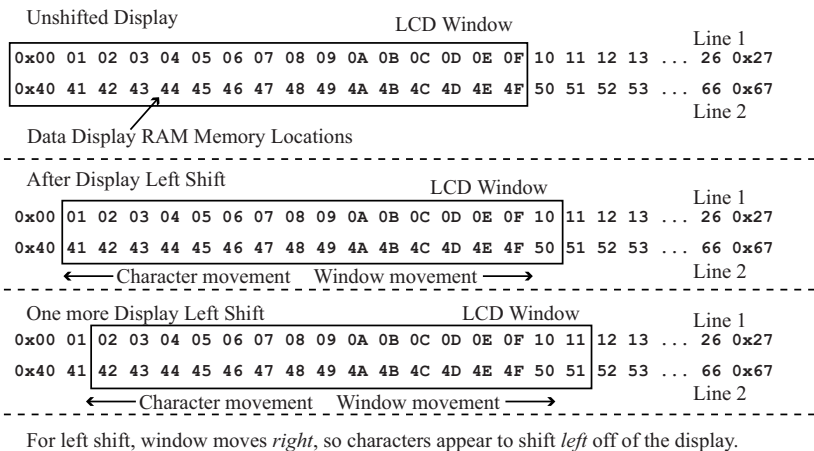


FIGURE 8.24 LCD data display RAM.

An internal address counter determines where the current data write operation places data. The address counter also specifies the cursor location. Initializing the display sets the address counter to zero, placing the cursor to the home position of location 0 (position 0 of line 1, upper left-hand corner of the display). A write data operation writes data to the current address location, and then increments or decrements the address counter depending on the mode setting (*entry mode set* command in Table 8.2). In increment mode, the address counter is incremented by one and the cursor moves one position to the right on the display. Each additional write places data at the current address counter location, and increments the address counter. Assuming the display is unshifted, the 17<sup>th</sup> write (to location 16) places data “off-screen” (the data is not visible), but the data is still contained in DD RAM. A right shift of the display has to be performed to see the data contained in location 16. Each LCD command requires a fixed amount of time execute. The PIC software communicating with the LCD can either have built-in delays that are guaranteed to exceed the required LCD command execution time, or the LCD can be polled via the *read busy flag* command to determine if the module is ready for another command. Before sending a command, a *polling loop* is used to continually read the busy flag status; the loop is exited when the busy flag returns “0”. Other commands exist for loading custom character fonts; see the datasheet [10].

**TABLE 8.2** LCD Command Subset

<b>Command</b>	<b>RS</b>	<b>R/W#</b>	<b>D7:D0</b>	<b>Description</b>
Clear Display	0	0	0000 0001	Clear display, returns cursor to home position (82 $\mu$ s ~ 1.64 ms)
Return Home	0	0	0000 001x	Returns cursor and shifted display to home (40 $\mu$ s ~ 1.64 ms)
Entry Mode Set	0	0	0000 01d0	Enable the display, set cursor move direction ( $d=1$ increment, $d=0$ decrement) (40 $\mu$ s)
Display On/Off	0	0	0000 1dcb	Display on/off ( $d$ ), Cursor on/off ( $c$ ), blink at cursor position on/off ( $b$ ) (40 $\mu$ s)
Cursor & Display Shift	0	0	0001 cr00	$c=1$ shift display, $c=0$ move cursor, $r=1$ right shift, $r=0$ left shift
Function Set	0	0	001i n000	8-bit interface ( $i=1$ ), 4-bit interface ( $i=0$ ), one line ( $n=0$ ), two lines ( $n=1$ ) (40 $\mu$ s) →

Set DD Address	0	0	1nnn nnnn	DD RAM address set equal to nnnnnnnn (40 $\mu$ s)
Read Busy Flag	0	1	fnnn nnnn	Busy flag (f) returns in D7 (1=Busy), D6:D0 contains address counter value (1 $\mu$ s)
Write Data	1	0	nnnn nnnn	Data nnnnnnnn written at current DD RAM address (46 $\mu$ s)
Read Data	1	1	nnnn nnnn	Data nnnnnnnn at current address location in DD RAM is returned (46 $\mu$ s)

Figure 8.25 shows two functions, `epulse()` and `lcd_write()`, that are used for writing to the LCD assuming the interface of Figure 8.23. Both functions use the macro definitions at the top of the page; these macros make code modifications easier if a different selection of parallel port signals is used for the LCD interface. The `EHIGH/ELow`, `RSHIGH/RSLOW`, and `RWHIGH/RLow` macros are used to set the E/RS/RW signal lines high or low. The `E_OUTPUT`, `RS_OUTPUT`, and `RW_OUTPUT` macros are used to configure the E, RS, and R/W# port lines as outputs. The `ADCON1 = 0x06` statement in the `E_OUTPUT` macro is used to configure all pins of PORTA as digital IO; the PORTA pins are analog inputs pins by default, as they are shared with the analog-to-digital converter subsystem. The `BUSY_FLAG` macro returns the value of the port signal that is the MSb of the 4-bit interface. The `DATA_DIR_RD` macro is used to set the pins used for the 4-bit data bus as inputs, while the `DATA_DIR_WR` macro is used to set the same pins as outputs. The `OUTPUT_DATA(x)` macro is used to write data to the 4-bit data bus of the LCD interface. The `epulse()` function simply pulses the E signal line high; the `DelayUs(1)` software delay is pessimistic and can be removed if the PIC clock speed is slow enough so that one instruction cycle meets the 500 ns minimum E pulse width. The `lcd_write()` function writes 1 byte of data passed in `cmd` to the LCD, assuming a 4-bit interface. If `chk_busy` is nonzero, the busy flag is polled until it returns nonzero before performing the write. Observe that in the busy flag loop, the first read returns the upper 4 bits, while the second read returns the lower 4 bits. The busy flag is the MSb of the upper 4-bit transfer. If `chk_busy` is zero, a pessimistic delay of 10 ms is performed before writing the byte instead of using the busy flag. After some commands, such as the *function set* command, the busy flag cannot be used so a delay must be performed instead. If `data_flag` is nonzero, the RS signal is set to “1” during the write; else it is set to “0”. Finally, if `df1ag` is zero, only the upper 4 bits are written (the initial command that selects the 4-bit interface requires only a single 4-bit transfer as the LCD is in 8-bit mode on power-up).

```

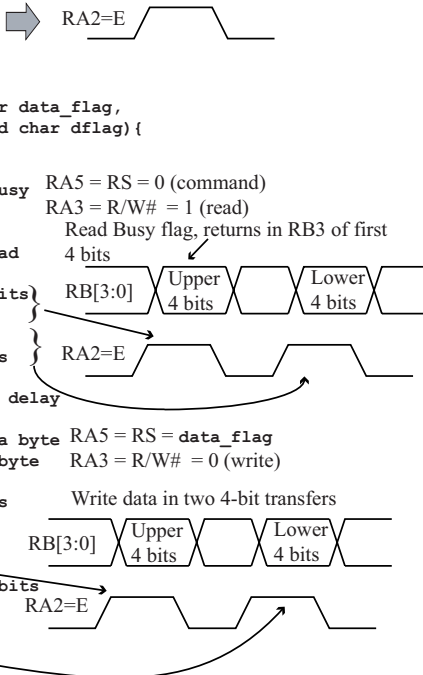
// macros to isolate interface dependencies
#define EHIGH      RA2=1
#define ELOW       RA2=0
#define E_OUTPUT   {ADCON1=0x06;TRISA2 = 0;}
#define RSHIGH     RA5=1
#define RSLow      RA5=0
#define RS_OUTPUT  TRISA5 = 0
#define RWHIGH     RA3=1
#define RWLOW      RA3=0
#define RW_OUTPUT  TRISA3 = 0
#define BUSY_FLAG  RB3
#define DATA_DIR_RD {TRISB3=1;TRISB2=1;\
                      TRISB1=1;TRISB0=1;}
#define DATA_DIR_WR {TRISB3=0;TRISB2=0;\
                      TRISB1=0;TRISB0=0;}
#define OUTPUT_DATA(x) {PORTB = x;}

void epulse(void){
    DelayUs(1); EHIGH; DelayUs(1);
    ELOW; DelayUs(1);
}

void lcd_write(
    unsigned char cmd, unsigned char data_flag,
    unsigned char chk_busy, unsigned char dflag){
    char bflag,c;
    if (chk_busy) {
        RSLow; //RS = 0 to check busy
        // check busy
        DATA_DIR_RD; //data pins inputs
        RWHIGH; // R/W = 1, for read
        do {
            EHIGH; DelayUs(1); //upper 4-bits
            bflag = BUSY_FLAG;
            ELOW; DelayUs(1);
            epulse(); // lower 4-bits
        } while(bflag);
    } else DelayMs(10); // no busy, do delay
    DATA_DIR_WR;
    if (data_flag) RSHIGH// RS=1, data byte
    else RSLow; // RS=0, command byte
    RWLOW; // R/W = 0, for write
    c = cmd >> 4; // send upper 4 bits
    OUTPUT_DATA(c);
    epulse();
    if (dflag) {
        c = cmd & 0x0F; //send lower 4 bits
        OUTPUT_DATA(c);
        epulse();
    }
}

```

Macros to isolate code from port pins used to implement LCD interface



**FIGURE 8.25** lcd\_write(), epulse() functions for the LCD interface.

The code in Figure 8.26 uses the lcd\_write() function within the putchar() function so that the formatted output function printf() can be used for writing strings to the LCD. The main() code first calls lcd\_init(), which initializes the display using the commands of Table 8.2. Observe that none of the lcd\_write() calls in lcd\_init() uses the busy flag for status checking; instead, the constant delay mode of lcd\_write() is used. After initialization, the address counter of the LCD is at location 0. The first printf() in main() writes to the first line of the LCD. Only the

```

void lcd_init(void) { ← Initialize the display
    DelayMs(50); //wait for device to reset on power-on, pessimistic
    lcd_write(0x20,0,0,0); // 4 bit interface
    lcd_write(0x28,0,0,1); // 2 line display, 5x7 font
    lcd_write(0x28,0,0,1); // repeat
    lcd_write(0x06,0,0,1); // enable display
    lcd_write(0x0C,0,0,1); // turn display on; cursor and blink is off
    lcd_write(0x01,0,0,1); // clear display, move cursor to home
    DelayMs(3); // wait for busy flag to be ready
}

// send 8 bit char to LCD
void putchar (char c) { } Define putchar() as a character write to the LCD
    lcd_write(c,1,1,1); } so that printf can be used for formatted output.
}

main(void) {
    // configure control pins as outputs
    // initialize as low
    E_OUTPUT; RS_OUTPUT; RW_OUTPUT; } Configure control, initialize low
    ELLOW; RLOW; RWLOW;
    lcd_init ();
    printf("*****Hello, my name is Bob*****"); } Write line 1,
    lcd_write(0xC0,0,1,1); // cursor to 2nd line } Set address counter to first
    printf("-----these lines are moving!-----"); } location of line 2,
    while(1) { write line 2
        // shift left
        lcd_write(0x18,0,1,1); } Loop continually left shifts,
        DelayMs(100); } causing lines 1 and 2 to scroll across
        DelayMs(100); } the display, moving right to left.
        DelayMs(100);
    }
}

```



**FIGURE 8.26** Write two strings to LCD and shift display (see CD-ROM file `./code/chap8/F_8_25_lcd_lines_4bit.c`).

first 16 characters of the `printf()` string are visible in the display, even though the entire string is stored in the LCD data display RAM. The following statement `lcd_write(0xC0,0,1,1)` sets the internal address counter to 0x40 (first position of second line), so that the next `printf` statement writes to the second line of the display. The 0xC0 byte in the `lcd_write()` function call is the *Set DD address* command, where 0xC0 = 0b1100000. The format of this command is 1nnnnnnn, where nnnnnnn is the data display address. Thus, the lower 7 bits of 0xC0 is 1000000, or 0x40, the address of the first location on the second line. An infinite loop is then entered in which the statement `lcd_write(0x18,0,1,1)` is followed by a 0.3-second delay. The 0x18 (0x00011000) command byte is the cursor & display shift command from Table 8.2 and has the format 0001cr00, with  $c = 1$ ,  $r = 0$  specifying a display left shift. The continual looping of this command causes the strings to scroll across the display moving right to left, with a 0.3-second delay between shifts. More sophisticated LCD modules allow graphical operations, such as turning on/off a pixel specified by a X,Y screen location.

## SUMMARY

---

Code written in a high-level language (HLL) such as C is usually clearer in its intent, has fewer source lines, and is more portable than code written in assembly language. As such, many microcontroller applications are written in a HLL rather than assembly. However, understanding assembly language and the implementation of HLL constructs in assembly language is critical in writing efficient HLL microcontroller applications. The HI-TECH software PICC-18 C compiler is used for the examples in this book and provides a powerful tool for experimenting with PIC18F242 applications. A simple PIC18F242 hardware system used to flash an LED only requires a power source, a clock source, and a reset switch. A PIC18Fxx2 has many different sources of reset, with status bits in the RCON register used to determine the reset source. Reducing power consumption is an issue in many microcontroller applications, and SLEEP mode in the PIC18Fxx2 can reduce current draw by a factor greater than 400 by stopping the internal clock. The watchdog timer runs on an independent clock source, and can be used to wake the PIC18Fxx2 from SLEEP mode and resume execution. The PIC18F242 has three parallel ports (PORTA, PORTB, PORTC) of varying capabilities, but all of them can implement bidirectional IO whose data direction is controlled by a corresponding data direction register (TRISA, TRISB, TRISC). An LCD module interface in 4-bit mode can be implemented using eight of the parallel port pins on the PIC18F242.

## REVIEW PROBLEMS

---

These problems assume access to a PIC18F242 system with the capabilities of the startup schematic of Figure 8.4.

1. The `-Mfile` option to the HI-TECH PICC-18 compiler produces a map file that specifies the memory locations of functions and variables. Compile the `ledflash.c` code with full optimization and give the locations of functions `main()` and `a_delay()`. Examine the compiled code for `a_delay()` and determine the memory location reserved for variable `i` (Hint: The auto variable `i` is not listed in the variable map, so import the hex file into MPLAB and look for the code that initializes `i` in the `for{} loop`).
2. Create a test program whose `main()` contains the code given in this problem. Compile this program with full optimization and without, and use the MPLAB StopWatch to determine the accuracy of the `DelayMs(1)` delay for a clock frequency of 40 MHz. (Hint: Find the code that implements the statement `PORT = ~PORTB` and put a breakpoint at this location.)



```

while(1){
    DelayMs(1);
    PORTB = ~PORTB;
}

```

3. Compile `ledflash.c` with and without optimization and compare the code (memory used for instructions) and data sizes (memory used for variables).
4. Modify the `a_delay()` function of the `ledflash.c` code to use `DelayMs()` to implement a delay that flashes the LED two times per second.
5. What is the maximum parameter value that can be used with the `DelayUs()` macro of Figure 8.6 for  $FOSC = 30 \text{ MHz}$ ?
6. In the schematic of Figure 8.4, erratic operation occurs if the connection from `VPP/MCLR#` pin to the LED and the pullup resistor is left off. Why is this?
7. Draw a schematic for a pushbutton switch that implements a high-true function; i.e., provides logic one when pressed and logic zero when not pressed. You can only use a resistor and a pushbutton.
8. Modify the code of Figure 8.11 to print messages that distinguish between WDT reset and WDT wake-up.
9. Modify the code of Figure 8.11 to provide a choice for software reset (the `RESET` instruction), and print out a message if this type of reset occurs. If this choice is selected, perform a software reset.
10. Modify the code of Figure 8.11 to provide a choice for stack overflow reset, and print out a message if this type of reset occurs. If this choice is selected, cause a stack overflow reset to occur.
11. From the PIC18Fxx2 datasheet, what is the typical  $I_{dd}$  current in HS mode for a crystal frequency of 10 MHz, and a  $V_{dd}$  of 5 V? (Hint: Look at the DC Graphs and Tables section.)
12. From the PIC18Fxx2 datasheet (see the `RESET` section) or Appendix A, what is the value of the bits in the Bank Select Register after a power-on reset?
13. For the LED/switch configuration of Figure 8.18, write code that blinks the LED once per second if the switch is pushed; else it blinks at twice per second.
14. For the LED/switch configuration of Figure 8.18, write code that repeats the following events: a press and release of the switch starts the LED blinking, a subsequent press and release stops the LED blinking.
15. Assume a low-true pushbutton input on `RB4`, and four high-true LEDs connected to pins `RB0` through `RB3`. Write C code that configures `PORTB` for this operation, with the LEDs initially off. Then, enter a loop, where each press and release of the switch turns the LEDs on in sequence, with the

LED previously on being turned off (e.g., 1<sup>st</sup> press/release, RB0 is 1/RB3 is 0, 2<sup>nd</sup> press/release RB1 is 1/RB0 is 0, 3<sup>rd</sup> press/release RB2 is 1/RB1 is 0, 4<sup>th</sup> press/release RB3 is 1/RB2 is 0, repeat).

16. What is needed external to pin RA4 if it is to provide a high voltage? Recall that pin RA4 is an open-drain output.
17. The HSPLL oscillator option multiplies the crystal frequency by 4 internally, while the HS option does not. If a PIC18 is drawing 16 mA using the HSPLL option, what is the expected current draw if the HS option is used with the same crystal?
18. Using an IDD versus FOSC graph in the PIC18 datasheet, compare expected versus computed IDD reduction for two different voltage/frequency pairs of your choosing.
19. Assume RA4 is configured as an output and the assignment `RA4 = 0` is executed. What does a read of RA4 return? What does a read of LA4 return? Now assume the assignment `RA4 = 1` is executed. What does a read of RA4 return? What does a read of LA4 return? Careful—pin RA4 is an open-drain output!
20. Discuss how a vertical scroll function might be implemented on the LCD module of Figure 8.23.

*This page intentionally left blank*

# 9

# Asynchronous Serial IO

## In This Chapter

- IO Channel Basics
- Synchronous Serial IO
- Asynchronous Serial IO
- The PIC18Fxx2 USART
- The RS232 Standard
- Serial IO Examples

This chapter introduces serial IO in the form of asynchronous serial transfer using the Universal Synchronous Asynchronous Receiver Transmitter (USART) subsystem of the PIC18Fxx2. A serial interface using RS232 signaling is implemented for the PIC18F242 reference system, allowing character input/output via a serial port connection to a personal computer.

## 9.1 LEARNING OBJECTIVES

---

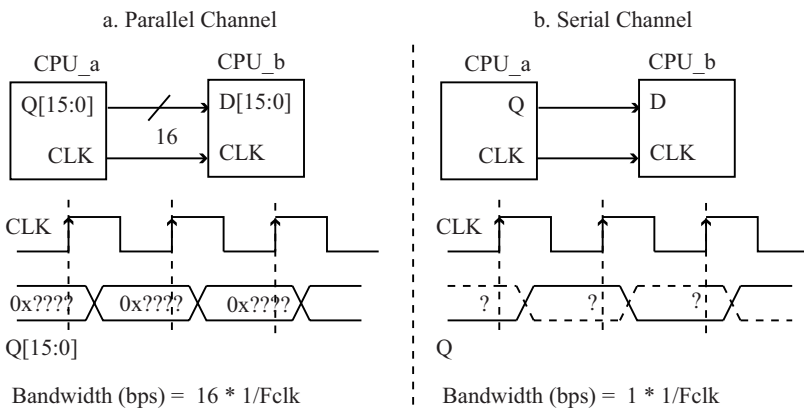
After reading this chapter, you will be able to:

- Describe the differences between synchronous and asynchronous serial data transfer.

- Draw the waveform for an asynchronous serial data transfer that includes a start bit, data bits, and stop bits.
- Write C code that sends and receives asynchronous serial data via the USART subsystem of the PIC18Fxx2.
- Implement a RS232-compatible interface for the PIC18F242.

## 9.2 IO CHANNEL BASICS

Parallel IO uses a group of signals for data transfer, with a clock or data strobe signal typically used for controlling the transfer. Figure 9.1 a shows a 16-bit parallel IO link between CPU\_a and CPU\_b, with a clock signal used to perform one data transfer each clock cycle. The *bandwidth* of a communication channel is usually expressed as the number of bytes transferred per second (B/s), or the number of bits transferred per second (b/s). Please observe the capitalization difference between Bps (Bytes/second) and bps (bits/second); Bps is related to bps via the relationship  $Bps = bps/8$ . For Figure 9.1a, the bandwidth is 600 MB/s ( $M = 10^6$ ) if the clock frequency is 300 MHz, because 2 bytes are transferred each clock cycle. Data sent 1 bit at a time is called *serial data transfer*, and Figure 9.1b shows a synchronous serial interface that uses a single bit line with a separate clock to accomplish the transfer. The bandwidth of this channel is  $1/16^{\text{th}}$  that of the bandwidth of Figure 9.1b, because the 16 data lines have been replaced by only one data line. The advantage of parallel IO is obvious in that it has  $n$  times the bandwidth of a serial channel, assuming both channels use the same data transfer rate, and the parallel channel has  $n$  data lines.



**FIGURE 9.1** Parallel versus serial IO example.

High bandwidth, low cost, and reliable transfers are the desirable properties of any IO channel. Unfortunately, these properties conflict with each other, as increasing bandwidth typically increases costs, and decreases the reliability of the IO transfer. Increasing IO channel bandwidth can be done by any combination of the following actions:

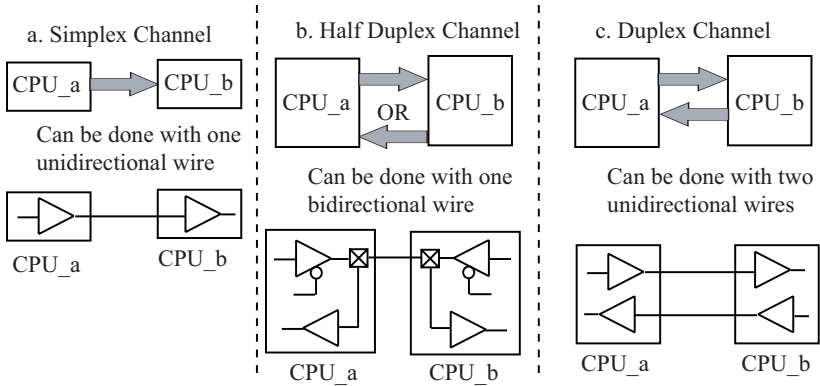
1. Increase the number of signals carrying data; e.g., increase a parallel IO channel from 8 bits to 16 bits.
2. Decrease the amount of time between data transfers; e.g., increase the clock speed of a parallel IO channel.
3. Use a signaling method that encodes more data in the same time interval; e.g., use a four-level voltage signaling method so that 2 bits are encoded in each signaling interval (00 = 0 V, 01 =  $1/3 V_{dd}$ , 10 =  $2/3 V_{dd}$ , 11 = V<sub>dd</sub>).

Methods 1 and 2 are the most common ways used to increase IO channel bandwidth. Doubling the number of signal lines in an IO channel doubles the bandwidth, but it also doubles the cost of the IO channel. Decreasing the time used for each transfer increases bandwidth, but also increases the complexity of the electronics used for driving and receiving data signals, increasing the cost of each data signal. One method of decreasing the time between transfers is to use reduced voltage swing on the data lines instead of requiring the data signals to transition fully between V<sub>dd</sub> and ground. Swinging a data line by 200 millivolts to indicate a change from “1” to “0” or vice versa is accomplished faster than requiring a signal to transition from 0 to V<sub>dd</sub>.

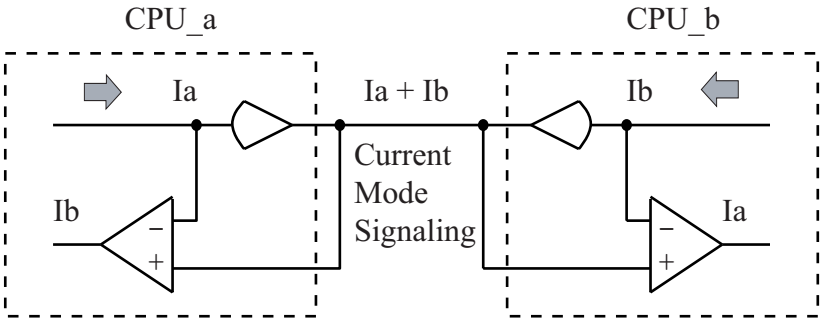
Figure 9.2 defines the terms *simplex*, *half-duplex*, and *duplex* in reference to communication channels. A simplex channel allows data transfer in one direction only. A half-duplex channel supports transfer in either direction, but in only one direction per data transfer. A duplex channel (also referred to as *full-duplex*) supports transfers in both directions simultaneously.

A physical connection between two systems is either a unidirectional wire (transfer in only one direction), or a bidirectional wire (transfer in either direction). In Chapter 8, “The PIC18Fxx2: System Startup and Parallel Port IO,” we saw that tristate buffers are required to implement a bidirectional port. A single unidirectional wire can implement a simplex channel, while a single bidirectional wire can implement a half-duplex channel. Two unidirectional wires can implement a duplex channel with each wire providing communication in one direction. An advanced electrical signaling method known as *current mode signaling* can be used to implement a duplex channel using a single wire. A simplified diagram of this concept is shown in Figure 9.3. Instead of driving the wire to a particular voltage level, data is represented by different *current* values. Current levels on a wire add

together, so if CPU\_a places  $I_a$  on the wire and CPU\_b places  $I_b$  on the wire, the total current on the wire is  $I_a + I_b$ . At each source, the amount of current placed on the wire is subtracted from the amount of current received. This means that CPU\_a sees the received current as  $I_b$ , while CPU\_b sees the received current as  $I_a$ . Current mode signaling is used in some chipsets for high-performance microprocessors; all of the devices discussed in this book use voltage mode signaling.



**FIGURE 9.2** Simplex, half-duplex, and duplex communication channels.



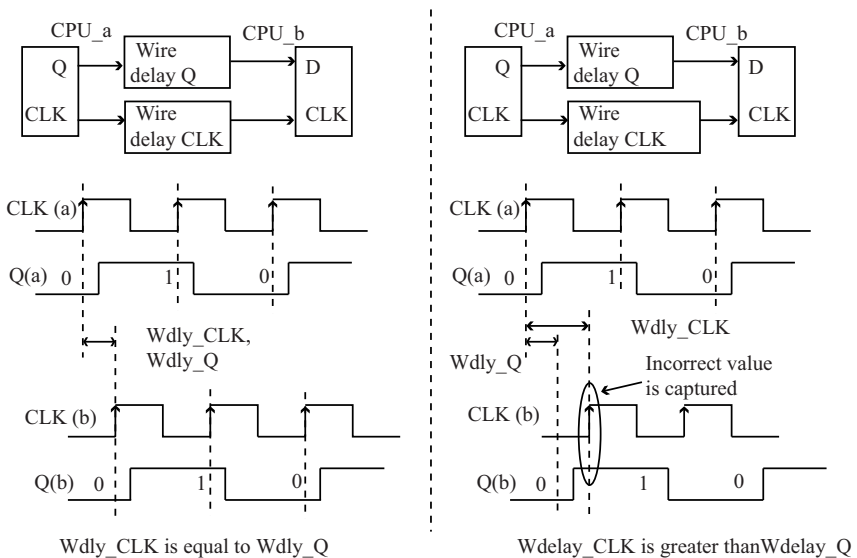
**FIGURE 9.3** Duplex communication using current-mode signaling.

The principle advantage of serial IO is that it is cheaper to implement in terms of integrated circuit and connector pin count than parallel IO. Serial IO is typically used for data transfer between devices that require external cabling, such as between a keyboard and a personal computer. This is because a serial cable requires fewer wires than a parallel cable, which reduces the cost. It also makes the cable less bulky and reduces the physical connector size, an issue when there are multiple IO

cable connections to a device. In addition, wires within an IO cable are subject to *crosstalk*, defined as a voltage change on a wire inducing a voltage change in a neighboring wire. Crosstalk can corrupt data transfers, resulting in an unreliable communication channel. Crosstalk increases with cable length and with higher signaling speeds. Methods for combating crosstalk increases the cabling costs, and thus serves as another reason for reducing the number of signals in an IO cable. Parallel IO is typically used for short distance communication between integrated circuits in the same system, where speed is important and cabling is not an issue.

### 9.3 SYNCHRONOUS SERIAL IO

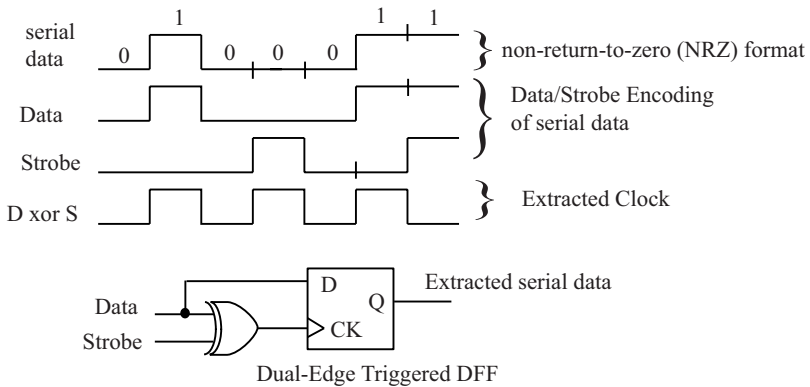
Synchronous serial IO either sends a clock as a separate signal as in Figure 9.1b, encodes the clock with the data, or uses a scheme that allows the receiving clock to remain synchronized to the bit stream. Sending a clock as a separate signal is an intuitive solution for synchronous IO, but at high signaling speeds, wire delay becomes significant especially in external cabling. If the wire delays of the serial data wire and clock signal are significantly mismatched, the active clock edge can clock in the wrong data bit at the receiver (see Figure 9.4).



**FIGURE 9.4** Synchronous serial IO: sending the clock with the data.



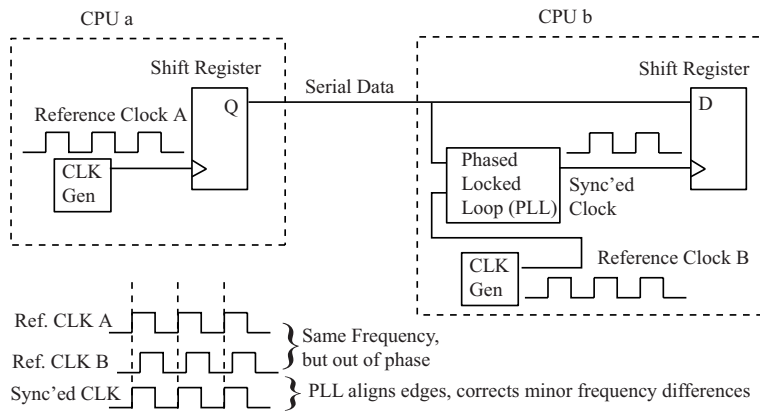
An alternate method of synchronous serial IO is to encode the clock in the data stream, so that the receiver can extract the clock. This method is used by the high-speed serial IO standard known as IEEE 1394, or FireWire, and Figure 9.5 shows how this is accomplished using *data strobe encoding*. The data line always contains the value of the serial data, while the strobe line encodes the clock. The value of the strobe line is dependent upon the data signal; between 2-bit intervals if the data signal does not change value, the strobe line changes value. If the data signal changes values, the strobe line remains at its current value. An XOR gate and a dual-edge triggered DFF is used to extract the serial data from the data/strobe signal pair as shown in Figure 9.5. This protocol can tolerate a wider range of wire delay differences than if the clock is sent as a separate signal, and the data rate can dynamically vary since the receiver extracts the clock. The serial data stream shown in Figure 9.5 is called *non-return-to-zero (NRZ)* format, which is the most intuitive method for representing serial data.



**FIGURE 9.5** Synchronous serial IO: encoding the clock with the data.

You may wonder at this point why such a complicated scheme is necessary. If the sender and receiver agree on the same clock rate, why does a clock need to be sent with the data? One problem with not sending a clock is that the sender and receiver generate their respective clocks independently, and even if the clock rates are perfectly matched there is no guarantee that the clocks are *in phase*, which means that their clock edges align. Moreover, it is not possible to generate perfectly matched clocks using different sources; some percentage mismatch in the clock frequency is always present. Even if the clock edges are somehow magically aligned initially, even very small clock frequency mismatches cause them to eventually become out of phase. Figure 9.6 shows how an analog circuit called a phase locked loop (PLL) is used to synchronize a receiver clock with an incoming serial bit

stream. Both systems use an independently generated reference clock that is the same frequency, within some percent tolerance mismatch. The PLL's function at the receiver is to produce a synchronized clock from the received serial bit stream and receiver reference clock, with the synchronized clock in phase with the sender's reference clock. To accomplish this, the serial bit stream must meet a minimum *transition density*; that is, there is a maximum time interval over which the bit stream's value can remain at the same voltage level. The PLL loses synchronization if the bit stream does not change voltage level for longer than this maximum time interval. The Universal Serial Bus (USB) and Controller Area Network (CAN) serial transmission standards use this synchronization method. Guaranteeing a particular transition density can be accomplished in multiple ways; Chapter 15, "Beyond the PIC18Fxx2," discusses this in more detail for the USB and CAN standards.



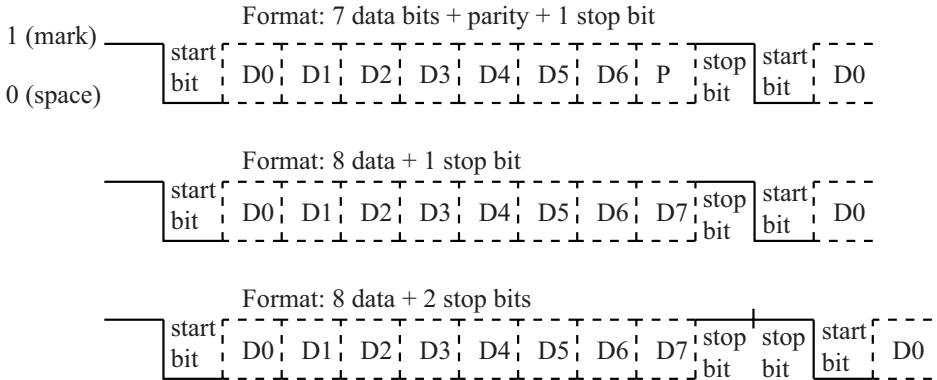
**FIGURE 9.6** Synchronous serial IO: maintaining synchronization with a PLL.

The form of synchronous data transmission supported by the USART subsystem of the PIC18Fxx2 is that of Figure 9.4, in which the clock is sent as a separate signal. Synchronous data transmission using the PIC18Fxx2 is covered in Chapter 11, "Synchronous Serial IO."

## 9.4 ASYNCHRONOUS SERIAL IO

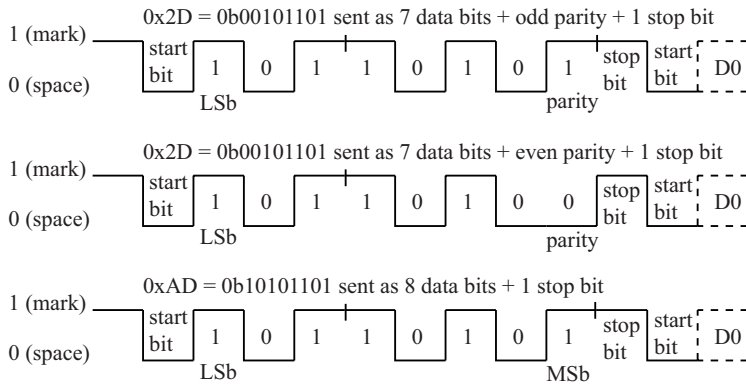
Asynchronous serial IO has none of the synchronization methods used by synchronous serial IO. Instead, the sender and receiver agree on a common data rate, and the sender sends its data to the receiver in NRZ format. This means that

asynchronous IO looks like Figure 9.6, except that the PLL is removed. The advantage of asynchronous serial IO over synchronous serial IO is that its hardware requirements are simpler, but asynchronous serial IO sacrifices bandwidth when compared with synchronous serial IO. Figure 9.7 shows an asynchronous serial data frame. Before transmission begins, the line is in the idle or *mark* condition, which is a logic “1”. The time required for sending 1 bit is referred to as a *bit time*. The start of transmission is indicated by a transition from the idle condition to logic “0”, known as the *space* condition. This first bit is called the *start* bit, and is how the receiver detects the beginning of a transmission. Data bits are sent least significant bit (LSb) to most significant bit (MSb), with common data formats being 7 data + even/odd parity or 8 data bits. The transmission ends with at least one *stop* bit, defined as the line remaining at “1” for at least 1 bit time. A total of 10 bit times are required for 1 start bit, 8 data bits, and 1 stop bit.



**FIGURE 9.7** Asynchronous data frame.

A *parity* bit is a bit added by the sender to provide error detection of single bit errors. Parity is either *even* or *odd*. The value of an even or odd parity bit is such that the total number of “1” bits in the *n* data bits + parity bit is even or odd, respectively. For example, if a 7-bit data field is 0b0101101, odd parity has a value of “1”, while an even parity bit is “0”. The receiver checks the value of the parity bit, and an incorrect parity indicates that some type of transmission error has occurred. A single parity bit is guaranteed to detect any single bit error; that is, only 1 bit of the *n* data bits + parity is received in error. If multiple bit errors occur, parity may or may not detect the error. Figure 9.8 gives three examples of asynchronous serial data frames.



**FIGURE 9.8** Example asynchronous serial data frames.

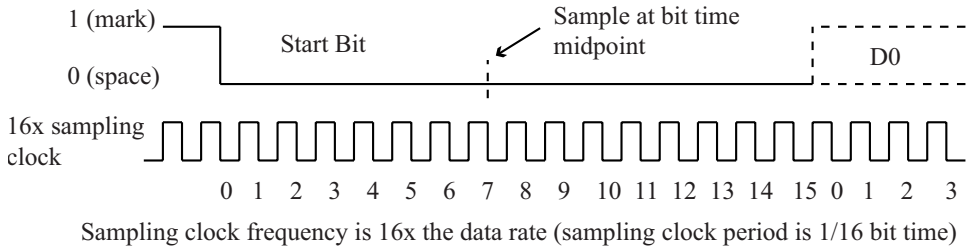
In Figure 9.7, each bit is sent within a *signaling interval*, with only 1 bit sent per signaling interval. This book refers to the signaling interval as a *bit time*, with the data rate of the channel in bits per second (bps) given in Equation 9.1.

$$\text{Data Rate (bps)} = \frac{1}{\text{bit\_time}} \quad 9.1$$

A term commonly used to refer to the data rate of an asynchronous channel is *baud rate*, whose definition is the number of signaling events per unit time. If only 1 bit is sent per signaling interval, baud rate is equal to bits per second. However, if more than 1 bit is sent per interval, such as a four-level voltage signaling scheme that sends 2 bits per signaling event, then bits per second is twice the baud rate. This book only discusses signaling methods in which 1 bit is sent per signaling interval, so *baud rate* is used interchangeably with *bit rate*.

Figure 9.9 shows how phase and frequency differences between receiver and sender clocks are handled. Given a data rate of  $y$  bps, the clock used by the sender/receiver for accessing the serial data is a multiple of this rate, usually 16x or 64x. Assuming a 16x clock, when the receiver detects a start bit (high to low transition), it counts 8 clock periods and then captures the input value. After this point, the receiver samples the input line every 16 clock periods, placing the sampling point near the center of the bit interval, giving the receiver maximum tolerance for mismatch between receiver and transmitter clocks. Over time, any receiver/transmitter clock mismatch shifts the sampling point away from the bit time midpoint, eventually causing a reception error when the sampling point is shifted out of a bit interval. Sampling at the midpoint of the bit time gives a 50% error margin, or  $\pm 8$  clock periods about the center of the clock interval. For a frame with 10 bit times, this gives a  $50\%/10 = 5\%$  error tolerance in sender/receiver clock mismatch. This is an optimistic mismatch assumption; a more pessimistic calculation accounting

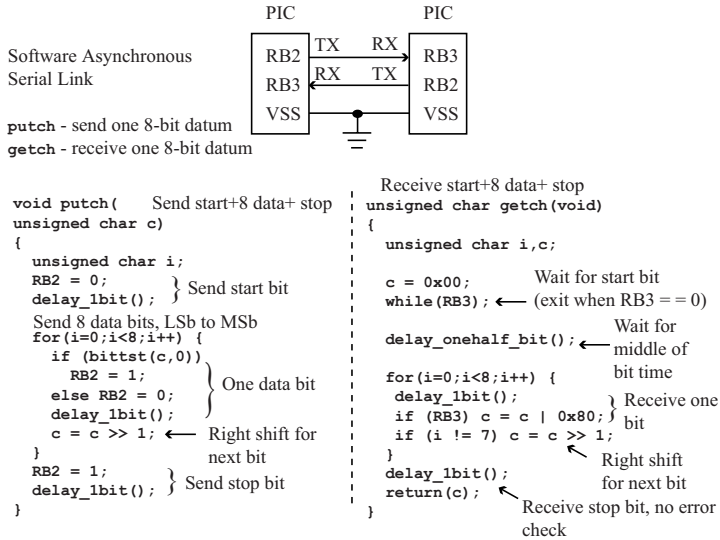
for rise and fall times of the input signal and maximum phase mismatch uses only a 30% margin (approximately  $\pm 5$  clock periods about the midpoint). This gives an error tolerance of  $30\%/10 = 3\%$  error tolerance for sender/receiver clock mismatch. Observe that the error tolerance decreases linearly as the number of bit times in the frame increases; for this reason, asynchronous transmission frames are typically limited to 10 bit times. This is not a problem for synchronous serial data transmission, which can send an unlimited number of bits per frame because the receiver remains synchronized to the serial input stream. Thus, even if asynchronous and synchronous serial channels have the same bit time, the synchronous channel has a higher effective data transfer rate because it does not have the overhead of the start and stop bits sent for every 8 data bits of the asynchronous transmission.



**FIGURE 9.9** Asynchronous serial data transmission.

Figure 9.10 shows a software-based asynchronous serial data link. Two PORTB pins, RB2 and RB3, implement transmit and receive, respectively, forming a duplex communication channel. The `putc()` function sends an 8-bit value serially from LSB to MSB using 1 stop bit. The `delay_1bit()` function is assumed to delay for 1 bit time. Observe that the `putc()` function assumes that RB2 is already in the idle (high) condition before sending the start bit. After sending the 8 data bits, the stop bit is sent, leaving the RB2 output in the idle condition. The `getch()` function returns an 8-bit value from the serial link by first waiting for a start bit (RB3 becomes a “0”). Once a start bit is detected, the function uses the `delay_onehalf_bit()` function to wait until the middle of the bit time. It then loops eight times, delaying a full bit time and then reading the RB3 input. The function delays for an additional full bit time before exiting to account for the stop bit. Software-driven serial links using parallel port bits can work well, but their maximum performance is limited by the accuracy of the delay functions used to implement bit delays. Also, even though the separate TX, RX lines of Figure 9.10 have the capability of implementing a duplex channel, the `putc()/getch()` functions as written cannot perform duplex communication. This is because all of the CPU’s resources are either spent transmitting or receiving a character; it cannot do both simultaneously using the functions of

Figure 9.10. One solution to this problem is to use dedicated hardware to implement the TX/RX functionality as seen in the next section.



**FIGURE 9.10** A software-based asynchronous serial data link.

**Sample Question:** For asynchronous serial transmission, with 1 bit sent per signaling interval, what is the bit time in microseconds for a baud rate of 57600?

**Answer:** From Equation 9.1, it is seen that:

$$\text{bit time} = 1/\text{baud\_rate} = 1/57600 = 1.74\text{e-}5 * 1\text{e}6 \mu\text{s/s} = 17.6 \mu\text{s}.$$

**Sample Question:** Assume a data format of 7 data bits + even parity. What is the parity bit value for the data 0x2A?

**Answer:** 0x2A = 0b0101010 (7 bits); the number of “1” bits is odd, so the parity bit value is “1”.

## 9.5 THE PIC18FXX2 USART

On less capable members of the PICmicro family, implementing a serial link using parallel port IO is the only available option. This approach becomes more difficult to implement at higher serial data rates, because the bit times become small, requiring more accuracy in the delay functions. Also, at higher data rates it requires

all of the CPU resources to implement the serial link. More powerful members of the PICmicro family, including the PIC18, have a hardware subsystem called the Universal Synchronous Asynchronous Receiver Transmitter (USART) that implements both asynchronous and synchronous data transmission. For asynchronous transmission, a write to a special function register is all that is required in terms of CPU resources; the USART handles the details of sending the start, data, and stop bits. For asynchronous reception, the USART automatically shifts in any serial data it receives, and sets a status flag indicating that data is ready. All the CPU has to do at that point is read a special function register to receive data.

The USART is the second major hardware subsystem, after parallel port IO, examined in this book. Hardware subsystems use special function registers in two different ways: either as *data registers* or as *control registers* for the subsystem. Data registers are either used for transferring data from the subsystem to the external pins (a write operation to an external device), or for transferring data from the external pins to the subsystem (a read operation from an external device). Control registers contain a mixture of *configuration* and *status* bits. Configuration bits specify the operating mode of the subsystem, while status bits indicate the operational state of the subsystem.

Table 9.1 gives a summary of the special function registers and bits used during asynchronous serial data transmission and receive. The TXIF (Transmit Interrupt Flag) bit indicates if the USART can accept new data for transmission; if TXIF = 1, a write to the TXREG causes that data to be shifted out of the TX pin of the USART subsystem.

**TABLE 9.1** SFR Data/Control for Asynchronous Transmit and Receive

Name	SFR(bit)	Type	Comment
TXREG	n/a	Data	Write to send async. serial data
RCREG	n/a	Data	Read to input async. serial data
TXIF	PIR1[4]	Status	if "1", can accept new data for transmit
RCIF	PIR1[5]	Status	if "1", has new data available
FERR	RCSTA[2]	Status	if "1", framing error occurred on receive
OERR	RCSTA[1]	Status	if "1", overrun error occurred on receive

The need for the TXIF status bit is seen Figure 9.11, which contains the USART transmit block diagram.

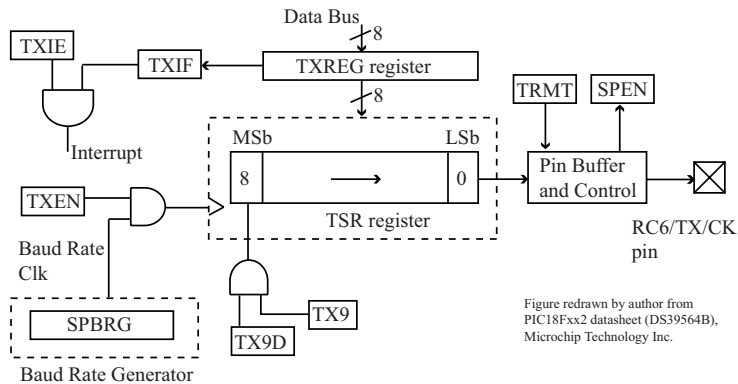


Figure redrawn by author from PIC18Fxx2 datasheet (DS39564B), Microchip Technology Inc.

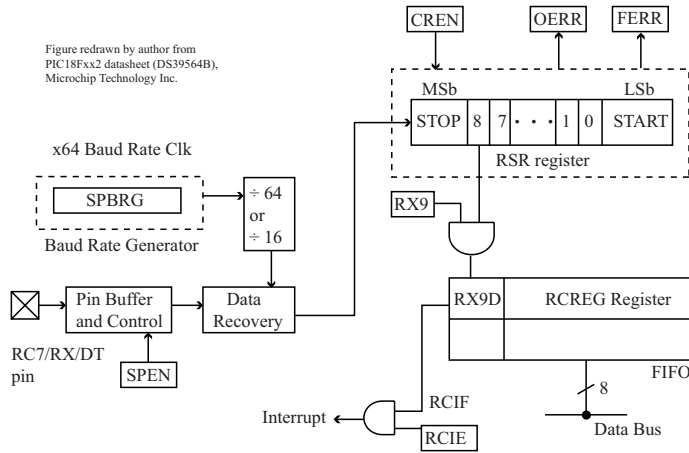
**FIGURE 9.11** USART transmitter block diagram.<sup>1</sup>

The TXREG register is a buffer register for the TSR register that performs the shifting of serial data through the TX pin. The TXIF bit is a “1” if TXREG is empty, thus the TXIF status bit must be checked before any write is done to TXREG, or data that is waiting to be sent may be lost. A write to the TXREG clears the TXIF bit to a “0” in the second instruction cycle after the write operation. The TXIF bit is a read-only bit; it is only cleared by a write to TXREG. Observe that if both TSR and TXREG are empty, a write to TXREG clears the TXIF bit, which is quickly set back to “1” once the TXREG contents are transferred to the empty TSR. At this point, a second value can be written to TXREG. Thus, two values can be written in quick succession to the USART transmit block if it is empty. However, a third write must wait for the serial shift operation in TSR to finish, causing TXREG to be emptied. Some USARTs on other microprocessors have up to 16 locations for buffering data in their transmit blocks. The PIC18 USART allows either 8 or 9 data bits to be transmitted; it does not support a 7-bit format with parity. Our examples always use 8-bit data, as that is the format supported by RS232 communication, a common form of asynchronous serial IO. An RS232 interface is discussed later in this chapter.

Figure 9.12 shows the USART receiver block diagram. Serial data enters via the RX pin and the receiver block automatically shifts the data into the RSR register upon detection of a start bit. The CREN (Continuous Receive Enable, RCSTA[4]) bit must be a “1” to enable the USART receiver block. The RSR contents are transferred to the RCREG after stop bit reception. This sets the RCIF (Receive Interrupt Flag) bit to “1” indicating available data in RCREG. The RCIF bit is cleared to “0” after a read from RCREG. A two-deep *first-in, first-out* (FIFO) buffer is used to implement RCREG, providing time flexibility for the main application in reading the

<sup>1</sup> Figure 9.11 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.’s prior written consent.





**FIGURE 9.12** USART receiver block diagram.<sup>2</sup>

RCREG contents. A FIFO buffer can hold multiple data items that are extracted from the buffer in the same order as written (Chapter 10 contains details on a software FIFO buffer implementation). If the RSR register and the RCREG FIFO are full (three characters total), the start bit detection of the fourth character triggers an *overflow* error, setting the OERR status bit to a “1”. The OERR bit is read-only; to clear it the CREN bit must be cleared to a “0”. When overflow occurs, no further values are transferred from RSR to RCREG until the OERR bit is cleared. A *framing* error occurs if a “0” is received for a stop bit, which sets the FERR status bit to a “1”. A separate FERR bit is maintained for each received 8-bit value. The FERR bit in RCSTA reflects the framing status for the value returned by the next read of RCREG. This means the FERR bit status must be checked before reading RCREG, as this updates FERR with the framing status for the next 8-bit datum in the FIFO.

Figure 9.13 shows the `putc()/getch()` functions of Figure 9.10 rewritten to use the PIC18 USART. The `putc()` function waits for the TXREG to become empty via the `while(!TXIF){}` loop that exits when TXIF is “1”, and then writes the input value `c` to the TXREG. The watchdog timer is cleared within this loop in the event it is enabled. The `getch()` function waits for data to be available via the `while(!RCIF){}` loop that exits when RCIF is “1”, and then returns the RCREG value. The watchdog timer is also cleared within this loop because it could be an arbitrarily long wait for input data. Obviously, the `c1rwdt` instructions are not needed in `putc()` and `getch()` if the watchdog timer is disabled. The C formatted IO functions `printf()` and `scanf()` supplied by the HI-TECH PICC-18 compiler use the `putc()` and `getch()` functions for single character IO. This provides a powerful method for performing ASCII data IO within PIC18 applications. Later in

<sup>2</sup> Figure 9.12 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.’s prior written consent.

```

void patch(unsigned char c)
{
    // wait until TXREG empty
    while (!TXIF){
        asm("clrwdt");
    };
    TXREG = c;
    asm("clrwdt");
}

unsigned char getch(void)
{
    // wait until data available
    while (!RCIF){
        // ok to wait forever for input
        // so clear watchdog timer
        asm("clrwdt");
    };
    return(RCREG);
}

```



**FIGURE 9.13** patch()/getch() functions for USART (see CD-ROM file ./code/common/serio.c).

this chapter we discuss the final details of using the PIC18 USART to form a serial data link to a personal computer.

The baud rate (BR) of the serial interface is controlled by the baud rate clock generator, and is calculated by Equation 9.2. The SPBRG register is an 8-bit register, meaning its value varies between 0 and 255. High- and low-speed modes are selected by the BRGH (High Baud Rate Select Bit, TXSTA(2)) configuration bit, with BRGH = 1 selecting high-speed mode.

$$BR = \frac{FOSC}{(S * (SPBRG + 1))} \quad S = 64 \text{ (low speed), } S = 16 \text{ (high speed)} \quad (9.2)$$

Equation 9.3 solves Equation 9.2 for SPBRG, because PIC18 applications need to know the SPBRG value required to achieve a particular baud rate.

$$SPBRG = \left[ \frac{FOSC}{(S * BR)} \right] - 1 \quad (\text{round to nearest integer}) \quad (9.3)$$

Figure 9.14 shows SPBRG values for common baud rates using FOSC values of 29.4912 MHz and 40 MHz. The SPBRG values are computed for both low- and high-speed modes using Equation 9.3, and rounded to the nearest integer. Because of this rounding, the actual achieved baud rate may be significantly different from the desired baud rate. Observe that for FOSC = 29.4912 MHz, the actual baud rate is exactly the desired baud rate, because the commonly used baud rates are all evenly divisible by powers of two into 29.4912 MHz. Differences between actual and desired baud rates appear for FOSC = 40 MHz, but in only a couple of cases does the percentage error exceed the conservative 3% error tolerance for asynchronous communication. Observe that in high-speed mode, some of the lower baud rates require SPBRG values greater than 255, which means that these baud rates are

unachievable in this mode. In low-speed mode for the FOSC = 40 MHz case, the SPBRG rounding causes unacceptable error for the two highest baud rates.

FOSC = 29.4912 MHz

Baud Rate	SPBRG (Hi Speed)	Actual	%err	SPBRG (Low Speed)	Actual	%err
230400	7	230400	0.0%	1	230400	0.0%
115200	15	115200	0.0%	3	115200	0.0%
57600	31	57600	0.0%	7	57600	0.0%
38400	47	38400	0.0%	11	38400	0.0%
19200	95	19200	0.0%	23	19200	0.0%
9600	191	9600	0.0%	47	9600	0.0%
4800	383	n/a		95	4800	0.0%

FOSC = 40 MHz

Baud Rate	SPBRG (Hi Speed)	Actual	%err	SPBRG (Low Speed)	Actual	%err
230400	10	227272.7	-1.4%	2	208333	-9.6%
115200	21	113636.4	-1.4%	4	125000	8.5%
57600	42	58139.53	0.9%	10	56818.2	-1.4%
38400	64	38461.54	0.2%	15	39062.5	1.7%
19200	129	19230.77	0.2%	32	18939.4	-1.4%
9600	259	n/a		64	9615.38	0.2%
4800	520	n/a		129	4807.69	0.2%

**FIGURE 9.14** SPBRG values for common baud rates.

The registers and bits involved in configuring the USART for asynchronous transmission and reception are given in Table 9.2. The special function registers involved are the SPBRG, TXSTA, and RXSTA registers.

**TABLE 9.2** Control Registers/Bits for Asynchronous Configuration

Name	SFR(bit)	Comment
SPBRG	n/a	This register contains the baud rate divisor
BRGH	TXSTA[2]	Baud rate control; if "1" high speed, else low speed
TX9	TXSTA[6]	If "1", 9-bit transmission, else 8-bit transmission
TXEN	TXSTA[5]	If "1", transmit is enabled, else is disabled
SYNC	TXSTA[4]	"0" for UART asynchronous mode
RX9	RCSTA[6]	If "1", 9-bit reception, else 8-bit reception
CREN	RCSTA[4]	If "1", receive is enabled, else is disabled
SPEN	RCSTA[7]	If "1", external pins selected for TX/RX function
TRISC6	TRISC[6]	Must be "0" so that RC6/TX/DT pin is an output
TRISC7	TRISC[7]	Must be "1" so that RC7/RX/CK pin is an input

Listing 9.1 gives a C function used in this book for initializing the USART. Two char parameters, `brg` and `hi_speed`, are used to specify the SPBRG values and high/low speed selection, respectively. Before exit, the CREN bit is cleared first and then set in case the USART had already been previously initialized; this ensures that the OERR status bit is cleared. In the example files provided on the companion CD-ROM, this function is in the file `./code/common/serial.c`, which is included via the statement `#include "serial.c"` in programs that use asynchronous serial IO. Two C compiler header files named `stdio.h`, and `ctype.h` define character IO and character manipulation functions; these standard header files are included in the PICC-18 C compiler installation. The `stdio.h` include file is necessary if the `printf()` and `scanf()` library functions are used.



**LISTING 9.1** USART initialization for asynchronous mode (see CD-ROM file `./code/common/serial.c`).

```
// standard header files for ascii IO functions,
// ascii manipulation functions
#include <stdio.h>
#include <ctype.h>

void serial_init(char brg, char hi_speed)
{
    // setup Async communication
    TX9 = 0;
    TXEN = 1;           // transmit enable
    SYNC = 0;          // async mode
    if (hi_speed) BRGH = 1; // hi speed mode
    else BRGH = 0;     // lo speed mode
    SPBRG = brg;       // set baud rate register
    bitset(TRISC, 7);  // RC7 input
    bitclr(TRISC, 6);  // RC6 output
    RX9 = 0;           // 8-bit reception
    SPEN = 1;          // serial port enable
    CREN = 0;          // clear enable first
    CREN = 1;          // now enable
}
```

**Sample Question:** What is the required SPBRG value assuming high-speed mode, a 12 MHz FOSC, and a desired baud rate of 19200?

**Answer:** Using Equation 9.2,  $SPBRG = (12.0e6 / (16 * 19200)) - 1 = 38.1$ ; so  $SPBRG = 38$ .

**Sample Question:** For the previous sample question, what is the percentage difference between the desired baud rate and the actual baud rate?

*Answer:* Using  $SPBRG = 38$ , the actual baud rate from Equation 9.2 is  $(12e6)/[16*(38+1)] = 19231$ .

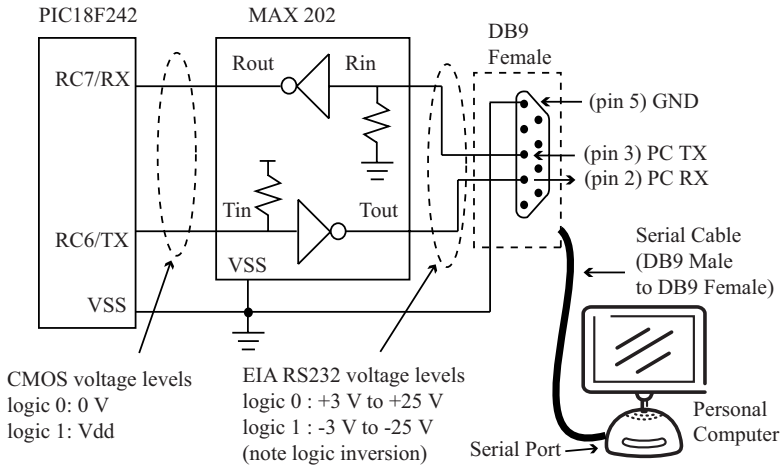
The %difference is  $(\text{actual}-\text{desired})/\text{desired}*100\%$ , so  $(19231-19200)/19200*100\% = 0.16\%$ .

## 9.6 THE RS232 STANDARD

---

The Electronic Industries Association standard EIA-RS232 defines signaling levels, external cabling, and handshaking protocols for asynchronous communication. RS232 cabling uses either a 9-pin connector (DB9) or a 25-pin connector (DB25), with the DB9 commonly used on personal computers. Figure 9.15 shows a minimal RS232 PIC-to-PC asynchronous serial connection. The MAX 202 RS232 transceiver from Maxim provides a conversion between RS232 logic levels and CMOS logic levels. A RS232 logic one has a range of  $-3\text{ V}$  to  $-25\text{ V}$  ( $-9\text{ V}$  typical), and a RS232 logic zero has a range of  $+3\text{ V}$  to  $+25\text{ V}$  ( $+9\text{ V}$  typical). This is a minimal connection, as there are many other signals defined in the RS232 standard that are used for external modem control and flow control. This minimal connection has no method for either device to signal the other device about its ability to receive data; it is assumed by the sender that the receiver is always able to accept new data. Two pins named CTS (Clear to Send, pin 8 on the DB9) and RTS (Request to Send, pin 7 on the DB9) can be used as handshaking lines to implement hardware flow control if desired. The RTS signal is an output from the PC (input to PIC), while the CTS signal is an output from the PIC (input to PC), and both signals are low true (logic 0 when asserted). Hardware flow control means that the PC will assert its RTS output, asking for permission to the send data. The PIC must assert the CTS line to tell the PC that it is ready to receive data; to stop the data flow from the PC, all that is necessary is for the PIC to negate the CTS output. Notice that this controls the flow of data in one direction only, from the PC to the PIC. This is because the RS232 standard was originally intended for communication between a terminal and a modem, designated as Data Terminal Equipment (DTE, the PC) and Data Communication Equipment (DCE, now represented by the PIC), respectively. In this historical model, the modem performed relatively low-speed communication over phone lines to a remote site. This means that the modem's input buffer could become full, requiring it to tell the DTE (the PC) to stop sending data. It was assumed that the DTE (the PC) was always ready to accept data from the modem (the DCE). On the PIC, implementing hardware flow control requires using two parallel port pins, which are precious resources on the 28-pin 18F242. For typical asyn-

chronous communication speeds, hardware flow control is not required and thus this book does not use it.



**FIGURE 9.15** Minimal RS232 PIC-to-PC connection.

A detailed view of the MAX202 is given in Figure 9.16. An on-chip voltage doubler and voltage inverter that uses external  $0.1 \mu\text{F}$  capacitors produces  $\pm 10 \text{ V}$  for use as RS232 voltage levels from the  $+5 \text{ V}$  supply. There are two pairs of transceivers in this package, which means that the RTS/CTS signals could be implemented in addition to the RX/TX signals if desired. Other RS232 transceivers from MAXIM such as the MAX3235E or MAX203 use internal capacitors for producing RS232 voltage levels.

A terminal program such as the *HyperTerminal* program included in Windows is required for RS232 communication via the minimal interface of Figure 9.15. The *HyperTerminal* configuration and terminal windows are shown in Figure 9.17. Personal computers have multiple serial ports, known as COM1, COM2, and so on. The number of serial ports and the correspondence of port numbers to external serial port connectors are manufacturer specific. When opening a *HyperTerminal* window, you must specify the particular COM port to use as well as the baud rate, data format, and flow control method. For the minimal RS232 PIC interface, a serial data format of 8 bits, no parity, 1 stop bit, and no flow control is required. Many personal computers, especially portable computers, no longer include an external serial port due to the emergence of the Universal Serial Bus (USB) as the new standard for serial communications (see Chapter 15). Fortunately, there exist USB-to-RS232 adapters that allow a USB port to be used as a RS232 serial port. Be aware

that some USB-to-RS232 adapters will not work with the minimal RS232 interface of Figure 9.15, as they require the additional RS232 handshaking signals.

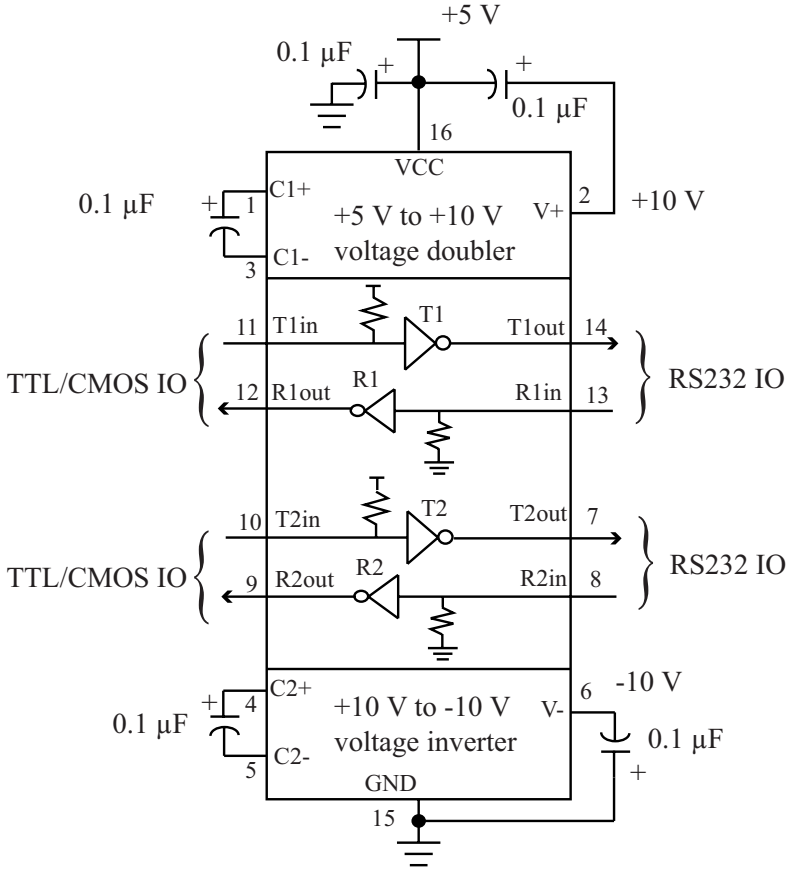
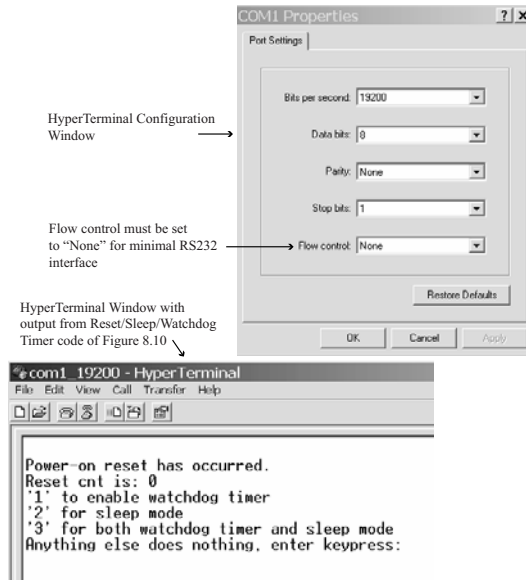


FIGURE 9.16 MAX202 detail.



**FIGURE 9.17** HyperTerminal configuration and terminal windows.

## 9.7 SERIAL IO EXAMPLES

Listing 9.2 contains a program used to test the minimal RS232 link of Figure 9.15 via an infinite loop that waits for a character to become available, increments the character, and then echoes it back. Thus, an "A" is echoed as "B", a "B" as "C", and so on. Incrementing the character is important, as this indicates that the PIC is performing the character echo, rather than some sort of local echo mode in whatever terminal program is being used for the test. The program of Listing 9.2 uses the `serial_init()` function of Listing 9.1, the `putch()`, `getch()` functions of Figure 9.13, and uses an SPBRG value calculated for a baud rate of 19200, high-speed mode, and  $FOSC = 29.4912$  MHz.



**LISTING 9.2** `echo.c` program for testing RS232 link.

```

main(void){
    unsigned char c;
    // init serial port
    // 19200 in HSPLL mode, crystal = 7.3728 MHz
    serial_init(95,1);
    while(1) {
        c = getch();    // wait for character
        c++;            // increment character
    }
}

```



```

        putchar(c);        // send it back
    }
}

```

Listing 9.3 attempts to automatically detect the baud rate via the `autobaud()` function before entering the character echo loop. The `autobaud()` function expects the user to set the baud rate by repeatedly typing the carriage return character (return key). The function uses a loop that inputs a character using a baud rate specified by a value from the `baud_rate` array. These baud rates are arranged from lowest to highest, with the lowest being 9600 and the highest 230400. The `baud_rate` array entries specify the SPBRG value for the target baud rate assuming  $F_{OSC} = 29.4912$  MHz and high-speed mode. The upper bits of the carriage return character (0x0D = 0b00001101) are read as “1”s if the PIC baud rate is set too low because the PC will have finished transmitting the character and returned the line to the idle state (“1”) while the PIC is still inputting bits at the slower baud rate. If the received character is not 0x0D, the UART is re-initialized to the next higher baud rate and `getch()` is called to receive another character. The loop is restarted at the lowest baud rate if either a framing or overrun error occurs.




---

**LISTING 9.3** *autobaud.c* program for automatically detecting baud rate.

---

```

void pcr1f (void){ // output a carriage return, line feed
    putchar(0x0a); putchar(0x0d);
}

// HSPLL mode, crystal = 7.3728 MHz
// so FOSC = 29,491,200
// baud rates: 9600, 19200,38400, 57600, 115200, 230400
char baud_rate[] = {191,95,47,31,15,7};

// enter carriage return repeatedly to set baud rate
void autobaud(void){
    char i, c, exit;

    i=0;
    serial_init(baud_rate[i],1);
    exit = 0;
    while(!exit) {
        // wait for character
        while (!RCIF);           // check RCIF bit
        c = RCREG;
        if (FERR || OERR) {
            // baud rate too fast, reset
            i = 0;
            CREN = 0;           // clear error
            CREN = 1;
        }
        if (c == 0x0D) {         // found baud rate

```

```

        printf("Found baud rate."); pcr1f();
        exit = 1;
    } else {
        i++;
        if (i > 5) i = 0;        // try again
        serial_init(baud_rate[i],1);
    }
}
}

// Try to automatically determine baud rate,
// then echo
main(void){
    char c;
    autobaud();
    while(1){
        // do character echo
        c = getch();
        c++;
        putchar(c);
    }
}

```

## Serial Port Debugging

An asynchronous serial port offers a versatile communication method for a stand-alone PIC18 system. However, implementing a functional serial port can be a frustrating experience, as there are many failure points. The most common error is swapping the TX and RX pins, or incorrectly identifying them on the DB9 connector. With a serial cable connected to the COM port of a personal computer, the voltage between the TX pin and GND is typically between  $-8\text{ V}$  and  $-10\text{ V}$ , which is a RS232 logic 1 (the mark condition). Because the RX pin is an input, any voltage measurement made on this pin can float, but will typically show  $0\text{ V}$ . Another common problem is leaving the ground (GND) pin unconnected. If the PIC system and the personal computer do not share a common ground, the serial port operation may be very erratic—sometimes appearing to work and sometimes not. Once you have correctly identified TX, RX, and GND on the DB9, debugging of the serial port is best done using the *echo* program of Listing 9.2 and an oscilloscope set for single-trigger mode that can capture the serial waveform of a character. The following step-by-step debugging process can be followed *in order* to trace serial port problems.

1. Verify that there is approximately  $\pm 10\text{ V}$  on the pins indicated in the MAX202 datasheet. This indicates that the capacitors are placed correctly.

2. Verify the correct HyperTerminal settings of 19200 baud, 8 bits + 1 stop bit + no parity, and no flow control. If garbled characters appear, this usually means that the baud rate is set to the wrong value.
3. Determine if the typed character is reaching the DB9 connector by placing the scope input on the TX pin of the DB9, and use single trigger mode, *rising edge* triggered. The voltage level on the TX pin of the DB9 should be approximately  $-8\text{ V}$  when the line is idle. Type a key on the keyboard; this should cause the scope to trigger and capture the serial waveform of the received character. If this does not happen, either the scope is configured wrong, or you have the TX pin identified incorrectly. Move the scope input to the Rin pin of the MAX202 and verify that the same character is being received at that point (if it is not, there is a wiring error between the DB9 TX pin and the MAX202).
4. Determine if the typed character is passing through the MAX202 by placing the scope input on the Rout pin of the MAX202, and use single trigger mode, *falling edge* triggered (note that this triggering is the reverse of the previous step because of the logic inversion performed by the MAX202). This voltage level should be at  $+5\text{ V}$  when the line is idle. Type a key on the keyboard; this should cause the scope to trigger and capture the serial waveform of the received character. If this does not happen, either the MAX202 is nonfunctional (check power/ground connections), or you have used a Rout pin that is not associated with the Rin pin that you verified in the previous step. If you capture a character at this point, place the scope input on the RX pin of the PIC and verify that the same character can be captured at the PIC (if no character is seen, there is a wiring error from the MAX202 to the PIC18).

At this point, you have verified that a character is reaching the PIC. The remaining steps trace the path from the PIC back to the DB9.

5. Determine if the received character is being transmitted by the PIC by placing the scope input on the PIC TX pin, and use single trigger mode, *falling edge* triggered. This voltage level should be at  $+5\text{ V}$  when the line is idle. Type a key on the keyboard; this should cause the scope to trigger and capture the serial waveform of the transmitted character. If this does not happen, the PIC is either not receiving the character in the first place (check the previous step), the PIC is programmed with the incorrect program (not *echo.c*), or the PIC is inoperative (check the clock waveform on the OSC1 pin, voltage level of the board). If you capture a serial character on the TX pin of the PIC, move the scope trace to the Tin pin of the MAX 202, and verify that the same character is being received at that pin.

6. Determine if the transmitted character is flowing through the MAX202 by placing the scope input on the Tout pin of the MAX202, and use single trigger mode, *rising edge* triggered. This voltage level should be at approximately  $-8$  V when the line is idle. Type a key on the keyboard; this should cause the scope to trigger and capture the serial waveform of the transmitted character. If this does not happen, the Tout pin being used is not associated with the Tin pin used in the previous step, or the MAX202 is nonfunctional. After verifying that a character is being transmitted, move the scope input to the RX pin on the DB9 and verify that the transmitted character is present at that point.

These steps should identify the problem if you have trouble implementing the minimal RS232 interface of Figure 9.15.

## **SUMMARY**

---

Data transfer between a processor and an external device can either be parallel (multiple bits at once) or serial (1 bit at a time). Parallel IO offers the highest bandwidth, but has the greatest cost in terms of pin count and mechanical difficulties in cabling. Serial IO is typically used for communication that requires cabling external to a microprocessor system. Synchronous serial IO means that the receiver remains synchronized to the bit stream. Receiver synchronization is maintained by either sending the clock as a separate signal, encoding the clock in the bit stream allowing the receiver to extract a clock, or guaranteeing a minimum transition density in the bit stream allowing the receiver to use a phase locked loop to maintain synchronization. Synchronous serial IO data streams can send an unlimited number of bits in a single transmission because the receiver remains synchronized to the bit stream. Asynchronous serial IO does not have any of these mechanisms, limiting transmitted data length to only a few bits, after which the line is returned to an idle state allowing the receiver to resynchronize at the start of the next transmission. A minimal duplex asynchronous link is implemented with three signals: transmit, receive, and ground. The PIC18 USART subsystem supports both asynchronous and synchronous serial data transmission. Serial data transmission is accomplished by writing the data to be sent into the TXREG after waiting for it to become empty by checking the TXIF status flag. Serial data is automatically input by the receive block of the USART with data availability indicated by the RCIF status bit. The RS232 standard provides signaling and connector specifications for implementing an asynchronous serial data interface. A minimal RS232 interface for a PIC18 consists of an integrated circuit for converting between CMOS and RS232 voltage levels, and the three-wire connection of receive, transmit, and ground. A personal com-

puter with an RS232 interface and a terminal program provides a versatile communication mechanism for a standalone PIC18 system.

## REVIEW PROBLEMS

---

1. What is the bandwidth in MB/s of a parallel data link that consists of 32 wires for data transfer, a clock speed of 8 MHz, and data transfer every second rising clock edge?
2. Assume a PIC18 with an FOSC of 40 MHz, and PORTB for data transfer. Suppose you want to transfer 256 bytes of data to an external device. How long does it take to transfer this data, using the code in this problem? Express this transfer rate in MB/s.

```

clr    cnt
lfsr   FSR0,data_array ;get address of data to transfer loop
movff  POSTINC0,PORTB  ; write data to PORTB
decfsz cnt              ;decrement counter
bra    loop             ;will loop 256 times
;;rest of code

```

3. What is 1 bit time in microseconds for a baud rate of 19200?
4. How long does it take to send 64 bytes of data at 57600 baud using asynchronous serial transmission assuming 8 data bits and 5 stop bit times between each character?
5. Give the maximum bandwidth of an asynchronous serial link operating at 115,200 baud in B/s assuming a format of 8-data bits and the minimum time of 1 stop bit between transmissions.
6. Draw the waveform for an asynchronous transmission assuming 8 data bits, 1 stop bit for the data value 0xA0.
7. Draw the waveform for an asynchronous transmission assuming 8 data bits, 1 stop bit for the data value 0x38.
8. Modify the `getch()` code of Figure 9.13 to check for USART overrun immediately after function entry. If USART overrun has occurred, execute a software reset via inline assembly code.
9. What is the parity bit for the 7-bit value 0x38 assuming even parity?
10. Change the `putch()` code of Figure 9.13 so that it replaces the MSb of the data to be sent with an even parity bit. (Hint: What value does the sequential exclusive-OR of the bits in the lower 7 bits give you?)
11. For an asynchronous serial transfer, assume a 16x clock on the receive side, and a data format of 16 data bits + 1 stop + 1 start. Using a conservative error tolerance of  $\pm 5$  clocks about the midpoint of the bit time, what is the

maximum % tolerance in frequency mismatch between the sender and receiver?

12. Assume a baud rate of 38400 and compute the maximum time before overrun given the USART receiver block diagram of Figure 9.12. Express this time in instruction cycles, assuming four clocks per instruction cycle and  $FOSC = 40 \text{ MHz}$ .
13. For  $FOSC = 6 \text{ MHz}$  and assuming high-speed mode, give the baud rates of Figure 9.14 that cannot be supported either because they do not fall in the 8-bit range of SPBRG or exceed 3% error (use a spreadsheet for these calculations).
14. For  $FOSC = 6 \text{ MHz}$  and assuming low-speed mode, give the baud rates of Figure 9.14 that cannot be supported either because they do not fall in the 8-bit range of SPBRG or exceed 3% error (use a spreadsheet for these calculations).
15. When would you expect a framing error to be more probable; if the sender's baud rate was higher than the receiver's, or vice versa? Explain.
16. Assume that the sender baud rate is higher than the receiver baud rate; under what conditions could a framing error occur?
17. It seems inconvenient that RS232 voltage levels are different from CMOS voltage levels. Offer a solid engineering reason as to why this was done (relate your answer to cost, performance, reliability, or functionality). Use the datasheet of the MAX202 to help your argument.
18. Look up the specifications for the EIA RS422 standard. What is the principle difference between this standard and the RS232 standard? What are its advantages, if any?
19. What changes are necessary to the `autobaud()` function of Listing 9.3 if the baud rates are searched from highest to lowest? Would you expect it to function equally well? Why or why not?
20. Change the `getch()` code of Figure 9.13 so that it assumes that the MSb of the received data is an odd parity bit and checks the received character for a parity error. Set a global variable `PERR` to "1" if a parity error is detected. (Hint: For computing parity, what value does the sequential exclusive-OR of the bits in the lower 7 bits give you?)

*This page intentionally left blank*

# 10

## Interrupts and a First Look at Timers

### In This Chapter

- Interrupt Basics
- PIC18 Interrupt Details
- Interrupt-Driven Asynchronous Serial Data Input
- Using a Software FIFO with Interrupt-Driven IO
- Other Interrupt Sources, Sleep Mode
- State Machine Programming for Interrupt-Driven IO
- The Timer Subsystem: Timer2
- Switch Debouncing Using a Timer
- A Rotary Encoder Interface
- A Numeric Keypad Interface
- On Writing and Debugging ISRs

This chapter discusses interrupts, which are of critical importance when implementing efficient input/output operations for microcontroller applications. Topics include interrupt fundamentals, PIC18 interrupt sources, and software techniques for implementing interrupt-driven IO. A first look at the powerful timer subsystem of the PIC18 uses a timer as a periodic interrupt source.

### 10.1 LEARNING OBJECTIVES

---

After reading this chapter, you will be able to:

- Discuss the general function of interrupts within a microprocessor, and interrupt implementation on the PIC18.
- Describe the difference between polled IO and interrupt-driven IO.



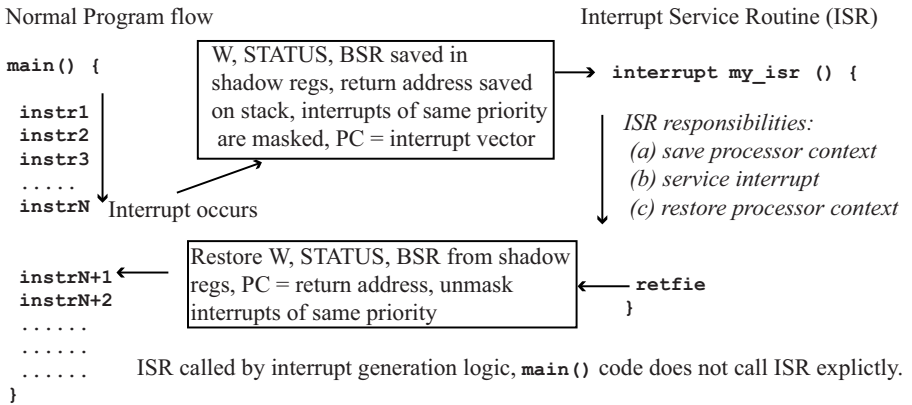
- Implement an interrupt service routine in C for the PIC18.
- Write C functions for performing interrupt-driven asynchronous serial IO.
- Implement software FIFO buffering for interrupt-driven IO.
- Implement an interrupt service routine using a state machine approach.
- Discuss the structure of the PIC18 Timer2 subsystem and use it to generate periodic interrupts.
- Implement an interrupt-driven interface for a rotary encoder.
- Implement an interrupt-driven interface for a keypad.

## 10.2 INTERRUPT BASICS

---

An *interrupt* in a microprocessor is a forced deviation from normal program flow by an external or internal event. On the PIC18 are many possible internal and external events such as rising or falling edges on external pins, arrival of serial data, timer expiration, and so forth that can cause interrupts. Figure 10.1 illustrates what happens when an interrupt occurs on the PIC18. During normal program flow, assume some external or internal event triggers an interrupt. After the current instruction is finished, the BSR, W, and STATUS registers are saved in the shadow registers, the return address is pushed on the stack, and the PC is set to a predetermined location called the *interrupt vector* causing execution to continue at that point. A `retfie` (return from interrupt) instruction is executed to return to normal program flow. The code that is executed when the interrupt occurs is referred to as the *interrupt service routine* (ISR). The ISR's function is to respond to whatever event triggered the interrupt. As an example, if the interrupt was triggered by the arrival of asynchronous serial data, the ISR would read the USART RCREG, save this data, and return. When viewing Figure 10.1, it is tempting to think of the ISR as a subroutine that is called by the `main()` program. However, the ISR is never manually called as a normal C function is called; instead, the ISR is invoked *automatically* by the PIC18 interrupt hardware on an interrupt occurrence. An ISR is said to execute in the *background*, while the normal program flow executes in the *foreground*. This book informally refers to background code as ISR code execution, and foreground code as `main()` code execution.

You may question at this point why this capability is needed. The IO examples presented in the last chapter used a technique referred to as *polling*, where a status flag is checked repeatedly to determine data availability. This is referred to as *polled IO*, and is usually an inefficient method for implementing IO operations. Imagine if your cell phone operated on the polled IO principle. This would mean that you would occasionally have to pull it out of your pocket or purse, open it, and ask “Hello, is there anybody there?” This may seem laughable, but this is how we have



**FIGURE 10.1** Interrupting normal program flow.

been accomplishing IO to this point. The problem with this approach is obvious—either you check your phone too often, which wastes your time, or you do not check it often enough, causing you to miss an important call. It is much more efficient to have the phone notify you of an incoming call. The ringer on your cell phone implements an interrupt; when the ringer sounds, you stop what you are doing and answer the phone, thus servicing the interrupt. This is known as *interrupt-driven IO*. On the PIC18, each interrupt source has an associated *interrupt flag bit* that becomes a “1” when the interrupt source event occurs. As an example, the RCIF bit is the receive character interrupt flag, and becomes a “1” when asynchronous serial data is available. Most interrupt flag bits are contained in two special function registers named PIR1 (peripheral interrupt request flag register 1) and PIR2 (peripheral interrupt request flag register 2).

Continuing the cell phone analogy, there are times when you do not want to answer the phone, like in a meeting or a movie theatre. At these times, you turn off the ringer, causing incoming calls to be ignored. On the PIC18, each interrupt source has an *interrupt enable bit* that must be a “1” in order for an interrupt to be invoked when the interrupt flag bit becomes a “1”. If the interrupt enable bit is “0”, the interrupt is *masked* or *disabled*. For example, the RCIE bit (receive character interrupt enable, PIE1[5]) is the interrupt enable for the RCIF interrupt. Most interrupt enable bits are in two special function registers, PIE1 (peripheral interrupt enable register 1) and PIE2 (peripheral interrupt enable register 2). It is important to understand that the interrupt enable bit being a “0” does not prevent the interrupt flag bit from becoming a “1”, just like turning off the phone ringer does not prevent incoming phone calls from arriving. A “0” interrupt enable bit only prevents an interrupt from being generated; in other words, it prevents a jump to the ISR.

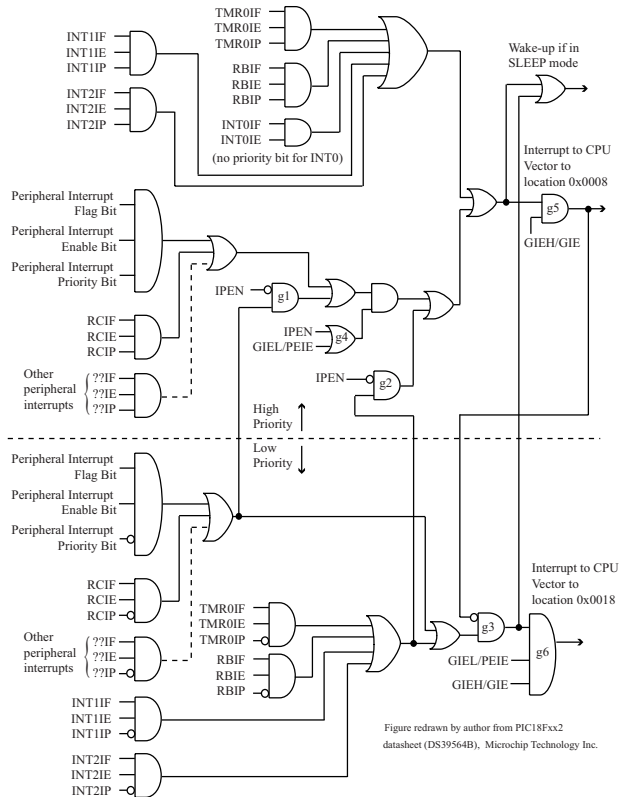
At the risk of overusing the cell phone analogy, assume you are talking on the phone with a friend, and that you are notified via call-waiting of an incoming call from your spouse, significant other, or other family member. You may say something like “Hold for a moment, there is an incoming call that I need to check,” and then switch to the other call. This means that the incoming call has a *higher priority* than the current call. On the PIC18, each interrupt source has an *interrupt priority bit* that when “0”, classifies the interrupt as a low priority interrupt, while a “1” classifies the interrupt as high priority. A high priority interrupt is allowed to interrupt a low priority interrupt, but not vice versa. Interrupt priorities are covered in more detail in the next section. The RCIP bit (receive character interrupt priority, IPR1[5]) is the interrupt priority bit for the RCIF interrupt. Most interrupt priority bits are contained in two special function registers named IPR1 (peripheral interrupt priority register 1) and IPR2 (peripheral interrupt priority register 2).

### 10.3 PIC18 INTERRUPT DETAILS

---

Figure 10.2 gives a logic diagram of PIC18 interrupt generation. If the IPEN bit (interrupt priority enable bit, RCON[7]) is a “1”, interrupt priorities are enabled and an interrupt is classified as either high priority or low priority, based on the setting of its interrupt priority bit. High priority interrupts vector to location 0x0008, and low priority interrupts vector to location 0x0018. The GIE/GIEH bit (global interrupt enable, INTCON[7]) enables all unmasked interrupts if its value is “1”, or masks all enabled interrupts if its value is “0”. The GIE/GIEH bit functions the same regardless of the IPEN bit setting. With priorities enabled (IPEN = 1), the PEIE/GIEL bit (peripheral interrupt enable, INTCON[6]) enables (“1”) or disables (“0”) all low priority interrupts. For a low priority interrupt to occur, the GIE/GIEH, PEIE/GIEL, interrupt flag, and associated interrupt enable bits must all be “1”. For a high priority interrupt to occur, the GIE/GIEH, interrupt flag, and associated interrupt enable bits must all be “1”. The PEIE/GIEL bit is automatically cleared before entering the low priority ISR; this prevents a low priority interrupt from being recognized during the execution of the ISR. However, a high priority interrupt can interrupt a low priority ISR. Because *any* interrupt pushes the return address on the return address stack and saves BSR, W, and STATUS in the shadow registers, a low priority ISR must save these registers in different temporary locations, as a high priority interrupt will overwrite the shadow register contents. Because of this, the shadow registers in practice are only useful for the high priority ISR. The GIE/GIEH bit is automatically cleared before entering the high priority ISR, preventing any interrupt from being recognized within the high priority ISR. A `retfie` instruction restores GIE/GIEH to a “1” on return from a high priority interrupt, or PEIE/GIEL to a “1” on return from a low priority interrupt. This re-

enables interrupts once normal program execution is resumed. The ISR should also clear the particular interrupt flag that caused the interrupt or disable the interrupt until the interrupt flag is cleared at a later time. In Figure 10.2, observe that interrupt INT0 has no priority bit, and is always classified as a high priority interrupt.



**FIGURE 10.2** PIC18 interrupt generation.<sup>1</sup>

If the IPEN bit (interrupt priority enable bit, RCON[7]) is a “0”, interrupt priorities are disabled and all interrupts are classified as high priority interrupts; the value of the individual interrupt priority bits have no effect. In this mode, setting the PEIE/GIEL bit to a “0” disables a subset of interrupts known as *peripheral interrupts*, which are those interrupts whose interrupt enable bits are contained in the PIE1 and PIE2 special function registers. The GIE/GIEH, interrupt flag, and associated interrupt enable bits must be “1” for a nonperipheral interrupt to occur; in addition to these bits the PEIE/GIEL bit must also be a “1” for a peripheral interrupt to occur. Having priorities disabled is the default mode of operation and is

<sup>1</sup> Figure 10.2 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.’s prior written consent.

compatible with previous PICmicro families. The applications covered in this book do not require interrupt priorities, and thus all C source examples that use interrupts disable the priority mechanism (`IPEN = 0`).

The responsibilities of an ISR can be summarized as:

1. **Save the processor context.** Any registers that are used by the ISR and may be used during normal program execution must be saved. The W, BSR, and STATUS registers are saved in the shadow registers for any interrupt. However, if the ISR modifies other registers such as FSRx or PRODH/PRODL that may be in use by the code being interrupted, these must be saved as well. As discussed previously, low priority interrupts should save the W, BSR, and STATUS registers in temporary locations because a high priority interrupt will overwrite the shadow register contents.
2. **Service the interrupt.** Perform the actions required when the interrupt occurs. If multiple interrupts are enabled, identify the interrupt source by checking the interrupt flag bits of the enabled interrupts. The associated interrupt flag must be cleared either directly or as a side effect of servicing the interrupt. If this is not done, the processor will hang in an infinite loop upon execution of the `retfie` instruction, as the interrupt flag being a “1” will immediately generate another interrupt. In some cases, it is not possible to clear the interrupt flag immediately, as it may not be possible to remove the interrupt source until a later time. In this situation, the ISR should disable the interrupt until the interrupt flag can be reliably cleared.
3. **Restore the processor context** and execute a `retfie` instruction to return to foreground code execution.

Figure 10.3 shows the assembly language structure for high and low priority interrupt service routines. Jumps in the form of `goto` instructions to the appropriate ISRs are placed at the vector locations `0x0008` and `0x0018`. The low priority ISR uses temporary locations `w_temp`, `status_temp`, and `bsr_temp` for storing the W, STATUS, and BSR registers, respectively, as the shadow registers may be corrupted by a high priority interrupt. Observe that the W register is stored first, using the `movwf` instruction, which does not affect the status flags. Thus, the next `movff` instruction that saves STATUS is saving the status flags as they were at the time of the interrupt. The STATUS register is restored last on exit, because the previous instruction `movf w_temp,w` that restores W affects the status register. The `retfie 1` instruction in the high priority ISR restores W, BSR, and STATUS from the shadow registers, while the `retfie 0` instruction used in the low priority ISR explicitly avoids restoring these registers from the shadow registers (the 0/1 argument to the `retfie` instruction controls restoring of the shadow registers).

```

CBLOCK 0x7D
    w_temp, status_temp, bsr_temp } Temporary storage
                                } for W, STATUS, BSR
ENDC

ORG 0x008
goto isr_high_priority } High priority interrupt vector
ORG 0x0018
goto isr_low_priority  } Low priority interrupt vector

ORG 0x????
isr_high_priority      High priority interrupt service routine
    ;; ISR high priority code
    ;; ...code goes here...
    retfie 1           Restore from shadow registers
                       ; use shadow reg

isr_low_priority       Low priority interrupt service routine
    movwf w_temp
    movff STATUS, status_temp } Save W, STATUS, BSR in temporary
    movff BSR, bsr_temp      } locations
    ;;...ISR CODE ...
    ;; ...code goes here...
    movff bsr_temp, bsr
    movf  w_temp, w           } Restore W, STATUS, BSR from temporary
    movff status_temp, STATUS } locations
    retfie 0
    
```

**FIGURE 10.3** ISR assembly language structure.

**Sample Question: Using Figure 10.2, why does  $GIE = 0$  disable all interrupts?**

*Answer:* When  $GIE = 0$ , the output of gates  $g_5$  and  $g_6$  are both forced low regardless of the other inputs, thus preventing any interrupt from being generated.

## 10.4 INTERRUPT-DRIVEN ASYNCHRONOUS SERIAL DATA INPUT

Figure 10.4 shows the *echo.c* program of the previous chapter rewritten to use interrupt-driven IO for asynchronous data reception. This example only illustrates the mechanics of using interrupts, as interrupt-driven IO is not required for the task of character echo.

The interrupt qualifier is used to identify the `pic_isr()` function as an ISR to the HI-TECH PICC-18 compiler. The compiler assumes a high priority interrupt by default; the additional qualifier `low_priority` used after `interrupt` would indicate a low priority interrupt. The `pic_isr()` function checks the RCIF bit to determine if asynchronous data reception triggered the interrupt; if this bit is set, the RCREG value is read and saved in the variable `received_char`. Checking the RCIF flag is not actually necessary if this is the only interrupt source that is enabled. The variable `got_char_flag` is then given a value of “1”, notifying `main()` that the ISR has placed valid data in the `received_char` variable. Recall that reading RCREG has the side effect of clearing the RCIF flag, which is needed to prevent an interrupt from being triggered on ISR exit due to RCIF still being a “1”. Typically, an ISR must

notify the foreground code that its action has occurred; hence the use of the `got_char_flag` variable. Any variable such as `got_char_flag` that is used by the ISR (background) to communicate with `main()` code (foreground) is called a *semaphore*. The setting of the semaphore by the ISR signals the `main()` code that the ISR has performed some action. The foreground code's resetting of the semaphore is an acknowledgment to the ISR that the semaphore has been recognized. In some cases, it may be required that the ISR not raise a new semaphore until it detects that the previous semaphore has been acknowledged.

```

volatile unsigned int got_char_flag;
volatile unsigned char received_char;

// interrupt service routine
void interrupt pic_isr(void)
{
    // see if this interrupt was
    // generated by a receive character
    if (RCIF) {
        // reading RCREG clears interrupt bit
        received_char = RCREG;
        got_char_flag = 1;
    }
}

main(void) {
    unsigned char c;

    // init serial port
    // 19200 in HSPLL mode, crystal = 7.3728 MHz
    serial_init(95,1);

    // enable interrupts
    IPEN = 0; // priorities disabled
    RCIE = 1; // receive interrupt enable
    PEIE = 1; // peripheral interrupts enabled
    GIE = 1; // global interrupts enabled
    while(1) {
        // wait for interrupt
        while (!got_char_flag);
        c = received_char;
        got_char_flag = 0; // clear flag
        c++; // increment char
        putchar(c); // send the char
    }
}

```

Use `volatile` qualifier for any variables modified within ISR; notifies compiler that variable can be modified between accesses.

`interrupt` qualifier for function notifies the compiler that this function is an ISR (high priority assumed).

If receive character interrupt, read character, save it, set flag to signal `main()` that interrupt occurred

Enable async receive character interrupt

Wait until ISR reads character, then read it and clear the ISR semaphore `got_char_flag`



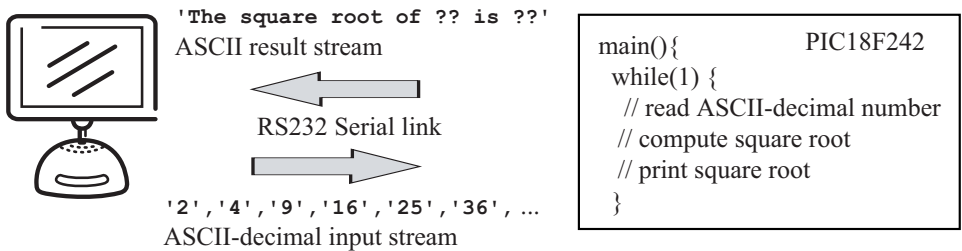
**FIGURE 10.4** *echo.c* rewritten to use interrupt-driven receive.

The `while(1){}` loop in `main()` waits for data to become available by monitoring `got_char_flag`; once this is nonzero, the value in `received_char` is incremented and echoed using the `putchar()` function. Observe that the `volatile` modifier is used in the variable declaration of the `got_char_flag` and `received_char` variables. This notifies the compiler that these variables can be modified by an external agent (e.g.,

an ISR) between successive accesses (reads), and prevents certain compiler optimizations from being applied.

Figure 10.5 gives an example where interrupt-driven IO is useful for improving an application’s operation. In this example, the PIC18 application is a loop that reads an ASCII-decimal number from the asynchronous serial port, computes the floating-point square root, and prints the result. Polled IO for serial data input works for this application, as long as there is enough time to calculate the square root and print the result before overrun occurs in the USART receive block.

While the result is printing, more data is being sent, and the data is lost unless buffered.



**FIGURE 10.5** Square root application.

Figure 10.6 gives an implementation of the square root application using polled IO and with no buffering other than what is available within the USART receive block. The ASCII-decimal number is input from the USART receive port using the `scanf()` library function, which requires a function named `getche()` that returns a single character and also echoes the character to the console. This is accomplished by calling `getch()` to read the character from the serial port, and then using `putch()` for character echo. The `%d` format code used with `scanf()` converts the ASCII-decimal string read from the serial port to a `int` data type, which is placed in the variable `ivalue`. The `scanf()` function expects a carriage return to mark the end of the string that it is scanning for input values. The assignment statement `temp_fp = ivalue` causes the compiler to generate code that converts `ivalue` to a floating-point value because `temp_fp` is declared as type `double`. The square root is calculated by the statement `root_fp = sqrt(temp_fp)`, where `sqrt()` is a C library call that requires a `double` as the type of its input parameter and uses a `double` for the type of its return value. The `printf()` statement displays the result via the serial port output.



```

#include "math.h"      ← need for sqrt() library function
unsigned char getch (void){
    // check RCIF bit
    while (!RCIF);
    return(RCREG);
}

// needed by scanf library call,
// get character and echo
unsigned char getche (void){
    unsigned char c;
    c = getch();
    putchar(c);
    return(c);
}

unsigned int ivalue, root;
double temp_fp, root_fp;

main(void){
    // 19200 in HSPLL mode, crystal = 7.3728 MHz
    serial_init(95,1);
    printf("No buffering."); putchar();
    printf("Hit any key to start..."); putchar();
    getch();
    while(1) {
        // read integer for input using scanf  Read ASCII-decimal integer from
        scanf("%d",&ivalue); ← async serial input using scanf ()
        // convert integer to floating point value
        temp_fp = ivalue; ← Convert to floating point type (double)
        // use library routine to compute floating point square root
        root_fp = sqrt(temp_fp); ← Compute square root using
                                   library function.
        //convert to nearest integer as demo compiler does not support
        // floats in printf() statements.
        root = (unsigned int)root_fp; ← Print result.
        printf("Square root of %d is: %d ",ivalue,root); putchar();
        DelayMs(5); // tuneable delay for USART overrun
    } ← Include tuneable delay to help force USART overrun as some terminal programs
        have a considerable idle time between characters, lowering the effective data rate.
}

```



**FIGURE 10.6** Square root code with polled IO and no buffering.

This program functions as expected when entering input values by manually typing characters into the terminal window. However, if one cuts and pastes a large number of entries into the terminal window, overrun error occurs within the USART because too many input characters arrive while the result string is being printed. Once USART overrun occurs, the RSR input shift register of the USART receiver block no longer transfers data into RCREG. This causes scanf () to hang, waiting for new input as seen in Figure 10.7. The next section discusses an approach for solving this problem. The number of input entries required to generate USART overrun is dependent upon the terminal program used to send serial data and the baud rate of the serial link. The DelayMs(5) function call is a tuneable software delay that is included to assist in forcing USART overrun.

```

No buffering.
Hit any key to start...
Square root of 2 is: 1
Square root of 4 is: 2
Square root of 9 is: 3
Square root of 16 is: 4
25 ← scanf() library function hangs when USART overrun
      occurs because data is no longer reaching RCREG

```

Cut and paste these values  
into terminal  
window to simulate  
continuous input stream

2
4
9
16
25
36
49
64
81
100
...

**FIGURE 10.7** Terminal output for square root code with polled IO and no buffering.

## 10.5 USING A SOFTWARE FIFO WITH INTERRUPT-DRIVEN IO

USART overrun occurs in the square root application because characters are arriving while the result is being printed. If the PIC18 USART had a larger hardware FIFO buffer, it might be possible to avoid overrun; it would depend on the number of consecutive input entries sent and the baud rate. It is obvious that changing the USART receive hardware is not an option. Instead, a FIFO buffer implemented in software must supplement the USART hardware buffering. Figure 10.8 gives the structure of a software FIFO with eight locations. Two pointers, named *head* and *tail*, are used for accessing the buffer. The FIFO is empty when head is equal to tail. Data is placed into the buffer by incrementing the head pointer, and then storing data at `buffer[head]`. This means that data is available in the buffer whenever head is not equal to tail. Data is taken out of the buffer by incrementing tail, and then reading data from `buffer[tail]`. Observe that data comes out of the buffer in the same order in which it is placed into the buffer; hence the first-in, first-out (FIFO) designation.

Figure 10.9 illustrates several pieces of data being placed into the buffer. When the head pointer reaches the end of the buffer, the next write operation must wrap the head pointer to the beginning of the buffer. For this reason, this data structure is also referred to as a *circular buffer*. In Figure 10.9c, the write operation leaves the head pointer equal to the tail pointer, causing the buffer to appear empty even though it contains eight valid data items. This is *buffer overrun*, which means that under these rules for buffer insertion and extraction, an *n* location buffer can hold a maximum of *n-1* data elements.

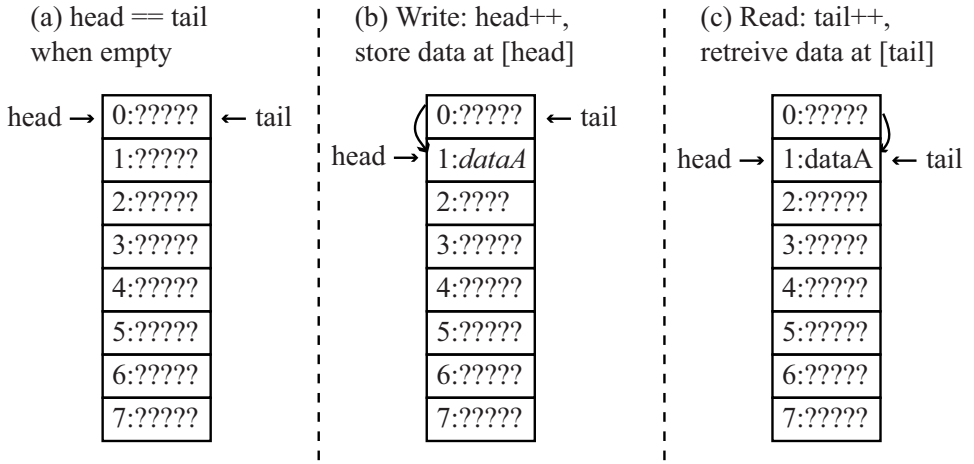


FIGURE 10.8 Software FIFO structure.

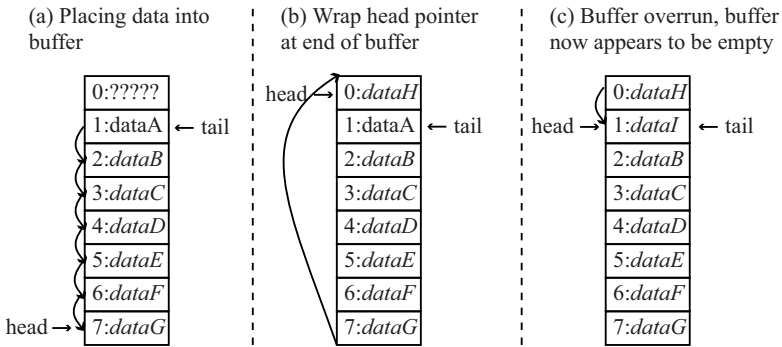


FIGURE 10.9 Software FIFO overrun.

Figure 10.10 shows the square root application modified to use interrupt-driven receive and a software FIFO for holding incoming data. The buffer size is set by `#define BUFMAX 32`, or 32 bytes. Variables of type `char` are used for the buffer (`iuchar[BUFMAX]`), head (`head`), and tail (`tail`) pointers. Within the ISR, if serial data has been received, `head` is incremented and wrapped to zero if it is equal to `BUFMAX`; then `iobuf[head] = RCEG` saves the data in the buffer. The `getch()` subroutine now waits for data availability via the `while(head == tail){}` loop, which exits when `head` is not equal to `tail`, indicating that the ISR has placed data into the buffer. Data is extracted from the buffer by incrementing `tail` and wrapping to zero if necessary,

and then data is read from `ibuf[tail]`. The only modification required for `main()` is initialization code that enables the RCIF interrupt.

```

#define BUFMAX 32
// head points to last character received
volatile unsigned char ibuf[BUFMAX], head, tail; } Variables added to support
software FIFO

unsigned char getch (void){
    unsigned char c;
    while (head == tail); ← Wait for data to be inserted into buffer by ISR
    tail = tail + 1;
    if (tail == BUFMAX) tail = 0; } Extract character from buffer using tail pointer
    c = ibuf[tail]; and return
    return(c);
}

// interrupt service routine
void interrupt pic_isr(void) {
    // see if this interrupt was
    //generated by a receive character
    if (RCIF) {
        head = head + 1;
        if (head == BUFMAX) head = 0; } ISR places data in buffer using the head pointer
        ibuf[head] = RCREG;
    }
}
unsigned ival, root;      putchar(), getch(), printf() functions not shown as they
double temp_fp, root_fp; are unchanged.

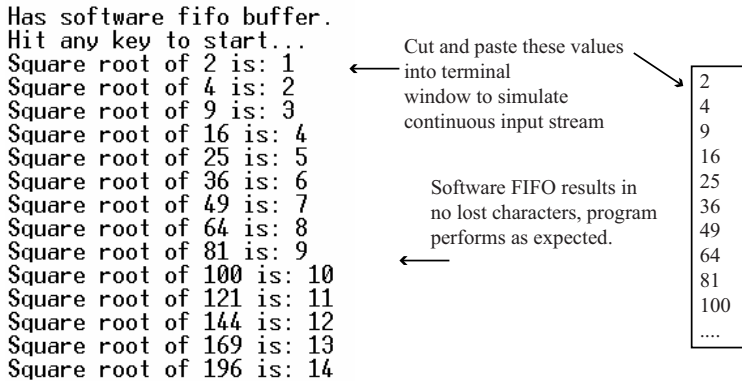
main(void) {
    // 19200 in HSPLL mode, crystal = 7.3728 MHz
    serial_init(95,1);
    // enable received character interrupt } Enable interrupt on
    IPEN = 0; RCIE = 1; PEEIE = 1; GIE = 1; } async. serial data receive
    printf("Has software fifo buffer."); printf();
    printf("Hit any key to start..."); printf();
    getch();
    while(1) {
        // read integer for input using scanf ← while(1) {} body is
        scanf("%d",&ival); unchanged; use of interrupt
        // convert integer to floating point value receive and software FIFO is
        temp_fp = ival; transparent.
        // use library routine to compute floating point square root
        root_fp = sqrt(temp_fp);
        root = (unsigned int)root_fp; // integer square root
        printf("Square root of %d is: %d ",ival,root); printf();
        DelayMs(5); // tuneable delay for USART overrun
    }
}
    
```



**FIGURE 10.10** Square root code with interrupt-driven receive and software FIFO.

Figure 10.11 shows a screenshot of the terminal output for the square root code of Figure 10.10. The application behaves as expected for the values that are cut and pasted into the terminal window. This does not mean that buffer overrun will never occur. Because each result output stream has more characters in it than the input stream, eventually overrun will occur if enough input values are continuously sent to the application. The software FIFO and interrupt-driven receive has delayed the occurrence of overrun, but not prevented it. Increasing the buffer size delays the occurrence of overrun even longer, but still does not prevent it as long as the outgoing stream (formatted result string) requires more bandwidth than the incoming stream (ASCII-decimal input). A software FIFO and interrupt-driven IO prevents

overflow for limited bursts of input data, where the FIFO buffer size is based on the expected worst-case input data burst.



**FIGURE 10.11** Terminal output for modified square root code.

**Sample Question:** Where would an overflow check of the software FIFO be placed in the code?

*Answer:* The software FIFO overflows if head becomes equal to tail after it is incremented. The buffer overflow check would be placed in the ISR because this is where head is modified when data is placed into the buffer.

### Interrupt-Driven Asynchronous Transmit

At this point, a natural question is, “Can asynchronous data transmit be interrupt-driven as well?” The answer is “yes,” and the changes required to `putch()` and the ISR are shown in Figure 10.12. For interrupt-driven transmit, `putch()` places data into the buffer, while the ISR extracts data from the buffer. The variables added to support the transmit software FIFO are `txbuf[BUFMAX]` (the buffer), `txhead` (the transmit head pointer), and `txtail` (the transmit tail pointer). After data is placed into `txbuf` by `putch()`, the transmit interrupt is enabled by the statement `TXIE = 1`. This is different from how the receive interrupt enable is handled; the `RCIE` bit is set to “1” in the initialization portion of `main()` and then left always enabled. Because the `TXIF` flag is a “1” whenever the `TXREG` is empty, we cannot leave the transmit interrupt always enabled, or else interrupts are continually generated.

```

#define BUFMAX 32
volatile unsigned char txbuf[BUFMAX]; } Variables added to support
volatile unsigned char txhead, txtail; } transmit software FIFO

void putchar (unsigned char c)
{
    char tmp;

    // must use tmp because we do not want ISR thinking
    // that buffer is empty
    tmp = txhead;
    tmp++;
    if (tmp == BUFMAX) tmp = 0;
    // wait until buffer space is freed
    while(tmp == txtail); ← wait for available space in buffer
    txbuf[tmp] = c;
    txhead = tmp; } place data in transmit software FIFO,
    // enable interrupt } enable transmit interrupt
    TXIE = 1;
}

void interrupt pic_isr(void)
{
    // receive interrupt
    if (RCIF) {
        head = head + 1;
        if (head == BUFMAX) head = 0;
        ibuf[head] = RCREG;
    } } Receive interrupt code, places data into
    // transmit interrupt } receive software FIFO
    if (TXIF) { //check TXIF bit
        if (txtail == txhead) {
            //buffer empty, disable interrupts } Transmit software FIFO empty,
            TXIE = 0; } so disable transmit interrupt
        } else {
            // get character
            txtail = txtail + 1;
            if (txtail == BUFMAX) txtail = 0;
            TXREG = txbuf[txtail];
        } } Extract data from transmit software FIFO,
    } } write to TXREG
}
}

```



**FIGURE 10.12** Interrupt-driven asynchronous data transmission.

Looking more closely at the `putchar()` code, observe that the `txhead` variable is first copied to the temporary variable `tmp`, which is then incremented and compared to `txtail`. If `tmp` is equal to `txtail`, the transmit software buffer is full, so the statement `while(tmp == txtail){}` loops until the ISR removes at least one character from the transmit buffer. The `tmp` variable is used instead of `txhead` to prevent the ISR code from falsely believing that the transmit buffer is empty, which is true when `txhead == txtail`. Once room exists in the transmit buffer, the `tmp` variable is copied to `txhead`, and data is placed at `txbuf[txhead]`. Within the ISR, it is imperative that both flag bits, `TXIF` and `RCIF`, be checked to discover which interrupt actually requires servicing now that two interrupt sources are enabled. If the `TXIF` flag is set, the ISR uses the condition `txhead == txtail` to determine if there is data in the transmit buffer. If this condition is true, the buffer is empty, and further transmit interrupts are disabled by the statement `TXIE = 0`. If the buffer has data, a char-

acter is extracted from the buffer using the `txtail` pointer and written to TXREG, which has the side effect of clearing the TXIF flag. The motivation for using interrupt-driven transmit is to recover the wasted instruction cycles that the CPU spends waiting for the TXREG to become empty. The size of the transmit software buffer should be large enough to accommodate the most common output data bursts without having to wait for free space in the buffer. However, the `getch()` function checks for transmit buffer overrun and waits if necessary, so no data is lost if the outgoing data requirements exceed the buffer size.

## 10.6 OTHER INTERRUPT SOURCES, SLEEP MODE

---

Many PIC18 hardware subsystems generate interrupts, and these are covered in the particular chapter that discusses the subsystem (e.g., Chapter 13 discusses use of the Timer1/Timer3 subsystems, so Timer1/Timer3 interrupts are covered in that chapter). In Figure 10.2, the three interrupts labeled INT0IF, INT1IF, INT2IF are associated with pins RB0/INT0, RB1/INT1, and RB2/INT2, respectively. These are called *external interrupts*, and can be configured via corresponding INTEDGx bits in the INTCON2 register to generate an interrupt on either a rising (INTEDGx = 1) or falling (INTEDGx = 0) input edge. The RBIF interrupt of Figure 10.2 is an *interrupt-on-change* feature associated with any bits RB[7:4] that are configured as inputs. The values of these pins are compared against the values latched into these bits on the last read, and any difference generates an interrupt if it is enabled. The RBIF flag remains set as long as the mismatch remains. To reset the RBIF flag, a read or write of PORTB must be done (except the MOVFF instruction) followed by a clear of the RBIF flag. An example use of the RBIF interrupt is presented in Section 10.11 using a keypad interface. The TMR0IF interrupt is covered in Chapter 13, “Timers,” which discusses the various timers within the PIC18.

In Chapter 8, “The PIC18Fxx2: System Startup and Parallel Port IO,” the watchdog timer was used to wake the processor from sleep mode, which is a standby mode useful for power conservation. There are many ways to wake the processor from sleep mode, all them having to do with actions on external pins or with internal subsystems that function on a clock source that is different from the main clock, like the watchdog timer (see the PIC18 datasheet for a complete list of actions that can wake the processor from sleep mode). In the square root application, any time spent waiting for serial data input is wasted, as there is no work to be performed if there is no input. Assume the processor is put into sleep mode if no input characters arrive within the watchdog timer interval. Is there a way for the USART subsystem to wake the processor upon arrival of data on the asynchronous serial input port? Unfortunately, the answer is “no,” as the receive block of the USART is inactive during sleep mode, because it operates from the main processor

clock that is stopped during sleep. Thus, the receive block cannot shift in a character during sleep. However, the RX input can also be tied to the RB0/INT0 input, and the falling edge of an arriving start bit used to wake the processor. Unfortunately, the arriving character is garbled as the USART wakes up after the falling edge of the start bit. The next character will be read correctly, as long as the RX input is idle long enough between the first character and the second characters to allow the USART to synchronize on the start bit of the second character. The amount of time needed for the PIC18 to begin operating after wakeup depends on the oscillator mode used. If an external clock is used (EC, ECIO clock modes), wakeup is almost immediate as there is no oscillator circuit startup time. The worst-case condition for wakeup time from sleep mode occurs if an external crystal is used (HS/PLL, HS, XT, LP modes), as the oscillator circuit is turned off during sleep. The crystal oscillator startup time can be significant [7], up to 120 ms depending on the crystal type used and oscillator mode. The sender must use at least one leading whitespace character in front of an ASCII-decimal number to serve as the wakeup character in case the receiver has gone to sleep after the last input value and delay the sending of the second character until the wakeup time has elapsed.

Figure 10.13 shows the square root application of Figure 10.10 modified to support this wakeup scheme. The code now has `asm("clrwdt")` placed in strategic places such as `putc()` and the RCIF code for the ISR to avoid watchdog timeout during normal operation. However, the wait loop `while(head == tail){}` in `getch()` specifically does not include an `asm("clrwdt")`, as we want to generate a WDT reset if new characters do not arrive within a timeout period. In `main()`, pin RB0 is configured as an input, and `INTEDG0 = 0` configures INT0IF to be set on a falling edge arrival. The test `TO == 0` is true if the WDT has expired, which indicates no input characters have arrived within a watchdog timer interval, and the processor should enter sleep mode. The watchdog timer is disabled (`SWDTEN = 0`) and the INT0IF interrupt is enabled (`INT0IE = 1`) before sleeping, as we only want a start bit arrival to wake the processor, not the watchdog timer. After being awakened by a falling edge on RB0 (start bit arrival), the asynchronous serial receive is re-enabled (`CREN = 1`), the watchdog timer enabled (`SWDTEN = 1`), and the `ignore_flag` variable is set indicating to the ISR that this input character should be ignored as it will be corrupted. If `main()` is entered by any type of reset other than watchdog timeout, the welcome message is displayed and the watchdog timer enabled.



```

void putch (
unsigned char c)
{
asm("clrwdt");
while (!TXIF) {
asm("clrwdt");
}
TXREG = c;
}

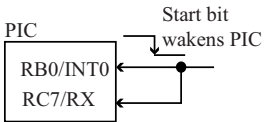
unsigned char getch (void)
{
unsigned char c;
// now wait for character
// WDT will cause timeout
// if no characters arrive
while (head == tail);
tail = tail + 1;
if (tail == BUFMAX)
tail = 0;
c = ibuf[tail];
return(c);
}

volatile ignore_flag;

void interrupt pic_isr(void)
{
if (INTOIF) {
// successfully woken
// the PIC, so disable,
// and clear flag
INTOIF = 0;
INTOIE = 0;
}
if (RCIF) {
asm("clrwdt");
if (!ignore_flag) {
head = head + 1;
if (head == BUFMAX)
head = 0;
ibuf[head] = RCREG;
} else ignore_flag = 0;
}
}

main(void) {
// set RB0 for input,
// falling edge interrupt
TRISB0 = 1;
INTEDG0 = 0;
// 19200 in HSPLL mode,
// crystal = 7.3728 MHz
serial_init(95,1);
// enable interrupts
IPEN = 0;
RCIE = 1;
PEIE = 1;
GIE = 1;
if (TO == 0) {
//WDT timer went off
//waiting for input
// enable RB0 interrupt
// before falling asleep
printf("Sleeping...");pcrlf();
SWDTEN = 0; // disable watchdog timer
INTOIF = 0;
INTOIE = 1; // enable RB0 interrupt
CREN = 0; // disable receive
asm("sleep");
CREN = 1;
ignore_flag = 1; // ignore this character
SWDTEN = 1; // enable watchdog timer
printf("Awake!");pcrlf();
} else {
// any other reset, print start message
SWDTEN = 1; // enable watchdog timer
pcrlf();
printf(
"Software RX FIFO buf + WDT on input.");
pcrlf();
printf("Hit any key to start...");pcrlf();
getch();
}
while(1) {
// read integer for input using scanf
scanf("%d",&ival);
//compute square root, print as integer
temp_fp = ival;
root_fp = sqrt(temp_fp);
root = (unsigned int)root_fp;
printf("Square root of %d is: %d",
ival,root); pcrlf();
}
}

```



**FIGURE 10.13** Square root code with RX software FIFO, wakeup support.

Figure 10.14 shows terminal output from the code of Figure 10.13, with the processor allowed to enter sleep mode and then be awakened by character input.

```

Software RX FIFO buff + WDT on input.
Hit any key to start...
Square root of 9 is: 3
Square root of 25 is: 5
Square root of 4 is: 2
Sleeping... ←———— No input, sleep
Awake!      ←———— Character arrival wakes PIC
Square root of 16 is: 4
Square root of 49 is: 7
Square root of 100 is: 10
Sleeping... ←———— No input, sleep
Awake!      ←———— Character arrival wakes PIC
Square root of 64 is: 8
Sleeping...
    
```

**FIGURE 10.14** Test of wakeup code.

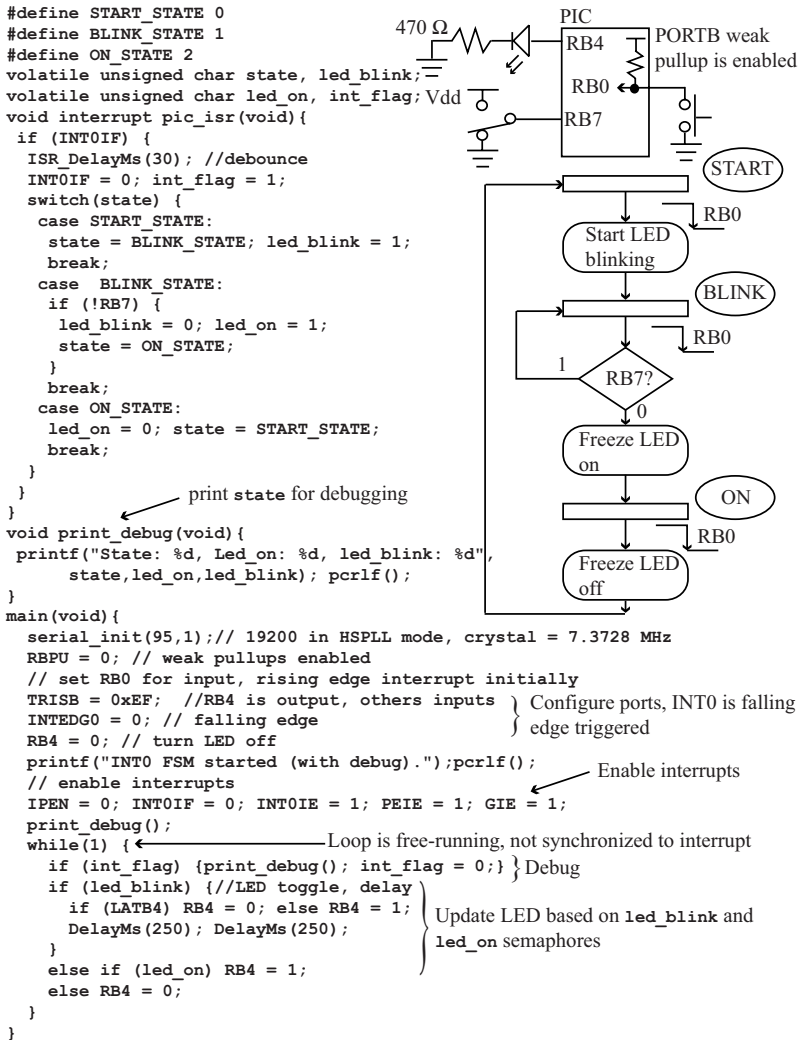
**Sample Question:** Given a low true pushbutton switch on the RB0 port, which is configured as an input, if the switch is pushed and released, what happens to the INT0IF flag assuming INT0IE = 0 and INTEDG0 = 1?

*Answer:* The INT0 input is configured to be rising edge triggered by the statement INTEDG0 = 1. A falling edge occurs on INT0 when the switch is pushed, but this does not affect the INT0IF flag. When the switch is released, a rising edge occurs and the INT0IF flag is set. However, no interrupt is generated because INT0IE = 0. Having INT0IE = 0 does not prevent the INT0IF flag from being set when a rising edge occurs on the RB0 pin.

## **10.7 STATE MACHINE PROGRAMMING FOR INTERRUPT-DRIVEN IO**

The interrupt service routine examples presented to this point have had fairly simple actions, which were repeated each time the interrupt occurred. Many input/output applications require an ISR to perform a sequence of actions over a span of multiple interrupts. A state machine approach can be used to solve this problem as was done previously in Chapter 8. However, there are some differences in using a state machine in an ISR versus using a state machine in the foreground code. Figure 10.15 shows a LED/switch IO example that is implemented using an interrupt-driven finite state machine.

While using interrupts for this example is not necessary, it illustrates an approach that proves useful in later chapters. The application uses interrupt-driven IO via the RB0/INT0 interrupt to detect pushbutton activations that control the state of an LED. The application begins in the START state with the LED off. The first switch activation starts the LED blinking, and moves to the BLINK state. The application remains in the BLINK state where the LED is kept blinking, as long as the RB7 input is a “1” on each switch activation. If the RB7 input is a “0” on switch



**FIGURE 10.15** Interrupt-driven LED/switch IO example.

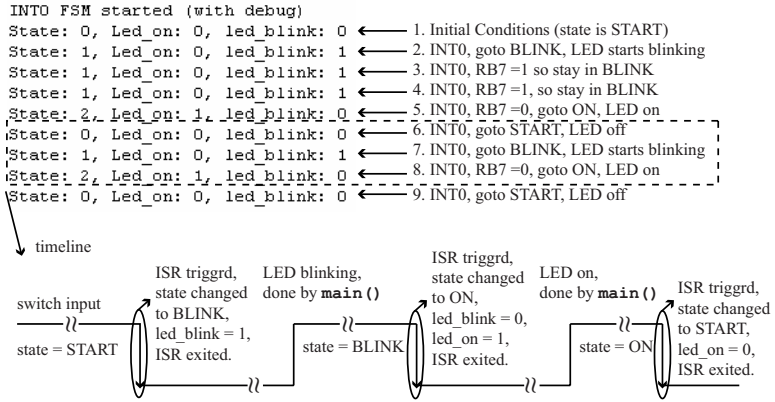
activation, the LED is frozen on, and the application progresses to state ON. The next switch activation turns the LED off, and the application returns to the START state. The interrupt service routine `pic_isr()` in Figure 10.15 implements the state machine with each state transition triggered by a falling edge on RB0/INT0, indicating a switch activation. The `ISR_DelayMs(30)` software delay function is used to debounce the switch input; the `ISR_DelayMs` function is a copy of the `DelayMs` function with only a name change. This is necessary because the HI-TECH PICC-18

compiler does not support function recursion, so the same function cannot be called from both an ISR and `main()`. Typically, delays or waits of any sort in an ISR are to be avoided, as the ISR should do its job as quickly as possible and exit. Section 10.9 examines a more efficient method for debouncing a switch that does not involve placing a delay in the ISR. Observe that the `ISR_DelayMs(30)` function call is placed before the `INT0IF = 0` statement that clears the interrupt flag. This is important, as we need all switch bounces to have settled before clearing the flag because each active edge caused by switch bounce sets the flag. The state variable contains the current state of the ISR, with a `C switch` statement used to select an action based upon the current state. Observe that unlike the state machine code used for the LED/switch IO example in Chapter 8, the case blocks do not wait for IO events since the ISR has been triggered by the pushbutton switch event. Instead, the case blocks modify semaphores `led_on`, `led_blink`, and `int_flag` that are used to communicate with the `main()` code, which is just a free-running loop that controls the LED status. The term *free-running* is used because the internal code of the `while(1){}` loop within `main()` does not have any wait conditions based on variables modified by the ISR. In a real application, the `main()` loop would be performing additional useful work, with the ISR handling IO events. Within the free running `while(1){}` loop of `main()`, the LED is toggled if `led_blink` is “1”, turned on if `led_on` is “1”, or is turned off if neither of these conditions is true. The `int_flag` is only used for debugging purposes; it is set by the ISR when a switch event occurs. Debugging information is printed from the `while(1){}` loop when `int_flag` is set, after which the `int_flag` semaphore is cleared.

Figure 10.16 gives sample terminal output for the code of Figure 10.15. The timeline shows three pushbutton activations. Observe that the active input edge triggers the ISR, and the code in the case block for a state is executed *after* the active edge occurs. Changing the state from START to BLINK in a case block means the BLINK case block is executed on the *next* active edge.

While the output of Figure 10.16 looks valid, there are some constraints that you may not have noticed. For example, during the `DelayMs()` function called by the `while(1){}` loop of `main()`, what happens if one or more switch events arrive? The ISR will process them, but they will be ignored by the `main()` code, as it is executing the delay loop within `DelayMs()`.

Figure 10.17 shows a `DelayMsKill()` function with a `kill_delay` variable that is used to short circuit the delay. The ISR now sets `kill_delay = 1` when halting the blinking of the LED. The `main()` code uses `DelayMsKill()` instead of `DelayMs()`, and now has better responsiveness to requested status changes for the LED. This simply illustrates that when dealing with interrupts, one must be careful about considering the interactions between the ISR and the foreground code.



**FIGURE 10.16** Terminal output for state machine ISR example.

```

char kill_delay;

void DelayMsKill(unsigned char cnt)
{
    unsigned char i;
    do {
        i = 20;
        do {
            DelayUs(50);
        } while(--i && !kill_delay);
    } while(--cnt && !kill_delay);
}

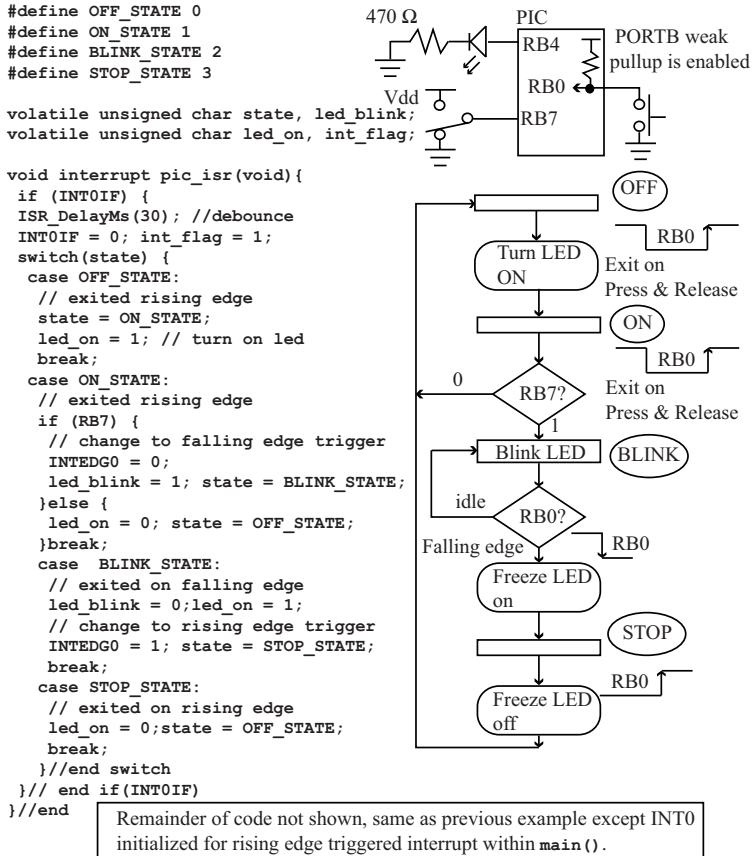
void interrupt pic_isr(void) {
    if (INT0IF) {
        ISR_DelayMs(30);
        INT0IF = 0; int_flag = 1;
        switch(state) {
            case START_STATE:
                kill_delay = 0; ← Ensure that this is zero when blinking
                state = BLINK_STATE; led_blink = 1; ← the LED
                break;
            case BLINK_STATE:
                if (!RB2) {
                    kill_delay = 1; ← Abort delay on the LED
                    led_blink = 0; led_on = 1; ← blink
                    state = ON_STATE;
                }
                break;
            case ON_STATE:
                led_on = 0; state = START_STATE;
                break;
        }
    }
}
    
```

In main(), replace DelayMs() with DelayMsKill()



**FIGURE 10.17** DelayMSKill() function (see CD-ROM file ./code/common/delay.h).

Figure 10.18 shows the ISR of a second interrupt-driven LED/switch IO problem that requires the active edge to be changed between states.

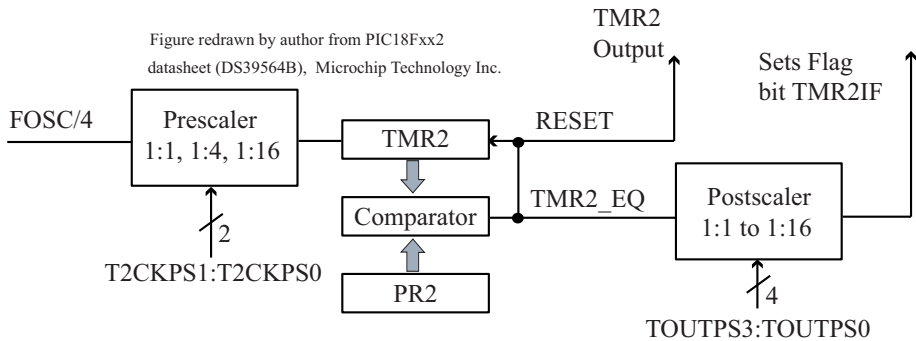


**FIGURE 10.18** LED/switch IO example with active edge changes.

States OFF and ON are exited after a switch press and release, so INT0 is configured for rising-edge triggering in these states. However, state BLINK is exited on a falling edge input to RB0, so state ON changes INT0 interrupt from rising-edge triggered to falling-edge triggered before transitioning to state BLINK. Dynamically changing the active interrupt edge for an INTx interrupt input is useful in Chapter 13, which discusses pulse width measurement.

## 10.8 THE TIMER SUBSYSTEM: TIMER2

It is only natural that a first look at PIC18 timers be done within this chapter, as a common timer application is periodic interrupt generation. The PIC18 has four timers (0–3) with Timer2 discussed here and the other timers covered in Chapter 13. As mentioned previously in the context of the watchdog timer, a timer is a counter that triggers an action when its count reaches a particular value. Figure 10.19 shows the Timer2 subsystem, which consists of an 8-bit timer, a *prescaler* (1, 4, 16), an 8-bit *period register* (PR2), and a *postscaler* (1 through 16). The prescaler divides the timer input clock, whose source is FOSC/4 (the instruction cycle clock). The 8-bit period register is used as the comparison value against the timer value; when they are equal, this asserts the Timer2 equal signal (TMR2\_EQ) and resets the timer to zero. The postscaler sets the Timer2 interrupt flag (TMR2IF) for every 1 of  $n$  TMR2\_EQ events, where  $n$  is 1 through 16. A postscaler value of 1:1 means that the TMR2IF bit is set for each TMR2\_EQ event, while a postscaler value of 1:16 means the TMR2IF bit is set once for every 16 TMR2\_EQ events.

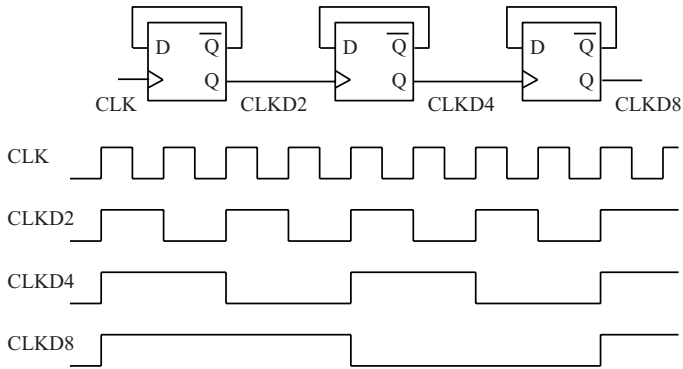


**FIGURE 10.19** Timer2 subsystem.<sup>2</sup>

You may wonder why the prescaler is only limited to values 1, 4, and 16 while the postscaler has values 1 through 16. The prescaler must be a high-speed circuit, as its function is to divide the input clock frequency. Figure 10.20 shows the classic DFF circuit for achieving a divide-by-2 of the input clock frequency. This circuit is fast and reliable with the minimum input clock period limited to  $T_{cq} + T_{su}$ , where  $T_{cq}$  is the propagation delay from clock-to-qn and  $T_{su}$  is the setup time of the D input. This value expressed as a frequency is  $1/(T_{cq}+T_{su})$  and is given as the DFF *toggle frequency* in datasheets. A divide-by- $2^n$  circuit is built by placing  $n$  of these circuits in series as shown in Figure 10.20. The CLKD2 (CLK/2), CLKD4 (CLK/4), and CLKD8 (CLK/8) signals are passed through another DFF (not

<sup>2</sup> Figure 10.19 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.'s prior written consent.

shown) clocked by CLK to resynchronize these signals to CLK. A postscaler does not have to be high speed and is implemented as a second small counter that is placed after the primary counter.



**FIGURE 10.20** Clock divide by 2.

The Timer2 interrupt flag period (TMR2IF\_PER) is given in Equation 10.1 as a function of PR2, TOSC, prescaler (PRE), and postscaler (POST) values. Recall that TOSC is the period of the primary clock, with  $TOSC = (1/FOSC)$ . The “+1” is present because it requires a clock cycle to reset Timer2 back to zero once PR2 and Timer2 become equal. This means Equation 10.1 simplifies to  $4 * FOSC * PRE * POST$  when  $PR2 = 0$ . If  $PR2 = 255$ , Equation 10.1 becomes  $256 * 4 * FOSC * PRE * POST$ . If the maximum values  $PRE = 16$  and  $POST = 16$  are used, Equation 10.1 becomes  $256 * 256 * TOSC * 4$ , which gives the same timeout value as using a 16-bit timer clocked by  $FOSC/4$  ( $2^{16} * TOSC * 4$ ).

$$TMR2IF\_PER = (PR2+1) * PRE * POST * TOSC * 4 \tag{10.1}$$

A common application of timers is periodic interrupt generation to accomplish some action that must be performed at fixed time intervals. Typically, multiple solutions exist for a Timer2 interrupt period, as Equation 10.1 has three variables (PRE, POST, PR2). Given a desired interrupt period, one approach is to pick values for PRE and POST and then solve Equation 10.1 for PR2 because PR2 has the largest range of the three adjustable variables. Equation 10.2 solves Equation 10.1 for PR2; note that the PR2 value must be rounded to the nearest integer.

$$PR2 = \left\lceil \frac{TMR2IF\_PER}{4 * TOSC * PRE * POST} \right\rceil - 1 \quad (\text{round to nearest integer}) \tag{10.2}$$



Figure 10.21a shows four solutions for a Timer2 periodic interrupt at a frequency of 4 kHz (an interrupt period of 250  $\mu$ s). The postscaler was adjusted for the first three cases to give the largest PR2 value that did not exceed 255. The last case (PRE = 1, POST = 1, PR = 1842) is included to show an invalid solution where PR2 exceeds 255, the maximum value of an 8-bit register. The %error between the desired frequency and actual frequency is due to rounding of the PR2 value to the nearest integer. In general, the smallest percent error between expected and actual frequencies usually occurs for the highest PR2 value, but this is not guaranteed. Figure 10.21b shows the effect on percent error as POST is increased, causing PR2 to decrease. The smallest percent error occurs for the case of PRE = 1, POST = 9, and PR2 = 204. The required accuracy for a periodic interrupt interval is application dependent; in most applications any of the valid solutions in Figure 10.21 would be acceptable.

(a) Timer2 solutions for 4 kHz interrupt frequency

FOSC	Pre	Post	Desired Frequency	PR	Actual Frequency	% Error
29491200	1	7	4000	262	4004.8	0.12%
29491200	4	2	4000	229	4007.0	0.17%
29491200	16	1	4000	114	4007.0	0.17%
29491200	1	1	4000	1842	4000.4	0.01%

← Invalid, PR2 > 255

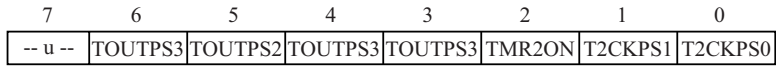
(b) Timer2 solutions for 4 kHz interrupt frequency, increasing postscaler value

FOSC	Pre	Post	Desired Frequency	PR	Actual Frequency	% Error
29491200	1	7	4000	262	4004.8	0.12%
29491200	1	8	4000	229	4007.0	0.17%
29491200	1	9	4000	204	3996.1	-0.10%
29491200	1	10	4000	183	4007.0	0.17%
29491200	1	11	4000	167	3989.6	-0.26%
29491200	1	12	4000	153	3989.6	-0.26%
29491200	1	13	4000	141	3993.9	-0.15%
29491200	1	14	4000	131	3989.6	-0.26%
29491200	1	15	4000	122	3996.1	-0.10%

**FIGURE 10.21** PR2 values for an interrupt frequency of 4 kHz.

Figure 10.22 shows the bit definitions of the Timer2 configuration register (T2CON). Note that the postscale bit field is one less than the desired postscaler value (for 1:4 use 0b0011, for 1:5 use 0b0100, for 1:16 use 0b1111, etc.).

T2CON: Timer2 Control Register



-- u -- : unimplemented

TMR2ON: Timer2 On Bit (1 is on, 0 is off)

TOUTPS3:TOUTPS2 Postscale Select

0000 = 1:1 Postscale

0001 = 1:2 Postscale

0010 = 1:3 Postscale

.....

1110 = 1:15 Postscale

1111 = 1:16 Postscale

T2CKPS1: T2CKPS0: Timer2 Clock Prescale

00 = Prescaler is 1

01 = Prescaler is 4

1x = Prescaler is 16

**FIGURE 10.22** Timer2 configuration register (T2CON).

**Sample Question:** Assuming  $FOSC = 30$  MHz, what Timer2 configuration will generate a periodic interrupt every 5 ms?

*Answer:* Using Equation 10.2 and letting  $POST = 16$ , we find:

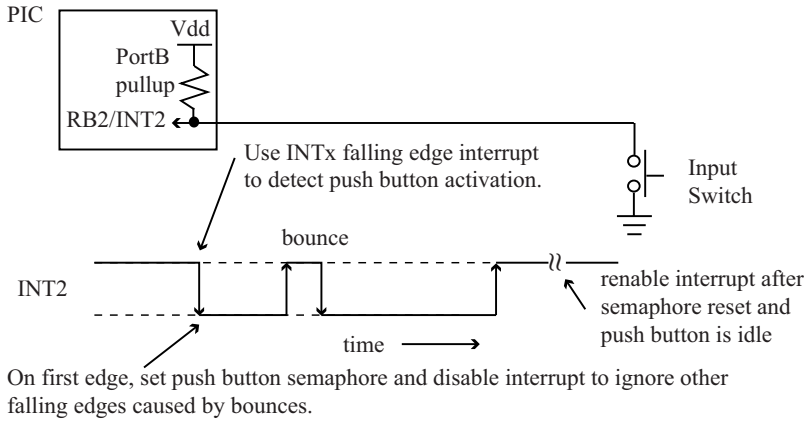
$$PR2 = \lceil 5 \text{ ms} / [(1/30 \text{ MHz}) * 4 * PRE * 16] \rceil = \lceil 0.005 / [3.33e-8 * 4 * PRE * 16] \rceil$$

This results in  $PR2 = 2343$  for  $PRE = 1$ ,  $PR2 = 585$  for  $PRE = 4$ , and  $PR2 = 145$  for  $PRE = 16$ . Thus, the only valid choice for  $POST = 16$  is  $PRE = 16$ ,  $PR2 = 145$ , as this is the only configuration that gives a  $PR2 < 255$ .

## 10.9 SWITCH DEBOUNCING USING A TIMER

The LED/switch examples of Section 10.7 use a 30 ms software delay in the ISR for switch debouncing. This is not the best method to use, as the ISR is stealing time from the foreground code via wasted cycles in the software delay loop. A better method is to use a timer for switch debouncing as shown in Figure 10.23. The goal is to create a semaphore that signals a press and release of the momentary switch in the presence of switch bounce. Timer2 is configured to generate periodic interrupts and the INTx input is configured as a falling edge interrupt.

The first falling edge from a switch activation triggers the ISR, which sets a semaphore and then disables the interrupt so that successive falling edges due to switch bounce do not generate interrupts. On each Timer2 interrupt thereafter, a counter is kept that tracks the number of successive Timer2 interrupts that the INT2 input remains high. If the INT2 input remains high long enough, it is considered idle and the INT2 interrupt is re-enabled.



**FIGURE 10.23** Using a timer to debounce an interrupt-driven switch input.

Figure 10.24 shows the C code implementation of this debounce scheme. The `button` variable is the semaphore that is set by the ISR when a falling edge occurs on the INT2 input indicating that switch activation has occurred. Observe that once the INT2 interrupt is recognized, it is disabled via `INT2IE = 0` and the `button_debounce` count value is cleared. The INT2IF flag is not cleared at this time because any switch bounce that is present sets the flag; the INT2IF flag cannot be reliably cleared until the switch bounce has settled. For each Timer2 interrupt, if the INT2 interrupt is disabled, this means that the last switch activation is being debounced. If the RB2 input is high, the debounce counter, `button_debounce`, is incremented. If the RB2 input is low, `button_debounce` is cleared. Once RB2 has tested high for DEBOUNCE consecutive Timer2 interrupts it is considered idle. If RB2 is idle and the previous semaphore has been acknowledged (i.e., `button` has been cleared by the foreground code), the INT2 interrupt is re-enabled via `INT2IE = 1` for triggering by a switch activation.

The `main()` code of Figure 10.24 initializes the INT2 input for falling edge triggering and Timer2 for periodic interrupt generation. The values `POST = 11`, `PRE = 16`, and `PR2 = 250` for a `FOSC = 29.4952 MHz` (PIC reference board) give a Timer2 interrupt period of approximately 6 ms. With `DEBOUNCE = 5`, this means that the RB2 input remaining high for approximately 24 to 30 ms is considered idle. The variation in the debounce time is because the Timer2 value is unknown when the switch activation occurs. Thus, the first Timer2 interrupt may occur anywhere in the 5 ms interrupt interval. The `while(1){}` loop is free-running with respect to the button semaphore; when the `button` semaphore is set, a message is printed and the semaphore is acknowledged by clearing it.

```

#define DEBOUNCE 5 ← 5 * 6 ms = 24 to 30 ms debounce time
volatile unsigned char button, button_debounce;

void interrupt pic_isr(void){
  if (INT2IF && INT2IE) {
    // pushbutton detected
    INT2IE = 0;  button = 1;  button_debounce = 0;
  }
  if (TMR2IF) { // debouncing timer
    TMR2IF = 0;
    if (!INT2IE) {
      if (RB2) {
        if (button_debounce != DEBOUNCE)
          button_debounce++;
      }
      else button_debounce=0;
      if (button_debounce == DEBOUNCE && !button){
        //button idle high ,re-enable interrupt
        INT2IF=0; INT2IE=1;
      }
    }
  }
}

main(void){
  serial_init(95,1); // 19200 in HSPLL mode, crystal = 7.3728 MHz
  // configure INT2 for falling edge interrupt input
  TRISB2 = 1; INT2IF = 0; INTEDG2 = 0; INT2IE = 1;
  RBPU = 0; // enable weak pullup on port B
  // configure timer 2
  // post scale of 11, prescale 16, PR=250, FOSC=29.4912 MHz
  // gives interrupt interval of ~ 6 ms
  TOUTPS3 = 1; TOUTPS2 = 0; TOUTPS1 = 1; TOUTPS0 = 0;
  T2CKPS1 = 1; PR2 = 250;
  // enable TMR2 interrupt
  IPEN = 0; TMR2IF = 0; TMR2IE = 1; PEIE = 1; GIE = 1;
  TMR2ON = 1 ;
  printf("Pushbutton with Timer2 Debounce"); printf();
  while(1) {
    if (button) {
      button=0; // acknowledge this semaphore
      printf("Push Button activated!"); printf();
    }
  } // end while
} //end main

```



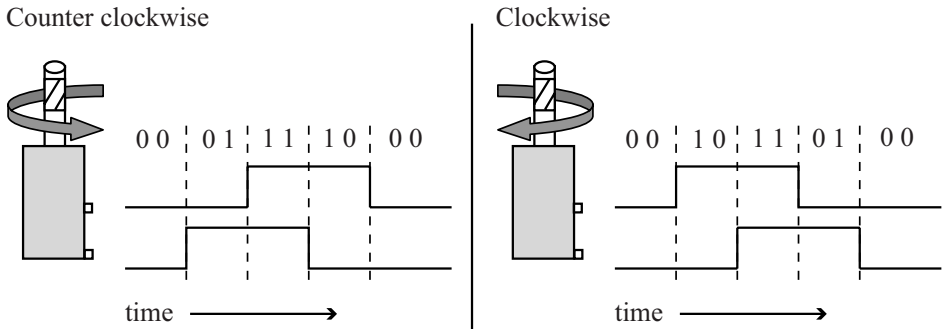
**FIGURE 10.24** Switch debounce implementation.

This approach sets the button semaphore on each press and release of the push button switch. If the interface requires that a press and hold be detected, a similar approach that waits for the input to be idle low can be used. The next section discusses a second example in which a periodic interrupt is used to sample noisy inputs to reject momentary pulses or glitches that may be present.

## 10.10 A ROTARY ENCODER INTERFACE

A rotary encoder is used to encode the direction and distance of mechanical shaft rotation. There are different ways to accomplish this; Figure 10.25 shows a 2-bit

Gray code rotary encoder. Clockwise rotation of the shaft produces the sequence 00, 10, 11, 01, and counterclockwise rotation produces 00, 01, 11, and 10. In a Gray code, adjacent encodings differ by only one bit position. Rotation direction is determined by comparing the current 2-bit value with the last value. For example, if the current value is 11 and the last value is 10, the shaft is rotating in a clockwise direction. One common use for a rotary encoder is as an input device on a control panel where clockwise rotation increments a selected parameter setting, while counter-clockwise rotation decrements the parameter. The rotary encoder of Figure 10.25 is an *incremental* encoder as the absolute position of the shaft is indeterminate; only relative motion is encoded. Some rotary encoders include more bits that provide absolute shaft position, in BCD or binary encoding. An  $n$ -position encoder outputs  $n$ -codes for each complete shaft rotation. Common values of  $n$  for 2-bit incremental rotary encoders are 16 and 32.



**FIGURE 10.25** Two-bit Gray code rotary encoder.

Rotary encoders use mechanical, optical, or magnetic means of detecting shaft rotation, with mechanical encoders being the least expensive and magnetic the most expensive. A key specification for optical and mechanical encoders is rotational life with optical ~ 1 million and mechanical ~ 100,000 rotations due to mechanical wear. Magnetic encoders are meant for high-speed rotational applications with encoder lifetime measured in thousands of hours for a fixed rotational speed in revolutions per minute (RPMs).

Figure 10.26 shows ISR code that uses INT0/INT1 edge triggered interrupts for a rotary encoder interface. The ISR triggers on the occurrence of an active edge on either INT0 or INT1. The ISR checks the flag bits INT0IF/INT1IF, determines which interrupt occurred, and toggles the appropriate edge bit INTEDG0/INTEDG1. The `update_state()` function then reads the INT0/INT1 inputs and compares them against the previous state to determine clockwise or counterclockwise

rotation of the encoder. If a valid state transition is found, the *count* variable is either incremented or decremented appropriately. Observe that an invalid state transition indicates that an illegal transition has occurred, perhaps caused by noise, and the state variable is reset to the current value of the INT0/INT1 inputs.

```

#define S0 0
#define S1 1
#define S2 2
#define S3 3

volatile unsigned char state, last_state;
volatile unsigned char count, last_count;

update_state(){
    state = PORTE & 0x3;
    switch(state) {
    case S0:
        if (last_state == S1) count++;
        else if (last_state == S2) count--;
        break;
    case S1:
        if (last_state == S3) count++;
        else if (last_state == S0) count--;
        break;
    case S2:
        if (last_state == S0) count++;
        else if (last_state == S3) count--;
        break;
    case S3:
        if (last_state == S2) count++;
        else if (last_state == S1) count--;
        break;
    }
    if (last_count != count) {
        // valid pulse
        last_state = state;
        last_count = count;
    } else {
        // invalid transition, reset last state
        last_state = state;
    }
}

void interrupt pic_isr(void){
    if (INT0IF || INT1IF) {
        if (INT0IF) {
            INT0IF = 0;
            // toggle active edge
            if (INTEG0) INTEG0 = 0; else INTEG0 = 1;
        }
        if (INT1IF) {
            INT1IF = 0;
            // toggle active edge
            if (INTEG1) INTEG1 = 0; else INTEG1 = 1;
        }
        update_state();
    }
}

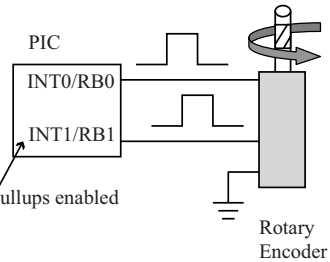
```

Update the *state* and *count* variables based upon the INT0/INT1 input values.

Check previous state and determine if rotating clockwise or counter-clockwise

Internal pullups enabled

Should not get here unless illegal transition occurred, attempt a recovery



**FIGURE 10.26** Two-bit rotary encoder interface using INT0/INT1 interrupts.

The `main()` code shown in Figure 10.27 initializes the INT0/INT1 active interrupt edges (INTEDG0/INTEDG1) by reading the current value of the INT0/INT1 inputs; if the input is high, the falling edge is chosen, else the rising edge is selected. This is necessary because the initial values of the rotary encoder outputs depend upon the current shaft position, which is unknown at `main()` startup. The state variable used by the ISR of Figure 10.26 to track the position of the rotary encoder is also initialized by `main()` based upon the INT0/INT1 inputs. The `while(1){}` loop in the `main()` code waits for a change on the count variable and prints its value once a change is detected.

```
main(void){
    unsigned char count_old;
    // set RB0, RB1 for input
    TRISB0 = 1; TRISB1 = 1;
    RBPU = 0; // enable weak pullups

    if (RB0) INTEDG0 = 0; // falling edge
    else INTEDG0 = 1; // rising edge
    if (RB1) INTEDG1 = 0; // falling edge
    else INTEDG1 = 1; // rising edge
} Initialize active edges based on
INT0/INT1 input values.

    last_state = PORTE & 0x03; // init last state

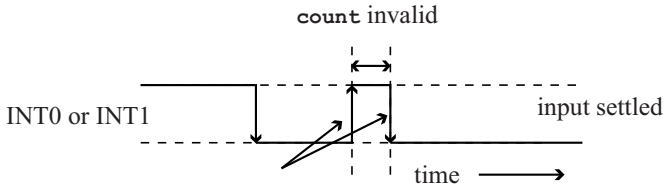
    serial_init(95,1); // 19200 in HSPLL mode, crystal = 7.3728 MHz
    pcrflf(); printf("Rotary Test Started"); pcrflf();
    // enable INT0,INT1 interrupts
    IPEN = 0; INT0IF = 0; INT0IE = 1;
    INT1IF = 0; INT1IE = 1;
    PEIE = 1; GIE = 1;
} Enable INT0/INT1 Interrupts

    printf("No Debounce"); pcrflf();
    printf("Rotary Switch Test Started");
    pcrflf();
    while(1) {
        //tip: don't put volatile variables in printf's, may change
        // by the time the printf gets around to printing it!
        if (count != count_old){
            count_old = count;
            printf("Count: %x",count_old); ← Print count variable when it
            pcrflf(); changes
        }
    }
}
```

**FIGURE 10.27** `main()` for initializing INT0/INT1 interrupts, state variable.

The code of Figures 10.26 and 10.27 works well as long as the signal transitions are noise free and clean of contact bounce, which is generally true of the signals produced by optical and magnetic encoders. However, mechanical encoders have contact bounce that will cause the count variable to change multiple times for a single shaft movement, potentially creating errors in code that samples the count value. Figure 10.28 illustrates this problem, as the ISR is triggered on each edge of the contact bounce, causing count to be modified each time. There is a possibility

that count can be sampled by the normal program flow when it contains an incorrect value, causing unreliable behavior.



Contact bounce edges, ISR triggered on each edge, modifies `count` each time as each state transition is valid since it is returning to the previous state on the bounce.

**FIGURE 10.28** Switch bounce in mechanical encoders.

As was done for the LED/switch IO example of the previous section, Timer2 is used as a periodic interrupt for debouncing the rotary encoder inputs. Figure 10.29 shows an ISR triggered by Timer2 that samples the INT0/RB0, INT1/RB1 inputs on each interrupt. The `int0_last`, `int1_last` variables contain the last stable values of the INT0, INT1 inputs. If an input is different from its last stable value and remains that way for `DEBOUNCE` consecutive interrupt periods, the input has reached a new stable value and the `update_state()` function is called to update the state, count variables. The `int0_cnt`, `int1_cnt` variables are used for tracking the number of consecutive interrupts that an input remains changed from its previous value. The count variable for an input is reset to zero if the input returns to its previous value before `DEBOUNCE` consecutive interrupt periods occurs. This approach uses Timer2 as the only interrupt source, the RB0/INT0 and RB1/INT1 interrupts are not enabled. The `update_state()` function is not shown in Figure 10.29, as it is the same function from Figure 10.26.

Figure 10.30 shows the `main()` code for configuring Timer2 as a periodic interrupt source. Timer2 is configured in the same manner as the switch debouncing example in which values of `FOSC = 29.4952 MHz`, `POST = 11`, `PRE = 16`, and `PR2 = 250` give an interrupt period of approximately 6 ms. With `DEBOUNCE = 5`, this means that any pulses of width less than 30 ms are rejected as switch bounce or noise. The pulse width rejection should be chosen based on worst-case datasheet values for contact bounce. The sampling period should be chosen to guarantee several samples between valid input changes, with the time between input changes dependent upon the maximum expected shaft rotation speed and the number of positions for the encoder.



```

volatile unsigned char state,last_state;
volatile unsigned char count,last_count;
volatile unsigned char int0_cnt,int0_last; } Variables for tracking stability
volatile unsigned char int1_cnt,int1_last; } of rotary encoder inputs
volatile unsigned char update_flag;

#define DEBOUNCE 5  ← Rotary encoder input must be stable
                    for this many consecutive Timer2
                    interrupts to be classified as a valid input

void interrupt pic_isr(void){
    if (TMR2IF) {
        // debouncing rotary inputs
        TMR2IF = 0;
        if (RB0 != int0_last) { ← Has RB0 input changed value?
            int0_cnt++; ← Yes, track number of times it remains stable
            if (int0_cnt == DEBOUNCE) { ← Stable for DEBOUNCE interrupt periods,
                update_flag = 1; } set update flag, record input value
                int0_cnt = 0;int0_last = RB0;
            }
        }
        // reset cnt, if pulse width
        // not long enough
        else if (int0_cnt) int0_cnt = 0; ← Reset counter if not
                                         stable for long enough

        if (RB1 != int1_last) {
            int1_cnt++;
            if (int1_cnt == DEBOUNCE) {
                update_flag = 1;
                int1_cnt = 0; int1_last = RB1;
            }
        }
        // reset cnt, if pulse width
        // not long enough
        else if (int1_cnt) int1_cnt = 0; } Check stability of RB1 input

        if (update_flag) {
            // can read the rotary inputs } Update state and count variables;
            update_state(); } update_state() function not shown as
            update_flag = 0; } it is unchanged from previous example.
        }
    }
}

```



**FIGURE 10.29** Using Timer2 to sample the rotary encoder outputs.

```

main(void) {
    unsigned char count_old;
    // set RB0, RB1 for input
    TRISB0 = 1; TRISB1 = 1;
    RBPU = 0; // enable weak pullups
    int0_last = RB0;
    int1_last = RB1;
    last_state = PORTE & 0x03; // init last state
    serial_init(95,1); // 19200 in HSPLL mode, crystal = 7.3728 MHz

    // configure timer 2
    // post scale of 11
    TOUTPS3 = 1; TOUTPS2 = 0; }
    TOUTPS1 = 1; TOUTPS0 = 0; } Postscale bits = 1010 for postscaler of 1:11
    // pre scale of 16 */
    T2CKPS1 = 1;          ← Prescale = 16
    TMR2ON = 1;          ← Turn on Timer2
    PR2 = 250;           ← Set period register

    // enable TMR2 interrupt
    IPEN = 0; TMR2IF = 0; TMR2IE = 1; } Enable Timer2 interrupt
    PEIE = 1; GIE = 1;
    printf("With Timer2 Debounce"); pcrLf();
    printf("Rotary Switch Test Started");
    pcrLf();
    while(1) {
        //tip: don't put volatile variables in printf's, may change
        // by the time the printf gets around to printing it!
        if (count != count_old) {
            count_old = count;
            printf("Count: %x", count_old);
            pcrLf();
        }
    }
}

```



**FIGURE 10.30** Configuring Timer2 for sampling the rotary encoder inputs (see CD-ROM file ./code/chap10/F\_10\_29\_rotint\_debounced.c).

## 10.11 A NUMERIC KEYPAD INTERFACE

A numeric keypad is a common element in a microcontroller system, as it provides an inexpensive method of input. A numeric keypad is simply a matrix of switches arranged in rows and columns and has no active electronics; a keypress connects a row and column pin together as shown in Figure 10.31.

The 4x3 numeric keypad of Figure 10.31 is shown connected to the PIC in Figure 10.32. The RB[3:1] port pins are configured as outputs driving low and connected to the row pins, while RB[7:4] are configured as inputs with the weak pullup enabled and connected to the column pins.

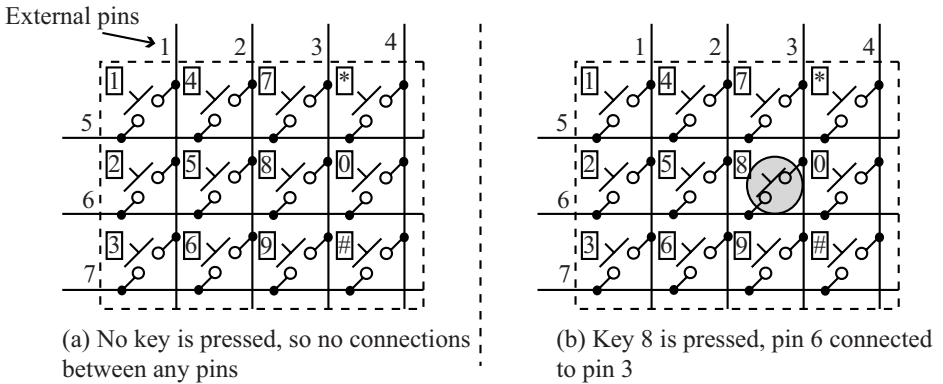


FIGURE 10.31 4x3 numeric keypad.

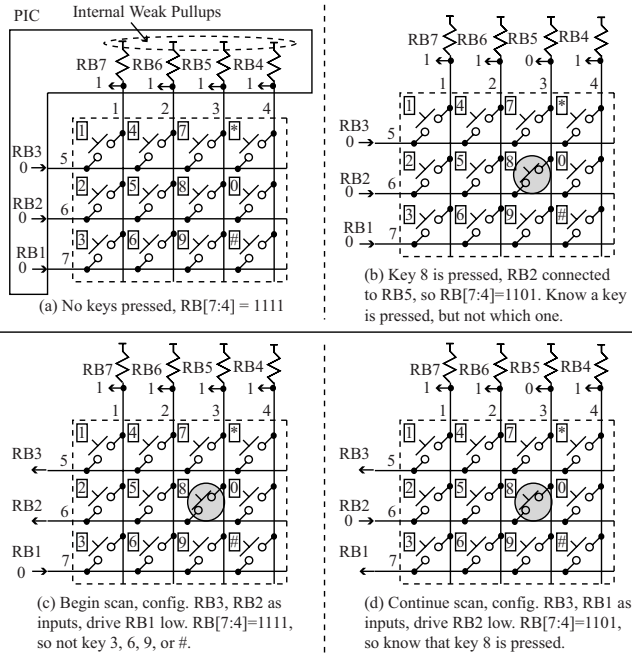


FIGURE 10.32 4x3 numeric keypad connected to PIC.

If no key is pressed,  $RB[7:4]$  reads as “1111”, as there are no connections to the  $RB[3:1]$  pins. Recall that the interrupt-on-change feature of PORTB generates an interrupt if the value of pins  $RB[7:4]$  changes from the last time a read was done on this port. By reading PORTB when no key is pressed and latching a “1111” into the

PORTB[7:4] register bits, the interrupt-on-change feature can be used to detect a keypress, as any keypress connects one of the RB[7:4] inputs to one of the RB[3:1] outputs, causing that input to become “0” and changing the RB[7:4] port value. In Figure 10.32b, key “8” is pressed, causing RB5 to become a “0”.

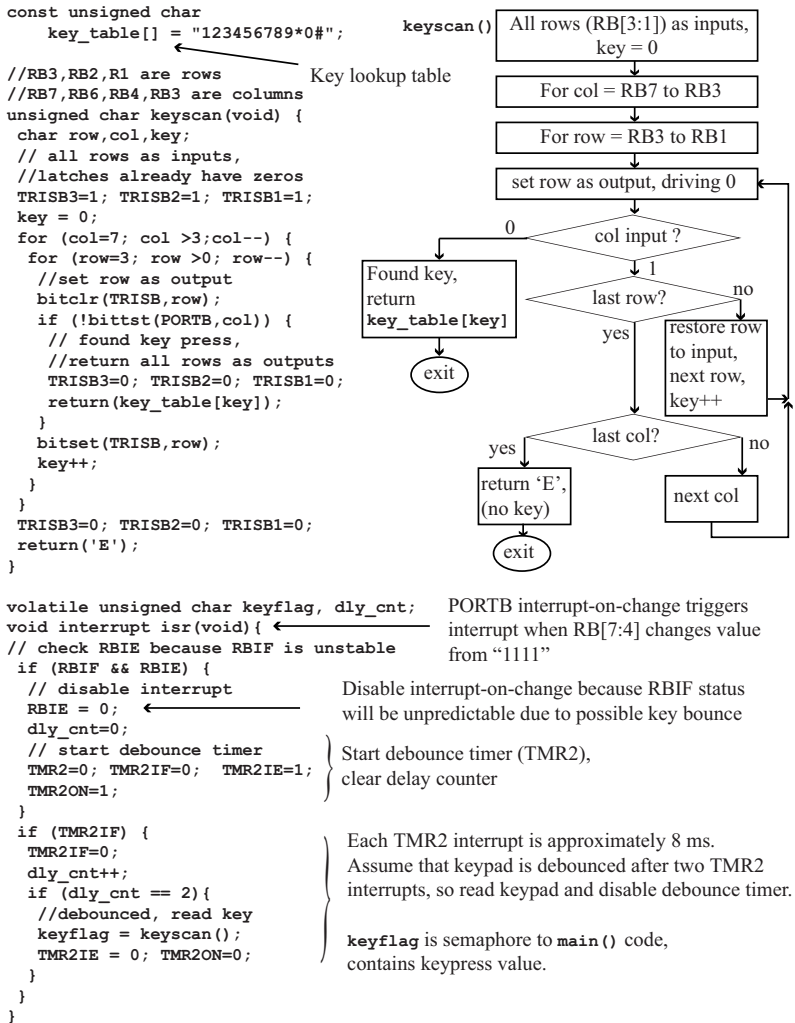
After a keypress is detected, determining which key is actually pressed is done through a procedure known as a *keypad scan*. A keypad scan steps through the rows, configuring each row port as an output driving low with the other ports configured as inputs. For each row configured as an output, the corresponding column input is checked to see if it is a “0”, indicating a connection between the row and column pin. In Figure 10.32c, RB1 is driving low but all of the column inputs RB[7:4] are high, so it is known that keys 3, 6, 9, and # are not pressed. In Figure 10.32d, RB2 is driving low and column input RB5 is found to be low, indicating that key 8 is pressed causing the connection between RB2 and RB5.

Figure 10.33 shows the ISR code and `keyscan()` function for the keypad interface. The `keyscan()` function scans the keypad as previously described using two nested loops. The outer loop steps through the column inputs, while the inner loop steps through the rows, configuring each row as an output driving low and checking the column input to see if it is zero. The `key` variable is incremented each time through the inner loop and is used as an index into the `key_table` string to return an ASCII value of the key that is pressed.

The ISR is initially triggered by RBIF, the interrupt-on-change of port RB[7:4] from “1111” to a different value indicating a key press. The RBIF interrupt is disabled once it has occurred, as clearing the RBIF flag is unreliable at this point as it may become set again because of key bounce. Timer2 is used to debounce the keypad and has been configured for an 8 ms interrupt period. After the RBIF interrupt is disabled, the `dly_cnt` variable that is used to track the number of Timer2 interrupts is cleared to zero, Timer2 is turned on, and the Timer2 interrupt is enabled. Once two Timer2 interrupts have occurred indicating that 16 ms has elapsed since a keypress was detected, the keypad is considered debounced and the keypad value is read by `keyscan`. The returned keypad value is placed in the `keyflag` variable, which is used as a semaphore to `main()` to indicate that a keypress has occurred. Observe that the test for an RBIF interrupt is written as `RBIF && RBIE` because the RBIF flag cannot be reliably cleared as long as a key is pressed or switch bounce is active.

The `main()` code for the keypad interface is shown in Figure 10.34. The initialization includes configuring RB[7:4] as inputs and RB[3:1] as outputs and enabling the weak pullup. Timer2 is configured for `PRE = 16`, `POST = 16`, and `PR2 = 229`, which gives an 8 ms timeout for `FOSC = 29.4912 MHz`.

The `while(1){}` loop first calls the `key_pad()` function, which waits until the RB[7:4] inputs have been stable at “1111” for 100 ms, indicating an idle condition



**FIGURE 10.33** ISR and keyscan() function keypad interface.

(no key is pressed). The key\_pad() function clears the RBIF flag before exiting, as the interrupt-on-change feature can now be used to detect a keypress since a “1111” is latched into RB[7:4] by the last PORTB read. The while(1){} loop then clears the keyflag semaphore, enables the interrupt-on-change interrupt (RBIE = 1), and then waits for the ISR to set the keyflag semaphore. After the keyflag semaphore becomes nonzero, its value is printed to the console as the keypress value. If the “#” key is pressed, a carriage return/line feed is also printed. Sample output from the keypad application is shown after the main() code of Figure 10.34.

```

// wait until keypad idle for 100 ms
// when exit RB[7:4] = 1111
keypad_idle() {
    char cnt,c;
    cnt = 0;
    while(cnt < 5) {
        c = PORTB & 0xF0;
        if (c == 0xF0) cnt++;
        else cnt=0;
        DelayMs(20);
    }
    // clear interrupt flag now that port is stable
    RBIF = 0; ← Clear RBIF flag after RB[7:4] is stable.
}

main(void){

    // 19200 in HSPLL mode, crystal = 7.3728 MHz
    serial_init(95,1);
    TRISB=0xF1; // RB3,RB2,R1 outputs
    RB3=0; RB2=0; RB1=0; // pull these low
    // enable the weak pullup on port B
    RBPU = 0;
    // configure timer 2
    // post scale of 16
    TOUTPS3 = 1; TOUTPS2 = 1;
    TOUTPS1 = 1; TOUTPS0 = 1;
    // pre scale of 16 */
    T2CKPS1 = 1;
    PR2 = 229; // interrupt interval of 8 ms
    // unmask interrupts
    IPEN = 0; PEIE = 1; GIE = 1;
    keyflag = 0;
    pcrLf();printf("Keypad test"); pcrLf();
    while(1) {
        // wait for key
        keypad_idle();
        keyflag = 0;
        RBIE = 1;
        while(!keyflag);
        printf("%c",keyflag);
        if (keyflag == '#')
            pcrLf();
    }
}

```

After a keypress, wait for keypad to become idle by checking that RB[7:4] remains "1111" for 100 ms. This is needed so that the next keypress is detected by interrupt-on-change of RB[7:4].

Configure PORTB for keypad interface

Configure Timer2 for PRE=16, POST=16, PR2=229. Interrupt interval is ~8 ms for FOSC=29.4912 MHz

Wait for keypad to become idle, clear **keyflag** semaphore. Enable interrupt-on-change, wait for ISR to set **keyflag** semaphore. Wait for the **keyflag** semaphore to become non-zero, then print keypress; if '#' then print CRLF

Keypad test  
123456789\*0# ← Console output for test of key pad code  
147\*2580369#  
369#  
#



**FIGURE 10.34** main() code for keypad interface (see CD-ROM file ./code/chap10/F\_10\_33\_keytst.c).

## 10.12 ON WRITING AND DEBUGGING ISRS

When writing ISRs, one must be careful not to place too much work within the ISR, as this can either cause other interrupt events to be missed or steal too much time away from the normal program flow. Within an ISR, there should never be a wait for an event—that is the function of the interrupt that triggers the ISR. As much work as possible should be placed in the normal program flow, with the ISR only performing time-critical operations.

Because most ISRs are time-sensitive, putting print statements that output data to a serial port in an ISR to examine ISR variables is usually not an option, as this destroys the interrupt timing. It may be valid to place a print statement in the foreground code to examine an ISR variable, but the variable value should always be copied to a temporary variable first because the ISR variable value may be changed by the time the print statement is executed. If you are trying to trace a variable value over several interrupt intervals, using a print statement may not be an option if the print is not fast enough to monitor the variable value. In this case, a *trace buffer* can be used in which copies of the variable over several interrupt intervals are kept, and then printed when the trace buffer fills up. Listing 10.1 shows modifications to the code of Figure 10.26 that adds a trace buffer (`trace`) with 16 entries (`#define TMAX 16`) for tracking changes to the count variable. Within the `update_state()` function, the count variable is saved in `trace` anytime its value changes. The `while(1){}` loop of `main()` prints the contents of the trace buffer after it fills up, and then empties the buffer by resetting the trace buffer count variable `tcnt` to zero.



**LISTING 10.1** Trace buffer modifications for code of Figure 10.26.

```
#define TMAX 16
volatile unsigned char trace[TMAX]; // trace buffer
volatile unsigned char tcnt; //pointer to trace buffer entry

update_state(){
    ...// code sections omitted...

    if (last_count != count) {
        // valid pulse
        last_state = state;
        last_count = count;
        // save count value in trace buffer
        if (tcnt != TMAX) trace[tcnt++] = count;
    } else {
        // invalid transition, reset last state
        last_state = state;
    }
} // end update_state()

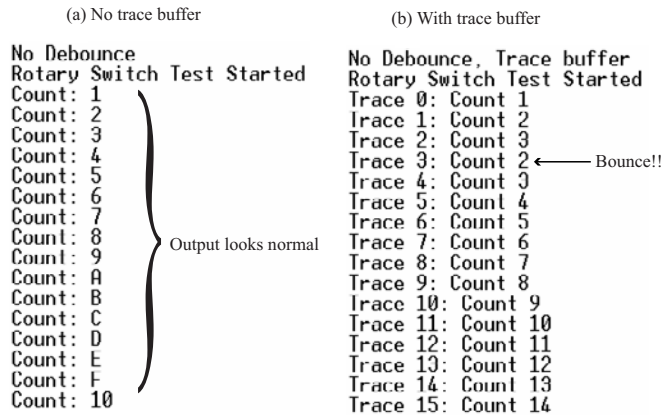
main(void) {
    // code omitted
    while(1) {
        // print trace buffer contents when full
        if (tcnt == TMAX){
            for (i=0; i< TMAX;i++) {
                printf("Trace %d: Count %d",i,trace[i]);
                pcrLf();
            }
            tcnt = 0;// mark trace buffer as empty
        }
    }
}
```

```

}
} // end main()

```

Figure 10.35 shows console output for the code of Figure 10.26 without a trace buffer and with a trace buffer (Listing 10.1). The code was tested with a mechanical rotary encoder that was turned rapidly in an effort to produce contact bounce. The console output of the trace buffer code over several tests clearly showed contact bounce with a typical output given in Figure 10.35b. The code without a trace buffer either showed expected values for count as seen in Figure 10.35a, or skipped values because the print statement in the `while{}`  loop could not keep up with the count variable update. Clearly, a trace buffer is required in this case to observe the actual changes made to the count variable.



**FIGURE 10.35** Console output comparison for Figure 10.27 code.

## SUMMARY

---

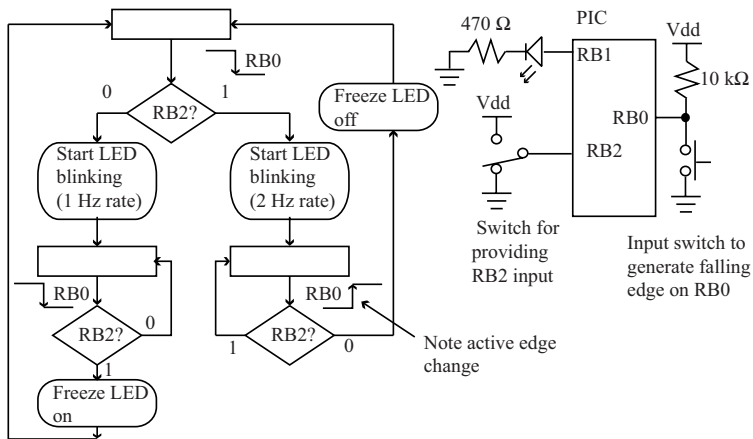
Interrupts are crucial for efficient input/output operations. Interrupt-driven IO means that an IO event generates an interrupt, suspending normal program execution while an interrupt service routine performs the requested IO operation. This is more efficient than polled IO, as buffers can be used to store incoming or outgoing data, and less CPU cycles are wasted in checking for data availability or IO operation termination. The PIC18 interrupt system provides many different interrupt sources, from external pin events to internal events generated by hardware subsystems. Some interrupts can wake the processor from sleep mode, which allows the processor to conserve power while waiting for IO activity. Complex



interrupt-driven IO sequences may require a finite state machine approach, in which the ISR uses a state variable to traverse different sets of actions that span multiple interrupt events. Timers have many uses, one of which is to efficiently de-bounce mechanical inputs.

## REVIEW PROBLEMS

1. The return and retfie instructions perform very similar actions, but have a key difference. What is this difference and why is it needed?
2. Modify the ISR code of Figure 10.10 to turn on an LED connected to RB2 when overrun of the software FIFO buffer is detected.
3. In the logic diagram of Figure 10.2, what is the function of gate  $g1$ ?
4. In the logic diagram of Figure 10.2, what is the function of gate  $g2$ ?
5. In the logic diagram of Figure 10.2, what is the function of gate  $g3$ ?
6. In the logic diagram of Figure 10.2, what is the function of gate  $g4$ ? (Hint: When  $IPEN = 0$ , you will also need to consider the function of gate  $g1$ .)
7. *Interrupt latency* is the elapsed time between an interrupt occurrence, and execution of the ISR's first instruction. Propose a method for measuring this value using an oscilloscope and a program of your design.
8. Write C code that implements the state chart of Figure 10.36. Use an approach similar to that used in Figure 10.15.



**FIGURE 10.36** State chart for sample application.

9. Assume an application heavily relies on software delay loops like `DelayMS()`, and requires that they be accurate. If the code also requires the use of interrupts, what can happen to the accuracy of the software delay loops? What can be done to preserve the accuracy? What would the side effect of this be?

The following questions assume that the same pushbutton input switch is connected to both RB2/INT2 and RB1/INT1; these inputs are brought low when the pushbutton is pressed. The code in Listing 10.2 is the base application for these questions. Each question is independent of the other questions; the requested code changes are not cumulative.

**LISTING 10.2** INTO/INT1 interrupt exercise.

---

```
void interrupt high_isr(void){
    if (INT2IF) {
        INT2IF = 0;    // STATEMENT A
    }
    if (INT1IF) {
        INT1IF = 0;    // STATEMENT B
    }
} // end high_isr()

void interrupt low_priority low_isr(void)
{
    if (INT2IF) {
        INT2IF = 0;    // STATEMENT C
    }
    if (INT1IF) {
        INT1IF = 0;    // STATEMENT D
    }
} // end low_isr()

main(void){
    // set RB1/RB2 for input, falling edge interrupt
    TRISB = 0x06;
    INTEDG2 = 0; INTEDG1 = 0;
    IPEN = 1; // priorities enabled
    // enable INT2 interrupt, low priority
    INT2IP = 0; INT2IF = 0; INT2IE = 1;
    // enable INT1 interrupt, high priority
    INT1IP = 1; INT1IF = 0; INT1IE = 1;
    PEIE = 1; GIE = 1;
    while(1); // do nothing
}
```

10. For Listing 10.2, give the order in which the statements A, B, C, D are executed, if they are executed at all, when the pushbutton is pressed and released one time. Be careful—priorities are enabled by the statement  $IPEN = 1$ .
11. Assume the statement  $IPEN = 1$  is changed to  $IPEN = 0$  in Listing 10.2. Give the order in which the statements A, B, C, D are executed, if they are executed at all, when the pushbutton is pressed and released one time.
12. Assume the statement  $INTEDG2 = 0$  is changed to  $INTEDG2 = 1$  in Listing 10.2. Give the order in which the statements A, B, C, D are executed, if they are executed at all, when the pushbutton is pressed and released one time.
13. Assume the statement  $INTEDG1 = 0$  is changed to  $INTEDG1 = 1$  in Listing 10.2. Give the order in which the statements A, B, C, D are executed, if they are executed at all, when the pushbutton is pressed and released one time.
14. Assume the statement  $INT2IP = 0$  is changed to  $INT2IP = 1$  in Listing 10.2. Give the order in which the statements A, B, C, D are executed, if they are executed at all, when the pushbutton is pressed and released one time.
15. Assume the statement  $INT1IP = 1$  is changed to  $INT1IP = 0$  in Listing 10.2. Give the order in which the statements A, B, C, D are executed, if they are executed at all, when the pushbutton is pressed and released one time.
16. Assume the statement  $INT1IE = 1$  is changed to  $INT1IE = 0$  in Listing 10.2. Give the order in which the statements A, B, C, D are executed, if they are executed at all, when the pushbutton is pressed and released one time.
17. Assume the statement  $INT2IE = 1$  is changed to  $INT2IE = 0$  in Listing 10.2. Give the order in which the statements A, B, C, D are executed, if they are executed at all, when the pushbutton is pressed and released one time.
18. Assume the statement  $PEIE = 1$  is changed to  $PEIE = 0$  in Listing 10.2. Give the order in which the statements A, B, C, D are executed, if they are executed at all, when the pushbutton is pressed and released one time.
19. Assume the statement  $GIE = 1$  is changed to  $GIE = 0$  in Listing 10.2. Give the order in which the statements A, B, C, D are executed, if they are executed at all, when the pushbutton is pressed and released one time.
20. Assume the code changes of problems 18 and 15 are both made. Give the order in which the statements A, B, C, D are executed, if they are executed at all, when the pushbutton is pressed and released one time.

Answer the following questions:

21. Give the PR2, POST, and PRE values for a Timer2 interrupt that generates an interrupt at a 2 kHz rate assuming  $FOSC = 10$  MHz. For a PRE value of your choice, use the POST value that gives the largest possible PR2 value.

22. Give the PR2, POST, and PRE values for a Timer2 interrupt that generates an interrupt at a 3.5 kHz rate assuming  $F_{OSC} = 8 \text{ MHz}$ . Use the POST value that gives the largest possible PR2 value.
23. Assume a Timer2 interrupt is using  $POST = 5$ ,  $PRE = 4$ , and  $PR2 = 63$ . How can these values be changed to generate an interrupt with a period that is 10 times longer?
24. Assume a Timer2 interrupt is using  $POST = 5$ ,  $PRE = 4$ , and  $PR2 = 40$ . If these values are changed to  $POST = 10$ ,  $PRE = 16$ , and  $PR2 = 120$  what is the relationship between the original interrupt period and the new interrupt period?
25. Assuming  $F_{OSC} = 25 \text{ MHz}$ , what is the longest interrupt interval that can be generated using Timer2?
26. Modify the keypad code to add an input software FIFO that has room for eight key values.
27. Modify the `update_state()` function of the rotary encoder code of Figure 10.26 such the count value is limited between `min` and `max` variable values.
28. Assume a low-true pushbutton input on RB0, and four high-true LEDs connected to pins RB4 through RB7. Write C code that configures PORTB for this operation, with the LEDs initially off. Then enter a loop, where each press and release of the switch turns the LEDs on in sequence, with the LED previously on being turned off (i.e., 1<sup>st</sup> press/release, RB4 is 1/RB7 is 0, 2<sup>nd</sup> press/release RB5 is 1/RB4 is 0, 3<sup>rd</sup> press/release RB6 is 1/RB5 is 0, 4<sup>th</sup> press/release RB7 is 1/RB6 is 0, repeat). Use an interrupt-driven approach for reading the pushbutton and use the debouncing approach of Figure 10.23 and Figure 10.24.

*This page intentionally left blank*

# 11

# Synchronous Serial IO

## In This Chapter

- The PIC18 and Synchronous Serial IO
- USART Synchronous Mode
- The Serial Peripheral Interface (SPI)
- SPI Examples: A Digital Potentiometer and a Serial EEPROM
- The I<sup>2</sup>C Bus
- I<sup>2</sup>C on the PIC18Fxx2
- The 24LC515 Serial EEPROM
- Double Buffering for Interrupt-Driven Writes

This chapter discusses synchronous serial IO as implemented on the PIC18. The Serial Peripheral Interface (SPI) and I<sup>2</sup>C (Inter IC) bus protocols are covered, with sample interfaces to a digital potentiometer and serial EEPROMs.

## 11.1 LEARNING OBJECTIVES

---

After reading this chapter, you will be able to:

- Compare and contrast synchronous serial IO on the PIC18 using the SPI and I<sup>2</sup>C protocols.
- Discuss the uses of digital potentiometers and serial EEPROMs in PIC18 applications.

- Interface a PIC18 to a serial EEPROM using the SPI protocol.
- Interface a PIC18 to a digital potentiometer using the SPI protocol.
- Interface a PIC18 to a serial EEPROM using the I<sup>2</sup>C protocol.
- Use interrupt-driven double buffering to implement continuous data stream applications.

## 11.2 THE PIC18 AND SYNCHRONOUS SERIAL IO

In Chapter 9, “Asynchronous Serial IO,” we discussed three methods for synchronizing a receiver to a serial data stream. The simplest method sends the clock as a separate signal, which is the mechanism used by the subsystems on the PIC18 that support synchronous serial IO. PIC18 synchronous serial IO options are summarized in Table 11.1. The USART subsystem was previously discussed in Chapter 9 for asynchronous IO; when used in synchronous mode the TX and RX pins are used for clock (CK) and data pins (DT), respectively.

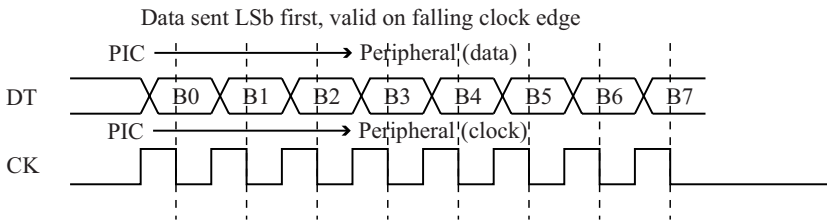
**TABLE 11.1** PIC18 Synchronous Serial IO Summary

Subsystem	Protocol	Classification	Pins (18F242)
USART	n/a	Half-duplex	RC6/TX/ <b>CK</b> (clock), RC7/RX/ <b>DT</b> (data)
MSSP	SPI	Duplex	RC3/ <b>SCK</b> /SCL (clock), RC4/ <b>SDI</b> /SDA (data in), RC5/SDO (data out)
MSSP	I2C	Half-duplex	RC3/SCK/ <b>SCL</b> (clock), RC4/ <b>SDI</b> / <b>SDA</b> (data)

The Master Synchronous Serial Port (MSSP) subsystem supports two industry-standard serial protocols, namely the Serial Peripheral Interface (SPI) and Inter IC (I<sup>2</sup>C) bus. All three mechanisms support both a *master* mode and a *slave* mode. In master mode, the PIC18 supplies the clock for all transactions, while in slave mode the external peripheral supplies the clock. Slave mode is useful for waking the processor from sleep mode via incoming serial data, but requires an intelligent peripheral capable of initiating data transfers. In this chapter, all synchronous serial IO examples use master mode; in other words, the PIC18 initiates all data transfers whether it be transmit or receive.

## 11.3 USART SYNCHRONOUS MODE

The synchronous mode of the USART subsystem is the simplest of the three synchronous serial protocols available on the PIC18. Figure 11.1 shows a timing diagram of USART synchronous transmit, which from a coding perspective is done in the same way as asynchronous transmit. If the TXIF bit is clear, TXREG can accept new data and a write to TXREG triggers synchronous transmit if TXEN (transmit enable) is set. Data is shifted out LSb first and is valid on the falling clock edge. There is no hardware method for changing the bit ordering to MSb first; the bit reversal would have to be done in software before the value is written to TXREG.



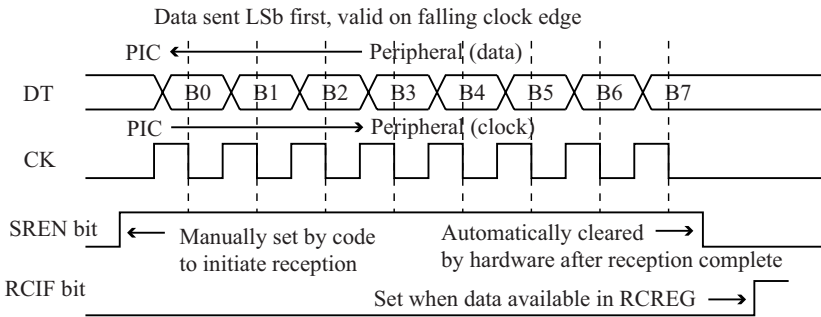
Transmit triggered by write to TXREG if TXEN = 1, or by setting TXEN = 1 if write to TXREG has already been done. TXIF bit is 1 if TXREG can accept new data.

**FIGURE 11.1** USART synchronous transmit.

The timing for synchronous reception, single byte receive is seen in Figure 11.2. Because this is master mode, reception is initiated by writing a “1” to the SREN (Single Receive Enable) bit. This causes eight clock pulses (or nine, if RX9 = 1) to be generated, with data clocked in on each falling edge. It is assumed that some external peripheral is actively driving the DT pin with new data on each clock pulse. The RCIF bit is set once all bits have been received. Continuous clock pulses are sent if the CREN (Continuous Receive Enable) bit is set; clearing CREN halts continuous reception. Observe that the asynchronous reception used earlier automatically inputs any received data on the RX pin, which is significantly different from this mode in which the PIC has to “ask” the peripheral to provide data by sending clock pulses to the peripheral.

Equation 11.1 gives the baud rate calculation for synchronous mode. The BRGH bit in asynchronous mode that selected low or high speed mode has no effect in this case. As with asynchronous transmission, the SPBRG register value in Equation 11.1 sets the baud rate. Because this is synchronous transmission where the clock is sent with the data, there are no “pre-defined” baud rates with synchronous peripherals as there are with asynchronous transfer. Instead, synchronous





**FIGURE 11.2** USART synchronous receive.

peripherals accept a wide range of input clock frequencies, with typically only a maximum value specified in the data sheet.

$$BR = \frac{FOSC}{(4 * (SPBRG + 1))} \tag{11.1}$$

Table 11.2 gives configuration bit settings for synchronous serial transmission. Even though the RC7/RX/DT pin is bidirectional (outgoing on data transmission, incoming on data reception), the TRISC[7] bit setting should not be changed during transmission because an internal multiplexer is used to select the USART TSR register as the data source of the output pin instead of the port data latch.

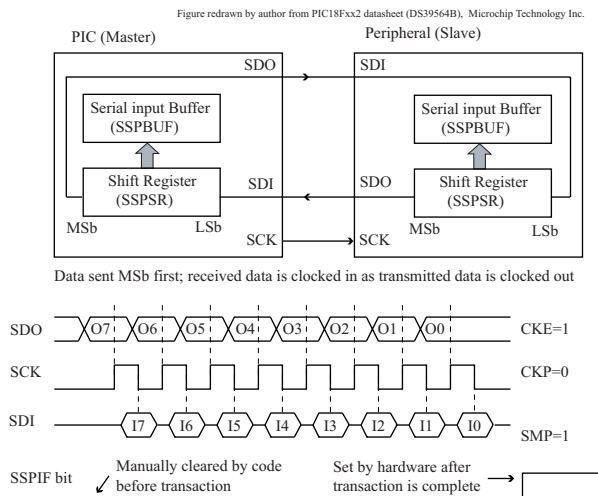
**TABLE 11.2** Control Registers/Bits for USART Synchronous Configuration

Name	SFR(bit)	Comment
SPBRG	n/a	This register contains the baud rate divisor.
CSRC	TXSTA[7]	If "1", synchronous master mode (selects BRG as clock source); else "0" for synchronous slave mode. This bit is ignored in asynchronous mode.
TX9	TXSTA[6]	If "1", 9-bit transmission; else 8-bit transmission.
TXEN	TXSTA[5]	If "1", transmit is enabled; else is disabled.
SYNC	TXSTA[4]	"1" for UART synchronous mode.
RX9	RCSTA[6]	If "1", 9-bit reception; else 8-bit reception.
SREN	RCSTA[5]	Set to "1" to enable single synchronous receive.
CREN	RCSTA[4]	"1" to enable continuous synchronous receive.
SPEN	RCSTA[7]	Must be "1" to configure CK/DT as serial port. →

TRISC6	TRISC[6]	Must be "0" so that RC6/TX/CK pin is an output.
TRISC7	TRISC[7]	Must be "1" so that RC7/RX/DT pin is an input.

## 11.4 THE SERIAL PERIPHERAL INTERFACE (SPI)

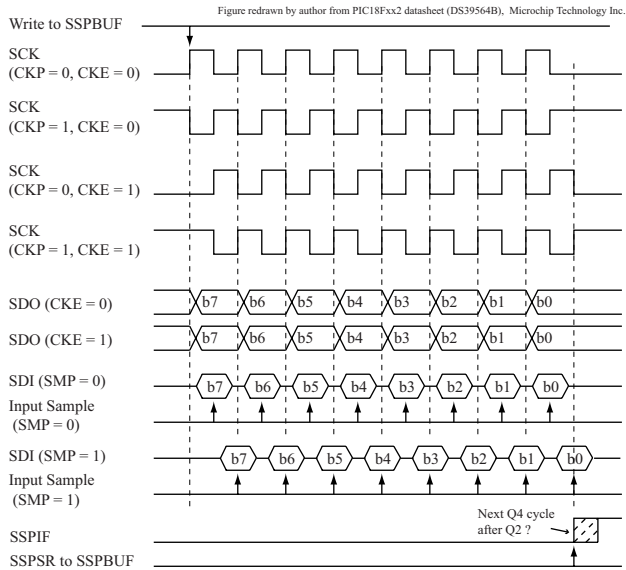
The Serial Peripheral Interface, originally developed by Motorola, is a three-wire synchronous serial link that has developed into a de facto standard due to its adoption by multiple semiconductor vendors. Figure 11.3 shows an SPI connection between a PIC and a peripheral device. An SPI port achieves full-duplex communication by shifting in data via the serial data input (SDI) pin while shifting out data through the serial data output (SDO) pin. In master mode, the PIC initiates all transactions by supplying the clock via the SCK pin. Observe that unlike the USART synchronous transmission, data is sent MSb first. Data is written to the SSPBUF register to initiate either transmit or receive. For receive (PIC from peripheral) operation, dummy data is written to SSPBUF if the peripheral device does not care about incoming data on its SDI pin. For transmit (PIC to peripheral) operation, the PIC can ignore the new data shifted into the SSPBUF register if no valid data is expected. The SSPIF (Master Synchronous Serial Port Interrupt Flag, PIR1[3]) is automatically set when a transaction is complete; it must be manually reset before the next transaction is initiated.



**FIGURE 11.3** Serial peripheral interface.<sup>1</sup>

<sup>1</sup> Figure 11.3 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.'s prior written consent.

Configuration bits CKE (clock edge select, SSPSTAT[6]), CKP (clock polarity select, SSPCON1[4]), and SMP (input sample select, SSPSTAT[7]) provide considerable flexibility for data transmit and receive. The CKE and CKP bits are used for transmit; CKE selects the active clock edge for SDO valid data while CKP selects the clock polarity, either idle high or idle low. Figure 11.4 shows the four cases for the CKE and CKP bit settings. Observe that for CKP = 0 (clock idle low), CKE = 0 has SDO stable on the falling clock edge, while CKE = 1 provides valid SDO data on the rising clock edge. For CKP = 1 (clock idle high) this is reversed, with CKE = 0 providing stable SDO data on the rising clock edge and CKE = 1 makes SDO valid on the falling clock edge. The SMP bit determines where the SDI input is sampled during receive, either in the middle of the SCK period (SMP = 0) or at the end of the SCK period (SMP = 1) as shown in Figure 11.4. The required settings for the CKE, CKP, and SMP bits depend upon the target peripheral.



**FIGURE 11.4** CKE/CKP/SMP cases for SPI transmission.<sup>2</sup>

The SCK frequency is controlled by the SSPM (Synchronous Serial Port Mode select, SSPCON1[3:0]) bits. The four choices for master mode are “0011” (Timer2 output divided by 2), “0010” (FOSC/64), “0001” (FOSC/16), and “0000”

<sup>2</sup> Figure 11.4 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.’s prior written consent.

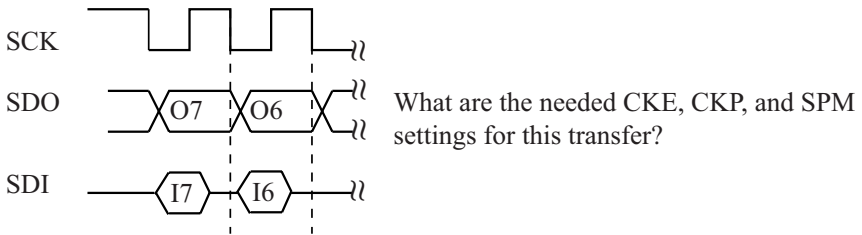
(FOSC/4). Table 11.3 summarizes the configuration bits used for SPI mode transfers. Observe that SCK, SDI, and SDO are shared with the PORTC pins and that TRISC must be used to configure these pins as inputs or outputs as shown in Table 11.3.

**TABLE 11.3** Control Registers/Bits for SPI Master Mode Configuration

Name	SFR(bit)	Comment
SSPEN	SSPCON1[5]	Must be "1" to enable SCK, SDO, SDI pins
SSPM[3:]	SSPCON1[3:0]	"0011", SPI Master Mode, SCK= TMR2/2 "0010", SPI Master Mode, SCK= FOSC/64 "0001", SPI Master Mode, SCK = FOSC/16 "0000", SPI Master Mode, SCK = FOSC/4
CKE	SSPSTAT[6]	For CKP = 0: "1": SDO valid on rising SCK edge "0": SDO valid on falling SCK edge For CKP = 1: "1": SDO valid on falling SCK edge "0": SDO valid on rising SCK edge
CKP	SSPCON1[4]	"1": SCK idle high, "0": SCK idle low
SMP	SSPSTAT[7]	"1": sample SDI at end of SCK in master mode "0": sample SDI in middle of SCK in master mode (must be a "0" in slave mode)
SSPIF	PIR1[3]	Set to "1" after transmission complete
TRISC3	TRISC[3]	Must be "0" so that RC3/SCK/SCL pin is an output
TRISC4	TRISC[4]	Must be "1" so that RC4/SDI/SDA pin is an input
TRISC5	TRISC[5]	Must be "0" so that RC5/SDO pin is an output

**Sample Question:** What are the required settings for CKP, CKE, and SMP for the SPI waveform specification shown in Figure 11.5?

**Answer:** The clock is idle high, so  $CKP = 1$ . Output data is stable on the rising clock edge, so  $CKE = 0$  by Figure 11.4. Data is sampled in the middle of SCK, so  $SMP = 0$ .



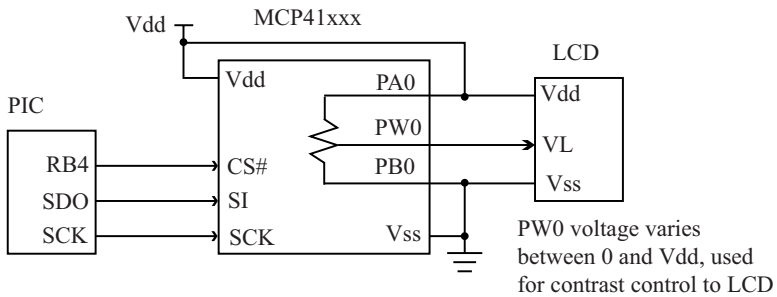
**FIGURE 11.5** A sample SPI waveform specification.

## 11.5 SPI EXAMPLES: A DIGITAL POTENTIOMETER AND A SERIAL EEPROM

Many peripheral devices such as analog-to-digital converters, digital-to-analog converters, digital potentiometers, and serial EEPROMs are available with SPI-compatible interfaces. As discussed previously, the advantage of a serial interface is low pin count at the cost of reduced IO bandwidth.

### The MCP41xxx Digital Potentiometer

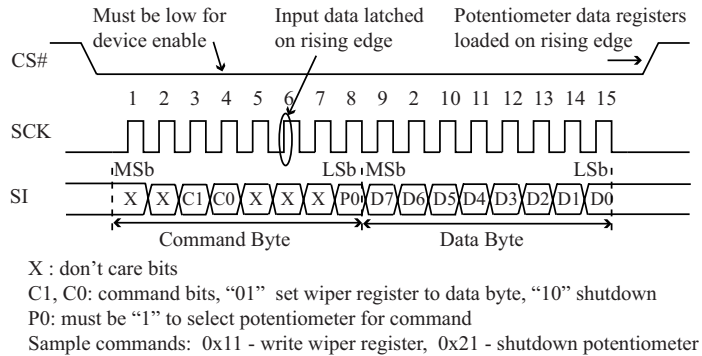
Figure 11.6 shows an application of a MCP41xxx digital potentiometer [12] as a contrast control for the LCD module discussed in Chapter 8, “The PIC18Fxx2: System Startup and Parallel Port IO.” A potentiometer is a device that provides a variable resistance. An analog potentiometer typically has three terminals; between two of the terminals the potentiometer’s full resistance is available (reference terminals PA0, PB0 in Figure 11.6). The third terminal is called the *wiper* (terminal PW0 in Figure 11.6), and this terminal provides a variable resistance when measured between the wiper and either one of the reference terminals. When the two reference terminals are connected to Vdd and ground, changing the wiper setting varies the voltage on the wiper terminal between Vdd and ground. An analog potentiometer’s wiper setting is changed via some mechanical interface; for example, turning a shaft. A digital potentiometer’s wiper setting is changed using a parallel or serial interface, with serial interfaces being the most common.



**FIGURE 11.6** PIC to MCP41xxx digital potentiometer interface.

The MCP41xxx digital potentiometer comes in 10 K (MCP41010), 50 K (MCP41050), and 100 K (MCP41100) configurations and uses an SPI port for setting the 8-bit wiper register for the potentiometer. In the configuration shown in Figure 11.6, a wiper value of 255 sets the PW0 output voltage to approximately  $255/256 * V_{dd}$ , while a value of 0 sets the PW0 output voltage to ground. The wiper register is set to 0x80 on power-up. Higher potentiometer values reduce the static current that is drawn by the potentiometer when it is active. For example, a 50 K potentiometer with  $V_{dd} = 5 \text{ V}$  draws  $5 \text{ V}/50 \text{ K} = 100 \mu\text{A}$  static current through the potentiometer resistance, while a 100 K potentiometer reduces this current by 50% to  $50 \mu\text{A}$ .

Figure 11.7 shows the command protocol for the MCP41xxx. Each transaction consists of 2 bytes, a command byte and a data byte. The CS# (Chip Select) input must be brought low to enable the device before any data is sent and brought high after transmission is finished in order to execute the command. The wiper register is set by the command byte 0x11 followed by the wiper register value. The shutdown command opens (disconnects) the potentiometer by opening the PA0 terminal and shorting the PW0 and PB terminals. This reduces total static current draw of the MCP41xxx to less than  $1 \mu\text{A}$ . The data byte for the shutdown command is ignored but it still must be sent for the command to be recognized. If MCP41xxx shutdown mode were to be used with the LCD application of Figure 11.6, you would want to reverse the PA0 and PB0 connections so that VL of the LCD is shorted to Vdd during shutdown, blanking the display. This would mean that a wiper code of 255 sets the PW0 voltage to near ground, while a code of 0 sets the PW0 voltage to Vdd.



**FIGURE 11.7** MCP41xxx command protocol.

Figure 11.8 gives code for testing the PIC to MCP41xxx interface. The `while(1){}` loop of `main()` prompts the user for an 8-bit value and sends this as the wiper register value to the MCP41xxx via the `spi_setpotmtr(unsigned char c){}` function. Within the `spi_setpotmtr()` function, the chip select of the MCP41xxx is brought low by the command `bitclr(PORTB, POTCS)` statement, where `POTCS` is defined as 4. This is equivalent to writing `RB4 = 0`, but the `bitclr` macro is used so that changing to a different `PORTB` pin for chip select only requires modifying the `#define POTCS 4` statement. After the chip select is asserted, the command byte (0x11) is written followed by the data byte passed to the function in the `c` parameter. Observe that after a byte is written to `SSPBUF`, the `while(!SSPIF)` loop waits for the `SSPIF` to become nonzero, indicating that the transmission is finished. The statement `SSPIF = 0` is then used to manually reset the `SSPIF` bit before the next transmission. The MCP41xxx chip select is negated by the `bitset(PORTB, POTCS)` statement before exiting `spi_setpotmtr()`. The SPI initialization code in `main()` uses a positive clock polarity (`CKP = 0`) and data transmitted on the rising edge (`CKE = 1`), as that matches the SPI specifications in the MCP41xxx datasheet. The `SCK` frequency of `FOSC/16` gives an `SCK` of approximately 1.8 MHz for the 29.4912 MHz `FOSC` of the PIC18F242 reference board. This `SCK` frequency is safely below the maximum 10 MHz `SCK` frequency of the MCP41050 device used for testing.

```

#include <pic18.h>
#include "config.h"
#include "serial.c"
#include "serio.c"
} Include files for configuration bits and
  asynchronous serial port IO

//RB4 is select for potentiometer
#define POTCS 4
} Function for setting potentiometer wiper register

spi_setpotmtr(unsigned char c){
    bitclr(PORTB, POTCS); // select potmtr ← Assert Chip Select
    SSPBUF = 0x11; // write command } Write command byte, wait for
    while(!SSPIF); // wait until transmitted } transmit to end, then reset SSPIF
    SSPIF = 0; // reset
    SSPBUF = c; // write data } Write data byte, wait for
    while(!SSPIF); // wait until transmitted } transmit to end, then reset SSPIF
    SSPIF = 0; // reset
    bitset(PORTB, POTCS); // deselect potmtr ← Negate Chip Select
}

main(void){
    unsigned char pv;

    // set select line for output
    bitclr(TRISB, POTCS);
    bitset(PORTB, POTCS); // deselect pot } Configure RB4 as an output, ensure
    } that it is high, deselecting MCP41xxx

    serial_init(95,1); // 19200 in HSPLL mode, crystal = 7.3728 MHz

    // configure SPI port for potentiometer
    CKE = 1; // data transmitted rising edge of SCK } Configure SPI port.
    CKP = 0; // clk idle is low } Must use CKE=1, CKP=0
    bitclr(TRISC,3); //SCK, output } as that is compatible with
    bitclr(TRISC,5); // SDO, output } datasheet specs for
    bitset(TRISC,4); // SDI pin is input, unused } MCP41xxx.
    // SPI Master Mode FOSC/16 } Use FOSC/16, sets SCK
    SSPM3 = 0; SSPM2 = 0; SSPM1 = 0; SSPM0 = 1; } as approx. 1.8 Mhz for
    SSPEN = 0; // reset Sync Serial port } 29.49 MHz FOSC
    SSPEN = 1; // enable Sync Serial port
    SSPIF = 0; // clear SPIF bit

    pcrf(); printf("Potentiometer test started"); pcrf();
    while(1) {
        printf("Input value (0-255): ");
        scanf("%d", &pv);
        pcrf();
        printf("Sending %d to pot.",pv);
        pcrf();
        spi_setpotmtr(pv);
    }
}
} Prompt user for 8-bit input value,
  send to potentiometer and
  use voltmeter to check
  potentiometer output value.

```

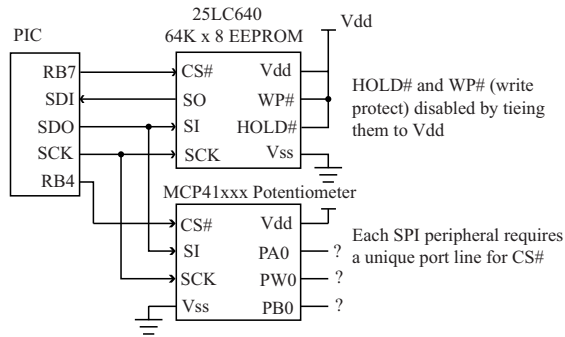


**FIGURE 11.8** Test code for PIC to MCP41xxx interface.

## The 25LC640 Serial EEPROM

Figure 11.9 shows a PIC18 to 25LC640 serial EEPROM [13] interface. The 25LC640 is a 64 Kb serial EEPROM with an internal 8K x 8 organization and uses an SPI port for communication. The HOLD# input allows a data transfer to be interrupted mid-stream and the WP# input disables write operations to the device. These capabilities are not needed in this example, so these pins are tied high to disable them.

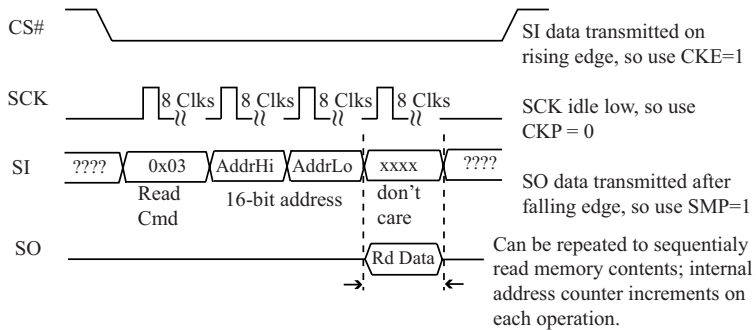




**FIGURE 11.9** PIC to 25LC640 serial EEPROM interface.

A serial EEPROM is useful as nonvolatile data storage when the 256 bytes of on-chip data EEPROM available in the PIC18 is not sufficient. The MCP41xxx from the previous example is included in Figure 11.9 to illustrate how multiple SPI peripherals coexist in the same system. Observe that the PIC18 uses RB7 as the EEPROM chip select, while RB4 is used as the potentiometer chip select. It is important that only one SPI peripheral be enabled at a time via its chip select to avoid confusion over which peripheral is being accessed. Without using external logic, each additional SPI peripheral requires a unique port line for chip select control.

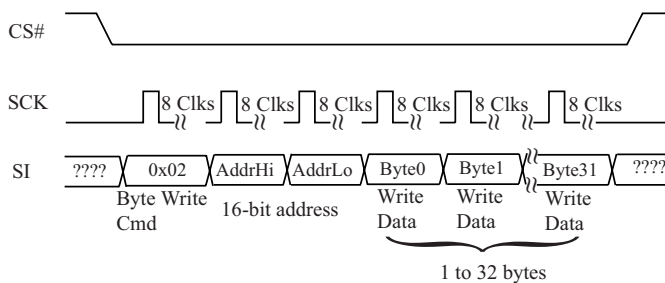
Figure 11.10 gives the read operation details for the 25LC640. The first byte sent is the command byte, which has a value of 0x03.



**FIGURE 11.10** 25LC640 read operation.

The next 2 bytes are the most significant byte and least significant byte of the 16-bit address. Because the 25LC640 has only 8K locations ( $2^{13}$ ), the upper 3 bits of the address MSB are ignored. This address value is loaded into an internal address counter within the 25LC640. The next SPI operation returns the contents of this address via the SO data line (the input data on the SI line is ignored by the 25LC640). This also increments the internal address counter. At this point, the operation can be terminated by negating CS# or another SPI operation will return the byte at the successive memory location. The entire memory contents of the 25LC640 can be sequentially read in this manner without sending another address value. The internal address counter rolls over to 0x0000 once the value 0x1FFF ( $2^{13} - 1$ , or 8191) is reached. As with all SPI transmissions, data in each byte is transmitted MSb first. The 25LC640 inputs data on the rising edge of SCK with output data produced after the SCK falling edge. PIC18 configuration bits of CKP = 0 (SCK idle low), CKE = 1 (output data stable on rising edge), and SMP = 1 (input data sampled at end of the SCK period) satisfy the SPI communication requirements of the 25LC640.

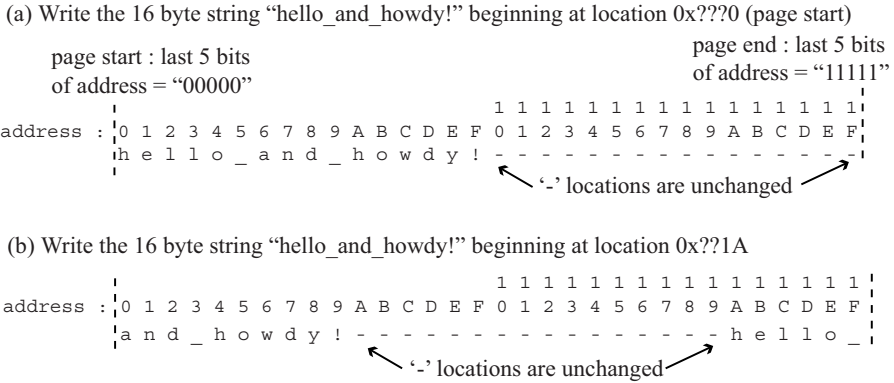
Figure 11.11 shows the write operation sequence for the 25LC640, which consists of a write command byte (0x2), the 16-bit address, and up to 32 data bytes. The write operation is triggered after CS# is negated and has a worst-case completion time of 5 ms. Once the bytes have been received by the serial EEPROM, it takes the same amount of time to write 1 byte as 32 bytes, so it is more efficient to write as many bytes at a time as possible.



**FIGURE 11.11** 25LC640 write operation.

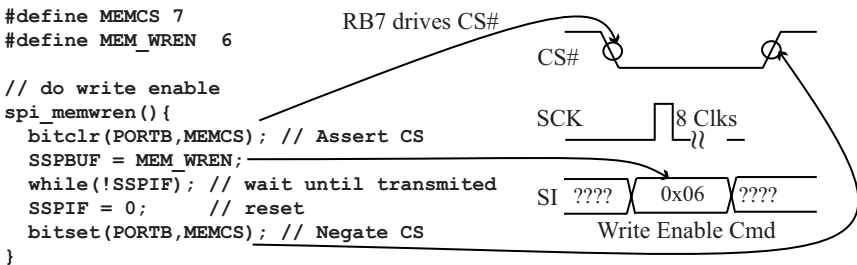
The incoming bytes are placed into an internal 32-byte write buffer using the lower 5 bits of the address, while the upper 8 bits of the address determine the memory page that is written. The starting address of a 32-byte page has its last 5 bits as all zeros, while the ending address has all ones for the last 5 bits. If the write command address does not begin on a 32-byte page boundary, multiple bytes sent in the write command may cause page wrapping to occur if the internal address

counter increments past a page boundary as shown in Figure 11.12b. In most applications, either a single byte is written and thus page wrapping cannot occur, or the starting address is forced to align to a page boundary and a complete page of 32 bytes is written.



**FIGURE 11.12** Page boundary wrapping during write.

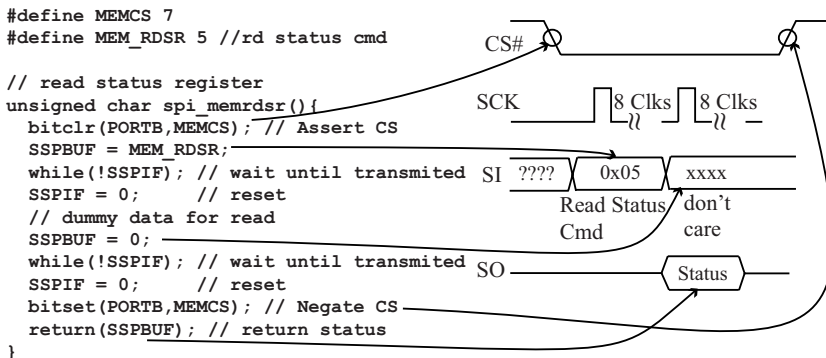
An internal write enable latch must be set before any write command is issued. Figure 11.13 shows a C function named `spi_memwren()` that accomplishes this by writing the command byte 0x06 to the 25LC640. The write enable latch is reset after a write operation is complete so this function must be called before every write.



**FIGURE 11.13** C code for setting the write enable latch (see CD-ROM file `./code/chap11/F_11_16_spimemtst.c`).

The write operation is internally self-timed, which means that the internal programming of the target locations continues until the device detects that the write has successfully completed. The 5 ms write completion time is a worst-case time; placing a software delay loop after the write operation could satisfy this constraint.

However, this is a waste of processor clock cycles and the write can actually complete much sooner than that. An internal status register provides a bit named WIP (write in progress, STATUS[0]) that is a “1” while the write operation is underway. Before starting a write operation, this bit can be polled via the *read status* command. Figure 11.14 shows a C function named `spi_memrdsr()` that reads the 25LC640 status register by sending the command byte 0x05 followed by a dummy byte that causes the status register byte to be returned to the PIC via the EEPROM SO output.



**FIGURE 11.14** C Code for reading the EEPROM status register (see CD-ROM file `./code/chap11/F_11_16_spimemtst.c`).

Both the `spi_memrdsr()` and `spi_memwren()` functions are used by the `spi_memrw()` function of Figure 11.15 that reads or writes 32 bytes to the serial EEPROM. For a write operation, the `write_flag` parameter is nonzero and the 32 bytes contained in `buf` are written to the starting location specified by `addr`. Before beginning a read or write operation, a `do-while{}`  loop polls the EEPROM status register via the `spi_memrdsr()` function and loops while the write-in-progress bit (`status[0]`) is one, indicating a previous write is still underway. Once the write-in-progress bit returns as zero, the loop exits and the `spi_memwren()` function is called to set the write enable latch. The 16-bit address is split into high byte and low byte by the statements `addr_hi = addr >> 8` and `addr_lo = addr & 0x00FF`, respectively. The write operation begins by asserting the chip select line via `bitclr(PORTB, MEMCS)`. The write command byte (0x02) is then sent, followed by the high address byte, and then the low address byte. A `for{}`  loop then sends the 32 bytes stored in the `buf` parameter. The `bitset(PORTB, MEMCS)` statement terminates the write operation by negating the chip select.

```

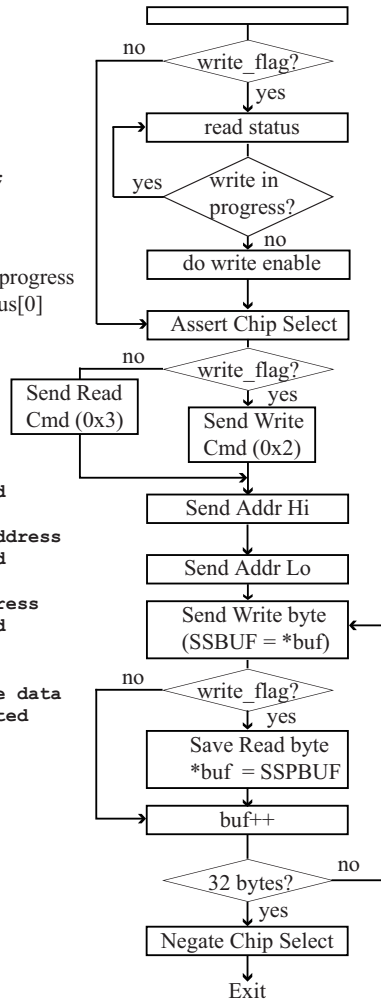
#define MEMCS 7
#define MEM_WRITE 2
#define MEM_READ 3

//write or read 32 bytes
spi_memrw(char *buf, unsigned int addr,
char write_flag){
    unsigned char addr_lo, addr_hi, status;
    char i;

    // ensure last write is finished
    do {
        status = spi_memrdr();
    }while (bittst(status,0));
    // do write enable if write
    if (write_flag) spi_memwren();
    addr_lo = addr & 0x00FF;
    addr_hi = (addr >> 8);
    bitclr(PORTB, MEMCS); // Assert CS
    if (write_flag) SSPBUF = MEM_WRITE;
    else SSPBUF = MEM_READ;
    while(!SSPIF); // wait until transmitted
    SSPIF = 0; // reset
    SSPBUF = addr_hi; // send high byte address
    while(!SSPIF); // wait until transmitted
    SSPIF = 0; // reset
    SSPBUF = addr_lo; // send low byte address
    while(!SSPIF); // wait until transmitted
    SSPIF = 0; // reset
    for (i=0;i<32;i++){ // send 32 bytes
        SSPBUF = *buf; //for read, don't care data
        while(!SSPIF); // wait until transmitted
        SSPIF = 0; // reset
        if (!write_flag) *buf = SSPBUF;
        buf++;
    }
    bitset(PORTB, MEMCS); // Negate CS
}

```

Write-in-progress bit is status[0]



**FIGURE 11.15** C code for reading/writing 32 bytes from/to the serial EEPROM (see CD-ROM file ./code/chap11/F\_11\_16\_spimemst.c).

For a read operation, the `write_flag` parameter is zero and 32 bytes are read from the EEPROM starting at location `addr` and written to the `buf` array during the `for{} loop`. The read command byte `0x3` precedes the high and low address bytes that specify the starting address for the read operation. There are no page boundaries for read operations and thus the entire EEPROM contents could be returned in one call to `spi_memrw()` if there was enough room in `buf` to store these values. Of course, there is not enough internal storage on the PIC18 to hold this many bytes, so a buffer size of 32 bytes was chosen for read to match the write page buffer size.

Figure 11.16 gives the C code for `main()` that uses the `spi_memrw()` function to test the serial EEPROM. After configuring the SPI port, the user is prompted to choose either write or read mode. In write mode, a loop prompts the user to enter 32 characters that are then written to the serial EEPROM. On each pass through the loop, the `memaddr` variable that specifies the starting location is incremented by 32 so that writes are performed to successive pages. In read mode, a 32-byte page is read and displayed each time the user enters a character, starting from location 0.

```

char membuf[32];
int memaddr;
main(void) {
    unsigned char mode,i;

    // set RB7 for output
    TRISB = 0x7F;
    bitset(PORTB,MEMCS);
    serial_init(95,1); // 19200 in HSPLL mode, crystal = 7.3728 MHz

    // configure SPI port for serial eeprom
    CKE = 1; // data transmitted rising edge of SCK
    CKP = 0; // clk idle is low
    SMP = 1; // sample data at end of period
    bitclr(TRISC,3); //SCK, output
    bitclr(TRISC,5); // SDO, output
    bitset(TRISC,4); // SDI pin is input
    // SPI Master Mode FOSC/16
    SSPM3 = 0;  SSPM2 = 0;  SSPM1 = 0;  SSPM0 = 1;
    SSPEN = 0;
    SSPEM = 1;
    SSPIF = 0; // clear SPIF bit
    pcrLf(); printf("Mem Test Started"); pcrLf();

    memaddr = 0;
    printf ("Enter 'w' for write mode, anything else reads: ");
    mode = getche();
    pcrLf();
    while(1) {
        if (mode == 'w') {
            printf("Enter 32 chars.");pcrLf();
            for(i = 0;i< 32;i++) {
                membuf[i] = getche();
            }
            pcrLf();printf("Doing Write");pcrLf();
            spi_memrw(membuf,memaddr,1); // do write
            memaddr = memaddr +32;
        } else {
            spi_memrw(membuf,memaddr,0); // do read
            for(i = 0;i< 32;i++) putchar(membuf[i]);
            pcrLf();
            printf("Any key continues read...");pcrLf();
            getch();
            memaddr = memaddr +32;
        }
    }
}

```

SPI Port configured for 25LC640 serial EEPROM operation.

These are the same settings as were used for the MCP41xxx digital potentiometer

Prompt user to enter 32 bytes, capture them, then write them to the serial EEPROM. Increment memory address by 32 for next write.

Read 32 bytes from the serial EEPROM, then display them. Repeat on each keypress.



**FIGURE 11.16** `main()` for testing PIC18 to serial EEPROM interface.

Figure 11.17 shows console output from testing the code of Figure 11.16. Three 32-byte strings are entered and written to the first three pages of serial EEPROM memory. Reset is asserted to terminate the write mode and then these strings are retrieved from the serial EEPROM and displayed.

```

Mem Test Started
Enter 'w' for write mode, anything else reads: w
Enter 32 chars.
I like stone walls, do you?      ← 1st 32 bytes
Doing Write
Enter 32 chars.
If you don't, it is ok...I guess ← 2nd 32 bytes
Doing Write
Enter 32 chars.
But please reconsider. SW&EVR!!! ← 3rd 32 bytes
Doing Write
Enter 32 chars.
                                     ← Pressed reset

Mem Test Started
Enter 'w' for write mode, anything else reads:
I like stone walls, do you?      ← 1st 32 bytes
Any key continues read...
If you don't, it is ok...I guess ← 2nd 32 bytes
Any key continues read...
But please reconsider. SW&EVR!!! ← 3rd 32 bytes
Any key continues read...

```

**FIGURE 11.17** Console output from testing code of Figure 11.16.

The 25LC640 serial EEPROM has additional capabilities that are not covered here, such as being able to write protect blocks of internal memory; see the datasheet [13] for details.

**Sample Question:** How long does it take to transfer the data required for a page write to the 25LC640 serial EEPROM assuming an FOSC = 40 MHz and a conservative 20 instruction overhead for every byte written over the SPI interface? Choose the highest SCK frequency possible for the EEPROM that still meets the 25LC640 maximum SCK specification of 3 MHz@ 5 V, industrial temperature range (do not use Timer2 to generate the SCK).

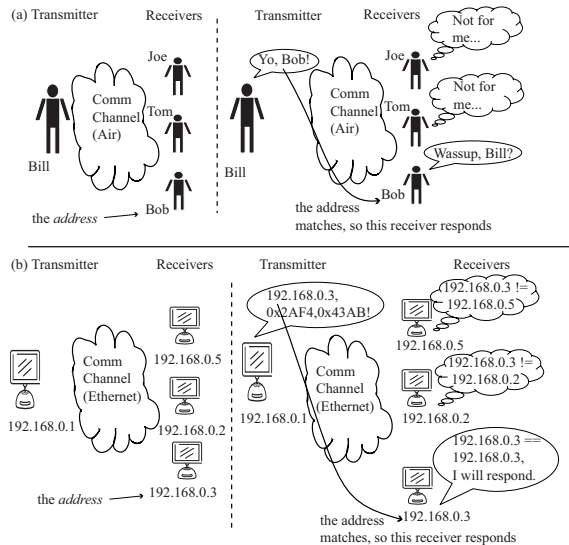
*Answer:* An SCK of FOSC/16 gives 2.5 MHz, which is less than the 3 MHz maximum for SCK. For FOSC = 40 MHz, each instruction takes 4/40 MHz = 0.1 μs. A SCK bit time is 1/2.5 MHz = 0.4 μs. The bytes sent are the byte write command, MSB EEPROM address, LSB EEPROM address, and 32 data bytes for a total of 35 bytes. Total bit times are 35\*8 = 280 bit times. The estimated time for the data transfer is:

$$35 \text{ bytes} * 20 \text{ instructions} * \text{instruction time} + 280 \text{ bits} * \text{SCK bit time}$$

$$35 * 20 * 0.1 \mu\text{s} + 280 * 0.4 \mu\text{s} = 182 \mu\text{s}.$$

## 11.6 THE I<sup>2</sup>C BUS

The Inter IC (I<sup>2</sup>C) bus [14] was introduced by Philips Semiconductor in the early 1980s and it has since become a de facto standard serial bus. The term *bus* in this context is a formal designation and is different from the previous casual usage of “bus” to describe groups of parallel wires. In this context, a *bus* is a communication channel in which there is one transmitter and multiple receivers. All receivers see data that is transmitted over the communication channel. Each receiver decodes transmitted messages and uses an *address* within the message to determine if it is the target of the message. The receiver that is the message target then replies back to the message transmitter over the same communication channel. Figure 11.18 gives two examples of bus communication channels. Figure 11.18a shows how normal conversation among a group of friends is essentially bus-based communication, as the transmitter uses the name of the person as the address when sending a message across the communication channel, which is air. The person who is addressed by name then responds to the transmitter. Figure 11.18b illustrates how Internet addresses are used on an Ethernet network for computer communication. An Ethernet network is a bus, as all computers monitor traffic on the network and only respond to those data packets whose header addresses match their assigned Internet address.



**FIGURE 11.18** Two examples of bus communication.



Figure 11.19 shows an I<sup>2</sup>C bus, which consists of a data line (SDA) and a clock line (SCL) used to implement half-duplex communication. Observe that the devices connected to the I<sup>2</sup>C bus do not have chip select signals like SPI-based peripherals. Instead, each device has a 7-bit address whose upper 4 bits (*mmmm*) are device specific and encoded within the device. The next 3 bits of the 7-bit address are typically personalized by external pins that are connected to either V<sub>dd</sub> or ground to provide logic 1 or logic 0, respectively. The address is always sent as the first byte of an I<sup>2</sup>C bus transaction, which is initiated by the bus master. The least significant bit of the address byte indicates the direction of the transfer. A “0” is a write operation (transfer from master to slave), while a “1” is a read (transfer from slave to master). Each I<sup>2</sup>C peripheral device decodes the address byte to determine if it is the target of the transmission, removing the need for individual chip select lines. Adding another I<sup>2</sup>C device to the bus does not require using an additional port on the PIC, which is a distinct advantage over SPI-based peripherals. Like the SPI protocol, the PIC can act as either a slave or master on the I<sup>2</sup>C bus; the examples in this book always use the PIC as the sole I<sup>2</sup>C bus master. The pullup resistors on the SCL/SDA lines are needed because these drivers are open-drain in order to provide *multi-master* capability. In a multi-master bus, any device can act as a bus master. This requires an *arbitration* mechanism that decides which device controls the bus in the case of simultaneous attempts to access the bus (see Chapter 15, “Beyond the PIC18Fxx2,” for a discussion of I<sup>2</sup>C bus arbitration). The most recent version of the I<sup>2</sup>C specification has support for a 10-bit address that is transmitted in the first 2 bytes of a transaction. The I<sup>2</sup>C peripherals used within this book’s examples use 7-bit addresses.

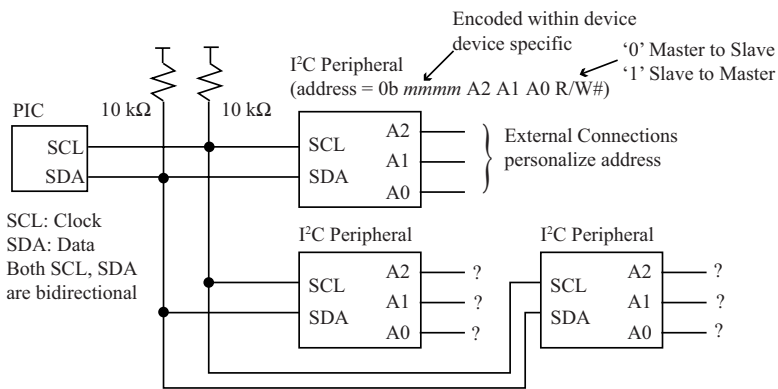
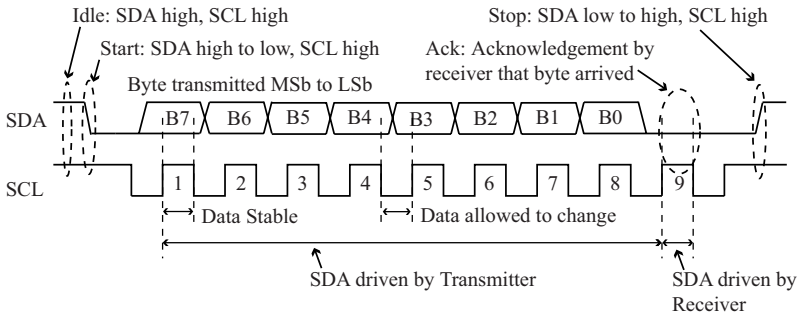


FIGURE 11.19 The I<sup>2</sup>C bus.

Figure 11.20 shows the details of an I<sup>2</sup>C bus transfer. The *idle* condition is when both SDA and SCL lines are high prior to the beginning of a bus transfer.



**FIGURE 11.20** I<sup>2</sup>C data transfer.

The master always provides the SCL signal and initiates an I<sup>2</sup>C transaction. The *start* condition, a high-to-low transition on SDA while SCL is high, signals the beginning of an I<sup>2</sup>C data transfer. The first byte after a start condition is always the address byte used to select a particular I<sup>2</sup>C peripheral. Multiple bytes can be sent within an I<sup>2</sup>C transaction with each byte sent MSb first. SDA data is stable while SCL is high and changes while SCL is low. Each byte transmission ends with a 9<sup>th</sup> bit time in which the transmitter stops driving the SDA line so that the receiver can *acknowledge* the byte transmission by pulling the SDA line low. If the receiver does not drive SDA low, the SDA pullup resistor keeps SDA high and the transmitter reads a “1” for the acknowledge bit instead of a “0”. A “1” acknowledgment bit is called a *not-acknowledge*, or a *NAK*, while a “0” acknowledgment bit is called an *ACK*. Typically, the transmitter will interpret this as an error condition and abort the transfer. There are multiple reasons why a receiver may not acknowledge a byte transmission; if this is the address byte, a coding mistake could result in the wrong address being used, or perhaps the receiver cannot accept new data or has experienced an internal failure. In any case, the acknowledge bit provides feedback to the transmitter on whether a byte has been received. The acknowledgment bit is nonoptional for normal transfers; each byte transmission includes an acknowledgment bit. The master provides the acknowledgment bit in the case where a slave sends a byte to the master for a read operation. After an acknowledgment bit, a slave can hold the SCL line low, which forces the master into a wait condition until the slave releases the SCL line. In this way, the slave can provide flow control on a byte-by-byte basis; this is the only time that the slave may drive the SCL line. The *stop* condition, defined as a low-to-high SDA transition while SCL is high, ends an I<sup>2</sup>C transaction and frees the bus, allowing it to be driven by another bus master.

After the stop condition, both SCL and SDA are undriven and thus pulled high by the pullup resistors. A *repeated start* condition is when another start is sent within a transaction; this ends the current transaction and begins a new transaction, thus allowing the current bus master to start a new transaction without relinquishing control of the bus. More details on special transactions such as CBUS transfers in which acknowledgment bits are not provided are found in the I<sup>2</sup>C bus specification [14].

## 11.7 I<sup>2</sup>C ON THE PIC18FXX2

Table 11.4 gives the control bits associated with the PIC18 I<sup>2</sup>C Master Mode (SSPM bits are “1000”), which provides for one bus master (the PIC18) and uses MSSP hardware for generating bit sequencing.

**TABLE 11.4** Control Registers/Bits for I<sup>2</sup>C Master Mode Configuration

Name	SFR(bit)	Comment
SSPEN	SSPCON1[5]	Must be “1” to enable SCL, SDA pins.
SSPM	SSPCON1[3:0]	“1000”, I <sup>2</sup> C Master mode.
WCOL	SSPCON1[7]	Master Transmit: “0” no collision; “1” indicates a write attempted when I <sup>2</sup> C conditions not valid for transfer, must be cleared in software.
SSPOV	SSPCON1[6]	Master Receive: “0” no overflow; “1” indicates a byte is received while the SSPBUF is still holding the previous byte (must be cleared in software).
ACKSTAT	SSPCON2[6]	“1” acknowledge received from slave, “0” no acknowledge received from slave.
ACKDT	SSPCON2[5]	Acknowledge bit sent back by master after receive from slave; default is “0”.
ACKEN	SSPCON2[4]	Set to “1” to begin acknowledge sequence, automatically cleared when completed.
RCEN	SSPCON2[3]	Set to “1” to enable receive in I <sup>2</sup> C master mode, automatically cleared when 8 bits are received.
PEN	SSPCON2[2]	Set to “1” to initiate STOP condition, automatically cleared when completed.
RSEN	SSPCON2[1]	Set to “1” to initiate a repeated START condition, automatically cleared when completed. →

SEN	SSPCON2[0]	Set to "1" to initiate a START condition, automatically cleared when completed.
R/W#	SSPSTAT[2]	"1" when transmit in progress, "0" otherwise. If this bit OR'ed with SEN, RSEN, PEN, RCEN, ACKEN is "0", the MSSP is idle.
BF	SSPSTAT[0]	In receive mode, "1" when SSPBUF is full, "0" otherwise. In transmit mode, "1" when transmit is in progress (does not include ACK receipt), "0" otherwise.
SSPIF	PIR1[3]	Set to "1" after 8-bit transmission plus acknowledgment receipt is complete.
TRISC3	TRISC[3]	Must be "1" so that RC3/SCK/CSL pin is an input to allow open-drain drive by MSSP module.
TRISC4	TRISC[4]	Must be "1" so that RC4/SDI/SDA pin is an input to allow open-drain drive by MSSP module.

Table 11.5 gives the actions that can be performed in the I<sup>2</sup>C Master Mode configuration. A complete I<sup>2</sup>C transaction is built by sequencing through a combination of the actions in Table 11.5. The actions cannot be queued; in other words, transmit data cannot be written to the SSPBUF register until the start condition has completed.

**TABLE 11.5** Available Actions in I<sup>2</sup>C Master Mode Configuration

Action	Description
Perform Start Condition	Set SEN bit, wait for it to be reset by hardware completion.
Perform Repeated Start Condition	Set RSEN bit, wait for it to be reset by hardware completion.
Perform Stop Condition	Set PEN bit, wait for it to be reset by hardware completion.
Perform an ACK/NAK	Copy acknowledge value (0 or 1) to ACKDT bit, set ACKEN and wait it for to be reset by hardware completion.
Transmit Data	Copy data to SSPBUF; SSPIF bit is set when transmission complete and acknowledgment received.
Receive Data	Configure the I <sup>2</sup> C port to receive data by setting the RCEN bit; the BF bit is set when data is received.

All of the actions of Table 11.5 involve waiting for status bits to be reset, indicating operation completion. The C code functions presented in this chapter that implement these actions rely on the watchdog timer to escape any infinite wait loops due to protocol or hardware failure. Furthermore, status information is tracked via a persistent variable so that when a WDT expiration occurs, the function that caused the problem is reported. This is *defensive programming* and provides a method for debugging I<sup>2</sup>C interface problems.

Figure 11.21 shows this strategy used in implementing the start condition via the C function `i2c_start()`. The `#define` statements give the possible values for the

```

#define I2C_IDLE_ERR      1
#define I2C_START_ERR    2
#define I2C_RSTART_ERR   3
#define I2C_STOP_ERR     4
#define I2C_GET_ERR      5
#define I2C_PUT_ERR      6
#define I2C_MISSACK_ERR  7
#define I2C_ACK_ERR      8
#define I2C_NAK_ERR      9
// error variable for acknowledge
persistent char i2c_errstat; ← Variable for tracking I2C function calls

i2c_idle(){ // wait for idle condition
    unsigned char byte1;
    unsigned char byte2;
    asm("clrwdt");
    i2c_errstat = I2C_IDLE_ERR;
    do {
        // byte1 has R/W bit.
        byte1 = SSPSTAT & 0x04;
        byte2 = SSPCON2 & 0x1F;
    }while (byte1 | byte2);
    asm("clrwdt");
    i2c_errstat = 0;
}

i2c_start(){
    i2c_idle();
    i2c_errstat = I2C_START_ERR; ← Remember this function for error tracking
    SEN = 1; // initiate start
    while (SEN); // wait until start finished } Do START condition. If WDT
    asm("clrwdt"); } expires, track error with i2c_errstat.
    i2c_errstat = 0; ← Clear variable used for tracking function calls.
}

void i2c_print_err(){
    pcr1f(); printf("I2C bus error is ");
    switch (i2c_errstat) {
        case 0: printf("None");break;
        case I2C_IDLE_ERR : printf("Idle");break;
        case I2C_START_ERR : printf("Start");break;
        case I2C_STOP_ERR : printf("Stop");break;
        case I2C_GET_ERR : printf("Get");break;
        case I2C_PUT_ERR : printf("Put");break;
        case I2C_MISSACK_ERR : printf("Missing Ack");break;
        case I2C_ACK_ERR : printf("Ack");break;
        case I2C_NAK_ERR : printf("Nak");break;
        default: printf("Unknown");
    }pcr1f();
}

```

Status codes for tracking I<sup>2</sup>C bus actions

I<sup>2</sup>C interface is idle if R/W#, SEN, RSEN, PEN, RCEN, and ACKEN bits are all clear.

Utility function for printing i2c\_errstat value in case of error.



**FIGURE 11.21** `i2c_idle()`, `i2c_start()`, `i2c_print_err()` functions (see CD-ROM file `./code/common/i2cmsu.c`).

persistent char `i2c_errstat` variable used for error tracking. Recall that the persistent modifier protects the variable from being touched by the initialization runtime C code, so this variable can track actions across processor resets. The `i2c_idle()` function waits until the I<sup>2</sup>C port is idle and then returns. The call to `i2c_idle()` by `i2c_start()` is not strictly necessary in a single master system, but is included here for completeness purposes; the `i2c_idle()` calls used in the following functions can be removed if performance is an issue. Within `i2c_start()`, the statement `i2c_errstat = I2C_START_ERR` records the current function being executed for error tracking purposes. The start condition is initiated by the statement `SEN = 1`; the `while(SEN) {}` loop waits for the MSSP hardware to reset this bit indicating start condition completion. If the watchdog timer expires during this time, the `main()` code that detects the timeout can use the utility function `i2c_print_err()` to print the `i2c_errstat` value to help track the source of the timeout.

Figure 11.22 shows the functions `i2c_rstart()` (repeated start condition), `i2c_stop()` (stop condition), and `i2c_ack(unsigned char ackbit)` (perform acknowledge with value `ackbit`). These functions use the `i2c_errstat` variable in the same manner as the `i2c_start()` function. The `ackbit` parameter of `i2c_ack()` is written to the ACKDT bit to specify the acknowledge bit value (0 = ACK, 1 = NAK).

Figure 11.23 shows functions for performing single-byte transfers in I<sup>2</sup>C master mode. The `i2c_put(unsigned char byte)` function transmits `byte` over the I<sup>2</sup>C port; transmission is triggered by the statement `SSPBUF = byte`. The `while(!SSPIF) {}` loop exits when the transmission and acknowledgment from the slave is complete. The ACKSTAT bit contains the value of the received acknowledgment. The expected value is typically “0” for normal operation; this function performs a software reset via `asm(“reset”)` and sets the error status with `i2c_errstat = I2C_MISSACK_ERR` if a “1” (NAK) is received. It is expected that the `main()` code will detect this software reset condition and use `i2c_print_err()` to display an appropriate error message.

The `i2c_put_noerr(unsigned char byte)` also transmits `byte` over the I<sup>2</sup>C port, but it returns the value of the ACKSTAT bit instead of performing error checking. In some cases, the NAK condition is returned intentionally by an I<sup>2</sup>C slave to indicate a *not ready* condition; this function is provided for use in those situations. The master must know a priori when it is valid for a slave device to return a NAK condition. The `i2cput_byte()` function waits for an idle condition before calling `i2c_put()`.

```

i2c_rstart(){// repeated start
    i2c_idle();
    i2c_errstat = I2C_RSTART_ERR;
    RSEN = 1; // initiate start
    // wait until start finished
    while (RSEN);
    asm("clrwdt");
    i2c_errstat = 0;
}

i2c_stop() {
    i2c_idle();
    i2c_errstat = I2C_STOP_ERR;
    PEN=1; // initiate stop, PEN=1
    //wait until stop finished
    while (PEN);
    asm("clrwdt");
    i2c_errstat = 0;
}

i2c_ack(unsigned char ackbit){
    // send acknowledge
    asm("clrwdt");
    ACKDT = ackbit;
    if (ackbit) i2c_errstat = I2C_NAK_ERR;
    else i2c_errstat = I2C_ACK_ERR;
    //initiate acknowledge cycle
    ACKEN = 1;
    // wait until acknowledge cycle finished
    while(ACKEN);
    asm("clrwdt");
    i2c_errstat = 0;
}

```

Performed Repeated Start Condition

Perform Stop Condition

ACK bit value, "0" is an acknowledge, "1" is a not-acknowledge

Initiate Acknowledgement, wait for completion



**FIGURE 11.22** i2c\_rstart(), i2c\_stop(), i2c\_ack() functions (see CD-ROM file ./code/common/i2cmsu.c).

The `i2c_get(unsigned char ackbit)` function receives a byte from the I<sup>2</sup>C port and sends an acknowledgment with value `ackbit`. The RCEN (receive enable) bit is used to initiate the receive condition; the RCEN bit is reset and the BF flag (buffer full) is set when SSPBUF contains new data. The BF bit is cleared upon reading the SSPBUF register. The `i2cget_byte()` function waits for an idle condition before calling `i2c_get()`.

The SSPADD register within the MSSP subsystem sets the bit rate of the I<sup>2</sup>C port as given by Equation 11.2.

$$BR = \frac{FOSC}{(4 * (SSPADD + 1))} \tag{11.2}$$

Listing 11.1 gives the function used in these examples to initialize the I<sup>2</sup>C port to Master mode. The `bitrate` parameter is written to the SSPADD register to set the SCL clock frequency, and the RC3/SCK/SCL and RC4/SDI/SDA pins are config-

```

unsigned char i2c_put(unsigned char byte ) {
    i2c_errstat = I2C_PUT_ERR;
    SSPIF = 0; //clear interrupt flag
    SSPBUF = byte; // write byte
    while(!SSPIF); // wait for finish an ack
    i2c_errstat = 0;
    asm("clrwdt");
    if (ACKSTAT) {
        //no acknowledge returned, so reset
        i2c_errstat = I2C_MISSACK_ERR;
        asm("reset");
    }
    return(0);
}
}

unsigned char i2c_put_noerr(unsigned char byte ) {
    i2c_errstat = I2C_PUT_ERR;
    SSPIF = 0; //clear interrupt flag
    SSPBUF = byte; // write byte
    while(!SSPIF); // wait for finish an ack
    i2c_errstat = 0;
    asm("clrwdt");
    if (ACKSTAT) return(1);
    return(0);
}
}

unsigned char i2c_putbyte(unsigned char byte) { }
    i2c_idle();
    return(i2c_put(byte));
}

unsigned char i2c_get(unsigned char ackbit) {
    unsigned char byte;

    i2c_errstat = I2C_GET_ERR;
    RCEN = 1; //initiate read event
    while(RCEN); // wait until finished
    asm("clrwdt");
    while (!BF); //also check buffer full
    asm("clrwdt");
    byte = SSPBUF; // read data
    i2c_errstat = 0;
    i2c_ack(ackbit);
    return(byte);
}

unsigned char i2c_getbyte(unsigned char ackbit) {
    i2c_idle();
    return(i2c_get(ackbit));
}
}

```

Initiate transmit by writing byte to SSPBUF register. SSPIF set when transmit is complete.

If returned ACK bit is "1", set error variable and do software reset.

Same as `i2c_put()` but do not do software reset on NAK, instead return the value of the acknowledgement bit.

Check for idle condition before sending byte.

Configure for reception and wait for byte to be received.

Read byte from SSPBUF and send acknowledgement

Check for idle condition before initiating receive.



**FIGURE 11.23** Functions for performing single-byte I<sup>2</sup>C transmit/receive (see CD-ROM file `./code/common/i2cmsu.c`).

ured as inputs. The `i2c_init()` function completes the list of support functions used in the examples of this book for performing I<sup>2</sup>C transfers on the PIC18F242.

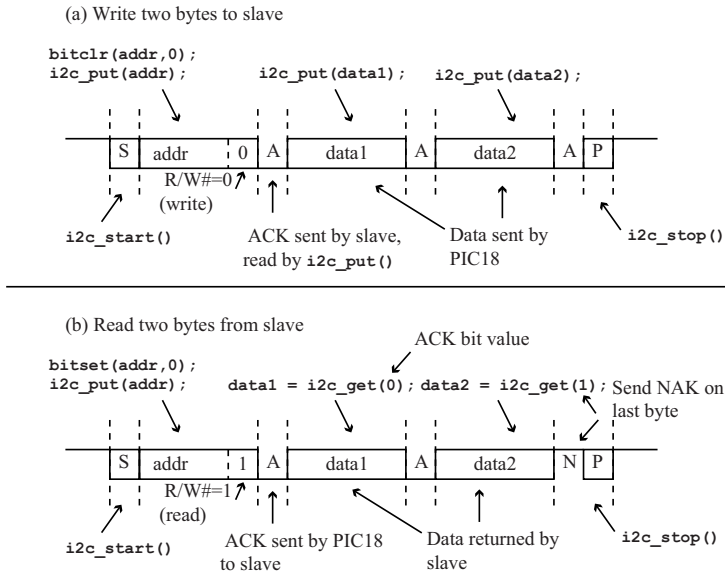


**LISTING 11.1** C function for initializing I<sup>2</sup>C Master mode (see CD-ROM file ./code/common/i2cmsu.c).

```

i2c_init(char bitrate){
    // enable I2C Master Mode
    SSPM3 = 1; SSPM2 = 0; SSPM1 = 0; SSPM0 = 0;
    SSPADD = bitrate; // set bus clk
    SSPEN = 1;
    bitset(TRISC,3);
    bitset(TRISC,4); // SDA, SCL pins are inputs
    SSPIF = 0; // clear SPIF bit
    i2c_errstat = 0; // clear error status
}
    
```

Figure 11.24(a) shows how the functions `i2c_start()`, `i2c_put()`, and `i2c_stop()` are used to write 2 bytes of data to an I<sup>2</sup>C slave device.



**FIGURE 11.24** Using the support functions to implement I<sup>2</sup>C transfers.

The `i2c_start()` function call begins the transaction, followed by an `i2c_put(addr)` that sends the address of the slave. The `bitclr(addr,0)` statement before the `i2c_put(addr)` ensures that the R/W# bit (LSb of the address) is cleared to “0”, indicating a write operation (master transfers data to slave). The next two `i2c_put()` function calls send two data bytes to the slave. The transaction is ended by an `i2c_stop()` function call. Figure 11.24(b) shows how the functions

`i2c_start()`, `i2c_put()`, `i2c_get()`, and `i2c_stop()` are used to read two data bytes from an I<sup>2</sup>C slave device. The `i2c_start()` function call begins the data transfer, followed by an `i2c_put(addr)` that sends the address of the slave. The `bitset(addr,0)` statement before the `i2c_put(addr)` ensures that the R/W# bit (LSb of the address) is set to “1”, indicating a read operation (slave transfers data to master). The next two `i2c_get()` function calls read two data bytes from the slave. The “0” parameter used in the first `i2c_get(0)` function call is the acknowledge bit value sent by the PIC18 to the slave after the byte is read from the slave. This value is “0” (an ACK) for all bytes read from the slave except for the last byte, in which a “1” (a NAK) is sent by the master to tell the slave that it should not start another data transfer. The transaction is ended by an `i2c_stop()` function call.

**Sample Question: Assume an I<sup>2</sup>C device requires a command byte written to it to tell it what internal register to return on the next read transaction. Write a sequence of function calls using the functions discussed in this section to accomplish this action. Assume the variables `dev_addr`, `cmd`, and `data` are used for the device address, command byte, and returned data byte, respectively. Assume the device requires an ACK bit value of “1” to halt the read transaction.**

*Answer:* We need a write transaction followed by a read transaction as shown in Listing 11.2.

**LISTING 11.2** Sample question solution.

```
i2c_start();
bitclr(dev_addr,0) // the LSB must be 0 for a write transaction
i2c_put(dev_addr); // send the I2C device address
i2c_put(cmd);      // send command byte
i2c_rstart();     // start new transaction
bitset(dev_addr,1) // the LSB must be 1 for a read transaction
i2c_put(dev_addr); // send the I2C device address
data = i2c_get(1); // get data, send ACK of '1' to halt read
i2c_stop();       // stop the transaction
```

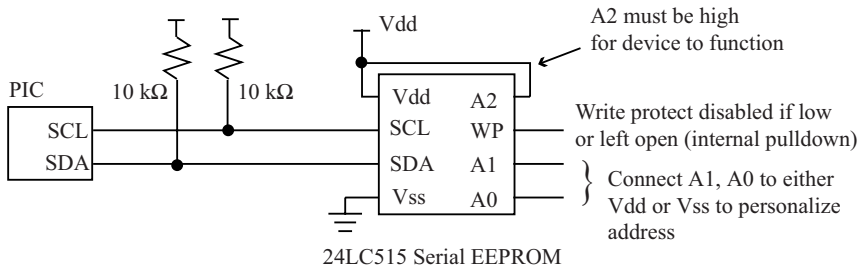
Observe that the LSb of the device address is forced to be a “0” in the write transaction, and a “1” in the read transaction. Function calls of `i2c_start()/i2c_stop()` could be used to replace the `i2c_rstart()` function call; in a multiple bus master situation, this gives other bus masters a chance to control the bus.

**Sample Question: Assuming  $FOSC = 29.4912$  MHz, what SSPADD value is required for an I<sup>2</sup>C bus rate of 100 kHz?**

*Answer:* Equation 11.2 can be solved for SSPADD as  $SSPADD = [FOSC/(4*BR)] - 1$ . Thus,  $SSPADD = [29.4912E6/(4*100e3)] - 1 = 72.7 = \sim 73$ .

## 11.8 THE 24LC515 SERIAL EEPROM

The 24LC515 512K serial EEPROM has an internal organization of 64K x 8 and uses an I<sup>2</sup>C port for communication. Figure 11.25 shows a PIC18 to 24LC515 interface using the I<sup>2</sup>C port. The write protect (WP) pin on the 24LC515 can be used to disable writes to the device; it can be left open or tied to V<sub>ss</sub> to enable writes. The A2 input is an unused input that must be tied high for the device to function correctly. The A1, A0 inputs are used to personalize the device address by connecting them to either V<sub>dd</sub> or ground. This allows up to four 24LC515 devices to exist on the same I<sup>2</sup>C bus.



**FIGURE 11.25** PIC18 to 24LC515 I<sup>2</sup>C interface.

Figure 11.26 shows the address byte format for the 24LC515. The upper 4 bits are fixed at “1010”. The 64K x 8 organization of the 24LC515 means that addresses are 16 bits with a range 0x0000 to 0xFFFF. However, the internal organization of the 64 x 8 memory is split into two 32K memory blocks, each with its own internal 15-bit address counter. The B (block select) bit of the address byte determines whether the current operation is to the low memory block (0x0000 through 0x7FFF) or high memory block (0x8000 through 0xFFFF). The least significant bit of the address byte is the R/W# bit as with all I<sup>2</sup>C address bytes.

Figure 11.27 shows the write operation for the 24LC515. The I<sup>2</sup>C address byte is followed by the high and low address bytes of the starting location for the write. The most significant bit of the high address byte is a don't care as the block select bit within the I<sup>2</sup>C address byte determines which memory block is being written; these 15 address bits are written to the internal address counter for the 32K block selected by the block select bit of the address byte. The internal page size of the 24LC515 is 64 bytes, so up to 64 bytes can be written in one write operation. Page wrapping occurs in the same way as discussed in Figure 11.12 for the 25LC640 serial EEPROM, except the starting page boundary is when the lower 6 bits of the address are all zeros and the ending page boundary has the lower 6 bits as all ones.

When doing multiple byte writes, the best practice is to write a complete page at one time and force the starting address to begin on a page address.

Address byte format for 24LC515 serial EEPROM

7	6	5	4	3	2	1	0
1	0	1	0	B	A1	A0	R/W#

B : Memory block select, if “0” then operation is to low memory block (0x0000-0x7FFF), if “1” then operation is to high memory block (0x8000-0xFFFF)

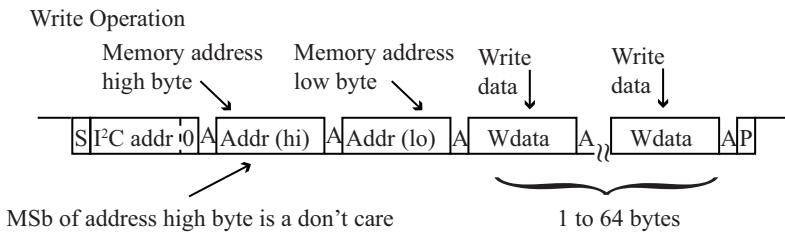
A1, A0: Used to personalized address, up to four LC515 EEPROMs can be on bus.

R/W#: “1” if read operation, “0” if write operation

Addressing Examples:

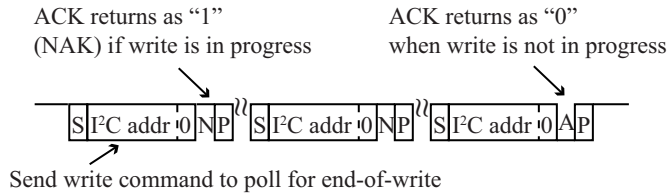
A1	A0	B	R/W#	Address	Operation
0	0	0	0	0xA0	Write to low block
			1	0xA1	Read from low block
		1	0	0xA8	Write to high block
			1	0xA9	Read from high block
0	1	0	0	0xA2	Write to low block
			1	0xA3	Read from low block
		1	0	0xAA	Write to high block
			1	0xAB	Read from high block
1	0	0	0	0xA4	Write to low block
			1	0xA5	Read from low block
		1	0	0xAC	Write to high block
			1	0xAD	Read from high block
1	1	0	0	0xA6	Write to low block
			1	0xA7	Read from low block
		1	0	0xAE	Write to high block
			1	0xAF	Read from high block

**FIGURE 11.26** Address byte format for the 24LC515.



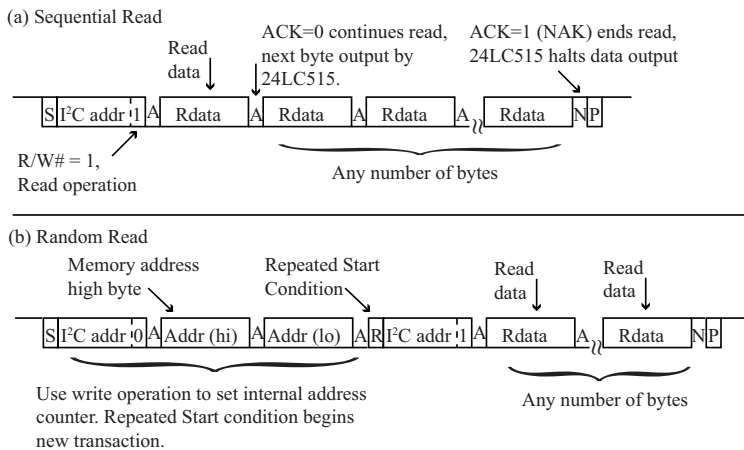
**FIGURE 11.27** Write operation for the 24LC515.

The worst-case write completion time is 5 ms. However, the end-of-write condition can be polled by sending the write command and checking the ACK bit status as shown in Figure 11.28. If the acknowledgment bit returns as “1” (a NAK), a write is still in progress. Once the acknowledge bit returns as “0”, the next operation can be started. It is more efficient to poll for end-of-write than to place a delay of 5 ms after each write operation.



**FIGURE 11.28** Polling for end-of-write.

Figure 11.29 shows read operation sequencing for the 24LC515. A sequential read (Figure 11.29a) returns the memory contents pointed to by the internal address counter of the 24LC515. Each data byte returned by the 24LC515 increments the internal address counter for the selected block. An acknowledgment bit of “0” returned by the PIC18 causes the 24LC515 to output another data byte. An acknowledgment bit of “1” returned by the PIC18 causes the 24LC515 to release the SDA line and to stop sending data. A sequential read can access the contents of one entire 32K memory block, either high or low, as determined by the block select bit sent in the I<sup>2</sup>C address byte at the beginning of the read transaction. When the internal address counter reaches the end of a 32K block (either 0x7FFF or 0xFFFF), it wraps around to the beginning of the block.

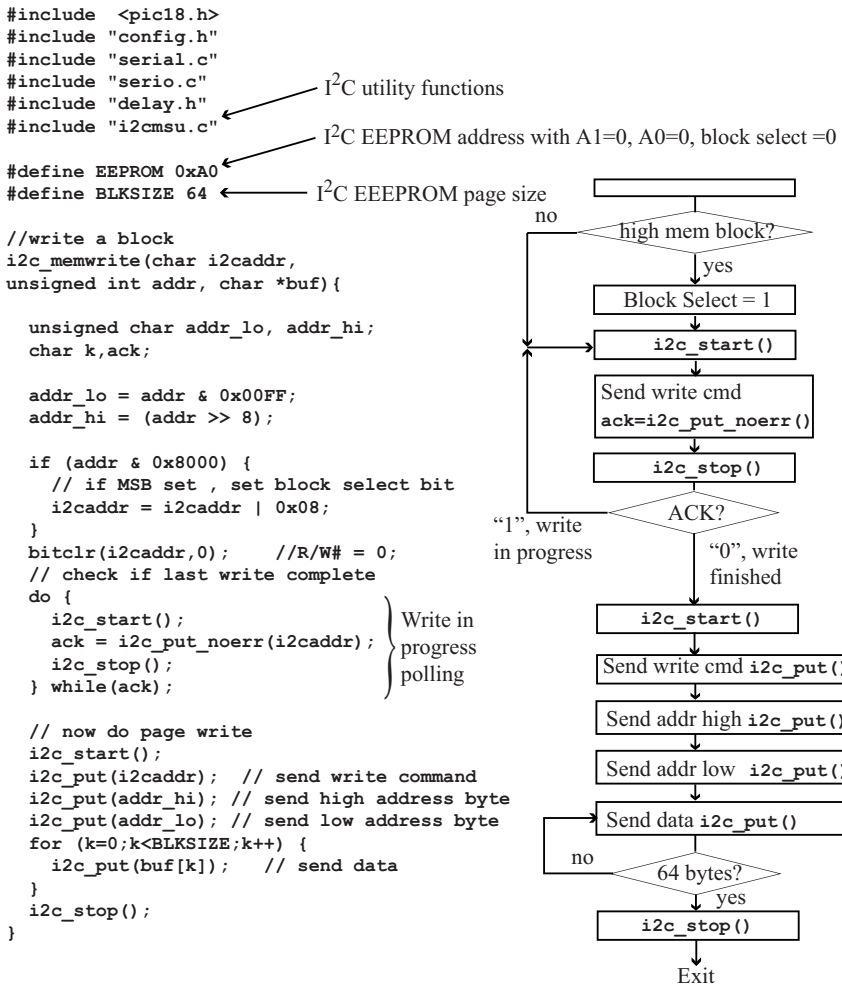


**FIGURE 11.29** Read operations for the 24LC515.

Figure 11.29(b) shows how to use the write command to set the internal address counter before beginning a sequential read. Sending a repeated start condition after the EEPROM address bytes halts the write command. Only the internal

address counter is affected by the write operation; an internal write to memory contents is not started. A repeated start condition holds SCL low while SDA is high for one-half of an I<sup>2</sup>C bit time, then brings SCL high for one-half of an I<sup>2</sup>C bit time while SDA is high, and then pulls SDA low while SCL is high to signal the start of a transaction. To send a repeated start condition, the `i2c_rstart()` function is used instead of the `i2c_start()` function.

The `i2c_memwrite()` function in Figure 11.30 implements the write operation of Figure 11.27. The 64 bytes pointed to by the `buf` parameter are written beginning at memory address `addr`, with parameter `i2caddr` containing the I<sup>2</sup>C address of the



**FIGURE 11.30** C function for a page write to the 24LC515 (see CD-ROM file `./code/common/i2c_memutil.c`).

EEPROM. The block select bit of `i2caddr` is set to “1” (high block) if the `addr` value has its MSb set, indicating that it is within the upper memory block of 0x8000-0xFFFF. A `do-while{}` loop checks for a write-in-progress by sending the write command and checking the returned acknowledgment status; the loop is exited when the acknowledgment bit returns as “0”. The page write is then performed by sending the write command, the high and low bytes of the address, and the 64 bytes pointed to by the `buf` parameter. The previously defined `i2c_start()`, `i2c_put_noerr()`, `i2c_put()`, and `i2c_stop()` functions are used to implement the I<sup>2</sup>C bus operations.

Figure 11.31 contains the `i2c_memread()` function that reads 64 bytes from the EEPROM. This function has the same parameters as `i2c_memwrite()` and also has the same initial code for setting the block select bit of `i2caddr` and polling for write-in-progress. After this is completed, the internal address counter is set by sending the write command followed by the high and low address bytes. The `i2c_rstart()` function is then used to perform the repeated start condition, which halts the write operation and starts a new transaction. The read command is sent and the 64 bytes are read from the EEPROM. An acknowledgment bit of “0” is sent for each byte except for the last byte for which an acknowledgment bit of “1” is sent to halt EEPROM data output. An `i2c_stop()` ends the transaction and the function exits.

The `main()` code that uses the `i2c_memread()` and `i2c_memwrite()` functions for testing reads and writes of the serial EEPROM is shown in Figure 11.32. The user is first prompted to enter either read or write mode. In write mode, the user enters 64-byte strings that are written to the EEPROM using `i2c_memwrite()`. Each string is written twice in succession to test the write-in-progress polling of `i2c_memwrite()`. In read mode, each key press reads 64 bytes from the EEPROM using `i2c_memread()` and the resulting string read from the serial EEPROM is printed to the console.

Figure 11.33 shows console output from a test of the Figure 11.32 code. Two 64-byte strings are entered, which means the first four pages of the EEPROM are written as each string is written twice. The read test reads back the first four pages of the EEPROM. The console output shows the expected string values.

```

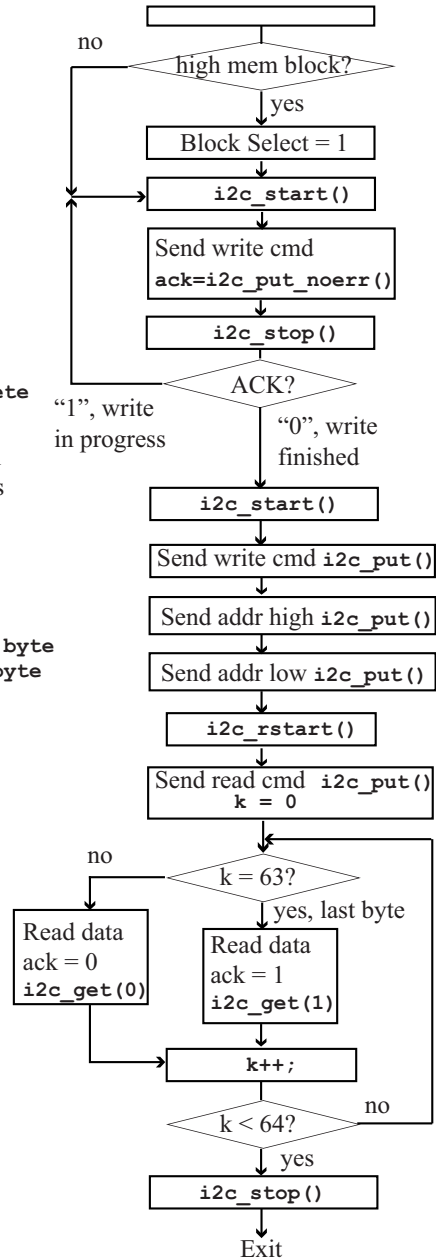
i2c_memread(char i2caddr,
unsigned int addr, char *buf){

    unsigned char addr_lo, addr_hi;
    char k,ack;

    addr_lo = addr & 0x00FF;
    addr_hi = (addr >> 8);

    if (addr & 0x8000) {
        // if MSB set , set block select bit
        i2caddr = i2caddr | 0x08;
    }
    bitclr(i2caddr,0); //R/W# = 0;
    // check if last write complete
    // write command will NAK if not complete
    do {
        i2c_start();
        ack = i2c_put_noerr(i2caddr);
        i2c_stop();
    } while(ack);
    //set address counter
    i2c_start();
    i2c_put(i2caddr); // send command
    i2c_put(addr_hi); // send high address byte
    i2c_put(addr_lo); // send low address byte
    bitset(i2caddr,0); //R/W# = 1;
    i2c_rstart(); // repeated start
    i2c_put(i2caddr); // send command
    for (k=0;k<BLKSIZE;k++) {
        if (k== 63) {
            // get last byte, do NAK
            buf[k] = i2c_get(1);
            i2c_stop();
        } else
            // get data, do ACK
            buf[k] = i2c_get(0);
    }
}

```



**FIGURE 11.31** C function for sequential read from the 24LC515 (see CD-ROM file `./code/common/i2c_memutil.c`).



```

char membuf[BLKSIZE]; ← Storage for test strings
unsigned int memaddr;

main(void) {
    unsigned char mode,i;
    // 19200 in HSPLL mode, crystal = 7.3728 MHz
    serial_init(95,1); ← Initialize Serial Port

    if (!RI) {
        RI = 1;
        printf("Software reset!");pcrlf();
        if (i2c_errstat) i2c_print_err();
    }
    if (!TO) {
        printf("Watchdog timer reset has occurred.\n");
        pcrlf();
        if (i2c_errstat) i2c_print_err();
    }
    i2c_init(73); ← Initialize I2C port, bus speed is
                  ~100 kHz @ FOSC=29.4912 MHz

    pcrlf(); printf("I2C Mem Test Started"); pcrlf();
    SWDTEN = 1; // enable watchdog timer
    memaddr = 0;
    printf ("Enter 'w' for write mode, anything else reads: ");
    mode = getche(); pcrlf();

    while(1) {
        if (mode == 'w') {
            printf("Enter %d chars.",BLKSIZE);pcrlf();
            for(i = 0;i< BLKSIZE;i++) {
                membuf[i] = getche();
            }
            pcrlf();printf("Doing Write");pcrlf();
            // write same string twice to
            //check Write Busy polling
            i2c_memwrite (EEPROM,memaddr,membuf);
            memaddr = memaddr +BLKSIZE;
            i2c_memwrite (EEPROM,memaddr,membuf);
            memaddr = memaddr +BLKSIZE;
        } else {
            // read 64 characters
            i2c_memread(EEPROM,memaddr,membuf);
            for(i = 0;i< BLKSIZE;i++) putchar(membuf[i]);
            pcrlf();
            printf("Any key continues read...");pcrlf();
            getch();
            memaddr = memaddr + BLKSIZE;
        }
    }
}

```

Software Reset, check if I<sup>2</sup>C function call was in progress

Watchdog Timer Reset, check if I<sup>2</sup>C function call was in progress

Input 64 character string from console, write to EEPROM twice to check functionality of write-in-progress status check.

Read 64 characters from EEPROM and print to console



**FIGURE 11.32** main() for testing I<sup>2</sup>C EEPROM read/writes.

```

I2C Mem Test Started
Enter 'w' for write mode, anything else reads: w
Enter 64 chars.
A person who graduates today and stops learning tomorrow is
Doing Write
Enter 64 chars.
uneducated the day after. Life long learning is very important.
Doing Write
Enter 64 chars.

```

Two strings entered; each string saved twice to EEPROM

```

I2C Mem Test Started ← Reset Pressed
Enter 'w' for write mode, anything else reads: r
A person who graduates today and stops learning tomorrow is
Any key continues read...
A person who graduates today and stops learning tomorrow is
Any key continues read...
uneducated the day after. Life long learning is very important.
Any key continues read...
uneducated the day after. Life long learning is very important.
Any key continues read...

```

Strings read back from EEPROM

**FIGURE 11.33** Console output from testing I<sup>2</sup>C EEPROM read/writes.

**Sample Question:** Write a sequence of function calls using the functions of Section 11.7 that will return the byte from location 0x80F0 within the 24LC515 Serial EEPROM. Assume A1 is tied low and A0 is tied high on the EEPROM. Write the byte that is read from the EEPROM into the data\_byte variable.

**Answer:** Two transactions are needed: a write transaction to send the address, and a read transaction to return the data byte. The required function calls are shown in Listing 11.3.

**LISTING 11.3** Sample question solution.

```

i2c_start();
i2c_put(0xAA);           // address byte, write command, high block
i2c_put(0x80);           // high byte of memory address
i2c_put(0xF0);           // low byte of memory address
i2c_rstart();
i2c_put(0xAB);           // address byte, read command, high block
data_byte = i2c_get(1);  // read byte, ack of '1' to stop read
i2c_stop();

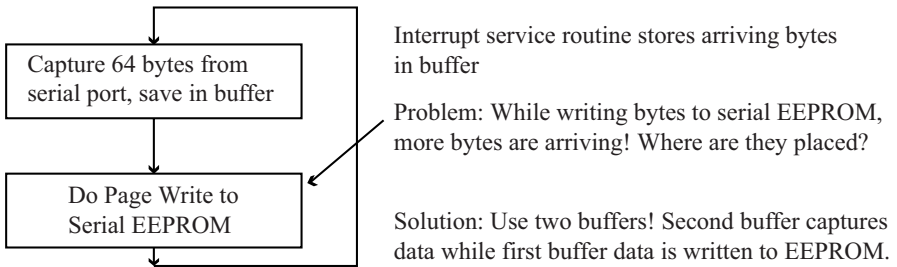
```

The address byte of 0xAA in the write transaction is 0b10101010. Bit3 is the block select bit and is a “1” because the address 0x80F0 is in the upper 32K memory block. Bit2 and Bit1 correspond to the A1 and A0 pins, respectively. Bit0 (the LSB) is a “0” because this is a write transaction, which is required to

set the internal address register. The address byte 0xAB of the second transaction has the LSB as “1” since it is a read transaction.

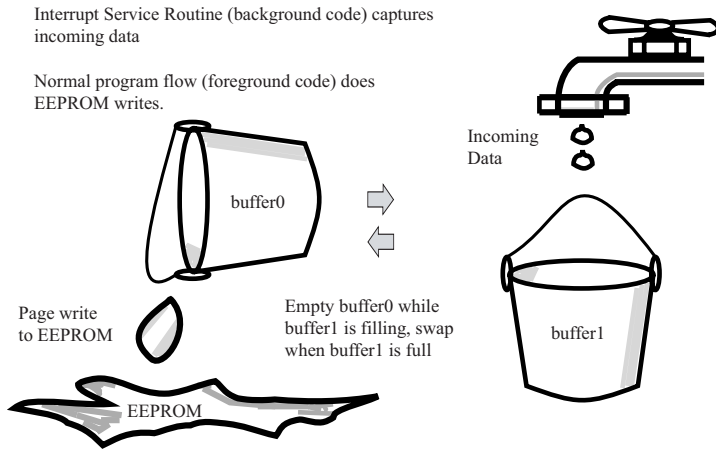
## 11.9 DOUBLE BUFFERING FOR INTERRUPT-DRIVEN WRITES

The previous examples that performed writes to SPI and I<sup>2</sup>C serial EEPROMs prompted the user to enter a string that was stored in a buffer, wrote that buffer to the serial EEPROM, and then prompted the user for another string. However, how would data that is arriving in a continuous stream be handled? Figure 11.34 shows the problem with using only one buffer to handle streaming input data. Once the buffer is full, a page write must be done to EEPROM to save the buffer contents. However, new data is continuously arriving; if the current buffer is used to save the incoming data, the old data is overwritten.



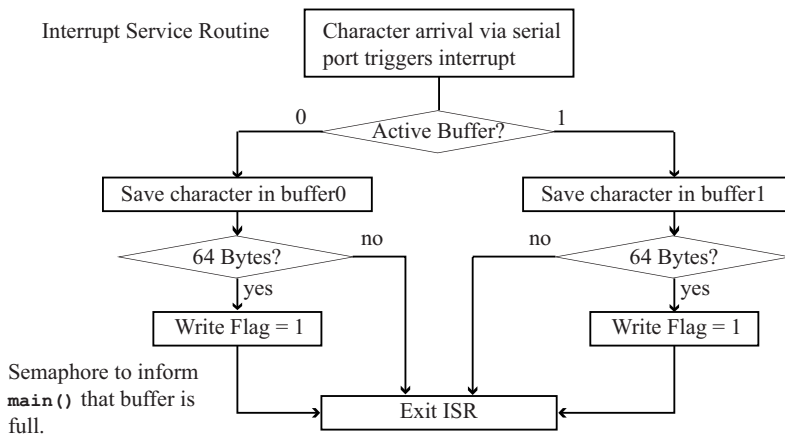
**FIGURE 11.34** Using one buffer to capture streaming data.

In Figure 11.35 it is seen that the solution to this problem involves using two buffers, named *buffer0* and *buffer1*. Once *buffer0* is filled with input data, it is swapped with *buffer1* and emptied (written to EEPROM), with *buffer1* used to store input data during the EEPROM write operation. After *buffer1* becomes full, it is swapped with the now empty *buffer0* and the process is repeated. The ISR captures incoming bytes while the foreground code writes the full buffer to EEPROM. This works as long as the buffer used to capture incoming data does not fill before the EEPROM write is finished; recall that in a streaming data application the outgoing bandwidth must exceed the incoming bandwidth or no amount of buffering will prevent eventual data loss due to buffer overflow. The incoming data must arrive by a different communication channel than that used to save the data; in this example, data arrives via the asynchronous serial port and is written to the EEPROM using the I<sup>2</sup>C port.



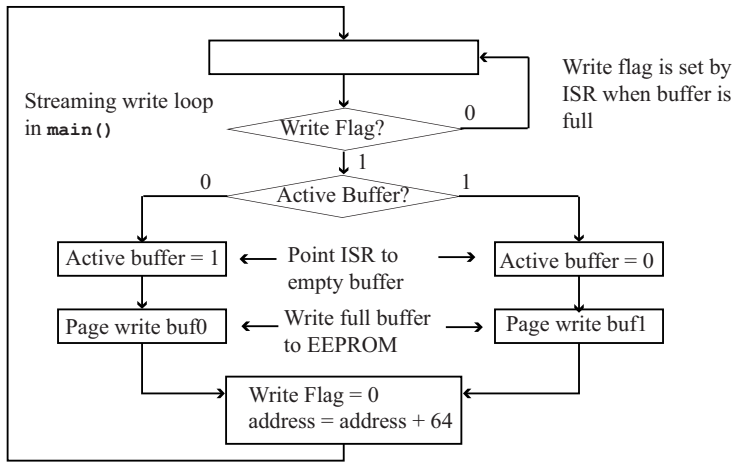
**FIGURE 11.35** Double buffering to capture streaming data.

Figure 11.36 shows the ISR flowchart for capturing streaming data. Two buffers, `buffer0` and `buffer1`, are used to store data, with the `active_buffer` flag determining the buffer currently used for input data. A character arrival at the asynchronous serial port triggers an interrupt, which causes the character to be stored in either `buffer0` or `buffer1` as determined by `active_buffer`. A buffer becomes full after 64 bytes because that is the page size of the I<sup>2</sup>C serial EEPROM. The `write_flag` is a semaphore that is set once the buffer becomes full; this notifies `main()` (the foreground code) that the current active buffer must be written to EEPROM.



**FIGURE 11.36** ISR flowchart for capturing streaming data.

Figure 11.37 shows the streaming write loop in `main()` that writes the streaming input data to EEPROM. The loop waits until `write_flag` is set, indicating that the current active buffer is full. It then changes the active buffer by toggling the `active_buffer` flag from 0 to 1 or 1 to 0 so that the ISR will now save data to the empty buffer. This is equivalent to swapping the full bucket with the empty bucket in Figure 11.35. Once the `active_buffer` flag is toggled, the full buffer is written to EEPROM, the `write_flag` semaphore is cleared, and the EEPROM address is incremented by 64. How do we know if the incoming data rate is not exceeding the outgoing data rate? If the `write_flag` is already set when the code loops back to the top of the `while(1){}` loop after writing the full buffer to EEPROM, buffer overflow has occurred. If overflow occurs, either the incoming data rate must be reduced or the outgoing data rate must be increased. Lowering the baud rate of the asynchronous serial port will reduce the incoming data rate. A new EEPROM with either a faster write time or larger internal page buffer will increase the outgoing data bandwidth. The C code implementation for Figures 11.36 and 11.37 is left as a suggested laboratory exercise in Appendix E, “Suggested Laboratory Exercises.”



**FIGURE 11.37** `main()` flowchart for capturing streaming data.

## SUMMARY

Synchronous IO is available on the PIC18 via the USART or MSSP subsystems. All modes provide separate clock and data signals. The USART synchronous IO is a half-duplex channel that provides valid data on the falling clock edge. The MSSP subsystem supports both SPI and I<sup>2</sup>C interfaces. The SPI mode is a full-duplex

channel in which configuration bits CKE, CKP, and SMP offer different combinations of clock polarity and active clock edge for serial data IO. The SPI mode requires that a separate parallel port signal be used as a chip select for each external SPI peripheral. The I<sup>2</sup>C bus is a half-duplex channel that is a true bus in that an initial address byte is used to select the active device for a transaction. This offers an advantage over the SPI port in that new devices can be added to the I<sup>2</sup>C bus with no additional control signals required. Serial EEPROMs offer nonvolatile external storage and support writing a page of data at a time by use of an internal write buffer to optimize write operations, which require a significant amount of time to complete. A double buffer scheme is required for capturing streaming input data as one buffer is emptied as the other buffer is used to hold incoming data.

## REVIEW PROBLEMS

---

Some of the following problems refer to device datasheets found at [www.maxim-ic.com](http://www.maxim-ic.com), [www.microchip.com](http://www.microchip.com), [www.semiconductors.philips.com](http://www.semiconductors.philips.com), [www.intersil.com](http://www.intersil.com), and [www.atmel.com](http://www.atmel.com). For the I<sup>2</sup>C questions, use the `i2c_start()`, `i2c_rstart()`, `i2c_stop()`, `i2c_put(char byte)`, `char i2c_get(char ackbit)` functions discussed in the chapter.

Answer the following questions about the Maxim MAX5439, a digital potentiometer with an SPI port.

1. Determine the correct settings for the PIC18 CKE and CKP configuration bits for interfacing to this device.
2. How many wiper positions does this digital potentiometer support?
3. Is there a method for determining the current wiper register contents? If yes, how is this done?

Answer the following questions about the Maxim MAX5408, a digital potentiometer with an SPI port.

4. Determine the correct settings for the PIC18 CKE and CKP configuration bits for interfacing to this device.
5. How many wiper positions does this digital potentiometer support?
6. Is there a method for determining the current wiper register contents? If yes, how is this done?
7. This potentiometer has a *zero-crossing detection* feature. What does this mode do and why is it included?
8. What is the maximum clock frequency supported for the SPI port?

Answer the following questions about the Atmel AT25256A, a serial EEPROM with an SPI port.

9. What is the organization of this device and total bit capacity?
10. Determine the correct settings for the PIC18 CKE and CKP configuration bits for interfacing to this device.
11. What is the maximum clock frequency supported for the SPI port?
12. What is the page buffer size?
13. How is a write-in-progress determined?

Answer the following questions about the Intersil X9221A, a digital potentiometer with an I<sup>2</sup>C port. Assume the A3, A2, A1, A0 pins are tied high.

14. How many wiper positions does this digital potentiometer support?
15. Write a sequence of I<sup>2</sup>C function calls that will set the wiper to a particular position.
16. Write a sequence of I<sup>2</sup>C function calls that will read the current wiper position.
17. What is the maximum clock frequency supported for the I<sup>2</sup>C port?

Answer the following questions about the Philips PCF8598C-2, a serial EEPROM with an I<sup>2</sup>C port.

18. What is the organization of this device and total bit capacity?
19. What is the maximum clock frequency supported for the I<sup>2</sup>C port?
20. What is the page buffer size?
21. How is a write-in-progress determined?
22. How long does a typical page write take?

Answer the following questions.

23. Compute the approximate amount of time it takes to transfer the data required for a page write to the Microchip 24LC515 serial EEPROM using a 400 kHz I<sup>2</sup>C clock rate and  $F_{OSC} = 20$  MHz. Assume the start and stop conditions each require one I<sup>2</sup>C bit time, and 20 instruction cycles of overhead for each byte sent over the I<sup>2</sup>C bus.
24. Using the assumptions of the previous problem, what is the approximate maximum baud rate that can be sustained on the serial port without overflow in the continuous data streaming application of Section 11.9? Assume that there are five stop bit times between each data arrival on the asynchronous serial port.

25. Assuming  $F_{OSC} = 30$  MHz, what SSPADD value is required for an I<sup>2</sup>C bus rate of 400 kHz?
26. Devise a scheme for measuring how long a typical self-timed write on the 24LC515 serial EEPROM actually takes. Determine if the typical write time is dependent upon the number of bytes that is actually written.



*This page intentionally left blank*

# 12



# Data Conversion

*By J.W. Bruce*

## In This Chapter

- Data Conversion Basics
- Analog-to-Digital Conversion
- PIC18Fxx2 Analog-to-Digital Converter
- Digital-to-Analog Conversion
- Digital-to-Analog Converter Example: The MAXIM 518

This chapter discusses a few of the many different analog-to-digital converter (ADC), digital-to-analog converter (DAC) architectures, and the advantages and disadvantages of each. The PIC18's successive approximation ADC and a serial DAC are covered and example applications are explained.

## 12.1 LEARNING OBJECTIVES

---

After reading this chapter, you will be able to:

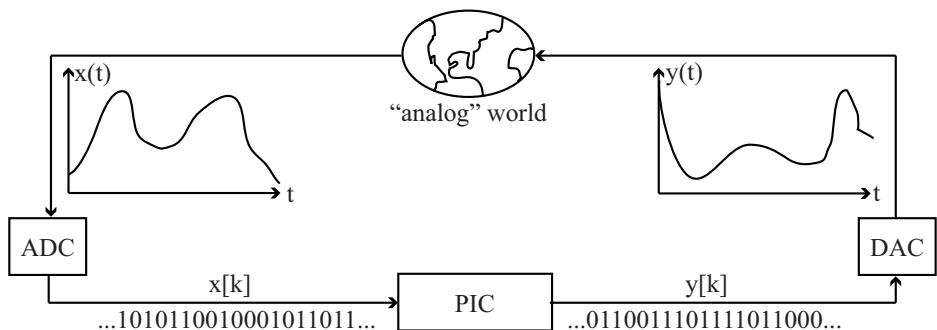
- Select the appropriate ADC and DAC architecture based on the application requirements.

- Implement a simple data acquisition system using the PIC18's analog to digital converter.
- Construct a parallel R-2R resistor ladder flash DAC using the PIC18.
- Implement a PIC18 to I<sup>2</sup>C serial DAC interface.
- Construct a simple three-function waveform generator with the PIC18.

## 12.2 DATA CONVERSION BASICS

As predicted by Moore's Law in 1964 [1], digital computing power has exponentially increased at ever smaller, incremental costs. For example, as we've seen in the previous chapters, the PIC18 has the capability of replacing several chips. With this increase of computing power, many applications usually accomplished with analog circuitry have found a new lease on life in the digital realm. However, the real world still is and will always continue to be a fundamentally analog place. To bring the digital processing of the PIC18 and its benefits to bear on real-world applications, the analog signal of interest must be translated into a format the PIC18 can understand. This is the function of the analog-to-digital converter (ADC). After processing by the PIC18, the resulting digital stream of information must be returned to its analog form by a digital-to-analog converter (DAC). Analog once again, the information may be "consumed" by the human senses, most often sight or hearing. An illustration of this information flow is shown in Figure 12.1.

ADCs and DACs are ubiquitous in computing systems. Many electronic products, including compact disc players, camcorders, digital cellular phones, modems,



**FIGURE 12.1** Typical use of a PIC to interface with the "analog" world.

computer sound cards, computer graphics adapters, and high-definition televisions contain one or more data converters. Because ADCs are so useful and required by so many small microprocessor applications, microprocessor architects often include an ADC as a built-in peripheral, and the Microchip PIC18 designers did just that.

From a programmer's viewpoint, the ADCs and DACs in Figure 12.1 can be regarded as black boxes. That is, an ADC accepts an input of some analog quantity, typically voltage, and provides an  $n$ -bit digital code output every  $f_s$  seconds that represents that analog input. The number  $f_s$  is said to be the ADC's *sampling frequency*. The black box DAC accepts an  $n$ -bit digital word input every  $f_s$  seconds and generates an equivalent analog output, usually voltage. The number  $f_s$  is the DAC's sample frequency. For many purposes, this is a sufficient interpretation of data converters. However, an understanding of how the data conversion is done will help you understand why there are limitations on ADC and DAC operation, and should help you in selecting data converters for the application at hand.

## **12.3 ANALOG-TO-DIGITAL CONVERSION**

---

The methods by which a digital code is generated within an ADC are diverse. A detailed discussion would fill several books (a few references on ADCs have been provided in the bibliography [34–37]). While ADCs can have almost any analog quantity (current, charge, voltage, temperature, acoustical pressure, etc.) as an input, the most common ADCs convert an analog voltage into a digital number. Usually, systems that are converting a wide variety of quantities first convert those signals into voltages, and then use a voltage-mode ADC to convert the value into a digital number. The digital number that an ADC generates can be in any encoding system, but is most typically represented in unsigned or signed binary.

ADCs and their capabilities are described by a bewildering number of parameters. A full discussion of ADC parameters is more appropriate with a more advanced electronics background, and the interested reader is encouraged to explore the data conversion references in Appendix H, "References." However, some basic descriptive parameters for ADCs must be understood to select and use them properly.

The speed of an ADC is measured as the minimum sampling period  $T_{min}$ , the shortest time required to convert an input voltage to a digital number. Minimum sampling period is equivalently reported as the maximum sampling frequency, the maximum number of samples that the ADC can convert in one second. The maximum sampling frequency  $f_{max}$  is found by  $f_{max} = 1/T_{min}$ . Of course, a faster ADC gives us a more accurate temporal picture of what the analog voltage input is doing, but this knowledge requires that our microprocessor must operate on and/or store more data.

An ADC's resolution is the smallest change in its analog input that is detectable at its output, usually a change of  $\pm 1$  in the output number. In other words, resolution represents the change in ADC input that corresponds to a 1 LSb change in output. ADC precision is the number of levels that the ADC can distinguish. Sometimes, ADC precision is quoted by the number of binary bits required to encode the number of levels. The ADC range is the total span over which inputs can be converted accurately. Quite often, the range extremes,  $V_{REF+}$  and  $V_{REF-}$ , in the case of voltage conversion, are provided as ADC inputs.

**Sample Question: How many bits of precision would an ADC require to distinguish 1  $\mu$ V (one microvolt =  $1.0e-6$ ) differences over a range 0–2 V?**

*Answer:* 
$$\text{precision} = \frac{\text{range}}{\text{resolution}} = \frac{2 \text{ V} - 0 \text{ V}}{1 \mu\text{V}} = 2,000,000 \text{ levels}$$

An ADC would need 21 bits of output ( $2^{21} = 2,097,152$ ) to encode these required 2,000,000 levels.

**Sample Question: What is the range and resolution of an 8-bit ADC with  $V_{REF+}=10 \text{ V}$  and  $V_{REF-} = -10 \text{ V}$ ?**

*Answer:* 
$$\text{range} = V_{REF+} - V_{REF-} = 10 \text{ V} - (-10 \text{ V}) = 20 \text{ V}$$

$$\text{resolution} = \frac{\text{range}}{\text{precision}} = \frac{20 \text{ V}}{2^8 \text{ levels}} = \frac{20 \text{ V}}{256} = 78.125 \text{ mV}$$

**Sample Question: What is the maximum sampling frequency for the 8-bit ADC in the preceding question if the minimum sampling period is 2.5  $\mu$ s?**

*Answer:* 
$$f_{\text{max}} = \frac{1}{T_{\text{min}}} = \frac{1}{2.5 \mu\text{s}} = 400 \text{ kHz}$$

**Sample Question: If the ADC is operating at maximum speed, how much storage is required to store one second of ADC output? One year's worth?**

*Answer:* 
$$1 \text{ sec storage} = \frac{1 \text{ byte}}{\text{sample}} \frac{400000 \text{ samples}}{\text{sec}} 1 \text{ sec} = 400,000 \text{ bytes} = 400\text{kpsps}$$

$$1 \text{ year} = 400\text{kpsps} \frac{1 \text{ B}}{\text{sample}} \frac{60 \text{ sec}}{\text{min}} \frac{60 \text{ min}}{\text{hr}} \frac{24 \text{ hr}}{\text{day}} \frac{365 \text{ days}}{\text{yr}} 1 \text{ yr} = 1.26 \times 10^{13} \text{ B}$$

Thus, we would need approximately 117 disk drives, each with 100 GB ( $\sim 10^{11}$  bytes, 1 GB =  $2^{30}$  bytes) of capacity to store a year's worth of data from our ADC. That is a lot of storage space!

Most ADCs have uniform stepsizes, the difference between the minimum and maximum voltages that correspond to the same output code. If stepsize is uniform or constant over the ADC range, the stepsize is equal to the resolution. (There are

some specialty ADCs with nonuniform stepsizes; for example, some audio ADCs have stepsizes that change logarithmically over their range to match the response of the human ear.) Using our black box view of ADCs, an  $n$ -bit ADC with uniform stepsizes divides its range into  $2^n$  equal segments. The ADC output is simply the number of the segment in which the ADC input lies. Mathematically, the ADC digital output at sample time  $k$  ( $x[k]$ ) is given in Equation 12.1.

$$x[k] = f \left\{ 2^n \cdot \frac{x(kT) - V_{REF-}}{V_{REF+} - V_{REF-}} \right\} \quad (12.1)$$

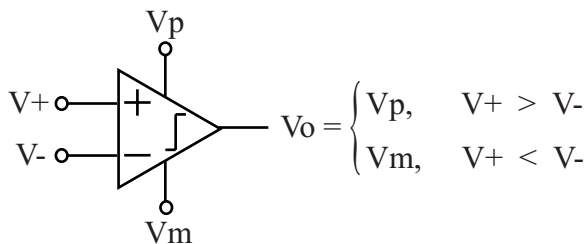
In Equation 12.1,  $T$  is the ADC sampling period,  $x(kT)$  is the input voltage  $V_{in}$  at time  $kT$ ,  $n$  is the number of output bits, and  $f$  is a function that converts its argument to an integer, typically by truncation or rounding. Because information is lost due to rounding or truncation in analog-to-digital conversion always introduces some error. The difference between the actual ADC input value and the value implied by the ADC's digital output is called *quantization error*, which can be made smaller by increasing the ADC's precision.

**Sample Question: Assuming our example 8-bit ADC in the prior examples performs conversion by rounding, what is the ADC output code for  $-7.25$  V?  $2.0$  V?**

Answer: 
$$\text{round} \left\{ 256 \cdot \frac{-7.25\text{V} - (-10\text{V})}{10\text{V} - (-10\text{V})} \right\} = \text{round} \{ 35.2 \} = 35 = 0x23$$

$$\text{round} \left\{ 256 \cdot \frac{2.0\text{V} - (-10\text{V})}{10\text{V} - (-10\text{V})} \right\} = \text{round} \{ 153.6 \} = 154 = 0x9A$$

The basic building block in nearly all ADCs is the voltage comparator. Figure 12.2 shows a voltage comparator circuit symbol. The circuitry inside a comparator can be quite complex, so we will use the comparator in Figure 12.2 as a black box.

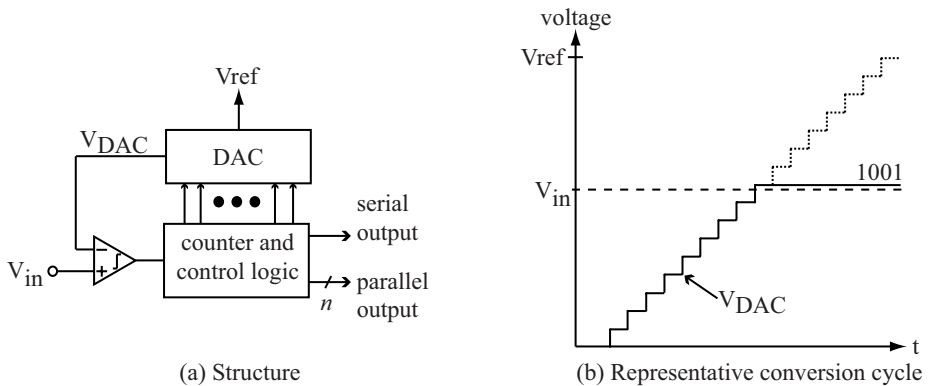


**FIGURE 12.2** Voltage comparator circuit symbol and operation.

If the positive input terminal voltage  $V+$  is greater than the negative input terminal voltage  $V-$ , the comparator output is the comparator's positive power supply voltage  $V_p$ . If  $V+$  is less than  $V-$ , the comparator's output is the negative supply voltage  $V_m$ . Therefore, we see that if  $V_p = V_{dd}$  and  $V_m = 0$ , the comparator in Figure 12.2 will generate a digital signal that can communicate with digital logic, like that in microprocessors. Because of this behavior, the voltage comparator is sometimes called a 1-bit ADC. By changing the comparator's input voltages  $V+$  and  $V-$  and using numerous comparators in different ways, an analog voltage can be compared to reference voltages and a digital number representation formed. There are many different algorithms, circuits, and configurations by which this can be done. In this section, we introduce three popular voltage-mode ADC architectures in use today: the counter ramp ADC, the successive approximation ADC, and the flash ADC.

### Counter Ramp ADC

One of the simplest of the ADC architectures is the counter ramp ADC. The structure of the counter-ramp ADC is shown in Figure 12.3. At the beginning of the conversion, the digital counter is reset to zero. This drives the analog output of the internal DAC to zero volts. The counter is then incremented, which causes the analog output of the DAC to increase in a stair-step fashion. When the counter has been clocked to a point where the DAC analog output is at a higher potential than the input voltage,  $V_{in}$ , the counter is stopped. At this time, the counter contains the digital code equivalent to the analog input voltage. This is shown graphically in Figure 12.3. After the digital value has been determined, it may be transmitted from the counter in parallel or shifted out serially via a shift register.



**FIGURE 12.3** Counter ramp ADC.

Counter ramp ADCs are not very efficient. Consider the  $n$ -bit counter ramp ADC. Since the input may be equal to the full-scale analog reference voltage, the counter must count through all  $2^n$  possible digital codes before the comparator stops the counter. In effect, the counter-ramp ADC performs an exhaustive search to find the nearest digital representation of the input voltage. This search will take up to  $2^n$  clock pulses. Therefore, the  $n$ -bit counter-ramp ADC sampling at  $f_s$  samples per second must run the internal counter at  $2^n f_s$  operations per second. For a large  $n$ , the internal counter clock and circuitry must be much faster than the sampling frequency. At high sampling rates with practical word sizes, the required internal circuit clock frequency becomes prohibitive. Because most signal processing applications require uniformly sampled data values, the counter ramp ADC allocates  $2^n$  clock cycles, the worst case, for every conversion regardless of the result. Therefore, counter ramp ADCs find use only in the slowest applications, usually with small to moderate output word lengths.

**Sample Question: Describe the conversion process for a 4-bit counter ramp ADC with  $V_{REF-} = 0\text{ V}$ ,  $V_{REF+} = 4\text{ V}$ , and  $V_{in} = 3.14159\text{ V}$ .**

*Answer:* The ADC's range is 4 V. The ADC resolution is  $4\text{ V}/16 = 0.25\text{ V}$ . Therefore, each 1 LSB increase in output corresponds to an increase in 0.25 V of input. When the conversion process begins, the counter is reset to 0b0000 and the ADC input voltage  $V_{in}$  is compared with the  $V_{DAC}$  voltage of 0 V. Since  $V_{in} > 0\text{ V}$ , the counter increases by 1 LSB and the input voltage is compared with the  $V_{DAC}$  voltage of 0.25 V. This comparison causes another increment in the counter. The counter continues increasing, with 0.25 V added to the  $V_{DAC}$  voltage each step, until the  $V_{DAC}$  voltage is greater than 3.14159 V. The first counter value that generates such a voltage is 0b1101, which produces  $V_{DAC} = 13/16 * 4\text{ V}$ , or  $V_{DAC} = 3.25\text{ V}$ . This counter state occurs at the 14<sup>th</sup> cycle. The code that is output from the counter ramp ADC is one less than this value, created by decrementing the counter to 0b1100, corresponding to  $V_{DAC} = 12/16 * 4\text{ V} = 3.0\text{ V}$ . If the counter ramp input voltage were very close to  $V_{REF+}$ , it would require 16 cycles to generate the maximum output code of 0b1111 (when the code reaches the maximum value of 0b1111 the conversion is halted, since incrementing the counter further will cause it wrap to 0b0000). Since the input voltage is not known, our 4-bit counter ramp ADC must anticipate a 16-cycle conversion time.

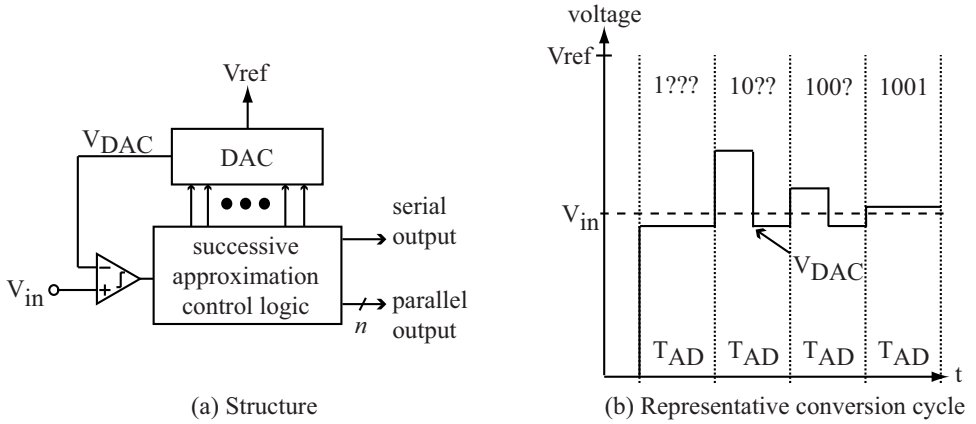
## Successive Approximation ADC

Like the counter ramp ADC, the successive approximation ADC converts the analog voltage present on its input to a digital code. However, the successive approximation ADC performs the conversion in a more efficient way—a binary search.



This makes the successive approximation ADC much faster than a counter ramp ADC at a cost of increased complexity for control logic.

Consider the block diagram of an  $n$ -bit successive approximation ADC as given in Figure 12.4. At the sample time, the ADC sets the MSb in the Successive Approximation Register (SAR) to “1”. All the remaining lower bits are reset to “0”. This digital “guess” is converted back to an analog value and is compared with the input.



**FIGURE 12.4** Successive approximation ADC.

Therefore, the SAR contains a digital code representative of mid-scale (0b100...0). The DAC produces a corresponding mid-scale analog output, which is halfway between the minimum ( $V_{REF-}$ ) and maximum voltage ( $V_{REF+}$ ) that could be presented at the ADC input. If the input is at a higher potential than the feedback analog representation of the “guess” ( $V_{in} > V_{DAC}$ ), the MSb is left set to “1”. If the input is at a lower potential than the feedback analog value, which is the case of  $V_{in} < V_{DAC}$ , the MSb is reset to “0”. In this step, the successive approximation ADC is determining the proper state of the MSb; in other words, whether the analog input value lies in the upper (MSb = 1) or lower (MSb = 0) half of the ADC’s range.

Now, the entire procedure is repeated for the second most significant bit. While the MSb is unchanged from the first approximation, the second MSb is set with the remaining lower bits reset. This digital code, an improved “guess,” is converted into an analog value ( $V_{DAC}$ ) and presented to the comparator. At this instant, the SAR value is either (0b1100...0) or (0b0100..0), depending on the outcome of the first approximation. If the analog input is at a higher potential than the feedback “guess” voltage, the second MSb is left at “1”. If not, the second MSb is reset to “0”. At the conclusion of this second approximation cycle, the two most significant bits in the

register determine whether the ADC input is located in the highest (0b11), next-to-highest (0b10), next-to-lowest (0b01), or lowest (0b00) fourth of the ADC's range.

Approximation cycles continue in this manner for each of the remaining lower order bits until all  $n$  bits have been examined. At the conclusion of each cycle, the SAR digital code is converted back to an analog voltage and compared against the input voltage. In this way, each approximation halves the *difference* between the ADC's input and the analog representation of the contents of the SAR. This is shown graphically in Figure 12.4.

Transmission of the digital code from the ADC may be done in two ways: serially or parallel. Each bit of the digital output code may be output from the ADC the instant it is computed. This particular flavor of successive approximation ADC is also known as the serial ADC. The digital code in the SAR may be stored for parallel transmission upon completion of the sample conversion, or transmitted serially at a later time using some defined network protocol like I<sup>2</sup>C or SPI. Nonetheless, when the time arises to convert the next sample, the contents of the SAR are reset and the entire procedure is repeated for the new analog voltage present on the input pin of the ADC.

A disadvantage of the successive approximation ADCs is the many internal operations that must occur for a single sample to be converted. In the  $n$ -bit converter,  $n$  approximations and comparisons must be performed in each sampling period. Therefore, an  $n$ -bit successive approximation ADC running at a sampling frequency of  $f_s$  samples per second must run its internal circuit at a rate of  $nf_s$  operations per second. However, this is much slower and cheaper to build than the required rate of the counter ramp ADC, especially for a large  $n$ . For a given sampling rate, the successive approximation ADC can convert with greater resolution than the counter ramp ADC. The successive approximation ADC iteratively cuts the voltage range in half as it searches for the digital representation of the input voltage. This binary search is more efficient and faster than the exhaustive search of the counter ramp ADC, but it also gives the successive approximation ADC a more complex architecture.

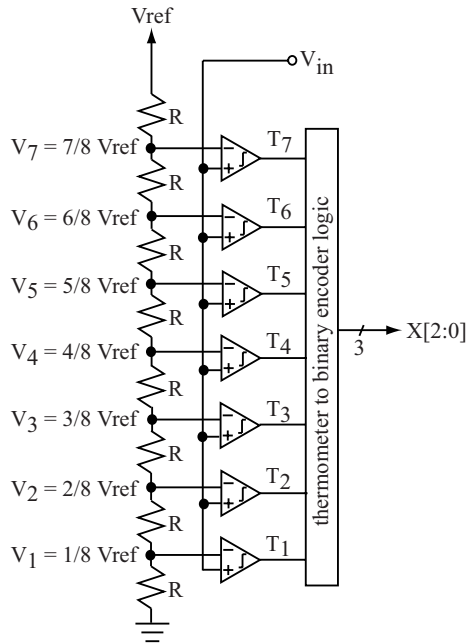
**Sample Question: Describe the conversion process for a 4-bit successive approximation ADC with  $V_{REF-} = 0\text{ V}$ ,  $V_{REF+} = 4\text{ V}$ , and  $V_{in} = 3.14159\text{ V}$ .**

*Answer:* The ADC's range is 4 V. The ADC resolution is  $4\text{ V}/16 = 0.25\text{ V}$ . Therefore, each 1 LSB increase in output corresponds to an increase in 0.25 V of input. When the conversion process begins, the SAR is set to 0b1000 and the ADC input voltage  $V_{in}$  is compared with the midrange voltage 2.0 V ( $8/16 * 4\text{ V}$ ). Since  $V_{in} > 2\text{ V}$ , the control logic leaves the MSb of the SAR set and sets the second MSb of the SAR. The SAR contents are now 0b1100, which represents the voltage  $12/16 * 4\text{ V}$ , or 3.0 V. Because  $V_{in} > 3.0\text{ V}$ , the  
→

second MSb of the SAR is left set. The control logic sets the third MSb of the SAR, now 0b1110. The SAR contents cause the DAC to create a voltage of  $14/16 \times 4$  V, or 3.5 V. Because  $V_{in} < 3.5$  V, the control logic clears the third MSb in the SAR and sets the SAR's LSB. The SAR contents on the fourth cycle are 0b1101, which corresponds to a comparison voltage of  $13/16 \times 4$ , or 3.25 V. The comparator determines  $V_{in} < 3.25$  V, so the LSB is cleared. The 4-bit digital result 0b1100 is computed in four cycles.

**Flash ADC**

The counter ramp ADC determines the output by examining each quantization level ( $2^n$  maximum operations), while the successive approximation ADC examines each bit ( $n$  operations). However, the flash ADC generates all of the output bits in one operation and thus has a speed advantage over the previous two architectures. This speed does come with a drawback—complexity. The flash ADC distributes the sampling process across the entire circuit. This requires much more circuitry as a result. The structure of a flash ADC circuit is shown in Figure 12.5.



**FIGURE 12.5** Resistor string flash ADC architecture.

An  $n$ -bit flash ADC contains  $2^n$  resistors,  $2^n - 1$  comparators and digital encoder logic. Referring to Figure 12.5, the string of resistors from the reference voltage to ground constructs a voltage divider. Assuming that all  $2^n$  voltage divider resistors have the same resistance, the divider generates  $2^n$  analog voltages between ground and the reference voltage. These analog voltages correspond to the points on the ADC transfer curve at which there is no quantization error. These analog voltages are the ones that the output codes of the ADC represent in digital form. Each of  $2^n$  voltage divider levels is the reference voltage input for their respective comparators. The comparators' other input is the flash ADC's input voltage,  $V_{in}$ . The output of the comparators is a thermometer code of the input voltage,  $V_{in}$ . It is named this because of its resemblance to a mercury thermometer.

Consider the case when  $V_{j+1} \geq V_{in} \geq V_j$ . The outputs of the comparators,  $T_j, T_{j-1}, \dots, T_1, T_0$ , will be "1", while the outputs,  $T_{j+1}, T_{j+2}, \dots, T_{2^n-1}$ , will be "0". Therefore, the outputs of the comparators will rise and fall with the input voltage,  $V_{in}$ . In similar fashion, the mercury level in a thermometer tracks the temperature.

Obviously, the large number of bits in the thermometer code is not an efficient representation of the value. It is the function of the encoder to "compact" the information to an efficient representation. The encoder logic accepts the  $2^n$  bits of the thermometer code and outputs the  $n$ -bit binary number corresponding to the number of "1"s in the thermometer code. This may be done a number of ways. The logic may be designed to "look" for the most significant bit in the thermometer code and output the binary number corresponding to that input line, much in the same way as a demultiplexer. However, this method is sensitive to errors in the comparators' thermometer code called *sparkles*. In ideal operation, the thermometer code consists of consecutive "1"s in the lower comparator outputs from  $T_1$  to  $T_j$ . For all comparators from  $T_{j+1}$  to  $T_{2^n-1}$ , the outputs are "0". A comparator output, which is erroneous and causes a departure from this pattern, is called a sparkle. Depending on the method of thermometer-binary encoding, sparkles may lead to gross errors in the output digital code of the ADC. To exacerbate matters, high-speed timing uncertainties may cause multiple sparkles to appear throughout the thermometer code. Various circuit techniques may be applied to suppress the effects of sparkles, such as comparing neighboring bits in the thermometer code, using Gray codes or thermometer code bit summing.

Despite these drawbacks, flash ADCs are extremely attractive because of their high speed. Since all output bits are determined at the same time, a flash ADC with a sampling rate of  $f_s$  samples per second runs at  $f_s$  operations per second. Thus, flash ADCs only need more circuitry to increase the output code word size. However, the number of comparators and resistors will double for each additional bit of output. Furthermore, the complexity of the thermometer-to-binary encoder logic also increases with the number of output bits. Because of their fast operation, flash ADCs

are typically used in high-speed, small word length applications, such as digital video, radar, and digital test and measurement equipment.

**Sample Question: Describe the conversion process for a 4-bit resistor string flash ADC with  $V_{REF-} = 0\text{ V}$ ,  $V_{REF+} = 4\text{ V}$ , and  $V_{in} = 3.14159\text{ V}$ .**

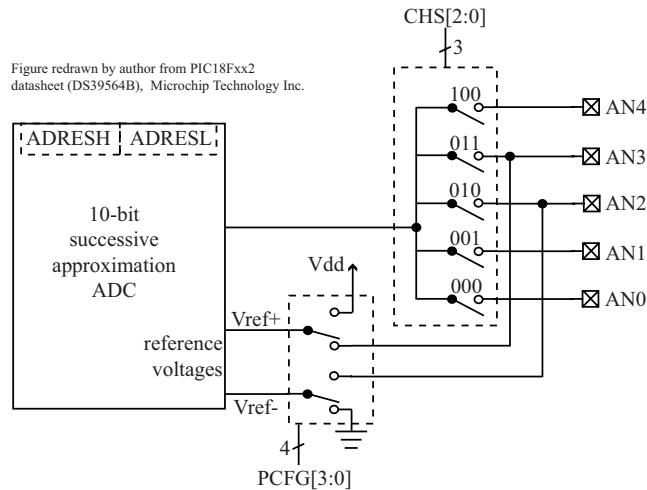
*Answer:* The ADC's range is 4 V. The ADC resolution is  $4\text{ V}/16 = 0.25\text{ V}$ . Therefore, the 16 resistor string reference voltages are 0.25 V, 0.50 V, 0.75 V, ..., 3.25 V, 3.5 V, and 3.75 V. Each of the 15 reference voltages is compared with  $V_{in}$  simultaneously. The lower 12 comparator outputs (up to and including the comparator with the 3.0 V reference voltage) are "1", and the upper three comparator outputs (comparators with 3.25 V, 3.5 V and 3.75 V references) are "0". The thermometer to binary encoder will represent the thermometer code with 12 ones by 0b1100. The flash ADC results are available in one cycle, the time for the comparator output to become stable plus the encoder delay.

## 12.4 PIC18FXX2 ANALOG-TO-DIGITAL CONVERTER

ADCs are used in so many small microprocessors and embedded systems that they are often included as a built-in peripheral. Microchip made just such a decision with the PIC 18Fxx2 microprocessors. The PIC18Fxx2 includes a multiple-channel 10-bit successive approximation ADC. The number of channels depends on the device and package chosen by the designer. For example, the PIC18F442 and PIC18F452 are available in 40- and 44-pin packages and support eight different input channels to the 10-bit ADC. The PIC18F242 and PIC18F252 are only available in 28-pin packages. With such a limited number of package pins, the '2x2 devices only support five different input channels to the internal 10-bit ADC. Other PIC18 devices provide 5–16 input channels to the ADC. We will look at the difference between PIC devices in Chapter 15, "Beyond the PIC18Fxx2." The required differences in using the PIC18's internal ADC between the different devices are usually minor and well documented in the datasheets. The remainder of this section focuses on the PIC18F242 device and its five ADC channels.

Just like the other PIC18 peripherals (USART, interrupt, SPI) that were previously covered, the PIC18's internal ADC is controlled by a number of dedicated configuration, enable, and flag register bits. Also, the external input connections to the ADC are restricted to specific pins—specifically, the PORTA pins on the PIC18F2x2 devices. Different PIC18 devices have the same basic ADC operation; they only differ in the number of analog input channels available for conversion. Figure 12.6 shows a simplified block diagram of the PIC18 ADC system.

The PIC18F2x2 devices support analog input channels AN0-AN4 on package pins RA0-RA3 and RA5, respectively.



**FIGURE 12.6** PIC18 ADC block diagram.<sup>1</sup>

## PIC18F242 ADC Configuration

The PIC18F242 does not require us to use all five analog input channels. In fact, we can use any number, from zero to all, of the ADC channels. The number of ADC channels to use is selected in the first ADC control register ADCON1. In Chapter 8, “The PIC18Fxx2: System Startup and Parallel Port IO,” we saw that the statement `ADCON1 = 0x06` configured PORTA for digital operation. The lower half of the ADCON1 register, bits PCFG[3:0] (ADCON1[3:0]), determine how many ADC and digital channels are available on PORTA of 2x2 devices. Figure 12.7 shows the PIC18F242 ADC configurations for each combination of PCFG[3:0] in ADCON1.

ADCON1 also selects the ADC reference voltage sources,  $V_{REF-}$  and  $V_{REF+}$ . These reference voltages specify the bounds between which the ADC will expect the analog input voltage to appear. The PIC18 ADC divides the range ( $V_{REF+} - V_{REF-}$ ) into  $2^{10} = 1024$  levels. The ADC output is represented as a 10-bit number in the register pair ADRESH:ADRESL. For example, a value of 0b1110 for the lower 4 bits of ADCON1 configures AN0 as an analog input, the remaining analog input pins as digital IO, and use  $V_{DD}/V_{SS}$  as  $V_{REF+}/V_{REF-}$ . In this mode, the upper ADC reference voltage  $V_{REF+}$  is  $V_{DD}$ , the PIC18’s main power supply voltage, and the lower ADC reference voltage  $V_{REF-}$  is  $V_{SS}$  (also called GND), the PIC18’s main ground voltage. If the application specifies that the ADC analog input value is guaranteed

<sup>1</sup> Figure 12.6 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.’s prior written consent.

PCFG[3:0]	AN4	AN3	AN2	AN1	AN0	V <sub>REF+</sub>	V <sub>REF-</sub>
00x0	A	A	A	A	A	V <sub>DD</sub>	V <sub>SS</sub>
00x1	A	V <sub>REF+</sub>	A	A	A	AN3	V <sub>SS</sub>
0100	D	A	D	A	A	V <sub>DD</sub>	V <sub>SS</sub>
0101	D	V <sub>REF+</sub>	D	A	A	AN3	V <sub>SS</sub>
011x	D	D	D	D	D	---	---
1x00	A	V <sub>REF+</sub>	V <sub>REF-</sub>	A	A	AN3	AN2
1001	A	A	A	A	A	V <sub>DD</sub>	V <sub>SS</sub>
1010	A	V <sub>REF+</sub>	A	A	A	AN3	V <sub>SS</sub>
1011	A	V <sub>REF+</sub>	V <sub>REF-</sub>	A	A	AN3	AN2
1101	D	V <sub>REF+</sub>	V <sub>REF-</sub>	A	A	AN3	AN2
1110	D	D	D	D	A	V <sub>DD</sub>	V <sub>SS</sub>
1111	D	V <sub>REF+</sub>	V <sub>REF-</sub>	D	A	AN3	AN2

Bits PCFG[3:0] are in register ADCON1[3:0].

Values shown are for the PIC18F2x2.

A = analog input (to ADC)  
 D = digital IO (according to status of TRISA register)

**FIGURE 12.7** PIC18 ADC port configuration control bits.

to lie in a smaller voltage range, these lower and upper voltage references can be provided to the PIC18. In this way, each of the 1024 possible ADC results will correspond to a more accurate voltage. As an example, a value of 0b1111 for the lower 4 bits of ADCON1 configures AN0 as an analog input, AN3/AN2 as V<sub>REF+</sub>/V<sub>REF-</sub> and AN1/AN4 as digital IO.

It is extremely important to mention at this point that any ADC channel or ADC voltage reference that is enabled should have its corresponding port direction (TRISA) bit set. This configures the package pin to be an input. If this bit is cleared (output mode), the PIC18 ADC will attempt to use the driven pin voltage (V<sub>OL</sub> or V<sub>OH</sub> in the datasheet) as the ADC input or ADC voltage reference. This is seldom desired or useful.

The result of the ADC conversion is 10 bits and is found in the register pair ADRESH:ADRESL that is 16 bits. The ADFM bit (ADCON1[7]) allows the programmer to determine the justification of the 10-bit ADC result in the 16-bit virtual register ADRESH:ADRESL. If ADFM is “1”, the ADC conversion result is right justified (the most significant six bits of ADRESH are cleared). If ADFM is “0”, the ADC result is left justified in ADRESH:ADRESL (the least significant 6 bits of ADRESL are cleared).

While the ADCON1 bits PCFG3:PCFG0 allow the PIC18 to have multiple channels of ADC enabled, the PIC18 can only convert one channel at a time. The ADC channel to be converted by the ADC is selected by the CHS2:CHS0 bits (ADCON0[5:3]). The three CHS2:CHS0 bits are the binary representation for the currently selected ADC channel.

Because the voltages on two different ADC input pins can be drastically different, we must give the PIC ADC circuit time to adjust and acclimate to the new voltage before starting a conversion. The PIC ADC input is a *sample and hold* circuit that steadies the voltage during the successive approximation conversion process.

That sample and hold circuit has an input capacitance that must charge up to the same voltage as the active ADC input pin, and this takes some time. The exact time depends on the impedance of the device generating the voltage for the ADC to convert, the temperature, and the PIC's V<sub>DD</sub>. The PIC18FXX2 datasheet gives exact formulas to compute the minimum time required to acquire the new voltage on an ADC channel change. With typical circuits and devices, 20  $\mu$ s is usually sufficient. It is very important to give the PIC ADC time to acclimate to the new ADC input voltage. If at least 20  $\mu$ s does not elapse between changing the active ADC channel and starting the ADC conversion, the ADC results may not be accurate.

ADCON0 also contains another very important ADC configuration bit, ADON (ADCON0[0]). When ADON = 1, the entire ADC module is enabled, powered up, and consuming power. When ADON = 0, the ADC module is disabled and does not consume power. If the ADC module is not being used, the ADC should be turned off via ADON = 0, which is the setting after reset.

Since the PIC18 ADC uses a successive approximation ADC architecture to generate the 10-bit result, it is a reasonable guess that the conversion will take at least 10 clock cycles. The PIC18Fxx2 datasheet specifies that the ADC requires  $12T_{AD}$  for accurate conversion results, where  $T_{AD}$  is at least 1.6  $\mu$ s. The PIC18 uses an extra  $T_{AD}$  period to set up the ADC before conversion and an extra  $T_{AD}$  period to copy the result to ADRESH:ADRESL after conversion. To guarantee that  $T_{AD}$  is greater than 1.6  $\mu$ s, the PIC18 provides seven options for selecting the ADC conversion clock. The ADC conversion clock is selected by the 3-bit field ADCS2:ADCS0 (ADCON1[6]:ADCCON0[7:6]) split over the ADC configuration registers ADCON0 and ADCON1. Figure 12.8 shows the ADC conversion clock select bit options. When ADCS2:ADCS0 is 0x3 or 0x7, the ADC hardware uses an internal RC oscillator to control the conversion. This internal RC oscillator is guaranteed to have a period from 2–6  $\mu$ s. The other six options allow the ADC to be controlled by a divided FOSC clock. The ADCS2:ADCS0 field allows for ADC

ADCON1 [6] ADCS2	ADCON0 [7:6] ADCS [1:0]	A/D Clock
0	00	$F_{OSC}/2$
0	01	$F_{OSC}/8$
0	10	$F_{OSC}/32$
x	11	$F_{ADC RC}$ (internal ADC oscillator)
1	00	$F_{OSC}/4$
1	01	$F_{OSC}/16$
1	10	$F_{OSC}/64$

**FIGURE 12.8** PIC18 ADC conversion clock select bits.



clock frequencies between FOSC/2 and FOSC/64. If the ADC is controlled by a divided FOSC clock, the divider must be selected such that  $T_{AD}$  is greater than 1.6  $\mu$ s, or ADC conversion results will not be accurate. The PIC18 ADC can perform an A/D conversion while the PIC is in sleep mode. However, sleep mode ADC operation requires the ADC conversion clock be generated by the ADC internal oscillator since the main PIC FOSC clock is not running.

The last PIC ADC option to configure before we can use the ADC is the ADC interrupt sources. The PIC ADC can generate an interrupt when the 10-bit conversion is complete. To enable the PIC ADC interrupts, we must first clear the ADIF (ADC interrupt flag, PIR1[6]) to prevent a spurious interrupt request when we enable interrupts. The ADC interrupt priority level is determined by the state of the ADIP bit (ADC interrupt priority bit, IPR1[6])—set for high priority, clear for low priority bit. Next, we need to set the ADIE bit (ADC interrupt enable, PIE1[6]) to make the ADC an interrupt source. The ADC interrupt is globally enabled by the collective states of the IPEN, GIE/GIEH, and PEIE/GIEL bits. (Refer back to Chapter 10, “Interrupts and a First Look at Timers,” for a review on the operation of these three important enable bits.) The bits that configure the PIC18’s internal ADC are summarized in Figure 12.9.

Name	SFR (bit)	Comments
ADON	ADCON0[0]	0 = ADC is powered off 1 = ADC is powered up
GO/DONE#	ADCON0[2]	0 = A/D conversion not in progress 1 = conversion in progress (set this bit to start ADC conversion)
CHS[2:0]	ADCON0[5:3]	ADC channel select bits 000 = AN0 001 = AN1 010 = AN2 011 = AN3 100 = AN4
ADCS[2:0]	ADCON1[6]:ADCON0[7:6]	ADC conversion clock select bits (selects clock source for ADC successive approximation cycles)
PCFG[3:0]	ADCON1[3:0]	ADC port configuration control bits (selects number of analog channels and ADC references)
ADFM	ADCON1[7]	0 = left justified in ADRESH:ADRESL 1 = right justified in ADRESH:ADRESL
ADIE	PIE1[6]	ADC interrupt enable
ADIP	IPR1[6]	ADC interrupt priority select
ADIF	PIR1[6]	ADC interrupt interrupt flag

**FIGURE 12.9** Summary of PIC18 ADC configuration registers.

## PIC18 ADC Operation

After the ADC channel, reference voltages, conversion clock, and interrupt enable have been properly configured, the PIC18 ADC is ready to convert the analog voltage on the PIC's package pin into a 10-bit unsigned binary number. The conversion process is started by setting the GO/DONE# bit (ADCON0[2]). This bit also serves as the ADC conversion complete flag; the GO/DONE# bit will remain high until the ADC is finished with its conversion cycle. After starting an ADC conversion, the PIC can directly poll the GO/DONE# bit until it clears. When the GO/DONE# bit is cleared, the 10-bit value in the register pair ADRESH:ADRESL represents the voltage on the ADC input pin as an unsigned binary number.

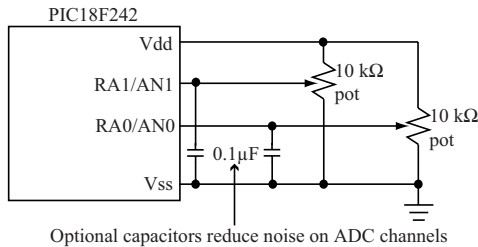
Another way to determine when the ADC conversion is complete is to use the ADC interrupts. This method allows the PIC to continue with other processing until the ADC conversion is finished and an interrupt is automatically generated. While the ISR is servicing the ADC interrupt request, the ISR must clear the ADIF bit and copy the ADC results from ADRESH:ADRESL for use by the ISR or some other part of the program.

If the next ADC conversion is to be done on another input channel, the new ADC channel must be selected by the CHS2:CHS0 bits. If the next conversion is to be done on the same channel, the CHS2:CHS0 bits do not need to be modified. Regardless of the method of determining when the ADC is finished with its conversion and which ADC channel is being used in the next conversion, setting the GO/DONE# bit starts another conversion cycle.

A potentiometer, often called a *pot*, is a variable resistor and is one of the simplest ways to generate various analog voltages. A potentiometer usually has three terminals. Between two terminals is the potentiometer's full resistance, typically a round number like 1 k $\Omega$ , 10 k $\Omega$ , or 50 k $\Omega$ . The potentiometer's third terminal is connected to the pot's wiper. The resistance between the wiper terminal and the other pot terminals changes as you turn the pot's knob. Potentiometers are often used as an input on a device's front panel when there is a need to allow the user to enter a fine-grain adjustable value. Figure 12.10 shows how the PIC18 can be connected to two potentiometers. The two capacitors connected to the pot wiper terminals are not required, but are helpful in reducing noise and for providing a more stable voltage at the PIC ADC input pins.

Figure 12.10 also shows segments of code that are used to read the voltage from both potentiometers and display the digital code and voltages on the screen via the serial interface. The code required for the serial interface (`getch()` and `printf()`) is found in Chapter 9, "Asynchronous Serial IO," and is not reproduced here. Upon entering `main()`, the first two `C` lines configure the PIC ADC. Setting the ADON bit in ADCON0 turns on the ADC. The combined ADCS2:ADCS0 bits in the ADCON0 and ADCON1 registers configure the ADC to use the internal ADC

oscillator. Clearing ADFM in ADCON1 configures the ADC to return the result left justified in ADRESH:ADRESL. The 4-bit field PCFG3:PCFG0 in ADCON1 is 0b0100 to select three ADC channels (on AN0, AN1, and AN3) using Vdd and Vss as reference voltages. It should be noted here that this code relies on the power-on reset condition of the TRISA register. (Any pin that is used for an ADC input must be configured to be an input in the corresponding port direction register.) After configuring the serial port, the program enters an infinite loop where each keypress initiates a read of ADC channels AN0 and AN1. The results are printed to the screen as a decimal number and the corresponding voltage.



Compile code with "-lf" option because of double in printf statement. Normal include files are not shown.

```
// use ADC to convert channel denoted by c and returns full 10 bit result
// NOTE: Assumes ADC is already on and proper ADC clk is selected
int ReadAdcChannel(unsigned char c) {
    unsigned char temp;

    temp = ADCON0 & 0xC1; // get current ADCON0 (clearing CHS2:CHS0)
    ADCON0 = temp | ((c & 0x7)<<3); // select the desired ADC channel
    DelayUs(20); // wait 20 us for ADC to get ready
    GODONE = 1; // start ADC conversion,
    while(GODONE); // then wait for ADC to finish
    if ( ADFM )
        return ((ADRESH << 8) | ADRESL;
    else
        return ((ADRESH << 2) | (ADRESL>>6) );
} // end ReadAdcChannel

void main(void) {
    int adc0, adc1;
    double va0, va1;

    ADCON0 = 0xC1; // turn on ADC and use internal ADC clk
    ADCON1 = 0x04; // int.ADC clk, 1.justify, AN0+AN1+AN3, Vref=Vdd
    serial_init(95,1); // 19200 in HSP1L mode, crystal=7.3728 MHz
    printf("Hit any key to read ADCs..."); pcrLf();
    while(1) {
        getch(); // wait for user to press any key
        adc0 = ReadAdcChannel(0); // get AN0 value and
        va0 = 5.0*adc0/1024; // convert into voltage
        adc1 = ReadAdcChannel(1); // get AN1 value and
        va1 = 5.0*adc1/1024; // convert into voltage
        // give user feedback about results
        printf("AN0=%4d (%1.3fV) AN1=%4d (%1.3fV)",adc0,va0,adc1,va1);
        pcrLf();
    } // end while
} // end main()
```



**FIGURE 12.10** PIC18 ADC converting two channels.

**Sample Question: What is the smallest voltage that the PIC circuit in Figure 12.10 can resolve?**

*Answer:* The code in Figure 12.10 initializes the PCFG[3:0] bits to 0b0100, which configures the PIC ADC to have three ADC channels AN3, AN1, and AN0 and  $V_{REF+} = V_{DD}$  and  $V_{REF-} = V_{SS}$ . If the PIC power supply  $V_{DD} = 5$  V, the PIC ADC's resolution is

$$\text{resolution} = \frac{V_{REF+} - V_{REF-}}{\text{precision}} = \frac{5 \text{ V} - 0 \text{ V}}{2^{10}_{\text{levels}}} = \frac{5 \text{ V}}{1024} = 4.88 \text{ mV}$$

The ADC conversion process is controlled and started in the `ReadAdcChannel1(n)` routine. This routine expects the desired ADC channel as its input. First, the `ADCON0` register is read into a temporary variable while clearing the channel select bit field (`CHS2:CHS0`). The new ADC channel is OR-ed into the temporary variable and written to `ADCON0`. The preceding step is careful not to disturb any other bits that may have been set or cleared by other routines. After the new ADC channel is selected, the PIC must wait at least 20  $\mu\text{s}$  for the new input voltage to appear and settle on the internal comparator. This is required because any change in the ADC channel is very likely to correspond to a change in voltage, and the ADC input holding capacitor requires time to acquire the new charge and voltage. The PIC datasheet provide more details, but suffice it to say that any ADC will require some time to settle when switching channels. Setting the `GO/DONE#` bit (`GODONE = 1`) in `ADCON0` starts the conversion process on the new voltage. The `GO/DONE#` bit also serves as a flag and remains high until the ADC is done. When `GO/DONE#` is cleared, the 10-bit ADC result is read into a 16-bit `int` data type and returned to the caller. Of course, the function `ReadAdcChannel1()` could be modified to return either left- or right-justified ADC results, but the function in Figure 12.10 is more general with little performance and space penalty.

Figure 12.11 shows a sample of the terminal output when using the circuit and code in Figure 12.10. Your results will be different because it is unlikely that you can mimic exactly the pot positions used to create Figure 12.11. Try connecting a digital voltmeter to the `RA0/AN0` and `RA1/AN1` pins on the PIC. The voltmeter readings will not match those computed by the PIC exactly. The voltages should be close, but will differ due to noise on the ADC lines and reference voltage differences in the voltmeter and PIC's ADC.

Note the changes in the ADC results on the last few readings in Figure 12.11. These reading were taken without adjusting the potentiometer settings. Recall that the ADC step size is very small, approximately 5 mV. The difference in readings is likely due to noise and dynamic comparator bias in the PIC ADC. It is very easy for these types of errors to be larger than 5 mV; therefore, ADC results are rarely "constant" when the step sizes are so small. Finally, the 10-bit ADC result is typically too

fine for use with many lower cost potentiometers. The low-cost pots, especially the “one-turn” pots, cannot generate voltages accurately enough for 10-bit conversion. The upper 8 bits (ADRESH when ADFM = 0 for right justification) of the ADC result is usually sufficient for these potentiometers.

```

Hit any key to read ADCs...
AN0= 673 (3.286V)    AN1=1017 (4.966V) }
AN0= 682 (3.330V)    AN1= 756 (3.691V) } turn AN1 pot while pressing a key
AN0= 680 (3.320V)    AN1= 297 (1.450V) }
AN0= 683 (3.335V)    AN1=  0 (0.000V) }
AN0= 680 (3.320V)    AN1=  0 (0.000V) }
AN0= 721 (3.521V)    AN1=  0 (0.000V) }
AN0= 901 (4.399V)    AN1=  2 (0.010V) }
AN0=1020 (4.980V)    AN1=  3 (0.015V) }
AN0= 779 (3.804V)    AN1=  1 (0.005V) } turn AN0 pot while pressing a key
AN0= 583 (2.847V)    AN1=  2 (0.010V) }
AN0= 305 (1.489V)    AN1=  0 (0.000V) }
AN0= 142 (0.693V)    AN1=  0 (0.000V) }
AN0= 135 (0.659V)    AN1= 177 (0.864V) }
AN0= 140 (0.684V)    AN1= 342 (1.670V) } turn AN1 pot while pressing a key
AN0= 139 (0.679V)    AN1= 680 (3.320V) }
AN0= 143 (0.698V)    AN1= 707 (3.452V) }
AN0= 141 (0.688V)    AN1= 710 (3.467V) }
AN0= 141 (0.688V)    AN1= 710 (3.467V) }
AN0= 141 (0.688V)    AN1= 708 (3.457V) } pressing a key repeatedly without
AN0= 142 (0.693V)    AN1= 707 (3.452V) } turning potentiometers
AN0= 137 (0.669V)    AN1= 708 (3.457V) }
AN0= 143 (0.698V)    AN1= 709 (3.462V) }
    
```

**FIGURE 12.11** Sample terminal output for ADC example.

The code in Figure 12.10 polled the ADC’s status bit via the statement `while(GODONE){}` to wait on the ADC to complete the conversion. This method, while effective, is inefficient. A more efficient use of the processor’s power is to be doing computations while the ADC is converting the channel. The code in Figure 12.12 uses the ADC circuit in Figure 12.10 and produces output similar to Figure 12.11. The major difference is that the `main()` routine can do useful work while the ADC conversion process is ongoing because an interrupt is used for returning the ADC value. If the portion of `main()` after the call to `StartAdcCycle()` is complex, the ADC result is ready for use when the `main()` reaches the `printf()` statement. While this example is a bit contrived, ADC completion interrupts are powerful when used in concert with timers. We will explore how timers and data converters work together later in this chapter and also in Chapter 14, “Capstone: Audio Sampling, Monitoring System, and Autonomous Robot,” which contains an example of using the PIC18 ADC and timers to sample voice input.

```

#define ADC_AN0 0           Compile code with "-If" option because of double in printf
#define ADC_AN1 1           statement. Normal include statements not shown.
#define ADC_IDLE 255       used by main() and pic_isr, so make global
volatile unsigned char AdcState; } and denote as volatile
volatile int  adc0, adc1;

void interrupt pic_isr(void) {
    unsigned char temp;
    if (ADIF) {
        ADIF = 0;
        switch (AdcState) {
            case ADC_AN0:
                adc0 = (ADRESH << 8) | ADRESL; // get AN0 result from ADC
                temp = ADCON0 & 0xC1; // clear bits except ADCS1:ASCS0 and ADON
                ADCON0 = temp | 0x08; // select AN1 on RA1 (for next ADC op)
                AdcState = ADC_IDLE; // goto IDLE state so main() will continue
                break;
            case ADC_AN1:
                adc1 = (ADRESH << 8) | ADRESL; // get AN1 result from ADC
                temp = ADCON0 & 0xC1; // clear bits except ADCS1:ASCS0 and ADON
                ADCON0 = temp; // select AN0 on RA0 (for next ADC op)
                AdcState = ADC_IDLE; // goto IDLE state so main() will continue
                break;
        } // end switch
    } // end if
} // end pic_isr()

void StartAdcCycle(unsigned char c) {
    AdcState = c; GODONE = 1;
} // end StartAdcCycle()

void main(void) {
    double va0, val;
    ADCON0 = 0xC1; // turn on ADC and use internal ADC clk
    ADCON1 = 0x84; // int.ADC clk, R.justify, AN0+AN1+AN3, Vref=Vdd
    serial_init(95,1); // 19200 in HSPLL mode, crystal=7.3728 MHz
    AdcState = ADC_IDLE;
    IPEN = 0; ADIE = 1; PEIE = 1; GIE = 1;
    printf("Hit any key to read ADCs..."); pcrLf();
    while(1) {
        c = getch();
        StartAdcCycle(ADC_AN0);
        printf("AN0=");
        while (AdcState != ADC_IDLE);
        va0 = 5.0 * adc0/1024;
        StartAdcCycle(ADC_AN1);
        printf("%4d (%1.3fV) and AN1=", adc0, va0);
        while (AdcState != ADC_IDLE);
        val = 5.0 * adc1/1024;
        printf("%4d (%1.3fV)", adc1, val); pcrLf();
    } // end while
} // end main()

```



**FIGURE 12.12** ADC example using ADC completion interrupts.

## 12.5 DIGITAL-TO-ANALOG CONVERSION

Just as there are many different ways to convert an analog quantity to a digital code in an ADC, designers have invented many ingenious methods to convert a digital code back into an analog signal in DACs. Since DACs perform the complementary

operation of ADCs, these two data converters have much in common. Like ADCs, the digital codes accepted by DACs follow many different coding schemes, although unsigned and signed binary representations are the most popular. Furthermore, digital codes are provided to the DAC via many different communication protocols—fully parallel, serially via nibble-wide parallel transfers, and bit serial, including I<sup>2</sup>C, SPI, and many others. Also like ADCs, DACs exist that create different analog quantities for their output, but the most common is the voltage output DAC.

Like the ADCs, DACs are also characterized by a huge number of parameters. Many of the DAC parameters have the same names as the ADC parameters. However, the DAC parameters have a slightly different meaning as the two devices perform complimentary functions. A DAC's precision is the number of output levels that the DAC can create. Like the ADC, DAC precision is represented as the number of output levels or the number of bits required to encode the number of output levels. DAC resolution is the smallest distinguishable change in the output, and represents the change in output from a  $\pm 1$  LSB change in DAC input. The DAC range is the total span over which DAC outputs can occur. DAC range can be computed through the DAC's reference inputs; for example,  $range = (V_{REF+} - V_{REF-})$  for a voltage DAC. Obviously, the DAC has digital inputs so it must have a digital clock input to signal when the input sample data is valid. The time between each DAC conversion is the DAC's period, and is almost always the same as the ADC's sample period. Assuming that our DAC in Figure 12.1 is an  $n$ -bit voltage output DAC with reference voltages  $V_{REF+}$  and  $V_{REF-}$ , the DAC output  $y(t)$  is shown in Equation 12.2.

$$\begin{aligned}
 y(t) &= \frac{\text{DAC input code}}{\text{precision}} \cdot \text{range} + \text{lower reference} & (12.2) \\
 &= \frac{y[k]}{2^n} \cdot (V_{REF+} - V_{REF-}) + V_{REF-}
 \end{aligned}$$

Although most DACs strive to create the input-output characteristic of Equation 12.2, the methods by which they achieve these results and the circuits they use vary widely. Each approach has advantages and disadvantages. You are encouraged to read about some of the other DAC architectures such as interpolating, charge sharing, current steering, and delta-sigma DACs. In this chapter, we will examine a DAC popular in small microprocessor applications: the flash DAC. A second DAC architecture, called a pulse-width modulation (PWM) DAC, is covered in Chapter 13 in conjunction with the timer discussion.

**Sample Question:** What is the voltage represented by an ideal 4-bit DAC with input 0xC and  $V_{REF-} = 0\text{ V}$  and  $V_{REF+} = 4\text{ V}$ ?

**Answer:** The DAC has a precision of  $2^4 = 16$  levels, and a range of  $(4\text{ V} - 0\text{ V}) = 4\text{ V}$ . The DAC input 0xC, or 12, will generate an output voltage of

$$y(t) = \frac{12}{16} \cdot 4\text{V} + 0\text{V} = 3.0\text{V}$$

Compare the reconstructed DAC result here with the ADC examples earlier in this chapter. The ADC input voltage 3.14159 V is converted to a 4-bit digital value 0xC. A 4-bit DAC using the same reference voltages converts the digital sample back into the voltage 3.0 V. The difference between the two voltages (e.g., 0.14159 V) is the quantization error. If we need voltage measurements that are more accurate, we would need to use an ADC and DAC with more precision.

## Flash DACs

A flash digital-to-analog converter, sometimes called a parallel DAC, is characterized by its capability to generate an output within a single clock cycle. The speed of a flash DAC is achieved by the parallel generation of a set of fixed references. The set of references is complete; they are capable of constructing all of the possible DAC output values. Thus, any desired voltage output can be created nearly instantly, making flash DACs very fast. There are many different ways to go about creating these reference voltages and this gives rise to different flash DAC architectures. We will examine two closely related flash DACs: the resistor string and resistor ladder flash DACs.

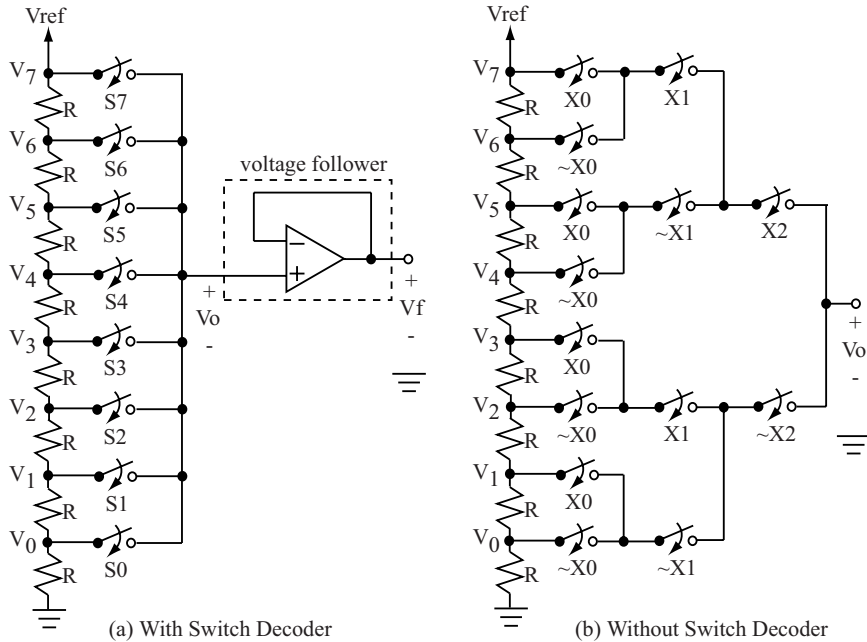
## Resistor String Flash DACs

Resistor string digital-to-analog converters use a resistor voltage divider network, connected between two reference voltages, to generate a complete set of output voltages. Each voltage divider tap in the resistor string corresponds to a DAC input code. An  $n$ -bit resistor string flash DAC uses at least  $2^n$  resistors. Some designs use additional resistors to create more accurate reference voltages, or voltages that correspond to rounded rather than truncated code values. Switches, controlled by the DAC's digital input, select the appropriate reference voltage to connect to the DAC output.

Resistor string DACs are available with many different input code word lengths. As an example, let's consider a 3-bit resistor string flash DAC architecture like the one in Figure 12.13. The resistor string divides the DAC reference voltage,  $V_{REF}$ , into  $2^3 = 8$  equally spaced voltages,  $V_0, V_1, \dots, V_7$ . The DAC architecture in Figure 2(a) uses  $2^3 = 8$  switches to connect the appropriate voltage to the DAC



output,  $V_o$ . The switch control signals,  $S_0, S_1, \dots, S_7$  are generated by a 3:8 decoder, which is not shown in Figure 12.13. As the number of bits in the DAC's input code increases, more and more switches are connected to the DAC's output  $V_o$ . This increases the capacitance at the DAC output node, making the DAC slower and limiting its maximum operating frequency.



**FIGURE 12.13** Resistor string flash DAC architecture.

An alternative resistor string DAC architecture in Figure 12.13(b) arranges the switches into a binary tree structure. This architecture does not need the dedicated 3:8 decoder. Decoding is inherent in the binary tree arrangement of the switches that are controlled by the DAC's digital input code bits,  $X_0, X_1, X_2$ , and their complements,  $\sim X_0, \sim X_1, \sim X_2$ . Furthermore, parasitic capacitance at the DAC output is reduced and operating speeds increased since the output is connected to fewer switches than the DAC in Figure 12.13(a).

A major disadvantage of both resistor string flash DACs in Figure 12.11 is the stringent voltage string resistor matching requirements. Since the DAC voltage division determines output voltages, each resistor must be almost perfect or every reference voltage will be incorrect. The number of resistors needed for larger DACs (e.g., eight or more bits), and the limitations of VLSI fabrication technology make

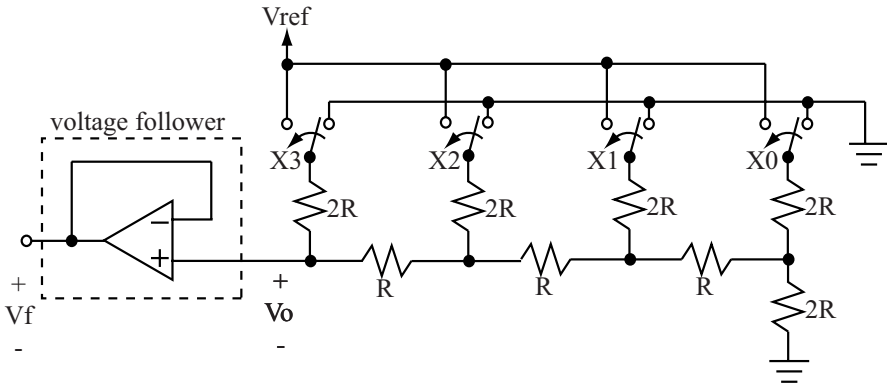
it difficult to create so many accurate and small resistors at an affordable price. Another disadvantage of the DACs in Figure 12.13 is their inability to drive a load without a buffer. If the DAC's load, which can be modeled as a resistor, draws much current, a current divider circuit is created. The DAC load will siphon current out of the voltage divider string and cause the voltages below the connection point to become inaccurate. The voltage follower shown in Figure 12.13(a) creates a copy of  $V_o$  at  $V_f$  without drawing much current from the resistor string (a voltage follower would also be used with the architecture of Figure 12.13(b)). Yet another disadvantage is power consumption. Since current is always flowing through the voltage divider, power is constantly being dissipated. Although the resistor value,  $R$ , can be increased to reduce power losses, larger resistors occupy more chip area.

In spite of these disadvantages, resistor string DACs are attractive because they guarantee *monotonicity*—the property that an increase in the DAC digital input code causes an increase in the DAC's analog output. Finally, the resistor string DACs in Figure 12.13 can be very fast because of the parallel nature of their design. Conversion speed is limited by the decoder speed (if present), switch speeds, and settling time and slew rate of any output amplifiers. Therefore, resistor string DACs are used in many high bandwidth applications such as digital video, RADAR, and communications.

### **R-2R Resistor Ladder Flash DAC**

Another DAC related to the resistor string flash DAC is the R-2R resistor ladder flash DAC. The R-2R resistor ladder DAC also uses voltage division to generate the DAC's output voltage, but does so in a clever way that uses many fewer resistors than the resistor string DAC. Instead of generating all possible voltage outputs, the resistor ladder DAC effectively rearranges its voltage divider network based on the DAC's digital input code. An  $n$ -bit R-2R resistor ladder flash DAC uses at least  $2n$  resistors and  $n$  switches. You can see that the resistor ladder DAC uses a much smaller number of components than the same size resistor string flash DAC, especially as the DAC input code word length  $n$  gets large. The DAC's digital input code bits control switches that make connections between resistors and virtually rearrange the resistor ladder network to form the DAC's output voltage. The R-2R resistor ladder flash DAC is especially well suited for use with a small microprocessor like the PIC18.

Just like all of the data converters that we have already examined, the resistor ladder DAC comes in many different word lengths. For discussion, let's look at the 4-bit R-2R resistor ladder DAC in Figure 12.14. This resistor ladder DAC uses four switches, three resistors of  $R$  ohms, and five resistors of  $2R$  ohms. The four switches connect the appropriate power supply voltage to the  $2R$  resistors.

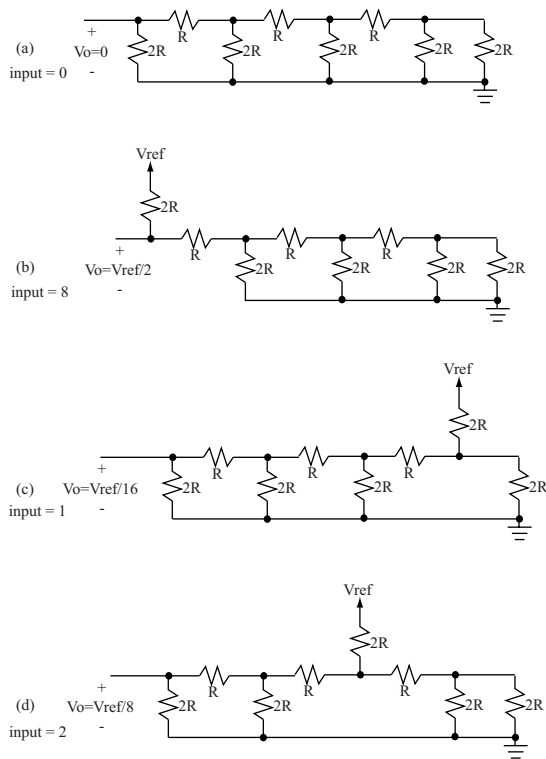


**FIGURE 12.14** R-2R resistor ladder flash DAC architecture.

The high supply voltage  $V_{dd}$  is connected to each  $2R$  resistor if the corresponding digital input code bit,  $X_0$ ,  $X_1$ ,  $X_2$ , or  $X_3$ , is “1”. Otherwise, if the digital input code bit is “0”, the lower supply voltage, usually ground, is connected to the appropriate  $2R$  resistor. While not obvious right now, the switch states will create an output voltage that is proportional to the digital input code. Like the resistor string DACs in Figure 12.13, the resistor ladder DAC in Figure 12.14 usually requires a voltage follower to prevent excessive current siphoning from the ladder that would cause voltage output errors.

To see how the resistor ladder DAC works, let’s consider a few examples. Consider the case when the 4-bit input code  $X_3 X_2 X_1 X_0$  is 0b0000. Since all four input bits are “0”, each  $2R$  resistor is connected to the lower supply voltage—ground in this example. The resistor ladder in Figure 12.15a has been redrawn to emphasize this fact. Obviously, the output voltage  $V_o$  must be 0 V since there is no other voltage source in the circuit.

Now, consider the case when the input code is 0b1000 (8). Figure 12.15(b) shows the equivalent circuit when the input is 0b1000. Applying voltage division repeatedly, it is found that  $V_o$  is  $V_{REF}/2$ . When the input code is 0b0001 (1) in Figure 12.15c, repeated application of voltage division gives  $V_o = V_{REF}/16$ . When the input code is 0b0010 (2) as shown in Figure 12.15d, we see that  $V_o = V_{REF}/8$ . Since the R-2R resistor ladder DAC uses only linear resistors, the *superposition* theory applies. Superposition says that we can find a system’s response to several inputs by simply adding up the individual output responses to each input acting alone. Therefore, if the R-2R resistor ladder DAC in Figure 12.14 has more than one switch connected to  $V_{REF}$ , we can find  $V_o$  by summing the appropriate individual responses in Figure 12.15. For example, if the input code is 0b1001, the resistor ladder output voltage  $V_o$  is the sum of the responses at  $V_o$  for the two individual cases when the sources act alone. So, we find that  $V_o = V_{REF}/2 + V_{REF}/16 = 9 V_{REF}/16$ .

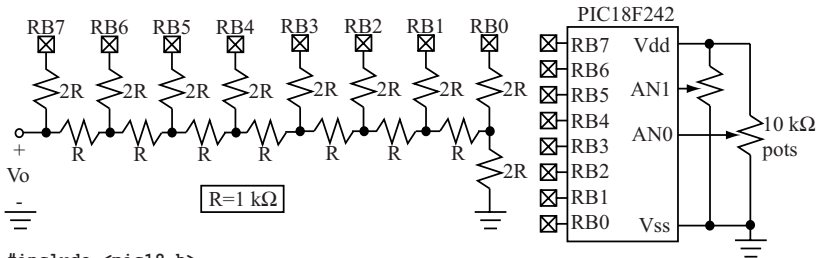


**FIGURE 12.15** Four-bit R-2R resistor ladder example.

To generalize this result to an  $n$ -bit resistor ladder DAC, we find that for a digital input code of  $X$ , the resistor ladder output voltage  $V_o$  is  $(X/2^n) V_{\text{REF}}$ , which is the desired result. The resistor ladder DAC generates an output voltage that is linearly proportional to the digital input code.

The resistor ladder DAC circuit in Figure 12.14 is actually much simpler to put into practice than it first appears. Switches and voltage supplies are not really needed because the PIC18 conveniently provides these in the form of IO port pins. When the PIC18 is driving a “1” from one of its IO pins, it is connecting that pin to the PIC18’s internal  $V_{\text{dd}}$ . When the PIC18 is pulling down, or driving, its IO pin to “0”, the PIC18 is really connecting that IO pin to its internal ground. Therefore, the PIC18 IO pins can be used to replace the switches,  $V_{\text{dd}}$ , and ground connections in Figure 12.14. Therefore, the PIC18 can very easily build an 8-bit R-2R resistor ladder flash DAC with 16 resistors as shown in Figure 12.16. Simply writing the digital code  $X$  to PORTB will create the voltage  $(X/256)V_{\text{dd}}$  at  $V_o$ . The resulting voltage is based on  $V_{\text{dd}}$  because the PIC18’s  $V_{\text{dd}}$  is our resistor ladder DAC’s  $V_{\text{REF}}$ . Of course, other PIC18 IO ports can be used instead of PORTB, and

the DAC input word can be shorter or longer depending on the application. But beware: whenever you split the DAC input word across several IO ports, it will take several PIC18 instruction cycles to update the entire DAC value. During this time, the DAC output voltage will likely be grossly incorrect.



```
#include <pic18.h>
volatile unsigned char  iper, iamp;
unsigned char  idx;
const unsigned char sinetbl[] = {127,133,139,146,152,158,164,170,176,181, \
187,192,198,203,208,212,217,221,225,229,233,236,239,242,244,247,249,250, \
252,253,253,254,254,254,253,253,252,250,249,247,244,242,239,236,233,229, \
225,221,217,212,208,203,198,192,187,181,176,170,164,158,152,146,139,133, \
127,121,115,108,102,96,90,84,78,73,67,62,56,51,46,42,37,33,29,25,21,18, \
15,12,10,7,5,4,2,1,1,0,0,0,1,1,2,4,5,7,10,12,15,18,21,25,29,33,37,42,46, \
51,56,62,67,73,78,84,90,96,102,108,115,121 };

void interrupt pic_isr(void) {
    unsigned char temp;
    if (TMR2IF) {
        temp = sinetbl[idx]; // get sine fcn value
        temp >>= iamp; // reduce amplitude based on AN1 input
        PORTB = temp; // write new DAC value
        TMR2IF = 0; // clear IRQ flag
        PR2 = 250-iper; // change TMR2 period based on AN0 input
        idx++; idx &= 0x7F; // fix ptr for next update; verify in range
    } // end if
} // end pic_isr

void main(void) {
    int temp;
    TRISB = 0; // make PORTB digital outputs
    ADCON0 = 0xC1; ADCON1 = 0x04; // ADON,int.ADC clk,AN0+AN1+AN3,Vref=Vdd
    idx = 0; iper = 0; iamp = 0;
    PR2 = 250; T2CON = 0x04; TMR2IF = 0; ← TMR2 matches every 34 μs
    IPEN = 0; TMR2IE = 1; PEIE = 1; GIE = 1; ← enable TMR2 interrupts
    while(1) {
        temp = ReadAdcChannel(0); } Read AN0 pot; scale to 0-127 range for use as itmr
        iper = temp>>3; } Wait 50 ms since user interface need not be fast.
        DelayMs(50);
        temp = ReadAdcChannel(1); } Read AN1 pot; scale to 0-3 range for use as iamp
        iamp = temp>>8; } Wait 50 ms since user interface need not be fast.
        DelayMs(50);
    } // end while
} // end main()
```



**FIGURE 12.16** Eight-bit R-2R resistor ladder DAC using PIC18’s IO port B.

If our DAC's load is purely capacitive or has a very large resistance, the circuit in Figure 12.16 works very well. However, the current flowing through the resistor ladder network is crucial to forming the voltage at  $V_o$ . Therefore, the R-2R resistor ladder DAC cannot be loaded by any circuit element that draws appreciable current. If a current drawing load is connected, the load will siphon current out of the resistor ladder and cause distortion at  $V_o$ . To prevent excessive current draw, we simply attach a voltage follower at  $V_o$ , just like the resistor string flash DAC in Figure 12.14. Furthermore, capacitance can be added to the voltage follower's feedback path to create an active low-pass filter to smooth out the jagged stair-step pattern visible at  $V_o$ . The low-pass filter at a DAC's output is sometimes called a *reconstruction filter* by digital signal processing experts.

Figure 12.16 lists code that creates a simple sinusoid function generator with the PIC18. The potentiometers on AN0 and AN1 control the sinusoid's frequency and amplitude, respectively. The PIC's internal ADC reads the voltage provided by each pot every 100 ms. The relatively slow update is fine for this kind of user interface function. In operation, the PIC seems to respond instantly to the changes in potentiometer settings. The code in Figure 12.16 uses Timer2 to create periodic interrupts so that the R-2R resistor ladder DAC on PORTB can be written with uniform periods between updates. At each Timer2 interrupt, the ISR gets a value from the sine lookup table. The sine data can be easily replaced with other values so that the PIC can create any arbitrary waveform like sawtooth and chirp signals. In fact, the lookup table can easily be changed to represent a single cycle of a saxophone recording making the PIC a simple music synthesizer. The lookup table `sinetb1` in Figure 12.16 contains 128 entries. The number of values can be increased to give the waveform more detail. The `const` modifier for `sinetb1` tells the compiler that this array contains constant data (the array values are not changed), and thus this data is stored in program memory instead of data memory. This is useful, especially for large lookup tables, which may not fit in data memory. During each ISR call, the lookup table value is read into a temporary variable, right shifted based on the value in `iamp` for amplitude scaling, and then written to PORTB to update the waveform via the R-2R DAC. Then, the Timer2 period register PR2 is updated with a new period based on `iper`. Larger values of `iper` reduce the timer's period. Finally, the ISR clears the Timer2 interrupt flag and updates the lookup table pointer for the next call.

During program initialization in Figure 12.16, the Timer2 period is set to 251 instruction cycles ( $PR2+1 = 250+1 = 251$ ) with 1:1 for prescaler and postscaler values. The value `iper` in the program can be any integer value between 0 and 127. With  $FOSC = 29.4912$  MHz (crystal of 7.3728 MHz and HSPLL option), this configuration generates a Timer2 interrupt every 34  $\mu$ s as shown in Equation 12.3.

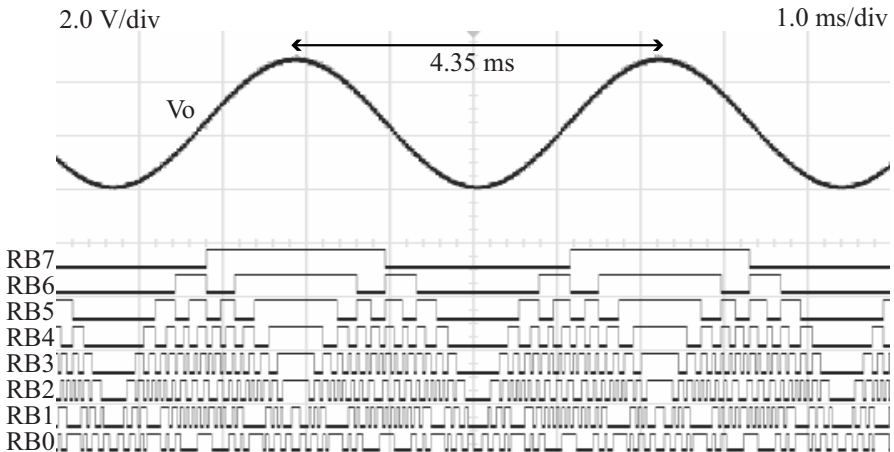
$$\text{TMR2 period} = \left[ \frac{251 \text{ instructions}}{(29,491,200/4) \text{ instructions/sec}} \right] \approx 34 \mu\text{s} \quad (12.3)$$

When  $i_{per} = 127$  ( $PR2 = 250 - 127 = 123$ ), the Timer2 interrupt occurs every  $16.8 \mu s$ . Since the lookup table must be accessed 128 times to create a full period of the sine waveform, the PIC will generate sinusoids with periods between  $2.15 \text{ ms}$  (Equation 12.4) and  $4.35 \text{ ms}$  (Equation 12.5).

$$[(128 \text{ updates})(16.8 \mu s/\text{update})] = 2.15 \text{ ms} \quad (12.4)$$

$$[(128 \text{ updates})(34 \mu s/\text{update})] = 4.35 \text{ ms} \quad (12.5)$$

Therefore, the circuit in Figure 12.16 generates sinusoids with frequencies between  $230 \text{ Hz}$  and  $465 \text{ Hz}$  with waveform amplitudes of  $V_{dd}$ ,  $V_{dd}/2$ ,  $V_{dd}/4$ , and  $V_{dd}/8$  Figure 12.17 shows the signals from PORTB and the DAC's  $V_o$  from the circuit in Figure 12.16.



**FIGURE 12.17** Output of 8-bit R-2R resistor ladder DAC.

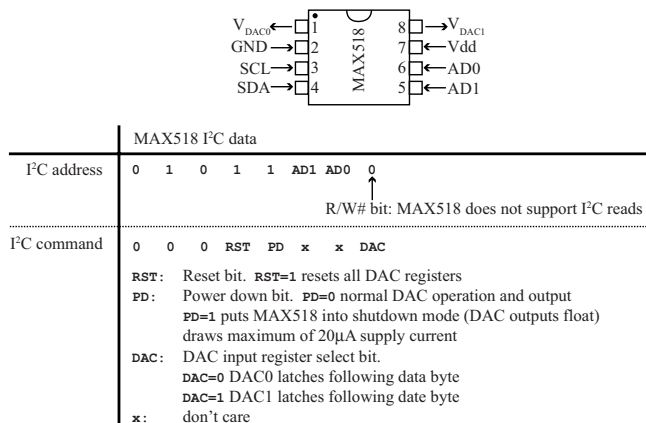
## 12.6 DIGITAL-TO-ANALOG CONVERTER EXAMPLE: THE MAXIM 518

DACs can be constructed with any of the architectures introduced in the previous section and numerous other architectures not discussed in this book. DAC architecture selection is usually application specific. When choosing an external DAC, several options should be considered; for example, number and value of external reference voltages, word length, number of channels, input communication scheme, and chip package type are just a few. Many of these parameters are

interrelated. If a single channel 24-bit DAC with parallel inputs is chosen, the DAC package will have at least 28 pins: 1 pin each for  $V_{dd}$ ,  $V_{ss}$ ,  $V_o$ , 24 pins for input data, and 1 pin for input latch or clock. If the DAC supports external reference voltages for the analog output or multiple output channels, the number of package pins rises quickly.

Parallel port IO pins are usually a microprocessor's most limited and expensive resource since a chip package with a few more pins is far more expensive than the corresponding increase in silicon area costs to support those extra pins. Because parallel IO pins are so precious, many common external components choose to use pin-saving schemes for communication like the synchronous serial IO interfaces SPI and I<sup>2</sup>C introduced in Chapter 11. In this section, we introduce the Maxim Integrated Products MAX518, a dual 8-bit DAC with an I<sup>2</sup>C bus interface. With this chip, we combine our newfound knowledge of I<sup>2</sup>C and DACs together to create a PIC circuit with two input ADC channels and three output DAC channels. We use this circuit to build a simple three-function waveform generator with control of the frequency and amplitude of the waveforms. The generated waveforms are sinusoid, square wave, and sawtooth.

The Maxim Integrated Products MAX518 is a dual 8-bit DAC with an I<sup>2</sup>C bus interface. Figure 12.18 shows the MAX518 package pin diagram and lists the MAX518 I<sup>2</sup>C address and command byte formats. The MAX518 external pins AD0 and AD1 determine the device's I<sup>2</sup>C address. The remaining five most significant device address bits are set internally to 0b01011. Recall from Chapter 11, "Synchronous Serial IO," that the LSB of the I<sup>2</sup>C address is the R/W# bit. The I<sup>2</sup>C master can only write to the MAX518, so the LSB of the I<sup>2</sup>C address is always "0". After the I<sup>2</sup>C master sends a valid MAX518 address, the MAX518 expects at least one command byte and a data byte.

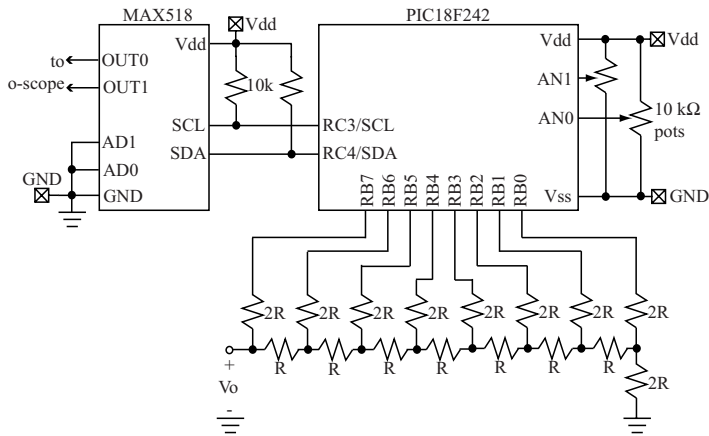


**FIGURE 12.18** MAX518 pin diagram and I<sup>2</sup>C information.



Figure 12.18 shows the format of the MAX518 command byte. The LSb of the command byte determines the DAC that latches the data byte that follows the command byte. The MAX518 DAC output voltage  $V_{DAC0}$  or  $V_{DAC1}$  changes upon the I<sup>2</sup>C stop condition. Since the MAX518 is a dual DAC, both DACs can be set in one I<sup>2</sup>C transaction composed of the following I<sup>2</sup>C transfers: start condition, I<sup>2</sup>C device address, command byte, data byte, command byte, data byte, stop condition. In this case, both DAC outputs change simultaneously upon the I<sup>2</sup>C stop condition.

Figure 12.19 shows the circuit used to generate the three waveforms. The R-2R DAC resistor ladder flash DAC connected to PORTB is the same circuit as shown in Figure 12.16. The code to operate the three-function waveform generator is shown in Figure 12.20. Much of the code to support the R-2R DAC in Figure 12.20 is the same as the code in Figure 12.16. The 10 k $\Omega$  potentiometer circuit in Figure 12.19 is the same as Figure 12.16. New in Figure 12.19 is the MAX518 dual 8-bit DAC with I<sup>2</sup>C interface that is connected to the PIC18 via the I<sup>2</sup>C bus on RC3/SCL and RC4/SDA pins.



**FIGURE 12.19** Three-function waveform generator circuit.

The program begins in `main()` by initializing the direction of the IO ports as needed for the R-2R DAC, the A/D converters, and the I<sup>2</sup>C bus. The I<sup>2</sup>C functions introduced in Chapter 11 are used again and the I<sup>2</sup>C USART is initialized with `i2c_init(14)`, which in conjunction with the 7.3728 MHz crystal and HSPLL option yields a 500 kps I<sup>2</sup>C data rate. This data rate exceeds the design specification of the MAX518 DAC. The MAX518 is limited to a maximum 400 kbps data rate. It is not recommended that you design products outside of the manufacturer's specifications. In the MAX518 chips tested in our labs, the higher data rate was

handled with no problem. The higher I<sup>2</sup>C data allows us to generate our sampled waveforms at higher frequencies.

```

#include <pic18.h>
#include "i2cmsu.c" ← see I2C routines and synchronous serial IO in Chapter 11

volatile unsigned char  itmr, iamp, doAdc;
unsigned char          idx, cnt;

void interrupt pic_isr(void) {
    unsigned char temp0, temp1;

    if (TMR2IF) {
        TMR2IF = 0;
        temp0 = sinetbl[idx]; } sinetbl[] is same lookup table used in Figure 12.
        temp0 >>= iamp; } reduce amplitude by right shifting; send to R-2R DAC
        PORTB = temp0;
        if (idx & 0x40) } generate 0 or 255 square wave based on idx2
            temp0=(0xFF>>iamp); } reduce amplitude by right shifting based on iamp
        else
            temp0=0;
        temp1 = idx<<1; } generate sawtooth from idx2
        temp1>>=iamp; } reduce amplitude by right shifting based on iamp
        WriteMAX518(temp0,temp1); ← send square and sawtooth wave data to DACs
        idx++; idx+=itmr; idx&=0x7F; ← { update ptr for next sample; simulate faster
        if (++cnt == 0) doAdc=1; } waveforms by increasing ptr by itmr
    } // end if
} // end pic_isr
    signal main() that it is time to read ADCs

void WriteMAX518(unsigned char c0, unsigned char c1) {
    i2c_start(); i2c_put(0x58); i2c_put(0x00); i2c_put(c0); } send c0 and c1 to
    i2c_put(0x01); i2c_put(c1); i2c_stop(); } MAX518's DAC0 and
    } // end WriteMAX518 } DAC1 via I2C

void main(void) {
    int temp;

    ADCON0=0xC1; ADCON1=0x84; // int. ADC clk, AN0+AN1+AN3, Vref=Vdd
    idx=0; cnt=0; itmr=0; iamp=0; doAdc=0;
    i2c_init(14); ← 500 Kbps datarate on I2C bus (may not work for all MAX518s)
    PR2 = 245; } set TMR2 to overflow every (245+1)*5=1230 instruction cycles (~6000 Hz)
    T2CON = 0x24; }
    TMR2IF = 0; IPEN = 0; TMR2IE = 1; PEIE = 1; GIE = 1;
    while(1) {
        doAdc=0;
        while (doAdc==0); } Read the AN0 ADC every 512 TMR2 interrupts
        temp = ReadAdcChannel(0); } Divide AN0 value by 32 to make 0 <= itmr <= 31
        itmr = temp>>5;
        doAdc=0;
        while (doAdc==0); } Read the AN1 ADC every 512 TMR2 interrupts
        temp = ReadAdcChannel(1); } Divide AN1 value by 128 to make 0 <= iamp <= 7
        iamp = temp>>8;
    } // end while
} // end main()

```



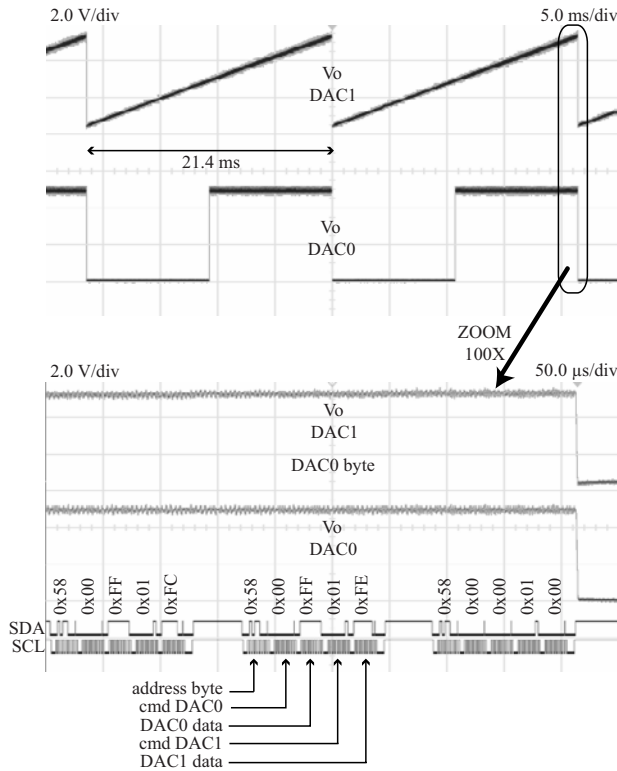
**FIGURE 12.20** Three-function waveform generator code.

After the I<sup>2</sup>C bus is initialized via the SPCON1 configuration register to be the I<sup>2</sup>C master, the Timer2 periodic interrupt is configured by setting PR2 = 245, the Timer2 prescaler to 1, and the postscaler to 1:5. With the HSPLL option and a 7.3728 MHz crystal, this produces a sample frequency of 5994 Hz (period is ~167 μs). At this point, it should be mentioned that the code in Figure 12.16 and Figure 12.20 both use Timer2 interrupts. However, the systems use Timer2 very differently. In Figure 12.16, the period pot adjusts the PR2 register at every Timer2 overflow. In this way, the period of Timer2 is reduced based on the analog voltage on AN0. While this generates a very smooth waveform, the system in Figure 12.16 is somewhat unusual in that the waveform samples are produced with a time-varying sampling interval. Most sampled data systems assume and operate with a uniform time period between sample values that does not change over time. This is precisely the approach taken by the circuit in Figure 12.19 and code in Figure 12.20. In this case, Figure 12.20 uses a sample rate of approximately 6 kHz, and this sampling frequency remains constant regardless of the desired output waveform frequency. To get the appearance of variable frequency waveforms, the Timer2 interrupt service routines must “skip” through the waveform data tables by varying amounts depending on the frequency pot voltage present on AN0.

Upon the periodic Timer2 interrupt, the PIC “services” the three DACs and prepares for the next sample generation. Examine the Timer2 service routine in Figure 12.20. The variable `temp0` is loaded with the needed value from the sinusoid lookup table via the lookup table pointer `idx`. The sinusoid value is right shifted (divided) by `iamp` if necessary. The value in `iamp` depends on the pot on AN0, as we shall soon see. The amplitude-adjusted sinusoid value is written to PORTB, and the R-2R generates the corresponding analog voltage. Next, the Timer2 ISR computes the square wave output value. If the lookup table pointer `idx` has its 7<sup>th</sup> bit set, the waveform is in its upper half of its period. If the lookup table pointer `idx` has its 7<sup>th</sup> bit cleared, the waveform is in the lower half-period. Using this test, the Timer2 ISR creates a full-scale square wave output with a sample values 0xFF and 0x00. Like the sinusoid waveform, the variable `iamp` is used to reduce the amplitude of the square wave sample value. Finally, the lookup table pointer `idx` itself is used as the sawtooth waveform data. Just like the sine and square waveforms, the sawtooth wave value is reduced by `iamp` too. The square wave and sawtooth waveform data values are sent to the MAX518 via the `WriteMAX518()` function. At this point in the Timer2 ISR, the waveform table pointer `idx` is updated in anticipation of the next sample instant, by incrementing `idx` to the next table value and possibly adding another variable increment based on the potentiometer connected to AN1. The counter `cnt` is used to set the `doAdc` flag every 256 Timer2 interrupts, regardless of the DAC waveform outputs and their frequency. The `doAdc` flag alerts the code in `main()` to sample the potentiometers on AN0 and AN1.

In Chapter 11, we saw that each I<sup>2</sup>C transfer requires that the I<sup>2</sup>C slave device address be transmitted after the start bit. Referring to the MAX518 datasheet and Figure 12.19 where the MAX518 address lines are grounded, it can be seen that the I<sup>2</sup>C address of our MAX518 is 0x58. After our MAX518 sees its address on the I<sup>2</sup>C bus, it acknowledges the I<sup>2</sup>C master (the PIC18), and waits for the command byte. The MAX518 data sheet specifies that the command byte “0x00” signals the MAX518 that data is about to be written to the DAC0 data register. Similarly, the command byte “0x01” signals the MAX518 that data is about to be written to the DAC1 data register. After the MAX518 command byte is written, the appropriate DAC data is sent over the I<sup>2</sup>C bus. The MAX518 supports writing both DAC data registers in a single I<sup>2</sup>C transfer of 5 bytes: I<sup>2</sup>C address, command to write one DAC, the DAC data, command to write the other DAC, DAC data. An operation of this form updates both MAX518 DACs simultaneously when the master sends the I<sup>2</sup>C stop condition. This I<sup>2</sup>C sequence has been combined into one function `WriteMAX518(c1,c2)`. This function requires arguments of the two unsigned bytes containing the DAC0 and DAC1 data. Figure 12.21 shows three complete I<sup>2</sup>C transactions near the end of the `idx` period. Note how the DAC output does not change until the PIC issues the I<sup>2</sup>C stop. Using the 500 kbps I<sup>2</sup>C data rate specified in Figure 12.20 and the circuit in Figure 12.19, we measured that a `WriteMAX518(c1,c2)` transfer requires approximately 125  $\mu$ s.

The `main()` in Figure 12.20 is slightly different from the example given in Figure 12.16. The `main` is still the familiar infinite loop. However, instead of reading the two ADC channels after 50 ms delays, the `main()` in Figure 12.20 reads each ADC only when `doADC` flag is 1. The `doADC` flag is set in the Timer2 ISR as discussed earlier. The result is right shifted to reduce the value of the result and places it in an appropriate range for `iamp` and `iper`. With a Timer2 interrupt period of 167  $\mu$ s and two ADC channels, the amplitude pot and the period pot will each be sampled every 512 Timer2 interrupts ( $512 \times 167 \mu\text{s} = 85.5 \text{ ms}$ ) since the `doADC` flag is checked twice within the `while(1){}` loop, once for each pot. This means that the pots are sampled at a much slower rate than the DAC generation of waveform samples. This is sufficient, as the human-machine interface changes relatively slowly. Even at this low sampling frequency (11.7 Hz), the user interface feels very responsive.



**FIGURE 12.21** Three-function waveform generator operation.

## SUMMARY

With the ever-dropping cost of microprocessors, we are embedding digital computers into nearly every conceivable application. However, these digital computers must ultimately communicate with the “real world,” which is an analog environment. Data conversion with ADCs and DACs is done in nearly every microprocessor system that interfaces with other systems, especially other systems that involve people. The data conversion devices are so useful that many embedded microprocessors include built-in ADCs and/or DACs. An ADC gives the microprocessor the capability to understand and operate on analog values generated by people, sensors, or other systems. A DAC allows microprocessors to use its digital processing to create or re-create analog values that people, sensors, or other systems can understand. Even if a microprocessor does not have a built-in or suitable ADC or DAC, an external data converter can be acquired and connected to the microprocessor via a direct parallel connection, an address/data bus, or a serial communications scheme.

**REVIEW PROBLEMS**

---

1. How many bits are required to represent a waveform in 4000 discrete levels?
2. How many different input voltages could you detect with a 24-bit ADC?
3. An audio CD can contain 80 minutes of stereo (two independent) audio tracks. Each track is sampled with 16-bit samples at 44.1 kHz. How much audio information, in bytes, is stored on an audio CD?
4. What is the data throughput (in MB/sec) of the audio CD in Problem 3 (in this context,  $1 \text{ MB/sec} = 10^6 \text{ B/sec}$ )?
5. Still and movie images are often represented by an 8-bit value for each color component, red, green, and blue. How many different colors can be encoded?
6. An HDTV screen contains  $1920 \times 1080$  pixels, with each RGB component represented by 8 bits of precision. Motion playback is 30 frames per second. How much storage is required to store the average two-hour movie? What is the data throughput during HDTV playback?
7. Assume a 4-bit successive approximation A/D with  $V_{\text{REF}+} = 4 \text{ V}$  and  $V_{\text{REF}-} = 0 \text{ V}$ . Trace the steps for producing a 4-bit output code if the input voltage is 1.8 V.
8. For an 8-bit flash ADC, how many comparators are needed? How many resistors are needed?
9. A 3-bit flash A/D has seven comparators, and each comparator output can be either 0 or 1. Assume a  $V_{\text{ref}+} = 4 \text{ V}$ ,  $V_{\text{ref}-} = V_{\text{SS}}$ . What is the 7-bit output of these comparators if the input voltage is 2.7 V? Give the 7-bit value in binary, with the LSB the comparator output with the smallest reference input and the MSB the comparator with the highest reference input.
10. How many clock cycles would you expect a 12-bit successive approximation A/D to require for a conversion?
11. The  $V_{\text{DD}}$  power supply is usually not used as a reference voltage for precision A/D measurements, as it varies with current load and temperature. The National Semiconductor LM4040AIZ is a component that provides a stable 4.096 V voltage reference with an accuracy of  $\pm 0.1 \%$ . What voltage values does this  $\pm 0.1 \%$  correspond to? How does this voltage translate into a percentage of 1 LSB of the 10-bit value produced by the PIC18 using the LM4040AIZ as a  $V_{\text{REF}+}$  value (assume  $V_{\text{REF}-} = 0 \text{ V}$ )?
12. For the PIC18, assume an FOSC of 40 MHz. Using Figure 12.8, what FOSC configurations *cannot* be used because they violate the minimum A/D clock period of 1.6  $\mu\text{s}$ ?

13. An 8-bit ADC has a lower reference voltage of 0 V and an upper reference voltage of 5 V. What output code corresponds to 0.449 V? To 3.91 V?
14. Repeat the previous problem for a 10-bit A/D and an upper reference voltage of 4.096 V.
15. An 8-bit DAC has a lower reference voltage of 0 V and an upper reference voltage of 5 V. What is the output voltage for codes of 0x7F? 0x4B? 0xCB?
16. Repeat the previous problem for a 10-bit DAC and an upper reference voltage of 4.096 V.
17. What is the principle advantage of a flash ADC architecture over a successive approximation ADC architecture? What is the principle disadvantage?
18. The Maxim Integrated Products MAX517 is similar to the MAX518 introduced in this chapter, except that the MAX517 generates only one analog output voltage. The MAX517 package pinout is identical to the MAX518 except that the MAX517 has a DAC voltage reference input instead of the MAX518 OUT1 pin. In fact, the MAX517 DAC allows the DAC reference voltage to change. This kind of DAC is called a “multiplying” DAC because it appears to multiply the DAC reference voltage by the fraction represented by the DAC digital input code. Use an 8-bit MAX517 and a R-2R resistor ladder DAC to construct a system to recreate the results in Figure 12.17, but do not truncate the sine waveform samples.
19. The National Semiconductor LM34 is a precision Fahrenheit temperature sensor. The LM34 produces an output voltage that is linear and equal to 10 mV/°F. The LM34 is accurate up to 300°F. If your PIC18 is using internal ADC references and  $V_{dd} = 5$  V, determine the precision in °F of your measurements.
20. Write a function that reads the LM34 on an arbitrary PIC18 ADC channel and returns the temperature to the calling routine in a double data type.
21. How many bits of resolution are required for an ADC to measure the LM34 temperature between 0°F and 120°F with a precision of 0.25°F?
22. The LM34 is very useful for measuring ambient room temperatures. However, its 10 mV/°F scale means that the most common room temperatures correspond to low voltages assuming an ADC  $V_{ref} = 5$  V. Design a single circuit using the PIC18’s internal ADC and a R-2R resistor ladder DAC that can measure ambient room temperatures between (i) 0°F and 120°F with a precision of 0.25°F, (ii) 0°F and 90°F with a precision of 0.2°F, (iii) 0°F and 60°F with a precision of 0.15°F, and (iv) 0°F and 30°F with a precision of 0.10°F.
23. Write C code that will configure the PIC18 ADC module for left justification, AN2, AN1 as analog inputs, AN3 as  $V_{REF+}$ , and  $V_{SS}$  as  $V_{REF-}$ . Use the internal FOSC clock, and configure the A/D clock such that it meets the

minimum clock period constraint of  $1.6\ \mu\text{s}$  assuming  $F_{\text{OSC}} = 12\ \text{MHz}$  (use the fastest internal clock choice that meets this constraint).

24. Write a function called `char analog_sum()` that performs a conversion on two analog inputs (AN2, AN1) and returns the sum of these values as a char value. Do not let the sum exceed 255 (0xFF) (Hint: You will need to use an unsigned int variable to hold the sum, and then clip this to 255). When changing A/D channels, use the `DelayUs()` function to delay  $20\ \mu\text{s}$  to give the A/D input a chance to settle. Since you don't know how often this function will be called, use this delay before starting each conversion.



*This page intentionally left blank*

# 13



## Timers

### In This Chapter

- The Timer0 Subsystem
- The Timer1 and Timer3 Subsystems
- Pulse Width Measurement Using Capture Mode
- Timer1/Timer3 Compare Mode
- Using Capture Mode for Infrared Decoding
- Timer2 and Pulse Width Modulation
- Using Capture Mode for Frequency Measurement

Use of the Timer2 subsystem for periodic interrupt generation was previously discussed in Chapters 10 and 12. However, this only scratches the surface of the capabilities and application of the PIC18 timer subsystems. This chapter discusses the use of PIC18 timers for time measurement, waveform generation, and pulse width modulation. Example applications include biphasic waveform decoding for infrared reception and DC motor control.

### 13.1 LEARNING OBJECTIVES

---

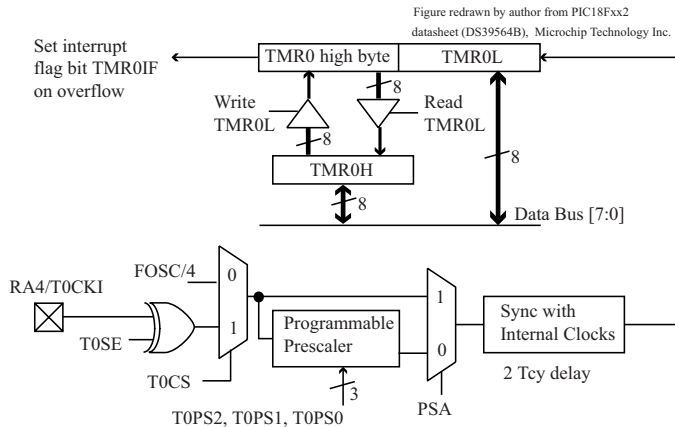
After reading this chapter, you will be able to:

- Discuss the specifics of the Timer0, Timer1, Timer2, and Timer3 subsystems.

- Use the capture subsystem to perform precise time measurements of external events.
- Use the PIC18 timer subsystem to decode an infrared receiver’s output signal that is either space-width encoded or biphase encoded.
- Use the pulse width modulation capability of the PIC18 timer subsystem to generate square waves with varying duty cycles and periods, which can be used to control the brightness of an external LED or the speed of a DC motor via pulse width modulation.
- Implement a real-time clock using a 32.768 kHz clock source and the PIC18 timer subsystem.

### 13.2 THE TIMER0 SUBSYSTEM

The Timer0 subsystem seen in Figure 13.1 can function as either an 8-bit or 16-bit counter/timer as selected by the T08BIT (T0CON[6]) configuration bit. The timer clock source is selected by the T0CS (T0CON[5]) configuration bit; clearing this bit selects the internal instruction cycle clock with frequency of FOSC/4.



**FIGURE 13.1** Timer0 subsystem in 16-bit mode.<sup>1</sup>

Setting T0CS selects the external T0CKI pin as the Timer0 clock source with the T0SE (T0CON[4]) configuration bit selecting rising or falling edge triggering. A prescaler can be optionally selected using the PSA (T0CON[3]) configuration bit. The prescaler has eight values ranging from 256:1 down to 2:1 that is configured via

<sup>1</sup> Figure 13.1 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.’s prior written consent.

the T0PS2:T0PS0 bits (T0CON[2:0]). The TMR0ON (T0CON[7]) bit is used to turn Timer0 on or off. Table 13.1 summarizes the configuration bits of the Timer0 control register (T0CON).

**TABLE 13.1** Bit Definitions for Timer0 Control Register T0CON

Name	SFR(bit)	Comment
TMR0ON	T0CON[7]	If "1", Timer0 on; if "0", Timer0 off.
T08BIT	T0CON[6]	If "1", 8-bit mode; if "0", 16-bit mode.
TOCS	T0CON[5]	If "1", Timer0 clock source is TOCKI pin. If "0", Timer0 clock source is instruction cycle clock.
TOSE	T0CON[4]	If "1", falling edge triggered on TOCKI. If "0", rising edge triggered on TOCKI.
PSA	T0CON[3]	If "1", bypass the prescaler; if "0", use the prescaler.
T0PS2:0	T0CON[2:0]	Set prescaler value as: 111 = 1:256; 110 = 1:128; 101 = 1:64; 100 = 1:32 011 = 1:16; 010 = 1:8; 001 = 1:4; 000 = 1:2

The interrupt flag TMR0IF is set when the counter rolls over, which is 0xFF to 0x00 in 8-bit mode or 0xFFFF to 0x0000 in 16-bit mode. When in 16-bit mode, the TMR0H register is a buffer register for the upper byte of the timer. A read from TMR0L triggers a copy operation from the upper byte of Timer0 to the TMR0H register, thus capturing the complete 16-bit timer value. If a buffer register was not used, a situation could arise in which the timer may be incremented between a read of the lower byte and a read of the upper byte, possibly causing a change in the upper byte value and thus producing an invalid 16-bit result. Similarly, a write to TMR0L triggers a copy operation from the TMR0H to the upper byte of the Timer0 register, thus updating all 16 bits of the Timer0 register simultaneously. This means that any read operation should read TMR0L first, then TMR0H, while a write operation should write TMR0H first, then write TMR0L. Reading TMR0H first then TMR0L returns the TMR0H value at the time of the previous TMR0L read, thus returning an incorrect 16-bit value.

## C Code Operations on 16-Bit Registers

Listing 13.1 shows options for reading/writing Timer0 in 16-bit mode using C code. The `unsigned int tmr0_tics` variable is used to write to Timer0, or as storage for the

Timer0 value. The `TMR0` label used on line 2 is defined as an `unsigned int` at location `0xFD6` (`TMR0L`) by the `pic18xx2.h` header included during the compilation process. The `TMR0L`, `TMR0H` registers are arranged in little-endian order in memory (locations `0xFD6`, `0xFD7`) and the PICC-18 compiler uses little-endian order for extended precision data types. Thus, the assignment `tmr0_tics = TMR0` of line 2 causes the compiler to assign `TMR0L` to the LSB of `tmr0_tics`, and `TMR0H` to the MSB of `tmr0_tics`, with the read of `TMR0L` occurring first. This is what is needed, because in 16-bit mode a read of `TMR0L` triggers a copy of the upper byte of Timer1 into the `TMR0H` register as discussed earlier. Line 3 shows how a `char *ptr` variable can be used to copy the `TMR0L`, `TMR0H` values to the LSB, MSB of `tmr0_tics` as `ptr` is initialized to point to `tmr0_tics` in line 1. Line 4 shows a read of Timer0 in the incorrect order of `TMR0H` first, followed by `TMR0L`. The `TMR0H` read returns an old value of the upper byte of Timer1 that was copied at the last access of `TMR0L`.

---

**LISTING 13.1** Reading/writing `TMR0` using C code.

---

```

unsigned int tmr0_tics;
char *ptr;

(1) ptr = (char *) &tmr0_tics;           // ptr points to LSB of tmr0_tics
(2) tmr0_tics = TMR0;                    // this works for read
(3) *ptr = TMR0L; *(ptr+1) = TMR0H;     // also works for read
(4) *(ptr+1) = TMR0H, *ptr = TMR0L;     // wrong order for read
(5) TMR0 = tmr0_tics;                    // wrong order for write
(6) TMR0H = *(ptr+1); TMR0L = *ptr;     // correct order for write
(7) TMR0H = (tmr0_tics)>>8; TMR0L = (tmr0_tics & 0xFF); //ok as well

```

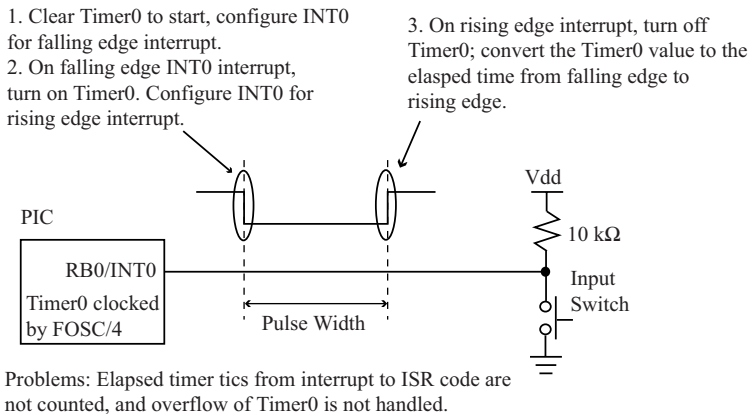
Be careful—when writing Timer0 if the assignment `TMR0 = tmr0_tics` is made as in line 5, the `TMR0L` is written first followed by `TMR0H`. This is the wrong order; the write to `TMR0L` triggers the copy of the `TMR0H` value to the 16-bit timer register, so the value written to `TMR0H` does not actually get copied to the register. For a write, the individual `TMR0L`, `TMR0H` registers must be used as shown in line 6, with `TMR0H` written first followed by a write to `TMR0L`. Line 7 also writes the Timer0 bytes in the correct order, as the statement `TMR0H = tmr0_tics>>8` copies the MSB of `tmr0_tics` to `TMR0H`. The PICC-18 compiler does not implement the right shift operations; instead, the compiler recognizes that this is simply a transfer of the `tmr0_tics` MSB to location `TMR0H` and generates the appropriate code.

In general, any PIC18 register pair that consists of high/low bytes is arranged in little-endian order in memory and the methods of Listing 13.1 can be used to access them. The 16-bit timers, Timer1 and Timer3, have constraints similar to Timer0 access and are discussed later in this chapter. Two other 16-bit registers covered in

this chapter are CCP1 (CCP1H:CCP1L) and CCP2 (CCP2H:CCP2L), which do not have any byte ordering constraints on their access.

## Pulse Width Measurement

A fundamental timer application is time measurement between two external events. In the digital world, an external event is either a rising or falling edge on an input pin. The time between two edges of the same type (falling-to-falling edge or rising-to-rising edge) is the period of a square wave, while the time between a rising-to-falling edge and falling-to-rising edge is high pulse width or low pulse width, respectively. The basics of event measurement are explored using the setup of Figure 13.2, in which the goal is to measure the low pulse width produced by activating a momentary switch.



**FIGURE 13.2** Pulse width measurement.

In this section, the RB0/INT0 input and the Timer0 subsystem is used for pulse width measurement using the steps shown in Table 13.2.

There are several weaknesses to this proposed scheme as will be discussed shortly. Equation 13.1 shows how to convert the Timer0 value (TMR0) to elapsed time, where TOSC is the internal clock period (1/FOSC) and TMR0PRE is the prescaler value.

$$\text{Pulse Width} = \text{TMR0} * \text{TOSC} * 4 * \text{TMR0PRE} \quad (13.1)$$

What are the pulse widths, minimum and maximum, that can be measured as predicted by Equation 13.1? As an example, assume FOSC = 40 MHz (TOSC = 25 ns), 16-bit mode, and TMR0PRE = 1. The minimum pulse width is  $1 * 25 \text{ ns} * 4 *$

**TABLE 13.2** Pulse Width Measurement Steps Using Timer0

Steps
1. Configure INT0 to generate an interrupt on a falling edge. Configure Timer0 to be clocked by the internal clock and clear Timer0.
2. If the interrupt service routine is triggered by an INT0 falling edge, start Timer0 and reconfigure INT0 to be rising edge triggered.
3. If the interrupt service routine is triggered by a rising edge, turn off Timer1 and copy the Timer1 value, which represents the elapsed time from the falling edge to the rising edge.

1 = 100 ns, while the maximum pulse width is  $65535 * 25 \text{ ns} * 4 * 1 = 6,553,500 \text{ ns}$  ( $\sim 6.6 \text{ ms}$ ). The maximum counter value used is 65535, as this scheme does not handle the case of Timer0 overflow. Can the minimum pulse width of 100 ns (one instruction cycle) actually be measured using this scheme? The answer is an emphatic *no*, as the interrupt latency and ISR code execution that changes the interrupt mode of INT0 from falling to rising edge triggered requires several instruction cycles, by which time the rising edge of the pulse has already occurred. Assuming 16-bit mode and  $\text{TMR0PRE} = 256$ , the minimum pulse width is  $1 * 25 \text{ ns} * 4 * 256 = 25,600 \text{ ns}$  (25.6  $\mu\text{s}$ ), and the maximum pulse width is  $65535 * 25 \text{ ns} * 4 * 256 \sim 1.68 \text{ s}$ . Observe that increasing the prescaler value extends the maximum pulse width time at the cost of reduced precision in pulse width measurement. To extend the maximum pulse width time without sacrificing precision, timer overflow must be handled, which is discussed in Section 13.4. One other weakness of this scheme is accuracy. The time from the falling edge to the timer being turned on by the ISR is not counted as part of the pulse width, while any timer tics that elapse from the rising edge to the timer being read by the ISR is erroneously added to the pulse width.

Do these problems preclude this method from being used for pulse width measurement? The answer depends on the accuracy required by the application. This method is suitable for the application of Figure 13.2, which is measuring the pulse width of a human-activated pushbutton switch. However, if precise time measurement of a high-frequency square wave (in the tens to hundreds kHz range) is necessary, a different method must be used as discussed in Section 13.4. For any time measurement scheme, it is important that the limitations be known so that their impact can be analyzed in the context of a proposed application.

The C code of Figure 13.3 implements the pulse width measurement scheme of Figure 13.2. The `#define TMR0TIC 1.0/(FOSCQ/4.0)*PRESCALE` statement assigns

```

#define FOSCQ 29491200
#define PRESCALE 128.0
#define TMR0TIC 1.0/(FOSCQ/4.0)*PRESCALE ← one TMR0 tic in seconds as float

unsigned int tmr0_tics, msec;
double pulse_width_float;
unsigned long pulse_width;
volatile unsigned char capture_flag;

void interrupt timer_isr(void) {
    if (INTOIF) {
        if (!INTEDG0) {
            //seen falling edge, start timer1, change active edge } Falling edge,
            TMR0ON = 1; INTEDG0 = 1; // rising edge interrupt } start timer
        } else {
            TMR0ON = 0; // turn off timer
            //read timer0 as 16-bit value
            tmr0_tics = TMR0;
            INTOIE = 0; //disable RB0 Interrupt
            capture_flag = 1;
        }
        INTOIF = 0;
    }
}

main(void) {
    serial_init(95,1); // 19200 in HSPLL mode, crystal = 7.3728 MHz
    // init timer0
    T0SBIT = 0; // 16-bit mode
    T0CS = 0; // internal clock
    PSA = 0; // use prescaler
    TOPS2 = 1; TOPS1 = 1; TOPS0 = 0; // prescaler = 128
    TRISB0 = 1; // RB0 is input
    IPEN = 0; PEIE = 1; GIE = 1; // enable interrupts
    pcrfl(); printf("(Timer0 version) Ready for button mashing!"); pcrfl();
    while(1) {
        capture_flag = 0;
        TMR0H = 0; // clear timer0, write low byte last
        TMR0L = 0;
        INTEDG0 = 0; // falling edge
        INTOIF = 0; INTOIE = 1; //RB0 Interrupt
        while(!capture_flag); // wait for capture
        // compute time in microseconds
        pulse_width_float = TMR0TIC * tmr0_tics * 1.0e6; } Convert Timer0 tics
        msec = (unsigned int)(pulse_width_float/1000.0); //milliseconds } to microseconds
        pulse_width = (long)pulse_width_float;
        //printf ("Switch pressed, timer ticks: %u, pwidth: %lu (us)",
        // tmr0_tics,pulse_width); // use with full compiler
        printf ("Switch pressed, timer ticks: %u, pwidth: %u (ms)",
            tmr0_tics,msec); // use with demo compiler
        pcrfl(); // Print pulse width result as Timer0 ticks and milliseconds.
    }
}

```

Note: Demo compiler does not support long, float printf formats.



**FIGURE 13.3** Code for pulse width measurement using Timer0.

TMR0TIC the value of one Timer0 tic in seconds; this is used in main() to convert the Timer0 value to microseconds. A prescale value of 128 is used in this example.

The timer\_isr() ISR in Figure 13.3 implements steps 2 and 3 of Table 13.2. Within the ISR, the int tmr0\_tics variable is used to store the Timer0 value at step 3. The assignment tmr0\_tics = TMR0 after the rising edge occurs copies the 16-bit timer value TMR0H:TMR0L to the tmr0\_tics variable in the correct order of



TMR0L first, then TMR0H as discussed earlier. The `capture_flag` variable is used as a semaphore to `main()` to indicate that a pulse width value has been captured. In this particular case, the `INT0IE` bit could have been used as the semaphore (capture is complete when `INT0IE` is cleared by the ISR), but the `capture_flag` variable was used for clarity purposes.

In `main()`, Timer0 is configured for 16-bit mode, clocked by the internal clock, and a prescale value of 128. In the `while(1){}` loop, the `INT0` interrupt is configured for falling edge triggered and then enabled. The `while(!capture_flag){}` statement waits for the ISR to capture a pulse width that is stored as Timer0 tics in the `tmr0_tics` variable. The statement `pulse_width_float = TMR0TIC * tmr0_tics * 1.0E6` converts the Timer0 value stored in `tmr0_tics` to microseconds as a floating-point number. The assignment of this calculation to a variable of type `double` is necessary if the compiler is to use floating-point operations for the calculation. The assignment `pulse_width = (long) pulse_width_float` converts to this a long integer (32-bit), as we are not interested in any fractions less than a microsecond. At this point, the Timer0 value (`tmr0_tics`) and the pulse width in milliseconds are printed to the console. Figure 13.4 shows several tests of the pulse width measurement code in Figure 13.3.

```
(Timer0 version) Ready for button mashing!
Switch pressed, timer ticks: 6836, pwidth: 118 (ms)
Switch pressed, timer ticks: 5265, pwidth: 91 (ms)
Switch pressed, timer ticks: 7233, pwidth: 125 (ms)
Switch pressed, timer ticks: 2878, pwidth: 49 (ms)
Switch pressed, timer ticks: 4718, pwidth: 81 (ms)
```

**FIGURE 13.4** Console output of pulse width measurement using Timer0.

**Sample Question:** *Timer0 is useful for generating periodic interrupts with a large period because of its 16-bit operation and the flexible prescaler (maximum value of 1:256). Assuming a 10 MHz FOSC and use of the internal clock, what configuration produces an interrupt with approximately a one-second period?*

**Answer:** With no prescaler, each Timer0 tic is  $4/(10 \text{ MHz}) = 0.4 \mu\text{s}$ . In 16-bit mode, the rollover period is  $0.4 \mu\text{s} * 65536 = 26,214.4 \mu\text{s}$ . The needed prescale value for a one-second interrupt period is  $1 \text{ s} / 26,214.4 \mu\text{s} \sim 38$ . The closest prescaler match is 32, which produces an interrupt with a period of  $26,214.4 \mu\text{s} * 32 \sim 0.84 \text{ s}$ .

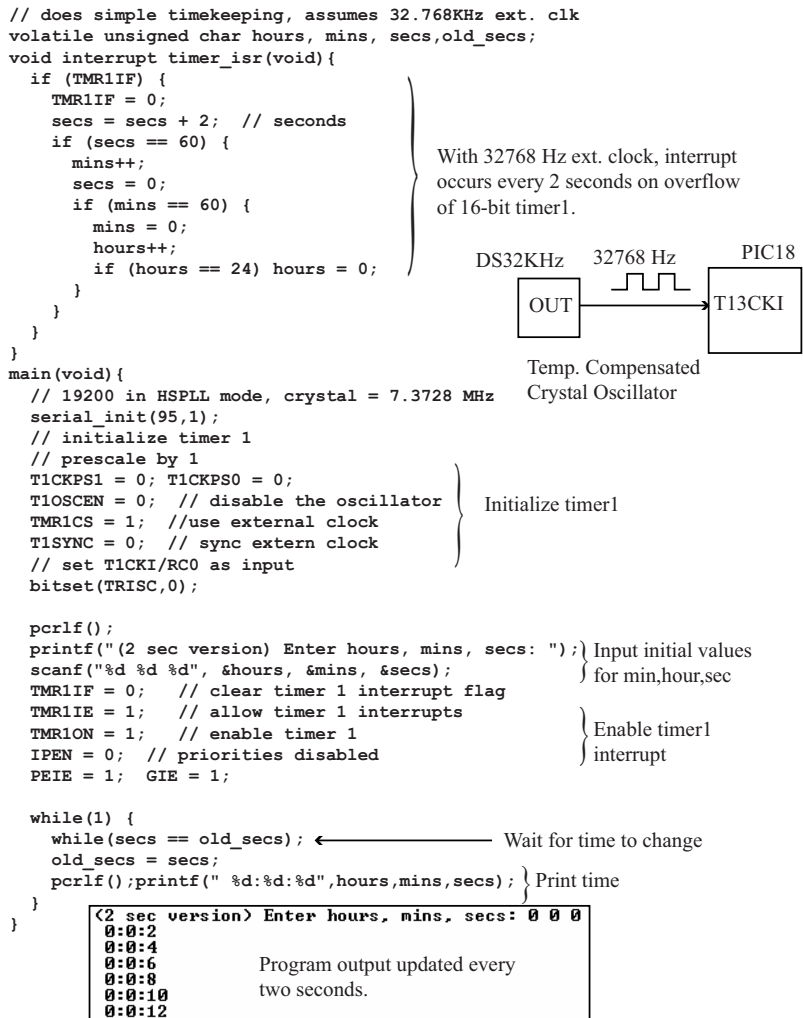


from 0xFFFF to 0x0000. Table 13.3 summarizes the Timer1 configuration bits contained in the T1CON register.

**TABLE 13.3** Bit Definitions for Timer1 Control Register T1CON

Name	SFR(bit)	Comment
RD16	T1CON[7]	If "1", read/write operation is one 16-bit operation. If "0"; read/write operation is two 8-bit operations. (For the PICC-18 compiler, use symbol T1RD16 or RD16 for Timer1, T3RD16 for Timer3.)
N/A	T1CON[6]	Unimplemented.
T1CKPS1:0	T1CON[5:4]	Set prescaler value as: 11 = 1:8; 10 = 1:4; 01 = 1:2; 00 = 1:1
T1OSCEN	T1CON[3]	If "1", oscillator is enabled; if "0" oscillator is shut off.
T1SYNC	T1CON[2]	If "1", asynchronous mode (ext. clock unsynchronized). If "0", synchronous mode (ext. clock synchronized). This bit is ignored if TMR1CS = 0.
TMR1CS	T1CON[1]	If "1", use the external clock input; if "0", use the internal instruction cycle clock.
TMR1ON	T1CON[0]	If "1", Timer0 on; if "0", Timer0 off.

Why is a second oscillator circuit included for Timer1/Timer3? One common use of this capability is to provide a clock/calendar function by use of a 32.768 kHz crystal or external clock source. Observe that 32768 Hz is a power of two ( $32768 = 2^{15}$ ); this means that when the 16-bit Timer1 rolls over from 0xFFFF to 0x0000 the elapsed time is  $2^{16}$  clock tics, or  $2^{16}/(2^{15} \text{ Hz}) = 2$  seconds. Figure 13.6 shows C code that uses Timer1 and an external 32.768 kHz clock to implement simple time keeping of seconds, minutes, and hours.

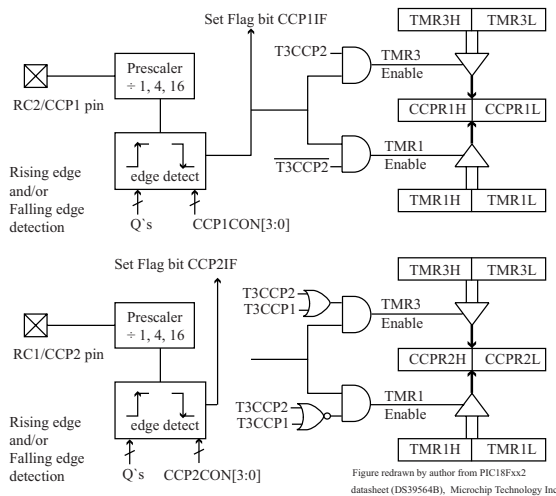


**FIGURE 13.6** Simple timekeeping using Timer1.

The ISR is triggered every two seconds by the TMR1IF flag, which then updates the secs, mins, and hours variables. The external clock is provided by the Dallas Semiconductor DS32KHz temperature compensated crystal oscillator [16]; using an external clock source such as this is only necessary if long-term, accurate time-keeping is needed with minimum variation due to temperature. Alternatively, the clock can be generated by connecting a 32768 Hz crystal/capacitor network across the T13CKI, T1OSCI pins. A shortcoming of the code of Figure 13.6 is that the secs variable is only updated every two seconds. Section 13.5 illustrates an alternate method using the capture/compare module that updates the secs variable every second.

### 13.4 PULSE WIDTH MEASUREMENT USING CAPTURE MODE

The pulse width measurement scheme of Section 13.2 has an accuracy shortcoming in that time between a falling edge occurrence and the Timer0 being turned on is not counted as part of the pulse width. Another accuracy problem is that any timer tics that elapse between a rising edge occurrence and Timer0 being read is incorrectly counted as part of the pulse width. The capture subsystem solves these problems by causing an automatic transfer of the timer register contents to a capture register on occurrence of an event. Figure 13.7 shows the capture subsystem, which consists of two 16-bit registers named CCPR1 and CCPR2 that can capture either the Timer1 or Timer3 values. When the CCPx pin is configured as an input, the capture system can be configured to trigger on either a rising edge or falling input edge. If the CCPx pin is configured as an output, a write to the port triggers a capture.



**FIGURE 13.7** Timer1/Timer3 capture.<sup>3</sup>

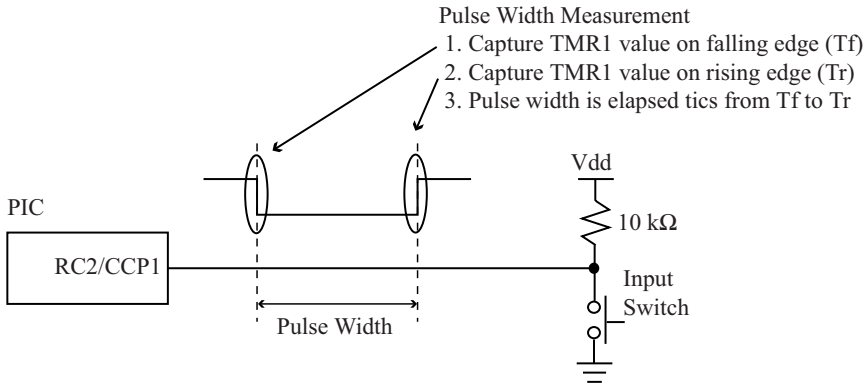
The CCP1CON/CCP2CON registers configure the capture modes for the CCP1/CCP2 pins, respectively. Both registers function in the same manner; Table 13.4 gives the bit definitions for the CCP1CON register. The compare and PWM functions are discussed later in this chapter.

<sup>3</sup> Figure 13.7 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.'s prior written consent.

**TABLE 13.4** Bit Definitions for CCP1CON Register

<b>Name</b>	<b>SFR(bit)</b>	<b>Comment</b>
N/A	CCP1CON[7:6]	Unimplemented
DC1B1:0	CCP1CON[5:4]	Lower 2 bits of 10-bit PWM duty cycle
CCP1M3:0	CCP1CON[3:0]	Mode select bits: 0000 = CCP1 module disabled 0001 = Reserved 0010 = Compare mode, toggle output on match, set CCP1IF bit 0011 = Reserved 0100 = Capture mode, every falling edge 0101 = Capture mode, every rising edge 0110 = Capture mode, every 4th rising edge 0111 = Capture mode, every 16th rising edge 1000 = Compare mode, initialize CCP1 low, on compare force CCP1 pin high (set CCP1IF bit) 1001 = Compare mode, initialize CCP1 high, on compare force CCP1 pin low (set CCP1IF bit) 1010 = Compare mode, generate software interrupt on compare match (set CCP1IF bit, CCP1 pin unaffected) 1011 = Compare mode, trigger special event which clears TMR1 register pair (set CCP1IF bit). For CCP2CON, the special event also triggers an ADC conversion if the ADC is enabled. 11xx = PWM mode

Figure 13.8 shows the new pulse width measurement scheme using Timer1 and capture mode. Initially, the capture mode is programmed for a falling edge on the CCP1 pin; once this occurs, the captured Timer1 value in the CCP1RH, CCP1RL register pair is saved and the capture mode is changed to a rising edge. After a new timer value is captured by a rising edge on pin CCP1, the difference between the rising edge timer value and the falling edge timer value represents the pulse width. Because the timer value is captured immediately by the event occurrence, there are no “missing” timer tics as occurred with the method used in Section 13.2.

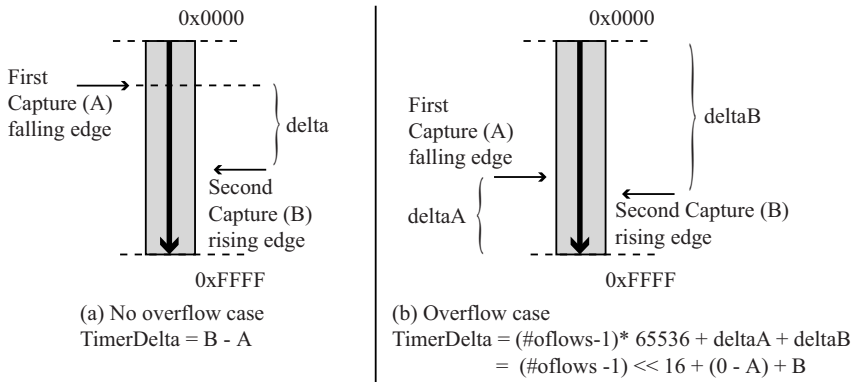


**FIGURE 13.8** Pulse width measurement using Timer1 and capture mode.

One significant difference between this scheme and the previous scheme is that the timer is always in operation; it is not started and then stopped as was done in Section 13.2. This means that the elapsed timer tics between the two captured timer values must be computed, and that timer overflow can occur between the two capture events because of the free-running nature of the timer. Figure 13.9a shows the case where timer overflow does not occur; the capture A value represents the falling edge timer value and the capture B value the rising edge timer value. The pulse width in timer tics is  $TimerDelta = B - A$  because timer overflow has not occurred. Figure 13.9b shows the case when the timer overflows and the pulse width in timer tics is computed as  $TimerDelta = (\#oflows - 1) * 2^{16} + (0 - A) + B$ . The *#oflows* variable counts the number of times the timer overflows, while the  $0 - A$  delta value is the number of timer tics to the first overflow. A long variable type (4 bytes) must be used to store this result since the result can be greater than 16 bits. A value of 1 is subtracted from the *#oflows* variable because the last overflow is accounted for by the *deltaB* value.

Figure 13.10 shows the interrupt service routine for computing the pulse width using Timer1 and the CCPR1H, CCPR1L capture registers.

The unsigned char *tmr1\_ov* variable is used to track Timer1 overflows; it is initialized to zero when a falling edge is captured so that the no overflow case is indicated by a zero value. The *tmr1\_ov* variable is incremented on each TMR1IF interrupt, which occurs when Timer1 rolls over from 0xFFFF to 0x0000. A CCP1IF interrupt indicates that a capture event has occurred. The 16-bit capture value in CCPR1H:CCPR1L is copied to the unsigned int *this\_capture* variable by the *this\_capture = CCPR1* statement, as the *CCPR1* label is defined as an unsigned int starting at location CCPR1L in the *pic18xx2.h* header file. A falling edge capture signals the start of the measurement, so the *this\_capture* value is saved to the *last\_capture* variable, the *tmr1\_ov* variable is reset to zero, and the capture mode



**FIGURE 13.9** Computing the elapsed timer tics between two events.

is reconfigured to a rising edge capture. On a rising edge capture, if the `tmr1_ov` is zero, no overflow has occurred and the elapsed timer tics is computed as:

$$\text{delta} = \text{this\_capture} - \text{last\_capture}$$

If `tmr1_ov` is nonzero, timer overflow has occurred and the pulse width is computed as per Figure 13.9b:

$$\text{delta} = ((\text{tmr1\_ov}-1) \ll 16) + (0 - \text{last\_capture}) + \text{this\_capture}$$

The `capture_flag` semaphore is set on the rising edge capture to signal `main()` that the pulse width capture is complete.

The `main()` code for pulse width measurement using Timer1 and capture mode is shown in Figure 13.11. Because the CCP1 input is shared with the RC2 port, the RC2 port must be configured as an input for CCP1 capture mode. Timer1 is configured to use the internal clock and a prescale value of 2. The prescale value was arbitrarily chosen for this example, as no particular precision or maximum pulse width times are stated. The `while(1){}` main loop first configures the CCP1 capture module for falling edge triggering and waits via the `while(!capture_flag){}` loop for the ISR to signal that the pulse width capture is complete. After the `capture_flag` is set, the `delta` value containing the elapsed timer tics that represent the pulse width is converted to milliseconds and is printed to the console.



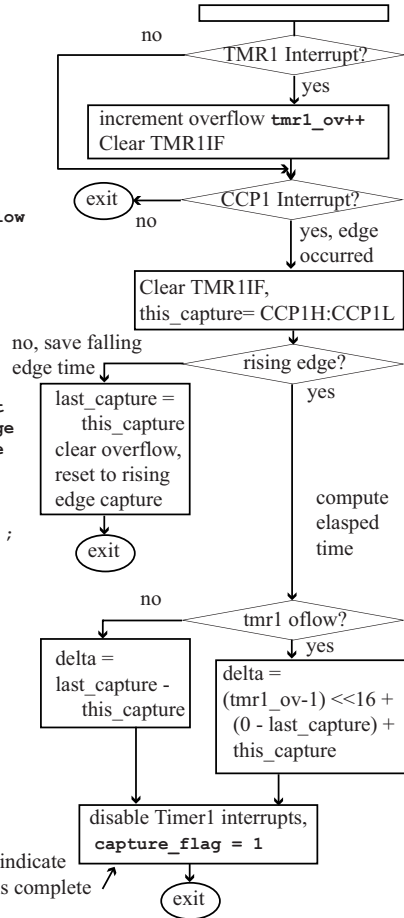
```

volatile unsigned int last_capture;
volatile unsigned int this_capture;
// this must be long
volatile unsigned long delta;
// timer 1 overflow cnt
volatile unsigned char tmr1_ov;
volatile unsigned char capture_flag;

timer_isr(void) {
    if (TMR1IF) {
        tmr1_ov++; // increment timer1 overflow
        TMR1IF = 0;
    }
    if (CCP1IF) {
        // read CCP1 as a 16-bit value
        this_capture = CCP1L;
        if (!bittst(CCP1CON,0) ) {
            //falling edge
            last_capture = this_capture;
            tmr1_ov = 0; // clear overflow count
            CCP1CON = 0x0; // turn off when change
            CCP1CON = 0x5; // capture rising edge
        } else {
            if (!tmr1_ov) {
                // no overflow at all
                delta = this_capture - last_capture ;
            }
            else {
                // compute delta time
                delta = tmr1_ov-1;
                delta = (delta << 16);
                last_capture = 0 - last_capture;
                delta = delta + last_capture;
                delta = delta + this_capture;
            }
        }
        // disable timer1 interrupt
        TMR1ON = 0; TMR1IE = 0; TMR1IF = 0;
        capture_flag = 1;
    }
    //clear capture interrupt flag
    CCP1IF = 0;
}

```

Semaphore to main() to indicate that pulse width capture is complete



**FIGURE 13.10** ISR for pulse width measurement using Timer1 and capture mode.

Figure 13.12 shows the console output for tests of the pulse width measurement code of Figures 13.10 and 13.11.

```

#define FOSCQ 29491200
#define PRESCALE 2.0
#define TMR1TIC 1.0/(FOSCQ/4.0)*PRESCALE
double pulse_width_float;
unsigned long pulse_width;
unsigned int msec;
int *ptr;

main(void) {
    serial_init(95,1); // 19200 in HSPLL mode, crystal = 7.3728 MHz
    // initialize timer 1
    T1CKPS1 = 0; T1CKPS0 = 1; // prescale by 2 } Configure Timer1
    T1OSCEN = 0; // disable the oscillator
    T1SYNC = 0; TMR1CS = 0; //use internal clock FOSC/4
    bitset(TRISC,2); // set CCP1 as input ← Configure CCP1/RC2 as input
    // enable interrupts for capture
    IPEN = 0; PEIE = 1; GIE = 1;
    ptr = (int *) &delta; //for printf of long type
    pcrLf();printf("(Timer1 version) Ready for button mashing!");pcrLf();
    while(1) {
        // configure capture
        CCP1IE = 0; // disable when changing modes
        CCP1CON = 0x0; // turnoff before changing } Capture falling edge, enable
        CCP1CON = 0x4; // capture every falling edge } CCP1 interrupt
        CCP1IF = 0; // clear CCP1IF interrupt flag
        CCP1IE = 1; // enable capture interrupt
        TMR1IF = 0; // clear timer 1 interrupt flag } Turn on the timer
        TMR1IE = 1; // allow timer 1 interrupts } and enable the timer1
        TMR1ON = 1; // enable timer 1 } interrupt
        // wait for falling edge } Wait for ISR to complete
        capture_flag = 0; } capture of pulse width
        while(!capture_flag);
        pulse_width_float = (delta * TMR1TIC)*1.0e6; //microseconds
        pulse_width = (long) pulse_width_float;
        msec = (unsigned int) (pulse_width_float/1000.0); //milliseconds
        // printf ("Switch pressed, timer ticks:%lu , pwidth: %lu (us)",
        // delta, pulse_width); // use with full compiler
        printf ("Switch pressed, timer ticks:0x%x%x, pwidth: %u (ms)",
        *(ptr+1), *ptr,msec); // use with demo compiler
        pcrLf(); ← Print pulse width result as Timer1 ticks (hex) and milliseconds.
    }
}

```

Note: Demo compiler does not support long, float printf formats.



**FIGURE 13.11** main() for pulse width measurement using Timer1 and capture mode (see CD-ROM file ./code/chap13/F\_13\_10\_swdetov.c).

```

(Timer1 version) Ready for button mashing!
Switch pressed, timer ticks:0x496DC, pwidth: 81 (ms)
Switch pressed, timer ticks:0x6DF3C, pwidth: 122 (ms)
Switch pressed, timer ticks:0x67DCD, pwidth: 115 (ms)
Switch pressed, timer ticks:0x4A61C, pwidth: 82 (ms)
Switch pressed, timer ticks:0x36E4F, pwidth: 60 (ms)

```

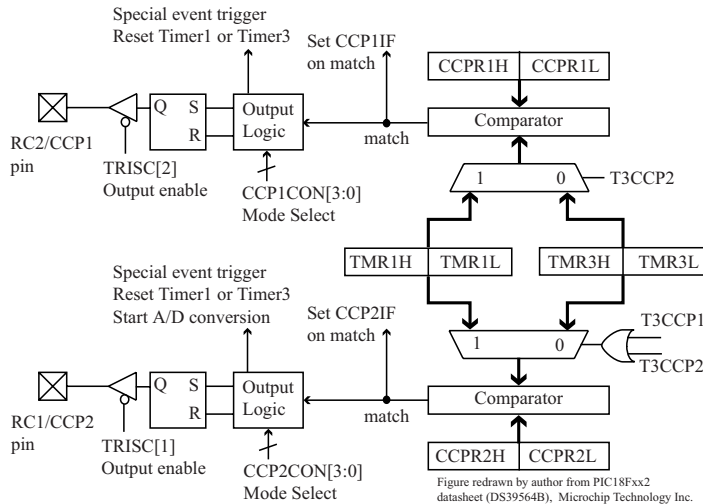
**FIGURE 13.12** Console output for pulse width measurement using Timer1 and capture mode.

**Sample Question:** *If the CCP2 input is used in Figure 13.8 instead of CCP1, what changes are required to the code of Figures 13.10 and 13.11 to use the CCPR2 register instead of the CCPR1 register?*

*Answer:* In the ISR of Figure 13.10, CCP1IF is replaced by CCP2IF, CCP1CON by CCP2CON, CCPR1L by CCPR2L, and CCPR1H is replaced by CCPR2H. In the main() code of Figure 13.11, the same replacements are made in addition to replacing CCP1IE with CCP2IE. The CCP2 input is unusual in that for the PIC18F242, it is either shared with the RC1/T1OSI/CCP2 pin (default setting) or the RB3/CCP2 pin based upon a program memory configuration bit. This provides an option for using the CCP2 input via the RB3/CCCP2 pin if the oscillator circuit for Timer1/Timer3 is being used. Either RC1 or RB3 must be configured as an input when using the CCP2 input capture.

### 13.5 TIMER1/TIMER3 COMPARE MODE

Compare mode operation is shown in Figure 13.13. Compare mode compares the contents of Timer1 or Timer3 against the contents of the CCPR1 or CCPR2 registers as selected by the T3CCP2 (T3CON[6]) and T3CCP1 (T3CON[3]) configuration bits, triggering an action on a successful match of the timer and compare value.



**FIGURE 13.13** Timer1/Timer3 compare mode operation.<sup>4</sup>

<sup>4</sup> Figure 13.13 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.'s prior written consent.

The CCPxCON[3:0] configuration bits (Table 13.4) select one of five possible actions in compare mode:

1. Trigger a special event on match (CCPxCON[3:0] = 1011).
2. Initialize the CCPx pin low and then force the CCPx pin high on match (CCPxCON[3:0] = 1000).
3. Initialize the CCPx pin high and then force the CCPx pin low on match (CCPxCON[3:0] = 1001).
4. Toggle the CCPx pin on match (CCPxCON[3:0] = 0010).
5. Generate a software reset on match (CCPxCON[3:0] = 1010). A software reset means that the CCPxIF flag is set on match, and an interrupt generated if it is enabled, but the status of the external CCPx pin is unchanged.

For any of these actions, the appropriate CCP1IF/CCP2IF interrupt flag is set. The special event trigger clears Timer1 or Timer3, whichever one was used for the match. Additionally, the special event trigger for CCP2 starts an ADC conversion by setting the GO/DONE# bit (ADCON[2]).

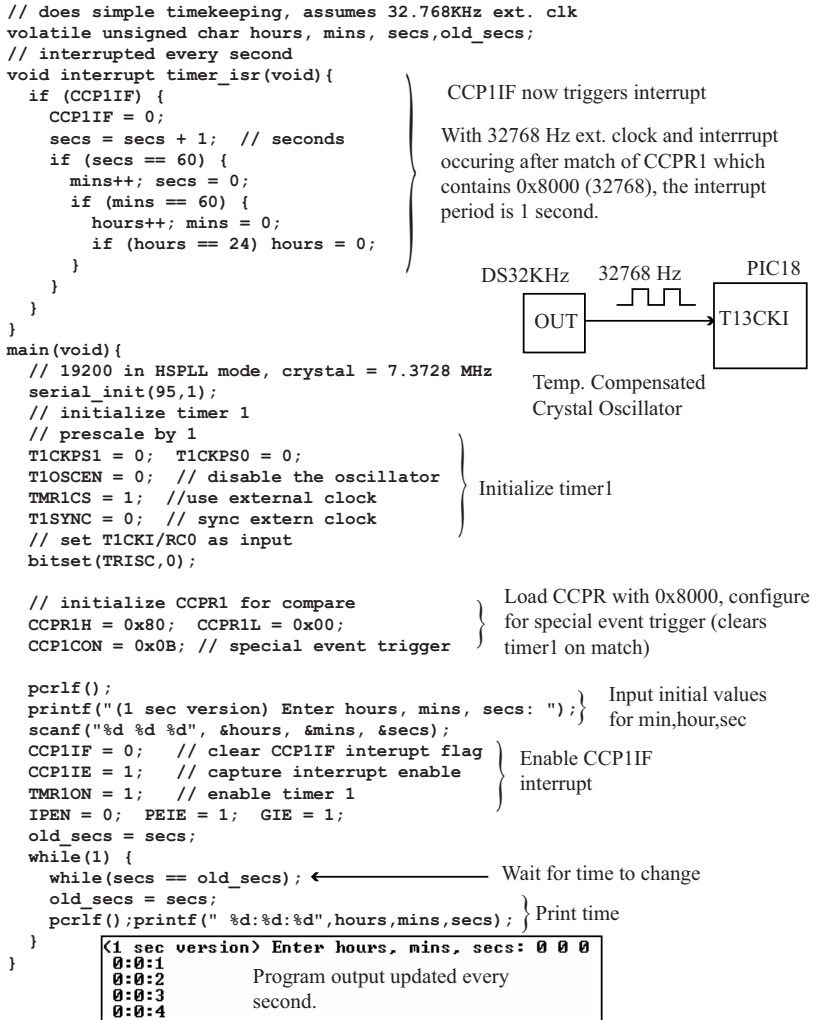
### **Periodic Interrupt Generation Using Compare Mode**

One use of the compare mode register CCPRx is to act as an interrupt interval control register with a timer base of either Timer1 or Timer3. Recall that in the simple timekeeping code of Figure 13.6, an interrupt was generated every two seconds as Timer1 rolled over from 0xFFFF to 0x0000 because it was clocked by an external 32.768 kHz clock. This caused the secs variable to be updated by the ISR every two seconds. To update the secs variable every second, the CCPR1 register is loaded with 0x8000 and the special event trigger used to reset Timer1 on match. Figure 13.14 shows the required modifications to the code of Figure 13.6 for using compare mode. Observe that the CCP1IF flag instead of the TMR1IF flag now triggers the ISR and the secs variable is incremented by one instead of by two.

### **Square Wave Generation Using Compare Mode**

Another use of the compare mode registers CCPRx is to act as a period control register for square wave generation by toggling the state of the CCPx pin on each match of CCPRx and either Timer1 or Timer3. This creates a square wave with a 50% duty cycle; whose period is twice that of the CCPRx register value as seen by Equation 13.2.

$$\text{SquareWave Period} = 2 * \text{CCPRx} * \text{TMR1/3\_CLKPERIOD} \quad (13.2)$$



**FIGURE 13.14** Simple timekeeping using Timer1 and compare mode.

Figure 13.15 shows two approaches for the ISR code that generates a square wave using the toggle capability of compare mode. Both ISRs assume the CCP1 register contains the match value and that the CCP1 pin has been configured to toggle on match of Timer1. In Figure 13.15a, the ISR clears the Timer1 value on interrupt by writing a 0x00 to both TMR1H and TMR1L. While this does generate a square wave, its period is slightly larger than that predicted by Equation 13.2 because of the Timer1 tics that elapse between the CCP1/Timer1 match and the clearing of Timer1 within the ISR.

(a) Clearing Timer1 in ISR during square wave generation

```
// uses Timer1, compare & toggle mode
// to generate sq wave
void interrupt timer_isr(void){
    if (CCP1IF) {
        // clear timer 1 to reset match
        // write TMR1L byte last!
        // triggers 16-bit write
        TMR1H = 0;
        TMR1L = 0;
        CCP1IF = 0;
    }
}
```

If want to clear timer1 while using output toggle mode, then must manually write zero to the register. Write low byte last to trigger writes of both bytes.

PROBLEM!!! Depending on the timer1 clock frequency, several timer1 tics will have elapsed before the timer can be cleared, affecting the square wave frequency.

(b) Incrementing CCP1H/CCP1L in ISR during square wave generation

```
// uses Timer1, compare & toggle mode
unsigned int match;
// uses Timer1, compare & toggle
// mode to generate sq wave
void interrupt timer_isr(void){
    if (CCP1IF) {
        // don't clear timer1,
        // change compare register
        match = match + HPERIOD;
        CCP1H = match >> 8;
        CCP1L = match & 0xFF;
        CCP1IF = 0;
    }
}
```

Instead of clearing timer1, increment the compare register by the half period value.

When writing the new compare value, must be careful to not generate a false match, so write the MSByte first.



**FIGURE 13.15** Two approaches for square wave generation using compare mode.

A better method is shown in Figure 13.15b in which the CCP1 compare register is incremented each time by a fixed amount (HPERIOD) to generate a new match value that is stored in the `unsigned int match` variable. Because Timer1 is never altered, there are no missing timer tics that can cause the generated square wave period to differ from the predicted square wave period. This assumes that HPERIOD is large enough so that the new CCP1 value can be written before Timer1 reaches the match value. Figure 13.16 shows the `main()` code that is paired with the ISR of Figure 13.15b for generating a square wave on the CCP1 pin output using Timer1 and CCP1. Once all of the configuration is done for Timer1, the CCP1 pin, and the CCP1 compare mode, the `while(1){}` loop of `main()` has nothing to do, as the compare mode logic and the ISR do all of the work of square wave generation. Equation 13.3 calculates the expected square wave period for the configuration shown in `main()` ( $F_{OSC} = 29.4912 \text{ MHz}$ , Timer1 prescale of 1, and  $2 * \text{match}$  period, where the match period is 256 Timer1 tics).

$$\text{SquareWave Period} = 2 * 256 * 1 / (29.4912 \text{ MHz} / 4) = 69.4 \mu\text{s} \quad (13.3)$$

This period is obtained by the code of Figure 13.15b and Figure 13.16 on the PIC18F242 reference system. However, a period of 73  $\mu\text{s}$  is generated when the ISR code of Figure 13.15a is used, because of the delay in clearing the Timer1 register in the ISR after the occurrence of a match between Timer1 and CCP1.

```
// half period in timer tics
#define HPERIOD 0x0100 ← Number of Timer1 tics for half period

main(void) {
    serial_init(95,1); // init serial port, 19200 BR
    // initialize timer 1
    T1CKPS1 = 0; T1CKPS0 = 0; // prescale by 1
    // use internal clock
    T1OSCEN = 0; TMR1CS = 0; T1SYNC = 0;
    T1RD16 = 1; // 16 bit r/w to timer1
    bitclr(TRISC,2); // set RC2/CCP1 as output
    // initialize CCP1 for compare
    match = HPERIOD;
    CCP1LH = (HPERIOD >> 8);
    CCP1L = (0xFF & HPERIOD);
    CCP1CON = 0x02; // toggle mode
    // capture interrupt enable
    CCP1IF = 0; CCP1IE = 1;
    TMR1ON = 1; // enable timer 1
    IPEN = 0; PEIE = 1; GIE = 1;
    printf("Configured!"); pcr1f();
    while(1); // interrupt does all work
}

```

} Configure timer1, use internal clock of FOSC/4, prescale = 1  
 } Load CCPR with match value, and configure for output toggle.  
 } Enable CCP1IF interrupt  
 } ISR and capture mode logic does all of the work of square wave generation



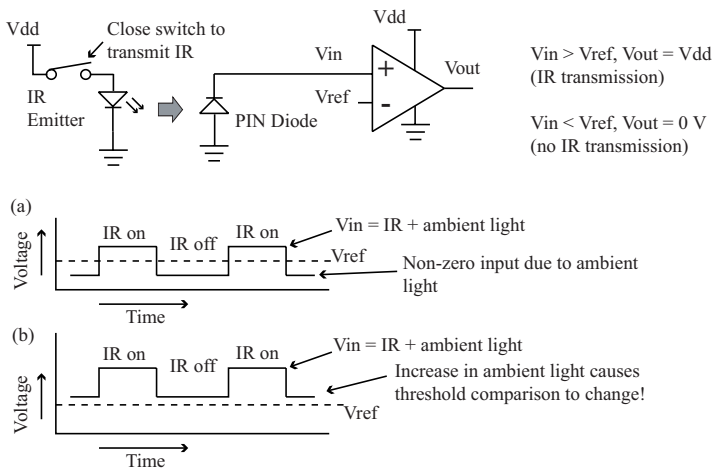
**FIGURE 13.16** main() code for square wave generation using compare mode (see CD-ROM file ./code/chap13/F\_13\_15b\_tmr1sqwave\_good.c).

**Sample Question:** The code of Figure 13.15b is careful to write the MSByte first, followed by a write of the LSByte to the CCP1 register to avoid a “false match.” Give an example where reversing the order of the writes may provide a false match.

**Answer:** Assume the CCP1, Timer1 match occurred for a value of 0x10A0 and that the next CCP2 match value is computed as 0x20B5. During the execution of the ISR, Timer1 continues incrementing. Thus, Timer1 is somewhere between the last match value 0x10A and the next match value of 0x20B5. If the LSByte of the CCP2 register is written first, CCP2 becomes 0x10B5 for the period of time between the write of the LSByte and the write of the MSByte. This could generate a match with the Timer1 value, toggling the CCP1 output prematurely. Writing the MSByte of CCP1 first avoids this false match possibility assuming that there is at least a 256 Timer1 tic difference between the old and new CCP1 values.

## 13.6 USING CAPTURE MODE FOR INFRARED DECODING

Infrared (IR) transmit and receive is a common method for wireless communication. Remote controls for televisions, VCRs, DVD players, and satellite receivers all use IR communication. IR light is just below visible light in terms of frequency within the electromagnetic spectrum. A simple scheme for IR transmit and receive is shown in Figure 13.17, in which an IR LED is turned on or off by a switch. The IR receiver is a PIN diode whose resistance varies based upon the amount of IR received, causing the output voltage to vary in the presence or absence of IR transmission.



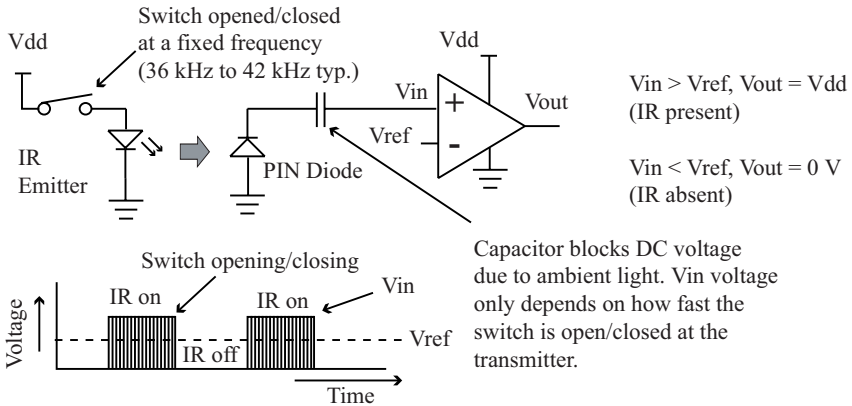
**FIGURE 13.17** IR transmit/receive, no modulation.

Because ambient light contains an IR component, the output of the IR detector is nonzero even when no IR is being transmitted. The input to the comparator is the output of the IR detector, which is compared against a reference voltage whose value should be between the voltage output of the IR receiver in the absence or presence of IR transmission as shown in waveform (a) of Figure 13.17. When  $V_{in} > V_{ref}$ , the output of the comparator is  $V_{dd}$  indicating an active IR transmission. When  $V_{in} < V_{ref}$ , the output of the comparator is  $0\text{ V}$  indicating no IR transmission. The problem with the scheme of Figure 13.17 is that a change in ambient lighting (perhaps a move from indoor lighting to outside sunshine) will change the quiescent output of the IR receiver, causing  $V_{in}$  either to be always above  $V_{ref}$  (waveform (b) of Figure 13.17) or always below  $V_{ref}$ .

Figure 13.18 shows an IR transmit/receive scheme that is not affected by ambient lighting conditions. To transmit IR, the switch is rapidly opened and closed to



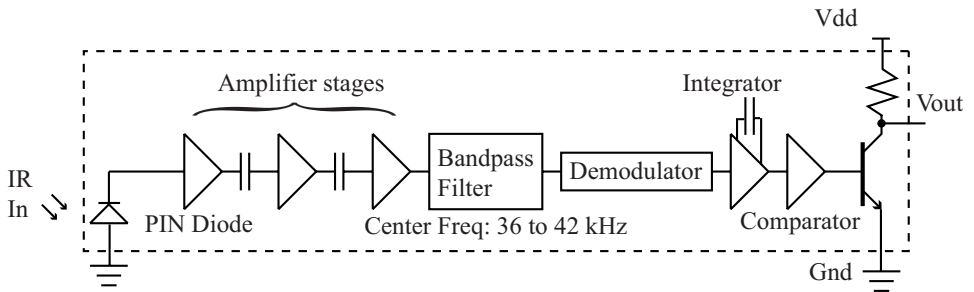
produce a modulated IR signal. A capacitor is used on the input of the comparator to block the DC component (nonchanging component) of the IR detector output due to ambient lighting conditions.



**FIGURE 13.18** IR transmit/receive with modulation.

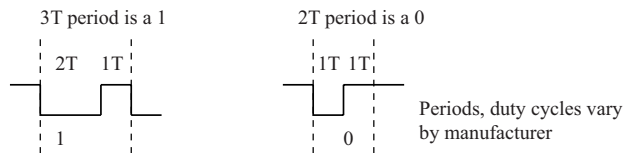
The voltage component that passes through the capacitor to the comparator input is the component that is changing due to the modulated IR input. This means that  $V_{in}$  is no longer affected by ambient light; the voltage seen on the capacitor output is dependent upon the frequency at which the switch is open and closed and the transmission length of one IR bit time. Typical modulation frequencies in commercial transmitters/receivers range from 36 kHz to 42 kHz with transmission bit times in the hundreds of microseconds. Commercial IR receivers such as the NJL30H/V00A (NJR Corporation) or GP1UM2xx (SHARP Microelectronics) integrate the IR detection diode with the electronics necessary to produce a clean digital output in the presence or absence of IR transmission. Figure 13.19 shows a sample block diagram for an integrated IR receiver with three pins: Vdd, ground, and Vout. The output is high in the absence of IR transmission and pulled low when a modulated IR signal is received.

Typical IR data links for remote control of home electronic systems are simplex, low-speed serial communication channels. Even though the NRZ (non-return-to-zero) encoding used for RS-232 serial data could be used for IR transmission, two other schemes known as *space-width* encoding and *biphase* encoding are commonly used for these applications. Figure 13.20a illustrates space-width encoding that encodes ones and zeros as different period lengths with different duty cycles. Typical period lengths are in the hundreds of microseconds with period length and duty cycle varying by manufacturer.

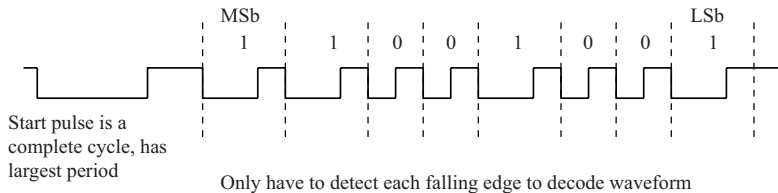


**FIGURE 13.19** Integrated IR receiver.

(a) Space-Width Encoding, 1 and 0 distinguished by period length



(b) Space-width encoding example, value is 0xC9. Can send multiple bytes in one transmission.

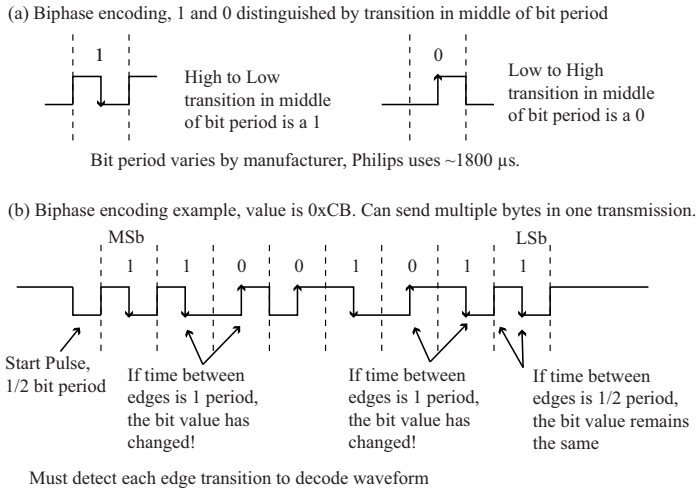


**FIGURE 13.20** Space-width encoding.

Figure 13.20b shows a serial data transmission using space-width encoding in which the first bit is a start pulse, followed by space-width encoded “1”s and “0”s. A “0” has a period of  $2T$  units with a 50% cycle, and a “1” has a period of  $3T$  units and a 33% duty cycle (the duty cycles and periods were arbitrarily chosen). Decoding this serial waveform is done by measuring the time between successive falling edges to distinguish between “1”s and “0”s. It is not necessary to determine the duty cycle, as “1” and “0” have distinct periods. Most space-width encoding schemes use a start pulse with a period significantly longer than a “1” or “0”. The periods of the start, “1”, “0” bits; the number of bits sent in a transmission, and their meanings in terms of commands for the target devices are all manufacturer specific.

Figure 13.21a shows *biphase* encoding, which is another encoding form sometimes used with IR transmissions. In biphase encoding, each bit period is the same

width with “1”s and “0”s distinguished by a high-to-low transition and a low-to-high transition, respectively, in the middle of the bit period.



**FIGURE 13.21** Biphas encoding.

Figure 13.21b shows a serial data transmission using biphas encoding. Observe that the start pulse is only one-half of a bit period. One method of decoding biphas waveforms is to measure the time between both rising and falling edge transitions. If the time between two edges is one bit period, this indicates that the current bit is the complement of the previous bit. If the time between two edges is one-half period and this is the last half of the bit period, then this bit is the same as the previous bit.

Figure 13.22 shows the ISR code for biphas signal decoding as produced by an IR universal remote control. The ISR measures the time between each successive edge on the CCP1 input pin using Timer1 and CCPR1 capture registers, with the pulse width in Timer1 tics stored in the unsigned long `delta` variable. This is very similar to the pulse width measurement code previously seen in Figure 13.10. On each edge arrival, the pulse width is computed, the active edge is toggled from rising-to-falling or falling-to-rising, the `edge_capture` variable is set to indicate an edge arrival, and the function `do_ircap()` is called to determine if a “1” or “0” bit has been received. The `do_ircap()` function is also called when Timer1 overflows.

```

volatile unsigned int last_capture, this_capture;
volatile unsigned long delta;
volatile unsigned char tmr1_ov; // timer 1 overflow cnt

#define MAXBYTES 8
volatile unsigned char cbuff[MAXBYTES];
volatile unsigned char bitcount, bytecount, bit_edge;
volatile unsigned char state, edge_capture, current_bit;
volatile unsigned char this_byte;

#define IDLE_TIME 4
#define BITCHANGE 10000 ← # of Timer1 tics to detect change from 0 to 1 or
                          vice versa
#define IDLE 0
#define START_PULSE 1
#define BIT_CAPTURE 2
#define IO_FINISH 3 } State definitions for do_ircap() function

void interrupt timer_isr(void) {
  if (TMR1IF) {
    tmr1_ov++; // increment timer1 overflow
  }
  if (CCP1IF) {
    // read CCP1 as 16-bit value
    this_capture = CCP1;
    if (!tmr1_ov) {
      // no overflow at all
      delta = this_capture - last_capture ;
    } else {
      delta = tmr1_ov-1;
      delta = (delta << 16);
      last_capture = 0 - last_capture;
      delta = delta + last_capture;
      delta = delta + this_capture;
    }
    last_capture = this_capture;
    tmr1_ov = 0; // clear timer 1 overflow count
    if (bittst(CCP1CON,0)) {
      CCP1CON = 0x0; //reset first
      CCP1CON = 0x4; //falling edge
    }
    else {
      CCP1CON = 0x0; //reset first
      CCP1CON = 0x5; // rising edge
    }
    edge_capture = 1;
  }
  do_ircap(); ← Call function that decodes captured edges
}

```

Compute number of Timer1 tics between last active edge and current active edge. Store this value in variable delta.

Reconfigure CCP1 to toggle active edge from falling edge to active edge or vice versa. Do this so can trigger on every edge.



**FIGURE 13.22** ISR for pulse-width measurement of biphas waveform (see CD-ROM file `./code/chap13/F_13_24_irdet_biphase.c`).

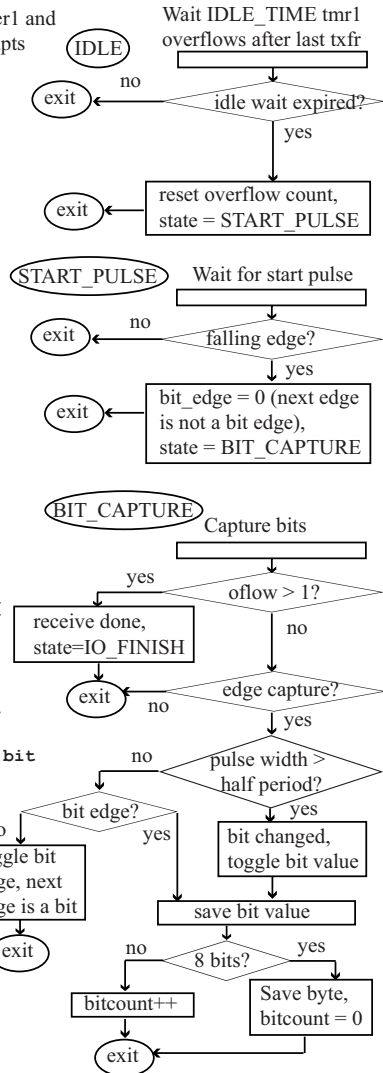
The real work of decoding the received biphas data is done by the `do_ircap()` function shown in Figure 13.23. The `#define BITCHANGE 10000` statement defines the number of Timer1 tics used to distinguish a measured pulse width between a half-period and a full period. The reference PIC18 system has  $FOSC = 29.4912$  MHz and is used to measure a biphas waveform with a bit period of  $1800 \mu s$ . Using a Timer1 prescaler value of 1, the Timer1 clock period is computed as  $[1/(29.4912E6/4)] * Timer1\_prescale = [1/(29.4912E6/4)] * 1 = 0.136 \mu s$ . Thus,

one period = 1800 us/ 0.136 us ~ 13235 Timer1 tics and one-half period is 6618 Timer1 tics, so 10000 is comfortably between these two values. A Timer1 overflow with a prescale value of 1 and a FOSC = 29.4912 MHz has an overflow period of approximately 8.9 ms.

```

// decode IR biphasse      Called for both timer1 and
void do_ircap(){          edge capture interrupts
    TMR1IF = 0; CCP1IF = 0;
    switch (state) {
    case IDLE:
        // wait for line to become idle
        if (tmr1_ov > IDLE_TIME){
            tmr1_ov = 0;
            state = START_PULSE;
            edge_capture = 0;
        }
        break;
    case START_PULSE:
        if (edge_capture) { // wait for edge
            edge_capture = 0;
            bit_edge = 0;
            state = BIT_CAPTURE;
        }
        break;
    case BIT_CAPTURE:
        // wait for edge or idle condition
        if (tmr1_ov > 1) { // finished
            // disable capture, timer1 interrupts
            CCP1IE = 0; TMR1IE = 0; TMR1ON = 0;
            state = IO_FINISH;
        } else if (edge_capture) {
            edge_capture = 0;
            //accumulating bits, MSB to LSB
            if ((delta > BITCHANGE) || bit_edge) {
                if (delta > BITCHANGE) {
                    // toggle current bit if wide pulse
                    current_bit = ~current_bit;
                }
                if (current_bit) bitset(this_byte,0);
                bitcount++;
                bit_edge = 0; // next edge is not a bit
                if (bitcount == 8) {
                    bitcount = 0;
                    cbuff[bytecount] = this_byte;
                    bytecount++;
                    this_byte = 0;
                } else{
                    this_byte = this_byte << 1;
                }
            } else if (!bit_edge)
                bit_edge = 1;
        }
        break;
    } //end switch
} // end do_ircap()

```



**FIGURE 13.23** Function do\_ircap() for decoding biphas serial data (see CD-ROM file ./code/chap13/F\_13\_24\_irdet\_biphase.c).

The `do_ircap()` function uses a finite state machine approach with states of `IDLE`, `START_PULSE`, `BIT_CAPTURE`, and `IO_FINISH`. The initial state of `IDLE` waits for the `tmr1_ov` variable to become greater than `IDLE_TIME` ( $= 4$ ), indicating that no edges have been received for approximately 36 ms. This is done to ensure that the state machine for IR reception is started when the input is idle. After the idle condition is detected, the state is changed to `START_PULSE`. In state `START_PULSE`, once an edge arrives indicating the arrival of the start pulse, the state is changed to `BIT_CAPTURE` and the `bit_edge` variable is changed to 0. If the `bit_edge` variable is 0, this means the next edge is expected to occur at the beginning of a bit period and is not a “1” or “0” bit transition edge. If the `bit_edge` variable is 1, this means the next edge is expected to occur in the middle of a bit period, indicating that a “1” or “0” has been received.

The `BIT_CAPTURE` state does the work of decoding “1”s and “0”s after the start pulse has been detected. Received bits are assumed to arrive MSb first and are stored in the `unsigned char this_byte` variable. The `bitcount` variable is used to keep track of the number of bits that have arrived; once 8 bits have been received the `this_byte` variable is written to the `cbuff` array. The `bytecount` variable tracks the number of bytes that have arrived. Within the `BIT_CAPTURE` state, if the overflow count is greater than one, this indicates the end of this transmission as no edges have arrived within one Timer1 overflow period so the next state is set to `IO_FINISH`, which is a semaphore to `main()` that indicates the receive has finished. If timer overflow is less than one but no edge capture has occurred, the state is exited (recall that `do_ircap()` is also called when a Timer1 overflow occurs). If an edge capture has occurred and the pulse width is greater than `BITCHANGE` Timer1 tics, this indicates that a data bit has been received and that it is the complement of the previous bit. The `current_bit` variable tracks the value of the last received bit so this value is complemented and is saved as the current bit value. If the pulse width is less than `BITCHANGE` Timer1 tics, a half-period pulse width has arrived; if the `bit_edge` variable is set, this indicates that this edge is a bit transition, so the `current_bit` variable is saved as the received bit value. If the `bit_edge` variable is cleared, it is set, as the next arrival edge is expected to be a bit transition edge.

Figure 13.24 shows the `main()` code for decoding biphasic serial data. Timer1 is configured for internal clock operation and a prescale of 1. The `reset_ir()` function (called by `main()` before the `while(1){}` loop is entered) configures the CCP1 capture input for falling edge triggering to capture the leading edge of the start pulse, resets the variables used by `do_ircap()`, and enables the Timer1 and CCP1 interrupts. The `current_bit` value is set to nonzero, as the biphasic protocol that this code was tested with always sent a “1” bit as the first data bit.

```

reset_ir() {
    state = IDLE; // look for idle
    CCP1CON = 0; // turn off when changing modes
    CCP1CON = 0x4; // capture every falling edge
    current_bit = ~(0);
    bitcount = 0; bytecount = 0;
    tmr1_ov = 0; // clear timer 1 overflow count
    last_capture = 0;
    // enable capture and timer1 interrupts
    CCP1IF = 0; CCP1IE = 1;
    TMR1IF = 0; TMR1IE = 1; TMR1ON = 1;
}

unsigned char i, t_bytecount, t_bitcount, t_this_byte, cnt, tbuff[MAXBYTES];
main(void) {
    serial_init(95,1); // 19200 in HSPLL mode, crystal = 7.3728 MHz
    // initialize timer 1, prescale by 1, internal clock
    T1CKPS1 = 0; T1CKPS0 = 0; T1OSCEN = 0; TMR1CS = 0;
    bitset(TRISC,2); // set CCP1 as input
    IPEN = 0; PEIE = 1; GIE = 1;
    cnt = 0;
    reset_ir();
    pcrLf(); printf("Ready for capture\n"); pcrLf();
    while(1) {
        // wait for IR data to arrive
        while (state != IO_FINISH);
        // copy interrupt data to safe place
        t_bytecount = bytecount;
        t_bitcount = bitcount;
        t_this_byte = this_byte;
        for (i = 0; i < t_bytecount; i++) {
            tbuff[i] = cbuffer[i]; cbuffer[i] = 0;
        }
        reset_ir();
        // print out last captured data
        if (t_bitcount != 0) {
            // adjust last byte assuming input bits are zero
            for (i=t_bitcount; i < 7; i++)
                t_this_byte = t_this_byte << 1;
            tbuffer[t_bytecount] = t_this_byte;
            t_bytecount++;
        }
        printf("%d): Received %d bytes, %d bits.",
            cnt, t_bytecount, t_bitcount); pcrLf();
        for (i = 0; i < t_bytecount; i++) {
            printf(" Byte RX: %x", tbuffer[i]); pcrLf();
        }
        cnt++;
    }
}

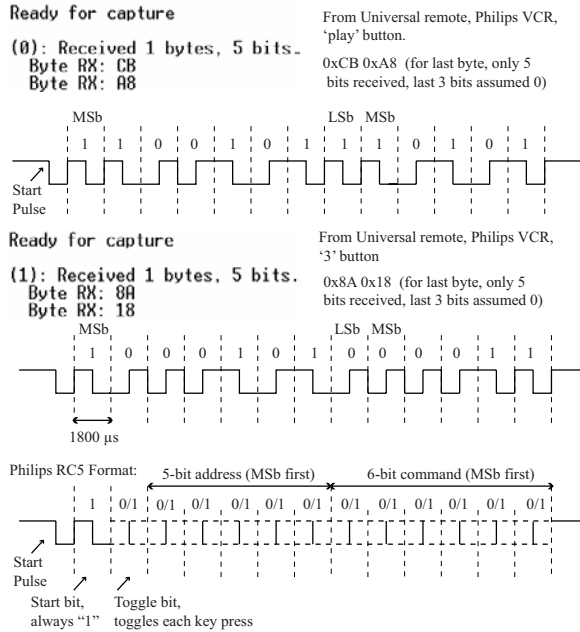
```



**FIGURE 13.24** main() for decoding biphas serial data.

The `while(1){}` loop waits for capture to be completed by the ISR via the `while(state!=IO_FINISH){}` statement. After this loop is exited, the variables used by the `do_ircap()` function are copied to temporary variables so that the IR capture can be re-enabled by a call to `reset_ir()` before the captured values are printed to the console. The `t_this_byte` value is padded with zeros and stored in the `tbuff` array if less than 8 bits were received for the last byte. At this point, the received byte values in the `tbuff` array are printed to the console.

Figure 13.25 shows the console output when testing the biphasc decode application using a universal remote control programmed for a Philips VCR. Each transmission sent two duplicate transmissions of 14 bits for a keypress on the universal remote control; only the first 14-bit transmission is shown.



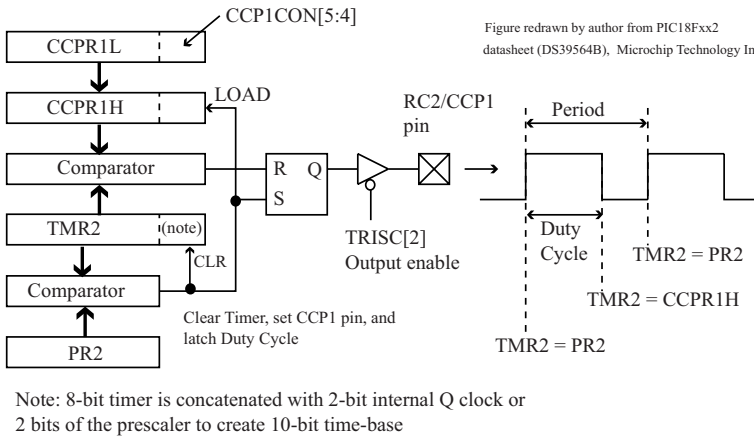
**FIGURE 13.25** Console output for decoding biphasc serial data.

The biphasc decode application does not understand the meanings of the bits and simply accumulates them into bytes as they arrive. The bottom waveform shows the meaning of the bits in Philips RC5 format. The first bit is a start bit and is always "1". The next bit is a toggle bit that is complemented on each keypress but remains the same if the button is held down, which causes the code to be repeatedly transmitted. The next 5 bits are an address field that specifies the type of device such as TV, VCR, SAT, and so forth. The last 6 bits is a command; observe that when a numeric button 0–9 is pressed, this contains the value of the numeric button.



### 13.7 TIMER2 AND PULSE WIDTH MODULATION

Pulse width modulation (PWM) is a technique that varies the duty cycle of a square wave in order to vary the average current delivered to an external device. PWM capability is provided through the use of Timer2 and the CCP1 registers as shown in Figure 13.26.



**FIGURE 13.26** PWM operation.<sup>5</sup>

The PR2 register sets the period of the generated square wave, while the CCP1H register provides the duty cycle. A match of the PR2 register and TMR2 value sets the CCP1 pin high, clears the TMR2 register, and transfers the CCP1L value to CCP1H to fix the duty cycle. A match of TMR2 with CCP2H resets the CCP1 pin, thus terminating the high portion of the square wave. Both PR2 and the duty cycle are extended to 10 bits of precision; the PR2 register by using the 2-bit internal Q clock or 2 bits of the prescaler and the duty cycle by using the CCP1CON[5:4] bits as the lower 2 bits of the 10-bit duty cycle value. Equation 13.3 gives the period of the resulting square wave, while Equation 13.4 gives the duty cycle as a time value.

$$\text{PWM Period} = (\text{PR2} + 1) * 4 * \text{TOSC} * \text{TMR2\_PRE} \tag{13.3}$$

$$\text{PWM Duty Cycle} = (\text{CCP1L} : \text{CCP1CON}[5:4]) * \text{TOSC} * \text{TMR2\_PRE} \tag{13.4}$$

If the lower 2 bits of the 10-bit duty are cleared (CCP1CON[5:4] = 00), the duty cycle as a percentage of the resulting square wave is given by Equation 13.5.

<sup>5</sup> Figure 13.26 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.'s prior written consent.

$$\% \text{ PWM Duty Cycle} = \frac{\text{CCPR1L}}{(\text{PR2} + 1)} \times (100\%) \quad (13.5)$$

If CCPR1L is greater than PR2, the SR latch of Figure 13.26 is never reset, as Timer2 is always reset before it can become equal to CCPR1H causing the CCP1 pin to remain high (100% duty cycle). A CCPR1L value of zero is a special condition that causes the CCP1 pin to always remain low (0% duty cycle). If PR2 is the maximum value of 255, a 100% duty cycle cannot be achieved.

## A PWM Example

Figure 13.27 shows how PWM can be used to control the brightness of an LED. The `main()` code configures the CCP1 pin for PWM mode and initially sets the duty cycle to 50% by setting the PR2 register to 255 and CCPR1L to 128 (bits CCP1CON[5:4] are cleared). The `while(1){}` loop prompts the user to enter the CCPR1L value, which is then written to CCPR1L, updating the duty cycle. The graph illustrates how the LED current varies linearly with duty cycle; a low duty cycle dims the LED since it decreases average current, while a high duty cycle brightens the LED as average current is increased.

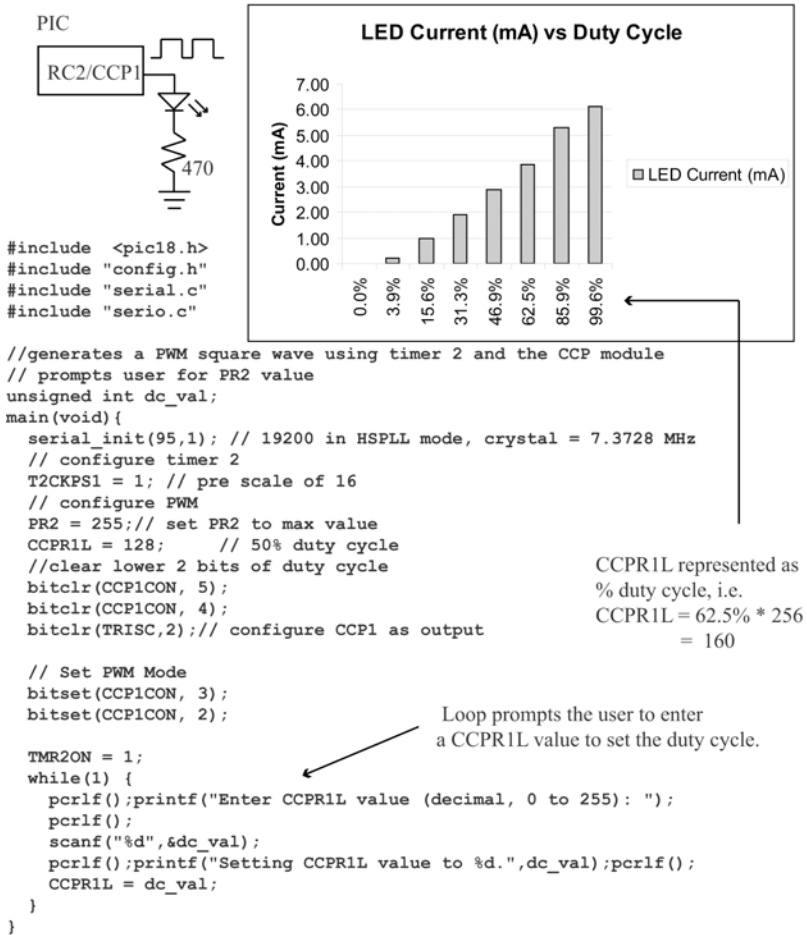
Figure 13.28 shows PWM control of a small DC motor such as that found in hobbyist robotic kits. The gate of the MOSFET is controlled by the PWM signal; the MOSFET is turned on when the PWM signal is high, thus modulating the current flow through the motor.

The motor speed is proportional to the PWM duty cycle. The diode, known as a *snubber* diode, is optional for small motors. The diode protects against voltage spikes that are induced if the motor continues to spin due to inertia after the MOSFET is turned off. The switches control the rotation direction of the DC motor. Low resistance analog switches such as the single-pole, double throw (SPDT) PI5A319 from Pericom [17] or the CD4053B triple two-channel analog multiplexer [18] from Texas Instruments provide a method for direction control using a parallel IO line from the PIC18.

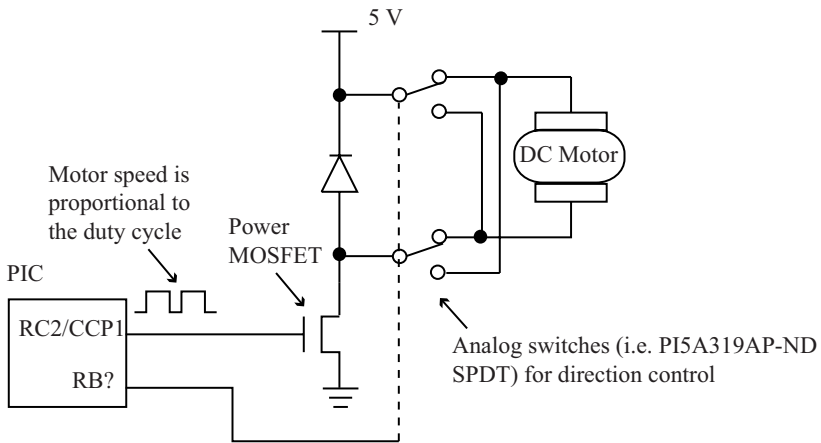
**Sample Question:** Assume a FOSC of 10 MHz. Give the PR2 and prescale values for the PWM mode that generates a square wave with a 75% duty cycle and a period of 6 kHz on the CCP1 output pin. Use the prescale value that gives the largest PR2 value. Only give the upper 8-bit value for the duty cycle register (CCPR1). Write code for `main()` that configures the PIC18 for this mode and ends with a `while(1){}` empty loop since the PWM hardware does all of the work of generating the square wave.

**Answer:** From Equation 13.3, Timer2 PWM period = (PR2+1) \* (4/FOSC) \* PRE (recall that the postscaler is NOT used for PWM period). Then:  
 (1/6 kHz) = (PR2+1) \* (4/10 MHz) \* PRE  
 PR2 = [(10 MHz/4) / (6 KHZ \* PRE)] - 1 →

For PRE = 1, PR2 = 416 (> 255, so too large). For PRE = 4, PR2 = 103.  
 For PRE = 16, PR2 = 25, so use PRE = 4, PR2 = 103.  
 For a 75% duty cycle, CCPR1L = 0.75 \*(PR2+1) = 0.75\*104 = 78.  
 Code to configure the PIC18 for this mode of operation is show in Listing 13.2.



**FIGURE 13.27** PWM control of an LED.

**FIGURE 13.28** PWM control of a DC motor.**LISTING 13.2** Configuring for PWM mode.

```

main(){
    T2CKPS1 = 0; T2CKPS0 = 1; // pre scale of 4
    // configure PWM
    PR2 = 103;                // set PR2 to max value
    CCP1L1 = 78;
    //clear lower 2 bits of duty cycle
    bitclr(CCP1CON, 5);
    bitclr(CCP1CON, 4);
    bitclr(TRISC,2);         // set CCP1 output

    // Set PWM Mode
    bitset(CCP1CON, 3); bitset(CCP1CON, 2);
    TMR2ON = 1;
    while(1);                // infinite loop, PWM hardware does all work
}

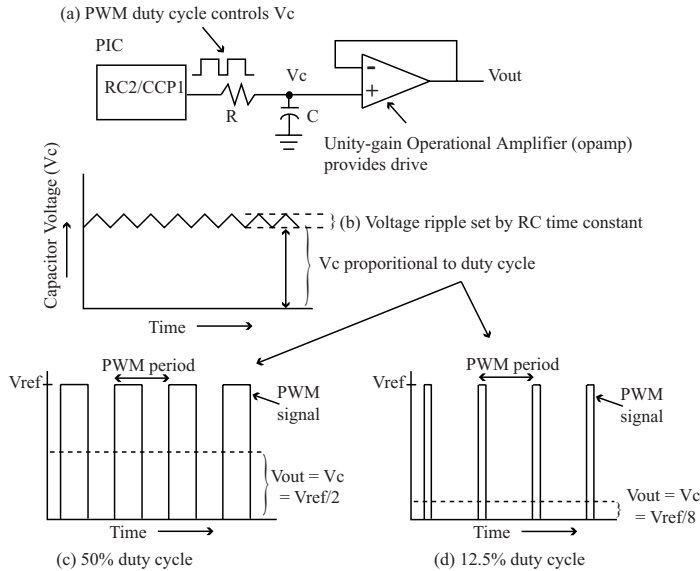
```

**A PWM DAC**

(Warning: This section assumes some reader knowledge of time domain and frequency domain characteristics of RC circuits; you may want to skip this section if you do not have this background.)

One other common application of PWM is as a “poor man’s digital-to-analog converter” in which the PWM signal is applied to a series resistor/capacitor network to produce a DC voltage that is proportional to the PWM duty cycle. This is illustrated in Figure 13.29. The operational amplifier is needed to provide current drive for whatever load is being driven by this voltage, as any current drained from the capacitor affects the voltage value. In this configuration, the operational

amplifier provides a gain of 1 (unity gain, voltage follower configuration), with the input current into the plus (“+”) terminal being negligible.



**FIGURE 13.29** PWM to control RC voltage.

The RC series network forms a low-pass filter that is driven by the PWM signal. The low-pass filter removes most of the high frequency content of the PWM signal (e.g., the switching component), leaving only the DC, or average value behind. The average value at the low-pass filter output is the desired DAC output. Figure 13.29c shows the PIC generating a PWM signal with a 50% duty cycle; the PWM signal is high 50% of the time. The RC filter removes the high frequency components and thus  $V_c = V_{out} = V_{ref}/2$ . Figure 13.29d shows a result with a 12.5% duty cycle PWM signal where  $V_c = V_{out} = V_{ref}/8$ . The voltage ripple in Figure 13.29b is the high frequency content of the PWM signal that has been attenuated by the low-pass filter.

Let’s examine the passive RC low-pass filter in Figure 13.29a. The filter’s cutoff frequency in radians/sec is  $\omega_0 = 1/(RC)$ . The filter’s natural cutoff frequency in Hertz is  $f_0 = \omega_0/(2\pi)$ . Beyond the cutoff frequency, the filter attenuates the PWM signal frequencies at 20 dB/decade. Therefore, we can expect the PWM signal components at  $10 \cdot f_0$  to be approximately 20 dB below those components at  $f_0$ , and signal components at  $100 \cdot f_0$  to be attenuated 20 dB below those at  $10 \cdot f_0$  and 40 dB below those at  $f_0$ . With this information, we can see that if the PWM frequency is well beyond the low-pass filter’s cutoff frequency, the switching (high frequency)

components will be greatly attenuated leaving behind only the PWM's low frequency components near DC. This filtering gives us the desired result, as a signal's DC component is equivalent to its average value.

Selection of the exact RC filter values is application specific, but some general rules of thumb can be helpful in getting started. The RC filter's cutoff frequency needs to be sufficiently high so that the highest frequencies from the DAC are not attenuated. Try using  $f_0$  equal to 5–10 times the PWM DACs sampling frequency. The PWM frequency should be far into the low-pass filter's stop band. PWM frequencies should be at least 100 times the low-pass filter cutoff so that PWM switching signal components are reduced by 40 dB or more. The PWM frequency can be reduced if the low-pass filter has strong attenuation. Multiple RC low-pass filter sections can be cascaded so that each section provides 20 dB/decade attenuation.

If the PWM DAC is only to supply a fixed reference voltage, the previous recommendations can be relaxed, with the principle goal to produce a reference voltage that meets some maximum voltage ripple specification. Equation 13.6 gives an approximate value for the RC time constant given a power supply voltage (Vdd), desired ripple, and PWM period. Equation 13.6 assumes that the RC time constant is at least 10 times greater than the period of the PWM duty cycle.

$$RC = \frac{V_{dd} * \text{PWM period} * 0.37}{\text{ripple}} \quad (13.6)$$

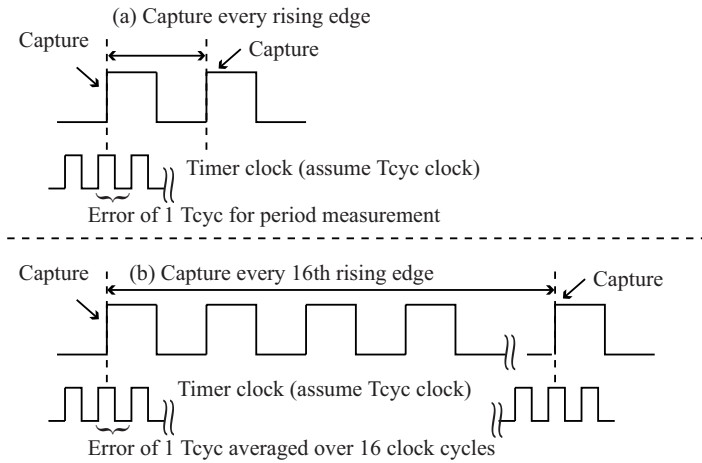
For example, if  $V_{dd} = 5 \text{ V}$ , PWM period = 0.5 ms (2 kHz), and ripple = 0.1 V, RC is computed as 0.009 s. An RC time constant of 0.009 can be approximated by using common R, C component values of  $R = 10 \text{ K}\Omega$ ,  $C = 1.0 \text{ }\mu\text{F}$  for an RC time constant of 0.01 s.

LCD displays that require positive and negative bias voltages outside of the supply rails often use a PWM signal driving a charge pump circuit to produce these voltages. This is similar to what is done internally by the MAX202/MAX232 RS232 driver chip (Chapter 9, "Asynchronous Serial IO") to produce  $\pm 10 \text{ V}$  from a 5 V supply.

## **13.8 USING CAPTURE MODE FOR FREQUENCY MEASUREMENT**

---

The last example of this chapter uses the PWM module to generate a square wave and the capture mode to measure its period. The timer base for capture mode is Timer1 clocked by the internal instruction clock. The capture mode prescaler is set to perform captures every 16<sup>th</sup> rising edge. If the square wave frequency is steady during the measurement period, using the prescaler means that the one instruction cycle uncertainty in the time measurement is spread over 16 periods instead of only one period as shown in Figure 13.30.



**FIGURE 13.30** Using the capture mode prescaler to reduce measurement error.

Figure 13.31 shows the ISR code for square wave period measurement. This is similar to the pulse width measurement code of Figure 13.10 except the active edge is not changed and the current edge capture time (`this_capture`) is saved as the last

```
volatile unsigned int last_capture, this_capture;
volatile unsigned long delta; // this must be long
volatile unsigned char tmr1_ov; // timer 1 overflow cnt
volatile unsigned char capture_flag;

void interrupt timer_isr(void){
    if (TMR1IF) {
        tmr1_ov++; // increment timer1 overflow
        TMR1IF = 0;
    }
    if (CCP2IF) {
        CCP2IF = 0; // CCP2IF = 0; clear capture interrupt flag
        this_capture = CCPR2; // read CCPR2 as 16-bit value
        if (!tmr1_ov) {
            // no overflow at all
            delta = this_capture - last_capture ;
        }
        else {
            delta = tmr1_ov-1;
            delta = (delta << 16);
            last_capture = 0 - last_capture;
            delta = delta + last_capture;
            delta = delta + this_capture;
        }
        last_capture = this_capture;
        tmr1_ov = 0;
        capture_flag++;
    }
}
}
```

Capture time is Timer1 tics between every 16th rising edge.

Because this is continuous capture, `last_capture` is set equal to `this_capture` before exit.



**FIGURE 13.31** ISR code for square wave period measurement (see CD-ROM file `./code/chap13/F_13_32_sqwavemeas.c`).

edge capture time (last\_capture) because the period is continuously being measured. The delta variable is set equal to the elapsed Timer1 tics between edge captures.

Figure 13.32 gives the main() code for the square wave period measurement. Timer1 is configured for a prescale of one and internal clock operation.

```

#define FOSCQ 29491200
#define TMR1PRE 1.0
#define TMR1TIC 1.0/(FOSCQ/4.0)*TMR1PRE
#define TMR2PRE 4.0

double period_float;
unsigned int meas_period,exp_period;
int pr2_val;

main(void) {
    serial_init(95,1); // 19200 in HSPLL mode, crystal = 7.3728 MHz
    // initialize timer 1
    T1CKPS1 = 0; T1CKPS0 = 0; // prescale by 1
    T1OSCEN = 0; // disable the oscillator
    TMR1CS = 0; //use internal clock FOSC/4
    T1SYNC = 0;
    bitset(TRISC,1); // configure CCP2 as input
    // configure capture
    CCP2CON = 0x07; // capture every 16th edge
    CCP2IF = 0; // clear CCP2IF interrupt flag
    CCP2IE = 1; // enable capture interrupt
    // turn on timer1
    TMR1IF = 0; TMR1IE = 1; TMR1ON = 1; // enable timer 1
    tmr1_ov = 0;
    pcrLf(); printf("Enter PR2 Value: "); // ← Enter PR2 value
    scanf("%d",&pr2_val); pcrLf();
    period_float = ((pr2_val+1)*4*TMR2PRE*1.0e9)/FOSCQ; } Compute expected
    exp_period = (unsigned int) period_float; } period in nanoseconds
    bitclr(TRISC,2); // set CCP1 as OUTPUT for pwm
    // configure timer2
    T2CKPS1=0;T2CKPS0=1; // prescale of 4
    // set up PWM
    PR2 = pr2_val; // set period
    CCP1RL = (pr2_val >> 1); // 50% duty cycle
    DC1B1=0; DC1B0=0;
    CCP1M3 = 1;CCP1M2 = 1; // PWM Mode
    TMR2ON=1;
    printf("Squarewave Measure Enabled"); pcrLf();
    IPEN = 0; PEIE = 1; GIE = 1; // enable interrupts
    while(1) {
        capture_flag = 0;
        while(!capture_flag); // wait for capture
        delta_old = delta >> 4; // divide by 16 for true freq } Wait for capture,
        period_float = (delta_old * TMR1TIC)*1.0e9; } value. Divide count by
        meas_period = (unsigned int) period_float; } 16 because capture
        printf ("Expected period: %u (ns), Measured period: %u (ns)", } prescaler is 16
            exp_period,meas_period);
        pcrLf();
        printf("Hit a key to continue."); pcrLf();
        getch();
    }
}

```

Squarewave generated  
by Timer2, PWM

PIC18



**FIGURE 13.32** main() code for square wave period measurement.



The capture mode is configured for every 16<sup>th</sup> edge using CCPR2 as the capture register because the CCPR1 capture register is used for PWM operation. The user is prompted to enter the PR2 value used to set the PWM period. The expected period is computed in nanoseconds by the statements:

```
period_float = ((pr2_val +1)*4 * TMR2PRE*1.0e6)/FOSCQ;
exp_period = (unsigned int) period_float;
```

where `pr2_val` is the PR2 register value and `TMR2PRE` is the Timer2 prescale value. Timer2 is configured for prescale by 4 and the PWM mode configured for a 50% duty cycle. The `while(1){}` loop waits for an edge capture as indicated by a nonzero value in the `capture_flag` semaphore. The `delta` value that contains the elapsed timer tics between 16 rising edges is divided by 16 by the statement `delta_old = delta >> 4` to get the period of one cycle. The measured period value is then computed in nanoseconds by the statements:

```
period_float = (delta_old*4*TMR1PRE*1.0e6)/FOSCQ;
meas_period = (unsigned int) period_float;
```

where `TMR1PRE` is the Timer1 prescale value. The expected and measured period values are then printed. Figure 13.33 shows the console output of the square wave measurement code for three different PR2 values. The expected measurement error of one instruction cycle over 16 input waveform cycles is computed for an instruction cycle time of  $FOSC=29.4912/4$  MHz (period = 135.6 ns) as  $135.6 \text{ ns}/16 = 8.5 \text{ ns}$ . The close agreement between expected and measured values of Figure 13.33 is to be expected since Timer1 and Timer2 are both clocked by the instruction cycle clock. The measurement error will only be apparent when measuring the frequency of an external clock source that is not synchronized to the internal clock.

```
Enter PR2 Value: 78
Squarewave Measure Enabled
Expected period: 42860 <ns>, Measured period: 42860 <ns>
Hit a key to continue. ←———— Pressed Reset

Enter PR2 Value: 34
Squarewave Measure Enabled
Expected period: 18989 <ns>, Measured period: 18988 <ns>
Hit a key to continue. ←———— Pressed Reset

Enter PR2 Value: 56
Squarewave Measure Enabled
Expected period: 30924 <ns>, Measured period: 30924 <ns>
Hit a key to continue.
```

**FIGURE 13.33** Console output for square wave period measurement.

**SUMMARY**

Table 13.5 gives a summary of the PIC18 timers. A partial list of timer uses include real-time clocks, periodic interrupt generation, time measurement of external or internal events, and waveform generation.

The internal instruction clock that has frequency  $F_{OSC}/4$  can be used as the clock source for all timers. Additionally, an external clock on the T0CKI pin can function as the clock source for Timer0. The clock source for Timer1 and Timer3 can be an external clock on the T1CKI pin or a crystal connected across the T1OSO/T1OSI pins. A precision real-time clock can be implemented by using an external 32.768 kHz clock source with a 16-bit timer; each timer rollover occurs at two-second intervals. The Capture/Compare/PWM module has internal CCPR1/CCPR2 registers (16 bits) and the PR2 register. The Capture module interacts with Timer1/Timer3 to provide time measurement between external events. This capability was used to decode biphas-encoded IR signals generated by a universal remote control. Biphas encoding uses a falling edge in the center of a bit period to signal a “1” and a rising edge to signal a “0”. Space width encoding is an

**TABLE 13.5** PIC18 Timer Summary

<b>Timer</b>	<b>Size</b>	<b>Prescaler</b>	<b>Postscaler</b>	<b>Ext. Clock?</b>	<b>Special Features</b>
Timer0	8 or 16 bits	256, 128, 64, 32, 16, 8, 4, 2; can be disabled	No	Yes	None
Timer1	16 bits	8, 4, 2, 1	No	Yes, also ext. crystal	Use with capture module for event measurement; use with compare module for waveform generation or period control.
Timer3	16 bits	8, 4, 2, 1	No	Yes, also ext. crystal	Same as Timer1; can also trigger A/D conversion on compare match.
Timer2	8 bits	16, 4, 1	1:1, 1:2, 1:3, 1:4, ... 1:16	No	Used as timebase for PWM via the PR2, CCPR1 registers.

alternate encoding method that uses different period widths to distinguish “1”s and “0”s. The Compare module interacts with Timer1/Timer3 and is useful for square wave generation, as the external pins CCP1 or CCP2 can be toggled on each successful match of a compare register with its associated timer register. Pulse width modulation (PWM) is a technique that varies the duty cycle of a square wave to modulate the current delivered to an external device. Applications for PWM include DC motor control and a simple digital-to-analog converter built from an RC network and a unity-gain operational amplifier. The PWM module of the PIC18 uses Timer2 as the time base, with the PR2 register providing the period and the CCP1L register controlling the duty cycle.

## REVIEW PROBLEMS

---

1. Given a 40 MHz FOSC and a prescale value of 2, what is the interrupt interval in microseconds for each rollover of Timer1?
2. Given a 12 MHz FOSC, it is desired to generate a periodic interrupt as close as possible to 1 ms using Timer0. Give the prescale value and mode (8-bit or 16-bit) as well as the actual interrupt period obtained.
3. Given a 20 MHz FOSC, what is the longest period interrupt that can be generated using Timer0?
4. Given a 15 MHz FOSC and a prescale value of 8, how many Timer1 tics equals to 20 ms?
5. Given a 30 MHz FOSC and a prescale value of 8, how many Timer1 tics equals to 5 ms?
6. Given the code of Figure 13.10, an FOSC of 40 MHz, and a prescale of 2, what is the longest pulse width that can be measured?
7. Why is the statement `delta = delta << 16` used in the code of Figure 13.10? Give an alternate method of implementing this using a pointer. (Hint: See Listing 13.1.)
8. Using the approach of Figure 13.15b, generate a square wave of frequency 8 kHz with duty cycle of 50% assuming a 40 MHz FOSC.
9. Using the approach of Figure 13.15b, generate a square wave of frequency 2 kHz with duty cycle of 25% assuming a 25 MHz FOSC. (Hint: To generate a square wave with a duty cycle other than 50%, you will need two different values to add to the CCP1R register, one for the high pulse width and one for the low pulse width.)
10. Draw a biphas encoded waveform for the 8-bit value 0x3B (MSb first).
11. Draw a biphas encoded waveform for the 8-bit value 0x45 (MSb first).

12. Draw a space-width encoded waveform for the 8-bit value 0x3B (MSb first). Assume that a “0” has a 25% duty cycle, a “1” has a 50% duty cycle, and that a start bit is a 50% duty cycle with a period of approximately 3x that of the “1”, “0” period.
13. Draw a space-width encoded waveform for the 8-bit value 0x45 (MSb first). Assume that a “0” has a 25% duty cycle, a “1” has a 50% duty cycle, and that a start bit is a 50% duty cycle with a period of approximately 3x that of the “1”, “0” period. A “0” is 2x the period of a “1.”
14. Describe in general the changes that would have to be made to the code of Figure 13.23 to decode space-width encoded data.
15. Give Timer2 prescale, PR2, and CCP1 values that will generate a 3 kHz square wave with a duty cycle of 30% assuming a 20 MHz FOSC using the PWM module.
16. Assume PWM mode, if PR2 is 0xA0 and CCP1 is 0x30, what is the duty cycle of the square wave?
17. Write C code that will count the number of falling edges of an input waveform and print the result using the Capture/Compare/Module. Assume the serial port and TIMER1 have already been configured in some manner. Your code has to configure the CCP1 module, and enable both TIMER1 and CCP1 interrupts. Prompt the user to press any key. After the first falling edge is detected, begin counting falling edges. If no falling edges occur for > 50 Timer1 interrupts, assume the input is idle, and print out a message that contains the number of falling edges that occurred. Use an *int* variable to hold the number of falling edges. Your code must have a clearly identified interrupt service routine; assume any code outside of the ISR is in `main()`.
18. Assume Timer1 has been programmed to generate a periodic interrupt. Write C code that configures Timer2 for prescale by 4 and the PWM module for a maximum period square wave with a 50% duty cycle. On each Timer1 interrupt, change the duty cycle in the following sequence: 60%, 70%, 80%, 90%, 100%, 0%, 10%, 20%, 30%, 40%, 50% (repeat).
19. Assume that you need two manually generated PWM signals on pins RB3 and RB4. Using Timer3, CCP2 compare mode, and a 20 MHz FOSC, generate a periodic interrupt at a frequency of 10 kHz. Using this interrupt, generate square waves on pins RB3 and RB4 with a frequency of 1 kHz (10 Timer3 interrupt periods). Define two variables `dc_rb3` and `dc_rb4` that allow the duty cycle of each square wave to be set to one of 11 values (0%, 10%, 20% ... up to 100%).
20. Assume an FOSC of 40 MHz. With a prescale of 16 for the CCP1 input, what is the expected percent error in measuring the frequency of a 75 kHz square wave input using capture mode?

*This page intentionally left blank*

# 14

## Capstone: Audio Sampling, Monitoring System, and Autonomous Robot

### In This Chapter

- Design of an Audio Record/Playback System
- Implementation of an Audio Record/Playback System
- Design of a Home Monitoring System
- The DS1621 Digital Thermometer
- Using the Nonvolatile Storage on the PIC18Fxx2
- Implementation of a Home Monitoring System
- Design and Implementation of an Autonomous Robot

This chapter presents three capstone projects that combine various hardware topics from the previous chapters. The projects are a digital recorder that can store audio input and play it back; a monitoring system with a real-time clock, motion sensor, and temperature sensor; and a three-wheeled robot that can be remotely controlled via a universal remote control or function autonomously using an IR proximity sensor to avoid obstacles. Functions for writing to the PIC18 Flash program memory and Data EEPROM as well as an interface to a DS1621 Digital Thermometer are presented on a just-in-time basis to support these capstone projects.

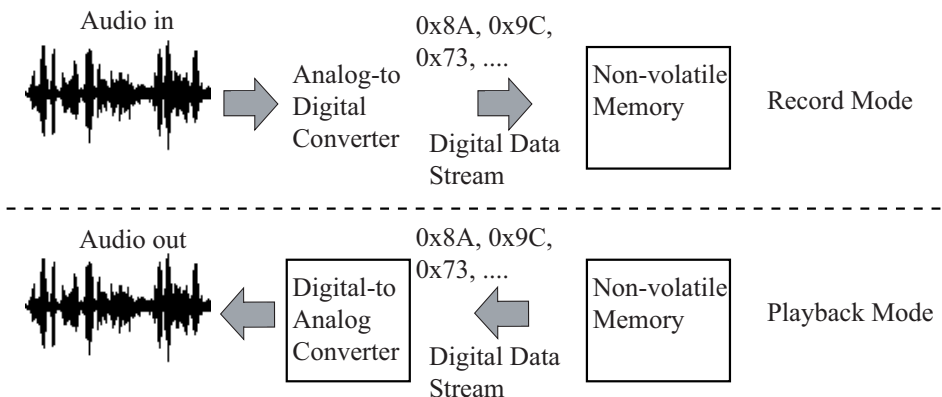
## 14.1 LEARNING OBJECTIVES

After reading this chapter, you will be able to:

- Design a system that can sample an audio signal, store it to a serial EEPROM, and then play it back.
- Design a monitoring system with a real-time clock, motion sensor, and temperature sensor.
- Use the DS1612 I<sup>2</sup>C temperature sensor with the PIC18 for monitoring temperature.
- Use the PIC18 internal Data EEPROM or Flash program memory for storing nonvolatile data.
- Design and build an autonomous wheeled robot with an IR interface using a PIC18 for control functions.

## 14.2 DESIGN OF AN AUDIO RECORD/PLAYBACK SYSTEM

The first project is a PIC18 system for capturing audio data for later playback. The simplest form of audio sampling uses a fixed sampling period and stores uncompressed digital data to a memory device as shown in Figure 14.1. A key parameter in audio recorders is the *sampling period*, which is the time between conversions of the incoming audio to digital data. The sampling period is fixed, and the sampling period used for playback must be the same as that used for record to faithfully reproduce the sampled audio. The inverse of the sampling period is the sampling frequency.



**FIGURE 14.1** Basic audio record/playback concept.

The quality of the audio playback improves as the sampling frequency is increased at the cost of increased memory requirements for audio storage. Audio is usually divided into two categories: music and speech. For music, sampling frequencies range from approximately 14 kHz up to 48 kHz. A sampling frequency of 8 kHz is considered adequate for speech data and is commonly used in voice recorders. A sampling frequency of 8 kHz has a sampling period of 125  $\mu$ s. Assuming 8-bit data, a 64K byte EEPROM can store  $64 * 1024 * 125 \mu\text{s} = 8.192 \text{ s}$  of speech.

For this design, our target is voice sampling at 8 kHz. The 24LC515 serial EEPROM is used for audio data storage, as we have previous experience in using that device for storing streaming input data from the serial port. The PIC18 ADC is used for conversion during record, and the MAX 517 DAC is used for data conversion during playback. The MAX 517 is the same as the MAX 518 discussed in Chapter 12, “Data Conversion,” except that it has only one internal DAC instead of two. During audio recording, the incoming digitized audio data is a continual data stream. This problem was first examined in Chapter 11, “Synchronous Serial IO,” for streaming data from the asynchronous serial port. In Section 11.9, an interrupt-driven double-buffered approach was used in which one buffer was designated as the active buffer to hold incoming data while the contents of the second buffer was emptied; in other words, stored to EEPROM. Once a buffer became full, the roles of the two buffers were swapped with the empty buffer becoming the buffer used for incoming data and the full buffer becoming the buffer whose contents are written to EEPROM. This same approach is used to handle the incoming audio data. We must ensure that the data rate of the outgoing data channel (EEPROM bandwidth) is greater than the data rate of the incoming data channel (audio data sampled at 8000 bytes/sec) as shown in Equation 14.1. Another way to state this constraint is shown in Equation 14.2, in which the time for 64 audio samples must be greater than the time it takes to store 64 bytes in a block write to the serial EEPROM. If this is not true, buffer overflow will occur.

$$\text{EEPROM Bandwidth (outgoing)} > \text{Audio Bandwidth (incoming)} \quad (14.1)$$

$$\text{Time for sampling 64 bytes} > \text{EEPROM write time} + \text{I2C transmit time} \quad (14.2)$$

Equations 14.3 through 14.5 show the calculation of the left and right quantities of Equation 14.2 assuming an I<sup>2</sup>C bus frequency of 400 kHz (1 bit time = 1/400 kHz).

$$125 \mu\text{s} * 64 \text{ bytes} > 5 \text{ ms (write time)} + (64 \text{ bytes} * 9 \text{ +start+stop}) * 1 \text{ bit time} \quad (14.3)$$

$$8 \text{ ms} > 5 \text{ ms} + (578) * (1/400 \text{ kHz}) \quad (14.4)$$



$$8 \text{ ms} > 6.4 \text{ ms} \tag{14.5}$$

Recall that each I<sup>2</sup>C transmission is 9 bit times because of the acknowledge bit; hence the 64\*9 value in Equation 14.3. The 5 ms constant is the worst-case write completion time for the 24LC515 serial EEPROM. The right-hand side of Equation 14.2 ignores the software loop overhead of sending the data bytes to the MSSP subsystem for I<sup>2</sup>C transmission. Equations 14.6 and 14.7 add a conservative estimate of 20 instruction cycles (1 instruction cycle ~ 0.135 μs @ FOSC = 29.4912 MHz) per byte. For a sampling period of 8 kHz, the constraint expressed by Equation 14.2 is satisfied as shown by Equation 14.7.

$$8 \text{ ms} > 6.4 \text{ ms} + 64 \text{ bytes} * 20 \text{ instr. cycles} * 0.135 \mu\text{s} \tag{14.6}$$

$$8 \text{ ms} > \sim 6.6 \text{ ms} \tag{14.7}$$

For playback, an 8-bit data sample has to be read from the serial EEPROM and written to the MAX 517 DAC in 125 μs if a playback rate of 8 kHz is to be achieved. Equations 14.8 through 14.11 calculate the approximate I<sup>2</sup>C bus rate to support an 8 kHz playback. Each EEPROM read requires an address byte sent to the EEPROM and the returned data byte from the EEPROM, while each DAC update requires an address byte, command byte, and data byte.

$$\text{Serial EEPROM read} + \text{DAC update} < 125 \mu\text{s} \tag{14.8}$$

$$[(2 * 9 + \text{start} + \text{stop}) + (3 * 9 + \text{start} + \text{stop})] * \text{bit time} < 125 \mu\text{s} \tag{14.9}$$

$$[20 + 29] * \text{bit time} < 125 \mu\text{s} \tag{14.10}$$

$$\text{bit time} < 2.55 \mu\text{s} \rightarrow \text{I}^2\text{C frequency} > \sim 390 \text{ kHz} \tag{14.11}$$

The calculated I<sup>2</sup>C frequency of 390 kHz is somewhat optimistic, as no software overhead is included. Equations 14.12 through 14.14 calculate the needed I<sup>2</sup>C bus frequency using the previous assumption of 20 instruction cycles per byte.

$$[20 + 29] * \text{bit time} + 5 \text{ bytes} * 20 \text{ instr. cycles} * 0.135 \mu\text{s} < 125 \mu\text{s} \tag{14.12}$$

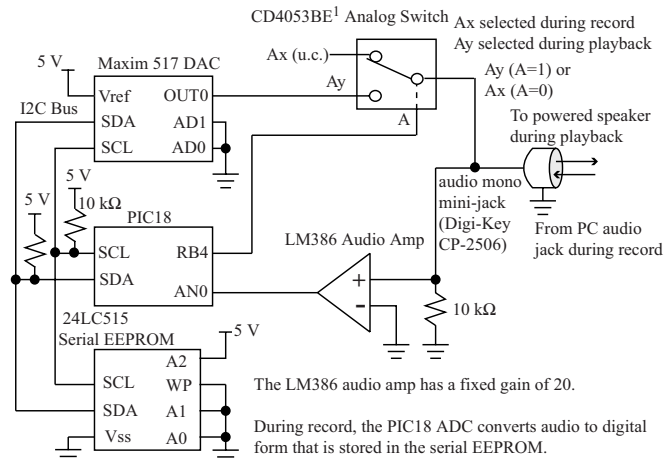
$$49 * \text{bit time} < 111.5 \mu\text{s} \tag{14.13}$$

$$\text{bit time} < 2.28 \mu\text{s} \rightarrow \text{I}^2\text{C frequency} > \sim 440 \text{ kHz} \tag{14.14}$$

The required I<sup>2</sup>C bus frequency of 440 kHz is above the maximum 400 kHz I<sup>2</sup>C bus speed specified in the datasheets for the 24LC515 serial EEPROM and MAX 517 DAC. While it is usually possible to clock components faster than their datasheet specification in a laboratory environment, one would never design a product that depended on component over-clocking. As such, we will settle for a 6 kHz sample and playback rate using the 24LC515 serial EEPROM and MAX 517 DAC. If you build the implementation presented in the next section, you are encouraged to attempt an 8 kHz record/playback rate by over-clocking the I<sup>2</sup>C bus to the 24LC515/MAX 517 components.

### 14.3 IMPLEMENTATION OF AN AUDIO RECORD/PLAYBACK SYSTEM

Figure 14.2 gives the schematic for the audio record and playback implementation. An audio mono mini-jack (Digi-Key PN# CP-2506) is used to interface the PIC18 reference board to a personal computer that provides audio during record or to powered speakers during playback.



<sup>1</sup>Tie INH, VSS, VEE pins to ground.

The LM386 audio amp has a fixed gain of 20.  
 During record, the PIC18 ADC converts audio to digital form that is stored in the serial EEPROM.  
 During playback, digital audio is retrieved from the serial EEPROM and converted by the MAX 517 DAC to analog voltages that drive an external powered speaker.

**FIGURE 14.2** Audio record/playback schematic.

The LM386 audio amplifier [19] provides a fixed gain of 20 in the minimal-external component configuration shown in Figure 14.2. The audio amplifier is

needed, as the output signal provided by the audio output jack of a personal computer typically has a peak-to-peak range of only a couple of hundred millivolts. During recording of the audio signal, the combination of the fixed 20x gain of the LM386 and the volume control on the PC provides the means for controlling the input signal magnitude to the PIC18 ADC. The LM386 also biases its output swing about  $V_{dd}/2$ , a nice feature that provides maximum data resolution when sampled by the PIC18 ADC when the reference voltages are configured as  $V_{ref+} = V_{dd}$ ,  $V_{ref-} = V_{ss}$ .

The CD4053B triple two-channel analog multiplexer [18] from Texas Instruments is used to implement a switch controlled by the RB4 output of the PIC18. During record, RB4 is low, which disconnects the MAX517 DAC output from the mini-jack. RB4 is high during playback allowing the MAX517 DAC output to drive the mini-jack connector. During playback, external powered speakers with volume control can be used for amplifying the output signal, or the mini-jack output can be connected to the audio input of a PC.

Table 14.1 gives the PIC18 resources used for the audio record/playback application. Timer2 is used to generate the periodic interrupt that sets the sampling rate for playback and record. The PIC18 ADC with a reference voltage of  $V_{dd}$  is used to sample the audio signal during record mode. The I<sup>2</sup>C bus is used to communicate with the serial EEPROM for storing the sampled audio, which is converted back to analog form by the MAX517 DAC during playback. The asynchronous serial port is used to communicate with the user during application execution to prompt the user for operating mode choice.

**TABLE 14.1** PIC18 Resources Used for Audio Record/Playback Application

PIC18 Resource	Comment
Timer 2	Timer2 interrupt period sets sample rate during playback and record.
ADC (pin AN0)	Used to sample audio signal during record mode. $V_{ref+}$ is $V_{dd}$ ; $V_{ref-}$ is $V_{ss}$ .
MSSP Module (I <sup>2</sup> C Master mode), SDA/SCL	Used to communicate with the 24LC515 serial EEPROM during record and playback, and with the MAX517 DAC pins during playback.
USART	Asynchronous serial port used for PIC18 programming via a bootloader. During application execution, the serial port is connected to a terminal application on the PC and is used to prompt the user for operating mode choice.

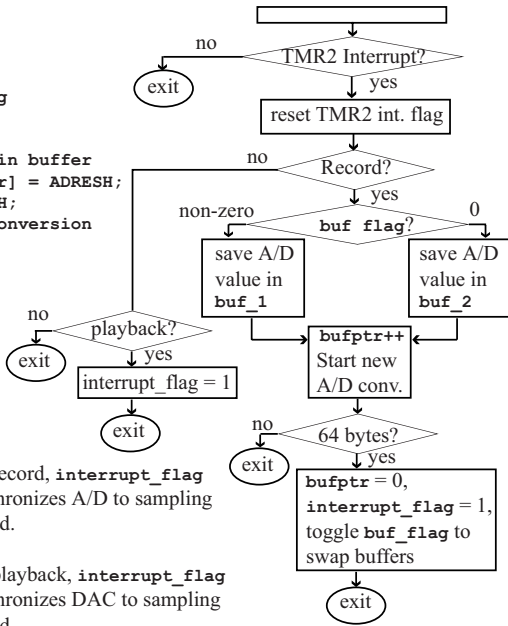
### Audio Application ISR and Configuration

Figure 14.3 shows the ISR code for the audio record/playback application. The ISR is triggered by a periodic Timer2 interrupt with an interval that is equal to the sampling period. During playback (p1ayback\_mode variable is nonzero), the only function of the ISR is to set the interrupt\_flag semaphore indicating that an interrupt has occurred.

```
volatile unsigned char bufptr, interrupt_flag;
volatile unsigned char buf_flag;
volatile unsigned char record_mode, playback_mode;
bdata volatile unsigned char buf_1[64], buf_2[64];

void interrupt
timer_isr(void) {

if (TMR2IF) {
//clear timer interrupt flag
TMR2IF=0;
if (record_mode) {
//read A/D register, save in buffer
if (!buf_flag) buf_2[bufptr] = ADRESH;
else buf_1[bufptr] = ADRESH;
GODONE = 1; // start new conversion
bufptr++;
if (bufptr == 64) {
bufptr = 0;
interrupt_flag = 1;
// toggle buffer flag
buf_flag = ~buf_flag;
}
}
if (playback_mode) {
interrupt_flag = 1;
}
} // end if (TMR2IF)
} // end timer_isr
```



**FIGURE 14.3** ISR for audio record/playback application (see CD-ROM file ./code/chap14/F\_14\_4\_audio.c).

On each Timer2 interrupt during audio recording (record\_mode variable is nonzero), the upper 8 bits of the ADC is stored in either the buf\_1 or buf\_2 buffer as determined by the buf\_flag variable, and a new A/D conversion is started. Once 64 bytes have been sampled, the interrupt\_flag semaphore is set to notify the foreground code that a buffer is full and that a write to serial EEPROM is required. The buf\_flag is toggled to swap the buffers so that an empty buffer is used for storing sampled audio data while the full buffer is written to EEPROM.

Figure 14.4 gives the C code for main() of the audio record/playback application. Pin RB4 is configured as an output initially low (MAX 517 is not driving the mini-jack), the I<sup>2</sup>C interface for a bus speed of approximately 400 kHz, and Timer2 for an interrupt rate of 6 kHz assuming FOSC = 29.4912 MHz.

```

#define EEPROM 0xA0 // I2C EEPROM, write lower blk
#define EEPROMR 0xA1 // I2C EEPROM, read lower blk
#define EEPROMW_HB 0xA8 // I2C EEPROM, write upper blk
#define EEPROMR_HB 0xA9 // I2C EEPROM, read upper blk
#define DAC 0x58 // I2C DAC 01011000
    } I2C address/cmd
    } defines for
    } EEPROM and DAC

main() {
    RB4 as output for playback/record
    TRISB4 = 0; RB4=0; // RB4 output, low } control on analog switch
    //ADC clk = Fsoc/32, channel 0, right justify } ADC config
    ADCON0 = 0x80; ADCON1 = 0x0E; ADON = 1;
    serial_init(95,1); // 19200 in HSPLL mode, crystal = 7.3728 MHz
    if (!TO || !RI) {
        if (!RI) { RI=1; printf("Software reset has occurred, press reset.\n"); }
        else { //watchdog timeout, disable timer
            SWDTEN=0; printf("Watchdog timer reset has occurred, press reset.\n"); }
        pcrflf();
        if (i2c_errstat) i2c_print_err();
        asm("sleep");
    }
    pcrflf(); SWDTEN = 1; // enable watchdog timer
    // enable I2C, about 400kHz, if HSPLL mode, crys. = 7.3728 MHz } I2C config
    i2c_init(17);
    // config timer 2, post scale of 1, prescale of 16, PR2 =76
    TOUTPS3 = 0; TOUTPS2 = 0; TOUTPS1 = 0; TOUTPS0 = 0; } Timer2 periodic interrupt
    T2CKPS1 = 1; // pre scale of 16 } at 6 kHz
    PR2 = 76; // 6 kHz
    printf("Enter 'r'(record), 'p'(playback), 'c'(calibrate), 'e'(examine): ");
    inchar=getch();
    if (inchar == 'c') {
        while(1) {
            GODONE = 1;
            while (GODONE); // wait for end of conversion
            adc_value = ADRESH; // upper 8-bits
            printf("%x",adc_value); pcrflf();
        }
        } Calibration test, continually
        } read A/D value and print to
        } console to test if audio
        } input is working

    if (inchar == 'e') {
        addr = 0;
        while(1) {
            i2c_memread(EEPROM,addr,buf_1); // do read
            for(i = 0;i< 64;i++) printf("%x ",buf_1[i]); pcrflf();
            printf("Any key continues read...");pcrflf();
            getch();
            addr = addr+64;
        }
        } Examine EEPROM
        } contents, use this to
        } verify that data is being
        } stored to EEPROM

    if (inchar == 'r') do_record(); // Record Audio
    if (inchar == 'p') do_playback(); // Playback Audio
    SWDTEN = 0; // disable watchdog timer
    asm("sleep");
} // end main

```



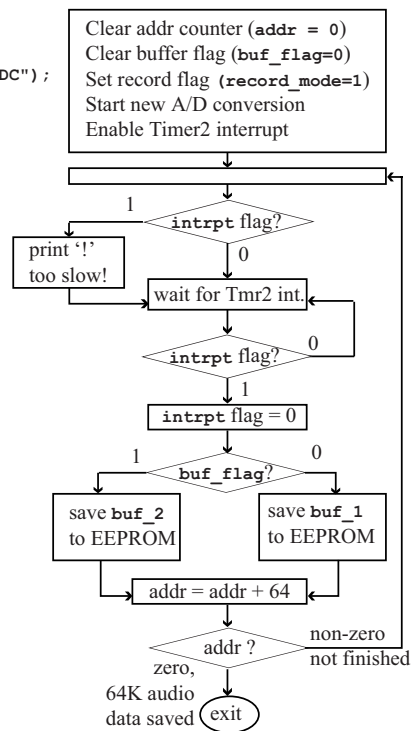
**FIGURE 14.4** main() for audio record/playback application.

The user is prompted for one of four operating modes: record, playback, calibration, or examine. Calibration and examine are debug modes. The calibration mode continually reads the upper 8 bits of the ADC and prints the result to the console; this should be used while the audio input is applied to determine if the ADC sampled values are demonstrating enough swing about the 0x80 midpoint. If the ADC sampled values are only slightly varying about the 0x80 midpoint, the PC volume control should be adjusted upward to provide more amplitude swing for the audio signal. The examine mode reads the contents of the EEPROM and prints the results in hex to the console. This mode is useful to determine if values are actually being stored to the serial EEPROM during record mode.

### Record Mode

Figure 14.5 gives the `do_record()` function of the audio record/playback application. Initialization code sets the `record_mode` variable, starts an A/D conversion, enables the Timer2 interrupt, and turns on Timer2.

```
do_record() {
  addr = 0; buf_flag = 0;
  printf ();
  printf("Capturing 64K of audio from ADC");
  printf ();
  record_mode = 1; //
  GODONE = 1; // start new conversion
  // enable TMR2 interrupt
  TMR2IF = 0; TMR2IE = 1;
  IPEN = 0; PEIE = 1; GIE = 1;
  TMR2ON = 1; // start timer 2 //
  do {
    if (interrupt_flag) putchar('!');
    while (!interrupt_flag) {
      asm("clrwdt");
    }; // wait for block write
    // do block write //
    interrupt_flag = 0;
    if (buf_flag)
      i2c_memwrite(EEPROM,addr,buf_2);
    else
      i2c_memwrite(EEPROM,addr,buf_1);
    addr = addr + 64;
    putchar('*');
    // exit when addr wraps to zero
  }while(addr);
  printf("64K Capture complete. ");
  printf("Press reset to continue.");
  printf ();
}
```



**FIGURE 14.5** `do_record()` for audio record/playback application (see CD-ROM file `./code/chap14/F_14_4_audio.c`).

The `addr` variable is used to track the address for the EEPROM writes; this is initialized to zero. The `buf_flag` variable, which specifies the buffer used by the ISR for storing sampled audio data, is also cleared to zero. The `do-while()` loop waits for the `interrupt_flag` semaphore to be set by the ISR indicating that a buffer is full. The `interrupt_flag` semaphore is then cleared and the full buffer written to EEPROM using the `i2c_memwrite()` function previously discussed in Chapter 11. The `addr` variable is incremented by the page size of 64 and the loop continues while the `addr` variable is nonzero. Because the `addr` variable is an unsigned `int`, it is a 16-bit value and wraps to `0x0000` once 1024 pages ( $64 \text{ Kbytes} = 2^{16} = 64 \times 1024$ ) are written, which is the capacity of the EEPROM. At the top of the `do-while()` loop, the `interrupt_flag` is checked to see if it is nonzero. If this occurs, this indicates that the ISR has already filled another buffer and that the write operation to the serial EEPROM is not keeping up with the sampling rate, causing a “!” character to be written to the serial port as an error indicator.

## Playback Mode

Figure 14.6 shows the `do_playback()` function for the audio record/playback application. Initialization code initializes the `addr` variable used to track the address for the EEPROM writes to zero, the `record_mode` variable to nonzero, and the RB4 output to high to allow the MAX 517 DAC output to drive the mini-jack.

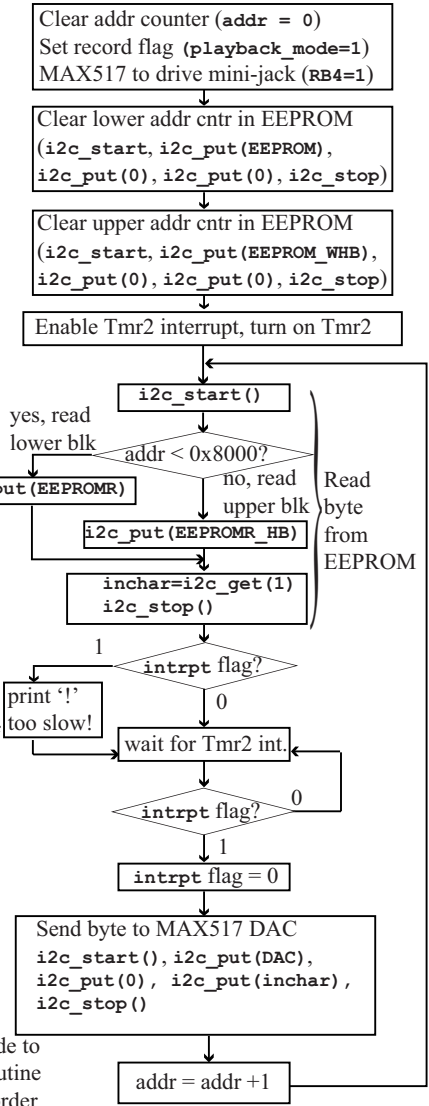
The EEPROM is read in a sequential manner, so the two address counters for the low and high blocks of the 24LC515 are initialized to zero by sending the write command with a value of zero, and then aborting the operation. The block select bit is set in the second write command to select the upper block. The DAC output is initialized to zero, Timer2 interrupts enabled, and Timer2 turned on before the playback loop is entered. The `do-while(1){}` playback loop is an infinite loop that reads a byte from the EEPROM via a sequential read and then sends this byte to the MAX 517 DAC. When reading the byte from the EEPROM, the `addr` variable is checked to see if the read command should be from the lower block (`addr < 0x8000`) or upper block (`addr >= 0x8000`). The byte read from the EEPROM is not written to the MAX 517 DAC until the `interrupt_flag` semaphore is set by the ISR, indicating that a sample period has elapsed. After the byte is written to the DAC, the `addr` variable is incremented before returning to the top of the loop. As in the `do_record()` function, the `interrupt_flag` variable is checked to see if it is set before the synchronizing `while(!interrupt_flag)` wait loop; if it is set prior to reaching the synchronizing wait, this means that the loop is not keeping pace with the sample rate and a “!” character is printed as an error condition.

Note the dotted box in Figure 14.6. You will probably find it necessary to optimize this code by flattening the subroutine calls to remove the `call/return` and parameter passing overhead to reach a 6 kHz (or beyond!) playback rate. Flattening

a subroutine call means to copy the code that implements the subroutine directly into the loop and remove any parameter passing. This is a good example for illustrating when optimized code structuring is required to meet a performance target.

```
do_playback() {
// playback approach
// step thru EEPROM sequentially
addr = 0;
playback_mode = 1;
RB4 = 1; // set switch for playback
pcrlf (); printf("Doing Playback");
pcrlf ();
// clear both internal addr cntrs.
i2c_start();
i2c_put(EEPROM); // write cmd, low blk
i2c_put(0); // send high address byte
i2c_put(0); // send low address byte
i2c_stop(); // send stop
// clear upper block address counter
i2c_start();
i2c_put(EEPROMW_HB); //block select = 1
i2c_put(0); // send high address byte
i2c_put(0); // send low address byte
i2c_stop(); // send stop
// initialize DAC to output zero
i2c_start(); i2c_put(DAC);
i2c_put(0x00); i2c_put(0x00);
i2c_stop();
// enable interrupts
IPEN = 0; TMR2IF = 0; TMR2IE = 1;
PEIE = 1; GIE = 1;
TMR2ON = 1 ; // start tmr2
interrupt_flag = 0; //clear semaphore
do_{
-----
i2c_start();
// read byte
if (addr & 0x8000) i2c_put(EEPROMR_HB);
else i2c_put(EEPROMR);
inchar=i2c_get(1); //NAK
i2c_stop();
// if int flag is set,
//we not keeping up with sample rate!!!
if (interrupt_flag) putchar('!');
// wait for interrupt
while (!interrupt_flag);
interrupt_flag = 0;
i2c_start(); // send to DAC
i2c_put(DAC);
i2c_put(0x00);
i2c_put(inchar);
i2c_stop();
addr = addr+1;
asm{"cli";};
}while(1); //continual loop
}
```

flatten this code to remove subroutine overhead, in order to improve efficiency for a higher sampling rate.

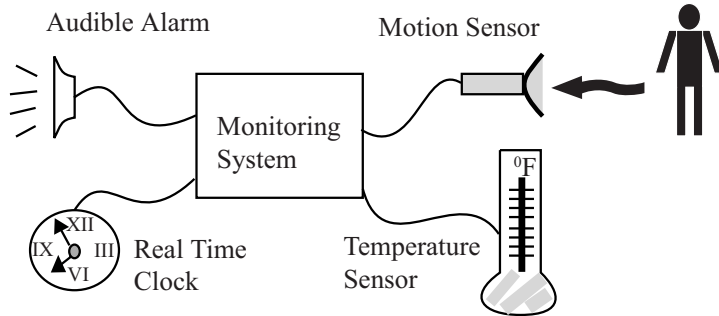


**FIGURE 14.6** do\_playback() for audio record/playback application (see CD-ROM file ./code/chap14/F\_14\_4\_audio.c).



## 14.4 DESIGN OF A HOME MONITORING SYSTEM

The second project of this chapter is a PIC18-based monitoring system whose basic concept is shown in Figure 14.7.



**FIGURE 14.7** Monitoring system concept.

System features are:

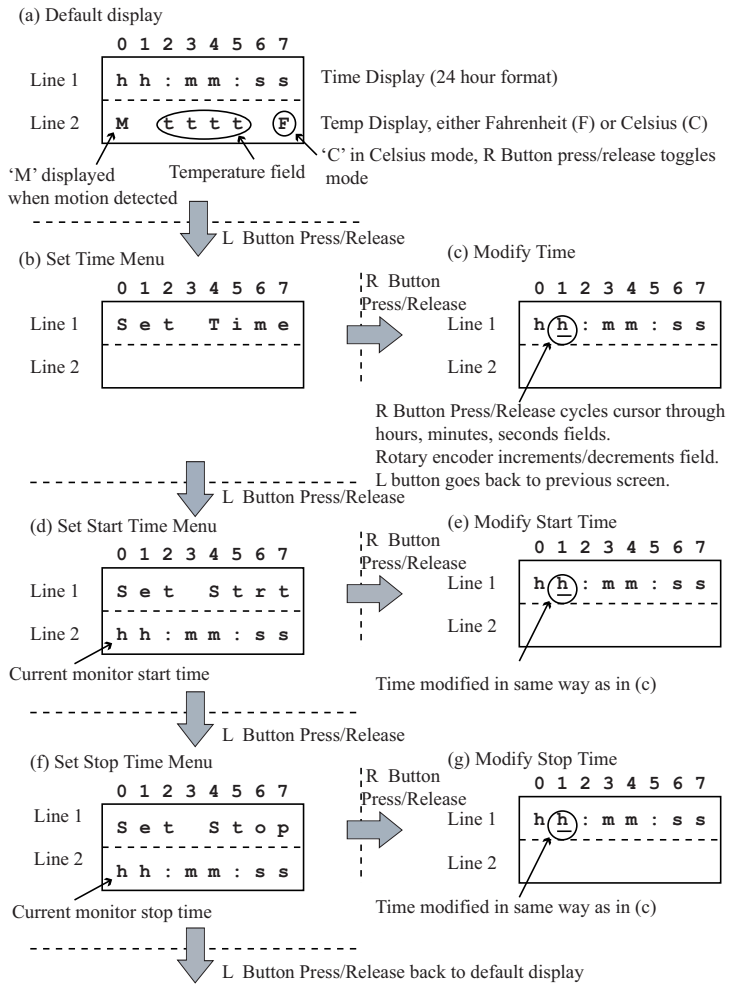
- 6 V to 9 V battery operation
- Real-time clock
- Temperature sensor
- Motion Sensor
- Audible alarm when motion sensor is enabled and motion detected
- LCD display for clock, temperature, status information
- Nonvolatile storage of alarm enable/disable times

A schematic of the monitoring system is shown in Figure 14.8. The real-time clock is implemented using an external 32.768 kHz clock source and Timer1 as discussed in Chapter 13, “Timers.”

A passive infrared motion detector [20] from HVW Technologies outputs a high true pulse that is approximately one second in duration when motion is detected; this is connected to input RB0 that is configured to generate an interrupt on a rising edge input. The DS1621 Digital Thermometer [21] is used for temperature sensing; this is discussed in detail in the next section. A MAX 667 voltage regulator [22] is used instead of the 7805 found on the PIC18 reference board as an example of an alternate voltage regulator IC. The MAX 667 has a shutdown input (SHDN) that can be used to turn off the +5 V output, but this capability is not required in this design and the SHDN pin is grounded for normal operation (see problem #7 at the end of this chapter for a push-to-turn-on application of the SHDN input).



ify the selected time. Each of these screens functions in the same manner; an *R* button press/release cycles the cursor through the hour, minutes, and seconds fields. In each field, a rotary encoder input is used to increment or decrement the selected field value. From any of the time modification screens (c, e, or g), an *L* button press/release returns to the previous screen.



**FIGURE 14.9** LCD screen formats.

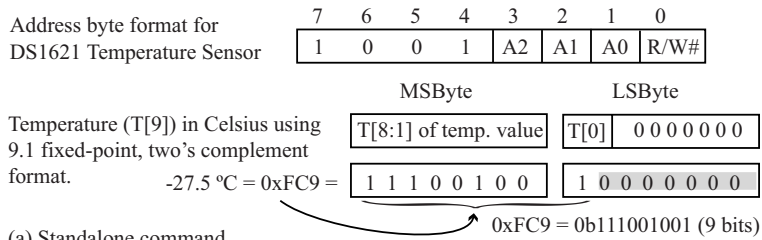
## 14.5 THE DS1621 DIGITAL THERMOMETER

The DS1621 Digital Thermometer has an I<sup>2</sup>C interface and produces a 9-bit temperature value in Celsius using a 9.1 fixed-point, two's complement format (the least significant bit provides half-degree precision, see Chapter 7, "Advanced Assembly Language: Higher Math," for a discussion of fixed-point integers). The temperature range that can be measured by the DS1621 is +125 °C to −55 °C. Pins A2, A1, and A0 on the DS1621 customize the I<sup>2</sup>C address byte as is typical with I<sup>2</sup>C devices. The DS1621 contains two 9-bit internal registers, TH and TL, which can be loaded with temperature values for thermostat control. The TOUT pin is a thermostat output that is asserted when the temperature exceeds or equals TH, and is reset when the temperature falls below or equals TL. This functionality is not used in this application and thus the TOUT pin is left unconnected in Figure 14.8.

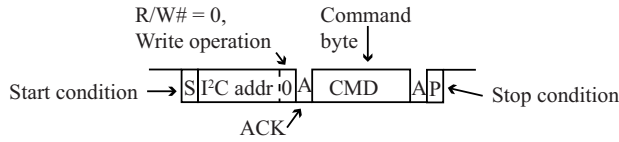
Figure 14.10 shows the I<sup>2</sup>C transactions for the DS1621. Write transactions send an address byte, command byte, and either zero (Figure 14.10a), one (Figure 14.10b), or two (Figure 14.10c) data bytes. No data bytes are sent for standalone commands, while one data byte is sent for an 8-bit write and two data bytes for a 16-bit write. An example of a standalone command is the *Start Convert* command (command byte = 0xEE) that starts a temperature conversion. An 8-bit write transaction is used to modify the configuration register, discussed later in this section. The TH and TL registers are written via a 16-bit write, which sends the most significant byte first, followed by the least significant byte. The two bytes specifying the 9-bit temperature value are *left* justified as shown in Figure 14.10. The MSByte contains bits T[8:1] of the 9-bit temperature value, while the LSByte bit 7 specifies bit T[0] of the temperature with the remaining bits cleared. The example temperature value in Figure 14.10 of −27.5°C is converted to its 9.1 fixed-point, two's complement format by converting 27.5 to its binary value and subtracting from zero as shown in Equation 14.15 (the 9-bit value 0b111001001 is sign extended as 0xFC9 in hex).

$$-27.5 = 0 - (27.5) = 0b00000000.0 - 0b00011011.1 = 0b111001001 = 0xFC9 \quad (14.15)$$

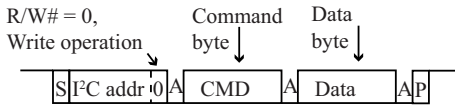
Read transactions start with a write transaction that sends the address and command byte. The write transaction is terminated by a repeated start condition followed by the address byte with the R/W# bit high. The DS1621 then transfers to the I<sup>2</sup>C bus master either one (Figure 14.10d) data byte for an 8-bit register read, or two (Figure 14.10e) data bytes for a 16-bit register read. The I<sup>2</sup>C bus master halts the read by providing a NAK after the last data byte followed by a stop condition. An example of an 8-bit read transaction is *Access Config* (command byte = 0xAC), which returns the value of the configuration register. A 16-bit read transaction is



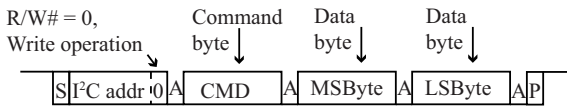
(a) Standalone command



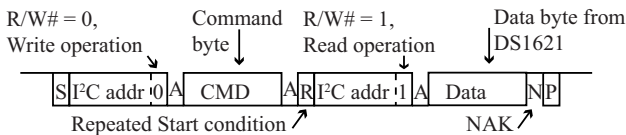
(b) 8-bit Write



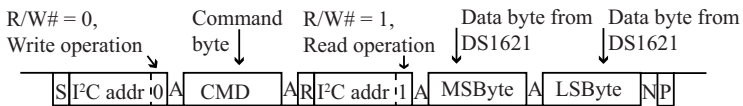
(c) 16-bit Write



(d) 8-bit Read



(e) 16-bit Read



**FIGURE 14.10** DS1621 I<sup>2</sup>C command format.

used to read the contents of registers that are greater than 8 bits in length; the MS-Byte is transferred first followed by the LSByte.

Figure 14.11 shows bit definitions for the DS1621 configuration register. The DONE flag is used for polling the DS1621 to determine if a temperature conversion is in progress.

	7	6	5	4	3	2	1	0
CONFIG Register	DONE	THF	TLF	NVB	1	0	POL	1SHOT
DONE	Conversion Done Flag: "1" when conversion is complete, "0" when conversion is in progress.							
THF	Temperature High Flag: "1" when temperature is greater than or equal to value in TH register; reset on power down or write to CONFIG register.							
TLF	Temperature Low Flag: "1" when less than or equal to the value in the TL register; reset on power down or write to CONFIG register.							
NVB	Nonvolatile Memory Busy flag: "1" when write to nonvolatile memory is in progress, "0" otherwise.							
POL	Polarity bit: "1" TOUT is active high, "0" TOUT is active low.							
1SHOT	One Shot Mode: "1" DS1621 only performs conversions upon receiving a Start Conversion command; "0" the DS1621 performs continuous conversions.							

**FIGURE 14.11** DS1621 configuration register.

The THF (Temperature High Flag) and TLF (Temperature Low Flag) bits are set when the temperature exceeds or is equal to the values stored in the TH and TL registers, respectively. The TH and TL register values are stored in on-chip non-volatile memory, and the NVB (Nonvolatile Memory Busy Flag) bit can be polled to determine if a write to nonvolatile memory is in progress. A write to nonvolatile memory on the DS1621 can take up to 10 ms. The POL (Output polarity) bit controls the polarity of the TOUT output. The 1SHOT mode bit controls whether the DS1621 continuously performs conversions (1SHOT = "1") or only performs a conversion upon receiving a Start Convert command.

Table 14.2 gives the command set of the DS1621. The Read Slope and Read Counter commands read the contents of internal registers used for temperature

**TABLE 14.2** DS1621 Command Set

Command	CMD Byte	Comment
Read Temperature	0xAA	16-bit read returns the temperature value
Start Convert	0xEE	Standalone command to start a conversion
Stop Convert	0x22	Standalone command to halt conversion in continuous mode
Access TH	0xA1	16-bit read or 16-bit write to TH
Access TL	0xA2	16-bit read or 16-bit write to TL
Access Config	0xAC	8-bit read or 8-bit write to CONFIG
Read Counter	0xA8	8-bit read returns the counter register value
Read Slope	0xA9	8-bit read returns the read slope register value

conversion; these commands are typically not needed in normal operation and the reader is referred to the datasheet [21] for details on their functionality.

Figure 14.12 shows functions that implement a subset of the DS1621 I<sup>2</sup>C transactions. The `ds1621_send0()` and `ds1621_send1()` functions perform the standalone command and 8-bit write transactions, respectively.

```

#define TEMPSENSE      0x90
#define ACCESS_CONFIG  0xAC
#define START_CONVERT  0xEE
#define READ_TEMP      0xAA
    } DS1621 Command subset used by
    } monitoring application

void ds1621_send0(unsigned char cmd){
    i2c_start();
    i2c_put(TEMPSENSE);
    i2c_put(cmd);
    i2c_stop();
    } } Standalone command write transaction
    } (no data byte)

void ds1621_send1(unsigned char cmd,unsigned char data){
    i2c_start();
    i2c_put(TEMPSENSE);
    i2c_put(cmd);
    i2c_put(data);
    i2c_stop();
    } } 8-bit write transaction

unsigned char ds1621_read1(unsigned char cmd){
    unsigned char c;
    i2c_start();
    i2c_put(TEMPSENSE);
    i2c_put(cmd);
    i2c_rstart();
    i2c_put(TEMPSENSE | 0x01);
    c = i2c_get(1); // NAK read
    i2c_stop();
    return(c);
    } } 8-bit read transaction

void ds1621_read2(unsigned char cmd, unsigned char *ptr){
    i2c_start();
    i2c_put(TEMPSENSE);
    i2c_put(cmd);
    i2c_rstart();
    i2c_put(TEMPSENSE | 0x01);
    *(ptr+1) = i2c_get(0); // get MSB, ack
    *(ptr) = i2c_get(1); // get LSB, nak
    i2c_stop();
    } } 16-bit read transaction,
    } the *ptr variable is assumed to point
    } to an int variable; the two returned
    } bytes from the DS1621 are stored in
    } the int variable in little endian order.

```



**FIGURE 14.12** C code for DS1621 I<sup>2</sup>C transactions (see CD-ROM file `./code/chap14/temp_module.c`).

The `ds1621_read1()` and `ds1621_read2()` functions perform the 8-bit read and 16-bit read transactions, respectively. The `char *ptr` parameter of the `ds1621_read2()` function is assumed to point to an `int` variable; the two returned

bytes from the DS1621 are stored in little endian order into the memory space referenced by ptr.

A test program that uses the functions of Figure 14.12 for DS1621 temperature conversion is shown in Figure 14.13. The program configures the DS1621 for one shot mode via the statement:

```
ds1621_send1(ACCESS_CONFIG,0x01)
```

```
main(void) {
    unsigned char mode;
    signed int temp2;
    signed int temp_c, temp_f; } Temperature value is a signed quantity. The signed
    qualifier is only used for emphasis, it is the
    default qualifier for type int

    serial_init(95,1); // 19200 in HSPLL mode, crystal = 7.3728 MHz
    if (!RI) {
        RI = 1;
        printf("Software reset!");pcrlf();
        if (i2c_errstat) i2c_print_err();
    }
    if (!TO) {
        printf("Watchdog timer reset has occurred.\n"); pcrlf();
        if (i2c_errstat) i2c_print_err();
    }
    i2c_init(73); // ~100 kHz @ FOSC = 29.4912 MHz
    pcrlf(); printf("Temp Sensor Test Started"); pcrlf();
    SWDTEN = 1; // enable watchdog timer
    // set one shot mode
    ds1621_send1(ACCESS_CONFIG,0x01); ← set one shot mode
    while(1) {
        // start conversion
        ds1621_send0(START_CONVERT); ← start conversion
        //wait for end
        do{
            mode = ds1621_read1(ACCESS_CONFIG);
        } while(!bittst(mode,7)); } Wait for conversion to end by
        polling DONE bit in config register
        ds1621_read2(READ_TEMP, (char *)&temp2); ← Read temperature value
        temp_c = (temp2 >> 8);
        temp_f = (temp_c*9)/5 + 32; } ← Convert left justified 9-bit value to
        8-bit integer (drop fractional part), and
        convert to Fahrenheit

        pcrlf();
        printf("Temp read: %x (16 bits), %d (C), %d (F)",
            temp2,temp_c,temp_f); } Print hex temperature as
        // wait for key input for next conv. 16-bit value and also as
        getch(); decimal Celsius and
        Fahrenheit
    }
    Temp Sensor Test Started
    Temp read: 1B00 <16 bits>, 27 <C>, 80 <F> ← Initial reading
    Temp read: 1C80 <16 bits>, 28 <C>, 82 <F> ← Touch with finger
    Temp read: 1980 <16 bits>, 25 <C>, 77 <F>
    Temp read: 1900 <16 bits>, 25 <C>, 77 <F> ← Applied cold source
    Temp read: 1880 <16 bits>, 24 <C>, 75 <F> ← Applied cold source
    Temp read: 1A00 <16 bits>, 26 <C>, 78 <F> ← Cold source removed,
    Temp read: 1A00 <16 bits>, 26 <C>, 78 <F> recovering to room temp.
```



**FIGURE 14.13** Test program for the DS1621 C functions.

This is an 8-bit write transaction that sets the 1SHOT bit of the CONFIG register. The while(1){} loop then starts a conversion by the standalone command



`ds1621_send0(START_CONVERT)`. A `do-while{}` loop then polls the DONE bit of the CONFIG register via the 8-bit read transaction `ds1621_read1(ACCESS_CONFIG)`; the `do-while{}` loop is exited when the DONE bit returns as “1” indicating a finished conversion. Using a 16-bit read transaction returns the 9-bit temperature value:

```
ds1621_send2(READ_TEMP, (char *) &temp2)
```

Observe that `(char *)&temp2` passes the address of the `temp2` variable as the pointer value required by the `ds1621_send2()` function. This value is then shifted right by 8 positions to convert the 9.1 fixed-point, left justified value to an 8-bit integer representing a signed Celsius temperature stored in `temp_c`. This drops the half-degree fractional portion of the temperature. After converting the Celsius value to a Fahrenheit value stored in `temp_f`, the raw 16-bit temperature value `temp2` is printed in hex along with the integer Celsius and Fahrenheit values. Sample output from the test program is shown in Figure 14.13 at the end of the listing with the temperature returned by the DS1621 varying as heat and cold sources are applied.

## **14.6 USING THE NONVOLATILE STORAGE ON THE PIC18FXX2**

---

The monitoring system has start and stop times for enabling the audible alarm that sounds when motion is detected. These start and stop times should be stored in nonvolatile memory so a user does not have to reenter them each time power is cycled. The 24LC515 I<sup>2</sup>C EEPROM could be used for this nonvolatile storage, but this is inefficient as only a few bytes of storage are required. There are two choices of on-chip nonvolatile storage for the PIC18: Flash program memory and Data EEPROM. The term “Flash memory” or just “Flash” is used from this point on as a shortened reference to Flash program memory. Table 14.3 compares and contrasts these two types of nonvolatile storage.

Both the Flash and Data EEPROM can be written under program control. The Flash memory holds the program code, but any unused space can also be used for data storage. Obviously, the Flash memory is the only choice if the nonvolatile storage requirements exceed the 256 bytes of the Data EEPROM. The Data EEPROM can be written a byte at a time, and performs an erase/write cycle so no explicit erase operation is needed. The Flash memory must be erased before written, with erasure limited to exactly 64 bytes at a time when done under program control. Also, the minimum write size for Flash memory is 8 bytes. The Data EEPROM has been optimized for a high number of writes with a minimum cell endurance of 100K writes, 10X more than the Flash cell endurance. The Data EEPROM has a refresh cycle requirement in that if a cell has not been written to after a total of 1 M writes to other locations, a refresh cycle has to be done (see the PIC18F242 data

**TABLE 14.3** Flash Program Memory versus Data EEPROM on the PIC18Fxx2

Specification	Flash Program Memory	Data EEPROM
Size (bytes)	8192 (18F242/442) 16384 (18F252/452)	256
Minimum write size	8 bytes	1 byte
Minimum erase size	64 bytes	N/A, uses erase before write
Cell Endurance	10K writes (minimum)	100K writes (minimum)
Refresh Cycle	N/A	1 M writes total (minimum)
Typical write time	2 ms	4 ms (erase before write)
Address Register/Data Reg	TBLPTR{L/H/U}/ (internal)	EEADR/EEDATA
Control Registers	EECON1/EECON2	EECON1/EECON2
Interrupt Driven Write?	No, CPU halts during write/erase	Yes (EEIF bit)
Typical write time	2 ms	4 ms (erase before write)

sheet for details). For this reason, if you have a mixture of frequently and infrequently updated locations, the infrequently updated data should be stored in the Flash program memory. Both the Flash and data EEPROM memory use the EECON1 and EECON2 registers to perform writes. The EECON2 register is not an actual register and is only used in the write sequence. The bit definitions for the EECON1 register are given in Table 14.4.

The EEPGD (EECON1[7]) bit controls whether the Flash or EEPROM memory is being accessed. The CFGS (EECON1[6]) bit allows programming of the configuration registers (briefly discussed in Chapter 8, “The PIC18Fxx2: System Startup and Parallel Port IO”), which reside in Flash program memory beginning at location 0x300000. See Appendix A, “PIC18Fxx2 Architecture, Instruction Set, Register Summary,” for listing of these registers and bit definitions, and the PIC18F242 datasheet for complete information. The FREE (EECON[4]) bit must be set to a “1” when erasing Flash memory. The WRERR (EECON[3]) is a status bit that can be checked to determine if a write completed normally. The WEN (EECON[2]) bit must be a “1” to enable writes to either Flash or Data EEPROM. Setting the WR (EECON[1]) bit begins a write operation to either the Flash or

Data EEPROM; it is cleared automatically after the write is complete. The RD (EECON[0]) bit is only used with the Data EEPROM and performs a read cycle when set; it is cleared automatically after the read cycle is complete.

**TABLE 14.4** EECON1 Register

Name	Bit	Comment
EEPGD	[7]	"1" to access Flash, "0" to access data EEPROM.
CFGS	[6]	"1" to access configuration memory, "0" for FLASH/EEPROM.
n/a	[5]	Unimplemented.
FREE	[4]	"1" to enable erase during FLASH write, "0" for write only.
WRERR	[3]	"1" if write operation terminated prematurely, "0" write terminated normally.
WREN	[2]	"1" enables write, "0" write protects Flash/EEPROM.
WR	[1]	"1" initiates write operation of Flash/EEPROM, cleared to "0" by hardware on write completion.
RD	[0]	"1" performs a EEPROM read, which takes one cycle and places data in the EEDATA register; RD cleared to "0" automatically afterwards.

## Data EEPROM Read/Write

Figure 14.14 gives C functions for noninterrupt driven Data EEPROM read and write. The `eedata_readbyte()` function reads a byte from Data EEPROM location `addr`. For a read or write, the `EEADR` register contains the Data EEPROM location being accessed. The `RD` bit is set to "1" to perform the read, and then the `EEDATA` register value is returned. The `eedata_writebyte()` function writes `byte` to Data EEPROM location `addr`. The `WREN` (write enable) bit is set to enable writes, interrupts are disabled, the sequence `0x55, 0xAA` is written to `EECON2` followed by setting the `WR` bit to begin the write. This exact sequence must be followed for the write operation to be started. These stringent requirements are intended to reduce the probability of a spurious write to Data EEPROM. Interrupts are re-enabled as the write operation can take up to 4 ms, and interrupt service does not disturb the write operation as long as another write is not attempted until the current write is finished. After triggering the write, the function waits for the `WR` bit to be cleared indicating that the write is complete. Before returning, the `EEIF` flag is cleared (set when write completed) and the `WREN` bit is cleared. The `EEIF` flag can be used to generate an interrupt and the write of several bytes to EEPROM handled on an interrupt-driven basis. In this case, the read function would be changed to check a

semaphore that indicates whether a block of EEPROM data had finished updating or not. The `eedata_writestring()` function uses the `eedata_writebyte()` function to write a string contained in `char *s` to location `addr`. Each byte is verified after write by using `eedata_readbyte()`; if any byte fails to verify the write is terminated and the function returns “1”. If all bytes verify, the function returns a “0”. The `ee_readstr` function reads a string containing a maximum of `max` characters from location `addr` in Data EEPROM into the `char *s` buffer.

```

unsigned char eedata_readbyte(unsigned char addr){
    EEADR = addr;  EEPGD = 0;  CFGS = 0;
    RD = 1;
    return(EEDATA);
}
void eedata_writebyte(unsigned char addr, unsigned char byte){
    char istat;
    EEADR = addr;  EEDATA = byte;
    EEPGD = 0;  CFGS = 0;  WREN = 1;
    istat = GIE;
    GIE = 0;
    EECON2 = 0x55;  EECON2 = 0xAA;
    WR=1;
    GIE = istat;
    // wait for write to complete
    while (WR) ;
    EEIF = 0;  WREN = 0;
}
//Write string to DATA EEPROM
char eedata_writestr(unsigned char *s, unsigned char addr){
    char c;
    while(*s) {
        eedata_writebyte(addr,*s);
        // verify
        c = eedata_readbyte(addr);
        if (c != *s) return(1);
        addr++; s++;
    }
    // write end of string
    eedata_writebyte(addr,*s);
    c = eedata_readbyte(addr);
    if (c != *s) return(1);
    return(0);
}
//Read string from DATA EEPROM
unsigned char eedata_readstr(unsigned char *s,
    unsigned char addr, unsigned char max){
    unsigned char c,cnt;
    cnt = 0;
    do {
        c = eedata_readbyte(addr);  addr++;
        *s = c;
        s++; cnt++;
    }while((c) && (cnt < (max-1)));
    // last byte non-zero, terminate this string
    if (c) *s = 0;
    return(addr);
}
    
```

Read one byte from location `addr` in Data EEPROM.

Write `byte` to location `addr` in data EEPROM. Write is non-interrupt driven; waits for write to complete.

The sequence ‘0x55’, ‘0xAA’ written to `EECON2` is necessary to enable the write.

The `WR` bit is cleared when the write operation is complete. Clear `EEIF` flag, `WREN` before exit.

Write string `*s` to location `addr` in data EEPROM.

Verify each byte after write by reading it back and doing a compare.

Return a ‘0’ on successfully writing all bytes, return a ‘1’ if any byte fails verification.

Read string from location `addr` in Data EEPROM, store in `*s`.

Read at most `max` characters.



**FIGURE 14.14** C functions for Data EEPROM read/write (see CD-ROM file `./code/chap14/ee_module.c`).

Figure 14.15 shows a program that tests the Data EEPROM read/write functions of Figure 14.14. The `while(1){}` loop in `main()` prompts the user for read or write mode; in write mode, a string is read from the serial port and written to Data EEPROM. After each write, the `addr` variable is incremented by the number of bytes written. In read mode, the first string is read from Data EEPROM starting at location 0. After the read, the `addr` variable is set to the new value returned by `ee_data_readstr()`, where `addr` is incremented by the number of bytes read from the EEPROM. Sample console output is given at the end of the `main()` code listing.

## Flash Program Memory Read/Write

Figure 14.16 gives *C* functions for reading a byte and reading a string from Flash memory. The `flash_readbyte()` function reads a byte from location `addr` in Flash memory. The `TBLPTR` register (originally discussed in Chapter 6, “Subroutines and Pointers”) is a 21-bit register used for pointing into Flash program memory. The statement `TBLPTR = (far unsigned char *) addr` copies the `addr` parameter containing the target Flash memory location into the `TBLPTR` register. The `far` qualifier on the `unsigned char *` data type indicates that this pointer is referencing Flash memory space and thus is 3 bytes long. The assignment causes the compiler to generate code that copies the lower 3 bytes of `addr` into the `TBLPTRL`, `TBLPTRH`, `TBLPTRU` registers, respectively (note the little endian order). The inline assembly statement `asm(“TBLRD*+”)` does a table read with post increment, transferring the byte referenced by `TBLPTR` into the `TABLAT` register, which is returned as the byte read from Flash memory. The `flash_readstr()` function reads a string containing a maximum of `max` characters from location `addr` in Flash memory into the `char *s` buffer.

Figure 14.17 shows *C* functions for Flash memory erase and write. The `do_flash_erase()` function erases a block of 64 bytes beginning at location `addr`, where `addr` is assumed to be aligned on a 64-byte boundary (this is not necessary, but unexpected results will occur due to the lower 6 bits of `addr` being ignored during the erase).

The `FREE` bit is set to 1 to indicate that this is an erase operation and not a write operation. The erase operation is started by the assignment `WR = 1`; the CPU stalls for the duration of the erase operation since instructions cannot be fetched when an erase or write to Flash memory is being performed. Observe that interrupts are disabled for the duration of the write by clearing `GIE`. The `flash_writebuf()` function writes 64 bytes contained in `buffer *buf` to location `addr` in Flash memory. An erase operation is performed first using `do_flash_erase(addr)`, and then the 64 bytes are written in groups of eight to Flash memory, as the minimum write operation is 8 bytes. The 8 bytes to be written are stored in an eight-location holding register, which is addressed by the lower 3 bits of the `TBLPTR` register. A

```

unsigned char eedata_readbyte(unsigned char addr){
    unsigned char c,istat;
#define BUFSIZE 64
unsigned char buf[BUFSIZE];
main(void){
    unsigned char c, cnt, addr,mode, *s;
    serial_init(95,1); // 19200 in HSPLL mode, crystal = 7.3728 MHz
    addr = 0;
    pcrLf();
    printf("PIC18 On-board EEPROM R/W Test");
    pcrLf();
    while(1){
        pcrLf();
        printf("Enter 'w' for write mode, else read: ");
        c=getche(); pcrLf();
        if (c != mode) addr=0; //reset address
        mode = c;
        if (mode == 'w') {
            printf("Enter string, %d chars max: ",BUFSIZE);
            pcrLf();
            s = buf; cnt = 0;
            do { // get a string
                c = getche();
                if (c != 0x0D) {
                    *s = c; s++; cnt++;
                }
            }while(c != 0x0D);
            *s = 0; s++;cnt++;
            pcrLf();
            if (eedata_writestr(buf,addr)) {
                printf("Write unsuccessful!"); pcrLf();
            }
            addr = addr + cnt;
        } else {
            // read a string from EEPROM
            addr = eedata_readstr(buf,addr,BUFSIZE);
            printf("String read: %s",buf);
            pcrLf();
        }
    }
}

```

} Prompt user to choose read or write mode.

} Read string from serial port, write to Data EEPROM using ee\_writestr

} Read string from Data EEPROM using ee\_readstr

PIC18 On-board EEPROM R/W Test ← Console output from test

Enter 'w' for write mode, else read: w

Enter string, 256 chars max: This is a test of write to EEPROM! ← First string written.

Enter 'w' for write mode, else read: w

Enter string, 256 chars max: A second string is written. ← Second string written.

Enter 'w' for write mode, else read: r

String read: This is a test of write to EEPROM! ← Both strings read back from Data EEPROM.

Enter 'w' for write mode, else read: r

String read: A second string is written.



**FIGURE 14.15** Test of Data EEPROM read/write.

preincrement table write instruction, `asm("TBLWT+*")`, is used so that when the eighth byte is written, the `TBLPTR` register is still pointing to the same 64-byte boundary in Flash memory as when the first byte was written. The `addr` variable is decremented (`addr--`) before the write loop is entered because of the preincrement table write. After the eighth byte of a group is written, the Flash write operation is performed in the same manner as in `do_flash_erase()` function, except that the

```

unsigned char flash_readbyte(long addr){
    TBLPTR = (far unsigned char *)addr;
    asm("TBLRD*+");
    return(TABLAT);
}

long flash_readstr(unsigned char *s, long addr, unsigned char max){
    unsigned char c,cnt;
    cnt = 0;
    do {
        c = flash_readbyte(addr);
        addr++;
        *s = c;
        s++; cnt++;
    }while((c) && (cnt < (max-1)));
    // last byte non-zero,
    // terminate this string
    if (c) *s = 0;
    return(addr);
}

```

Read one byte from location `addr` in Flash memory.  
 Note the use of the `long` type for `addr`

Read string from location `addr` in Flash EEPROM, store in `*s`.  
 Read at most `max` characters.



**FIGURE 14.16** C functions for Flash memory read (see CD-ROM file `./code/chap14/flash_module.c`).

FREE bit is cleared indicating a write operation. The 64 bytes are verified after the write is completed by using the `flash_readbyte()` function; if any byte fails verification, a “1” is returned; else, a “0” is returned.

Figure 14.18 shows a program that tests the Flash memory read and write functions. The `while(1){}` loop in `main()` prompts the user for either write, test, or read mode. In write mode, a string is captured from the serial port and written to Flash memory at location `0x3F80` using the `flash_writebuf()` function (location `0x3F80` is at the end of the Flash memory on the PIC18F242). In read mode, the data at Flash location `0x3F80` is read using the `flash_readstr()` function and displayed on the console. The test mode reads the `const char test_string[]` from Flash memory using `flash_readstr()` and displays it on the console. The `const` (constant data) modifier for `test_string[]` tells the compiler that this data should reside in program memory and not be copied to the file register memory. Sample console output is given at the end of the `main()` code listing.

```

do_flash_erase(long addr){
    char istat;
    TBLPTR = (far unsigned char *)addr;
    EEPGD=1;FREE=1;WREN=1;CFGs=0;
    istat = GIE; GIE = 0;
    EECON2 = 0x55; EECON2 = 0xAA;
    WR = 1; // CPU stalls
    asm("NOP");
    GIE = istat; FREE=0;WREN=0;
}

// write 64 bytes , assumes 'addr' is on 64 byte boundary!
char flash_writebuf(long addr, char *buf){
    char i, j,istat, *s;

    do_flash_erase(addr);
    // before write, decrement address
    addr--;
    TBLPTR = (far unsigned char *)addr;
    i = 0; j= 0; s = buf;
    while (i < 64) {
        TABLAT=*s;
        asm("TBLWT+");
        j++;i++;s++;
        if (j == 8) {
            // 8 bytes written to internal buff, do write
            EEPGD=1;FREE=0;WREN=1;CFGs=0;
            istat = GIE;
            GIE = 0;
            EECON2 = 0x55; EECON2 = 0xAA;
            WR = 1; // CPU stalls
            asm("NOP");
            GIE = istat;
            WREN = 0;
            j = 0;
        }
    }
    // verify contents written correctly
    i = 0; s = buf;
    addr++; // get back to start
    while(i < 64) {
        j = flash_readbyte(addr);
        if (j != *s) return(1);
        i++;s++;addr++;
    }
    return(0);
}

```

Do erase operation in Flash memory at location `addr` which is assume to start on 64-byte boundary.

Write 64 bytes in `*buf` to location `addr` in Flash

Do erase first

Decrement address before start so that first table write triggers new 8-byte row boundary in holding register.

Write 64 bytes in eight groups of eight bytes.

Each group of 8 bytes fills up the internal holding registers, which then must be written using a Flash write operation.

Verify that 64 bytes were correctly written by reading back the 64 bytes and comparing against `*buf` contents.

Return "1" if any byte fails to verify.

Return "0" on successful verification.



**FIGURE 14.17** C functions for Flash memory erase and write (see CD-ROM file `./code/chap14/flash_module.c`).



```

#define BUFSIZE 64
unsigned char buf[BUFSIZE];
#define FLASH_BUF 0x3F80 // flash address ← Target address in Flash to store data
const char test_string[]="Test of 'const' string in program memory";
long flash_ptr;

main(void){
  unsigned char c, mode, *s;
  serial_init(95,1); // 19200 in HSPLL mode, crystal = 7.3728 MHz
  pcrf(); printf("PIC18 On-board FLASH R/W Test"); pcrf();
  flash_ptr = FLASH_BUF;
  while(1){
    pcrf(); printf("'w' (write), 't' (read const. string), 'r' (read): ");
    c=getche(); pcrf(); mode = c;
    if (mode == 'w') {
      flash_ptr = FLASH_BUF;
      printf("Enter string, %d chars max: ",BUFSIZE);pcrf();
      s = buf;
      do { // get a string
        c = getche();
        if (c != 0x0D) {*s = c; s++;}
      }while(c != 0x0D);
      *s = 0; s++; pcrf();
      if (flash_writebuf(flash_ptr,buf)) {
        printf("Write unsuccessful!"); pcrf();
      }
    }
    else if (mode == 't' || mode == 'r'){
      if (mode == 't') flash_ptr = (long) &test_string[0];
      else flash_ptr = FLASH_BUF;
      // read from flash memory
      flash_readstr(buf,flash_ptr,BUFSIZE);
      printf("String read: %s",buf);
      pcrf();
    }
  }
}

```

Prompt user for mode

Read string from serial port, write to Flash memory using flash\_writebuf

Use either FLASH\_BUF or test\_string[] as starting address for flash read

Read from Flash using flash\_readstr, print to console

```

PIC18 On-board FLASH R/W Test
'w' <write>,'t' <read const. string>, 'r'<read>: t
String read: Test of 'const' string in program memory
'w' <write>,'t' <read const. string>, 'r'<read>: w
Enter string, 64 chars max:
Hello, how are you?
'w' <write>,'t' <read const. string>, 'r'<read>: r
String read: Hello, how are you?
'w' <write>,'t' <read const. string>, 'r'<read>: w
Enter string, 64 chars max:
I am fine, thanks!
'w' <write>,'t' <read const. string>, 'r'<read>: r
String read: I am fine, thanks!

```



FIGURE 14.18 Test of Flash memory read and write.

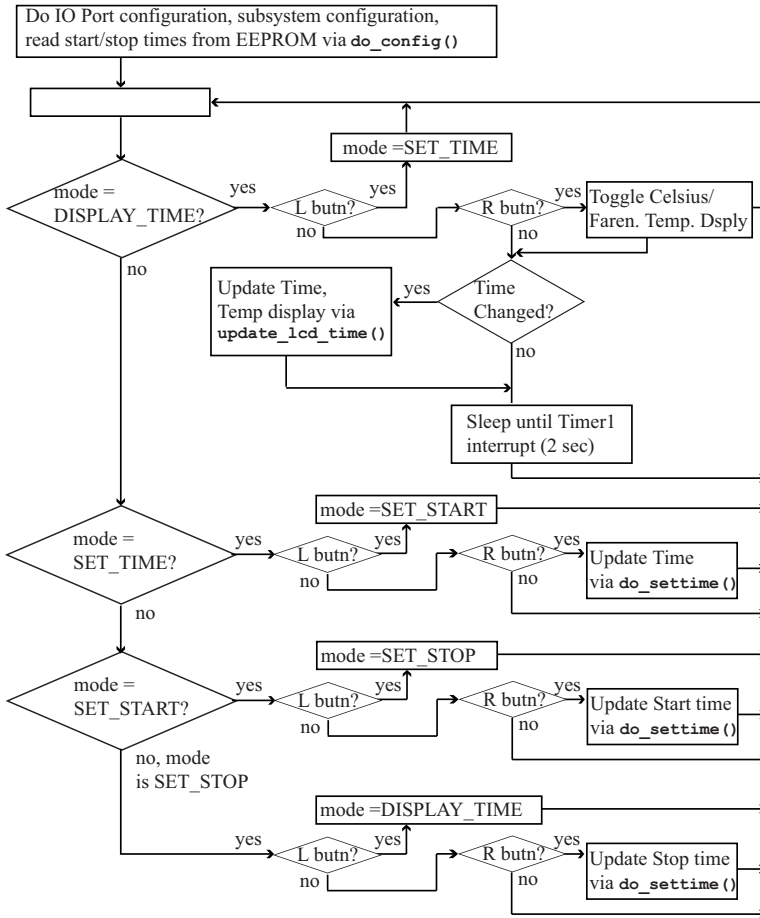
## 14.7 IMPLEMENTATION OF A HOME MONITORING SYSTEM

At this point, the mechanisms for implementing the temperature sensing and non-volatile data storage capabilities required by the home monitoring system have been discussed. We can now move to the next step, which is the code implementation. Table 14.5 shows the PIC18F242 resources used by the home monitoring application. Most of the PIC18 subsystems are involved except for Timer0 and the A/D converter.

**TABLE 14.5** PIC18 Resources Used for Home Monitoring Application

PIC18 Resource	Comment
RB2	Left pushbutton input, falling edge triggered interrupt.
RB1	Right pushbutton input, falling edge triggered interrupt.
RB0	Motion sensor input, rising edge triggered interrupt.
RB7, RB6	Rotary encoder input, periodically sampled.
RA[3:0], RA4, RB[5:4]	LCD interface.
Timer1	Used to implement time-of-day clock, T13CKI input driven by external 32.768 kHz clock. Generates an interrupt every two seconds on Timer1 rollover.
Timer3	Used for debouncing pushbutton inputs and sampling rotary encoder inputs, configured to generate a periodic interrupt of approximately 9 ms.
Timer2/PWM/CCP1 output	Timer2 and PWM used for generating square wave that drives an external speaker for the audible alarm.
MSSP Module (I <sup>2</sup> C Master mode), pins SDA/SCL	Used to communicate with the DS1621 Digital Thermometer.
Data EEPROM	Used to store Start and Stop monitoring times.
USART	Asynchronous serial port used for PIC18 programming via a bootloader. Not used during the application execution.

The home monitoring system is the most complex application presented in this book and is covered in a top-down manner starting with the `main()` code flowchart in Figure 14.19. This implements the four top-level displays shown in Figure 14.9a, b, d, and f.



**FIGURE 14.19** main() flowchart for home monitoring application.

The left button cycles through the four top-level modes of *display time*, *set time*, *set alarm start time*, and *set alarm stop time*. The primary mode is the display time mode, and in this loop the system is woken every two seconds by Timer1 wrapping from 0xFFFF to 0x0000, at which point the display is updated with the current time and temperature. The system is also woken on any pushbutton or motion sensor activity. The other three modes are used for altering the time of day, alarm start time, or alarm stop time via the pushbuttons and rotary encoder.

Figure 14.20 gives a summary of the primary variables and functions in the home monitoring application. Previously covered functions such as those used for the LCD or DS1621 are not listed. The user should refer to Figure 14.20 when reading code that follows in the remainder of this section.

Variable(s)	Comment
left_button, right_button	semaphores for left, right button pushes
left_debounce, right_debounce	Left, Right button debounce counters
motion	semaphore for motion detection
hour, min, sec	time-of-day in hours, minutes, seconds
cur_time	time-of-day in seconds
hour1, min1, sec1	alarm start time in hours, minutes, seconds
hour2, min2, sec2	alarm stop time in hours, minutes, seconds
alarm_start, alarm_stop	alarm start/stop time in seconds
t_hour, t_min, t_sec	variables adjusted by do_settime() function
raw_temp, temp_c, temp_f	16-bit raw temperature, Celsius temperature, Fahrenheit temperature
temp_mode	when "0" display Fahrenheit, when "1" display Celsius
int0_last, int1_last	last sampled values of rotary encoder inputs
int0_cnt, int1_cnt	debounce counters for rotary encoder inputs
rotcount, last_rotcount	Counter values updated when rotary encoder changes in update_rotary_state()
rotmin, rotmax	rotcount is clipped to these values in update_rotary_state()
state, last_state	tracks state of rotary encoder
rot_dir	indicates direction of rotary encoder turn
mode	current display mode in primary IO loop of main().
Functions	Comment
do_config()	Do IO Port, Subsystem configuration
update_lcd_time()	Update LCD time/temperature display
do_settime()	Perform user modification of either time-of-day, alarm start time, or alarm stop time
update_rotary_encoder()	Read rotary encoder inputs and update state
do_debounce()	Called by Timer3 ISR to debounce pushbutton and rotary encoder inputs
read_start_stop()	Read alarm start/stop times from EEPROM
save_start_stop()	Save alarm start/stop time to EEPROM
new_choice()	Display new menu choice on LCD
update_lcd_time1()	Write a time value to LCD on specified line

**FIGURE 14.20** Variables and subroutines summary of the home monitoring application.

Figure 14.21 shows a portion of `main()` for the home monitoring application; the `DISPLAY_TIME` mode contained in the top level `while(1){}` loop is shown in its entirety. The `left_button` and `right_button` variables are semaphores that are set by the ISR if these buttons are pushed. Within the `while(mode==DISPLAY_TIME){}` loop, any button activity cancels an audible alarm if it's sounding by setting the PWM duty cycle to zero (`CCPR1L = 0x00`). Also, the `alarm_count` variable is zeroed; this is used to implement a delay of `ALARM_DELAY` seconds before the audible alarm is re-activated once a button has been pushed. If the time has changed (`old_sec != sec`), the LCD display is updated via the `update_lcd_time()` function. The `motion` variable is a semaphore that is set by the ISR if motion is detected; the audible alarm is turned on if the current time (`cur_time`) is within the `alarm_start/alarm_stop` window. At the end of the `while(mode==DISPLAY_TIME){}` loop, the processor sleeps if no audible alarm is sounding and no debouncing of pushbutton or rotary inputs is in progress. The processor is awakened on the next Timer1 interrupt.

Figure 14.22 gives the remainder of `main()` and contains the code that implements the *set time*, *set alarm start time*, and *set alarm stop time* modes. In each mode, a left button activation advances to the next mode, while a right button activation enters the time modification mode. Each mode has a set of three variables that define the time to be modified: `hour/min/sec` for the *set time* mode, `hour1/min1/sec1` for the *set alarm start time* mode, and `hour2/min2/sec2` for the *set alarm stop time* mode. In each case, the appropriate three-variable set that defines the time to be modified is copied to the temporary variables `t_hour/t_min/t_sec`,

and the `do_settime()` function is called to perform the actual time modification. On return, the `t_hour/t_min/t_sec` values are copied back to the appropriate three-variable set. For the *set alarm start time* and *set alarm stop time* modes, the values are saved back to data EEPROM after returning from `do_settime()`.

```

#define DISPLAY_TIME 0
#define SET_TIME 1
#define SET_START 2
#define SET_STOP 3
    } States definitions for state machine IO in primary
    } input loop

main(void) {
    char unsigned mode;
    do_config(); //configure subsystems
    mode = DISPLAY_TIME; temp_mode = 0; old_sec = sec;
    while(1) {
        switch (mode) {
            case DISPLAY_TIME:
                } Default mode is to display
                } time and temperature
                while(mode==DISPLAY_TIME) {
                    if (left_button) {
                        CCPRL1 = 0x00; // turn off alarm
                        alarm_count = 0; // clear alarm delay
                        left_button = 0; //clear flag
                        mode = SET_TIME;
                    }
                    } Left button pushed, so turn
                    } off alarm by setting duty cycle
                    } to zero, clear semaphore,
                    } set new mode to SET_TIME
                    if (right_button) {
                        CCPRL1 = 0x00; // turn off alarm
                        alarm_count = 0;
                        right_button = 0; //clear flag
                        temp_mode = ~temp_mode;
                    }
                    } Right button pushed, so turn
                    } off alarm, toggle temperature
                    } display mode (Fahrenheit ? Celsius)
                    if (old_sec != sec) {
                        old_sec = sec;
                        if (alarm_count <= ALARM_DELAY)
                            alarm_count = alarm_count+2;
                        update_lcd_time();
                        if (motion) {
                            // see if should turn on alarm
                            motion = 0; INT0IF = 0; INT0IE = 1;
                            if (alarm_count > ALARM_DELAY) {
                                if ((cur_time > alarm_start) &&
                                    (cur_time < alarm_stop)) {
                                    CCPRL1 = 0x30; // turn on alarm
                                }
                            } // end if alarm_count > alarm_display
                        } // end if (motion)
                    } //end if (old_sec != sec)
                    GIE = 0;
                    if (INT1IE && INT2IE &&
                        !int0_cnt && !int1_cnt && !CCPR1L) {
                        // no debouncing or alarm so sleep,
                        // wake on timer1 rollover every 2 sec
                        asm("sleep");
                    }
                    } Will wake on next rollover of
                    } Timer1 (every 2 seconds)
                    GIE = 1;
                } //end while(
                break;
    }

```



**FIGURE 14.21** `main()` part I of home monitoring application.

```

case SET_TIME:
    new_choice(stime_msg);
    while(mode==SET_TIME) {
        if (left_button) {left_button = 0; mode = SET_START;}
        if (right_button) {
            right_button = 0;
            GIE=0;t_hour=hour;t_min=min;t_sec=sec; GIE=1;
            do_settime();
            GIE=0; hour=t_hour;min=t_min;sec=t_sec;
            cur_time = hour*60+min*60+sec;
            GIE=1;
            new_choice(stime_msg);
        }
    } break;
case SET_START:
    t_hour=hour1;t_min=min1;t_sec=sec1;
    new_choice(mon1_msg);update_lcd_time1(0xC0);
    while(mode==SET_START) {
        if (left_button) {left_button = 0; mode = SET_STOP;}
        if (right_button) {
            right_button = 0;
            GIE=0;t_hour=hour1;t_min=min1;t_sec=sec1; GIE=1;
            do_settime();
            GIE=0;hour1=t_hour;min1=t_min;sec1=t_sec; GIE=1;
            save_start_stop();// write start/stop to eeprom
            new_choice(mon1_msg);update_lcd_time1(0xC0);
        }
    }break;
case SET_STOP:
    t_hour=hour2;t_min=min2;t_sec=sec2;
    new_choice(mon2_msg);update_lcd_time1(0xC0);
    while(mode==SET_STOP) {
        if (left_button) {left_button = 0; mode = DISPLAY_TIME;}
        if (right_button) {
            right_button = 0;
            GIE=0;t_hour=hour2;t_min=min2;t_sec=sec2;GIE=1;
            do_settime();
            GIE=0;hour2=t_hour;min2=t_min;sec2=t_sec; GIE=1;
            save_start_stop();
            new_choice(mon2_msg);update_lcd_time1(0xC0);
        }
    }break;
} //end switch (mode)
} // end while(1)
} // end main()

```

Change to next mode on left button press

Copy hour/min/sec to temporary variables, let do\_settime() use rotary encoder inputs to adjust time

Change to next mode on left button press

Adjust alarm start time, then save to Data EEPROM

Change to next mode on left button press

Adjust alarm stop time, then save to Data EEPROM



**FIGURE 14.22** main() part II of the home monitoring application (see CD-ROM file ./code/chap14/F\_14\_21\_alarm.c).

Figure 14.23 shows the do\_config() function called by main() to perform the port IO and subsystem configuration to match the resource usage of Table 14.5. Observe that a DS1621 temperature conversion is started and polled until complete so that a valid temperature will be ready when displayed by update\_lcd\_time().

```

do_config() {
    char unsigned status;
    RBP0 = 0; // enable the weak pullup on port B
    read_start_stop(); // read old alarm values from eeprom } Read start/stop
    alarm_start = hour1*60+min1*60+sec1; } monitor times from
    alarm_stop = hour2*60+min2*60+sec2; } Data EEPROM
    i2c_init(72); // init I2C interface
    ds1621_send1(Access_CONFIG,0x01); // one shot mode
    ds1621_send0(START_CONVERT); // start conversion } Init DS1621 and take
    do{ // wait until first conversion is done } initial temperature
        status = ds1621_read1(Access_CONFIG); } reading
    } while(!bittst(status,7));

    // rotary setup
    ROT0_IN; ROT1_IN;
    int0_last = ROT0; int1_last = ROT1; } Initialize rotary encoder
    last_state = 0; } inputs and state
    if (ROT0) bittst(last_state,0);
    if (ROT1) bittst(last_state,1);

    E_OUTPUT; RS_OUTPUT; RW_OUTPUT; EL0W; RSL0W; RWL0W; } LCD interface init
    lcd_init(); update_lcd_time();

    // initialize timer 1, prescale by 1, ext. async. clock
    T1CKPS1 = 0; T1CKPS0 = 0; TIOSCEN = 0; TMR1CS = 1; T1SYNC = 1; } Timer1
    bitset(TRISC,0); // set T1CKI/RC0 as input } init

    // config. tmr2 for PWM mode, 0% duty cycle,pre=16, CCP1 output
    T2CKPS1 = 1; PR2 = 255; CCP1IL = 0;
    bitclr(CCP1CON, 5); bitclr(CCP1CON, 4); bitclr(TRISC,2); } Timer2 frequency
    bitset(CCP1CON, 3); bitset(CCP1CON, 2); } is ~ 1800 Hz
    TMR2ON = 1;

    // use timer3 for debounce, int. clock
    T3CKPS1 = 0; T3CKPS0 = 0; TMR3CS = 0; T3SYNC = 0; } Timer3 init, PRE=1 for
    TMR3IF = 0; TMR3IE = 1; TMR3ON = 1; } periodic interrupt ~ 9 ms

    // pushbutton input configuration, fall. edge interrupt
    TRISB1 = 1; INT1IF = 0; INTEDG1 = 0;INT1IE = 1;
    TRISB2 = 1; INT2IF = 0; INTEDG2 = 0;INT2IE = 1; } Pushbutton input
    // motion sensor input, rising edge interrupt } and motion sensor
    TRISB0 = 1; INTEDG0 = 1; INT0IF=0; INT0IE=1; } input configuration

    // enable timer 1 and general interrupts
    TMR1IF = 0; TMR1IE = 1; TMR1ON = 1; } Timer1 on and interrupts enabled
    IPEN = 0; PEIE = 1; GIE = 1;
}

```



**FIGURE 14.23** do\_config() function (see CD-ROM file ./code/chap14/F\_14\_21\_alarm.c).

Figure 14.24 shows the do\_settime() function that performs the time modification for the *set time*, *set alarm start time*, and *set alarm stop time* modes in Figure 14.22. The t\_hour, t\_min, t\_sec variables are displayed on LCD line 1 and the cursor is set to the hours field. In the while(1){} loop, one of the variables t\_hour, t\_min, or t\_sec is copied to the rotcount variable and the appropriate rotary encoder min/max clipping values are set (rot\_min/rot\_max) based upon the current LCD cursor position. This is done because the ISR for the rotary encoder modifies the rotcount variable and clips it to within the range set by rot\_min and rot\_max.

The pushbutton and rotary encoder semaphores are cleared, then interrupts are enabled, and the function waits for one of the semaphores to change value. If the `right_button` semaphore is set, the LCD cursor is advanced to the next field. If the `rotcount` value is changed, `rotcount` is copied back to the appropriate `t_hour`, `t_min`, or `t_sec` variable. If the `left_button` semaphore is set, the `do_settime()` function is exited.

```

//right button advances cursor, left button exits
void do_settime(){
  char pos;
  GIE=0;
  pos = 0x81; // hours position           } Write time value to specified
  lcd_write(0x0E,0,0,1); // turn on cursor } line, set cursor to hours position
  while(1){
    switch (pos) {
      case 0x81: rotcount=t_hour;rot_min=0;rot_max=23; } hours position, copy t_hour
        break;                                         } to rotcount for modification
      case 0x84: rotcount = t_min;rot_min=0;rot_max=59; } minutes position, copy t_min
        break;                                         } to rotcount for modification
      case 0x87: rotcount = t_sec;rot_min=0;rot_max=59; } seconds position, copy t_sec
        break;                                         } to rotcount for modification
    }
    update_lcd_time1(0x80);
    lcd_write(pos,0,1,1); //position cursor
    // wait for change of input
    last_rotcount = rotcount;
    right_button=0; left_button=0;          } Clear semaphores for button inputs and
    GIE=1;                                   } rotary encoder inputs
    while(!left_button && !right_button    } Wait for any semaphore to change, ISR
      && (rotcount == last_rotcount));    } will update rotcount if rotary inputs
    GIE=0;                                   change
    if (rotcount != last_rotcount){
      // update values
      switch (pos) {
        case 0x81: t_hour=rotcount;break;
        case 0x84: t_min=rotcount;break;
        case 0x87: t_sec=rotcount;break;
      }
    }
    if (right_button) {
      pos = pos+3;
      if (pos == 0x8A) pos = 0x81;
    }
    if (left_button) {
      // turn off cursor
      lcd_write(0x0C,0,0,1);
      break; // exit loop
    }
  } // end while(1)
  // before returning, clear input flags, restore interrupts
  last_rotcount = rotcount;
  right_button=0;
  left_button=0;
  GIE=1;
}

```



**FIGURE 14.24** `do_settime()` function (see CD-ROM file `./code/chap14/F_14_21_alarm.c`).



Figure 14.25 shows some miscellaneous functions that are called by `main()`, `do_settime()`, and `do_config()`. The `update_lcd_time()` function is called by the display time mode to update the LCD with the current time (line 1) and temperature (line 2). The current temperature is read from the DS1621 and a new conversion is started. After converting the temperature to Fahrenheit, either the Celsius or Fahrenheit value is displayed based on the `temp_mode` value. If the motion semaphore is set, an “M” is printed in the first character position of line 2. The

```

#define BUFSIZE 64
char buf[64];

update_lcd_time1(unsigned char pos) {
    lcd_write(pos,0,1,1); // 1st line
    printf("%02d:%02d:%02d",t_hour,t_min,t_sec);
}
}

update_lcd_time(){
    unsigned char status, c;
    lcd_write(0x80,0,1,1); // 1st line
    printf("%02d:%02d:%02d",hour,min,sec);
    lcd_write(0xC0,0,1,1); // 2nd line
    // check if temp conversion done, if yes, update
    status = ds1621_read1(ACCESS_CONFIG);
    if (bittst(status,7)){
        ds1621_read2(READ_TEMP,(char *)&raw_temp);
        // start new conversion
        ds1621_send0(START_CONVERT);
    }
    if (motion) c = 'M'; else c = ' ';
    temp_c = raw_temp;
    temp_c = temp_c >> 8;
    temp_f = (temp_c*9)/5 + 32;
    if (temp_mode) printf("%c%5d C",c,temp_c);
    else printf("%c%5d F",c,temp_f);
}

const unsigned char stime_msg[]="Set Time";
const unsigned char blank_msg[]=" ";
const unsigned char mon1_msg[]="Mon Strt";
const unsigned char mon2_msg[]="Mon Stop ";

new_choice(const char *s){
    lcd_write(0x80,0,1,1); // 1st line
    printf("%s",s);
    lcd_write(0xC0,0,1,1); // 2nd line
    printf("%s",blank_msg);
}

read_start_stop() {
    eedata_readstr(buf,0,BUFSIZE);
    sscanf(buf,"%d %d %d %d %d %d",&hour1,&min1,&sec1,&hour2,&min2,&sec2);
}

save_start_stop() {
    sprintf(buf,"%d %d %d %d %d %d",hour1,min1,sec1,hour2,min2,sec2);
    eedata_writestr(buf,0);
}

```

Write time value to specified LCD line

Called during DISPLAY\_TIME mode to display time/temperature

Update time on LCD line 1

Update temperature and motion flag on LCD line 2

Strings for each mode

Update LCD display with new mode string

Read start, stop times from Data EEPROM

Save start, stop times to Data EEPROM



**FIGURE 14.25** Miscellaneous support functions (see CD-ROM file `./code/chap14/F_14_21_alarm.c`).

read\_start\_stop() function reads the alarm start/stop times from data EEPROM and is called from within do\_config(). The save\_start\_stop() function writes the alarm start/stop times to Data EEPROM and is called from within do\_settime(). The update\_lcd\_time1() and new\_choice() functions are called from within the primary IO loop of main() to update the LCD display when modes change.

Figure 14.26 shows the ISR for the home monitoring application. The Timer1 interrupt causes the time of day to be advanced two seconds. The INT1 and INT2 interrupts are used for the pushbutton inputs. These interrupts set their appropriate semaphores and also disable the interrupt until the input is debounced. The

```
// variables for ISR and ISR support functions
volatile unsigned char left_button, left_debounce, right_button, right_debounce;
volatile unsigned char motion, alarm_count, hour, min, sec, old_sec;
volatile unsigned char t_hour, t_min, t_sec, hour1, min1, sec1;
volatile unsigned char hour2, min2, sec2, update_flag;
volatile unsigned char int0_cnt, int0_last, int1_cnt, int1_last;
unsigned long alarm_start, alarm_stop, cur_time;
volatile signed int raw_temp, temp_c, temp_f, temp_mode;

void interrupt isr(void) {
    if (TMR1IF && TMR1IE) { // time-of-day
        TMR1IF=0; sec = sec + 2; // seconds
        cur_time=cur_time+2;
        if (sec > 59) {
            min++; sec = sec-60;
            if (min == 60) {
                hour++; min = 0;
                if (hour == 24) {
                    hour = 0; cur_time = 0;
                }
            }
        }
    }

    if (INT1IF && INT1IE) { // pushbutton detected
        INT1IE = 0; right_button = 1; right_debounce = 0;
    }

    if (INT2IF && INT2IE) { // pushbutton detected
        INT2IE = 0; left_button = 1; left_debounce = 0;
    }

    if (INT0IF && INT0IE) { // motion sensor
        motion = 1; INT0IE = 0;
    }

    if (TMR3IF) { // debouncing timer
        TMR3IF = 0;
        do_debounce();
    }
}

// Comments for ISR actions:
// Timer1 rollover, adjust time of day
// Right button pressed, set semaphore, clear debounce count
// Left button pressed, set semaphore, clear debounce count
// Motion detected, set semaphore
// Timer3 periodic interrupt, do debounce and sample rotary inputs
```



**FIGURE 14.26** Interrupt service routine for the home monitoring application (see CD-ROM file ./code/chap14/F\_14\_21\_alarm.c).

INT0 interrupt sets the motion sensor semaphore (`motion`) and disables the interrupt; the INT0 interrupt is re-enabled in `main()` after the semaphore is detected and cleared. The Timer3 interrupt is configured to generate a periodic interrupt of 9 ms and is used to debounce the pushbutton inputs and sample the rotary encoder inputs via the `do_debounce()` function shown in Figure 14.27.

```
do_debounce() {
    //debounce pushbuttons
    if (!right_button && !INT1IE) {
        if (RB1) right_debounce++;
        else right_debounce=0;
        if (right_debounce == DEBOUNCE) {
            //re-enable interrupt
            INT1IF=0;INT1IE=1;
        }
    }
    if (!left_button && !INT2IE) {
        if (RB2) left_debounce++;
        else left_debounce=0;
        if (left_debounce == DEBOUNCE) {
            //re-enable interrupt
            INT2IF=0;INT2IE=1;
        }
    }
    if (ROT0 != int0_last) { // debounce rotary inputs
        int0_cnt++;
        if (int0_cnt == DEBOUNCE) {
            update_flag = 1; int0_cnt = 0; int0_last = ROT0;
        }
    }
    // reset cnt, if pulse width not long enough
    else if (int0_cnt) int0_cnt = 0;
    if (ROT1 != int1_last) {
        int1_cnt++;
        if (int1_cnt == DEBOUNCE) {
            update_flag = 1; int1_cnt = 0; int1_last = ROT1;
        }
    }
    // reset cnt, if pulse width not long enough
    else if (int1_cnt) int1_cnt = 0;
    if (update_flag) {
        // can read the rotary inputs
        update_rotary_state(); update_flag = 0;
    }
}
}
```

} Debounce right pushbutton, re-enable interrupt only after button has been idle high for DEBOUNCE Timer3 interrupt periods

} Debounce left pushbutton, re-enable interrupt only after button has been idle high for DEBOUNCE Timer3 interrupt periods

} ROT0 input changed, debounce. If stable long enough, set update flag.

} ROT1 input changed, debounce. If stable long enough, set update flag.

} If update\_flag is set, then update rotary encoder state.



**FIGURE 14.27** `do_debounce()` function (see CD-ROM file `./code/chap14/F_14_21_alarm.c`).

Debouncing of the pushbutton inputs is done in the same manner as in Chapter 10, “Interrupts and a First Look at Timers.” A pushbutton input is considered debounced if it remains high for DEBOUNCE consecutive Timer3 interrupts, after which the appropriate interrupt is re-enabled.

Sampling and debouncing of the rotary inputs is performed in the same manner as in Chapter 10. If a rotary encoder input is different from its previous value for DEBOUNCE consecutive Timer3 interrupts, this is considered a valid change in

value and the `update_rotary_state()` function in Figure 14.28 is called. The `update_rotary_state()` has the same basic structure as originally presented in Chapter 10. The `rotcount` variable is either incremented or decremented based on the change in rotary encoder state. Additionally, the `rotcount` value is clipped to remain within the range `rot_min` to `rot_max`.

```

#define ROT1      RB7
#define ROT0      RB6
#define ROT1_IN  TRISB7=1
#define ROT0_IN  TRISB6=1
#define ROT_S0 0
#define ROT_S1 1
#define ROT_S2 2
#define ROT_S3 3
volatile unsigned char state,last_state,rotcount,last_rotcount;
volatile unsigned char rotdir,rot_min,rot_max;
update_rotary_state(){
    state = 0;
    if (ROT0) bitset(state,0);
    if (ROT1) bitset(state,1); } Change state based on rotary encoder inputs
    switch(state) {
    case ROT_S0:
        if (last_state == ROT_S1) {
            rotcount++; rotdir = 1; last_state = state;
        }else if (last_state == ROT_S2) {
            rotcount--;rotdir = 0; last_state = state;
        }break;
    case ROT_S1:
        if (last_state == ROT_S3) {
            rotcount++;rotdir=1; last_state = state;
        }
        else if (last_state == ROT_S0) {
            rotcount--; rotdir=0; last_state = state;
        }break;
    case ROT_S2:
        if (last_state == ROT_S0) {
            rotcount++; rotdir=1;last_state = state;
        }
        else if (last_state == ROT_S3) {
            rotcount--;rotdir=0;last_state = state;
        }break;
    case ROT_S3:
        if (last_state == ROT_S2) {
            rotcount++; rotdir=1;last_state = state;
        }
        else if (last_state == ROT_S1) {
            rotcount--; rotdir=0;last_state = state;
        }break;
    } //end switch
    if (rotdir){ // incremented
        if (rotcount == (rot_max+1)) rotcount = rot_min;
    } else { //decremented
        if (!rot_min && (rotcount > rot_max))
            rotcount = rot_max;
        else if (rotcount == (rot_min-1)) rotcount = rot_max;
    }
} //end update_rotary_state()
    
```

Increment or decrement  
rotcount based on current  
state and last state of rotary  
encoder.

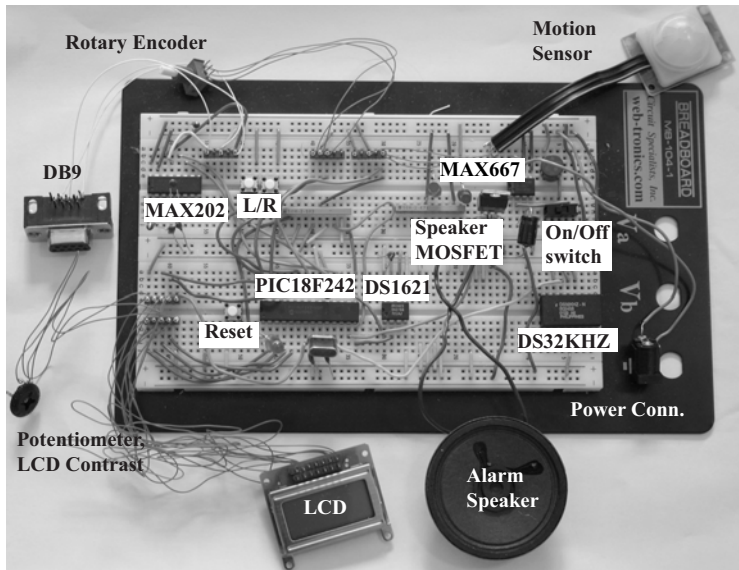
Set rotdir variable to "1" if  
incremented, to "0" if  
decremented

Clip rotcount  
to between rot\_max  
and rot\_min.



**FIGURE 14.28** `update_rotary_state()` function (see CD-ROM file `./code/chap14/rot_module.c`).

Figure 14.29 shows a prototype of the home monitoring application. Wire-wrap was used with headers to connect to off-board components such as the LCD, potentiometer, rotary encoder, and so forth. In this prototype, use of the PIC18 sleep functionality in the display time mode reduced average current consumption from 18 mA to 8 mA.

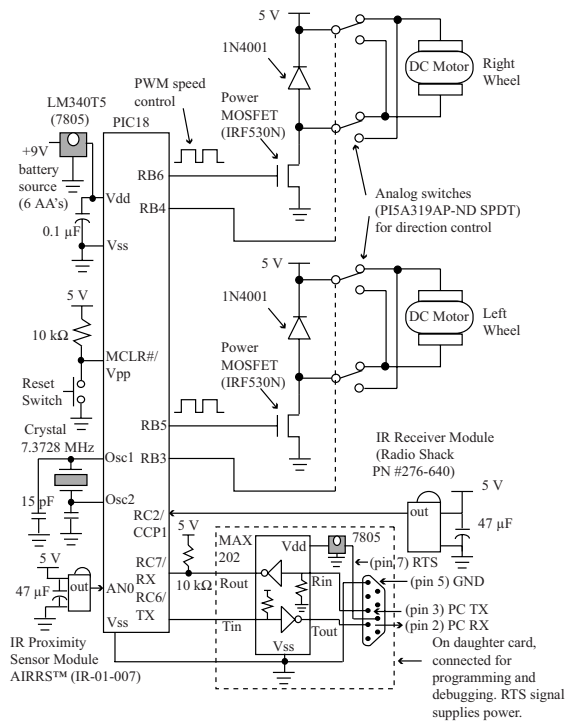


**FIGURE 14.29** Prototype of the home monitoring application.

## 14.8 DESIGN AND IMPLEMENTATION OF AN AUTONOMOUS ROBOT

Small, autonomous robots have become popular as a means for stimulating interest in embedded systems. Many regional IEEE student design contests feature autonomous robots as the design target. Small autonomous robots also make for good demonstrations to high school students during engineering recruiting drives. Figure 14.30 gives the schematic of a wheeled robot design that can be manually driven via an IR remote control or function autonomously using an IR proximity sensor to detect forward obstructions.

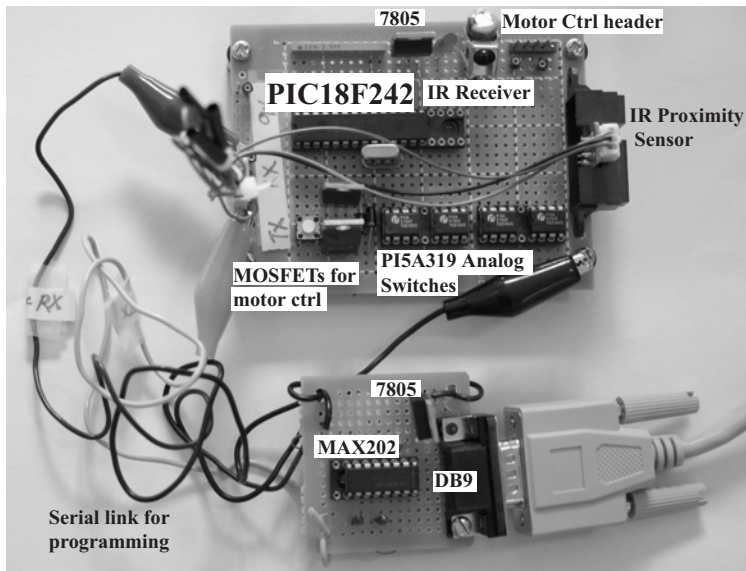
The design assumes a three-wheeled chassis with left and right wheels and a back pivot wheel. The IR receiver discussed previously in Chapter 13 is used for sending commands to the robot in manual drive mode. The infrared proximity sensor in Figure 14.30 outputs a voltage that is  $\sim 1$  V when no obstruction is pre-



**FIGURE 14.30** Schematic of robot application.

sent and increases to ~3 V when an obstruction is placed directly in front of it. The PIC18 ADC is used to sample this sensor input to determine if there is an obstruction in front of the robot during autonomous drive mode. Pulse width modulation is used to control the speed of the separate DC motors that drive the left and right wheels. Individual speed control of each wheel is needed, as a turn is accomplished by slowing one wheel as the other wheel is kept at the same speed. A left turn is accomplished by slowing the left wheel; a right turn by reducing the speed of the right wheel. The automated PWM mode of the PIC18 cannot be used, as the Timer2/CCP1 combination only implements a single PWM channel. Instead, manual PWM is implemented on port pins RB6 and RB5 via periodic interrupts. This approach produces a coarse-grained PWM, but it is sufficient for this application. The schematic of Figure 14.30 shows the RS232 interface used for programming mounted on a separate daughter card, as it is not needed during robot operation. A weak pullup (10K resistor) is needed on the PIC18 RX input to force a “1” (idle condition) when the MAX202 chip is not driving the RX input, so that the serial bootloader on the PIC18 that reads the RX port after reset does not see a floating input. The RS232 daughter card is self-powered through the RS232 port by using

the RTS (request-to-send) handshaking line as a Vdd source. Connector clips are used to connect the TX, RX, and GND lines of the daughter card to the main board during programming operation, which can be performed while the main board is sitting on the robot chassis. A picture of the main board and RS232 daughter card is shown in Figure 14.31. Small component PC boards from Radio Shack (PNs #276-168, #276-148) and wire-wrap sockets/interconnect were used for creating the prototype.



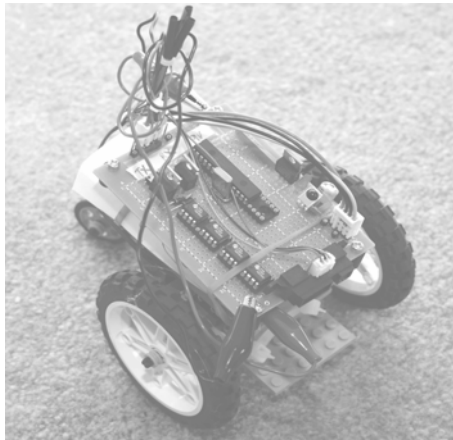
**FIGURE 14.31** Prototype board for robot application.

Figure 14.32a shows the prototype board mounted on a three-wheeled chassis built from a LEGO Mindstorms™ robot kit. The 9 V battery pack within the controller module of the Mindstorms kit is used as the power source for the PIC18 board. Rubber band and cable tie engineering is used to mount the prototype board on top of the battery pack and to keep the battery pack securely on the chassis. Figure 14.32b shows a printed circuit board (PCB) implementation of the robot electronics mounted on the same chassis.

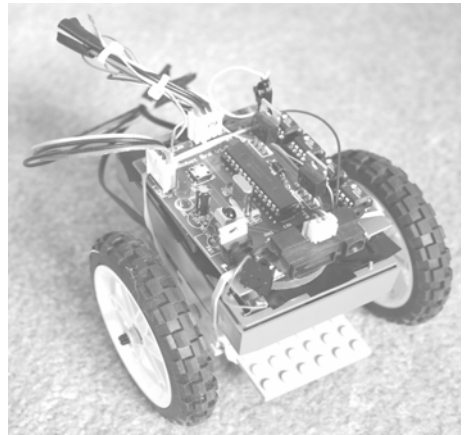
Table 14.6 gives the PIC18 resources used for the robot application. The CCP2 compare mode is used as period control for Timer3 that generates a periodic interrupt used to update the PWM outputs for the two wheels and to sample the ADC that is connected to the IR proximity sensor output.

The IR receiver output is decoded using Timer2 and the CCP1 compare mode for pulse width measurement in exactly the same manner as discussed in Chapter

13. The USART is used during programming of the PIC18 via a serial bootloader and was also useful during debugging of the application.



a) Wirewrap Prototype



b) PCB version

**FIGURE 14.32** Robot electronics mounted on a three-wheel chassis.

**TABLE 14.6** PIC18 Resources Used for Robot Application

PIC18 Resource	Comment
RB5/RB6	Manual PWM outputs for left/right wheel speed control.
RB3/RB4	Direction control for left/right wheels.
Timer3/CCP2 Compare	CCP2 is used as a period register for Timer3 to generate a periodic interrupt for RB5/RB5 PWM control and to sample the IR sensor output.
Timer2/CCP1 Capture Input	Used for infrared receiver output decoding.
ADC (AN0)	Used to read the IR proximity sensor output.
USART	Asynchronous serial port used for PIC18 programming via a bootloader. Not used during the application execution.

Figure 14.33 shows the primary variables/functions used for the robot application (the variables and functions used in decoding the IR output are not shown, see Chapter 13). Refer to these variables and functions as the code for the robot application is discussed in the remaining figures of this section.



Variable(s)	Comment
dc_left, dc_right	duty cycle for left/right wheel PWM
period_left, period_right	period counters for left/right wheel PWM
adc_val, old_adc_val	A/D sampled value
match	CCP2 match value for Timer3
cur_time	time-of-day in seconds
Functions	Comment
auto_drive()	Performs robot automatic drive mode
manual_drive()	Performs robot manual drive mode
handle_cmd()	Executes command received from IR receiver
phillips_convert()	Convert received IR byte to Phillips RC5 format
all_stop()	Slow to a stop
dir_fwd()	Set direction of both wheels to forward
dir_back()	Set direction of both wheels to backward
do_turn()	Execute a turn
reset_ir()	Reset for next IR reception
do_ircap()	Capture bytes from IR receiver

Variables for IR capture not shown, see Chapter 13.

**FIGURE 14.33** Variables/functions for robot application.

Figure 14.34 shows the main() function for the robot application. The initialization code configures the PIC18 subsystems per the usage given in Table 14.6. Timer3 is configured for a prescale of 2 and uses CCPR2 as a period control register; a value of 0x0800 for MATCH\_INTERVAL gives a periodic CCP2IF interrupt

```

main(void) {
    //serial_init(95,1); //debug only
    // initialize timer1, prescale by 1, internal clock
    T1CKPS1 = 0; T1CKPS0 = 0; T1OSCEN = 0; TMR1CS = 0; } Timer1 config
    bitset(TRISC,2); // set CCP1 as input for IR Capture
    // set up everything for PWM, direction control
    bitclr(TRISB,OP_LEFT); bitclr(TRISB,OP_RIGHT);
    bitclr(TRISB,OP_RDIR); bitclr(TRISB,OP_LDIR); } Port direction for wheel
    bitclr(PORTB,OP_LEFT); bitclr(PORTB,OP_RIGHT); control
    bitclr(PORTB,OP_RDIR); bitclr(PORTB,OP_LDIR);
    // init timer3, prescale by 2, int. clock
    T3CKPS1 = 0; T3CKPS0 = 1; TMR3CS = 0; T3SYNC = 0;
    // TMR3 with CCP2, TMR1 with CCP1
    T3CCP2=0;T3CCP1=1;
    // setup capture mode, enable capture interrupt
    CCP2CON = 0x02; CCP2IF = 0; CCP2IE = 1;
    CCPR2H = (MATCH_INTERVAL >> 8);
    CCPR2L = (0xFF & MATCH_INTERVAL);
    TMR3ON = 1; } Timer3 configuration
    // configure A/D for IR sensor, right. just, channel 0
    TRISA = 0xFF;
    ADCON1 = 0x0E; ADCON0 = 0x80; ADON = 1; } A/D configuration
    IPEN = 0; PEIE = 1; GIE = 1;
    while(1) {
        switch (mode) {
            case 0: manual_drive(); } Robot is either in manual
            break; or
            case 1: auto_drive(); automatic drive modes
            break;
        }
    }
}

```



**FIGURE 14.34** main() function of the robot application.

of approximately 556  $\mu\text{s}$  using an  $\text{FOSC} = 29.4912 \text{ MHz}$ . The `while(1){}` loop of `main` calls either the `manual_drive()` or `auto_drive()` functions depending upon the current mode setting.

Figure 14.35 shows the interrupt service routine that is responsible for updating the PWM outputs, sampling the ADC input, and decoding the IR receiver input. The `CCP2IF` interrupt occurs at 556  $\mu\text{s}$  intervals when the `CCPR2` register value matches the `Timer3` value. The `match` variable is incremented by `MATCH_INTERVAL` and this value is written to the `CCPR2` register to update it to the next `Timer3` match value. The ADC input is read and a new A/D conversion is

```

#define PWM_MAX 10 // # of timer3 interrupts for one PWM period
#define OP_LEFT 5 // left wheel port bit
#define OP_RIGHT 6 // right wheel port bit
#define OP_LDIR 3 // left wheel direction port bit
#define OP_RDIR 4 // right wheel direction port bit

//Manual PWM control variables
unsigned char dc_left,dc_right, period_left, period_right;

// use timer3, CCPR2 for PWM generation
#define MATCH_INTERVAL 0x0800

unsigned int match;
unsigned char adc_val, old_adc_val;

void interrupt isr(void){
    if (CCP2IF) {
        CCP2IF = 0;
        adc_val = ADRESH; // get AD value
        GODONE = 1; // new conversion
        old_adc_val = adc_val;
        // don't clear timer1, change compare register
        match = match + MATCH_INTERVAL;
        CCPR2H = match >> 8;
        CCPR2L = match & 0xFF;
        CCP2IF = 0;
        period_left++;
        if (period_left == PWM_MAX) {
            period_left = 0;
            if (!dc_left) bitclr(PORTB,OP_LEFT);
            else bitset(PORTB,OP_LEFT);
        }
        else if (period_left == dc_left) {
            bitclr(PORTB,OP_LEFT);
        }
        period_right++;
        if (period_right == PWM_MAX) {
            period_right = 0;
            if (!dc_right) bitclr(PORTB,OP_RIGHT);
            else bitset(PORTB,OP_RIGHT);
        }
        else if (period_right == dc_right) {
            bitclr(PORTB,OP_RIGHT);
        }
    }
}

```

} Sample ADC to read IR distance sensor

} CCPR2 interval has expired, update  
} CCPR2 with new value for next match

} Update left wheel PWM Period.

} If at end of period, reset left\_period  
} variable. If left wheel duty cycle is  
} zero, turn off output port, else turn it  
} on to start high portion of square wave.

} If left wheel high-portion of square  
} wave is finished, turn off output port.

} Update right wheel period and  
} duty cycle in same manner as  
} left wheel.

} Not shown: TMR1IF and CCP1IF for handling IR data reception, see Chapter 13



**FIGURE 14.35** Interrupt service routine for the robot application (see CD-ROM file `./code/chap14/F_14_34_robot.c`).

started. The `period_left` and `period_right` variables track the period of the square waves used for PWM of the left/right wheels. When these counter values are equal to `PWM_MAX` (a value of 10), the period is finished and these counters are reset. This gives a square wave period of 5560  $\mu\text{s}$  (a frequency of  $\sim 180$  Hz). The `dc_left` and `dc_right` variables control the high pulse width of the PWM square waves. When the period counter is equal to the duty cycle counter (`period_left == dc_left` or `period_right == dc_right`), the square wave output is set to zero. At the beginning of a period, the square wave output is set high unless the duty cycle count is zero. Increasing the high pulse width of the PWM square wave (higher values of `dc_left/dc_right`) increases wheel speed. The Timer1 and CCP1 interrupts for decoding the IR receiver output are not shown, as it is the same code presented in Chapter 13.

Figure 14.36 shows the functions for performing manual (`manual_drive`) and autonomous (`auto_drive`) operation. The `manual_drive()` function simply waits for an IR command and then executes it via the `handle_cmd()` function. The `auto_drive()` function also uses the `handle_cmd()` function to execute an IR command if one is received, and moves autonomously in the absence of an IR command. The robot moves forward until an obstruction is detected by the ADC input value exceeding `BLOCKAGE_THRESHOLD`. If obstructed, the robot comes to a full stop. Then the robot backs up, slowing the left wheel so that the front of the robot swings to the right. Then the robot moves forward, slowing the right wheel so that the turn started during the backward movement is continued. After 0.6 seconds of forward motion, both wheels are set to the same speed to straighten the robot out.

Figure 14.37 shows the `handle_cmd()` function used in both manual and autonomous modes to execute IR commands. The `philips_convert()` function called by `handle_cmd()` converts the last received byte to Philips RC5 format (see Chapter 13). Because the universal remote control (Radio Shack remote control programmed with code 333) used to test this robot always sent two duplicate IR sequences for each button press, the command is not executed until the current command matches the last received command indicating the two duplicate commands have been received.

The `switch(cmd){}` statement within `handle_cmd()` executes the received IR command as summarized in Table 14.7.

Figure 14.38 shows miscellaneous support functions used within the robot application. These are fairly simple and are explained by the annotation contained within the figure.

```

#define BLOCKAGE_THRESHOLD 60 ←———— A/D value of IR sensor input if blockage
#define CLEAR_THRESHOLD 40 ←———— A/D value of IR sensor input if no blockage
#define FWD_SPEED 4 ←———— Forward motion speed
#define BACK_SPEED 8 ←———— Backward motion speed

auto_drive(){
  reset_ir();
  while(mode == AUTO_DRIVE) {
    if (state == IO_FINISH) {
      // if IR arrives, execute it
      handle_cmd();
      reset_ir();
    } else {
      // auto driving
      if (dc_left || dc_right) { // moving...
        if (old_adc_val > BLOCKAGE_THRESHOLD) {
          // STOP!!!!!!
          dc_left = 0; dc_right = 0;
          Delay_tens(3);
        }
        // stopped
        if (old_adc_val < CLEAR_THRESHOLD) {
          // go forward again
          dir_fwd(); do_speedup (FWD_SPEED);
        } else {
          // backup, pivot
          dir_back();
          do_speedup (BACK_SPEED);
          Delay_tens(3);
          // slow down on wheel so it pivots
          dc_left = 2;
          Delay_tens(8); dc_left = 0; dc_right = 0;
          Delay_tens(2);
          dir_fwd(); do_turn (BACK_SPEED,2);
          Delay_tens(6);
          dc_left = FWD_SPEED; dc_right = FWD_SPEED;
        }
      } //end if(dc_left...)else{}...
    } //end if(state == IO_FINISH)else{}..
  } //end while(mode == AUTO_DRIVE)
} // end auto_drive()

manual_drive(){
  while(mode == MAN_DRIVE) {
    reset_ir();
    // wait for IR data to arrive
    while (state != IO_FINISH);
    handle_cmd();
  }
}

```

While in auto drive, will accept IR commands. If an IR command is present, then execute it via the `handle_cmd()` function.

While moving, stop if blockage is detected.

Stopped, but no blockage so start moving forward again

Stopped, and obstruction is detected. Backup and slow down left wheel so that bot turns as it backs up. Go forward but now keep right wheel slow so that it keeps turning in same direction. End pivot by going forward with both wheels at the same speed.



**FIGURE 14.36** Manual/auto drive functions (see CD-ROM file `./code/chap14/F_14_34_robot.c`).

```

handle_cmd(){
  cmd = this_byte;
  philips_convert();           ← Convert IR byte to Philips RC5 Format
  if (last_cmd != cmd) {last_cmd = cmd;return;} ← Wait until duplicate cmd
  last_cmd = 0; cbuff[0] =0 ; cbuff[1] =0 ;           arrives before processing
  switch (cmd) {
  case STOP:
    all_stop(); } The all_stop() function does a gradual slow to a stop.
    break;
  case SPEEDUP:
    if (dc_left < PWM_MAX)dc_left += 2; } Increase speed of both wheels by
    if (dc_right < PWM_MAX)dc_right += 2; } increasing the duty cycle PWM
    break;
  case SLOWDOWN:
    if (dc_left) dc_left -= 2; } Decrease the speed of both wheels
    if (dc_right)dc_right -= 2; } by decreasing the duty cycle PWM
    break;
  case GOFORWARD:
    bitclr(PORTB,OP_RDIR); } Set both motor direction switches
    bitclr(PORTB,OP_LDIR); } to forward, and set the slowest
    if (dc_left > dc_right) dc_right = dc_left; } wheel to the speed of the fastest
    else dc_left = dc_right; } wheel
    break;
  case TURNLEFT:
    if (dc_left) dc_left -= 2; } Slow left wheel
    break;
  case TURNRIGHT:
    if (dc_right) dc_right -= 2; } Slow right wheel
    break;
  case REVERSE:
    dc_left = 0; dc_right = 0;
    DelayMs(100); DelayMs(100);
    DelayMs(100); DelayMs(100);
    bitset(PORTB,OP_RDIR);
    bitset(PORTB,OP_LDIR);
    dc_right = FWD_SPEED; dc_left = FWD_SPEED;
    break; } Stop, wait, change motor
    } direction switches,
    } then turn motors back on
  case SPIN:
    dc_left = 0; dc_right = 0;
    DelayMs(200); DelayMs(200);
    bitset(PORTB,OP_RDIR);
    dc_right = FWD_SPEED;dc_left = FWD_SPEED;
    break; } One wheel turns forward,
    } the other wheel turns backward
  case MODESWAP:
    if (mode == AUTO_DRIVE) mode = MAN_DRIVE;
    else mode = AUTO_DRIVE;
    all_stop();
    dir_fwd();
    break; } Toggle between manual
    } and auto drive modes
  }
}

```



**FIGURE 14.37** handle\_cmd() function of the robot application (see CD-ROM file ./code/chap14/F\_14\_34\_robot.c).

**TABLE 14.7** IR Command Mappings for Robot Control

<b>Univ. Remote Button</b>	<b>Code</b>	<b>Function</b>
VCR Stop	0x36	STOP—slow to a stop.
VCR Pause	0x29	SPEEDUP—increment both wheel duty cycle variables by 2.
VCR Record	0x37	SLOWDOWN—decrement both wheel duty cycle variables by 2.
VCR Play	0x35	FORWARD—set forward direction for both motors, and set the slowest wheel speed equal to the highest wheel speed.
VCR Rewind	0x32	TURN LEFT—decrement left wheel duty cycle variable by 2.
VCR Fast Forward	0x34	TURN RIGHT—decrement right wheel duty cycle variable by 2.
VCR numeral 5	0x05	REVERSE—stop, change direction switches on both wheels, then accelerate.
VCR numeral 9	0x09	SPIN—stop, set direction switch of left wheel opposite of right wheel, then accelerate.
VCR numeral 1	0x01	MODESWAP—toggle between auto/manual drive modes.

```

// on radio_shack universal remote code 333 VCR
// 0x0 - 0x9 buttons 0-9.
#define STOP      0x36 // stop (stop)
#define SPEEDUP   0x29 // speedup (pause)
#define SLOWDOWN  0x37 // slowdown (rec)
#define GOFORWARD 0x35 // forward (play)
#define TURNLEFT  0x32 // turn left (rewind)
#define TURNRIGHT 0x34 // turn right (fast fwd)
#define REVERSE   0x05 // reverse (#5)
#define SPIN      0x09 // spin (#9)
#define MODESWAP  0x01 // man/auto tgl (#1)
} Philips RC5 command codes from Radio Shack
  Universal remote control programmed with VCR device code of 333

#define MAN_DRIVE 0x00
#define AUTO_DRIVE 0x01

philips_convert(){
  i = cbuff[0]; // last full byte
  if (bittst(i,0)) bitset(cmd,6);
  else bitclr(cmd,6);
  cmd = cmd >> 1;
} Convert last byte received from IR transmitter to Philips RC5 format

Delay_tens(char k){
  while(k){ DelayMs(100);k--; } Software delay loop in tenths of seconds
}

all_stop() {
  while(dc_left || dc_right) {
    if (dc_left) dc_left--;
    if (dc_right)dc_right--;
    DelayMs(50);
  }
  Delay_tens(2);
} Function for slowing to a stop

dir_fwd() {bitclr(PORTB,OP_RDIR); bitclr(PORTB,OP_LDIR);} } Change wheel
dir_back() {bitset(PORTB,OP_RDIR); bitset(PORTB,OP_LDIR);} } direction

do_speedup(char target) {
  dc_left = 0; dc_right = 0;
  while (dc_left != target) {
    dc_left++;dc_right++;
    DelayMs(75);} Accelerate to target speed.
}

do_turn(char ltarg,char rtarg) {
  dc_left = 0; dc_right = 0;
  while ((dc_left != ltarg) || (dc_right != rtarg)) {
    if (dc_left != ltarg) dc_left++;
    if (dc_right != rtarg) dc_right++;
    DelayMs(75);
  }
} This gradually sets L/R wheel speed to match ltarg/rtarg.
}

```



**FIGURE 14.38** Support functions for robot application (see CD-ROM file ./code/chap14/F\_14\_34\_robot.c).

## SUMMARY

This chapter presented three capstone projects that use the PIC18 hardware features covered in Chapters 8 through 13. The audio project is small enough that it can be performed within an instructional laboratory environment. The home mon-

itoring and autonomous robot applications both require a considerable time investment and are better suited for multi-week project assignments. The home monitoring application introduced two topics not covered in previous chapters: the DS1621 Digital Thermometer, and storing data in the on-chip Flash program and Data EEPROM memories.

## **SUGGESTED PROJECT MODIFICATIONS**

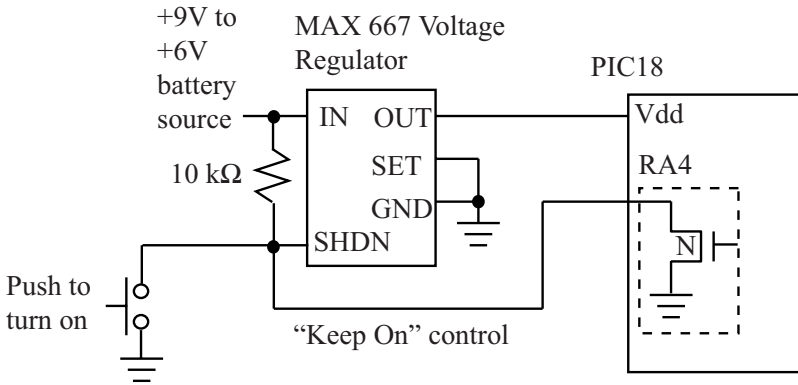
---

Instead of review problems, this section contains suggested modifications for the three capstone projects given in this chapter.

1. For the audio sampling application, implement a simple compression algorithm in which the difference between successive samples is recorded as a 4-bit two's complement value instead of an absolute 8-bit value. This doubles the amount of audio that can be saved for a particular sampling frequency at a quality cost. How does the audio quality compare with the previous method?
2. The I<sup>2</sup>C bus is the bottleneck for the playback function of the audio sampling application. Replace the MAX517 serial DAC with a parallel DAC such as the MAX 507. What is the fastest achievable sampling rate now? Is playback still the limiting factor for sampling rate?
3. For the home monitoring application, use the thermostat function of the DS1621 and the TOUT output to add a temperature trigger to the monitoring system. Modify the code to allow entry of TEMP HIGH and TEMP LOW values that are written to the DS1621 TH/TL registers. If the TOUT output is asserted, sound an audible alarm.
4. The code for the home monitoring application sounds an audible alarm if motion is detected between the alarm start and stop times. The alarm is only cancelled when a pushbutton is activated. The alarm produces a significant current drain on the battery and does not need to be constantly sounding. Add two user-modifiable times that specify alarm duration and interval; if motion is detected, the alarm should sound for the specified duration at the indicated interval (e.g., produce an audible alarm for 10 seconds every 2 minutes). The PIC18 should sleep as is currently done if the alarm is not sounding. These values should be stored in Data EEPROM and user-modifiable in the same manner as the other times.
5. Modify the robot application so that the VCR numeral #6 causes the robot to traverse a figure-8 pattern. Test this in a large room without any obstructions.



6. Buy an off-the-shelf robot from a hobbyist Web site and replace its “brain board” with a PIC18 board similar to the design presented here. Consider adding sound to it so that it makes noises as it putters around, perhaps playing your school’s fight song!
7. For a project that is battery operated such as a robot, implement a push-to-turn-on mechanism using a MAX667 voltage regulator as shown in Figure 14.39. The SHDN input must be low for the MAX667 to provide 5 V; activating the push button switch grounds the SHDN input and provides 5 V to the PIC18. The open-drain output RA4 is then asserted by the PIC18 to keep the SHDN input pulled low. The PIC18 can turn off power to itself by negating the RA4 output; this would typically be done to prolong battery life if no activity is detected for some pre-determined time period. If you are using this mechanism with the serial bootloader, the pushbutton must be kept activated during the entire programming operation.



**FIGURE 14.39** Push-to-turn-on power control.

# 15

## Beyond the PIC18Fxx2

### In This Chapter

- External Memory Interfacing
- Other PIC Family Members
- Bus Arbitration in I<sup>2</sup>C
- The Controller Area Network (CAN)
- The Universal Serial Bus (USB)
- A Brief Survey of Non-PIC Microcontrollers
- Real Time Operating Systems

This chapter surveys microcontrollers other than the PIC18Fxx2 from Microchip and other companies. Advanced interface standards such as the Controller Area Network (CAN) and the Universal Serial Bus (USB) are briefly examined, as well as issues concerning external memory interfacing and multi-master busses.

### 15.1 LEARNING OBJECTIVES

---

After reading this chapter, you will be able to:

- Discuss the roles of static random access memory (SRAM) and dynamic random access memory (DRAM) in microprocessor systems.

- Compare and contrast features of other PICmicro family members with the PIC18Fxx2.
- Compare and contrast features of microcontrollers from other semiconductor manufacturers with the PIC18Fxx2.
- Compare and contrast features of the Controller Area Network (CAN) and Universal Serial Bus (USB) with serial interface standards used by the PIC18Fxx2.
- Discuss bus arbitration for multi-master buses and how bus arbitration is accomplished for CAN and I<sup>2</sup>C busses.
- Discuss the basic concepts behind real time operating systems.

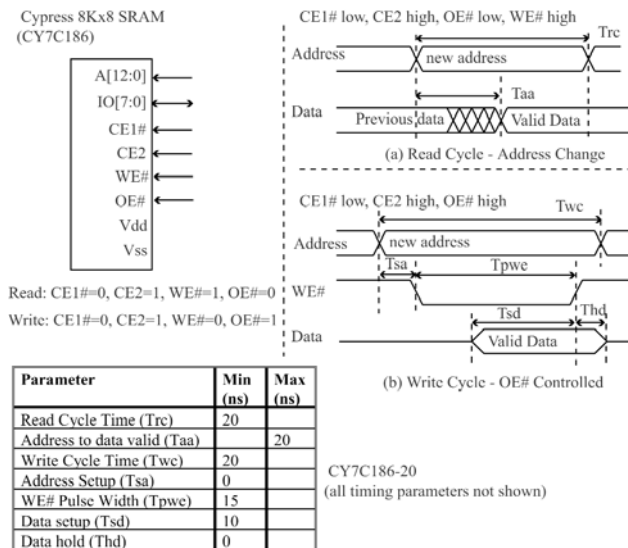
## 15.2 EXTERNAL MEMORY INTERFACING

---

Our discussion of memory technologies to this point has been limited to the non-volatile on-chip PIC18Fxx2 Flash program/Data EEPROM memories and off-chip memory using serial Flash EEPROMs. Two other common memory types are Static Random Access Memory (SRAM) and Dynamic Random Access Memory (DRAM). Both types are volatile and have read/write times that are equal, with read/write times ranging from a few nanoseconds to a few 10s of nanoseconds. Note that this is very different from the EEPROM memory covered previously in which write times are in the millisecond range because of its nonvolatile nature. SRAM and DRAM memories are polar opposites in terms of application. SRAM is optimized for speed first and density second, providing high-speed read/write storage capability for small microprocessor ( $\mu$ P) and microcontroller ( $\mu$ C) systems. An SRAM interface has low complexity, meaning that it connects to a  $\mu$ P/ $\mu$ C with a minimum of “glue logic”; for example, external logic between the  $\mu$ P/ $\mu$ C and the memory device. DRAM is optimized for density first and speed second, and is used as the primary memory for high-performance microprocessor systems such as those that use Intel<sup>®</sup> Pentium<sup>®</sup> and AMD Athlon<sup>™</sup> microprocessors. Single-word accesses to DRAMs are relatively slow; DRAMs are optimized for block transfers in which groups of bytes are transferred between the DRAM and a microprocessor. Block transfers are optimized in DRAMs because high-performance microprocessors have on-chip high-speed SRAM memory referred to as *cache memory* that services most of the instruction/data fetch needs of the  $\mu$ P. Blocks transfers (transfers of multiple bytes) between DRAM and the  $\mu$ P are used to fetch large blocks of closely related instructions/data when needed instructions/data cannot be located in the on-chip  $\mu$ P cache memory. Both SRAMs and DRAMs have synchronous versions that add clock signals to their interfaces—SSRAMs are synchronous SRAMs, while SDRAM, DDR/DDR2-DRAM (double data rate DRAM), and

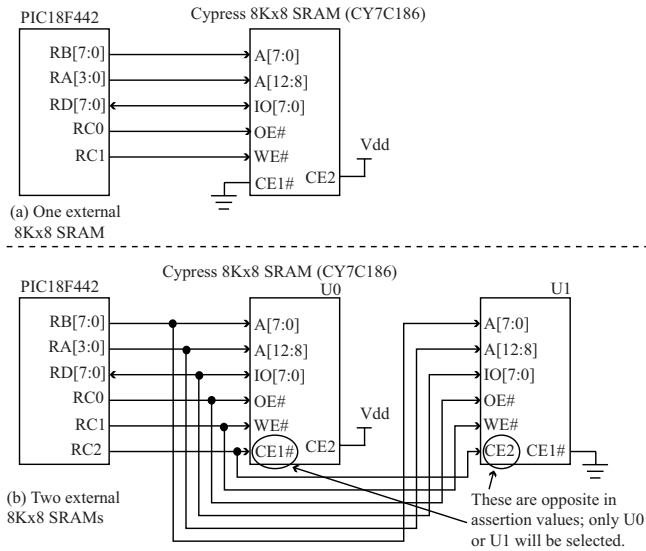
RDRAM (Rambus DRAM) are all forms of synchronous DRAM. Asynchronous DRAM is largely obsolete, as synchronous DRAM provides higher throughput for block transfers. DRAM interfacing is complex and is handled by chipset logic that connects high-performance microprocessors to the memory, IO busses, and external devices that are common in today’s personal computer platforms. Some high-performance 32-bit microcontrollers and digital signal processors include on-chip SDRAM controllers that eliminate glue logic between external SDRAM and the processor. In this section, we concentrate on asynchronous SRAM, as that is the most common memory type used for memory expansion in small  $\mu\text{P}/\mu\text{C}$  systems. DRAM interfacing is not discussed further in this book.

Figure 15.1 shows the interface and basic read/write timing of a Cypress Semiconductor 8Kx8 SRAM. The *chip select* signals CE1# and CE2 must be asserted (CE1# = 0, CE2 = 1) for the device to be active during a read or write. The WE# input selects whether a read (WE# = 1) or a write (WE# = 0) operation is performed. In a read operation, the SRAM outputs the contents of the location selected by the 13-bit address input A[12:0] on data pins IO[7:0] if the output enable pin (OE#) is asserted and the chip is selected. In a write operation, the data pin contents (IO[7:0]) are written to the location selected by the 13-bit address input A[12:0] when the write enable input (WE#) is pulsed low and the chip is selected; the input data is latched by the SRAM on the rising edge of the WE# signal. The timing diagrams shown in Figure 15.1 are a subset of the read/write modes and timing parameters found in the CY7C186 datasheet [24].



**FIGURE 15.1** Cypress CY7C186 8Kx8 SRAM.

The CY7C186 can be used to augment the internal 4K data storage of a PICF18442 as shown in Figure 15.2a. A PIC18F442 has 40 external pins and has D and E parallel ports in addition to the A, B, and C ports found in the PIC18F242. In Figure 15.2a, ports A and B are used for address, port D for data, and port C for the WE# and OE# pins. Performing a read from the CY7C186 involves placing the address on ports A/B, configuring port D for input, and then asserting the output enable low (OE#, port RC0) while keeping the write enable (WE#, pin RC1) high. A write to the CY7C186 is accomplished by placing the address on ports A/B, configuring port D for output, placing the write data on pins D[7:0], and then pulsing the write enable low (WE#, pin RC1) and keeping the output enable (OE#, port RC0) high. Observe that both chip enables (CE1#, CE2) of the CY7C186 in Figure 15.2a are always asserted by strapping them to ground and Vdd, respectively.



**FIGURE 15.2** PIC18442 to Cypress CY7C186 8Kx8 SRAM.

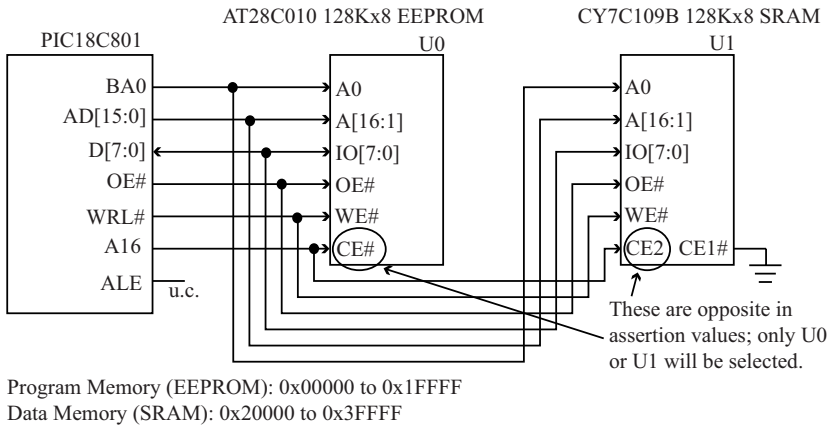
Figure 15.2b shows how a second CY7C186 is added by using output RC2 to drive the CE1# input of one SRAM (U0) and the CE2 input of the other SRAM (U1). This means that the U0 SRAM is selected when RC2 is low, while the U1 SRAM is enabled when RC2 is high. This illustrates the usefulness of multiple chip select inputs on SRAMs by providing flexibility in designing the enable logic for systems with multiple SRAM chips.

## An External Memory Bus

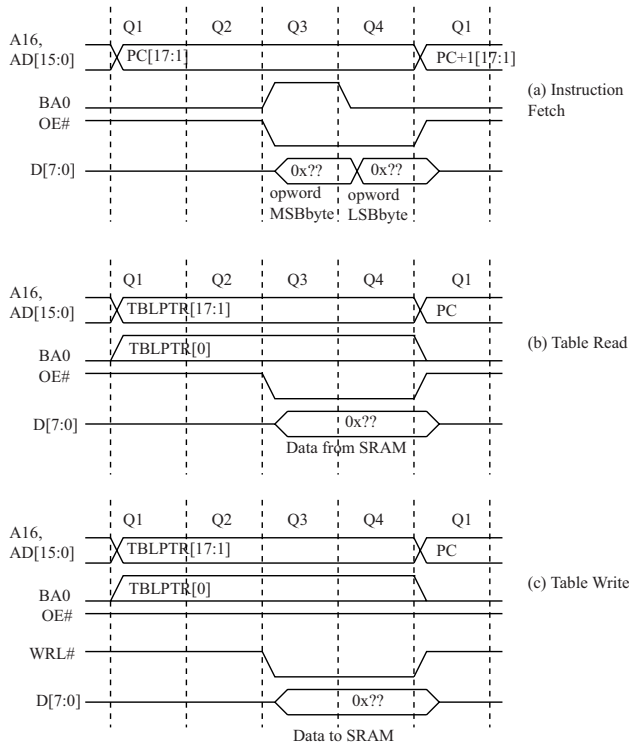
The fast read/write cycle times of the CY7C186 are essentially wasted in the system of Figure 15.2, as it takes several instruction cycles to perform a read or write to the external memory due to the necessity of manipulating several port registers. The PIC18Fxx2 family is not designed to implement an external SRAM memory interface in an efficient manner. So, is this the case with all PIC microcontrollers? The answer is no; several PIC  $\mu$ C family members have an *external bus* feature that allows efficient and flexible interfacing to external memories with parallel interfaces. One PIC  $\mu$ C with an external bus interface is the PIC18C801 *ROM-less* microcontroller [25]; it has no on-chip program memory and fetches all instructions from external memory. It supports 1.5K of internal data memory but can also use its external bus interface for data memory reads/writes to external SRAM using table read/write instructions. The 18C801  $\mu$ C comes in either 80- or 84-pin packages and supports up to 2 MB of external memory.

Figure 15.3 shows an interface between a PIC18C801 and two external memory devices: an Atmel 128Kx8 EEPROM [27] and a Cypress 128Kx8 SRAM [26]. The interface shown is the 8-bit *de-multiplexed* mode in which address and data are placed on separate pins (address on A[17:16], AD[15:0] and data on D[7:0]). The BA0 output is the least significant bit of the address. The OE# line is driven low for data transfer during a read operation; the WRL# is driven low for data transfer during a write operation. The ALE (address latch enable) output is used in the multiplexed mode in which address and data are sent in different Q clock cycles over AD[15:0]; this pin is not used in the de-multiplexed mode. Observe that PIC address pins {AD[15:0], BA0} form the 17-bit address needed for the 128Kx8 memory devices ( $128K = 128 \times 1024 = 2^7 \times 2^{10} = 2^{17}$ , so a 17-bit address is required). The PIC address pin A16 drives the CE# input of the EEPROM and the CE2 input of the SRAM. Hence, the EEPROM is selected for addresses 0x00000–0x1FFFF while the SRAM is selected for addresses 0x20000–0x3FFFF.

Figure 15.4 shows the external bus operation of the PIC18C801 for opcode fetch, table read, and table write. The instruction fetch (Figure 15.4a) must read a 16-bit instruction word from external memory in one instruction cycle; it does this by fetching the most significant byte in Q3 and the least significant byte in Q4 of the instruction cycle. During these read operations, the external address lines A16/AD[15:0] contain the program counter bits PC[17:1], while BA0 specifies the least significant bit of the address. The OE# line is asserted during Q3 and Q4 so that the external memory drives the data pins D[7:0].



**FIGURE 15.3** PIC18C801 to Atmel 128x8 EEPROM and Cypress CY7C109B 128Kx8 SRAM.



**FIGURE 15.4** PIC18C801 external bus operation.

For noninstruction fetches, a table read or write is the method by which data access is accomplished via the external bus. Figure 15.4b shows that during a table read, the external address lines A16/AD[15:0] contain the TBLPTR register bits [17:1], while BA0 specifies the least significant bit of the address. The OE# line is asserted for both Q3 and Q4 cycles in a table read. In a table write (Figure 15.4b), the address is presented in the same manner as in a table read, but the WRL# is asserted during Q3 and Q4 while OE# is negated and the data pins D[7:0] are driven by the PIC18C801 during the same cycles.

The PIC18C801 also supports an 8-bit multiplexed mode (uses less external pins than the de-multiplexed mode) and a 16-bit mode (useful for slower memory); see the datasheet [25] for details.

## 15.3 OTHER PIC FAMILY MEMBERS

The previous section introduced a new PIC microcontroller, the PIC18C801. Microchip has several microcontroller families spanning a wide price/performance range to target as many different applications as possible. Figure 15.5 gives a partial summary of the PIC microcontroller families offered by Microchip; this list is not inclusive and some features are truncated for space reasons (see the Microchip Web site for complete details).

Feature	PIC10	PIC12	PIC14	PIC16	PIC17	PIC18	dsPIC®
instr. width	12	12,14	14	14	16	16	24
data width	8	8	8	8	8	8	16
# of instr.	33	35	35	35	58	76	84
Pgm Words	256 to 512	512 to 2K	4K	512 to 8K	4K to 16K	2K to 64K	4K to 48K
SRAM data (bytes)	16 to 24	25 to 128	192	25 to 368	232 to 902	768 to 3968	512 to 8K
Data EEPROM	0	0 to 256	0	0 to 256	0	0 to 1024	1K to 4K
IO	4	6	20	6 to 36	33 to 66	16 to 72	12 to 68
Analog	Cmptrr.	ADC, Cmptrr.	Cmptrr.	Cmptrr, ADC	ADC	ADC, (2) Cmptrrs	ADC
Interface			I <sup>2</sup> C	I <sup>2</sup> C, USART, SPI, low speed USB, LCD	I <sup>2</sup> C, USART, SPI	I <sup>2</sup> C, USART, SPI, CAN, LCD	I <sup>2</sup> C, USART, SPI, CAN, CODEC
Timers	(1) 8-bit, (1)WDT	(0/1/2) 8-bit, (0/1) 16-bit, WDT	(1)8-bit, (1) 16-bit, WDT	(0/1/2) 8-bit, (0/1/2)16-bit, WDT	(2) 8-bit, (2)16-bit, WDT	(0-3)8-bit, (1-3)16-bit, WDT	(3-5) 16-bit, WDT
Max CLK (MHz)	4	20	20	40	33	40	30
pkg pins	6, 8	8	28	14,18,20, 28	40,44,64, 68,80,84	28,40,44, 64,68,80, 84	18,28,40, 44,64,80

**FIGURE 15.5** PIC microcontroller family summary.

The PIC10 family has only four members (PIC10F200/2/4/6) and is targeted at the extreme low-end microcontroller market. It comes in either 6-pin (SOT-23) or 8-pin packages (8 PDIP), can be operated by an internal 4 MHz clock or external



clock source, has an 8-bit timer, a maximum of 4 I/O pins, and a maximum of 512 instruction words and 24 bytes of SRAM. Two family members include an analog comparator. It is the embodiment of a “tiny” microcontroller and is priced in proportion to its power.

The PIC12 family has 18 members and its differentiator from the PIC10 family is an on-chip A/D (4-bit, 8-bit, 10-bit) and more on-chip memory, both program and data.

The PIC14 family has only one member (PIC14000). Its differentiator from previous lower end families is that it has onboard analog components to create a slope conversion type A/D converter and an I<sup>2</sup>C port. It also includes two comparators.

The PIC16 family is the most versatile family after the PIC18 line. It has 130+ members, and the PIC16F873 is pin-compatible with the PIC18F242. It supports all of the same on-chip peripherals that the PIC18 family does, except for the Controller Area Network module (discussed in the next section). One PIC16 family member even supports a low-speed Universal Serial Bus module. The principle differentiator from the PIC16 and the PIC18 family is the instruction set architecture. The PIC16 does not have OV, N flags and thus signed comparisons, especially multi-byte comparisons, are cumbersome to implement. The PIC16 does not have the 8x8 multiply unit or any of the branch instructions; all conditional execution is implemented using the “test and skip next instruction” approach. Also, the PIC16 implements a maximum of four banks of data memory, each with 128 locations versus the 16 banks (maximum) of 256 locations support by the PIC18 architecture. The PIC16 does not have a `movff` instruction that allows a single instruction transfer between banks; any bank-to-bank transfer on the PIC16 must manipulate the bank select bits. The PIC16 has only one FSR register instead of the three registers (FSR0/1/2) implemented in the PIC18, and has none of the post, pre, increment, decrement addressing modes of the PIC18. The PIC16 also does not have the table read/write operations. These instruction set differences are summarized in Table 15.1.

The PIC17 family has only 10 members, with three members at end-of-life at the time of this writing. This was Microchip’s first high-performance microcontroller family, but has been largely supplemented by the PIC18 and dsPIC® families. Additional capability over the predecessor PIC16 family was an 8x8 hardware multiplier, an external memory interface, and new instructions such as table read/write.

The PIC18 family has 86 family members and is actively growing. Two additional hardware modules not supported in the PIC18Fxx2 but found in other PIC18 family members support the Controller Area Network bus, discussed in Section 15.5, and the Universal Serial Bus, covered in Section 15.6.

**TABLE 15.1** Instruction Set Architecture: PIC16 versus PIC18

<b>Feature</b>	<b>PIC16</b>	<b>PIC18</b>
Data Memory	4 banks/128 locations each	16 banks/256 locations each
movff instruction?	no	yes
conditional execution	test/skip next instruction	test/skip next instruction, flag branches
Hardware Stack	8 levels	31 levels
Flags	Z, DC, C	N, V, Z, DC, C
Hardware Multiply	none	8x8
Table Read Instructions TBLRD(*/*+/*-/+*) TBLWT(*/*+/*-/+*)	none	TBLRD(*/*+/*-/+*)
FSR Registers	1 (FSR/INDF)	3 (FSRx, INDFx, POSTDECx, POSTINCx, PREINCx)

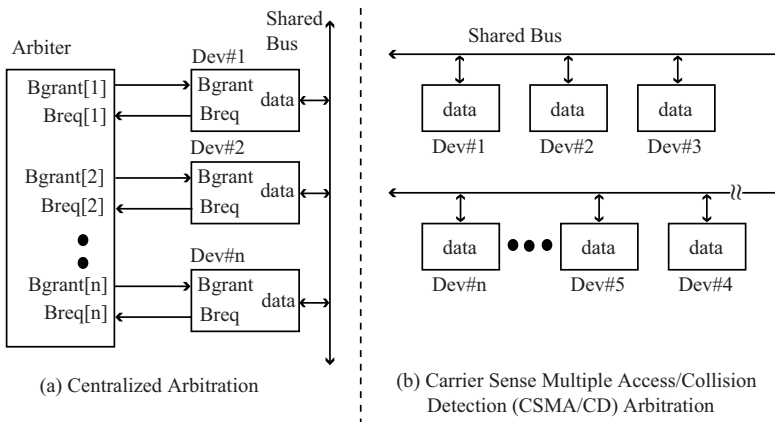
The dsPIC<sup>®</sup> is optimized for digital signal processing applications; in other words, for processing streaming data such as digitized audio. The instruction word width is 24 bits and the internal data path is 16 bits; all other previous PIC family members have an internal 8-bit data width. The advantage of increasing the internal data width from 8 bits to 16 bits is obvious; a 16-bit addition on a dsPIC<sup>®</sup> takes one instruction cycle instead of two instruction cycles. All internal data registers and data memory on the dsPIC<sup>®</sup> are 16 bits wide. DSP applications are computationally intensive and the dsPIC<sup>®</sup> arithmetic features have been significantly enhanced over previous PIC family members. Arithmetic support includes a hardware signed/unsigned divider capable of 32/16, 16/16 operations (see Chapter 7, “Advanced Assembly Language: Higher Math”). A separate DSP engine is included in addition to a 16-bit ALU; the DSP engine has a signed/unsigned 17x17 hardware multiplier, a 40-bit adder/subtractor with two accumulators and saturation logic (see Chapter 7), and a barrel shifter for multiple-position shifts in a single instruction cycle. A multiply-accumulate instruction that performs a multiplication and then adds the result to an accumulator register is included. DSP instructions use the DSP engine, while MCU instructions flow through the 16-bit ALU. Branch instructions have been enhanced to include signed branches that allow direct comparison of signed operands with the branch condition based on combinations of the OV/N/Z flags. Peripheral enhancements include a Quadrature Encoder Interface (QEI) module that provides hardware support for decoding rotary encoder inputs (quadrature inputs). Quadrature encoder logic decodes quadrature inputs and

provides a direct increment/decrement signal to a 16-bit up/down counter. The QEI module includes a programmable digital noise filtering feature that does not update quadrature inputs unless they have been stable for three consecutive clock cycles. An enhanced PWM module has six PWM channels. Other hardware features are two USART channels, I<sup>2</sup>C interface, a 12-bit A/D, SPI interface, a Data Converter Interface (DCI), and a CAN interface. The DCI supports automated synchronous serial transfer to external audio coder/decoders (codecs), ADCs, and DACs.

### 15.4 BUS ARBITRATION IN I<sup>2</sup>C

The previous section mentioned the Controller Area Network (CAN) as another interface standard supported by PIC microcontrollers. Before examining the CAN standard, the more general topic of *bus arbitration* is covered, as it is applicable to both CAN and the previously discussed I<sup>2</sup>C bus. In Chapter 12, “Data Conversion,” it was stated that the I<sup>2</sup>C bus supports multiple bus masters (*multi-master*); in other words, any I<sup>2</sup>C device on the bus can initiate a transaction. However, in our examples, we always assumed that the PIC was the sole bus master and the PIC initiated all I<sup>2</sup>C transactions. If a bus supports multiple bus masters, there must be a *bus arbitration* mechanism that decides which device assumes control of the bus when there are simultaneous attempts by different bus masters to access the bus.

Figure 15.6 shows two common methods of bus arbitration for multi-master buses. In centralized arbitration (Figure 15.6a) a device wishing to communicate on the shared bus requests permission to access the bus via a bus request (*breq*) signal to an *arbiter*, which grants the device the bus via the bus grant (*bgrant*) line.

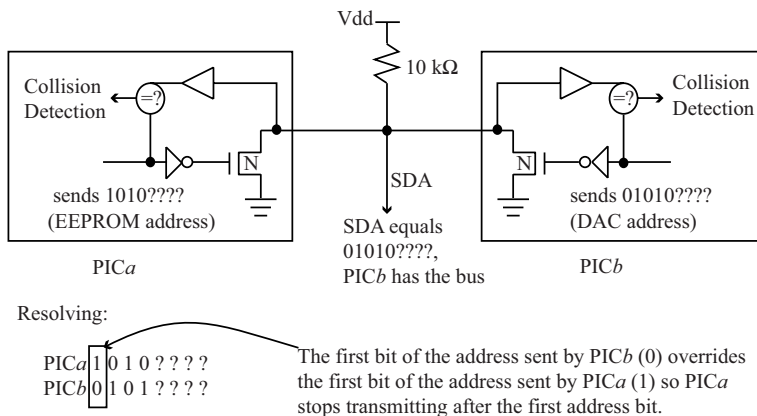


**FIGURE 15.6** Two methods of bus arbitration.

In the case of simultaneous requests, the arbiter uses a *priority scheme* to choose which device is granted bus access. A *fixed priority* scheme has static priorities assigned to each device; device #1 always has the highest priority and device #*n* the lowest. A fixed priority scheme can result in one device monopolizing the bus, so a rotating priority scheme is more common, in which priorities are dynamically rotated among devices in an attempt to provide equal access to the bus. The disadvantage with centralized arbitration is that each device on the bus must have its own pair of bus request/bus grant lines. Centralized arbitration is most common in backplane busses found within computer systems that have a fixed number of IO slots, and hence, a fixed number of devices that can be present on the bus.

Figure 15.6b shows an arbitration scheme called *Carrier Sense Multiple Access/Collision Detection (CSMA/CD)*, which is useful when it is unknown a priori the number of devices that will be connected to a bus. In CSMA/CD, a device wanting bus access waits until the bus is idle, and then begins transmitting. If multiple devices transmit, there will be a data *collision* on the bus. A device must be able to sense a collision, and then determine independently of the other devices what action to take. A transmitter detects a collision by sensing the bus state during transmission; if the bus state does not match what the transmitter is sending, a collision has occurred. Local area networks based on Ethernet use CSMA/CD; when a collision occurs all transmitters stop sending data, wait for a random interval (the *back-off interval*), and try again. If another collision occurs, the interval wait time is increased (typically doubled), and another random wait is done. This continues until a transmitter is successful at gaining access to the bus. While this works, it also wastes time because of the need for all transmitters to wait for the random interval.

The I<sup>2</sup>C bus also uses CSMA/CD for arbitration, but resolves conflicts in a manner different from Ethernet. Figure 15.7 illustrates how arbitration is performed on the I<sup>2</sup>C bus. Assuming both PIC*a* and PIC*b* begin transmitting at the



**FIGURE 15.7** I<sup>2</sup>C bus arbitration.

same time, the first data sent after the start condition is the address byte of the I<sup>2</sup>C slave device. Each device also transmits a clock signal over the SCL line in addition to driving the SDA signal as each device is initiating a transfer. A “0” on the SDA bus overrides a “1” because of the open-drain output used to drive SDA (and also SCL). Each device senses the SDA line during transmission; if a device detects that the SDA state is different from what it has transmitted, the device immediately ceases transmission, giving up the bus. In this example, PIC<sub>a</sub> is initiating a transfer to an EEPROM (address 1010????), while PIC<sub>b</sub> is beginning a transfer to a DAC (address 0101????). PIC<sub>a</sub> stops transmitting after it sends the first (most significant) bit of its address because the initial “0” sent by PIC<sub>b</sub> overrides the “1” sent by PIC<sub>a</sub>. Observe that PIC<sub>a</sub>’s transmission did not disturb PIC<sub>b</sub>’s transmission, so no data is lost and no time is wasted by the arbitration. What if both PIC<sub>a</sub> and PIC<sub>b</sub> attempt to write to the same device? Because the address bits are the same, the arbitration continues through the data bits, until either some difference is detected or the transaction completes if both devices send exactly the same data. At this point, you should now understand the reason for the pullup resistors on the I<sup>2</sup>C bus; the drivers for the SDA/SDL lines are open-drain to permit multi-master bus arbitration by having a “0” state override a “1” state. This conflict resolution in CSMA/CD is simple and effective, and is used in other CSMA/CD busses as will be seen when the Controller Area Network is discussed in the next section.

One other method of sharing a bus by multiple devices that can initiate transactions is *time domain multiplexing* (TDM), which sidesteps the issue of arbitration by assigning fixed time intervals to devices for bus access. This means that each device on the bus must keep an accurate track of time in order to track its assigned bus access slot. While no arbitration is needed in this scheme, it is not an efficient use of bus bandwidth because if a device has nothing to transfer over the bus during its assigned time, that bus bandwidth is unused.

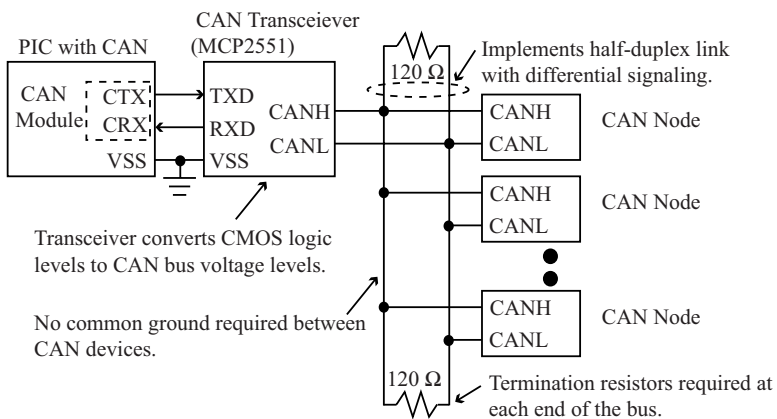
## 15.5 THE CONTROLLER AREA NETWORK (CAN)

---

The automotive market is important for microcontroller manufacturers, as a typical car or truck has in the ten’s of microcontrollers within it. The number of microcontrollers within vehicles keeps increasing as automobiles evolve to mobile computing platforms that also happen to carry people between locations. An automobile is a harsh environment from an electrical noise perspective and contains electrical systems dispersed through the vehicle with communication distances measured in meters. CAN [28] is a half-duplex serial bus designed as a communication mechanism for intelligent peripherals within an automotive system. CAN’s signaling mechanism is designed to combat the inherent electrical noise found within vehicles. The CAN bus only uses two wires for communication, keeping

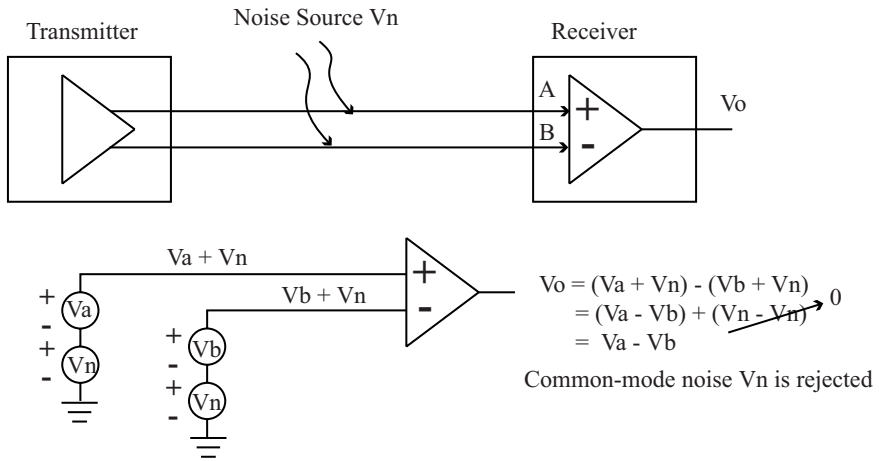
electrical cabling size to a minimum, thus making it easier to route within the crowded compartments of an automobile. CAN is a true bus in the formal sense like I<sup>2</sup>C; CAN transactions are visible to all peripherals connected to the bus and each transaction includes an 11-bit identifier that is used by receivers to filter messages. The CAN bus is multi-master in that any node on the bus can initiate a transaction, with arbitration handled similarly to I<sup>2</sup>C arbitration.

Figure 15.8 shows a PIC with an internal CAN module connected to a CAN bus. A CAN bus is implemented as two wires, CANH/CANL, which uses *differential signaling* (discussed in Figure 15.9) to form a half-duplex communication channel.



**FIGURE 15.8** PIC/CAN system.

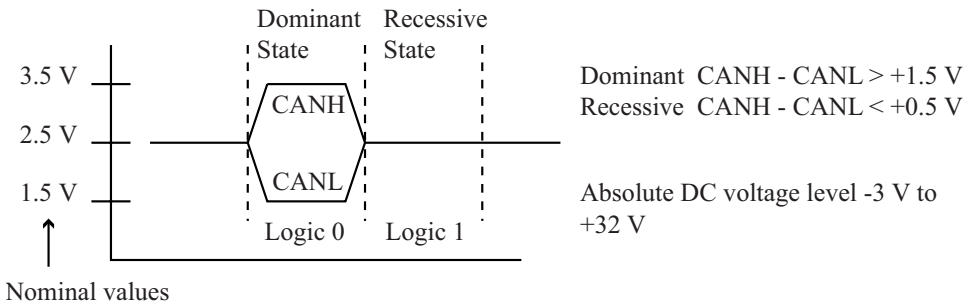
A transceiver chip like the MCP2551 is required to convert from CAN bus voltage levels to CMOS logic levels (similar to the MAX232 chip for the RS232 standard). A PIC CAN module has separate transmit (CTX) and receive (CRX) pins that are multiplexed by the CAN transceiver onto CANH/CANL. The CANH/CANL wires implement differential signaling; a pair of wires is used to signal a logic state, either “0” or “1”. To this point, all signaling methods discussed in this book have been single-ended; one wire is used to signal a logic “0” or “1”. The disadvantage of differential signaling is that it doubles the number of wires needed and as such is primarily used for serial transfers. The advantage of differential signaling is *common-mode noise rejection* as shown in Figure 15.9.



**FIGURE 15.9** Common-mode noise rejection in differential signaling.

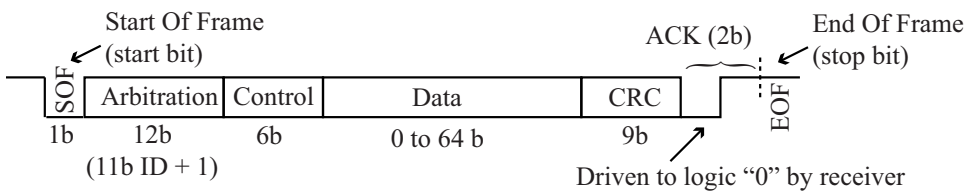
*Common-mode noise* is noise that is injected equally (or nearly so) onto all wires within a cable. A CAN bus within an automobile can be relatively lengthy and has ample opportunity to pick up noise from neighboring cable bundles or from other nearby systems. Any common-mode noise is rejected at the receiver as the two input signal voltages are subtracted from each other to form the received voltage. Differential signaling is commonly used in external cabling that carries high-speed signals outside of a computing system.

Non-return-to-zero (NRZ) asynchronous transmission is used on the CTX/RTX pins that connect the PIC to the transceiver of Figure 15.8. The differential signaling method used on the CANH/CANL wire pair is shown in Figure 15.10. A logic “1” is called the *recessive state*, and is defined as when  $CANH - CANL < +0.5 \text{ V}$ . A logic “0” is called the *dominant state*, and is defined as when  $CANH - CANL > 1.5 \text{ V}$ . The recessive state (logic “1”) is the bus idle state. The dominant state (logic “0”) overrides the recessive state (logic “1”); if one transmitter sends a “0” and a second transmitter sends a “1”, the bus will contain a “0” state (the dominant state). Absolute DC voltage levels can range from  $-3 \text{ V}$  to  $+32 \text{ V}$ , and CAN transceivers such as the MCP2551 must be able to survive transient voltage surges of  $-150 \text{ V}$  to  $+100 \text{ V}$ . Data rates range from 10 Kbs to 1 Mbs with the maximum data rate limited by the CAN bus length. On a CAN bus, all CAN nodes must agree on the data rate. The physical signaling shown in Figure 15.10 is not defined by the CAN 2.0 standard [28], but rather by the ISO-11898 specification that was created to ensure compatibility between CAN nodes in an automotive system. This means that the CAN protocol can be used with different physical signaling methods as long as the CAN 2.0 specifications are met.



**FIGURE 15.10** CANH/CANL differential signaling.

Data transmissions are sent in frames, with each frame split into fields, and with each field containing 1 or more bits as shown in Figure 15.11. There are six different frame types: standard data, extended data, remote, error, overload, and interframe space. A standard data frame is shown in Figure 15.11. The start-of-frame bit (start bit) is a logic “0” that signals the beginning of a frame. The arbitration field contains an 11-bit ID and a 12<sup>th</sup> bit called the RTR bit, which is “0” for a data frame and “1” for a remote frame. An extended data frame has a longer identifier than a standard data frame. An identifier is not an address in the I<sup>2</sup>C sense; it does not have to uniquely identify either the sender or receiver. All nodes on the CAN bus receive the message; each node decides whether to act on the message based upon the value of the identifier and the contents of the message. The CAN module for PIC microcontrollers contains multiple filter/mask registers that are used to determine if a received frame should be accepted or rejected; these filter/mask registers use the ID field for accept/reject decisions. Once a frame has been accepted, it is transferred to an internal message buffer for further processing by the PIC application.



**FIGURE 15.11** A CAN standard data frame.

The ID field is used for multi-master arbitration in the same manner as I<sup>2</sup>C arbitration. A CAN node must wait until the bus is idle before attempting transmission. Multiple CAN nodes that simultaneously attempt to transmit monitor the CAN bus as the ID field is sent. A “0” state (dominant) overrides a “1” state (reces-

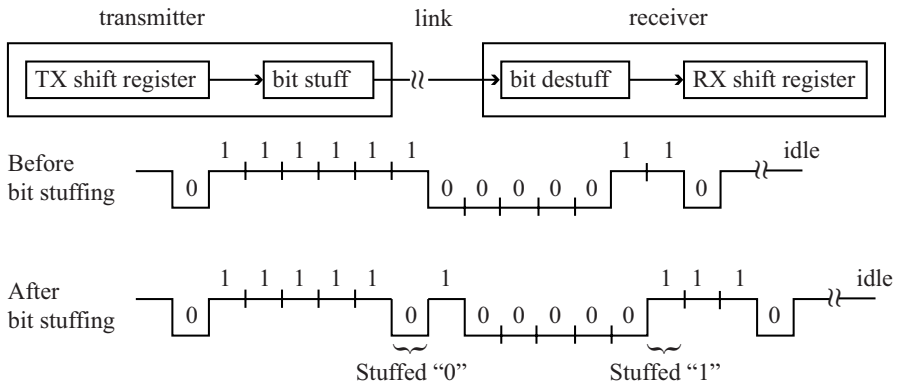


sive); a CAN module ceases transmission if it detects a difference between the CAN bus state and what it has transmitted. Like I<sup>2</sup>C bus arbitration, this results in no lost bus time or in any corrupted messages. The ID field is transmitted most significant bit to least significant bit so the message with the lowest numerical ID field wins during arbitration. This assigns a fixed priority to message identifiers. Arbitration is only performed on the ID field and the CAN specification does not define what occurs if two messages with the same identifier are sent simultaneously. As such, assignment of message identifiers within a CAN system must be done in such a way as to guarantee that simultaneous transmission of messages with the same ID does not occur (the CAN spec does define this case for a collision between a standard data frame and a remote data frame, but the RTR bit in the arbitration field determines priority in this case).

One strength of the CAN protocol is error detection. The Cyclic Redundancy Check (CRC) field is a checksum based on the SOF, arbitration, control, and data fields that can detect a number of different errors including five randomly distributed errors, any odd number of errors, and burst errors of less than 15 in length. The control field includes a message length so every frame received is also checked for the correct length, and each frame is also acknowledged by the receiver during the acknowledge field time (similar to the ACK bit in the I<sup>2</sup>C protocol, except this is for the entire frame).

Because of the large number of bits sent in one frame, there must be a mechanism that allows the receiver to remain synchronized to the bit stream or else cumulative error between transmitter and receiver clock mismatch will cause incorrect sampling of the received bits. This is accomplished through a technique known as *bit stuffing*, shown in Figure 15.12. Bit stuffing is done by the transmitter by adding an extra bit that is the complement of the preceding bit if it detects that 5 bits of the same value have been transmitted. This is done to guarantee that every 6-bit interval contains a data transition (a guaranteed *transition density*), which allows a Phase Locked Loop (PLL) or Digital Phased Locked Loop (DPLL) circuit at the receiver to synchronize the sampling clock to the bit stream.

The bit stuffing and bit destuffing is invisible to the user and is done automatically by the transmit and receive hardware. Figure 15.12 shows an example where both a “0” and “1” are added by the bit stuffing logic to the data stream. Observe that the “1” did not actually have to be added to the bit stream to guarantee a transition in 6-bit intervals as a “1” was present in the bit stream after the five “0” bits. However, the bit stuffing logic does not know this and so the “1” bit is stuffed into the bit stream anyway. *Bit destuffing* by the receiver is the opposite procedure; if 5 bits of the same value are received, the next bit is assumed to be a stuffed bit and is removed from the data stream. Bit stuffing is also useful for bit-level error detection, as a stuck-at-0 or stuck-at-1 failure in the transmitter causes the bit destuffing logic to detect an error in the received bit stream when the received bit value does



**FIGURE 15.12** Bit stuffing in CAN.

not match the expected polarity of a stuffed bit. The usage and formatting of the remaining frame types of remote frame, error frame, overload frame, and interframe spacing are not discussed and the reader is referred to the CAN specification [28].

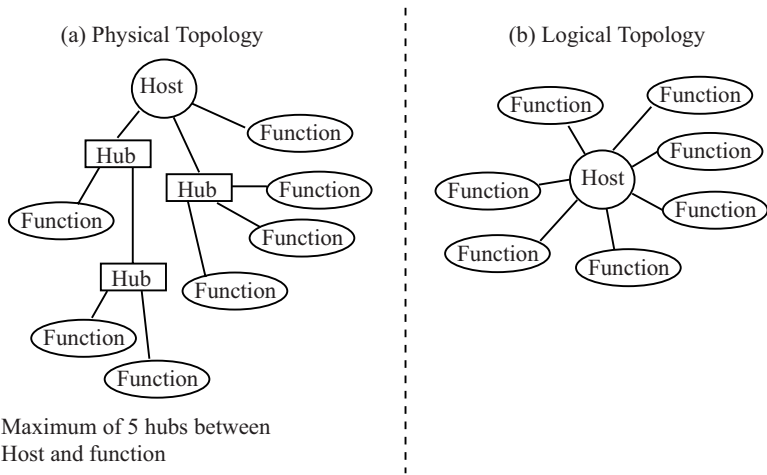
Significant hardware resources are required to implement a CAN interface, and the CAN modules used by PIC microcontrollers contain a large number of status, configuration, and data registers. PIC18 family members with CAN modules include the PIC18Fxx8 microcontrollers [29]; the user is referred to the datasheet of this family for more information.

## 15.6 THE UNIVERSAL SERIAL BUS (USB)

The Universal Serial Bus [30] is a high-speed serial protocol that has largely replaced the use of RS232 for serial communication on personal computers. Microchip provides the USB interface on some PIC18 family members such as the PIC18F2455/4455, and microcontrollers with USB interfaces are also available from other semiconductor companies. USB is a complex specification, and a complete discussion is beyond the scope of this book. USB is briefly summarized here to contrast and compare some of its features with the CAN protocol discussed in the previous section. USB supports data transfer speeds of 1.5 Mb/s (low-speed device), 12 Mb/s (full speed), and 480 Mb/s (high speed). Figure 15.13a shows that the physical topology of a USB network consists of the *host*, *hubs*, and *functions*. The host initiates all transactions in USB, as the bus does not support multiple bus masters. All communication is between the host and functions, which are USB-enabled devices such as keyboards, mice, speakers, memory cards, and so forth.

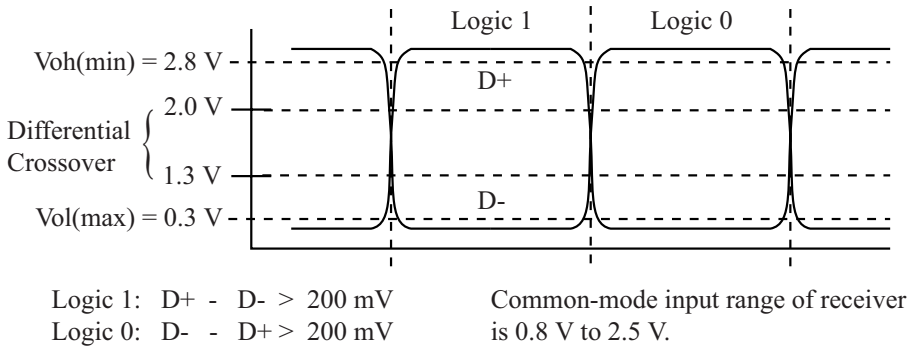
A hub simply provides a connection point to grow the physical topology; a hub has an upstream port that communicates up the hierarchy to the host and multiple

downstream ports that can either connect to a hub or function. A hub can also split the network into different speed regions with high-speed transfer upstream (host side) and either low-speed or full-speed downstream. Logically, each endpoint appears to be directly connected to the host as shown in Figure 15.13b.



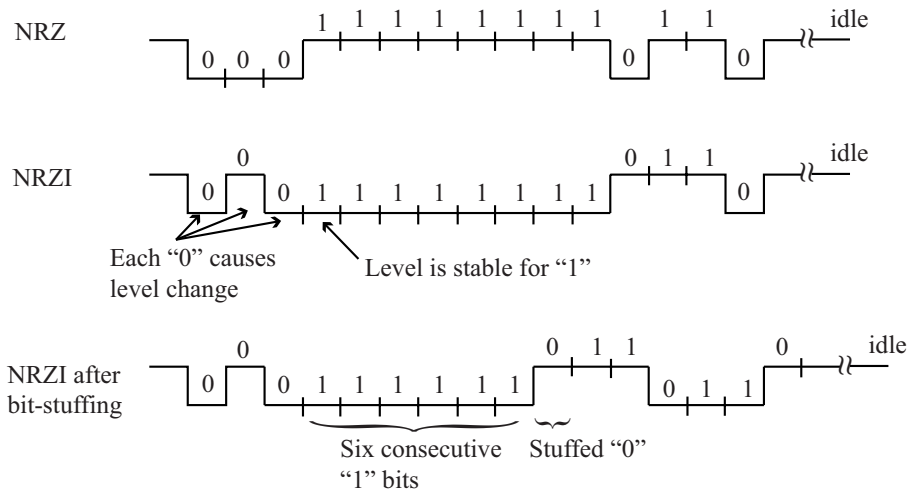
**FIGURE 15.13** USB physical and logical topologies.

At the physical level, USB and CAN have similarities in that both implement half-duplex communication using differential signaling, and both use bit stuffing to maintain synchronization. However, they differ in details relating to signaling levels and data encoding. Figure 15.14 shows the electrical signaling used in USB for low- and full-speed modes (the electrical signal levels for high-speed mode are significantly different and are not discussed).



**FIGURE 15.14** USB electrical signaling.

Data is carried by the D+/D− signal pair; in low- and full-speed mode a “1” is signaled when  $(D+) - (D-) > 200 \text{ mV}$  and a “0” signaled when  $(D-) - (D+) > 200 \text{ mV}$ . Differential signaling is used in USB for the same reason it is used in CAN—to make the signaling more resistant to common-mode noise. The USB cable also carries V<sub>dd</sub> and ground, allowing USB devices to maintain a common ground with the host and to be powered from the cable. The data encoding method used in USB is called Non Return to Zero Invert (NRZI), a somewhat unfortunate name as it is not the inverse of NRZ encoding. NRZI encoding changes signal level anytime a “0” is sent, while a “1” maintains the same signal level. This means that a string of “0”s causes the line to change with each bit, while a string of “1”s leaves the line quiescent. Bit stuffing is used in USB to force a guaranteed signal transition density as is done in CAN signaling; however, the NRZI encoding means that strings of “1”s trigger the bit stuffing mechanism. A “0” is inserted into the bit-stream by the transmitter when six consecutive “1” bits are detected. Figure 15.15 shows an example of NRZI encoding and the bit stuffing used within USB.

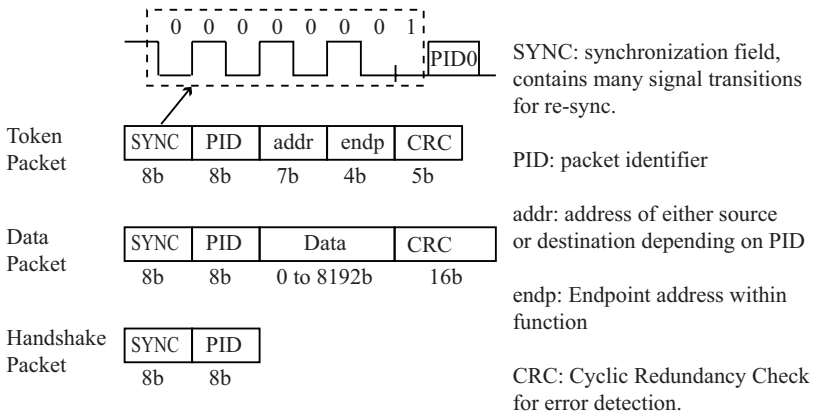


**FIGURE 15.15** NRZI encoding and bit stuffing in USB.

USB transactions occur in packets, with bytes within packets sent least significant bit to most significant bit. Common packet types are *token*, *handshake*, and *data* packets. Token packets identify the type of transaction, data packets contain the data being transferred between host and function, and handshake packets are used for acknowledgment, flow control, and error signaling. There are four types of USB transactions: *bulk*, *control*, *interrupt*, and *isochronous*. A bulk transaction transfers data between the host and function with a handshake packet sent for every data

packet, and also supports flow control and retry. A control transaction is also used for data transfer and allows transfer of multiple data packets without handshake packets, thus containing less packet overhead than a bulk transaction. An interrupt transaction allows for periodic polling of functions via a low overhead transaction. An isochronous transaction is intended for functions that require a guaranteed data bandwidth; real-time data delivery, such as audio or video data.

Figure 15.16 shows the formats of common packet types in USB. Every packet starts with a SYNC field (8 bits for low/full speed, 32 bits for high speed) used to re-synchronize the receiver to the data stream; the SYNC field contains a high density of signal transitions for this purpose. The packet identifier (PID) specifies the type of packet; Figure 15.16 shows only a subset of the available packet types within USB (not shown are so-called *special* packets). The address field of the token packet identifies the source or destination address of the transaction, depending on the packet identifier. Address 0 is the default address for a new device connection to the network and is reserved, which means that a USB network can contain 127 external functions. The host is responsible for assigning an address (*bus enumeration*) to a function when a device is connected to the network; USB devices can be dynamically removed and added to the network. The 4-bit endpoint field specifies a location within the function. The CRC field is used for error detection.



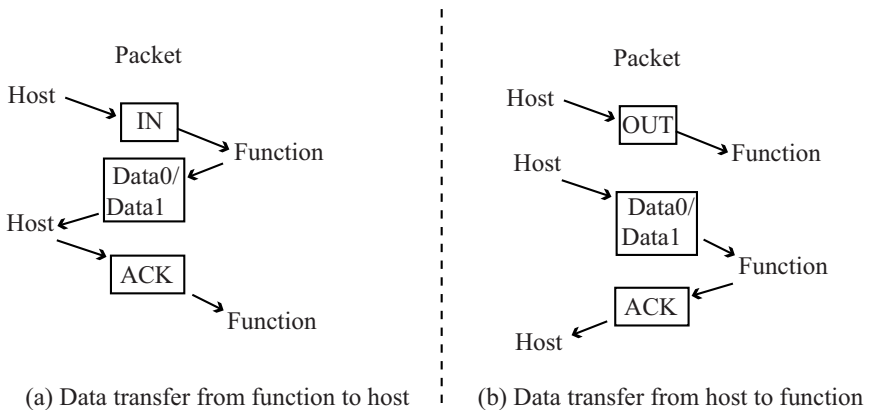
Token packet (IN, OUT, SOF, SETUP) used to initiate a transaction and identify a transaction type.

Data packet (DATA0/1/2, MDATA) used to transfer data between host and function.

Handshake packet (ACK, NAK, STALL, NYET) used for acknowledgement of packets, flow control, and error signaling.

**FIGURE 15.16** Common packet types/formats in USB.

Figure 15.17 shows data transfer from host to function and from function to host in a bulk transaction. A transfer from function to host consists of an IN token packet from the host requesting data; the function returns either a DATA0 or DATA1 data packet that the host acknowledges via an ACK handshake packet. The DATA0/DATA1 packets are alternated between transactions as a way of synchronizing multiple data transactions. A transfer from host to function consists of an OUT token packet from the host followed by either a DATA0 or DATA1 packet that the function acknowledges via an ACK handshake packet.



**FIGURE 15.17** Data transfer in a bulk transaction.

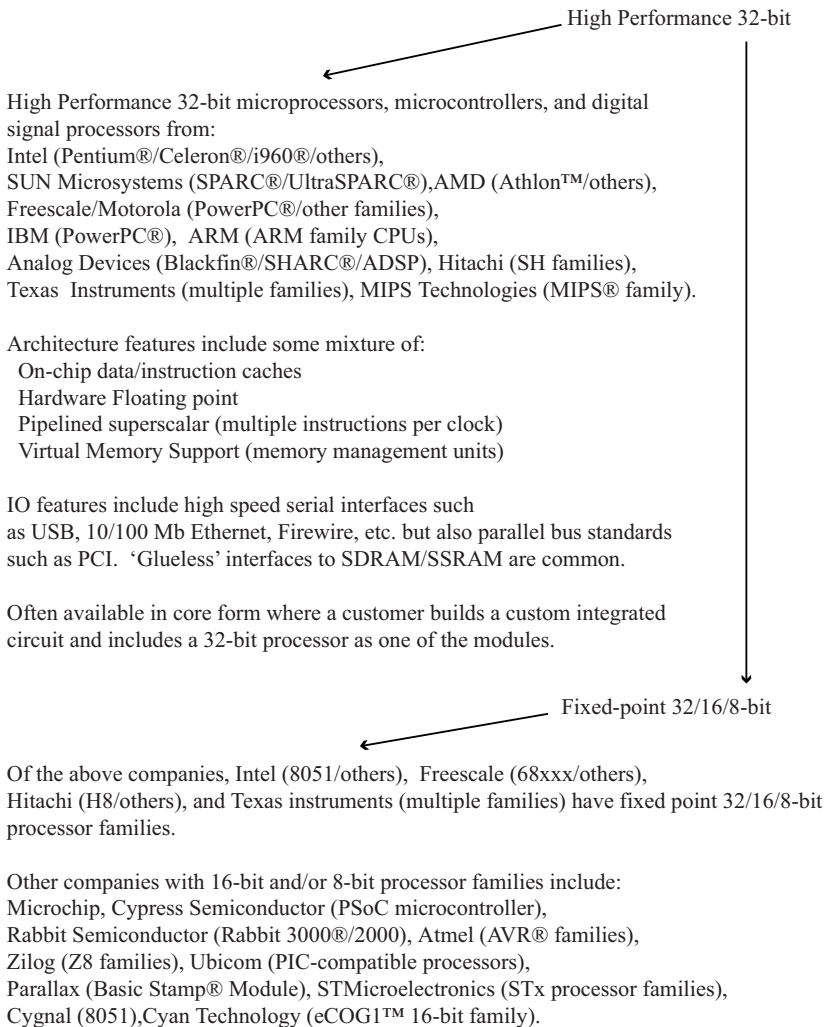
This discussion is only a brief summary of packet types and USB transactions; the reader is referred to the USB 2.0 specification [30] for more information. USB support in microcontrollers requires considerable hardware resources as well as software support in terms of microcontroller firmware that implements the nuances of the USB protocol. Unlike CAN, most microcontrollers implement the D+/D− pin interface directly within the microcontroller and do not use an external transceiver IC to translate between CMOS and USB voltage levels. The PIC18F2455 is one microcontroller currently available from Microchip that implements a USB interface; other companies with microcontrollers that support USB include Cypress Semiconductor, Atmel Corporation, and Intel.

## **15.7 A BRIEF SURVEY OF NON-PIC MICROCONTROLLERS**

The microprocessor, digital signal processor, and microcontroller universe is not infinite, but is large enough that this section can only provide a cursory look at the

processor families available from different companies. Figure 15.18 lists some of the companies and product offerings in the  $\mu\text{P}/\text{DSP}/\mu\text{C}$  universe. The author makes no claims as to completeness of this list, as new companies arise each year to add constellations to the  $\mu\text{P}/\text{DSP}/\mu\text{C}$  universe.

Figure 15.18 splits processors into two categories: high-performance 32-bit and 8/16/32-bit fixed-point. Some of the features listed under the high-performance 32-bit processor category have not been discussed in this book and may be new to you;



**FIGURE 15.18** Microprocessor, digital signal processor, microcontroller universe.

these features are briefly described here. Instruction and data caches were previously mentioned in Section 15.2 and are on-chip high-speed memories that service much of the instruction and data needs of the processor. If a processor has instruction and data caches, it also has external memory, typically SDRAM of some type, that holds the instruction and data that cannot fit within the cache memories. The *cache controller* within the processor is responsible for swapping blocks of instructions/data to/from external memory and on-chip cache. A processor that is *pipelined* means that on average it completes one instruction per clock cycle and that at any given time it has several instructions within it, each at a different state of completion. The PIC18 is not a pipelined processor; it takes four clock cycles to execute one instruction. A pipelined *superscalar* processor can complete more than one instruction per clock cycle. *Virtual memory* means that the processor can execute programs that are larger than the physical memory of the computer system; external storage such as a hard disk is used to store instructions and data that will not fit in physical memory. A *memory management unit* (MMU) on the processor is responsible for detecting when an instruction or data access is not in physical memory; this generates an interrupt that causes the needed instruction/data to be swapped into physical memory from disk. Modern operating systems such as Windows and Linux require processors that support virtual memory. Floating-point was discussed in Chapter 7; a hardware floating-point unit executes instructions that perform floating-point operations. If there is one feature that differentiates a processor between the high- and low-performance camps it is hardware floating-point. This is because hardware floating-point implies a 32-bit architecture and high-performance computation needs, thus necessitating the inclusion of all of the other features that define “high performance.” Many high-performance processors are available as “cores.” This means that a company wishing to build an application-specific integrated circuit (ASIC) can include the processor as a module within its integrated circuit and place customized logic around it to address specific needs. High-performance 32-bit processors are even being included within field programmable gate arrays (FPGAs) from companies such as Xilinx and Altera.

The low-performance category, designated as fixed-point 32/16/8-bit processors, is differentiated from the high-performance category by lack of hardware floating-point support. The Freescale 68XXX family (marketing name Dragonball™) is an example of an embedded processor core that is 32-bit internal without hardware floating-point support. This category is dominated by 8-bit processor families and contains additional companies over those listed in the 32-bit high-performance category. One popular 8-bit processor is the 8051, originally developed by Intel in the 1980s. This processor has been licensed by Intel to several companies that now make 8051-compatible microcontrollers. The 8-bit processors from Atmel and Zilog are flash programmable, with family members available in DIP packages that are suitable for hobbyist prototyping, and feature basically the



same IO module set as the PIC18/PIC16 processors. The Rabbit 2000/3000 processors are based on Z80/Z180 architectures (a processor popular in the 1980s), feature an external memory bus, and are available only in high pin count, surface mount packages. The Rabbit 2000/3000 does not have on-chip program memory; development kits with external FLASH EEPROM memory and SRAM are available from Rabbit Semiconductor. The Basic Stamp® from Parallax is included in this list due to its popularity with hobbyists. However, a Basic Stamp is actually a printed circuit board module that contains an 8-bit PIC processor from either Microchip or Uvicom, a clock source, an in-circuit programming interface, and executes a program that interprets programs written in the BASIC programming language. The Programmable System-on-a-Chip (PSoC) microcontroller from Cypress Semiconductor is unique in that in addition to an 8-bit processor core that is Flash programmable, it contains both analog and digital programmable logic modules that allows a user to customize on-chip peripherals based on the application. One interesting aspect of the low-performance processor list is that 8-bit processor families far outnumber 32/16-bit processor families. This is because the extra performance from a 32/16-bit processor is usually not needed in low-performance families; if extra performance is needed, simply increasing the clock speed on an 8-bit processor is usually a good enough solution.

How does one choose a processor? Of course, application requirements are a critical driver, but typically many processors are able to perform the same task adequately. From a company viewpoint, processor availability (can the vendor ship the volume that I need?), reliability, and volume price are critical. From educator or hobbyist viewpoints, other questions such as those that follow often decide the processor choice.

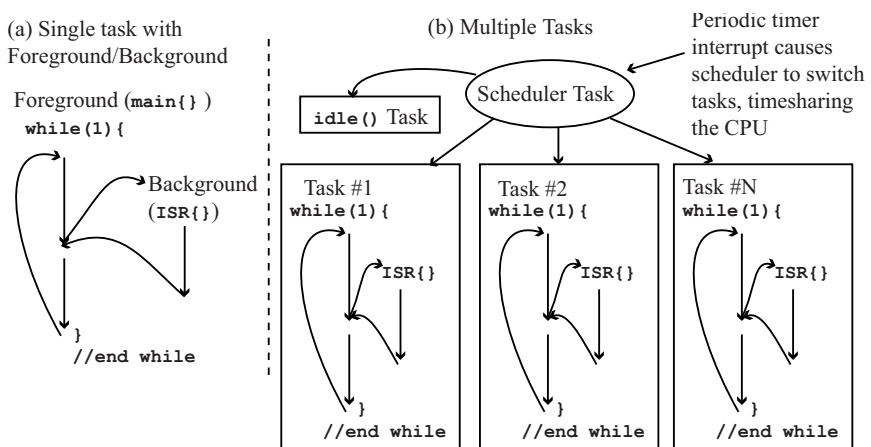
- Is the processor available in DIP packages so I can prototype with it? If not, are there development kits available and what do they cost?
- Is there a free or inexpensive development system available so I can write and simulate assembly language programs?
- Is the processor FLASH programmable? How will I program the processor? Can I do it in-circuit or do I need an external programmer? How expensive is a programmer?
- Is there a C compiler available? How expensive is the C compiler? Is an educational version of the C compiler available?
- Can I purchase processors/programmers in small quantities from a supplier such as Digi-Key?
- Are there sites on the WWW where I can find sample code for common problems?

Exploring the  $\mu\text{P}/\text{DSP}/\mu\text{C}$  universe can be rewarding. It is almost guaranteed that whatever processor provides your first learning experience in the  $\mu\text{P}/\text{DSP}/\mu\text{C}$  universe will not be the only processor you will encounter in your engineer/hobbyist/scientist career. You will quickly discover that different processor families all share common themes that allow you to apply previous hard-earned lessons when exploring the nuances of a new processor family.

## 15.8 REAL-TIME OPERATING SYSTEMS

If you peruse enough embedded system sites, you will encounter references to *real-time operating systems* (RTOS). What is a RTOS and when is it needed? In a nutshell, a RTOS allows a CPU to share its resources between multiple tasks, or threads of execution. You are probably carrying a device in your purse or pocket that is running an RTOS if you own a cell phone or PDA. An example of a CPU being shared between multiple tasks is when your wireless-enabled PDA is downloading e-mail while you are checking your calendar, updating your address book, or playing a game. Figure 15.19a shows the execution model that we have used in this book; that of a `main()` foreground task with interrupts triggering an ISR background task.

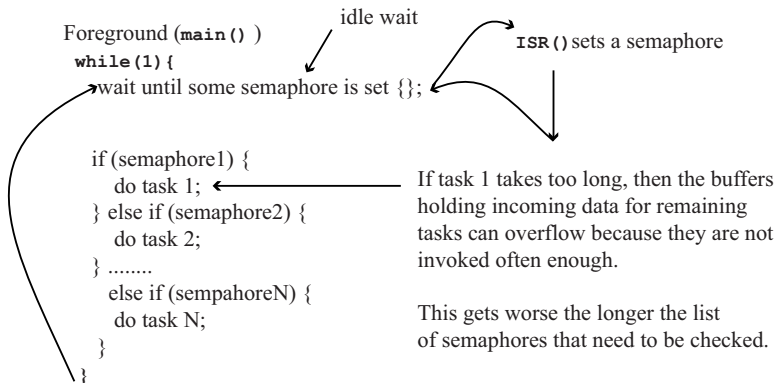
Figure 15.19b illustrates how an RTOS shares the CPU among multiple tasks. A periodic timer interrupt causes a *scheduler* task to switch the CPU between different tasks, with each task receiving a slice of the CPU's time. A *task switch* involves saving the context of the task (register contents and other data that defines the task's state); a task executes until its allotted time expires or the task suspends itself



**FIGURE 15.19** Real-time operating system concept.

to wait for an IO event. If all tasks are waiting for an IO event, the `idle()` task is executed by the scheduler. Tasks can be executed in a round-robin fashion in which each task receives an equal amount of time, or priorities can be assigned in which some tasks get a larger slice of the CPU's time. Tasks can be spawned dynamically by other tasks via system calls to the scheduler. An RTOS comes at a price; the overhead of task switching consumes CPU cycles and memory space is required for saving task context.

Figure 15.20 illustrates an example where an RTOS may be required for application. The code that was written for the home monitoring project in Chapter 14, "Capstone: Audio Sampling, Monitoring System, and Autonomous Robot," resembles the `while(1){}` loop shown in Figure 15.20 where a wait is done on a set of semaphores that are set by the ISR indicating that one of several IO sources has produced input. The code then checks the semaphores and processes the data produced by the IO event.



**FIGURE 15.20** The need for an RTOS.

A problem arises if processing the data from one interrupt source takes so long that the buffers used for other interrupt sources overflow because they are not being emptied often enough. This becomes more of a problem if the data sources are *real-time* data sources such as audio or video that require servicing at fixed rates. Hence, a *real-time* operating system provides the mechanism for sharing the CPU among the different tasks so each task can fulfill its data processing requirements.

An RTOS provides more than just the task switching mechanism. It also provides library functions for common operations such as dynamic memory management, queue creation/management, software timers, and message/semaphore creation/management for task communication. A RTOS is usually distributed in

source form since RTOS companies target many different processor families. Most commercial RTOS vendors focus on the high-end 32-bit processor market, as that is where the power of an RTOS is usually needed. However, even 8-bit processors can benefit from an RTOS if the application is complex enough. An open source RTOS named FreeRTOS ([www.freertos.org](http://www.freertos.org)) is available and has been ported to both the Atmel ATmega32 and Microchip PIC18 (requires the Microchip MCC18 C compiler) processors. The FreeRTOS distribution is one way to get some hands-on experience with RTOS basics.

## SUMMARY

---

This chapter presented a brief survey of microprocessor topics outside of the immediate realm of the PIC18xx2 microcontroller family such as external memory interfacing, SRAM/DRAM memory technologies, the CAN and USB interface standards, the basic motivation behind real-time operating systems, and alternate microcontroller families available from Microchip and other semiconductor companies. If you are interested in understanding the basic features of high-performance processors, a computer architecture book such as [5] is appropriate. Delving deeper into any of the alternate  $\mu$ P/DSP/ $\mu$ C devices listed in Figure 15.18 is as simple as visiting any of the company Web sites and perusing their application notes, reference manuals, and datasheets. The CAN and USB specifications are readily available online and provide additional details on these interfaces. References [32] and [33] provide more information on real-time operating systems.

As this is the last chapter of the book, a valid question is, “Where do I go from here?” You could challenge yourself by implementing some of the code examples or projects in this book on a non-Microchip 8-bit microcontroller; this forces you to generalize the lessons that you have learned in programming the PIC18Fxx2. As a step beyond the projects explored in Chapter 14, Internet-enabled embedded systems are becoming common. One platform for experimenting in this area is TINI® (Tiny InterNet Interfaces) modules that implement an Ethernet interface, support programming TINI applications in Java, C, or assembly language, and allow sensor data read by a TINI board to be retrieved by a Web server over the Internet! More information on TINI is found in [31]. If most of the material in this book is new to you, you are only just beginning to explore the  $\mu$ P/DSP/ $\mu$ C universe. It can turn into a life-long exploration that is both fun and intellectually rewarding—good luck on your travels!

## SUGGESTED SURVEY TOPICS

---

Instead of review problems, this section contains additional survey topics over those discussed in this chapter. In an educational environment, these problems are best assigned as end-of-course reports.

1. Select a non-Microchip 8-bit processor from the list in Figure 15.18. Visit the company Web site and download both the reference manual for the processor architecture and the specific datasheet for a particular family. Compare and contrast the processor with the PIC18Fxx2 by answering the following questions:
  - i. What is the width of the instruction word?
  - ii. What is the width of the internal data registers?
  - iii. What is the width of the program counter or instruction pointer?
  - iv. How much on-chip RAM does the processor have?
  - v. Does the processor have an external bus for accessing program or data memory?
  - vi. Does the processor have signed comparison instructions? If “yes,” give an example.
  - vii. Implement the operation  $i = k + j$  in the processor’s assembly language where  $i, k, j$  are char variables.
  - viii. Does the processor have special instructions for multiply and/or divide? If “yes,” give an example.
  - ix. Does the processor have a fixed-sized or variable sized stack?
  - x. Does the processor have push/pop instructions for accessing the stack? If “yes,” give an example.
  - xi. How is indirect addressing (similar to the FSR/INDF capability on the PIC18) implemented? Give an example.
  - xii. How many clock cycles does it take for an addition operation to complete?
  - xiii. Give the machine code format for an addition operation.
  - xiv. What flags are supported in the status register of the processor?
  - xv. Does the processor have an instruction that supports a multiple position shift? If “yes,” give an example.
  - xvi. What serial standards are supported? Asynchronous? SPI? I2C? What is the maximum synchronous transfer rate?
  - xvii. Does it have an on-chip ADC? If “yes,” how many channels and what precision?

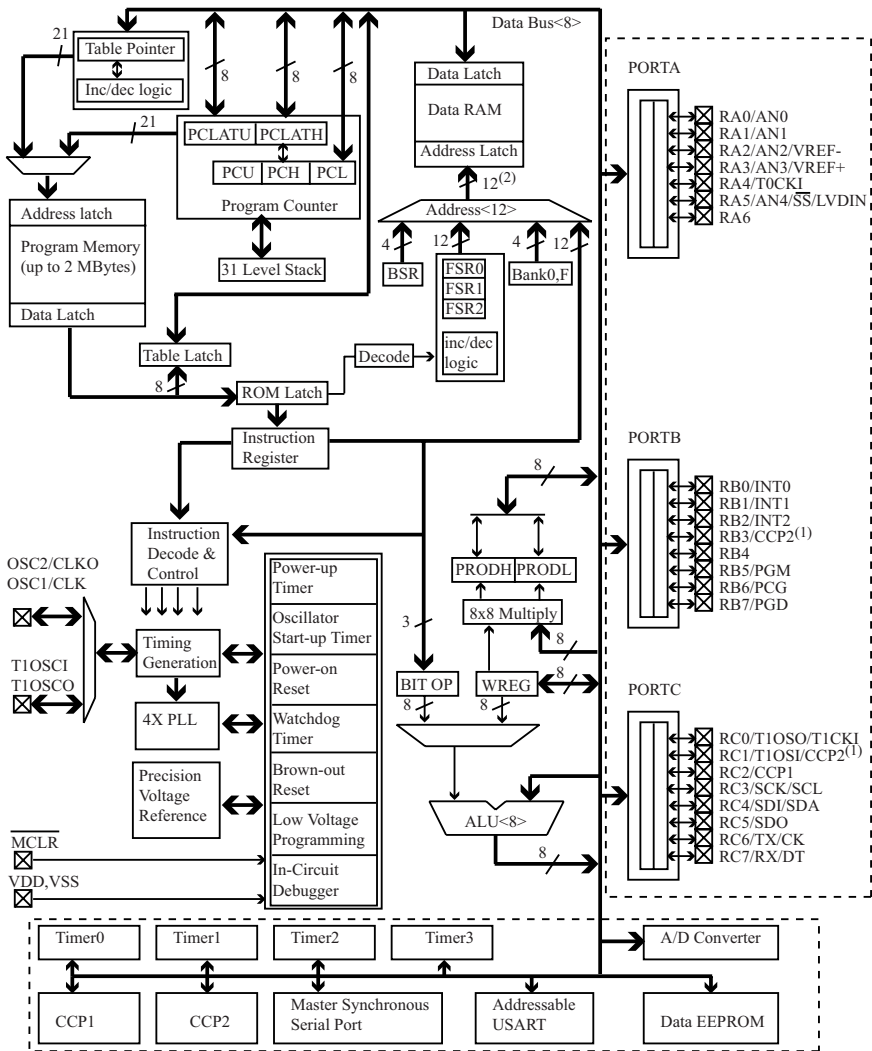
- xviii. At reset, from what location is the first instruction fetched?
  - xvix. What location in memory contains the interrupt vector(s)? Does the processor support multiple interrupt priorities?
  - xx. What are the minimum and maximum package sizes in terms of pin counts? For the maximum package size, what is the parallel port count?
  - xxi. What is the device's maximum clock frequency?
  - xxii. What is the Vdd range supported by the device?
  - xxiii. How many timers and what sizes are supported?
  - xxiv. Is PWM supported? If "yes," how many channels?
2. FireWire (IEEE standard 1394) is another high-speed serial interface standard. Peruse the Web and answer the following questions.
- i. What are the electrical signaling levels used in FireWire?
  - ii. How is addressing done?
  - iii. Does it support multiple bus masters? If "yes," how is arbitration done?
  - iv. Give an example transfer on FireWire and compare it to a USB transfer.
3. Pick one of the high-performance 32-bit processors from Figure 15.18 and answer the same questions of problem #1 in addition to the following:
- i. How long does it take to do a single precision floating-point add? A single precision floating-point divide?
  - ii. What size on-chip data and instruction caches does it support?
  - iii. Does it support a glueless interface to DRAM? If "yes," what types of synchronous DRAMs are supported?
  - iv. Does it have a memory management unit?
  - v. If it is superscalar, how many instructions per clock can it typically execute?
4. Do research on the LIN (Local Interconnect Network) bus and answer the following questions.
- i. How many wires does it need?
  - ii. Classify it as duplex, half-duplex, or simplex.
  - iii. How is addressing to devices handled?
  - iv. What are the signaling levels?

- v. Is it synchronous or asynchronous?
  - vi. What is the maximum transfer speed?
  - vii. What is the maximum number of devices allowed on the bus?
5. DDR-DRAM and RDRAM are both dynamic memory devices, but have significantly different interfaces. Find datasheets on each and answer the following questions.
- i. How do the electrical signaling levels of the two devices differ?
  - ii. What are the clocking requirements of each?
  - iii. How are chip selects handled for each?
  - iv. Describe how a block read transfer is accomplished for each device.
6. The System Management Bus (SMBus) is a two-wire bus that was derived from the I<sup>2</sup>C standard by Intel in 1995. It is currently used in PC motherboards and various microcontrollers. Download the SMBus specification [38] and answer the following questions.
- i. Give a couple of key differences between SMBus and I<sup>2</sup>C. For the key differences you chose, give the reasoning behind these changes.
  - ii. Can you give an advantage of SMBus over the I<sup>2</sup>C bus? Defend your answer.
  - iii. Are the I<sup>2</sup>C devices used in the projects of Chapter 14 (the MAX517 DAC, 24LC515 Serial EEPROM, and the DS1621 Digital Thermometer) compatible with the SMBus? Defend your answer.

**A****PIC18Fxx2 Architecture,  
Instruction Set, Register  
Summary**

This appendix contains a summary of the PIC18Fxx2 architecture, instruction set, and registers. The PIC18Fxx2 family contains the PIC18F242, PIC18F252, PIC18F442, and PIC18F452 members. The PIC18F2x2 architecture has only three parallel ports because of package pin limitations, while the PIC18F4x2 architecture has five parallel ports. Figure A.1 shows a block diagram of the PIC18F2x2 architecture. The machine code and flag settings for each PIC18Fxx2 instruction are given in Figures A.2, A.3, and A.4. An RTL description of each instruction is found in Figures A.5 and A.6. The Special Function Register bit definitions are found in Figures A.7 and A.8. The memory map of the Special Function Registers is shown in Figure A.9. The program memory configuration registers are summarized in Figure A.10, with individual bit definitions found in Figure A.11.





**Note 1:** Optional multiplexing of CCP2 input/output with RB3 is enabled by selection of configuration bit.  
**2:** The high order bits of the Direct Address for the RAM are from the BSR register (except for the MOVWF instruction).  
**3:** Many of the general purpose I/O pins are multiplexed with one or more peripheral module functions. The multiplexing combinations are device dependent.

Figure redrawn by author from PIC18Fxx2 datasheet (DS39564B), Microchip Technology Inc.

**FIGURE A.1** PIC18F2X2 block diagram.<sup>1</sup>

<sup>1</sup> Figure A.1 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.'s prior written consent.

Mnemonic, Operands	Description	Cycles	16-Bit Instruction Word		Status Affected	Notes			
			MSb	LSb					
<b>BYTE-ORIENTED FILE REGISTER OPERATIONS</b>									
ADDWF	f,d,a	Add WREG and f	1	0010	01da	ffff	ffff	C,DC,Z,OV,N	1,2
ADDWFC	f,d,a	Add WREG and Carry bit to f	1	0010	00da	ffff	ffff	C,DC,Z,OV,N	1,2
ANDWF	f,d,a	AND WREG with f	1	0001	01da	ffff	ffff	Z,N	1,2
CLRF	f,a	Clear f	1	0110	101a	ffff	ffff	Z	2
COMF	f,d,a	Complement f	1	0001	11da	ffff	ffff	Z,N	1,2
CPFSEQ	f,a	Compare f with WREG, skip =	1(2 or 3)	0110	001a	ffff	ffff	None	4
CPFSGT	f,a	Compare f with WREG, skip >	1(2 or 3)	0110	010a	ffff	ffff	None	4
CPFSLT	f,a	Compare f with WREG, skip <	1(2 or 3)	0110	000a	ffff	ffff	None	1,2
DECF	f,d,a	Decrement f	1	0000	01da	ffff	ffff	C,DC,Z,OV,N	1,2,3,4
DECFSZ	f,d,a	Decrement f, Skip if 0	1(2 or 3)	0010	11da	ffff	ffff	None	1,2,3,4
DECFSNZ	f,d,a	Decrement f, Skip if Not 0	1(2 or 3)	0100	11da	ffff	ffff	None	1,2
INCF	f,d,a	Increment f	1	0010	10da	ffff	ffff	C,DC,Z,OV,N	1,2,3,4
INCFSZ	f,d,a	Increment f, Skip if 0	1(2 or 3)	0011	11da	ffff	ffff	None	4
INFSNZ	f,d,a	Increment f, Skip if Not 0	1(2 or 3)	0100	10da	ffff	ffff	None	1,2
IORWF	f,d,a	Inclusive OR WREG with f	1	0001	00da	ffff	ffff	Z,N	1,2
MOVF	f,d,a	Move f	1	0101	00da	ffff	ffff	Z,N	1
MOVFF	f <sub>s</sub> ,f <sub>d</sub>	Move f <sub>s</sub> (source) to 1 <sup>st</sup> word f <sub>d</sub> (destination) 2 <sup>nd</sup> word	2	1100	ffff	ffff	ffff	None	
MOVWF	f,a	Move WREG to f	1	0110	111a	ffff	ffff	None	
MULWF	f,a	Multiply WREG with f	1	0000	001a	ffff	ffff	None	
NEGf	f,a	Negate f	1	0110	110a	ffff	ffff	C,DC,Z,OV,N	1,2
RLCF	f,d,a	Rotate Left f through Carry	1	0011	01da	ffff	ffff	C,Z,N	
RLNCF	f,d,a	Rotate Left f (No Carry)	1	0100	01da	ffff	ffff	Z,N	1,2
RRCF	f,d,a	Rotate Right f through Carry	1	0011	00da	ffff	ffff	C,Z,N	
RRNCF	f,d,a	Rotate Right f (no Carry)	1	0100	00da	ffff	ffff	Z,N	
SETF	f,a	Set f	1	0110	100a	ffff	ffff	None	
SUBFWB	f,d,a	Subtract f from WREG with borrow	1	0101	01da	ffff	ffff	C,DC,Z,OV,N	1,2
SUBWF	f,d,a	Subtract WREG from f	1	0101	11da	ffff	ffff	C,DC,Z,OV,N	
SUBWFB	f,d,a	Subtract WREG from f with borrow	1	0101	10da	ffff	ffff	C,DC,Z,OV,N	1,2
SWAPF	f,d,a	Swap nibbles in f	1	0011	10da	ffff	ffff	None	4
TSTFSZ	f,a	Test f, skip if 0	1(2 or 3)	0110	011a	ffff	ffff	None	1,2
XORWF	f,d,a	Exclusive OR WREG with f	1	0001	10da	ffff	ffff	Z,N	
<b>BIT-ORIENTED FILE REGISTER OPERATIONS</b>									
BCF	f,b,a	Bit Clear f	1	1001	bbba	ffff	ffff	None	1,2
BSF	f,b,a	Bit Set f	1	1000	bbba	ffff	ffff	None	1,2
BTFSZ	f,b,a	Bit Test f, Skip if Clear	1(2 or 3)	1011	bbba	ffff	ffff	None	3,4
BTFSZ	f,b,a	Bit Test f, Skip if Set	1(2 or 3)	1010	bbba	ffff	ffff	None	3,4
BTG	f,b,a	Bit Toggle f	1	0111	bbba	ffff	ffff	None	1,2

- Note 1:** When a PORT register is modified as a function of itself (e.g., MOVF PORTB, 1, 0), the value used will be that value present on the pins themselves. For example, if the data latch is '1' for a pin configured as input and is driven low by an external device, the data will be written back with a '0'.
- 2:** If this instruction is executed on the TMR0 register (and, where applicable, d = 1), the prescaler will be cleared if assigned.
- 3:** If Program Counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a NOP.
- 4:** Some instructions are 2-word instructions. The second word of these instructions will be executed as a NOP, unless the first word of the instruction retrieves the information embedded in these 16-bits. This ensures that all program memory locations have a valid instruction.
- 5:** If the Table Write starts the write cycle to internal memory, the write will continue until terminated.

Figure redrawn by author from PIC18Fxx2 datasheet (DS39564B), Microchip Technology Inc.

**FIGURE A.2** Byte-oriented, bit-oriented file register operations.<sup>2</sup>

<sup>2</sup> Figure A.2 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.'s prior written consent.

Mnemonic, Operands	Description	Cycles	16-Bit Instruction Word		Status Affected	Notes
			MSb	LSb		
<b>CONTROL OPERATIONS</b>						
BC	n	Branch if Carry	1 (2)	1110 0010 nnnn nnnn	None	
BN	n	Branch if Negative	1 (2)	1110 0110 nnnn nnnn	None	
BNC	n	Branch if Not Carry	1 (2)	1110 0011 nnnn nnnn	None	
BNN	n	Branch if Not Negative	1 (2)	1110 0111 nnnn nnnn	None	
BNOV	n	Branch if Not Overflow	1 (2)	1110 0101 nnnn nnnn	None	
BNZ	n	Branch if Not Zero	2	1110 0001 nnnn nnnn	None	
BOV	n	Branch if Overflow	1 (2)	1110 0100 nnnn nnnn	None	
BRA	n	Branch Unconditionally	1 (2)	1101 0nnn nnnn nnnn	None	
BZ	n	Branch if Zero	1 (2)	1110 0000 nnnn nnnn	None	
CALL	n, s	Call subroutine (1 <sup>st</sup> word) 2 <sup>nd</sup> word	2	1110 110s kkkk kkkk 1111 kkkk kkkk kkkk	None	
CLRWDT	—	Clear Watchdog Timer	1	0000 0000 0000 0100	$\overline{TO}$ , $\overline{PD}$	
DAW	—	Decimal Adjust WREG	1	0000 0000 0000 0111	C	
GOTO	n	Go to address (1 <sup>st</sup> word) 2 <sup>nd</sup> word	2	1110 1111 kkkk kkkk 1111 kkkk kkkk kkkk	None	
NOP	—	No Operation	1	0000 0000 0000 0000	None	
NOP	—	No Operation	1	1111 xxxx xxxx xxxx	None	4
POP	—	Pop top of return stack (TOS)	1	0000 0000 0000 0110	None	
PUSH	—	Push top of return stack (TOS)	1	0000 0000 0000 0101	None	
RCALL	n	Relative Call	2	1101 1nnn nnnn nnnn	None	
RESET	—	Software device RESET	1	0000 0000 1111 1111	ALL	
RETFIE	s	Return from interrupt enable	2	0000 0000 0001 000s	GIE/GIEH, PEIE/GIEL	
RETLW	k	Return with literal in WREG	2	0000 1100 kkkk kkkk	None	
RETURN	s	Return from Subroutine	2	0000 0000 0001 001s	None	
SLEEP	—	Go into Standby mode	1	0000 0000 0000 0011	$\overline{TO}$ , $\overline{PD}$	

- Note 1:** When a PORT register is modified as a function of itself (e.g., MOVF PORTB, 1, 0), the value used will be that value present on the pins themselves. For example, if the data latch is '1' for a pin configured as input and is driven low by an external device, the data will be written back with a '0'.
- 2:** If this instruction is executed on the TMR0 register (and, where applicable, d = 1), the prescaler will be cleared if assigned.
- 3:** If Program Counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a NOP.
- 4:** Some instructions are 2-word instructions. The second word of these instructions will be executed as a NOP, unless the first word of the instruction retrieves the information embedded in these 16-bits. This ensures that all program memory locations have a valid instruction.
- 5:** If the Table Write starts the write cycle to internal memory, the write will continue until terminated.

Figure redrawn by author from PIC18Fxx2 datasheet (DS39564B), Microchip Technology Inc.

**FIGURE A.3** Control operations.<sup>3</sup>

<sup>3</sup> Figure A.3 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.'s prior written consent.

Mnemonic, Operands	Description	Cycles	16-Bit Instruction Word		Status Affected	Notes
			MSb	LSb		
<b>LITERAL OPERATIONS</b>						
ADDLW	k	Add literal and WREG	1	0000 1111 kkkk kkkk	C,DC,Z,OV,N	
ANDLW	k	AND literal with WREG	1	0000 1011 kkkk kkkk	Z,N	
IORLW	k	Inclusive OR literal with WREG	1	0000 1001 kkkk kkkk	Z,N	
LFSR	f,k	Move literal (12-bit) 2 <sup>nd</sup> word to FSRx 1 <sup>st</sup> word	2	1110 1110 00ff kkkk 1111 0000 kkkk kkkk	None	
MOVLB	k	Move literal to BSR<3:0>	1	0000 0001 0000 kkkk	None	
MOVLW	k	Move literal to WREG	1	0000 1110 kkkk kkkk	None	
MULLW	k	Multiply literal with WREG	1	0000 1101 kkkk kkkk	None	
RETLW	k	Return with literal in WREG	2	0000 1100 kkkk kkkk	None	
SUBLW	k	Subtract WREG from literal	1	0000 1000 kkkk kkkk	C,DC,Z,OV,N	
XORLW	k	Exclusive OR literal with WREG	1	0000 1010 kkkk kkkk	Z,N	
<b>DATA-MEMORY ↔ PROGRAM MEMORY OPERATIONS</b>						
TBLRD*		Table Read	2	0000 0000 0000 1000	None	
TBLRD*+		Table Read with post-increment		0000 0000 0000 1001	None	
TBLRD*-		Table Read with post-decrement		0000 0000 0000 1010	None	
TBLRD+*		Table Read with pre-increment		0000 0000 0000 1011	None	
TBLWT*		Table Write	2 (5)	0000 0000 0000 1100	None	
TBLWT*+		Table Write with post-increment		0000 0000 0000 1101	None	
TBLWT*-		Table Write with post-decrement		0000 0000 0000 1110	None	
TBLWT+*		Table Write with pre-increment		0000 0000 0000 1111	None	

- Note 1:** When a PORT register is modified as a function of itself (e.g., MOVF PORTB, 1, 0), the value used will be that value present on the pins themselves. For example, if the data latch is '1' for a pin configured as input and is driven low by an external device, the data will be written back with a '0'.
- 2:** If this instruction is executed on the TMR0 register (and, where applicable, d = 1), the prescaler will be cleared if assigned.
- 3:** If Program Counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a NOP.
- 4:** Some instructions are 2-word instructions. The second word of these instructions will be executed as a NOP, unless the first word of the instruction retrieves the information embedded in these 16-bits. This ensures that all program memory locations have a valid instruction.
- 5:** If the Table Write starts the write cycle to internal memory, the write will continue until terminated.

Figure redrawn by author from PIC18Fxx2 datasheet (DS39564B), Microchip Technology Inc.

**FIGURE A.4** Literal, table read/write operations.<sup>4</sup>

<sup>4</sup> Figure A.4 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.'s prior written consent.

Mnemonic, Operands	Description	RTL Description	Comments
ADDLW k	Add literal and WREG	$(W) + k \rightarrow \text{dest}$	
ADDWF f,d,a	Add WREG and f	$(W) + (f) \rightarrow \text{dest}$	
ADDWFC f,d,a	Add WREG and Carry bit to f	$(W) + (f) + (C) \rightarrow \text{dest}$	
ANDLW k	AND literal with WREG	$(W) .\text{AND. } k \rightarrow W$	
ANDWF f,d,a	AND WREG with f	$(W) .\text{AND. } (f) \rightarrow \text{dest}$	
BC	Branch if Carry	if (Carry flag is 1) $(PC)+2+2n \rightarrow PC$	
BCF f,b,a	Bit Clear f	$0 \rightarrow \text{f[b]}$	
BN	Branch if Negative	if (Negative flag is 1) $(PC)+2+2n \rightarrow PC$	
BNC	Branch if Not Carry	if (Carry flag is 0) $(PC)+2+2n \rightarrow PC$	
BNN	Branch if Not Negative	if (Negative flag is 0) $(PC)+2+2n \rightarrow PC$	
BNOV	Branch if Not Overflow	if (Overflow flag is 0) $(PC)+2+2n \rightarrow PC$	
BNZ	Branch if Not Zero	if (Zero flag is 0) $(PC)+2+2n \rightarrow PC$	
BOV	Branch if Overflow	if (Overflow flag is 1) $(PC)+2+2n \rightarrow PC$	
BRA n	Branch Unconditionally	$(PC)+2+2n \rightarrow PC$	
BSF f,b,a	Bit Set f	$1 \rightarrow \text{f[b]}$	
BTFSC f,b,a	Bit Test f, Skip if Clear	skip if $(\text{f[b]}) = 0$	
BTFSS f,b,a	Bit Test f, Skip if Set	skip if $(\text{f[b]}) = 1$	
BTG f,b,a	Bit Toggle f	$\sim(\text{f[b]}) \rightarrow \text{f[b]}$	
BZ	Branch if Zero	if (Zero flag is 1) $(PC)+2+2n \rightarrow PC$	
CALL k[s]	Call subroutine	$(PC)+4 \rightarrow \text{TOS}, k \rightarrow \text{PC}[20:1]$ , if $s = 1$ { save to shadow regs $(W) \rightarrow \text{WS}, (\text{STATUS}) \rightarrow \text{STATUSS},$ $(\text{BSR}) \rightarrow \text{BSS}$ }	
CLRF f,a	Clear f	$0 \rightarrow f, 1 \rightarrow Z$	
CLRWDT —	Clear Watchdog Timer	$0 \rightarrow \text{WDT}, 0 \rightarrow \text{WDT Postscaler},$ $1 \rightarrow \text{TO}, 1 \rightarrow \text{PD}$	TO, PD bits in RCON register
COMF f,d,a	Complement f	$\sim(f) \rightarrow \text{dest}$	
CPSEQ f,a	Compare f with WREG, skip =	skip if $(f) = (W)$	
CPFSGT f,a	Compare f with WREG, skip >	skip if $(f) > (W)$	unsigned comparison
CPFSLT f,a	Compare f with WREG, skip <	skip if $(f) < (W)$	unsigned comparison
DAW —	Decimal Adjust WREG	if $[W[3:0] > 9]$ or $[DC=1]$ then $(W[3:0]) + 6 \rightarrow W[3:0]$ else $(W[3:0]) \rightarrow W[3:0]$  if $[W[7:4] > 9]$ or $[C=1]$ then $(W[7:4]) + 6 \rightarrow W[7:4]$ else $(W[7:4]) \rightarrow W[7:4]$	Do decimal adjust after addition operation of two BCD numbers to post correct to BCD result.
DECf f,d,a	Decrement f	$(f) - 1 \rightarrow \text{dest}$	
DECFSZ f,d,a	Decrement f, Skip if 0	$(f) - 1 \rightarrow \text{dest}, \text{skip if result} = 0$	
DECFSNZ f,d,a	Decrement f, Skip if Not 0	$(f) - 1 \rightarrow \text{dest}, \text{skip if result} \neq 0$	
GOTO l	Go to k	$k \rightarrow \text{PC}[20:1]$	
INCF f,d,a	Increment f	$(f) + 1 \rightarrow \text{dest}$	
INCFSZ f,d,a	Increment f, Skip if 0	$(f) + 1 \rightarrow \text{dest}, \text{skip if result} = 0$	
INFSNZ f,d,a	Increment f, Skip if Not 0	$(f) + 1 \rightarrow \text{dest}, \text{skip if result} \neq 0$	
IORLW k	Inclusive OR literal with WREG	$(W) .\text{OR. } k \rightarrow W$	
IORWF f,d,a	Inclusive OR WREG with f	$(W) .\text{OR. } (f) \rightarrow \text{dest}$	
LFSR f,k	Move literal to FSRf	$k \rightarrow \text{FSRf}$	
MOVF f,d,a	Move f	$f \rightarrow \text{dest}$	
MOVFF f <sub>s</sub> ,f <sub>d</sub>	Move f <sub>s</sub> (source) to f <sub>d</sub> (dest)	$(f_s) \rightarrow f_d$	
MOVLB k	Move literal to BSR<3:0>	$k \rightarrow \text{BSR}$	

FIGURE A.5 Instruction set RTL description (part 1).

Mnemonic, Operands	Description	RTL Description	Comments
MOVLW k	Move literal to WREG	$k \rightarrow W$	
MOVWF f,a	Move WREG to f	$(W) \rightarrow f$	
MULLW k	Multiply literal with WREG	$(W) * k \rightarrow \text{PRODH:PRODL}$	
MULWF f,a	Multiply WREG with f	$(W) * (f) \rightarrow \text{PRODH:PRODL}$	
NEGF f,a	Negate f	$\sim(f) + 1 \rightarrow f$	perform 2's comp.
NOP	No Operation		
POP	Pop top of return stack (TOS)	$(\text{TOS}) \rightarrow \text{bit bucket}$	
PUSH	Push top of return stack (TOS)	$(\text{PC})+2 \rightarrow \text{TOS}$	
RCALL n	Relative Call	$(\text{PC})+2 \rightarrow \text{TOS}$ , $(\text{PC}) + 2 + 2n \rightarrow \text{PC}$	
RESET	Software device RESET	Executes a MCLR# reset in software	
RETFIE s	Return from interrupt enable	$(\text{TOS}) \rightarrow \text{PC}$ , $1 \rightarrow \text{GIE/GIEH or PEIE/GIEL}$ , if $(s = 1)$ { //restore from shadow regs $(\text{WS}) \rightarrow \text{W}$ , $(\text{STATUS}) \rightarrow \text{STATUS}$ , $(\text{BSRS}) \rightarrow \text{BSR}$ }	
RETLW k	Return with literal in WREG	$k \rightarrow \text{W}$ , $(\text{TOS}) \rightarrow \text{PC}$	
RETURN s	Return from Subroutine	$(\text{TOS}) \rightarrow \text{PC}$ , if $(s = 1)$ { //restore from shadow regs $(\text{WS}) \rightarrow \text{W}$ , $(\text{STATUS}) \rightarrow \text{STATUS}$ , $(\text{BSRS}) \rightarrow \text{BSR}$ }	
RLCF f,d,a	Rotate Left f through Carry	$(f[n]) \rightarrow (\text{dest}[n+1])$ , $(f[7]) \rightarrow \text{C}$ , $\text{C} \rightarrow \text{dest}[0]$	
RLNCF f,d,a	Rotate Left f (No Carry)	$(f[n]) \rightarrow (\text{dest}[n+1])$ , $(f[7]) \rightarrow \text{dest}[0]$	
RRCF f,d,a	Rotate Right f through Carry	$(f[n]) \rightarrow (\text{dest}[n-1])$ , $(f[0]) \rightarrow \text{C}$ , $\text{C} \rightarrow \text{dest}[7]$	
RRNCF f,d,a	Rotate Right f (no Carry)	$(f[n]) \rightarrow (\text{dest}[n-1])$ , $(f[0]) \rightarrow \text{dest}[7]$	
SETF f,a	Set f	$0xFF \rightarrow f$	
SLEEP	Go into Standby mode	$0 \rightarrow \text{WDT}$ , $0 \rightarrow \text{WDT postscaler}$ , $1 \rightarrow \text{TO}$ , $0 \rightarrow \text{PD}$	TO, PD in RCON register
SUBFWB f,d,a	Subtract f from WREG with borrow	$(W) - (f) - \sim(\text{C}) \rightarrow \text{dest}$	
SUBLW k	Subtract WREG from literal	$k - (W) \rightarrow \text{dest}$	
SUBWF f,d,a	Subtract WREG from f	$(f) - (W) \rightarrow \text{dest}$	
SUBWFB f,d,a	Subtract WREG from f with borrow	$(f) - (W) - \sim(\text{C}) \rightarrow \text{dest}$	
SWAPF f,d,a	Swap nibbles in f	$(f[3:0]) \rightarrow \text{dest}[7:4]$ , $(f[7:4]) \rightarrow \text{dest}[3:0]$	
TBLRD*	Table Read	$(\text{Prog Mem}(\text{TBLPTR})) \rightarrow \text{TABLAT}$	
TBLRD*+	Table Read, post-increment	$(\text{Prog Mem}(\text{TBLPTR})) \rightarrow \text{TABLAT}$ , $(\text{TBLPTR}) + 1 \rightarrow \text{TBLPTR}$	
TBLRD*-	Table Read, post-decrement	$(\text{Prog Mem}(\text{TBLPTR})) \rightarrow \text{TABLAT}$ , $(\text{TBLPTR}) - 1 \rightarrow \text{TBLPTR}$	
TBLRD*+	Table Read, pre-increment	$(\text{TBLPTR}) + 1 \rightarrow \text{TBLPTR}$ , $(\text{Prog Mem}(\text{TBLPTR})) \rightarrow \text{TABLAT}$	
TBLWT	Table Write	$(\text{TABLAT}) \rightarrow \text{Holding Register}$	
TBLWT	Table Write, post-increment	$(\text{TABLAT}) \rightarrow \text{Holding Register}$ , $(\text{TBLPTR}) + 1 \rightarrow \text{TBLPTR}$	
TBLWT	Table Write, post-decrement	$(\text{TABLAT}) \rightarrow \text{Holding Register}$ , $(\text{TBLPTR}) - 1 \rightarrow \text{TBLPTR}$	
TBLWT	Table Write, pre-increment	$(\text{TBLPTR}) + 1 \rightarrow \text{TBLPTR}$ , $(\text{TABLAT}) \rightarrow \text{Holding Register}$	
TSFBSZ f,a	Test f, skip if 0	skip if $f = 0$	
XORLW k	Exclusive OR literal with WREG	$(W) .\text{XOR. } k \rightarrow \text{W}$	
XORWF f,d,a	Exclusive OR WREG with f	$(W) .\text{XOR. } (f) \rightarrow \text{dest}$	

FIGURE A.6 Instruction set RTL description (part 2).

File Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR	
TOSU	---	---	---	Top-of-Stack upper Byte (TOS[20:16])						---0 0000
TOSH	Top-of-Stack High Byte (TOS[15:8])								0000 0000	
TOSL	Top-of-Stack Low Byte (TOS[7:0])								0000 0000	
STKPTR	STKFUL	STKUNF	---	Return Stack Pointer						00-0 0000
PCLATU	---	---	---	Holding Register for PC[20:16]						---0 0000
PCLATH	Holding Register for PC[15:8]								0000 0000	
PCL	PC Low Byte (PC[7:0])								0000 0000	
TBLPTRU	---	---	bit21 (2)	Program Memory Table Pointer Upper Byte (TBLPTR[20:16])						--00 0000
TBLPTRH	Program Memory Table Pointer High Byte (TBLPTR[15:8])								0000 0000	
TBLPTRL	Program Memory Table Pointer Low Byte (TBLPTR[7:0])								0000 0000	
TABLAT	Program Memory Table Latch								0000 0000	
PRODH	Product Register High Byte								xxxx xxxx	
PRODL	Product Register Low Byte								xxxx xxxx	
INTCON	GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF	0000 000x	
INTCON2	RBPV	INTEDG0	INTEDG1	INTEDG2	---	TMR0IP	---	RBIP	1111 -1-1	
INTCON3	INT2IP	INT1IP	---	INT2IE	INT1IE	---	INT2IF	INT1IF	11-0 0-00	
INDF0	Uses contents of FSR0 to address data memory – value of FSR0 not changed (not a physical register)								n/a	
POSTINC0	Uses contents of FSR0 to address data memory – value of FSR0 post-incremented (not a physical register)								n/a	
POSTDEC0	Uses contents of FSR0 to address data memory – value of FSR0 post-decremented (not a physical register)								n/a	
PREINC0	Uses contents of FSR0 to address data memory – value of FSR0 pre-incremented (not a physical register)								n/a	
PLUSW0	Uses contents of FSR0 to address data memory – value of FSR0 not changed (not a physical register) Offset by value in WREG								n/a	
FSR0H	---	---	---	---	Indirect Data Memory Address Pointer 0 High Byte				---- 0000	
FSR0L	Indirect Data Memory Address Pointer 0 Low Byte								xxxx xxxx	
WREG	Working Register								xxxx xxxx	
INDF1	Uses contents of FSR1 to address data memory – value of FSR1 not changed (not a physical register)								n/a	
POSTINC1	Uses contents of FSR1 to address data memory – value of FSR1 post-incremented (not a physical register)								n/a	
POSTDEC1	Uses contents of FSR1 to address data memory – value of FSR1 post-decremented (not a physical register)								n/a	
PREINC1	Uses contents of FSR1 to address data memory – value of FSR1 pre-incremented (not a physical register)								n/a	
PLUSW1	Uses contents of FSR1 to address data memory – value of FSR1 not changed (not a physical register) Offset by value in WREG								n/a	
FSR1H	---	---	---	---	Indirect Data Memory Address Pointer 1 High Byte				---- 0000	
FSR1L	Indirect Data Memory Address Pointer 1 Low Byte								xxxx xxxx	
BSR	---	---	---	---	Bank Select Register				---- 0000	
INDF2	Uses contents of FSR2 to address data memory – value of FSR2 not changed (not a physical register)								n/a	
POSTINC2	Uses contents of FSR2 to address data memory – value of FSR2 post-incremented (not a physical register)								n/a	
POSTDEC2	Uses contents of FSR2 to address data memory – value of FSR2 post-decremented (not a physical register)								n/a	
PREINC2	Uses contents of FSR2 to address data memory – value of FSR2 pre-incremented (not a physical register)								n/a	
PLUSW2	Uses contents of FSR2 to address data memory – value of FSR2 not changed (not a physical register) Offset by value in WREG								n/a	
FSR2H	---	---	---	---	Indirect Data Memory Address Pointer 2 High Byte				---- 0000	
FSR2L	Indirect Data Memory Address Pointer 2 Low Byte								xxxx xxxx	
STATUS	---	---	---	N	OV	Z	DC	C	--x xxxxx	
TMR0H	Timer0 Register High Byte								0000 0000	
TMR0L	Timer0 Register Low Byte								xxxx xxxx	
TOCON	TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0	1111 1111	
OSCCON	---	---	---	---	---	---	---	SCS	---- --0	
LVDCON	---	---	IRVST	LV DEN	LV DL3	LV DL2	LV DL1	LV DL0	--00 0101	
WDTCON	---	---	---	---	---	---	---	SWDTE	---- --0	
RCON	IPEN	---	---	RI	T0	PD	POR	BOR	0--1 11qq	
TMR1H	Timer1 Register High Byte								xxxx xxxx	
TMR1L	Timer1 Register Low Byte								xxxx xxxx	
T1CON	RD16	---	T1CKPS1	T1CKPS0	T1OSCEN	T1SYNC	TMR1CS	TMR1ON	0-00 0000	

Legend: x = unknown, u = unchanged, - = unimplemented, q = value depends on condition

Shaded cells are unimplemented, read as "0"

Note 1: RA6 and associated bits are configured as port pins in RCIO and ECIO Oscillator mode only and read '0' in all other Oscillator modes.

2: Bit 21 of the TBLPTRU allows access to the device configuration bits

3: These registers and bits are reserved on the PIC18F2X2 devices; always maintain these clear.

Figure redrawn by author from PIC18Fxx2 datasheet (DS39564B), Microchip Technology Inc.

**FIGURE A.7** Register file summary (part 1).<sup>5</sup>

<sup>5</sup> Figure A.7 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.'s prior written consent.

File Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR
TMR2	Timer2 Register								0000 0000
PR2	Timer1 Period Register								1111 1111
T2CON	—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0	-000 0000
SSPBUF	SSP Receive Buffer/Transmit Register								xxxx xxxx
SSPADD	SSP Address Register in I2C Slave mode. SSP Baud Rate Reload Register in I2C Master Mode.								0000 0000
SSPSTAT	SMP	CKE	D/Ā	P	S	R/W	UA	BF	0000 0000
SSPCON1	WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0	0000 0000
SSPCON2	GCEN	ACKSTAT	ACKDT	ACKEN	RCEN	PEN	RSEN	SEN	0000 0000
ADRESH	A/D Result Register High Byte								xxxx xxxx
ADRESL	A/D Result Register Low Byte								xxxx xxxx
ADCON0	ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/DONE	—	ADON	0000 00-0
ADCON1	ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0	00-- 0000
CCPR1H	Capture/Compare/PWM Register1 High Byte								xxxx xxxx
CCPR1L	Capture/Compare/PWM Register1 Low Byte								xxxx xxxx
CCP1CON	—	—	DC1B1	DC1B0	CCP1M3	CCP1M2	CCP1M1	CCP1M0	--00 0000
CCPR2H	Capture/Compare/PWM Register2 High Byte								xxxx xxxx
CCPR2L	Capture/Compare/PWM Register2 Low Byte								xxxx xxxx
CCP2CON	—	—	DC2B1	DC2B0	CCP2M3	CCP2M2	CCP2M1	CCP2M0	--00 0000
TMR3H	Timer3 Register High Byte								xxxx xxxx
TMR3L	Timer3 Register Low Byte								xxxx xxxx
T3CON	RD16	T3CCP2	T3CKPS1	T3CKPS0	T3CCP1	T3SYNC	TMR3CS	TMR3ON	0000 0000
SPBRG	USART1 Baud Rate Generator								0000 0000
RCREG	USART1 Receive Register								0000 0000
TXREG	USART1 Transmit Register								0000 0000
TXSTA	CSRC	TX9	TXEN	SYNC	—	BRGH	TRMT	TX9D	0000 -010
RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D	0000 000x
EEDR	Data EEPROM Address Register								0000 0000
EEDATA	Data EEPROM Data Register								0000 0000
ECON2	Data EEPROM Control Register 2 (not a physical register)								---- ----
ECON1	EEPGD	CFGS	—	FREE	WRERR	WREN	WR	RD	xx-0 x000
IPR2	—	—	—	EEIP	BCLIP	LVDIP	TMR3IP	CCP2IP	---1 1111
PIR2	—	—	—	EEIF	BCLIF	LVDIF	TMR3IF	CCP2IF	---0 0000
PIE2	—	—	—	EEIE	BCLIE	LVDIE	TMR3IE	CCP2IE	---0 0000
IPR1	PSPIP(3)	ADIP	RCIP	TXIP	SSPIP	CCP1IP	TMR2IP	TMR1IP	1111 1111
PIR1	PSPIF(3)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000
PIE1	PSPIE(3)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000
TRISE(3)	IBF	OBF	IBOV	PSPMODE	—	Data Direction bits for PORTE		0000 -111	
TRISD(3)	Data Direction bits for PORTD								1111 1111
TRISC	Data Direction bits for PORTC								1111 1111
TRISB	Data Direction bits for PORTB								1111 1111
TRISA	—	TRISA6(1)	Data Direction bits for PORTA					—	-111 1111
LATE(3)	—	—	—	—	—	Read PORTE Data Latch, Write PORTE Data Latch		---- -xxx	
LATD(3)	Read PORTD Data Latch, Write PORTD Data Latch								xxxx xxxx
LATC	Read PORTC Data Latch, Write PORTC Data Latch								xxxx xxxx
LATB	Read PORTB Data Latch, Write PORTB Data Latch								xxxx xxxx
LATA	—	LATA6(1)	Read PORTA Data Latch, Write PORTA Data Latch (1)					—	-xxx xxxx
PORTE(3)	Read PORTE pins, Write PORTE Data Latch								---- -000
PORTD(3)	Read PORTD pins, Write PORTD Data Latch								xxxx xxxx
PORTC	Read PORTC pins, Write PORTC Data Latch								xxxx xxxx
PORTB	Read PORTB pins, Write PORTB Data Latch								xxxx xxxx
PORTA	—	RA6(1)	Read PORTB pins, Write PORTB Data Latch (1)					—	-x0x 0000

Legend: x = unknown, u = unchanged, - = unimplemented, q = value depends on condition

Shaded cells are unimplemented, read as "0"

**Note 1:** RA6 and associated bits are configured as port pins in RCIO and ECIO Oscillator mode only and read '0' in all other Oscillator modes.

**Note 2:** Bit 21 of the TBLPTRU allows access to the device configuration bits

**Note 3:** These registers and bits are reserved on the PIC18F2X2 devices; always maintain these clear.

Figure redrawn by author from PIC18Fxx2 datasheet (DS39564B), Microchip Technology Inc.

**FIGURE A.8** Register file summary (part 2).<sup>6</sup>

<sup>6</sup> Figure A.8 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No



Address	Name	Address	Name	Address	Name	Address	Name
0xFFF	TOSU	0xFDF	INDF <sup>(3)</sup>	0xFBF	CCPR1H	0xF9F	IPR1
0xFFE	TOSH	0xFDE	POSTINC <sup>(3)</sup>	0xFBE	CCPR1L	0xF9E	PIR1
0xFFD	TOSL	0xFDD	POSTDEC <sup>(3)</sup>	0xFBD	CCP1CON	0xF9D	PIE1
0xFFC	STKPTR	0xFDC	PREINC <sup>(3)</sup>	0xFBC	CCPR2H	0xF9C	—
0xFFB	PCLATU	0xFDB	PLUSW <sup>(3)</sup>	0xFBB	CCPR2L	0xF9B	—
0xFFA	PCLATH	0xFDA	FSR2H	0xFBA	CCP2CON	0xF9A	—
0xFF9	PCL	0xFD9	FSR2L	0xFB9	—	0xF99	—
0xFF8	TBLPTRU	0xFD8	STATUS	0xFB8	—	0xF98	—
0xFF7	TBLPTRH	0xFD7	TMR0H	0xFB7	—	0xF97	—
0xFF6	TBLPTRL	0xFD6	TMR0L	0xFB6	—	0xF96	TRISE <sup>(2)</sup>
0xFF5	TABLAT	0xFD5	T0CON	0xFB5	—	0xF95	TRISD <sup>(2)</sup>
0xFF4	PRODH	0xFD4	—	0xFB4	—	0xF94	TRISC
0xFF3	PRODL	0xFD3	OSCCON	0xFB3	TMR3H	0xF93	TRISB
0xFF2	INTCON	0xFD2	LVDCON	0xFB2	TMR3L	0xF92	TRISA
0xFF1	INTCON2	0xFD1	WDTCON	0xFB1	T3CON	0xF91	—
0xFF0	INTCON3	0xFD0	RCON	0xFB0	—	0xF90	—
0xFEf	INDF <sup>(3)</sup>	0xFCF	TMR1H	0xFAF	SPBRG	0xF8F	—
0xFEE	POSTINC <sup>(3)</sup>	0xFCE	TMR1L	0xFAE	RCREG	0xF8E	—
0xFED	POSTDEC <sup>(3)</sup>	0xFCD	T1CON	0xFAD	TXREG	0xF8D	LATE <sup>(2)</sup>
0xFEC	PREINC <sup>(3)</sup>	0xFCC	TMR2	0xFAC	TXSTA	0xF8C	LATD <sup>(2)</sup>
0xFEB	PLUSW <sup>(3)</sup>	0xFCB	PR2	0xFAB	RCSTA	0xF8B	LATC
0xFEa	FSR0H	0xFCA	T2CON	0xFAA	—	0xF8A	LATB
0xFE9	FSR0L	0xFC9	SSPBUF	0xFA9	EADDR	0xF89	LATA
0xFE8	WREG	0xFC8	SSPADD	0xFA8	EEDATA	0xF88	—
0xFE7	INDF <sup>(3)</sup>	0xFC7	SSPSTAT	0xFA7	EECON2	0xF87	—
0xFE6	POSTINC <sup>(3)</sup>	0xFC6	SSPCON1	0xFA6	EECON1	0xF86	—
0xFE5	POSTDEC <sup>(3)</sup>	0xFC5	SSPCON2	0xFA5	—	0xF85	—
0xFE4	PREINC <sup>(3)</sup>	0xFC4	ADRESH	0xFA4	—	0xF84	PORTE <sup>(2)</sup>
0xFE3	PLUSW <sup>(3)</sup>	0xFC3	ADRESL	0xFA3	—	0xF83	PORTD <sup>(2)</sup>
0xFE2	FSR1H	0xFC2	ADCON0	0xFA2	IPR2	0xF82	PORTC
0xFE1	FSR1L	0xFC1	ADCON1	0xFA1	PIR2	0xF81	PORTB
0xFE0	BSR	0xFC0	—	0xFA0	PIE2	0xF80	PORTA

**Note 1:** Unimplemented registers are read as “0”.  
**Note 2:** This register is not available on PIC18Fxx2 devices.  
**Note 3:** This is not a physical register.

Figure redrawn by author from PIC18Fxx2 datasheet (DS39564B), Microchip Technology Inc.

**FIGURE A.9** Special function register map.<sup>7</sup>

<sup>7</sup> Figure A.9 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.’s prior written consent.

File Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Debug/ Unpgmed Value
0x300001 CONFIG1H	—	—	OSCSEN	—	—	FOSC2	FOSC1	FOSC0	--1- 1111
0x300002 CONFIG2L	—	—	—	—	BORV1	BORV0	BOREN	PWRTEN	---- 1111
0x300003 CONFIG2H	—	—	—	—	WDTPS2	WDTPS1	WDTPS0	WDTEN	---- 1111
0x300005 CONFIG3H	—	—	—	—	—	—	—	CCP2MX	---- -1-1
0x300006 CONFIG4L	DEBUG	—	—	—	—	LVP	—	STVREN	1--- -1-1
0x300008 CONFIG5L	—	—	—	—	CP3	CP2	CP1	CP0	---- 1111
0x300009 CONFIG5H	CPD	CPB	—	—	—	—	—	—	11-- ----
0x30000A CONFIG6L	—	—	—	—	WRT3	WRT2	WRT1	WRT0	---- 1111
0x30000B CONFIG6H	WRTD	WRTB	WRTC	—	—	—	—	—	111- ----
0x30000C CONFIG7L	—	—	—	—	EBTR3	EBTR2	EBTR1	EBTR0	---- 1111
0x30000D CONFIG7H	—	EBTRB	—	—	—	—	—	—	-1-- ----
0x3FFFE DEV1D1	DEV2	DEV1	DEV0	REV4	REV3	REV2	REV1	REV0	(1)
0x3FFFF DEV1D2	DEV10	DEV9	DEV8	DEV7	DEV6	DEV5	DEV4	DEV3	0000 0100

Legend: x = unknown, u = unchanged, - = unimplemented, q = value depends on condition

Shaded cells are unimplemented, read as "0"

**Note 1:** See Configuration bit definitions

Figure redrawn by author from PIC18Fxx2 datasheet (DS39564B), Microchip Technology Inc.

**FIGURE A.10** Configuration register summary.<sup>8</sup>

<sup>8</sup> Figure A.10 adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved. No further reprints or reproductions may be made without Microchip Inc.'s prior written consent.

Config. Bits and Device IDs	Definitions
<u>OSCSEN</u>	Allows switching main clock source between timer1 and main oscillator 1: Oscillator system clock option is disabled (main oscillator is source) 0: Oscillator system clock option is enabled (oscillator switching is enabled)
FOSC[2:0]	Oscillator Selection bits 111 = RC oscillator w/ OSC2 configured as RA6 110 = HS oscillator with PLL enabled/Clock frequency = (4 x FOSC0) 101 = EC oscillator w/ OSC2 configured as RA6 100 = EC oscillator w/ OSC2 configured as divide-by-4 clock output 011 = RC oscillator 010 = HS oscillator 001 = XT oscillator 000 = LP oscillator
BORV[1:0]	Brown-out Reset Voltage bits (11 = VBOR set to 2.5 V, 10 = VBOR set to 2.7 V, 01 = VBOR set to 4.2 V, 00 = VBOR set to 4.5 V)
BOREN	Brown-out Reset Enable bit (1 = Brown-out Reset enabled, 0 = Brown-out Reset disabled)
<u>PWRTEN</u>	Power-up Timer Enable bit 1 = Power-up Timer disabled, 0 = Power-up Timer enabled
WDTPS[2:0]	Watchdog Timer Postscale Select bits (111 = 1:128, 110 = 1:64, 101 = 1:32, 100 = 1:16, 011 = 1:8, 010 = 1:4, 001 = 1:2, 000 = 1:1)
WDTEN	Watchdog Timer Enable bit (1 = WDT enabled, 0 = WDT Disabled, control is placed on the SWDTEN bit)
CCP2MX	CCP2 Mux bit (1 = CCP2 input/output is multiplexed with RC1, 0 = CCP2 input/output is multiplexed with RB3)
<u>DEBUG</u>	Background Debugger Enable Bit 1 = Background Debugger disabled. RB6 and RB7 configured as general purpose I/O pins 0 = Background Debugger enabled. RB6 and RB7 are dedicated to In-Circuit Debug
LVP	Low Voltage ICSP Enable bit (1 = Low Voltage ICSP enabled, 0 = Low Voltage ICSP disabled) When enabled, V <sub>dd</sub> applied to RB5/PGM enters programming mode.
STRVREN	Stack Full/Underflow Reset Enable bit 1 = Stack Full/Underflow will cause RESET, 0 = Stack Full/Underflow will not cause RESET
CP[3:0]	Code Protect bits (used to disable external reads/writes of Program Memory, see PIC18F242 datasheet for details)
CPD	Data EEPROM Code Protect Bit (used to protect against external read/writes of Data EEPROM) 1 = Data EEPROM not code protected, 0 = Data EEPROM code protected
CPB	Boot Block (0x000000-0x0001FF) Code Protection Bit (protects against Boot Block external reads/writes) 1 = Boot Block not code protected, 0 = Boot Block code protected
WRT[3:0]	Write Protection Bits (used to protect code memory against table writes, see PIC18F242 datasheet for details)
WRTD	Data EEPROM Write Protection bit (protects against external writes of DATA EEPROM) 1 = Data EEPROM not write protected, 0 = Data EEPROM write protected
WRTB	Boot Block (0x000000-0x0001FF) Code Protection Bit (protects against Block Block external writes) 1 = Boot Block not write protected, 0 = Boot Block write protected
WRTC	Configuration Registers (0x300000-3000FF) Write Protection bit 1 = Configuration registers not write protected, 0 = Configuration registers write protected
EBTR[3:0]	Table Read Protection bits (used to disable table reads of program memory, see PIC18F242 datasheet for details)
EBTRB	Boot Block (0x000000-0x0001FF) Table Read Protection bit 1 = Boot Block not protected from table reads executed in other blocks 0 = Boot Block protected from table reads executed in other blocks
DEV[2:0]	Device ID bits (000 = PIC18F252, 001 = PIC18F452, 100 = PIC18F242, 101 = PIC18F442)
REV[4:0]	Revision ID bits

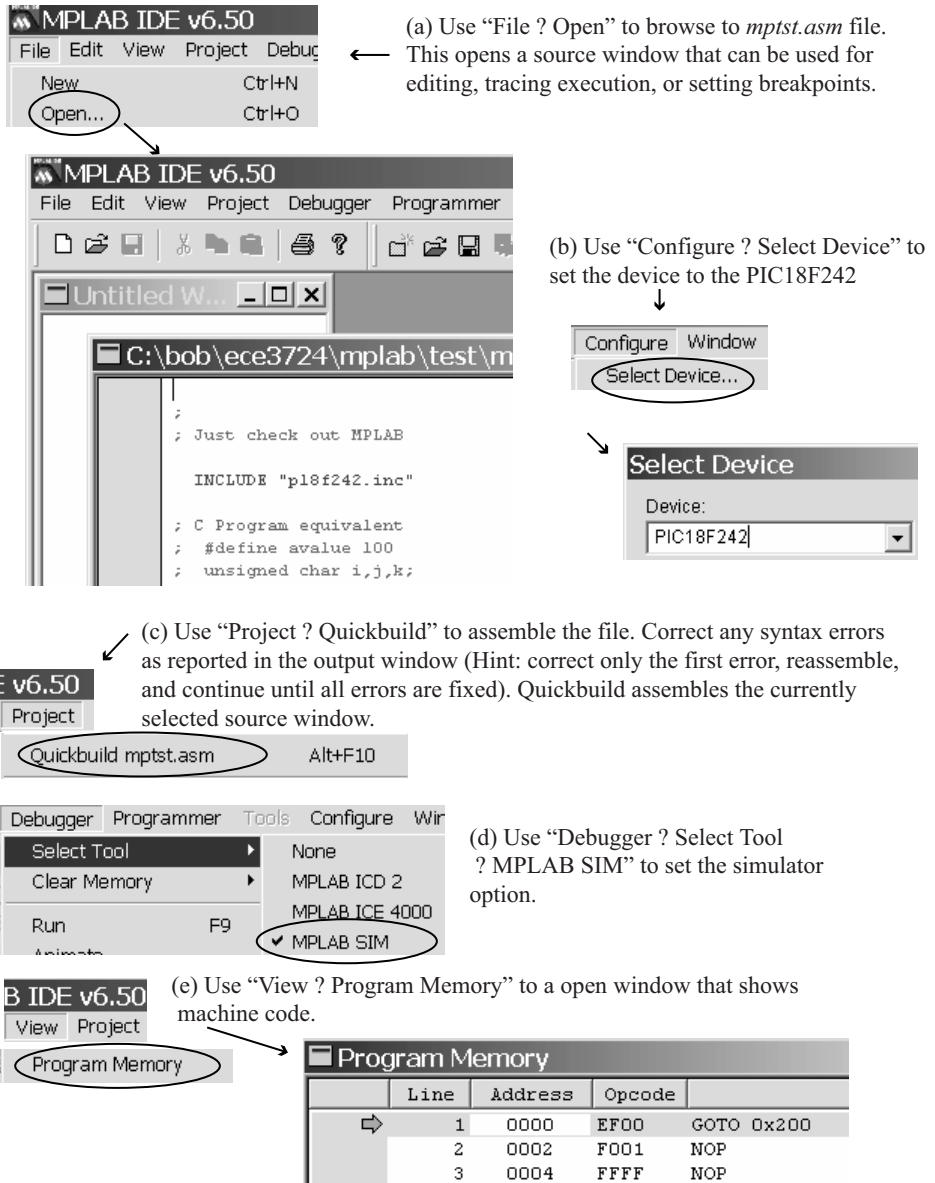
**FIGURE A.11** Configuration register bit definitions.

# B

## Microchip MPLAB Quickstart

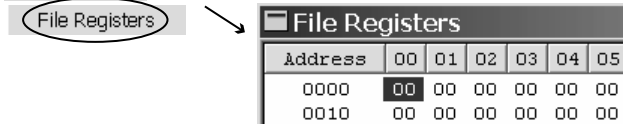


This appendix contains a brief introduction to the Microchip MPLAB integrated design environment. The sample program used is the *mptst.asm* program (Listing 3.9) found in the *code/labs* directory on the companion CD-ROM. The latest version of MPLAB can be downloaded from the Microchip Web site ([www.microchip.com](http://www.microchip.com)). Figures B.1, B.2, and B.3 give the most common MPLAB commands for assembling and simulating the assembly language programs in this book. Figure B.1 shows how to use the *Quickbuild* command to assemble an assembly language source file. Figure B.2 shows how to use the simulator/debugger within MPLAB. Figure B.3 shows how to use the stopwatch function to measure code execution. See Appendix C, “HI-TECH PICC-18 C Compiler Demo for the PIC18F242,” for information on how to build a project that contains multiple source files within MPLAB.

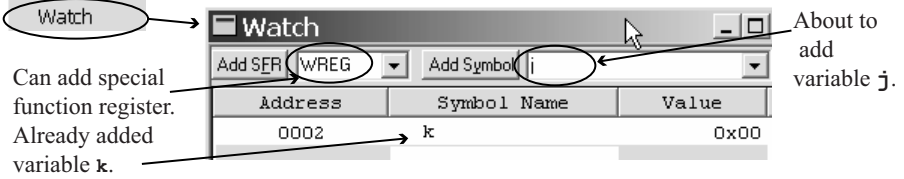


**FIGURE B.1** Using Quickbuild to assemble a file. Screenshots ©2005 Microchip Technology, Inc. Reprinted with permission. All rights reserved.

(f) Use “View ? File Registers” to open a window that shows data memory. These contents are zero initially.



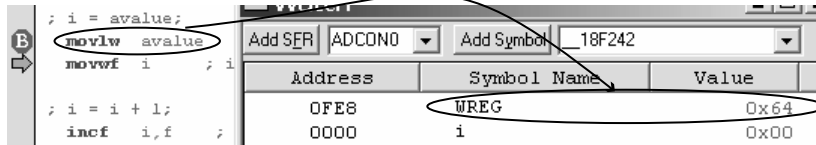
(g) Use “View ? Watch” to open a window for watching data memory or special function register values as they change.



(h) Right click in source window to bring up menu that allows a breakpoint to be set/cleared at the current cursor location.



(i) Use F7 to single step through program; green arrow in source window tracks program execution. Use F6 to execute a processor reset.

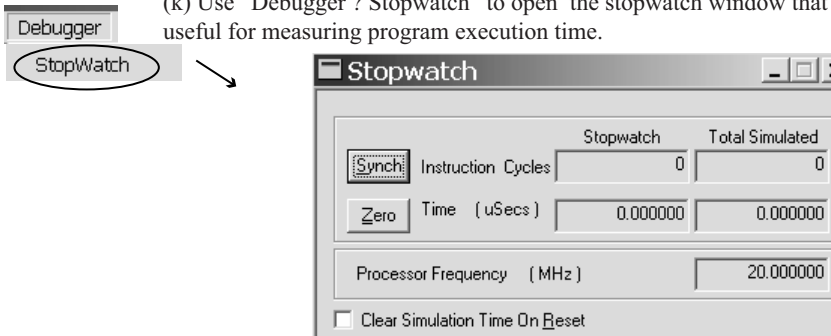


(j) “Debugger ? Clear Memory ? All Memory” is useful for returning memory to its initial state. The program must be recompiled if all memory is cleared.

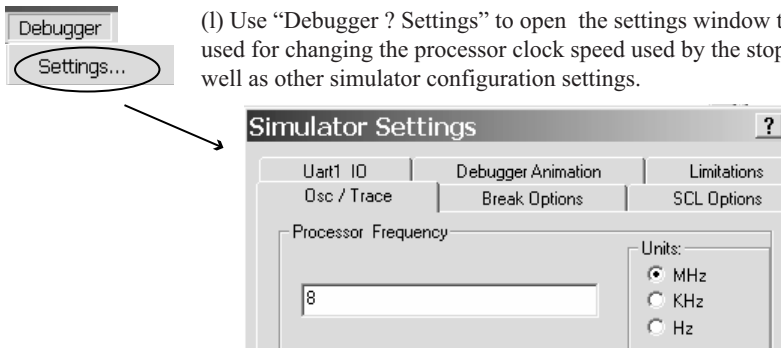


**FIGURE B.2** Using the MPLAB SIM debugger. Screenshots ©2005 Microchip Technology, Inc. Reprinted with permission. All rights reserved.

(k) Use “Debugger ? Stopwatch” to open the stopwatch window that is useful for measuring program execution time.



(l) Use “Debugger ? Settings” to open the settings window that is used for changing the processor clock speed used by the stopwatch, as well as other simulator configuration settings.



**FIGURE B.3** Using the MPLAB SIM stopwatch. Screenshots ©2005 Microchip Technology, Inc. Reprinted with permission. All rights reserved.

# C

## HI-TECH PICC-18 C Compiler Demo for the PIC18F242



This appendix contains a brief introduction to the HI-TECH PICC-18 C compiler demo that is found in the *hitech* subdirectory on this book's companion CD-ROM. The PICC-18 C compiler is enabled for 120 days after installation and only produces code for the 18F242. The executable found in the *hitech* subdirectory is self-installing; MPLAB should be installed before installing the PICC-18 compiler. Some compiler options are disabled in this demo compiler. These options are:

- -D to predefine macro symbols
- -L to specify libraries to be scanned by the linker
- -L-opt to allow specification of additional linker options
- -O-opt to allow specification of additional OBJTOHEX options
- -M map file generation
- -NORT to disable runtime code inclusion
- -PRE to only preprocess source files
- -RESRAM to reserve RAM ranges
- -RESROM to reserve ROM ranges
- -U to undefine macro symbols
- -V for verbose compilation messages

In practice, the only effect of the demo limitations on the C programs included in this book is the lack of `printf` support for floats and longs, due to the omission of the `-L` compiler flag. Only a few book examples use `printf` statements with float/long data types for program output, and the lack of support does not affect the overall program functionality.

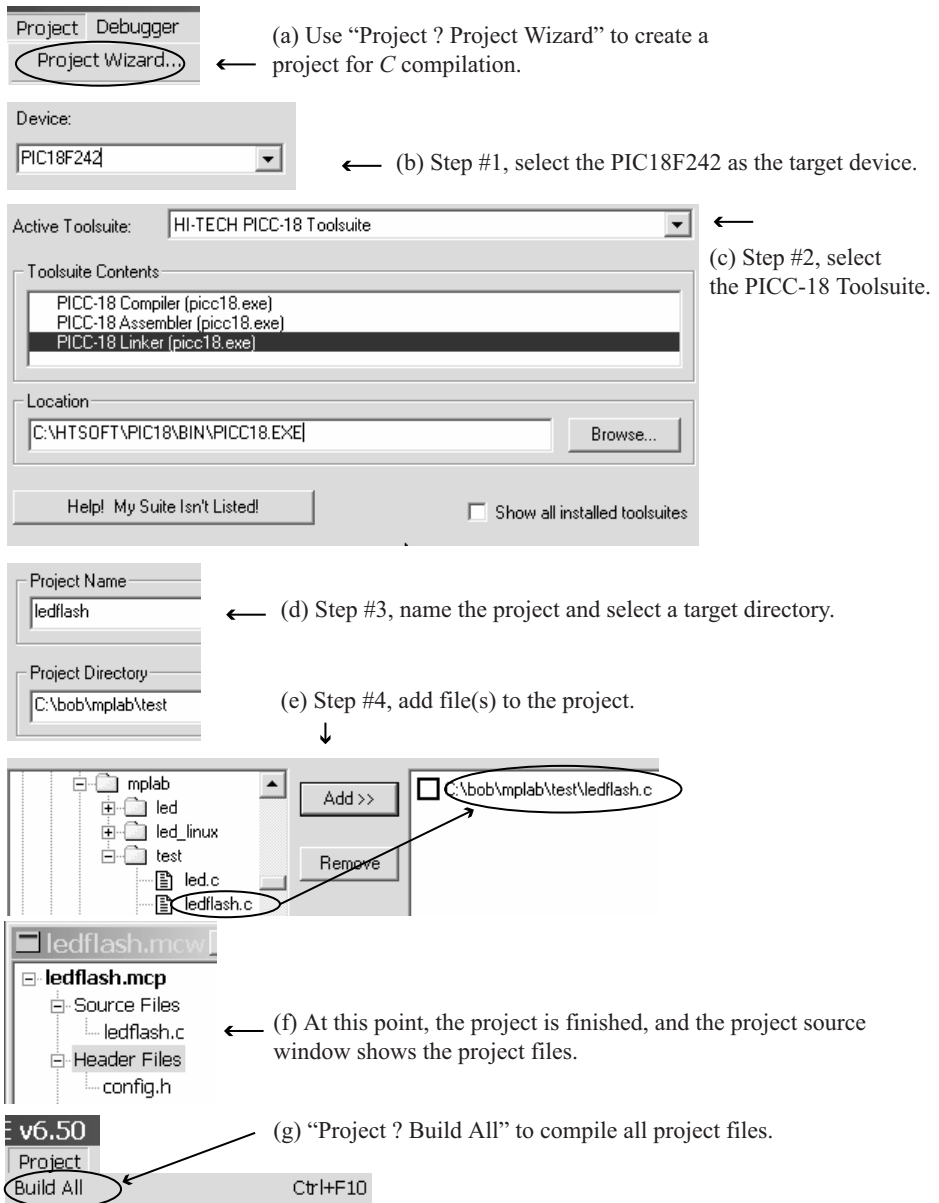
During installation of the PICC-18 demo compiler, you will be asked if you want to install the MPLAB™ 6 PICC-18 Toolsuite Plugin. You should answer “yes” to this question, as this allows you to build projects within MPLAB that specify the PICC-18 compiler toolsuite. This plugin has been tested to be compatible with MPLAB 7 as well.



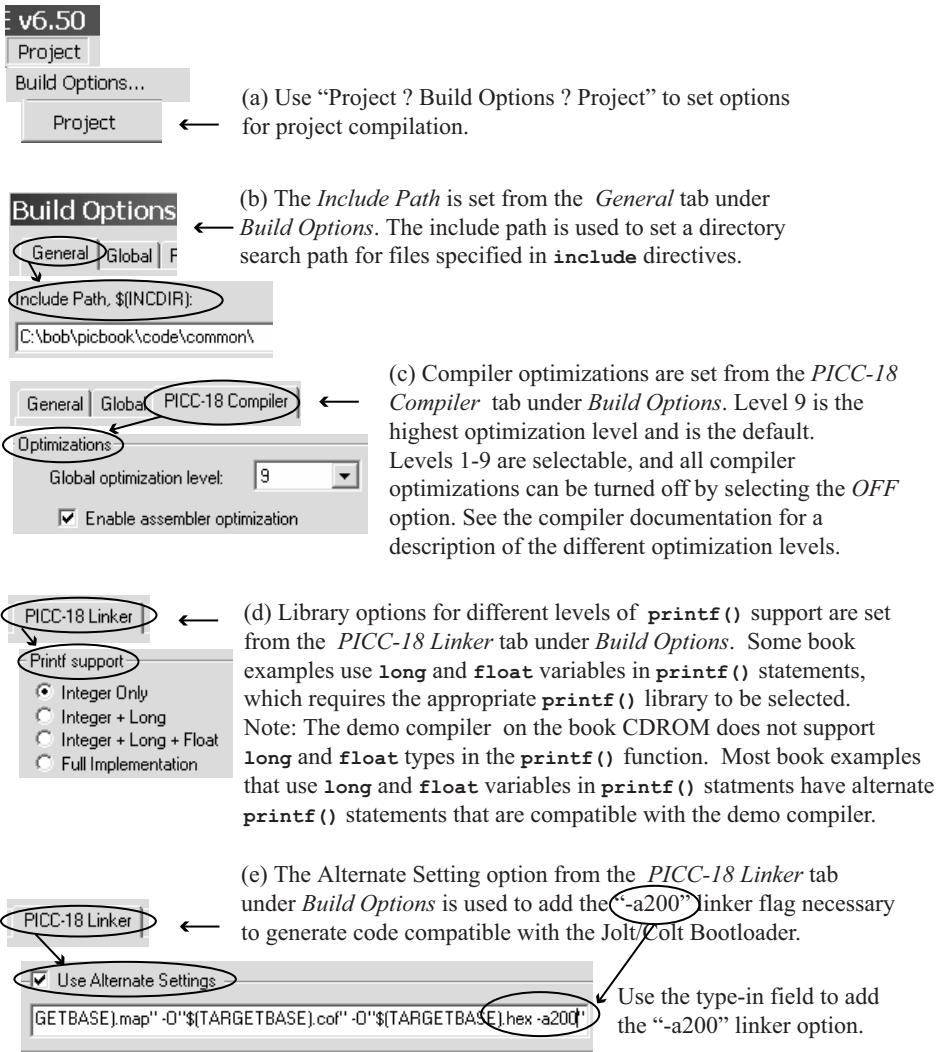
Figure C.1 shows the steps required to create a project for C compilation using the *ledflash.c* example. The PICC-18 tool suite is chosen in Figure C.1c; for an assembly language project you would choose the Microchip MPASM™ tool suite. In Figure C.1d, the project name determines the name of the final hex file that is produced (*project\_name.hex*). The files *ledflash.c* and *config.h* are added to the project in Figure C.1e. The project is compiled by using the “Project → Build All” command in Figure C.1g.



Figure C.2 shows how to modify common compiler and linker options for a project. All of the C source examples on the companion CD-ROM that contain the `main()` entry function use `#include` statements to include other required source and header files. This means that the only file that actually has to be entered into the project list is the file that contains the `main()` entry function assuming the *include file path* is set to the directory that contains the included files. Figure C.2b shows how to set the directory search path for `#include` files. Different levels of compiler optimizations can be selected as shown in Figure C.2c. A few book examples use `long` and `float` variables within `printf()` statements; Figure C.2d shows how to select the appropriate `printf()` library assuming a full version of the compiler. Code that is to be downloaded into the PIC18F242 using the Jolt/Colt serial bootloader (Appendix F) must have the “-a200” linker option specified as shown in Figure C.2e. This causes the generated code to be offset in memory by 0x0200 locations; this is required as the bootloader firmware resides in locations 0x000 to 0x01FF.



**FIGURE C.1** Creating a project. Screenshots ©2005 Microchip Technology, Inc. Reprinted with permission. All rights reserved.



(a) Use “Project ? Build Options ? Project” to set options for project compilation.

(b) The *Include Path* is set from the *General* tab under *Build Options*. The include path is used to set a directory search path for files specified in `include` directives.

(c) Compiler optimizations are set from the *PICC-18 Compiler* tab under *Build Options*. Level 9 is the highest optimization level and is the default. Levels 1-9 are selectable, and all compiler optimizations can be turned off by selecting the *OFF* option. See the compiler documentation for a description of the different optimization levels.

(d) Library options for different levels of `printf()` support are set from the *PICC-18 Linker* tab under *Build Options*. Some book examples use `long` and `float` variables in `printf()` statements, which requires the appropriate `printf()` library to be selected. Note: The demo compiler on the book CDROM does not support `long` and `float` types in the `printf()` function. Most book examples that use `long` and `float` variables in `printf()` statements have alternate `printf()` statements that are compatible with the demo compiler.

(e) The Alternate Setting option from the *PICC-18 Linker* tab under *Build Options* is used to add the “-a200” linker flag necessary to generate code compatible with the Jolt/Colt Bootloader.

Use the type-in field to add the “-a200” linker option.

**FIGURE C.2** Compiler/linker options. Screenshots ©2005 Microchip Technology, Inc. Reprinted with permission. All rights reserved.

  
**D****Notes on the C Language**

This appendix contains a few brief notes on the C programming language based on questions asked by students when performing the laboratory exercises in Appendix E, “Suggested Laboratory Exercises.” This book only covers a subset of the C language and the coverage is intended to be adequate for a student already conversant in some other high-level language, either object oriented (e.g., C++, Java) or procedural (e.g., BASIC, Pascal).

**D.1 FORMATTED IO (*PRINTF*, *SCANF*, *SPRINTF*, *SSCANF*)**

---

The formatted IO statements `printf()` and `scanf()` can be confusing if you are new to C. This appendix only covers the formatted IO features used in this book’s examples; please refer to a C textbook if you require a more complete description. The `printf()` statement is used for formatted ASCII output; within the PICC-18 C compiler environment the `printf()` library function calls the `putch()` function to output each ASCII byte. Chapter 9, “Asynchronous Serial IO,” gives a `putch()` function that outputs the byte to the PIC18Fxx2 serial port. The parameters to the `printf()` function are a format string and an argument list. The format string can contain a mixture of normal characters and conversion specifications; a conversion specification determines how an argument value is converted to an ASCII format. Conversion specifications begin with a “%” character; two “%” characters in a row are needed if a “%” is desired in the final output string. Figure D.1a gives some examples of `printf()` conversion specifications.

```
char c;
c = 0x41;
printf("%c %d %x", c, c, c);
```

(a) Some format specifications. Replace %d with %u for unsigned int types. Replace %d with %ld for long types, and %d with %lu for unsigned long types.

A As an ASCII character      65 In decimal      41 In hex

```
printf("A:%10d B:%-10d C:%04x \n", c, c, c);
```

A:                  65    B: 65                  C:0041

Field width is 10 digits, left justified      Field width is 10 digits, right justified      Field width is 4 characters, pad with '0' digits

(b) Floating point format specifications

```
float q;
q = 0.001234;
printf("A: %6.3f B: %f C: %e\n", q, q, q);
```

A: 0.001      B: 0.001234      C: 1.234000e-03

Field width is 6 digits, with three digits to right of decimal point      No field width or precision specified      Scientific notation

(c) print to an in-memory buffer

```
char c, buf[20];
c = 0x41;
sprintf(buf, "%c %d %x", c, c, c);
```

← after execution, buf contains  
A 65 41

**FIGURE D.1** printf() examples.

A %c formats an argument as a single ASCII character, %d specifies an ASCII decimal format, and %x is used to format an ASCII hex number. The field width is specified as a number before the format character; %10d specifies a field width of 10 digits. Left justification is the default; a negative sign in the field width (e.g., %-10d) performs right justification. A leading zero can be specified in the field width as in %04x to pad the number with leading zeros. A leading 1 (letter l) is used in the format specification if the argument is a long rather than an int (i.e., %ld instead of %d). A %u can be specified in place of a %d for an unsigned data type. Floating-point (float or double) format specifications use %f or %e as shown in Figure D.1b. A format specification of the form %n.mf specifies a field width of n digits, with m digits

of precision to the right of the decimal point. The `%e` specification causes the floating-point number to be printed in scientific notation. The `sprintf()` function is used to print the string to an in-memory buffer; the first argument to `sprintf()` is the char buffer for the output string as shown in Figure D.1c.

The `scanf()` function is used for formatted ASCII input; within the PICC-18 C compiler environment the `scanf()` library function calls the `getche()` function (get character and echo) to input each ASCII byte. The `getche()` function is assumed to echo its character using the `putch()` function. The `scanf()` function expects *pointers* to the variables in the parameter list; hence, the `&` operator (*address-of operator*) is placed in front of any variables that are passed to `scanf()`. Figure D.2 shows examples of `scanf()` with `%d` (decimal), `%x` (hex), and `%f` (float) numbers. The leading `0x` on the hex number input is optional. The `scanf()` function skips over any white space (space characters, tabs, etc.) it encounters when scanning the input for a match to the format specification. The `sscanf()` (string `scanf()`) function can be used to read values from an in-memory buffer; the first argument to `sscanf()` is the char buffer to scan.

```

unsigned char c;
float q;

printf("Enter decimal number: ");
scanf("%d", &c);
printf("The number is %d\n", c);

printf("\nEnter hex number: ");
scanf("%x", &c);
printf("The hex number is %x\n", c);

printf("\nEnter a float number: ");
scanf("%f", &q);
printf("The float number is %f\n", q);

```

<pre> printf("Enter decimal number: "); scanf("%d", &amp;c); printf("The number is %d\n", c); </pre>	→	<pre> Enter decimal number: 49 The number is 49 </pre>
<pre> printf("\nEnter hex number: "); scanf("%x", &amp;c); printf("The hex number is %x\n", c); </pre>	→	<pre> Enter hex number: 0xEC The hex number is ec </pre>
<pre> printf("\nEnter a float number: "); scanf("%f", &amp;q); printf("The float number is %f\n", q); </pre>	→	<pre> Enter a float number: 1.239202 The float number is 1.239202 </pre>

**FIGURE D.2** `scanf()` examples.

## **D.2 FOR C++ PROGRAMMERS**

---

A simple, but common error made by C++ programmers when adjusting to C is variable declarations. In C++, variable declarations can be placed anywhere within a function as shown in Listing D.1.

**LISTING D.1** C++ Variable declarations.

---

```

main() {    // compiled with a C++ compiler
    char c;
    c = 0x41;
    c++;

    int i;    //variable declaration
    i++;
}

```

However, in *C* the second variable declaration `int i` must either go at the top of the function with the `char c` declaration or be enclosed in a block using `{}` as shown in Listing D.2. However, be careful—variables declared within a block `{}` are not visible to statements outside of that block.

**LISTING D.2** C Variable declarations.

---

```

main() {    //compiled with C compiler
    char c;
    c = 0x41;
    c++;
    { // be careful, 'i' scope is limited to within brackets!
        int i;    //variable declaration
        i++;
    }
}

```

**D.3 FOR NEW PROGRAMMERS**


---

If you don't have much programming experience, the number of syntax errors reported by the *C* compiler after compiling your first program may overwhelm you. A useful tip relevant to almost any programming language is to fix only the first syntax error; many of the remaining errors are most probably side effects of that first syntax error. Do not be intimidated by a listing of 100+ errors; concentrate on fixing the first one and many of the remaining errors will vanish on the next compile.

**D.4 FOR EXPERIENCED C PROGRAMMERS**


---

This book's usage of the *C* language is kept fairly simple to aid the new *C* programmer's understanding of the example programs. If you are comfortable with the *C* language, you are encouraged to modify the book's examples to make use of the more powerful features of the language. For example, macros can be employed to

automatically compute the numbers for the serial port baud rate, I<sup>2</sup>C bus speed, timer interrupt periods, and so forth to make the examples independent of a particular FOSC value. In some of the more complex examples, use of struct data types for related variables may be appropriate.

If you are an experienced C programmer (perhaps from an X86 platform), there are many shortcuts for certain code segments discussed in book examples. For example, the code in Chapter 10, “Interrupts and a First Look at Timers,” for placing data into a software FIFO is written as:

```
head = head + 1;
if (head == BUFMAX) head = 0;
ibuf[head] = RCREG
```

Here, BUFMAX is a power of 2, and head is a char variable. An experienced C programmer might write instead:

```
ibuf[(++head)%BUFMAX] = RCREG;
```

This works because % is the C modulo operator and BUFMAX is a power of 2, meaning that it is evenly divisible into 256, which is the number of possible code values for the head variable. While the number of C statements is reduced from 3 to 1, the amount of assembly code generated is much higher as the modulo operation requires a division operation, an expensive operation in terms of machine code and instruction cycles on the PIC18. So, be careful when writing “tight” C code—it might not translate into “tight” machine code once it is compiled.



*This page intentionally left blank*

# E

## Suggested Laboratory Exercises

This appendix contains the laboratory exercises for a semester-length course taught at Mississippi State University since summer 2004. We first began teaching a PIC-based introductory microprocessor course in fall 2003 using the PIC16F873, and switched to the PIC18F242 in summer 2004.

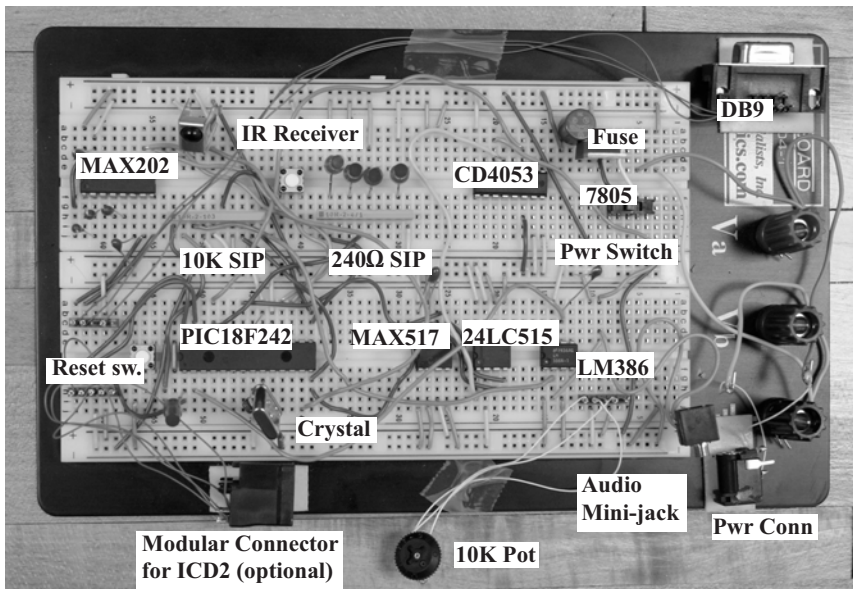
### E.1 LAB SETUP

Table E.1 lists the lab equipment assumed by these experiments. Each lab station should have an oscilloscope and a multimeter. In addition, if each student does not have a portable PC, every lab station must have a desktop PC. Each lab station should either have a LAN connection or the lab should be wireless enabled.

**TABLE E.1** Suggested Lab Equipment

Equipment	Comment
Networked PC	Lab station only needs a LAN connection if student has a portable PC.
Multimeter	Basic instrumentation.
Oscilloscope	Basic instrumentation.
PICSTART Programmer	This is used for programming PICs with the serial bootloader and initial test programs; it can be shared among multiple stations.
Soldering and wire wrap tools/supplies	Either soldering or wire wrap is used for external connectors; multiple stations can share this.
Universal remote control	Used by Experiment 12 for IR waveform decoding.

Because a serial bootloader is used to program the PIC18F242 in a majority of the hardware labs, each PC must either have a serial port or a USB-to-serial port adapter. The lab also must have some method of programming a PIC18F242 without a serial bootloader; we use a Microchip PICSTART Plus programmer (Appendix F, “The Jolt/Colt Serial Bootloaders”) shared among 10 lab stations. Software installed on each PC to support the PIC18F242 experiments are MPLAB, the HI-TECH PICC-18 compiler, and the Jolt/Colt bootloaders (Appendix F). The lab experiments consist of 13 experiments: one digital-logic based, four assembly language based, and eight hardware based. In the last week of the semester we hold a lab practicum to evaluate student skills for ABET assessment purposes. The hardware experiments use a parts kit and a prototyping board purchased by each student. Through the eight hardware experiments, a student builds a PIC18F242 system that has an external I<sup>2</sup>C serial EEPROM, I<sup>2</sup>C DAC, asynchronous serial interface, a potentiometer, a mini-jack for audio input/output, and an IR receiver. Wire wrap is used for external connectors like the DB9 required for the serial port; the lab has wire wrap supplies (tools/wire) shared by the students. The lab also has a couple of soldering irons for creating reliable connections to external connectors if students prefer that over wire wrap. Figure E.1 shows a picture of the prototyping board at the end of the semester after all experiments have been added to the board.



**FIGURE E.1** Protoboard at semester end.

The parts kit list used during the fall 2004 semester is shown in Figure E.2 along with supplier part numbers. We purchase the part kits pre-assembled from Electronix Express ([www.elexp.com](http://www.elexp.com)) at a substantially reduced price over what can be ordered by students in single quantities from parts suppliers. The protoboard and wiring kit are usually purchased in a previous digital logic course. The modular connector was previously included to support in-circuit programming via the Microchip ICD2 programmer (Appendix F), but this has been dropped and is now optional.

Quantity	Part Description	Part #
1	PIC 18LF242	PIC18LF242-I/SP-ND (DK)
1	7.3728 MHz Crystal	X019-ND (DK)
2	15 pF ceramic capacitors, radial	495-1002-1-ND (DK)
1	LM 386 Audio Amp	LM386N-1-ND (DK)
1	MAX 517 D/A	MAX517BCPA-ND (DK)
1	24LC515 Serial EEPROM	24LC515-I/P-ND (DK)
1	Audio 2.5 mm mono jack	CP-2506-ND (DK)
1	Slide switch for power	EG1903-ND (DK)
1	10 k $\Omega$ Potentiometer	P4C1103-ND (DK)
2	500 mA fast-acting PCB Fuse	WK3041BK-ND (DK)
2	Pushbutton momentary on	P8009S-ND (DK)
1	DB-9 Female	182-709F-ND (DK)
3	5 Position Header	WM4003-ND (DK)
1	7805 5 V Regulator	LM340T-5.0-ND (DK)
1	Wall Transformer 9 V	T514-P5P-ND (DK)
1	10 k $\Omega$ SIP, 5 isolated resistors	4310R-2-103 (DK)
1	470 $\Omega$ SIP, 5 isolated resistors	4310R-2-471 (DK)
1	Power Connector	CP-006A-ND (DK)
4	Red LED	P428-ND (DK)
1	Green LED	P429-ND (DK)
1	CD4053BE Analog Mux	296-2059-5-ND (DK)
1	MAX202 RS232 Interface	MAX202CPE-ND (DK)
8	0.1 $\mu$ F tantalum capacitors, radial	P2053-ND (DK)
1	IR Receiver Module	276-640 (Radio Shack)
1	Protoboard	WB-104-2+J (Web-tronics.com)
1	Extra Wiring Kit (optional)	WK-1 (Web-tronics.com)
1	6 ft DB9 Male to DB9 Female serial cable (optional)	25700 (Jameco)
1	Wire Wrap Tool (optional)	276-1570 (Radio Shack)
1	Modular Connector (optional)	A9084-ND (DK)

**FIGURE E.2** Parts kit list.

Based on MSU experience, the following points are key for a successful lab course using this approach:

- It is best if students have had some previous experience with protoboards before this lab. In the MSU ECE/CSE curriculum, students are required to take a digital logic course as a prerequisite with the digital lab experience providing them with some protoboard experience in wiring 74XX logic. This gives them an introduction to DIP packages, how protoboard wiring works, an introduction to an oscilloscope and a multimeter, and some basic circuit debugging (which is greatly increased in this course!). A circuits course is *not* a prerequisite except for circuit fundamentals as presented in a physics course; we have majors from electrical engineering, computer engineering, computer science, and software engineering who take this lab so a circuits prerequisite is not possible.
- The teaching assistants (TAs) for the lab must be talented, knowledgeable, dedicated, patient, and have had previous microcontroller experience to assist students in the inevitable hardware debugging problems. We recruit our TAs from the graduates of this course; the first TAs came from a traditional microcontroller course that has since been replaced by our second course in embedded systems. We limit enrollment in each lab section to 10 students because the hardware labs dramatically increase the “help me!” load on the TA. The first two hardware labs are the worst from a TA load perspective when students are bringing their initial PIC18F242 setup to life; our labs are open so students can work on their protoboards outside of normal lab hours if needed.
- The TA must have a reference board built for demo purposes and for checking bad part problems by plugging the part into the reference board.
- A spare parts supply is an absolute necessity, as students demonstrate an amazing talent for destroying ICs. The TA, course instructor, IEEE/HKN, or some central shop can sell these. Fuses have been our top sellers.

It is rewarding to watch students with near-zero prototyping skills gain confidence as the semester progresses, and show pride in their board as they build it up from a collection of ICs and wires to something that records and plays audio in the final experiment.



In the following experiment descriptions, the *prelab* activity should be completed before the student enters the lab. All of the files and zip archives referenced in the lab descriptions are contained in the *code/labs* directory on the companion CD-ROM.

## **E.2 EXPERIMENT 1: A STORED PROGRAM MACHINE (CHAPTERS 1, 2)**

This experiment has the students implement the Number Sequencing Computer from Chapter 2 using the digital logic simulation package from a previous digital

logic course. At MSU, this is the Altera Maxplus package. We give the students the design already entered in Altera Maxplus and have them write a program that outputs their own student ID number or phone number. The ROM module in Altera MAXPLUS reads its initial contents from an external file; the students must assemble their program and modify the ROM file that specifies the machine code for their program. Students also step through the logic simulation within Altera MAXPLUS and determine how many clock cycles it takes to output the two number sequences based on the LOC input. The *nsc.zip* archive on the book's supplemental lab Web site ([www.reesemicro.com](http://www.reesemicro.com)) contains the Altera Maxplus design for the number sequencing computer in Chapter 2. In this lab, the TAs also assist the students in installing MPLAB, the Jolt Bootloader, and the PICC18 compiler on their portable PCs.

### **E.3 EXPERIMENT 2: PIC18XX2 INTRODUCTION (CHAPTER 3)**

This experiment introduces the student to the PIC18Fxx2 instructions covered in Chapter 3 and the MPLAB environment.

#### **Prelab**

1. Assemble and simulate the “Simple” example in Listing 3.9 within MPLAB (filename is *mptst.asm*). Verify that the final value of  $k$  (location 0x02) contains the expected value of 0xC9 when the `goto here` statement is reached.
2. Change the value of the *avalue* symbol to be the decimal equivalent of the last two digits of your student ID. If the last two digits of your student ID are “00”, use the first two digits. Compute the new expected value of  $k$  when the `goto here` statement is reached; verify this value by assembling and simulating the program. Capture a screen shot that shows the program memory and file register contents when the `goto here` statement is reached.

#### **Lab Activity**

##### **Exploring Data Memory Storage**

1. Modify the *mptst.asm* file so that the `CBLOCK` starting location is 0x80. Re-assemble and re-execute the program. Verify that the `i`, `j`, `k` variables are updated at locations 0x80-0x82. When the program memory is viewed, the instruction at program location 0x202 is now different from the original

*mptst.asm* file. Record this difference, and determine the reason for this behavior.

2. Modify the *mptst.asm* file so that the `CBLOCK` starting location is `0x100`. Re-assemble and re-execute the program. Use the File Registers window to determine what locations are changed when the `i`, `j`, `k` variables are modified. Record these locations and determine the reason for this behavior. Hint: Look at the definition of the BSR (Bank Select Register).
3. To the program in step 2, add the statement `movlb 0x1` just after the `main` label and before the statement `movlw myid`. Re-assemble and re-execute the program. The `movlb` instruction is “Move Literal to Bank Select Register”; `movlb 0x1` moves the value `0x1` to the BSR (BSR = Bank Select Register). Use the File Registers window to determine what locations are changed when the `i`, `j`, `k` variables are modified. Record these locations and determine the reason for this behavior.
4. Based on the knowledge gained from steps 2 and 3, modify the *mptst.asm* program so that the `i`, `j`, `k` variables are located at `0x200-0x202`. Re-assemble, re-execute, and verify that these locations are modified.

### Exploring the Data Movement Instructions

1. In MPLAB, open a File Registers window that shows locations `0x0000` through `0x0120`. Click on the first line of “...” under the ASCII column and type the first five letters of your first name, and the seven digits of your phone number. Then, type enough spaces so that the entire type-in column has been filled. Record what appears in the hex locations `0x0` through `0x0F`. Justify the reason for these values (and explain in the lab report).
2. Using the *mptst.asm* file as a starting point, create a new program with a series of data movement instructions that moves the contents of locations `0x0-0x7` to locations `0x100-0x107`. Verify the operation of this program by typing data of your choice into locations `0x0-0x7`, executing your program, and verifying that the data is moved to locations `0x100-0x107`. Use `movlb` and `movf/movwf` combinations to do this; you may *not* use the `movff` instruction in this program. Starting at location `0x200` in Program Memory, determine the number of bytes needed for this program (most PIC18 instructions need 2 bytes, some need 4 bytes).
3. Create a new version of the program done in step 2 that uses the `movff` instruction to accomplish the data movement (i.e., `movff 0x0, 0x100` moves the contents of location `0x0` to `0x100`). Re-assemble, and verify that your program exhibits the same behavior as the program in step 2. Determine the number of bytes required for this program in the same manner as done in step 2. Which program took fewer bytes?

**Number Sequencing Computer Revisited**

The *shownum.asm* program emulates the execution of the Number Sequencing Computer in Chapter 2 by implementing the program of Listing 2.1. The *loc* input and *dout* output are emulated via memory locations.

- Single step through *shownum.asm* in MPLAB for both local and nonlocal number sequences. Note that location 0x0 corresponds to *loc*, and 0x01 to *dout*. Initially, *loc* is cleared via `c1rf loc, r` forcing the entire number sequence to be copied to *dout*. To force the shorter number sequence, either manually edit the value of *loc* in the file registers or insert the statement `bsf loc, 0` to set the LSB of *loc* to 1, giving *loc* a nonzero value.
- Modify *shownum.asm* to display the digits of your phone number (or some reasonable facsimile). Assemble and verify the operation.
- Use the *Stopwatch* command in the simulator and record the instruction cycles and time for displaying the full number sequence for a clock speed of 20 MHz. Use the “Debugger→settings” menu to change the clock speed to 8 MHz and do the same. Verify that your recorded times match predicted times by adding up the instruction cycle counts of each instruction, and multiply by 4x the clock period (each instruction cycle is 4 clock periods).
- Repeat #3 for displaying the truncated number sequence (*loc* is nonzero).

**E.4 EXPERIMENT 3: UNSIGNED 8-BIT OPERATIONS (CHAPTER 4)**

This experiment explores 8-bit unsigned arithmetic, logical, and shift operations (Chapter 4). This experiment assumes the students have access to an x86 C compiler on either a Windows or Linux machine to verify C code results.

**Prelab**

The *op.zip* archive contains five C program variations (*ops\_var{1-5}.c*); a typical one is shown in Listing E.1. The C source code has `printf()` statements after each computation; these have been removed in Listing E.1 for space reasons. The program variations re-arrange the order of the statements.

**LISTING E.1** Unsigned 8-bit logical, arithmetic, shift operations.

```
#include <stdio.h>
#define bitset(var,bitno) ((var)|=(1<<(bitno)))
#define bitclr(var,bitno) ((var)&=~(1<<(bitno)))
#define bittst(var,bitno) (var & (1 << (bitno)))
```



```

unsigned char i,j,k;

main() {
    i = 0x24; j = 0xC5; k = 0x4D;
    k++; // (a) increment
    i--; // (b) decrement
    k = j & i; // (c) bitwise and
    i = i >> 3; // (d) right shift by 3
    k = ~k; // (e) bitwise negation
    k = k ^ j; // (f) bitwise XOR
    i = j - k; // (g) subtraction
    j = j << 2; // (h) left shift by 2
    j = j | 0xC0; // (i) bitwise OR
    k = k + j; // (j) addition
    k = bitset(k,3); // (k) set a bit to '1' */
    j = bitclr(j,5); // (l) clear a bit to '0'
    printf ("i: 0x%x, j: 0x%x, k: 0x%x\n",i,j,k);
}

```

1. For your assignment, if the last digit of your student ID is 0 or 1, use *ops\_var1.c*; if it is 2 or 3, use *ops\_var2.c*, and so forth. Compile this program with your favorite C compiler and note the value of *i*, *j*, *k* at each step. The *bitset*, *bitclr* macros simply implement the equivalent *bsf* and *bcf* PIC18 operations.
2. The *loop.zip* archive contains five C program variations (*loops\_var{1-5}.c*); the *main()* code for a typical one is shown in Listing E.2. The program variations re-arrange the order of blocks a, b, c, d, and e.

---

#### LISTING E.2 Conditional operations.

```

unsigned char i,j,k;
main(){
    i = 0x24; j = 0x45; k = 0xFD;
    // begin 'a'
    while ( !bittst(j,7) ) { j = j << 1; }
    printf("i: 0x%x, j: 0x%x, k: 0x%x\n",i,j,k);
    // end 'a'
    // begin 'b'
    for (j=0; j != 10; j++) {i = i + j; }
    printf("i: 0x%x, j: 0x%x, k: 0x%x\n",i,j,k);
    // end 'b'
    // begin 'c'
    do {
        k--;
    } while (bittst(k,3));
    printf("i: 0x%x, j: 0x%x, k: 0x%x\n",i,j,k);
    // end 'c'
    // begin 'd'
    if (i < k) {

```

```

    k--;
} else {
    j++;
}
printf("i: 0x%x, j: 0x%x, k: 0x%x\n",i,j,k);
// end 'd'
// begin 'e'
if (k > j) {
    i++;
} else {
    j++;
}
printf("i: 0x%x, j: 0x%x, k: 0x%x\n",i,j,k);
// end 'e'
}

```

For your assignment, if the last digit of your student ID is 0 or 1, use *loops\_var1.c*; if it is 2 or 3, use *loops\_var2.c*, and so forth. Compile this program with your favorite C compiler and note the value of *i*, *j*, *k* at the conclusion of each code block a, b, c, d, and e. The `bittst(var, bitno)` macro returns the value of bit #*bitno* in variable *var*.

## Lab Activity

1. Convert your assigned *ops\_var{1-5}.c* and *loops\_var{1-5}.c* programs to PIC18 assembly language and verify their operation in MPLAB. The `printf` statements in the C code listings are only included for debugging purposes; use the WATCH window of MPLAB to observe the values of *i*, *j*, and *k* as you single step through your assembly language program. Verify that the assembly language produces the same values for *i*, *j*, *k* at each step.
2. For the *ops\_var* program, calculate the total number of 8-bit operations performed (each 8-bit assignment counts as 1, each logical/add/sub/inc/dec counts as 1, and each shift operation counts as 1). For shift operations, count each required shift as 1 (*a >> 3* counts as 3 shift operations). Make this calculation based on the C code, not the assembly code. Take the execution time value recorded for the *ops\_var* program, and divide by the total number of 8-bit operations to get an average execution time per 8-bit operation. Take the inverse of this value to get the number of 8-bit operations per second that can be expected from the PIC18F242 running at 20 MHz. Compare this to what is obtained by taking the `addwf` instruction and computing the same values based on its execution time. Why are the values different? Discuss this in your report.

## E.5 EXPERIMENT 4: EXTENDED PRECISION AND SIGNED OPERATIONS (CHAPTER 5)

---

This experiment explores PIC18xx2 extended precision and signed operations (Chapter 5). This experiment assumes the students have access to an x86 C compiler on either a Windows or Linux machine to verify C code results.

### Prelab

1. The *intop.zip* archive contains five program variations *intop\_var{1-5}.c* similar to the *op* variations of the previous experiment except the variables are *unsigned int* types. Pick a program variation in the same manner in the previous experiment and record the values of the *i*, *j* variables after each operation.
2. The *compare.zip* archive contains five program variations named *unsigned\_var{1-5}.c* that use *unsigned int* types and five variations named *signed\_var{1-5}.c* that use *signed int* types. These are similar to the *loop\_ops* program of the previous experiment. Pick one program variation of each type in the same manner as the previous experiment and record the values of the *i*, *j* variables after each code block.

### Lab Activity

1. Convert each of your assigned program variations *intop\_var{1-5}.c*, *unsigned\_var{1-5}.c*, and *signed\_var{1-5}.c* to PIC18 assembly language and verify within MPLAB that your code produces the same values for *i*, *j* as the original C code.
2. For the *intop\_var* program, calculate the total number of 16-bit operations performed (each 16-bit assignment counts as 1, each logical/add/sub/inc/dec counts as 1, and each shift operation counts as 1). For shift operations, count each required shift as 1 ( $a \gg 3$  counts as three shift operations). Make this calculation based on the C code, not the assembly code. Take the execution time value recorded for the *intop\_var* program, and divide by the total number of 16-bit operations to get an average execution time per 16-bit operation. Take the inverse of this value to get the number of 16-bit operations per second that can be expected from the PIC18 running at 20 MHz. Compare this to what is obtained by taking the *addwf* instruction, computing the same values based on its execution time, and

multiplying by 2. Compare these two values and discuss reasons for any differences.

- Pick the machine code for either a `bov` or `bnoV` in your `signed_var` program and show that the displacement value in the machine code is correct given the PC and target address values.

## **E.6 EXPERIMENT 5: POINTERS AND SUBROUTINES (CHAPTER 6)**

This experiment covers PIC18 subroutine and pointers, which are discussed in Chapter 6. This experiment assumes the students have access to an x86 C compiler on either a Windows or Linux machine to verify C code results.

### **Prelab**

The `ptrlab.zip` archive contains some C programs for experimenting with pointers and subroutines. Table E.2 describes these programs and gives student assignments based on the last digit of their student ID number.

**TABLE E.2** Program Assignments

<b>Last Digit of Student ID</b>	<b>Assigned C Program, Parameter Block Location</b>
0, 1, 2 or 3	<code>strflip.c</code> ; copies <code>s1</code> to <code>s2</code> , reversing the case of all letters. Use 0x58 for the location of the <code>dopr()</code> parameter block ( <code>ptr1</code> , <code>ptr2</code> ).
4, 5, or 6	<code>strup.c</code> ; exchange <code>s1</code> and <code>s2</code> , upcasing all chars in the new <code>s2</code> . Use 0x5C for the location of the <code>dopr()</code> parameter block ( <code>ptr1</code> , <code>ptr2</code> ).
7, 8, or 9	<code>strchg.c</code> ; exchange <code>s1</code> and <code>s2</code> , reversing the case of all chars in the new <code>s1</code> . Use 0x60 for the location of the <code>dopr()</code> parameter block ( <code>ptr1</code> , <code>ptr2</code> ).

Make a good faith effort to convert your assigned program to PIC18 assembly language before your assigned lab time; use the lab time for seeking TA assistance and debugging of your program.

## Lab Activity

### Pointers and Assembly Language

1. Verify that your PIC18 assembly program duplicates the actions of your assigned C program.

### The PICC-18 C Compiler

1. The *ptrlab.zip* archive contains a C program named *cstrcnt.c* that is compatible with the PICC-18 C compiler. Create a project for *cstrcnt.c* and compile it using the PICC-18 C compiler (see Appendix C for guidance on using the PICC-18 compiler within MPLAB). Record the number of bytes used for program ROM and RAM data. Do *not* execute the program yet.
2. Using the *.map* file that is created, determine the memory locations for the *s1* and *s2* strings. Use the file register window and examine these locations; you will find that they contain zeros because the strings are actually stored in program memory and are copied to data memory by the initialization code executed before `main()` is reached.
3. Begin single stepping through the program, using the Program Memory window (not the source window) to track progress. Watch the locations assigned to *s1* and *s2*; eventually, you will see characters begin appearing in these locations as the strings are copied from program memory to data memory. Determine the location in PROGRAM MEMORY where the *s1* and *s2* strings are stored and give this in your report.
4. Modify the compiler options in MPLAB to add the “-a200” option and re-compile. You should see all program code shift down by 0x200 locations in program memory. This option will be necessary for the hardware labs in order to generate compatible code for the serial bootloader used to program the PIC18F242.
5. Modify the compiler options in MBLAB to remove all compiler optimization and compare with the values obtained in step 1.
6. Modify your assigned C program so that it is compatible with the PICC-18 C compiler and compile it with full code optimization. Fill in the memory locations specified in Figure E.3. You will need to examine the machine code for the `ptr1`, `ptr2`, *s1* (program memory), *s2* (program memory) locations, as these are not contained in the *.map* file.

Variable or Function	Location (in HEX)
s1 string (in program memory)	
s1 string (in data memory)	
s2 string (in program memory)	
s2 string (in data memory)	
main() entry point	
dostr() entry point	
ptr1	
ptr2	

**FIGURE E.3** Variable, function locations for *cstrcnt.c*.

## E.7 EXPERIMENT 6: HARDWARE STARTUP (CHAPTER 8)

This experiment has the student construct the PIC18F242 system of Figure 8.4 as well as the serial port interface of Figure 9.15 using a protoboard. The hardware debugging checklist at the end of these lab exercises is useful for identifying hardware problems.

### Prelab

1. Wire the PIC18F242 schematic shown in Figure 8.4 on your protoboard and add the MAX 202 chip as shown in Figure 9.15 and Figure 9.16. Use the slide switch in the parts kit to implement on/off between the power connector and the 7805. Solder wires to the power jack to create a reliable power connection to your board.
2. Use the 5-pin header in your parts kit for the TX, RX, Gnd pins required by the serial port. If you do not have access to wire wrap supplies, wait until lab time to add the external DB9 connector.

### Lab Activity

#### *ledflash.c* Verification



1. Have the TA program your PIC18 with a hex file produced by compiling the *ledflash.c* program. The companion CD-ROM contains a pre-compiled

*ledflash.c* hex file named *ledflash\_hsp11.hex* that was produced using the PIC18 configuration options found in the *config.h* file, which has the HS-PLL oscillator option enabled. The crystal will not oscillate until the PIC18 is programmed with a hex file that has the correct configuration bit settings for using an external crystal as a clock source.

2. Apply power via your wall transformer and use a multimeter to verify that you have 5 V to your PIC18. Use the oscilloscope and verify that your crystal is producing a sinusoidal waveform, and that the reset button produces a low true pulse on the  $V_{pp}/MCLR\#$  pin when pressed. Verify that the LED on port RB1 flashes after power is applied.

### Serial Port Verification

1. Use the wire wrap tool available from the TA and wire the DB9 connector to the 5-pin header you have used to bring the TX, RX, and Gnd signals off board. Use a serial cable and connect your board to the serial port of a PC. Have the TA program your PIC with the *echo.c* program. This program reads a character from the serial port, increments it, and then echoes it back using a baud rate of 19200. Thus, an “a” typed from the keyboard echoes as “b”, “b” as “c”, and so on. Use HyperTerminal or some other serial port terminal program to verify that the asynchronous serial interface on your board is working. If it does not work, use the serial port debugging tips in Section 9.7 and the debugging checklist at the end of this appendix to isolate the problem.
2. Have the TA program your PIC with the serial bootloader program (*bootload\_hsp11.hex*).
3. Compile the *ledflash.c* program using the PICC-18 compiler and use the “-a200” flag to produce a hex file that is compatible with the bootloader (the resulting hex file has the default name of *ledflash.hex*). The companion CD-ROM contains a pre-compiled *ledflash.c* hex file named *ledflash\_hsp11\_a200.hex* that was produced using the “-a200” compiler flag and the PIC18 configuration options found in the *config.h* file. Use the Jolt or Colt bootloader (Appendix F) to download either the *ledflash.hex* file or the *ledflash\_hsp11\_a200.hex* file into your PIC18 and verify its operation.



**Current Measurement and *reset.c* Test**

1. Compile the *reset.c* program and download it into your PIC18F242 system. Exercise the *reset.c* program in the same manner as shown in Figure 8.12.
2. Fill in the current measurements for Figure E.4 using the directions in the following steps. See the prototyping hints at the end of this appendix for instructions on how to measure current using a multimeter.
3. To measure the 7805 current draw, disconnect its +5 V output from the rest of the circuit and put the ammeter in series with the 9 V input terminal.
4. To measure the current draw of the power LED, determine the difference in current draw when the power LED is inserted in the circuit, and when it is removed. To calculate the expected current draw, use the equation  $I = V/R$ ; where  $V = V_{dd} - 0.7 \text{ V}$  and  $R = 470 \ \Omega$ . The measured and computed values will differ somewhat because the pin driver and LED add extra resistance to the circuit.
5. To determine the current draw of the PIC in normal operation, place the PIC in sleep mode via the menu choice of the *reset.c* program and record the difference in total current draw. Determine the expected current draw for the PIC normal mode operation using the *Typical IDD vs. FOSC over VDD (HS/PLL mode)* graph in the PIC18 datasheet. Note: the X-axis is the *external* crystal frequency.
6. Measure the current draw of the Maxim 202/232 chip by measuring the input current of the Vdd pin or by noting the difference in current draw when the chip is removed from the protoboard.
7. The expected total current draw of your board is the sum of the expected current draw of the individual components.
8. The configuration bits of the PIC18 must be changed to make HS current measurements. These instructions assume that the current program in the PIC18 is the *reset.c* program. Use the Jolt “Read All” command to read all of the PIC18 program memory contents and current configuration bit settings. Change the FOSC configuration to HS, and use the “Program Config” option to program the new configuration bit setting. After changing the configuration bits you will need to set the baud rate to 4800, as the PIC18 is now operating at one-fourth the clock frequency of the HS/PLL mode. Measure the total board current draw and the PIC current draw in the same manner as before. The expected PIC18 current draw can be found in the *typical IDD vs. FOSC over VDD (HS mode)* graph in the PIC18 datasheet. You should discover that the PIC18 in HS/PLL mode consumes about 4x the current of HS mode as it is operating 4x faster.



Measurement	Expected (mA)	Actual (mA)	%Diff (Actual-Exp.)/Exp.*100
7805 Voltage Regulator	N/A, 3 mA to 10 mA.		
MAX202/232			
Power LED			
PIC18 (normal, HS/PLL)			
Total Power (HS/PLL)			
PIC18 (normal, HS)			
Total Power (HS)			

**FIGURE E.4** Current measurements for experiment 6.

## E.8 EXPERIMENT 7: LED/SWITCH IO AND INTRODUCTION TO ASYNCHRONOUS SERIAL IO (CHAPTERS 8, 9)

---

This experiment has the student implement a LED/Switch IO problem using a finite state machine approach. The student is also introduced to the status bits of the PIC18 USART module.

### Prelab

1. Figure E.5 contains an LED/switch IO problem assignment. For your assigned problem, draw a state chart similar to that of Figure 8.20 and make a good faith effort to implement the C code using the same structure as in Figure 8.21 (the *ledsw1.c* file can be used as a starting point).
2. Draw the RS232 waveform for an 8-bit data value, 1 start, 1 stop bit, LSB sent first. The 8-bit data value is based on the last digit of your Student ID as follows: 0) “e”, 1) “F”, 2) “n”, 3) “S”, 4) “3”, 5) “Z”, 6) “z”, 7) “#”, 8) “u”, 9) “p”. Use an ASCII table to determine the 7-bit value of your character (the 8th bit is zero).
3. Demo to the TA a spreadsheet that calculates the bit time in microseconds for any baud rate from 2400 to 115200. Recall that a bit time is equal to one over the baud rate, and that a microsecond is  $10^{-6}$  seconds.

Last Digit of Student ID	LED/Switch Assignment
0 or 5	Push button = RB2; select input = RB4, LED = RB0 1. LED off initially 2. After press and release, blink 3 times, freeze off. 3. After press and release, if select input = 1 goto (1) else goto next step. 4. Blink when pressed and held down. After the third release, goto (5). 5. Blink rapidly (twice as fast as 4). On press and release, go to (1).
1 or 6	Push button = RB1; select input = RB2, LED = RB6 1. LED on initially 2. After press and release, begin blinking. 3. After press and release, halt blinking. If select input = 0 goto (1), else goto next step. 4. LED off. On press only, goto 5. 5. Blink rapidly 6 times while button held down. If button released before all 6 blinks complete, go to (1). If all 6 blinks complete, freeze off, and release remains in this state. A subsequent press repeats (5).
2 or 7	Push button = RB2; select input = RB3, LED = RB7 1. LED on initially 2. On press, begin blinking. On release, go to 3. 3. LED off. After press and release, if select input = 0 goto (1), else goto next step. 4. Blink 4 times. If all four blinks complete then: if select input = 1, go to 1, else goto next. Any press during the four blinks aborts and goto next step. 5. Blink rapidly while button held down. On release, goto 1.
3 or 8	Push button = RB0; select input = RB4, LED = RB7 1. LED off initially 2. On press and release, blink 2 times, goto 3. 3. LED off. Blink while button held down. On release, if input select = 1, then goto (1), else goto next. 4. Blink rapidly. On a press and release goto next. 5. LED off. On press, turn LED on. On release, goto (1).
4 or 9	Push button = RB1; select input = RB5, LED = RB0 1. LED off initially. 2. After two press/releases, begin blinking. Another press and release, stops blinking, goto next step. 3. On press, turn LED on. On release, if select input =1, goto 1, else goto next. 4. Blink rapidly. If 8 blinks are completed, goto 1. Any press/release before this go to 5. 5. LED on. Any press and release, goto 1

**FIGURE E.5** LED/switch IO assignment.

4. Demo to the TA a spreadsheet that calculates the value to be written to the SPBRG register given an oscillator frequency value, a target baud rate, and either high-speed or low-speed baud rate mode. The PIC18F242 datasheet section titled “USART Baud Rate Generator” should prove helpful.

## Lab Activity

1. Program your PIC18F242 with your LED/Switch IO solution and verify its operation. Use a software delay (i.e., `DelayMs(30)`) after each test of a switch input to protect against switch bounce.
2. In the *echo.c* program from the previous lab, the `getch()` subroutine waits for a character to be ready in the TXREG. Modify this subroutine to detect a framing error by checking the FERR bit in the USART status register (RCSTA register). Turn on an LED when an error is detected. The LED should be off after reset. Do the same for the overrun error (OERR) status bit (this requires a second LED). Use any port bits you wish for the LED outputs.
3. Change the baud rate from 19200 baud in HyperTerminal and verify that a framing error condition is produced. Verify that you are checking the correct bit in the RCSTA register. Framing errors occur when the stop bit is detected as clear, which is most likely to happen when the actual baud rate is lower than the expected baud rate.
4. Modify the *echo.c* program to prompt the user for a choice that forces USART overrun by entering a software delay loop of five seconds or longer in which the USART is not read. During this time, type in enough characters to cause overrun. Use the delay functions from the *delay.h* file.
5. Use the scope in single trigger capture mode to capture the character appearing on the RX pin of the PIC. Demonstrate that the character waveform you capture matches the waveform for the character you were assigned in prelab.

## E.9 EXPERIMENT 8: INTERRUPTS (CHAPTER 10)

---

This experiment covers material on interrupts from Chapter 10.

### Prelab

1. Make a good faith effort to implement your assigned LED/Switch IO problem of the previous lab in an interrupt driven mode similar to that of Figure 10.15 using the INT0/INT1/INT2 interrupts. Use a software delay of 30 ms within the ISR to debounce the switch inputs (interrupts INT0, INT1, INT2). This is not the most efficient way to debounce a switch input, but is sufficient for this lab.

- The *root\_rxfifo.c* program inputs a decimal number in ASCII format, computes the floating-point square root, and displays the result. The serial port is used for all I/O (19,200 baud is the default baud rate). This program uses an interrupt approach and a receive software FIFO for serial data input as discussed in Chapter 10. The size of the receive software FIFO is set by the `#define BUFMAX 2` statement. Read this program and ensure that you understand how the serial IO is performed as you will be modifying this program during lab. Table E.3 gives baud rate assignments that are needed when modifying the *root\_rxfifo.c* program.

**TABLE E.3** Baud Rate Assignments

Last Digit of Student ID	Baud Rate Assignment
0 or 1	4800, 19200, 57600, 115200
2 or 3	4800, 19200, 38400, 115200
4 or 5	4800, 9600, 38400, 115200
6 or 7	4800, 9600, 57600, 115200
8 or 9	4800, 19200, 38400, 115200

## Lab Activity

- Program your PIC18F242 with your LED/Switch IO solution and verify its operation.
- Program your PIC with the *root\_rxfifo* program and familiarize yourself with its operation. Type in one number at a time and verify that the floating-point square root is computed. When compiling *root\_rxfifo.c*, select the `printf` library that supports both long and floating-point data types (see Appendix C).
- Open the file *root\_test1.txt* in a text editor such as Notepad, select/copy a range of input values, and use the HyperTerminal “Edit → Paste-to-Host” command to copy this selection into the terminal window. What happens when a large number of entries are pasted at a time? Use the scope and determine the time it takes HyperTerminal to send several successive characters when pasting characters into the buffer. RECORD THIS TIME, you will need it to answer questions in the report. You should observe that the *root\_rxfifo* program operates correctly for a small number of entries pasted

into the window, but not for a large number. The failure is due to simultaneously receiving characters during transmission of the result string. While characters are being transmitted, the receive port is not checked. At some point, the RCREG input FIFO buffer fills to capacity, and characters are lost. Note: When the Paste-to-Host command in HyperTerminal is used, each character is sent at the specified baud rate. However, there is considerable dead time between each character that is sent. This explains why it takes more than the expected number of input characters to cause a buffer overrun problem.

4. Modify the `root_rxfifo.c` program to use the external DIP switch in your parts kit to select between 4 different baud rates after reset (see Table E.3). The DIP switch contains two switches allowing four different combinations. Use any two PORTB inputs you wish. Enable the weak-pullup on the Port B pins so that external pullups are not required for these inputs. You may need to use the low-speed mode to reach some of your assigned baud rates. Examine the `serial_init()` subroutine in the `serial.c` include file to understand how to pass in values for the SPBRG register and how to select between low- and high-speed mode. Use a multimeter to verify the close/open positions of the DIP switch. Test all four baud rates. Use an oscilloscope, and capture a character waveform. Be prepared to show the TA that the bit time of the captured waveform matches the expected bit time.
5. Modify the `pic_isr()` subroutine to turn on an LED if a software FIFO buffer overrun error occurs. This is not the same as the USART overrun error. This overrun error occurs if the `pic_isr()` subroutine is called, and placing another character into the software FIFO causes it to appear empty (overrun occurs if the head pointer is equal to the tail pointer after incrementing and wrapping the head pointer). After turning on the error LED, enter sleep mode to halt the PIC18. Test the program by pasting several numbers into the HyperTerminal window, causing buffer overrun. Discover how many numbers it takes within approximately  $\pm 5$  to cause overrun for your second slowest baud rate.
6. Increase the software FIFO buffer size to 8. Test overrun again with the same baud rate of the previous step and verify that the number of entries required to cause overrun has increased. Because of how HyperTerminal sends characters, you may find that even small increases in the software FIFO size may greatly increase the number of entries required for overrun or that you may not be able to force an overrun at all.
7. Overrun occurs in the `root_rxfifo.c` program when the input data rate exceeds the output data rate (characters needed to print the result). Using the scope measurement you made of the time it took HyperTerminal to send several successive characters, and justify the numbers you recorded for

buffer overrun based on input data rate vs. output data rate. Do this by computing the time it takes to fill up the input buffer given the rate at which characters are being sent by HyperTerminal, and compare this to the time it takes to print the result string. (The PIC is sending characters back at about the maximum rate that can be sustained by the TX channel to the PC.)

## **E.10 EXPERIMENT 9: MORE INTERRUPTS, THE I<sup>2</sup>C BUS, AND A SERIAL EEPROM (CHAPTER 11)**

---

This lab covers material on the I<sup>2</sup>C bus and the 24LC515 Serial EEPROM from Chapter 11.

### **Prelab**

1. Add the 24LC515 Serial EEPROM to your protoboard as shown in Figure 11.25. Do not forget the 10 k $\Omega$  pullup resistors! Connect both A1 and A0 to ground.
2. Create a spreadsheet that calculates the correct SSPADD value for a desired I<sup>2</sup>C bus speed.

### **Lab Activity**

The lab activity first has you verify the correct operation of the 24LC515 EEPROM, after which this is used to capture streaming ASCII data from the serial port.

#### **Verification of the I<sup>2</sup>C Bus Hookup and *i2cmemtst.c* Modifications**

1. Verify that the *i2cmemtst.c* program operates correctly on your protoboard. This program is discussed in Section 11.8 (`main()` is shown in Figure 11.32).
2. Change the wiring of the A1/A0 pins to alter the address of the EEPROM. Modify the *i2cmemtst.c* program to use this new address. Use the following values for A1/A0 based on the last digit of your student ID: a) 0,1,2 use "01"; b) 3,4,5 use "10"; c) 6,7,8,9 use "11". The `#define EEPROM 0xA0` statement in *i2cmemtst.c* must be changed to accommodate the new A1/A0 values. Review the address byte formatting in the 24LC515 datasheet to determine the new value.

3. Modify *i2cmemst.c* to prompt the user to choose one of two different I<sup>2</sup>C bus rates. One rate is 500 kHz (or as close as you can achieve), and the other is based on the last digit of your student ID: a) 0 or 1 (50 kHz), b) 2 or 3 (100 kHz), c) 4 or 5 (175 kHz), d) 6 or 7 (250 kHz), e) 8 or 9 (325 kHz). Even though 500 kHz is above the maximum specification for the memory part, it should still work, as datasheet specifications are very conservative. However, this is only being done for illustrative purposes. Never design a system that relies on a part working beyond its specified performance.
4. Demonstrate the operation of *i2cmemst.c* using the modified EEPROM address with your assigned I<sup>2</sup>C bus speed. Capture the first 5 bytes of a page write or sequential read via the oscilloscope and explain the formatting/purpose of each byte to the TA.

### Interrupt-Driven Streaming Writes

1. Using *i2cmemst.c* as a starting point, create a program that implements streaming data capture as discussed in Section 11.9. Use the flowcharts of Figures 11.36 and 11.37 to help your understanding of how to solve this problem. Your program should prompt the user for either read or write mode. In write mode, capture streaming text from the serial port and save it to EEPROM. Test your program write mode by using the “Transfer → Send Text File” command in HyperTerminal to send a complete file. Do not use large files for test cases, as HyperTerminal takes too long to send text files even with a baud rate of 115200. In read mode, dump the contents of the serial EEPROM to the screen; detect a keypress to start or stop the EEPROM content listing. To allow HyperTerminal to display the text with carriage return/line feeds, add the following check in your code for characters sent to the console:

```

if ((c == 0xd) || (c == 0xa))
    pcr1f();
else putch(c);

```

2. Verify that your program works for your assigned I<sup>2</sup>C bus rates.

## **E.11 EXPERIMENT 10: INTRODUCTION TO DATA CONVERSION (CHAPTER 12)**

---

This experiment covers topics on data conversion from Chapter 12.

### **Prelab**

1. Add the MAX 517 DAC to your protoboard. The connections are the same as shown in Figure 12.19, except the OUT1 pin is connected to Vdd (this is the reference voltage). Connect the AD1/AD0 inputs to ground.
2. Connect the potentiometer in your parts kit to the RA0/AN0 input of the PIC18F242.
3. Familiarize yourself with the *dactst.c* code that will be used during the lab activity. This code is similar to the code of Figure 12.10 in that it reads a voltage on the AN0 input using the PIC18F242 ADC, and then displays the value on the console (only the 10-bit hex value is displayed).
4. Create a spreadsheet that calculates an ADC output voltage given a Vref, an input code value, and the number of ADC input bits. Example: For Vref = 5 V, input code of 128 (decimal), number of bits = 8, the output voltage is 2.5 V. The Excel functions POWER(x,y); DEC2HEX(x), HEX2DEC(x) are useful (the hex conversion is optional).
5. Create a spreadsheet that calculates a DAC output code given a Vref, Vin, and the number of DAC output bits. Example: For Vref = 5 V, Vin = 2.5 V, number of bits = 8, the output code should be 128.

### **Lab Activity**

#### **Verification of DAC/ADC Operation**

1. Verify that the *dactest.c* program functions on your protoboard.
2. Fill in the measurements for Figures E.6 and E.7.



Measurement	Value
a) Vdd (measure this value!)	
b) 1 LSb (Vdd/1024)	
c) PIC RA0 input voltage when displayed code is 0x200 (this should be Vdd/2 but may not be because of offset voltage)	
d) PIC RA0 input voltage when displayed code is 0. This is the offset voltage.	
e) Corrected Input voltage for code=0x200 (c - d)	
f) Absolute Error = Expected (Vdd/2) - Corrected (e)	
g) Relative Error in LSb ((f) / 1 LSb). An acceptable error is less than 1/2 LSb	

**FIGURE E.6** ADC measurements.

Measurement	Value
a) Vdd (measure this value!)	
b) 1 LSb (Vdd/256)	
c) MAX517 output voltage when displayed code is 0x200 (this should be Vdd/2 but may not be because of offset voltage)	
d) MAX517 output voltage when displayed code is 0. This is the offset voltage.	
e) Corrected Input voltage for code=0x200 (c - d)	
f) Absolute Error = Expected (Vdd/2) - Corrected (e)	
g) Relative Error in LSb ((f) / 1 LSb). An acceptable error is less than 1/2 LSb	

**FIGURE E.7** DAC measurements.

**dactst.c Modifications**

1. Change the wiring of the AD1/AD0 pins to alter the address of the DAC. Modify the *dactest* program to use this new address. Use the following values for AD1/AD0 based on the last digit of your student ID: a) 0, 1, 2 use “01”; b) 3, 4, 5 use “10”; c) 6, 7, 8, 9 use “11”. The `#define DAC 0x58` statement in *dactst.c* must be changed to accommodate the new A1/A0 values.

Review the I<sup>2</sup>C address byte formatting in the MAX517 datasheet to determine the new value.

2. Modify the *dactst* program to allow the user to choose one of 4 different transformations to the output waveform: a) clipping, b) multiply by two, c) divide by two, d) inversion. Use the clipping ranges as shown in Table E.4 based upon the last digit of your student ID. Use shifts to do the multiply by two or divide by two. Inversion means that the input voltage value should be subtracted from the maximum value. For example, if *input\_value* is the noninverted 8-bit output value for the DAC, the inverted value would be  $0xFF - \text{input\_value}$ .

**TABLE E.4** Assigned Clipping Ranges

Last Digit of Student ID	Clipping Range
0	Between 1.1 V and 4.3 V
1	Between 1.3 V and 3.9 V
2	Between 0.8 V and 4.1 V
3	Between 0.9 V and 4.2 V
4	Between 1.2 V and 4.4 V
5	Between 1.0 V and 4.0 V
6	Between 1.4 V and 3.8 V
7	Between 1.6 V and 4.6 V
8	between 1.5 V and 4.5 V
9	between 1.7 V and 3.7 V

3. Verify the operation of your modified *dactest* program for all choices by using an oscilloscope or multimeter to monitor the ADC input voltage and DAC output voltage.

## **E.12 EXPERIMENT 11: TIMER INTRODUCTION AND WAVEFORM GENERATION (CHAPTERS 10, 13)**

---

This experiment covers the use of Timer2 for periodic interrupts (Chapter 10) and PWM (Chapter 13).

## Prelab

1. Connect the audio jack connector in your parts kit to the output of the DAC. Also connect an LED to pin RC2/CCP1 of the PICF242, and the potentiometer to the RA0/AN0 input. The audio jack allows an external speaker to be driven by the DAC output. This capability is crucial for the last experiment, so it is tested in this experiment.
2. Demo to the TA a spreadsheet that calculates the values required for Figure E.8. Your assigned target frequencies are in Table E.5. The frequencies in Table E.5 use the PWM mode for generating the square wave; this does *not* use the postscaler for frequency calculation. The spreadsheet should calculate the PR2 values given a target frequency, and prescaling factors of 1, 4, and 16. The spreadsheet should also truncate the PR2 value to an integer value, and then compute the %diff between the actual frequency obtained and desired frequency. Choose the prescale and PR2 value that gives the lowest %diff value.

**TABLE E.5** Assigned Frequencies

Last Digit of Student ID	Use These Frequencies
0 or 1	2500 Hz, 10 kHz, 121 kHz
2 or 3	3600 Hz, 15 kHz, 133 kHz
4 or 5	4200 Hz, 28 kHz, 144 kHz
6 or 7	5500 Hz, 37 kHz, 151 kHz
8 or 9	6100 Hz, 49 kHz, 165 kHz

3. Become familiar with the *sqwave.c*, *ledpwm.c*, and *sinegen.c* programs, as they are used in this experiment.

## Lab Activity

### ***sqwave.c* Program**

The *sqwave.c* program uses the PWM module to generate a square wave on the RC2/CCP1 output. The program prompts the user to enter Timer2 prescale and PR2 values.

1. Use the *sqwave.c* program to check the values you computed for Figure E.8. Use a scope to monitor the output waveform on pin RC2/CCP1.

Target Frequency	Prescale	PR2	Actual Frequency	%Diff
a)				
b)				
c)				

**FIGURE E.8** Prescale, PR2 values.

### ***ledpwm.c* Program**

The *ledpwm.c* program outputs a square wave of a fixed frequency, but allows dynamic update of the duty cycle by reading the AN0 analog input. This 10-bit value is used to set the value of the duty cycle. Adjusting the potentiometer adjusts the duty cycle of the square wave on the RC1/CCP1 pin. Connect an LED to the RC1/CCP1 output so that the LED turns on when a high voltage is on the RC1/CCP1 output.

1. Verify the operation of *ledpwm.c* on your PIC. What happens to the LED brightness as you adjust the duty cycle via the potentiometer? Monitor the waveform generated on pin RC2/CCP1 with the oscilloscope.
2. Use a multimeter to measure the current through the LED for various duty cycles and complete Figure E.9, which requires current measurements for two different duty cycles. The two duty cycles, based on your student ID, are: a) 0/1 5%/25%; b) 2/3 10%/30%; c) 4/5/6 15%/35%; d) 7/8/9 20%/40%. Also, measure current at the 85% duty cycle, and at a duty cycle midway between the two above. After recording your current measurements in lines (1) and (2) of Figure E.9, compute the expected current for line 3 (halfway between 1st and 2nd duty cycles) and line 4 (85%) duty cycles. Ideally there is a linear relationship between the current and duty cycle. Use the first two measurements to compute a straight line slope that is used to predict the currents for the last two duty cycles.

### ***sinegen.c* Program**

The *sinegen.c* program generates a sine wave using a table lookup approach via the MAX517 DAC. The program prompts the user to choose between a 16-entry and a 64-entry table. Timer2 is used to trigger an interrupt that reads the next entry from the table. The interrupt interval is set by a prescale value of 4, a postscale value of 3,

Duty Cycle (%)	Duty Cycle (μs)	Measured Current (mA)	Expected Current (mA)	% Diff
(1) Pt1			N/A	N/A
(2) Pt2			N/A	N/A
(3) (Pt2+Pt1)/2				
(4) 85%				

**FIGURE E.9** PWM current measurements.

and the PR2 value that is set by the ADC AN0 input. The sine wave period is `table_size * interrupt_interval`; the 16-entry and 64-entry sine wave tables are in *sinegen.h*. The PR2 value is limited by *sinegen.c* to be between 25 and 100.

1. Verify *sinegen.c* operation on your PIC. Hook the audio jack output to some external powered speakers or headphones. Vary the period of the sine wave via the potentiometer and make primitive music.
2. Use the scope to monitor the output of the DAC. Note what happens for the 16-table case when the frequency is increased to near its maximum value. The interrupt interval becomes too small for the DAC to be updated with the new table value because of the I<sup>2</sup>C bus speed. This causes waveform values to be skipped, degrading waveform quality.
3. Fill in the computations and measurements required in Figure E.10. See the comments after the table for hints on obtaining these values.

Measurement	Expected	Measured
a. Min. Timer interrupt interval (PR2=25, PRE=4, POST=3), μs		N/A
b. Max. Timer2 interrupt interval (PR2=100, PRE=4, POST=3), μs		N/A
c. Min sinewave frequency, 16-entry table (Hz)		
d. Max sinewave frequency, 16-entry table (Hz)		
e. Min sinewave frequency, 64-entry table (Hz)		
f. Max sinewave frequency, 64-entry table (Hz)		
g. Measured I <sup>2</sup> C bus speed (kHz)	N/A	
h. DAC update time (μs)		
i. Max sinewave freq. without losing DAC update values (Hz)		

**FIGURE E.10** *sinegen.c* measurements.

Values (a), (b) can be computed from the datasheet formula for Timer2 interrupt interval.

For (c) through (f), the period of the sinewave is the interrupt interval times the number of table entries for the sinewave; the frequency is the inverse of the period.

For (h), compute the DAC update time by multiplying the number of I<sup>2</sup>C bit times required for the DAC update by the measured I<sup>2</sup>C bus speed. The measured value can be obtained by using the scope on the I<sup>2</sup>C bus.

For (i), the measured DAC update time determines the minimum time interval for each new sinewave value. The number of entries in the sinewave times this value gives the minimum period of the sinewave that can be reliability generated without skipping values. Sinewave values are skipped when the Timer2 interrupt interval becomes less than the DAC update time.

### Arbitrary Waveform Generation

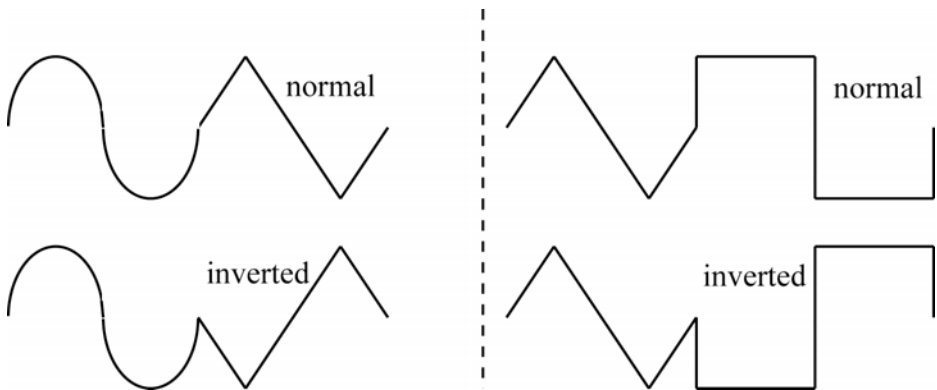
1. Modify *sinegen.c* to generate an arbitrary waveform as described in Table E.6. Following this table are detailed hints on how to implement the arbitrary waveform generator.

Table E.6 provides the details of the arbitrary waveform that you are required to generate. The waveform is one or more sine wave cycles, followed by one or more triangle wave cycles, followed by one or more square wave cycles that are then repeated. A 1x period is 64 time units; waveforms can have periods of 1x, 2x, or 0.5x. The waveform generated by *sinegen.c* has a 1x period by this definition. The interrupt interval for this program should be set in the same way as in *sinegen.c*. The triangle wave and square wave can also be inverted; Figure E.11 shows examples of inverted triangle waves and square waves. If one cycle of a waveform is inverted, all cycles are inverted.

Your program must track the current cycle number and the corresponding waveform to be generated. The `tabmax` variable in *sinegen.c* determines the period of a waveform; this value can be changed from cycle to cycle depending on the waveform being generated (i.e., for 2x period `tabmax = 128`, for a 0.5x period `tabmax = 32`). Write separate subroutines for square wave and triangle wave generation. The easiest way to implement this capability is by using lookup tables for all three waveforms. An alternate method is to compute the value of each point given the current table index. This computation is easy for the square wave and more difficult for the triangle wave.

**TABLE E.6** Waveform Assignments

Last Digit of Student ID	Waveform
0	1 cyc sine, 2 cyc triangle (0.5x per), 1 cyc square (0.5x per)
1	2 cyc sine, 1 cyc triangle (2x per), 2 cyc square (0.5x per)
2	1 cyc sine, 1 cyc triangle (0.5x per), 1 cyc square (2x per)
3	2 cyc sine, 2 cyc triangle (2x per), 1 cyc square (2x per)
4	1 cyc sine, 2 cyc triangle (2x per, inverted), 1 cyc square (2x per)
5	1 cyc sine, 2 cyc triangle (2x per, inverted), 1 cyc square (0.5x per)
6	2 cyc sine, 1 cyc triangle (0.5x per), 2 cyc square (0.5 x per, inverted)
7	1 cyc sine, 1 cyc triangle (0.5x per, inverted), 1 cyc square (0.5 x per, inverted)
8	2 cyc sine, 2 cyc triangle (0.5x per, inverted), 1 cyc square (2x per)
9	1 cyc sine, 2 cyc triangle (0.5x per, inverted), 2 cyc square (2x per, inverted)



**FIGURE E.11** Arbitrary waveform examples.

## **E.13 EXPERIMENT 12: TIME MEASUREMENT AND IR WAVEFORM DECODING (CHAPTER 13)**

---

This program covers the use of the capture/compare module for time measurement (Chapter 13). This lab assumes that the student has access to a universal remote control.

### **PRELAB**

1. Connect a momentary switch to the RC2/CCP1 input.
2. The program *swdetov.c* uses Timer1 and the capture module to measure the pulse width of a momentary switch. Read this program and understand its operation, as you will need to modify it to fulfill the lab requirements.

### **Lab Activity**

#### **Pulse Width Measurement Using *swdetov.c***

1. The *swdetov.c* program uses the PIC18F242 capture module to measure the low pulse width of a momentary switch as discussed in Section 13.4. Verify the operation of *swdetov.c* on your PIC18F242. The “timer tics” that is printed is the elapsed timer tics between the edges; the pulse width is the computed time in microseconds.
2. Fill in Figure E.12 for three button pushes. Use a scope in single trigger mode and capture the low pulse width.

<b>Trial</b>	<b>Tics</b>	<b>Computed Width</b>	<b>Measured Width</b>	<b>%Diff</b>

**FIGURE E.12** Momentary switch pulse width results (original *swdetov.c*).



**swdetov.c Modification**

1. Modify *swdetov.c* to use CCP2 as the input pin, capture register CCPR2, and Timer3 as the timebase.
2. Fill in Figure E.13 for three button pushes. Use a scope in single trigger mode and capture the low pulse width.

Trial	Tics	Computed Width	Measured Width	%Diff

**FIGURE E.13** Momentary switch pulse width results (modified *swdetov.c*).

**IR Waveform Decoding**

1. Place the IR receiver module (Radio Shack PN #276-640) from your parts kit on the protoboard, and connect the OUT pin to the RC2/CCP1 pin of the PIC.
2. On the universal remote, locate a manufacturer setting that produces space-width encoded output as discussed in Chapter 13 (use the oscilloscope to verify that the output waveform is space-width encoded). Write a program similar to the biphase decoding program of Figures 13.22, 13.23, and 13.24 to perform space-width decoding. Space-width decoding is easier than biphase decoding, as the only measurement required is the time between every falling edge of the incoming waveform because “0” and “1” bits have different periods. Only print the first 2 bytes of a received waveform.

**E.14 EXPERIMENT 13: AUDIO RECORD/PLAYBACK (CHAPTER 14)**

---

This experiment implements the audio record/playback project of Chapter 14.

**PRELAB**

1. Implement the audio playback/record schematic of Figure 14.2 on your PIC18F242 system.
2. Read Sections 14.2 and 14.3 and ensure that you understand the *audio.c* code (Figure 14.3 through 14.6), as you will be modifying this code during lab.

**Lab Activity**

1. Verify that you can record and playback audio at a 6 kHz sample rate using the *audio.c* file.
2. Flatten the code of the playback loop within *audio.c* until you can achieve an 8 kHz playback rate. This also requires running the I<sup>2</sup>C bus faster than the maximum datasheet specification of 400 kHz.
3. If you are feeling ambitious, implement the suggested modification contained in problem #1 at the end of Chapter 14!

**E.15 HARDWARE DEBUGGING CHECKLIST**

---

Debugging hardware problems requires a methodical approach and the use of available instrumentation such as a multimeter and oscilloscope. The following are debugging checklists that are useful for identifying hardware problems.

**“My board used to work and now it doesn’t.”**

- |   |                              |                             |
|---|------------------------------|-----------------------------|
| 1. Used multimeter to measure V <sub>dd</sub> on PIC.         | yes <input type="checkbox"/> | no <input type="checkbox"/> |
| 2. Both VSS pins on PIC are connected to ground?              | yes <input type="checkbox"/> | no <input type="checkbox"/> |
| 3. Used scope to see if oscillator working.                   | yes <input type="checkbox"/> | no <input type="checkbox"/> |
| 4. Used scope to ensure reset line works.                     | yes <input type="checkbox"/> | no <input type="checkbox"/> |
| 5. Checked PIC with older test program ( <i>ledflash.c</i> ). | yes <input type="checkbox"/> | no <input type="checkbox"/> |
| 6. Checked PIC in TA reference board to see if board problem. | yes <input type="checkbox"/> | no <input type="checkbox"/> |

**“My fuse keeps blowing, help!”**

To track shorts, perform the following steps in order:

1. Connect the multimeter in series with power to monitor current.
2. Disconnect one half of the protoboard from the other half and determine which half the problem is in. Only connect power for a brief period of time to see if short still exists.
3. Remove all ICs from problem half of board and see if short is fixed.
4. If short still exists, remove any capacitors.
5. If short still exists, remove any switches.
6. If short still exists, remove any LEDs.
7. If short still exists, it must be a direct wiring connection between Vdd/Gnd.

**“My RS232 interface does not work”**

- |   |                              |                             |
|---|------------------------------|-----------------------------|
| 1. MAX202 is producing $\pm 10$ V.  | yes <input type="checkbox"/> | no <input type="checkbox"/> |
| 2. MAX202 Vdd/Gnd connected.  | yes <input type="checkbox"/> | no <input type="checkbox"/> |
| 3. Used scope to see if PC/HyperTerminal is transmitting.   | yes <input type="checkbox"/> | no <input type="checkbox"/> |
| 4. Does HyperTerminal have the right COM port selected?   | yes <input type="checkbox"/> | no <input type="checkbox"/> |
| 5. Is Flow control set to “NONE” in HyperTerminal?  | yes <input type="checkbox"/> | no <input type="checkbox"/> |
| 6. Is the cable connected to the correct COM port on the PC?  | yes <input type="checkbox"/> | no <input type="checkbox"/> |
| 7. Used scope to check MAX202-to-PIC (RX) link (type a character in HyperTerminal and verify character arrives RX pin of MAX202 and RX pin of PIC). | yes <input type="checkbox"/> | no <input type="checkbox"/> |
| 8. Used scope to check PIC(TX)-to-MAX202 link (program PIC with <i>echo</i> program and check if the PIC TX pin is echoing character).              | yes <input type="checkbox"/> | no <input type="checkbox"/> |
| 9. Used scope to check MAX202-to-PC link.   | yes <input type="checkbox"/> | no <input type="checkbox"/> |
| 10. If receiving garbage, does measured bit time match baud rate?   | yes <input type="checkbox"/> | no <input type="checkbox"/> |

One quick tip: If power is off to your board, and your power-on LED is still dimly lit and you have an RS232 cable connected, this may indicate that you have reversed the TX/RX pins on your DB9 to MAX202 chip connection.

**“Jolt does not work”**

1. On Jolt startup, you get a “*main* class not found” error. Verify that your CLASSPATH environment variable is set correctly and that the *comm.jar* file (see Appendix F, Section F.2, “Jolt Installation”) is copied to the location indicated by the CLASSPATH variable. Review all of the Jolt installation steps and verify that you have performed each correctly.
2. The Jolt *Program* option is not working (the progress bar does not advance). This usually indicates a problem with the serial port connection. When trying to program, Jolt periodically sends a handshake character via the serial port to the PIC. To debug, monitor the RX line on the PIC with a scope and verify that that the handshake character is arriving. If no character is arriving, debug your RS232 interface following the preceding steps. If a character is arriving, look at the TX output of the PIC—the Jolt firmware on the PIC should be trying to respond. If the TX output has no activity, reprogram your PIC with the bootloader firmware.

**“My I<sup>2</sup>C interface does not work”**

1. Verified both SCL and SDA have Vdd via pullups when idle. yes  no
2. Do you have SCL/SDA swapped? (SCL is the CLOCK!) yes  no
3. Verified transmission by PIC on SCL/SDA. yes  no
4. Verified I<sup>2</sup>C device address in your program. yes  no
5. Verified I<sup>2</sup>C device address on A1/A0 pins of EEPROM/DAC. yes  no
6. Is the I<sup>2</sup>C Device (EEPROM/DAC) sending an ACK? yes  no

**“My A/D does not work”**

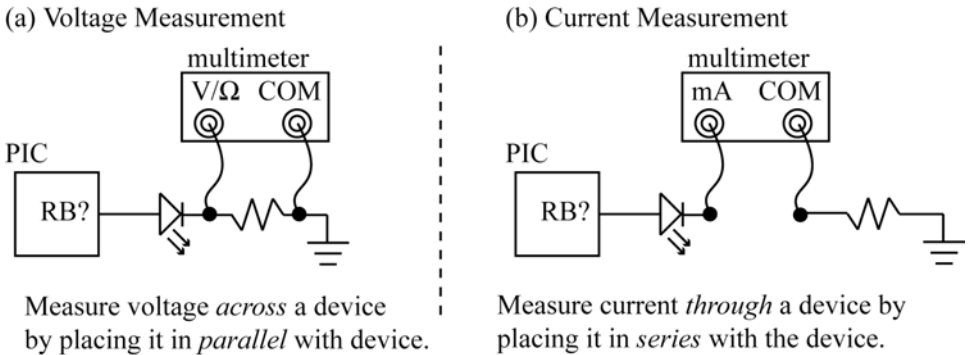
1. Used multimeter to check PIC A/D input voltage variation. yes  no
2. Is the analog input connected to the correct A/D input? yes  no
3. Is PORTA configured for analog input? yes  no
4. Used printf() statement to print individual A/D result bytes. yes  no

## E.16 INSTRUMENTATION AND PROTOTYPING HINTS

This section contains a few instrumentation and prototyping hints to aid you in performing the suggested experiments.

### Voltage, Resistance, Current Measurement

A digital multimeter (DMM) is a common instrument for measuring voltage, resistance, and current. Figure E.14a shows how to use a DMM to measure voltage across a resistor. A resistance measurement is made in the same manner, except the DMM front panel buttons should be set to resistance instead of DC voltage.



**FIGURE E.14** Voltage, current measurement.

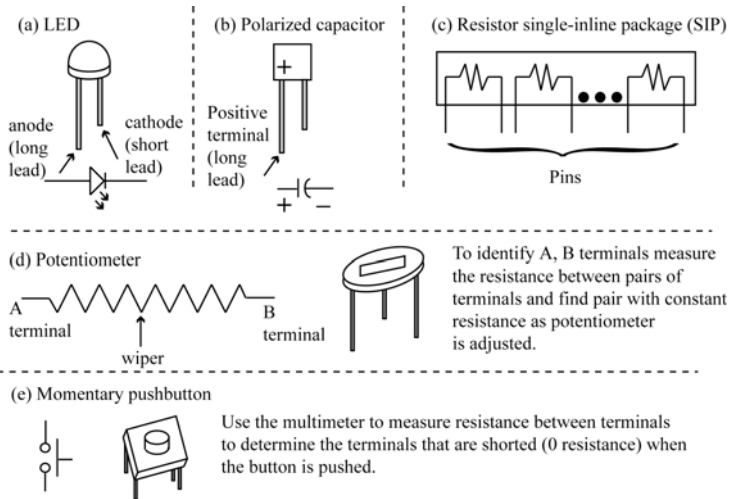
Observe that to measure voltage, no physical changes to the circuit wiring has to be done. This is not true with current measurement, as shown in Figure E.14b. The DMM must be placed in series with a device whose current is being measured in order for the current to flow through the DMM and then into the device. This requires physically breaking the circuit connection, and routing the wiring to the *mA* and *COM* terminals of the DMM.

### Passive Components: LEDs, Capacitors, Resistors, Switches

Figure E.15 shows some of the passive components in the parts kit of Figure E.2. A light emitting diode (LED) conducts current when the anode has a voltage approximately 0.7 V higher than the cathode; LED brightness increases as current increases. The short lead is the cathode, the long lead is the anode.

The 15 pF capacitors used with the crystal in Figure 8.4 to form the PIC18F242 clock source are not polarized, which means that it does not matter which direction the capacitors are connected in the circuit. The 0.1  $\mu$ F capacitors used between

V<sub>dd</sub> and V<sub>SS</sub> on the PIC18 and with the MAX202 are polarized and have a clearly marked positive (+) terminal; the negative (–) terminal should be connected to ground. The resistors of the parts kit in Figure E.2 are in a single inline package (SIP) as shown in Figure E.15c, each resistor is connected between two pins on the package. A potentiometer will have at least three terminals as shown in Figure E.15d (the potentiometer of the parts kit in Figure E.2 has four terminals as the wiper terminal is replicated on two pins). To determine the terminals marked as A and B in Figure E.15d, use a DMM to measure the resistance between pairs of terminals until you find the terminal pair whose resistance does not change when the potentiometer is adjusted. Pushbutton switches as shown in Figure E.15e have no pin markings; you must use the DMM to measure the resistance between terminal pairs to determine which terminal pair is shorted (zero resistance) when the pushbutton is pressed.

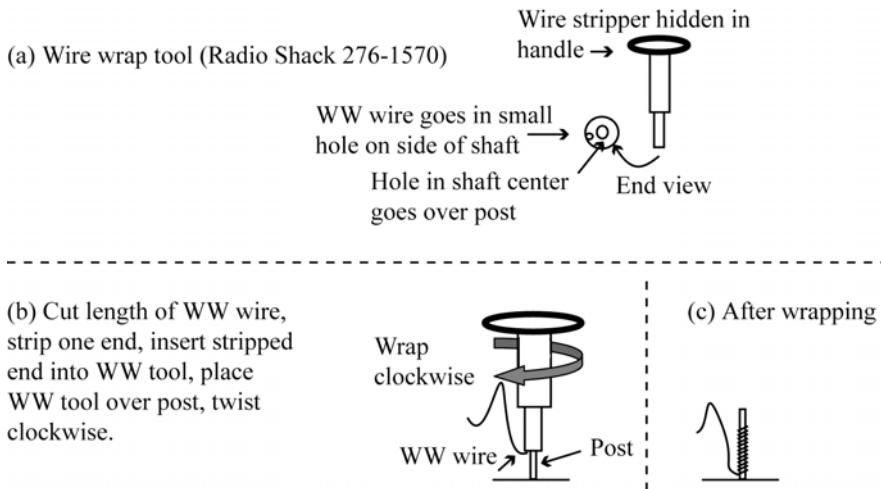


**FIGURE E.15** Passive components.

## Wire Wrapping

Wire wrap is useful to create secure connections to components that do not plug directly into the protoboard. In Figure E.1, wire wrap is used to connect to the DB9, potentiometer, and modular connector. Figure E.16 shows the steps for creating a wire wrap (WW) connection using a WW tool and WW wire from Radio Shack. WW wire is 30 gauge wire, and the ends of a wire wrap connection must be stripped before wrapping. The Radio Shack WW tool has a wire stripper contained in the

handle. The end of the WW tool has a large center hole for fitting the tool over a post, and a small hole on the side that is used to hold the wire. To create a WW connection, cut a piece of wire of appropriate length and strip approximately  $\frac{1}{2}$  to  $\frac{3}{4}$  inch from either end. Place a stripped end into the small hole of the WW tool and push the wire into the tool until the insulation prevents the wire from going any further. Then, place the WW tool over a post, and twist clockwise, holding the wire to keep it taut while wrapping. This should wrap the stripped portion of the wire around the post. Repeat this with the other stripped end of the wire at the destination post. To unwrap, place the WW tool over the post, push down firmly, and turn counter-clockwise.



**FIGURE E.16** Wire wrapping.

# F

## The Jolt/Colt Serial Bootloaders



This appendix discusses use of the Jolt and Colt serial bootloaders for downloading PIC18F242 programs using the serial port interface. Martin Dubuc wrote both programs, and the Jolt/Colt home pages are found at <http://mdubuc.freeshell.org/{Jolt/Colt}>. The bootloader programs are self-extracting executables named *bootldr/ColtSetup.exe* and *bootldr/JoltSetup.exe* on this book's companion CD-ROM. Jolt has more features than Colt in terms of viewing the code to be programmed and altering configuration bit settings, but requires installation of the Java Runtime Environment. Both Jolt and Colt are compatible with Windows XP. At Mississippi State University, Colt is generally preferred by students because of its simpler installation.

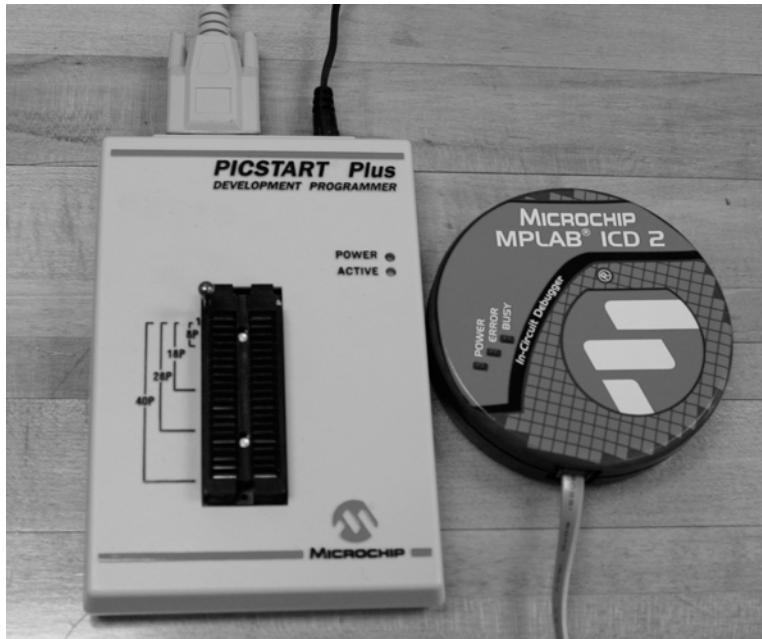
### F.1 PROGRAMMING THE JOLT/COLT FIRMWARE

---

Each bootloader consists of two parts: firmware that resides on the PIC18 and a client that runs on the PC. The PC client reads a hex file and sends the program memory contents over the PC serial port to the PIC18 bootloader firmware that programs the PIC18 program memory with the incoming bytes. EEPROM data memory and configuration bits can be programmed by the bootloader as well. The PIC18 bootloader firmware is the same for both Colt and Jolt. The bootloader firmware is in a file named *bootload.hex* that is found within the respective default installation directories (*C:/Program Files/Colt PIC18F Bootloader*, *C:/Program Files/Jolt PIC18F Bootloader*). A version of the bootloader hex file with the configuration bits set to options used for the book PIC18F242 reference system is found in *code/labs/bootload\_hsp11.hex*.

Programming the bootloader firmware requires use of an external PIC programmer. Figure F.1 shows a picture of two external programmers available from Microchip: the PICSTART Plus and the ICD2.

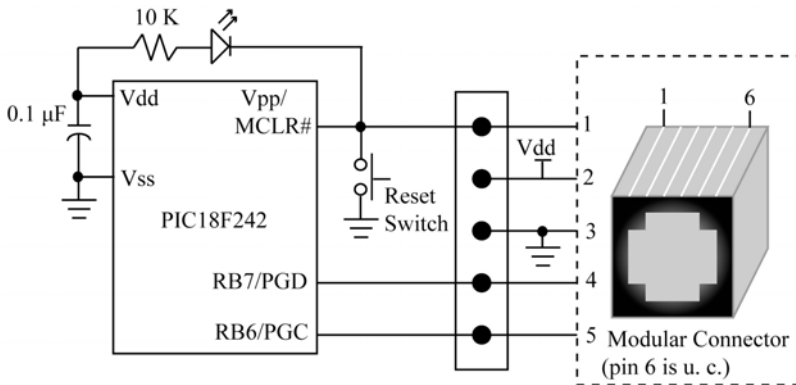




**FIGURE F.1** PICSTART Plus and ICD2 programmers.

The PICSTART Plus has a 40-pin ZIF (zero insertion force) socket for holding PIC devices and communicates with the PC via the serial port. It can program a wide variety of PIC microcontrollers and is the programmer the author uses in the laboratory environment at Mississippi State University. The disadvantage of the PICSTART Plus is that the PIC18 has to be removed from the protoboard for programming; the Jolt/Colt bootloaders allow in-circuit programming via the serial port.

An alternative to the Jolt/Colt bootloaders for in-circuit programming is the ICD2 programmer, which supports both in-circuit programming and limited in-circuit debugging of PIC microcontrollers. The ICD2 communicates with the PC either through a serial port or a USB port. Figure F.2 shows the necessary modification to the PIC18 startup schematic of Figure 8.4 to support the ICD2. During in-circuit serial programming (ICSP), voltage pulses of approximately 12 V are applied to the V<sub>pp</sub>/MCLR pin. The LED isolates the rest of the V<sub>dd</sub> bus from the high-voltage pulses applied on V<sub>pp</sub> during programming. The RB6/PGC pin is the clock used for serial programming data sent over the RB7/PGD pin.



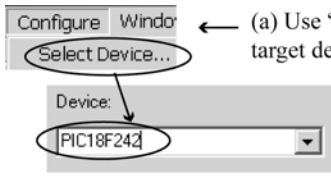
**FIGURE F.2** ICD2 programmer connection.

If the ICD2 programmer interface is used, a serial bootloader such as Jolt is unnecessary. One disadvantage to using the ICD2 is that if pins RB7, RB6 are driven by other active circuitry on your board, these must be isolated during programming, perhaps by DIP switches.

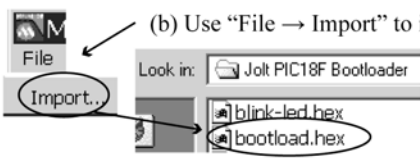
MPLAB is used as a front-end for both the PICSTART Plus and ICD2 programmers. Figure F.3 shows the steps for programming a PIC18 using the PICSTART Plus programmer.

The “File → Import” option (Figure F.3 b) is used for loading a pre-existing hex file into MPLAB for programming or simulation. The “Configure → Configuration Bits” option (Figure F.3c) is used for examining and/or modifying configuration bit settings. When programming the Jolt/Colt firmware into your PIC18, it may be necessary to use this option to change the configuration bit setting for the oscillator to match your system. After changing the configuration bits, the “File → Export” option is useful for saving a new copy of the hex file with the modified configuration bits. Steps (d), (e), (f), and (g) in Figure F.3 show how to use the PICSTART Plus to program a PIC18. The final programming step also includes verification of the downloaded code by reading the PIC18 program memory and comparing it with the target code.

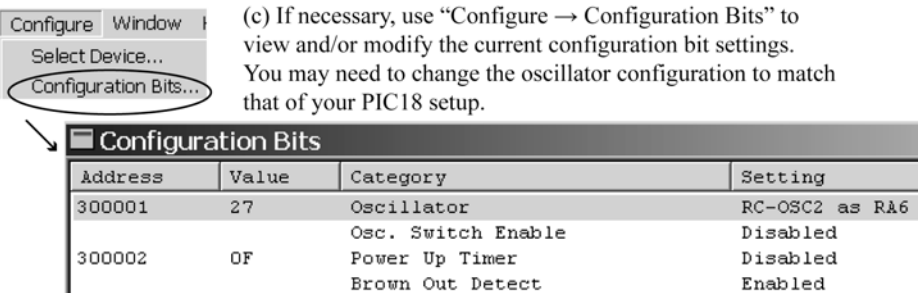
(a) Use "Configure → Select Device" to select the PIC18F242 as the target device.



(b) Use "File → Import" to import a hex file for programming.

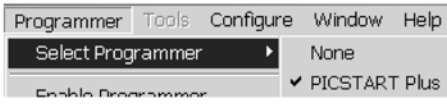


(c) If necessary, use "Configure → Configuration Bits" to view and/or modify the current configuration bit settings. You may need to change the oscillator configuration to match that of your PIC18 setup.

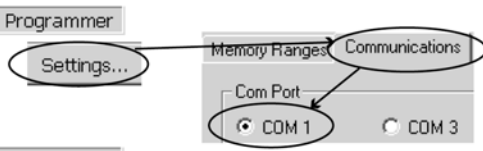


Address	Value	Category	Setting
300001	27	Oscillator	RC-OSC2 as RA6
		Osc. Switch Enable	Disabled
300002	0F	Power Up Timer	Disabled
		Brown Out Detect	Enabled


(d) Use "Programmer → Select Programmer" → PICSTART PLUS" to select the programmer.




(e) If this is the first time using the programmer, then use "Programmer → Settings" to display the settings window. Use the *Communications* tab to set the COM port used for communication.




(f) Use "Programmer → Enable Programmer" to enable communication with the programmer.



(g) Place the PIC18F242 device in the PICSTART programmer ZIF socket, and use "Programmer → Erase Flash Device" to erase the device.



(h) Use "Programmer → Program" to program the device with the hex file currently in memory. A verification is automatically performed after the device is programmed.



**FIGURE F.3** Using the PICSTART PLUS. Screenshots ©2005 Microchip Technology, Inc. Reprinted with permission. All rights reserved.

## F.2 JOLT INSTALLATION

---

The most up-to-date installation instructions and Jolt version can be found at <http://mdubuc.freeshell.org/Jolt>. The following off-line installation instructions are compatible with Jolt V1.0. Execute the self-installing executable *bootldr/JoltSetup.exe* to install Jolt; the default installation location is *C:\Program Files\Jolt PIC18F Bootloader*. Before running Jolt, the following steps must be completed for correct operation.

### Java Runtime Environment Installation

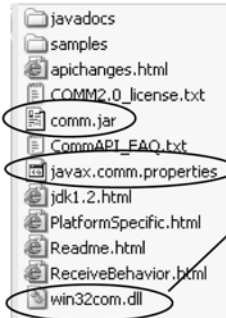


Jolt is written in Java, and thus either the Java 2 Platform, Standard Edition (J2SE) Java Runtime Environment (JRE) or J2SE Software Development Kit (J2SE SDK) must be installed. A starting point for these downloads is found at <http://java.sun.com/j2se>. A version-specific URL for downloading version 5.0 of the J2SE is <http://java.sun.com/j2se/1.5.0/index.jsp>. The Jolt version on the companion CD-ROM has been tested with version 5.0 of the JRE. If you are installing Java only for use with Jolt, it is recommended that you install the runtime environment (J2SE JRE), as it is much smaller than the software development kit. The remaining instructions assume that the J2SE JRE has been installed.

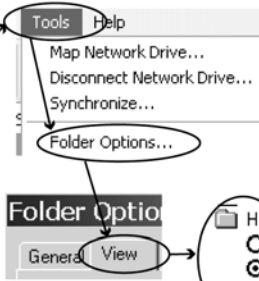
### Java Communications API

1. The Java Communications API must be installed after the J2SE JRE installation for Jolt to communicate over the serial port. The Java Communications API is found at <http://java.sun.com/products/javacomm/> and is distributed in the form of a ZIP archive. Download this ZIP archive and unpack it into some temporary directory on your PC. Three files must be copied from the JavaComm API folder into the JRE folder as shown in Figure F.4.
2. Copy the *win32com.dll* file from the JavaComm API folder to the JRE *bin* folder. If this file is not visible, use “Tools → Folder Options” to change folder options to display file extensions for known file types and to show hidden files and folders as shown in Figure F.4 (b, c).
3. Copy the *javax.comm.properties* file from the JavaComm API folder to the JRE *lib* folder.
4. Copy the *comm.jar* file from the JavaComm API folder to the JRE *lib/ext* folder.
5. Two changes must also be made to system variables as shown in Figure F.5. The CLASSPATH variable must be created and the path to the *comm.jar*

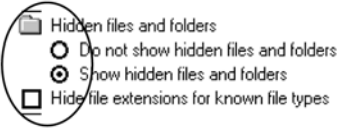
(a) Three files of interest from the JavaComm API folder.



(b) If *win32com.dll* is not visible, choose “Tools → Folder Options” from the folder window menu bar.



(c) Show file extensions and hidden files.



(d) Copy files from the JavaComm API folder to the JRE installation folder.



← (e) *win32comm.dll* goes in the *bin* folder.

← (f) *javax.comm.properties* goes in the *lib* folder.

← (g) *comm.jar* goes in the *lib/ext* folder.

**FIGURE F.4** Copying files from JavaComm folder to Java JRE folder.

file placed on it. The PATH variable must be modified to include the *bin* folder of the JRE installation.

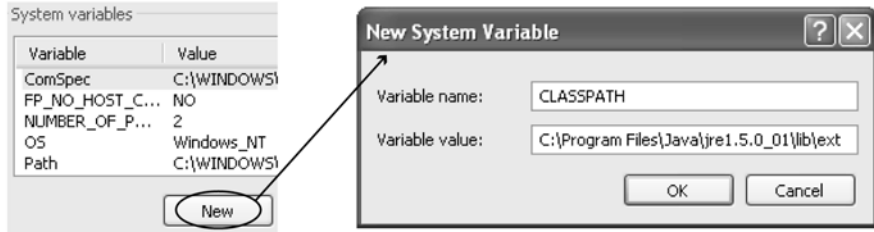
(a) Choose “Start → Control Panel → Performance and Maintenance → System”. In classic view, this is shortened to “Start → Control Panel → System”.



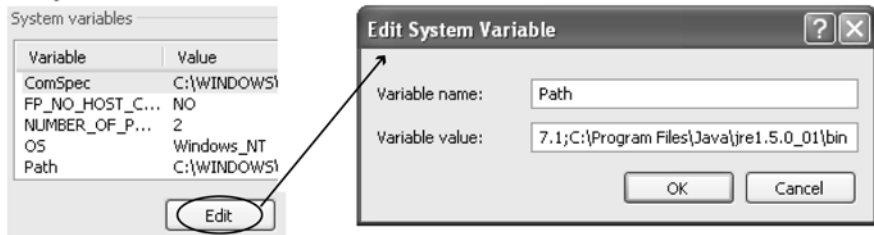
(b) In the System Properties window, choose “Advanced → Environment Variables” to display the Environment Variables window.



(c) In the Environment Variables window, choose “New” and create a new system variable named CLASSPATH and add the path to the *comm.jar* file. In this example, the complete path name is *C:\Program Files\Java\jre1.5.0\_01\lib\ext*.



(d) In the Environment Variables window, choose “Edit” and edit the PATH system variable to add the JRE *bin* folder. In this example, the complete path name is *C:\Program Files\Java\jre1.5.0\_01\bin*. A semicolon (“;”) is used to separate directories in the PATH environment variable.



**FIGURE F.5** Modifying the CLASSPATH and PATH environment variables.

### F.3 RUNNING JOLT

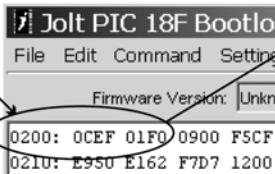
If the installation steps of the previous section were completed correctly, running the Jolt bootloader opens the Jolt window as shown in Figure F.6. The Jolt code window displays the program bytes to be downloaded.



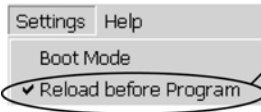
(a) Jolt window, no code loaded.



(b) Use “File → Open Hex File” to load a hex file. The hex file must be compiled with the “-a200” linker option for the PICC-18 C compiler to relocate the program to address 0x0200.



(c) The first instruction should be a `goto` as this is the reset vector. For the PIC18F242 compiled using PICC-18, if the first two bytes displayed are not 0x0C, 0xEF (instruction word 0xEF0C = `goto`) then you have forgotten this option!



(d) A useful setting is “Reload before Program”, which reloads the hex file before programming.



(e) Use “Command → Program” to download a program into a PIC via the serial port. The progress bar tracks the download status.



(f) For best results, turn the PIC off or hold the reset button down before using “Command → Program”. Once the progress window appears, turn on the PIC or release reset.



(g) Use this to set COM port and baud rate. The Jolt bootloader auto detects baud rate, so set this as high as can be reliably supported on your system. A value of 38400 was used on the book PIC18 reference system.

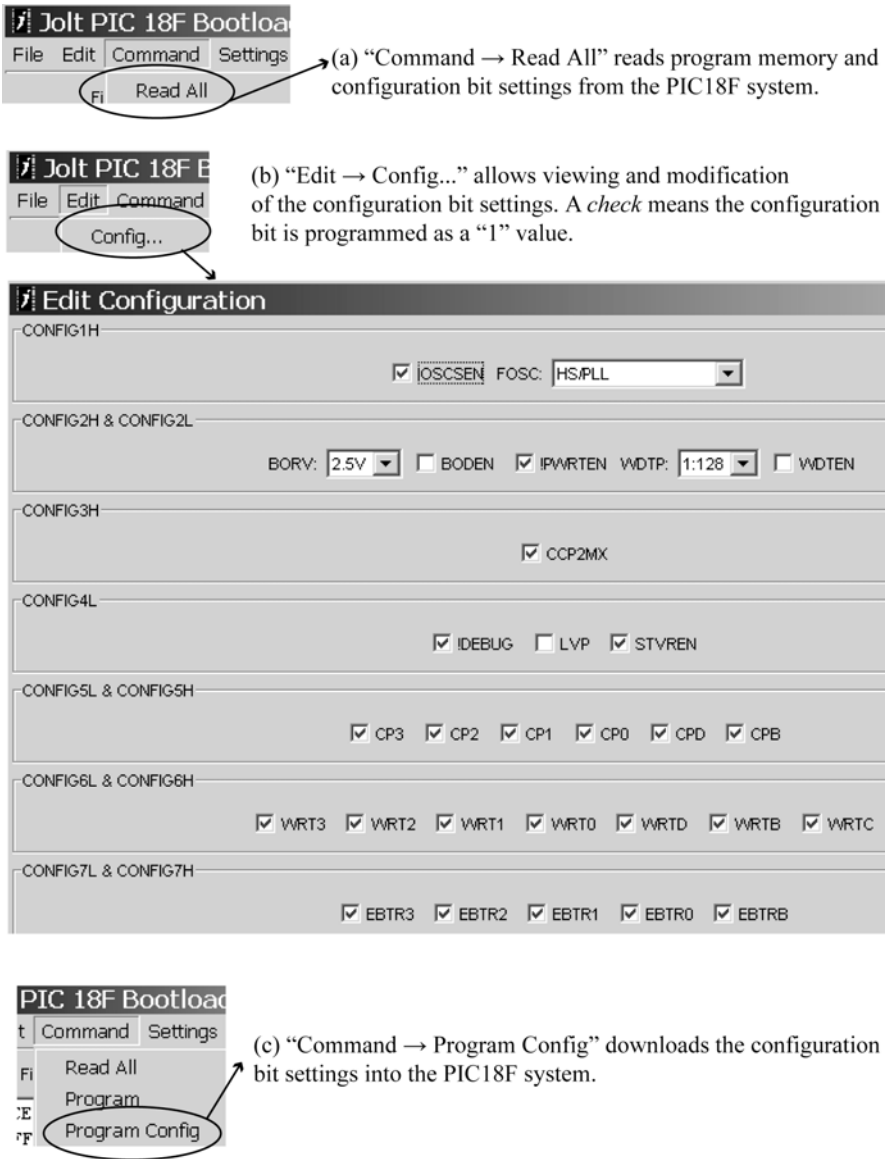
**FIGURE F.6** Jolt execution.

When preparing hex files for use with Jolt, the code must be relocated to begin at location 0x0200, as the Jolt firmware occupies the first 512 bytes of program memory. For the PICC-18 C compiler this is accomplished by using the “-a200” linker flag (see Appendix C, “HI-TECH PICC-18 C Compiler Demo for the PIC18F242”). After relocation, address 0x0200 contains the reset vector, which should be some form of branch or goto instruction. For PIC18F242 programs compiled with the PICC-18 C compiler, the first two bytes at location 0x0200 should be 0x0C, 0xEF (instruction word 0xEF0C, a goto). Forgetting to relocate the code during compilation is the most common error when using Jolt. The *settings* menu choice allows modification of default actions by Jolt. The “Settings → Reload before Program” option is useful, as this causes Jolt to always reload a hex file before programming. Executing “Command → Program” opens the Programming Device window that shows a progress bar that tracks the program download. When “Command → Program” is executed, the Jolt client sends a handshaking byte over the serial port; when the PIC18 firmware detects this handshake byte it responds with an acknowledge byte. When the client receives the acknowledge byte, it responds with program data. The PIC18 bootloader firmware is executed immediately after reset or power up; if the firmware does not detect the handshaking byte within two seconds, it jumps to the user program at location 0x200. The best way to establish the PC client to PIC18 firmware connection is to either hold the PIC18 reset button down or power off the PIC18 before executing “Command → Program”; after the progress window appears, release the PIC18 reset or turn on power to the PIC18. The debug checklist in Appendix E, “Suggested Laboratory Exercises,” gives tips on debugging the serial interface and Jolt operation. The Jolt firmware automatically detects the baud rate, so set this as high as can be reliably supported on your system. A value of 38400 was used for the PIC18F242 reference system in this book.

Figure F.7a shows how to read the current configuration bit settings from the PIC18 using “Command → Read All”.

The configuration bits can then be viewed and/or modified by opening the configuration bit window using the “Edit → Config...” command (Figure F.7b). Use “Command → Program Config” to download the new configuration bit settings into PIC18 system (Figure F.7c). It is recommended that you always read the current configuration bits from the target PIC18 system before editing their values in the Jolt configuration bit window.





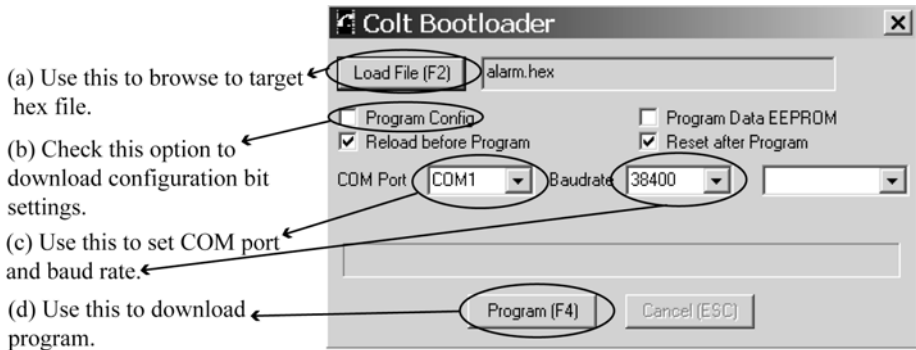
**FIGURE F.7** Modifying configuration bits with Jolt.

## F.4 COLT BOOTLOADER INSTALLATION AND EXECUTION



The Colt bootloader is a streamlined version of the Jolt bootloader. The self-installing Colt executable named *bootldr/ColtSetup.exe* on the companion

CD-ROM installs the Colt PIC18 bootloader; no further installation steps are necessary. Figure F.8 shows the Colt bootloader window.



**FIGURE F.8** Colt bootloader window.

All commands for locating the target hex file, program options, COM port/baud rate settings, and initiating program download are immediately available from this window. Programs must be relocated to begin at location 0x0200; use the “-a200” linker flag for the PICC-18 C compiler (see Appendix C). Configuration bits cannot be viewed or modified within Colt; they can only be downloaded into the target PIC18 system. Colt does not support reading program memory and configuration bit settings from the target PIC18 target system. Colt uses the same firmware as Jolt, and thus automatically detects the baud rate. A value of 38400 was used for the PIC18F242 reference system in this book.

### Known Problems with Colt V0.4

Colt serial communication did not function on at least one Windows 2000 system with Service Pack 4. Colt functioned as expected under Windows XP with Service Pack 1.

*This page intentionally left blank*

# G

# Circuits 001

This appendix gives a hobbyist level introduction to basic circuits, and covers the passive components used in this book's schematics.

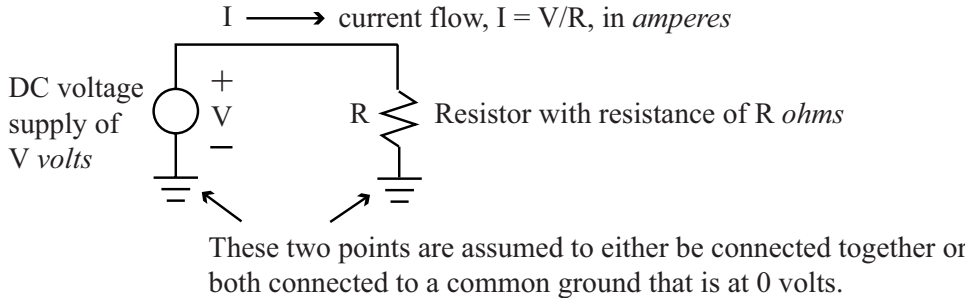
## G.1 VOLTAGE, CURRENT, RESISTANCE

---

*Current* is the flow of electrons through a conductor. A *conductor* is anything that allows current flow. A good conductor offers little *resistance* to current flow; in other words, it does not take much work for current to flow within a good conductor. In rough terms, the amount of work it takes to move electrons between two points on a conductor is *voltage*. The voltage difference between one end of a conductor and the other end of a conductor indicates the *resistance* of the conductor. If the voltage drop is high, the resistance is high; conversely, if the voltage drop is low, the resistance is low. A voltage supply provides a source of current at a fixed voltage level. Current is measured in Amperes (A), with a few milliamperes (mA, 1 mA = 0.001 A) being the typical current requirements of the integrated circuits used in this book. Voltage is measured in *volts* (V) and resistance is measured in *ohms* ( $\Omega$ ). The PIC18 and the integrated circuits in this book require a *Direct Current* (DC) voltage supply, typically with a voltage value of +5 V. A DC voltage supply means that the current flows in one direction only, and that the voltage is a constant value, either positive or negative. By contrast, the power available for household appliances from wall plugs is *Alternating Current* (AC), where the voltage varies in a sinusoidal fashion between  $\pm 120$  V with a frequency of 60 Hz. The AC current direction reverses itself each time the voltage value crosses 0 V.

## Ohm's Law

A *resistor* is a component with a fixed resistance value that is used to control current flow in an electrical circuit. Figure G.1 shows a basic DC circuit consisting of a DC voltage supply and one resistor with value  $R$ .



**FIGURE G.1** Voltage/current relationship with one resistor.

Equation G.1 gives *Ohm's Law*, which expresses the current ( $I$ ) flowing through the resistor as a function of voltage ( $V$ ) and resistance ( $R$ ).

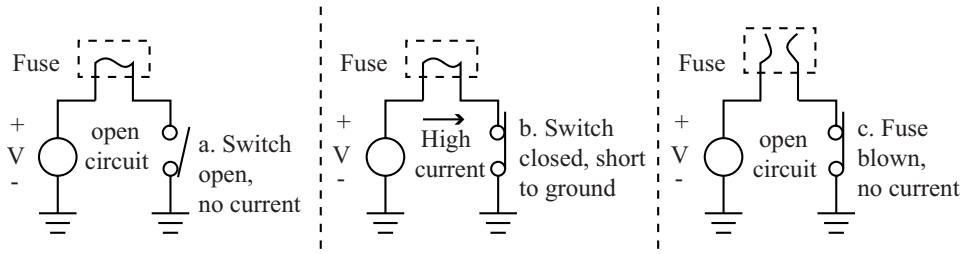
$$I = \frac{V}{R} \quad (\text{G.1})$$

An ideal DC voltage source supplies the current predicted by Equation G.1; thus, a resistance of zero causes infinite current flow. A zero resistance or very low resistance path is called a *short*, and causes large currents to flow. A physical power supply obviously cannot supply infinite current, and thus will either fail after a short period of time or blow an internal *fuse*, breaking the circuit path. Figure G.2 shows how a fuse is used to protect against shorts.

A fuse is a thin conductor that physically separates, breaking the connection, after a maximum rated current is reached. In Figure G.2a, the switch is open so no current is flowing. In Figure G.2b, the switch is closed, creating a short between  $V_{dd}$  and ground. In Figure G.2c, the fuse has blown, creating an open path and stopping current flow.

Equation G.2 is another form of Ohm's Law that expresses voltage across a resistor as the product of current and resistance.

$$V = I * R \quad (\text{G.2})$$

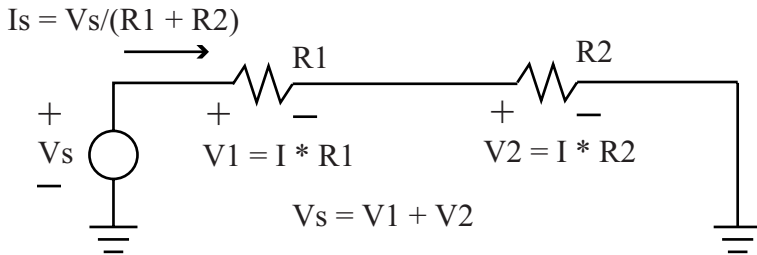


**FIGURE G.2** Using a fuse to protect against shorts.

### Resistors in Series

Figure G.3 shows a circuit with two resistors connected in *series*.

In this case, the current flowing through both resistors is the same and is expressed by Equation G.3, where the total resistance of the circuit is the sum of  $R_1$  and  $R_2$ .



**FIGURE G.3** Resistors in series.

$$I_s = \frac{V_s}{(R_1 + R_2)} \tag{G.3}$$

Equations G.4 and G.5 give the voltages  $V_1$  and  $V_2$  across each resistor.

$$V_1 = I_s * R_1 \tag{G.4}$$

$$V_2 = I_s * R_2 \tag{G.5}$$

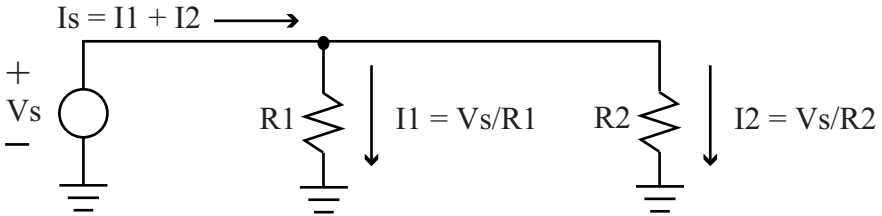
Resistors in series form a *voltage divider*, with the sum of voltages across the resistors equal to the voltage supply value as shown in Equation G.6.

$$V_s = V_1 + V_2 \quad (\text{G.6})$$

Voltage dividers are used in Chapter 12, “Data Conversion,” to build analog-to-digital and digital-to-analog converters. Observe that if  $R_1 = R_2$ ,  $V_1 = V_2 = V_s/2$ ; the voltage divides equally between the two resistors.

### Resistors in Parallel

Figure G.4 shows a circuit with two resistors connected in *parallel*.



**FIGURE G.4** Resistors in parallel.

In this case, the voltage across each resistor is the same and is equal to the power supply voltage  $V_s$ . However, the current flowing through each resistor is dependent upon the resistance value as given in Equations G.7 and G.8.

$$I_1 = \frac{V_s}{R_1} \quad (\text{G.7})$$

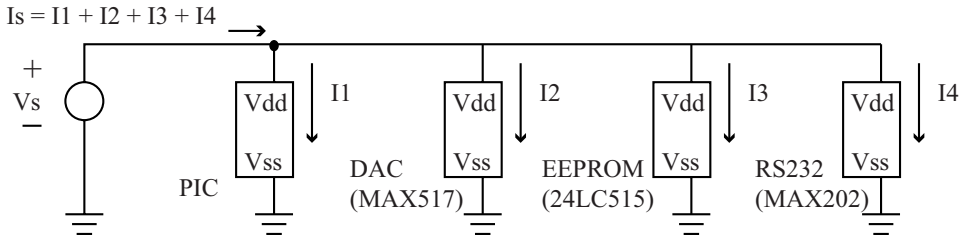
$$I_2 = \frac{V_s}{R_2} \quad (\text{G.8})$$

Resistors in parallel form a *current divider*, with the sum of the currents through the resistors equal to the total current drawn from the power supply ( $I_s$ ) as shown in Equation G.9.

$$I_s = I_1 + I_2 \quad (\text{G.9})$$

When measuring the total current through a system like the PIC18F242 reference board in this book, the current draw of each individual integrated circuit can be determined by simply removing it from the board, since the current draw of each integrated circuit adds to the total current draw. Figure G.5 illustrates this

concept. Observe that the integrated circuits are connected in parallel (all supplied with the same voltage).



**FIGURE G.5** Current draw in a total system.

## Polarization

Most circuit elements have two terminals through which current flows. The terminals can either be *polarized* (positive and negative terminals) or *unpolarized*. A DC power supply is *polarized*, it has clearly marked positive (+) and negative (–) terminals. The negative terminal is at zero volts (ground) and the positive terminal is the voltage output. A resistor is *unpolarized*; its operation is not affected by the direction in which its terminals are connected in a circuit.

## Diodes

A *diode* is a two-terminal device that allows current flow in one direction only. A diode's two terminals are named the *anode* and the *cathode*; when the voltage on the anode is approximately 0.7 V higher than the cathode voltage, current flows through the diode. Thus, a diode is a polarized device, as circuit operation is dependent upon how its terminals are connected in a circuit. On physical diodes, the cathode terminal is identified by either a band at one end or by being the shorter of the two leads. Figure G.6 shows some simple diode circuits.

In Figure G.6a, no current is flowing through the diode, as the anode voltage is only 0.3 V. In Figure G.6b, current flows through the diode, as the anode voltage is greater than the cathode voltage by more than 0.7 V. In Figure G.6c, no current is flowing as the diode direction is reversed in the circuit; the only way for current to flow in this circuit is if  $V_s$  produces a negative voltage. The resistor is included in series with the diode in Figure G.6 simply to limit the current flow within the circuit. A diode has internal resistance but its value depends upon the diode type. A light emitting diode (LED) emits visible light in proportion to the current flowing through it; the higher the current, the brighter the light.



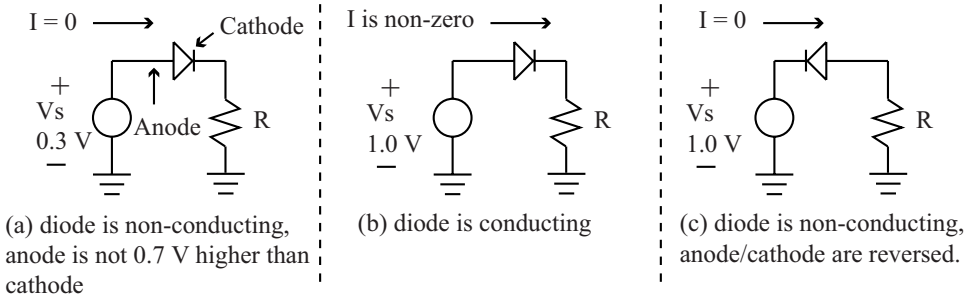


FIGURE G.6 Diode circuits.

## G.2 CAPACITORS

A *capacitor* is a two-terminal device that comes in both unpolarized and polarized varieties. In the PIC18F242 startup schematic (Figure 8.4), the capacitors used with the crystal are unpolarized, while the capacitor connected between the Vdd and Vss pins of the PIC18F242 is polarized. Equation G.10 gives the time-dependent current flow  $i(t)$  through a capacitor as a function of capacitance ( $C$ ) and the voltage rate of change ( $dv/dt$ ) across the capacitor.

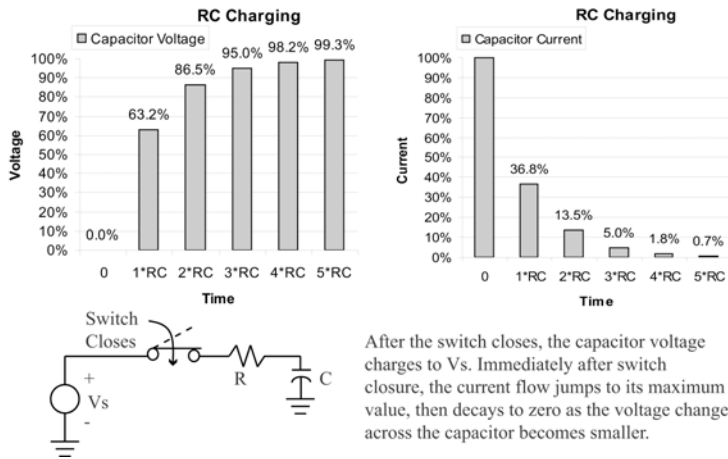
$$i(t) = C \frac{dv}{dt} \tag{G.10}$$

In intuitive terms, Equation G.10 says that if the voltage across a capacitor is not changing, then no current flows through the capacitor. Equation G.11 gives the time dependent voltage across a capacitor as a function of capacitance and current.

$$v(t) = \frac{1}{C} \int i \, dt \tag{G.11}$$

In intuitive terms, Equation G.11 says that a capacitor stores charge, increasing its voltage, as current flows through it. Figure G.7 shows the effect of Equations G.10 and G.11 in an RC series circuit.

When the switch is open, the current through the capacitor and the voltage across the capacitor are both zero. At time  $t = 0$ , the switch closes and the capacitor charges up in an exponentially decaying fashion to  $V_s$ . The current jumps to its maximum value immediately after the switch closure due to the instantaneous change in voltage (maximum  $dv/dt$ ), and then exponentially decays to zero as the change in voltage across the capacitor ( $dv/dt$ ) decreases. The Y-axis is time and is marked in RC units, where the  $R \cdot C$  product is called the *time constant* of the RC series circuit. The larger the time constant, the longer it takes for the capacitor to charge.

**FIGURE G.7** RC series circuit.

In the PIC18F242 reference system, the polarized capacitors placed across the  $V_{dd}$  and  $V_{ss}$  pins of the PIC18F242 are used to assist in supplying transient current needs caused by high-frequency digital switching. Capacitors used in this manner are called decoupling capacitors. Polarized capacitors are also used on the MAX202 RS232 interface device for charge storage when converting +5 V to the  $\pm 10$  V used for RS232 communication.

*This page intentionally left blank*

- [1] P. Ceruzzi, *A History of Modern Computing, Second Edition*. Cambridge, MA: The MIT Press, 2003.
- [2] G. Ifrah, *The Universal History of Computing*. New York: John Wiley & Sons, Inc, 2001.
- [3] M. Morris Mano, *Computer System Architecture, Second Edition*. Englewood Cliffs, NJ: Prentice-Hall, 1982.
- [4] Intel Corporation, "Appendix C: IA-32 Instruction Latency and Throughput," *IA-32 Intel Architecture Optimization Reference Manual*, Order Number 248966-011, pp. C.1–C.22.
- [5] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, Second Edition*. San Francisco, CA: Morgan Kaufmann Publishers, Inc., 1996.
- [6] Microchip Technology Inc., *PIC18FXX2 28/40-pin High Performance, Enhanced FLASH Microcontrollers with 10-bit A/D*, DS39564B, 2002. Available online at [www.microchip.com](http://www.microchip.com).
- [7] Dan Matthews, *AN849: Basic PICmicro Oscillator Design*, Microchip Technology Inc., DS00849A, 2002. Available online at [www.microchip.com](http://www.microchip.com).
- [8] Microchip Technology Inc., *PICmicro 18C MCU Family Reference Manual*, DS39500A, 2000. Available online at [www.microchip.com](http://www.microchip.com).
- [9] Hantronix, *HDM162116L-5 Dimensional Drawing, 16 Character x 2 Lines LCD Module*. Available online at [www.hantronix.com](http://www.hantronix.com).
- [10] Hantronix, *HDM162116L-5 LCD Module Commands*. Available online at [www.hantronix.com](http://www.hantronix.com).
- [11] Maxim Integrated Products, *MAXIM +5V, RS-232 Transceivers with 0.1uF External Capacitors, MAX200-MAX211/MAX213*, 19-0065, Rev 6, 10/03. Available online at [www.maxim-ic.com](http://www.maxim-ic.com).
- [12] Microchip Technology Inc., *MCP41XXX/42XXX Single/Dual Potentiometer with SPI™ Interface*, DS11195C, 2003. Available online at [www.microchip.com](http://www.microchip.com).

- [13] Microchip Technology Inc., *25AA640/25LC640 64K SPI™ Bus Serial EEPROM*, DS21223F, 2003. Available online at [www.microchip.com](http://www.microchip.com).
- [14] Philips Semiconductors, *The I<sup>2</sup>C Bus Specification Version 2.1*, 2001. Available online at [www.semiconductors.philips.com](http://www.semiconductors.philips.com).
- [15] *24AA515/25LC515/24FC515 512K I<sup>2</sup>C™ CMOS Serial EEPROM*, Microchip Technology Inc., DS21673C, 2003. Available online at [www.microchip.com](http://www.microchip.com).
- [16] Maxim Integrated Products, *DS32KHz 32768 Temperature-Compensated Crystal Oscillator*, REV 041603. Available online at [www.maxim-ic.com](http://www.maxim-ic.com).
- [17] Pericom Semiconductor Corporation, *Precision Wide Bandwidth Analog Switches PI5A317A/318A/319A*, PS7059F 0210/03. Available online at [www.pericom.com](http://www.pericom.com).
- [18] Texas Instruments, *CD4053B Triple 2-Channel Analog Multiplexer*, Revised October 2003. Available online at [www.ti.com](http://www.ti.com).
- [19] National Semiconductor, *LM 386 Low Voltage Audio Power Amplifier*, DS006976, August 2000. Available online at [www.national.com](http://www.national.com).
- [20] HVW Technologies Inc, *PIR Sensor Passive Infrared Motion Detector*, PN #IR-01-003, [www.HVWTech.com](http://www.HVWTech.com).
- [21] Maxim Integrated Products, *DS1621 Digital Thermometer and Thermostat*, REV 102299. Available online at [www.maxim-ic.com](http://www.maxim-ic.com).
- [22] Maxim Integrated Products, *MAX 667 +5V/Programmable Low-Dropout Voltage Regulator*, 19-3894, Rev 3, 10/94. Available online at [www.maxim-ic.com](http://www.maxim-ic.com).
- [23] Optrex Corporation, *Dot Matrix Character LCD Module User's Manual*. Available online at [www.optrex.com](http://www.optrex.com).
- [24] Cypress Semiconductor, *CY7C186 8Kx8 Static RAM*, Document #:38-05280, Revised March 22, 2002. Available online at [www.cypress.com](http://www.cypress.com).
- [25] Microchip Technology Inc., *PIC18C601/801 High Performance ROM-less Microcontrollers with External Memory Bus*, DS39541A, 2001. Available online at [www.microchip.com](http://www.microchip.com).
- [26] Cypress Semiconductor, *CY7C109B/CY7C1009B 128Kx8 Static RAM*, Document #:38-05038, Revised August 24, 2001. Available online at [www.cypress.com](http://www.cypress.com).
- [27] Atmel, *AT28C010 1M (128Kx8) Paged Parallel EEPROM*, Document #:38-05038, Rev 0353E-06/99. Available online at [www.atmel.com](http://www.atmel.com).
- [28] Robert Bosch GmbH, *CAN Specification/Version 2.0*, 1991. Available online at [www.can.bosch.com](http://www.can.bosch.com).
- [29] Microchip Technology Inc., *PIC18FXX8 28/40-pin High Performance, Enhanced FLASH Microcontrollers with CAN*, DS41159C, 2003. Available online at [www.microchip.com](http://www.microchip.com).

- [30] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips, *Universal Serial Bus Specification*, Revision 2.0, April 27, 2000. Available online at [www.usb.org](http://www.usb.org).
- [31] Maxim Integrated Products, *Getting Started with TINI*, Rev 0, 7/04. Available online at [www.maxim-ic.com](http://www.maxim-ic.com).
- [32] Qing Li, Caroline Yao, *Real-Time Concepts for Embedded Systems*, CMP Books, ISBN:1578201241, July 2003, 294 pp.
- [33] Frank Kolnick, *The QNX 4 Real-Time Operating System*, Basis Computer Systems, ISBN:0921960018, July 2000, 960 pp.
- [34] J.W. Bruce, *Nyquist-rate digital-to-analog converter architectures*, IEEE Potentials, vol. 20, no. 3, pp. 24–28, August 2001.
- [35] J.W. Bruce, *Nyquist-rate analog-to-digital converter architectures*, IEEE Potentials, vol. 17, no. 5, pp. 36–39, December 1998.
- [36] R.J. Baker, *CMOS: Circuit Design, Layout, and Simulation 2/e*, Wiley-Interscience and IEEE Press, Chapters 28–29, 2005.
- [37] B. Razavi, *Principles of Data Conversion System Design*, IEEE Press, 1995.
- [38] SBS Implementers Forum, *System Management Bus Specification*, Version 2.0, August 3, 2000. Available online at [www.smbus.org](http://www.smbus.org).

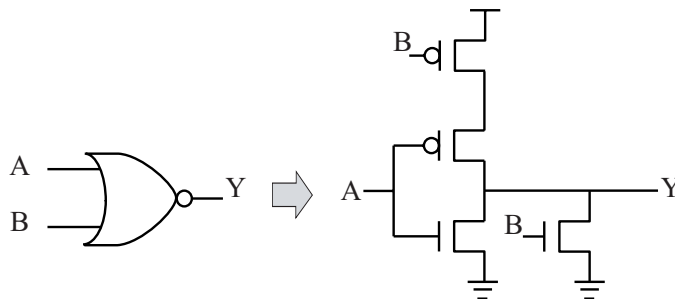
*This page intentionally left blank*

# Answers to Review Problems

This appendix contains answers to the odd-numbered review questions at the end of Chapters 1 through 13.

## I.1 CHAPTER 1

1.  $25 = 32, 26 = 64$ , so 6 bits.
3.  $120 = 0x78 = 0b0111\ 1000$
5.  $0xF4 = 0b1111\ 0100$
7.  $0b1011\ 0111 = 0xB7 = 11 \cdot 16 + 7 = 183$
9.  $0xB2 - 0x9F = 0x13$ .  $\sim 0x9F = \sim(0b1001\ 1111) = 0b0110\ 0000 = 0x60$ . So  $0xB2 - 0x9F = 0xB2 + \sim 0x9F + 0x01 = 0xB2 + 0x60 + 0x01 = 0x13$ .
11. See Figure I.1.

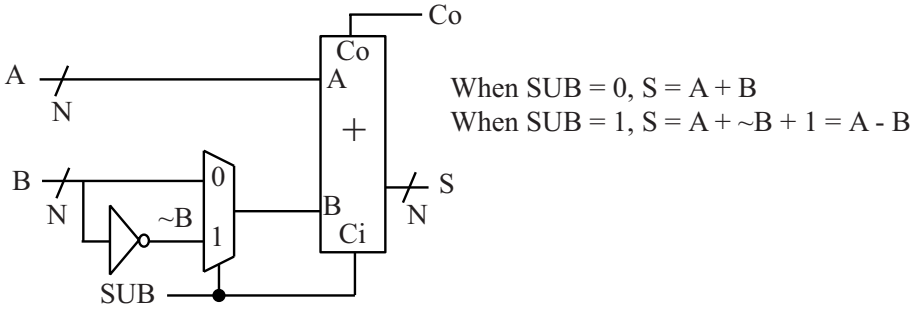


**FIGURE I.1** CMOS 2-input NOR (problem 1.11).

13.  $0x2A \ll 1 = 0x54$

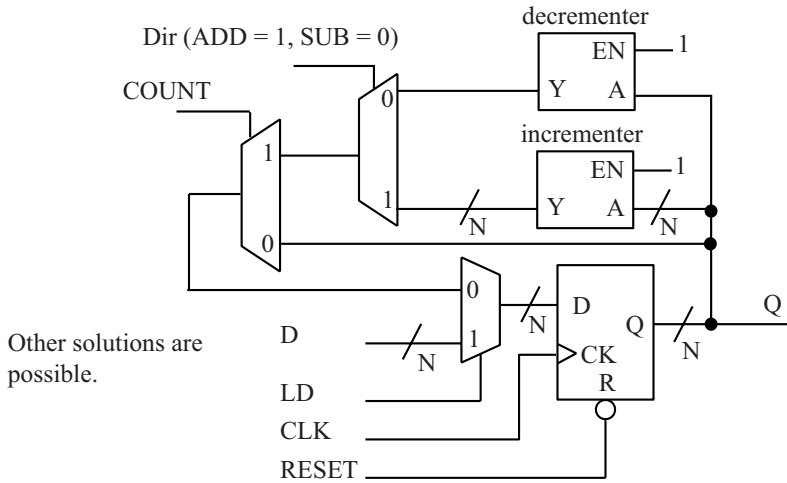


15.  $0.3 \times \text{period} = 20 \mu\text{s}$ ; so  $\text{period} = 20 \mu\text{s} / 0.3 = 66.7 \mu\text{s}$ . Frequency =  $1 / (66.7 \mu\text{s}) = 15 \text{ kHz}$
17. See Figure I.2.



**FIGURE I.2** Adder/subtractor (problem 1.17).

19. See Figure I.3.



**FIGURE I.3** Up/down counter (problem 1.19).

## I.2 CHAPTER 2

1.

Location	Machine Code	Mnemonics
		START:
0	01 0111	JC LOC_IS_1
1	10 0000	OUT 0
3	10 0010	OUT 2
4	10 0101 0	OUT 5
5	10 0111	OUT 7
6	00 0000	JMP START
		LOC_IS_1:
7	10 0001	OUT 1
8	10 0011	OUT 3
9	10 0110	OUT 6
10	10 1000	OUT 8
11	00 0000	JMP START

3. See Table I.1. Recall that the JC instruction takes the jump if LOC = 1.

**TABLE I.1** Problem 2.3

Cycle	Location	Comment
1	0	OUT 2, DOUT = 2 = 0b0010, LOC = LSb = 0
2	1	OUT 5, DOUT = 5 = 0b0101, LOC = LSb = 1
3	2	JC 5, DOUT = 5 = 0b0101, LOC = LSb = 1, so take jump
4	5	OUT 9, DOUT = 9 = 0b1001, LOC = LSb = 1
5	6	JC 2, DOUT = 9 = 0b1001, LOC = LSb = 1, so take jump
6	2	JC 5, DOUT = 9 = 0b1001, LOC = LSb = 1, so take jump
7	5	OUT 9, DOUT = 9 = 0b1001, LOC = LSb = 1
8	6	JC 2, DOUT = 9 = 0b1001, LOC = LSb = 1, so take jump
9	2	JC 5, DOUT = 9 = 0b1001, LOC = LSb = 1, so take jump
10	5	OUT 9, DOUT = 9 = 0b1001, LOC = LSb = 1

5. It takes 13 instructions.

```

START:
  JC LOCAL
  OUT 1
  OUT X1
  OUT X2
  OUT X3
LOCAL:
  OUT Y1
  OUT Y2
  OUT Y3
  OUT Z1
  OUT Z2
  OUT Z3
  OUT Z4
  JMP START
    
```

7. The first change is to increase the number of memory locations from 16 to 32. This causes the memory address bus to increase from 4 bits ( $2^4 = 16$ ) to 5 bits ( $2^5 = 32$ ). This means the Program Counter has to increase from 4 bits to 5 bits. Finally, the instruction size has to increase by 1 bit because the JC/JMP instruction data field specifies a location, which now requires 5 bits. Therefore, the new memory size is 32x7. See Figure I.4.

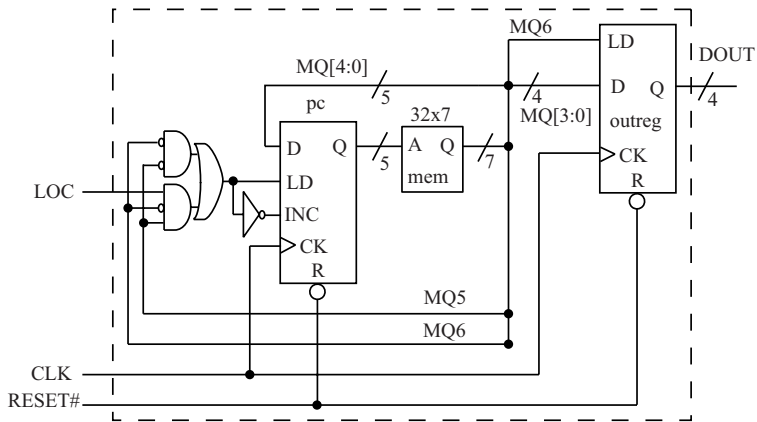


FIGURE I.4 Modified NSC design (problem 2.7).

9. The opcode field must be increased from 2 bits to 3 bits; this changes the memory from a 16 x 6 memory to a 16 x 7 memory (each location now contains 7 bits).

**I.3 CHAPTER 3**

---

1. `addwf 0x030, f` = 0010 01da ffff ffff ; destination is *f*, so *d* = 1. Location is 0x030, which is in the access bank, so *a* = 0. The *f* field = 0x30. So, 0010 0110 0011 0000 = 0x2630.
3. `goto 0x043E` = 1110 1111 *k*<sub>7</sub>kkk kkkk<sub>0</sub>, 1111 *k*<sub>19</sub>kkk kkkk kkkk<sub>8</sub>  
To determine the *k* field, take the target address, convert to 21-bit binary number, and discard the LSb, as the jump target address is always an even address.  
0x0043E = 0b (0 0000 0000 010)(0 0011 111)0, the digits in bold are the *k* bits (19 through 0). Parentheses show the *k*<sub>19</sub>-*k*<sub>8</sub> and *k*<sub>7</sub>-*k*<sub>0</sub> bit groupings.  
`goto 0x043E` = 1110 1111 0001 1111, 1111 0000 0000 0010 = 0xEF1F,0xF002
5. 0xC2A5 0xF100 is a two-word PIC18 instruction. The upper 4 bits of the first instruction word (1100) indicate this is a `movff` instruction. The lower 12 bits of the first instruction word specify the source address, while the lower 12 bits of the second instruction word specify the destination address. Thus, this is the instruction `movff 0x2A5,0x100`.
7. You don't know because BANKED means the Bank Select Register (BSR) provides the upper 4 bits of the instruction, and the contents of the BSR have not been specified. Even though 12 bits are specified in the address 0x230, the BSR must have the value 0x2 in it for location 0x230 to be modified.
9. The `movwf` and `addwf` instruction each require one instruction cycle, while the `goto` requires two instruction cycles, for a total of four instruction cycles. Each instruction cycle is four clock cycles, so a total of four instruction cycles \* 4 clocks = 16 clock cycles is required. A 16 MHz clock has a period of 1/(16.0E6) = 6.25E-8 = 62.5 ns. The total time needed is:  
$$16 \text{ clocks} * 62.5 \text{ ns} = 1000 \text{ ns} = 1.0 \mu\text{s}$$
11. `movff 0x100,0x200`  
`movff 0x101,0x201`  
`movff 0x102,0x202`  
`movff 0x103,0x203`
13. *W* is zero since the operation is *W* – *W*.
15. Move literal to *W* (`movlw`) moves the value 0x5C to *W*. So, *W* = 0x5C.
17. The instruction `addwf 0x5A, f` adds the contents of location 0x5A to *W*, and stores the result back in location 0x5A. Thus:

$$\text{location } 0x5A = (0x5A) + 0x3D = 0x3B + 0x3D = 0x78.$$

Location 0x5A now contains 0x78.

19.

```

movlb 0x2      ; must point BSR at bank2
movf 0x250,w   ; read 0x250 into W
addwf 0x251, w ; add W to 0x251, store in w
movlb 0x0      ; point at bank 0 (not needed as 0x5A is in
               ; access bank)
movwf 0x05A    ; save result in 0x05A

```

## I.4 CHAPTER 4

---

1. Location 0x001 = (0x001) - W = 0x7A - 0xD7 = 0xA3. Flags are C = 0 (borrow), Z = 0.
3. W = (0x000) & 0xD7 = 0xA0 & 0xD7 = 0x80. Flags are C = 0, Z = 0.
5. Location 0x4F changed to 0x00. No flags are affected, so C = 0, Z = 0.
7. Bit 4 of location 0x001 is cleared, so new value of 0x001 is 0x6A. No flags affected, so C = 0, Z = 0.
9. W = (0x001) << 1 (shift left) = 0x7A << 1 = 0xF4. Flags are C = 0, Z = 0.
11. W = (W) ^ 0x4E = 0xD7 ^ 0x4E = 0x99. Flags are C = 0, Z = 0.

13. BSR is set to 1 because k is in bank 1; variables i, j are in the access bank so the BSR register is not used for instructions that reference i, j using our standard assumptions for the access bit setting.

```

movlb 0x1      ; set BSR to 1 because K is in bank1
bcf STATUS, C  ; clear carry
rrcf i,f       ; i = i >> 1
movf k,w       ; w = k
iorwf i,f      ; i = i | k

```

15.

```

movlb 0x1      ; set BSR to 1 because K is in bank1
clrf i,f       ; i = 0
loop_top
movf i         ;
subwf k,w      ; i-k
bz end_loop    ; skip if i == k
incf k         ; k++
bcf STATUS,C   ; clear C so LSb gets a 0 value on shift
rlcf i,f       ; i= i << 1
bra loop_top
end_loop
...rest of code...

```

17. Note: !i || j is the same as i == 0 || j != 0.

```

movlb 0x1      ; set BSR to 1 because K is in bank1
movf i,f       ; test i

```

```

bz      if_body    ; do if i == 0
movf   j,f        ; test j
bz      else_body  ; do else if neither i==0 or j!=0
if_body
movf   i,w        ; here if i==0 or j !=0
addlw  2          ; w = i + 2
movwf  k          ; k = i + 2
bra    end_if     ; finished
else_body
bcf    STATUS,C   ; clear Carry so MSb gets 0 on shift
rlcf  k,f         ; k = k << 1
end_if
rest of code ....

```

19. Note that  $k = i * 12 = i * (8+4) = (i * 8) + (i * 4) = i \ll 3 + i \ll 2$ .

```

movlb  0x1        ; set BSR to 1 because K is in bank1
movff  i,k        ; k = i
bcf    STATUS,C   ; clear Carry so that LSb gets 0 on shift
rlcf  k,f         ; k = i << 1
bcf    STATUS,C   ; clear Carry so that LSb gets 0 on shift
rlcf  k,f         ; k = i << 2
movf   k,w        ; w = i << 2
bcf    STATUS,C   ; clear Carry so that LSb gets 0 on shift
rlcf  WREG,W      ; w = i << 3
addwf  k,f        ; k = i << 2 + i << 3

```

21. The machine code 0x9A04 is 0b1001 1010 0000 0100. The upper 4 bits indicate that this is a BCF instruction. The next 3 bits of 101 indicate bit #5. The access bit setting is "0" (use access bank), and the lower 8 bits is 0x04. Thus, the instruction is BCF 0x004, 5, ACCESS.

## I.5 CHAPTER 5

---

1.

```

top
movf   k,w
subwf  i,f        ; i LSB = i LSB - k LSB
movf   k+1,w
subwfb i+1,f      ; i MSB = i MSB - k MSB
; need to do [ i - (j+k) ] for comparison, use
; temporary location, do j+k first
movf   j,w
addwf  k,w        ; w = i LSB + k LSB
movwf  tmp        ; save LSB to tmp LSB, a temporary
                  location
movf   j+1,w
addwfc k+1,w      ; w = i LSB + k LSB
movwf  tmp+1      ; save MSB to tmp MSB

```

```

movf    tmp,w
subwf   i,w           ;i LSB - tmp LSB
movf    tmp+1,w
subwfb  i+1,w         ;i MSB - tmp MSB
bnc     top           ;branch C=0, borrow, so i < (j+k)
....end of loop, rest of code...

```

3.

```

movf    i,w
iorwf   j,w           ; W = i LSB | j LSB
movwf   k             ; save to k LSB
movf    i+1,w
iorwf   j+1,w         ; W = i MSB | j MSB
movwf   k+1           ; save to k MSB

```

5.  $+42 = 0x2A$ , so  $-42 = 0 - 42 = 0x00 - 2A = 0xD6$ 

7. The MSb is 1, so the number is negative (-).

Magnitude is  $0x000 - 0xBA3 = 0x45D = 1117$ .Final answer:  $-1117$ .9. The number is negative (MSb = 1), so extend with all "1"s (0xF hex digits) so the sign extended value is value is  $0xFF85$ .11.  $0x90 - 0x8A = 0x06$ , the flag settings are  $Z=0, N=0, V=0, C = 1$ . This is a negative number minus a negative number, which is the same as a negative number plus a positive number and this cannot produce two's complement overflow.13.  $0x2A - 0x81 = 0xA9$ , the flag settings are  $Z=0, N=1, V=1, C = 0$ . This is positive number minus a negative number, which is the same as a positive number plus a positive number. The result should be positive, but a negative number is the result, which means that two's complement overflow occurred.15.  $i > j$  is false as  $i$  is negative,  $j$  is positive, so  $k = 0$  (Boolean false returns 0).17. This is a signed right shift because  $k$  is declared as a signed char, so you need to keep the sign digit when shifting. $0xA0 = 0b1010\ 0000$  (-96) $0xA0 \gg 1 = 0b1101\ 0000 = 0xD0$  (-48) $0xA0 \gg 2 = 0b1110\ 1000 = 0xE8$  (-24)19. For  $k \geq j$  test, do  $k - j$ . If  $k \geq j$ , then the result is positive ( $N = 0, V = 0$ ). But if overflow occurs, then  $N = 1, V = 1$ .

```

movf    j,w
subwf   k,w           ; k - j
bov     V_1           ; branch if V = 1
bnn     if_body       ; do if_body if V = 0, N = 0
bra     end_if        ; reach here if V = 0, N = 1, skip if body
V_1
bnn     end_if        ; skip if body if V = 1, N = 0
if_body
bcf     STATUS,C     ; clear C for right shift

```

```

    btfsc i,7      ; skip if number is positive (sign = 0)
    bsf  STATUS,C ; number is negative, so set C=1 to keep
sign=1
    rrcf i,f      ; do right shift
    bcf  STATUS,C ; repeat for next shift
    btfsc i,7
    bsf  STATUS,C
    rrcf i,f
end_if
...rest of code...

```

21. The instruction `bc there` is jumped to if  $C = 1$ . The instruction following `bc there` is executed if  $C = 0$ . So, replace:

```

bc there      ; branch to there if C=1, but there is too far away
next
..some instruction..

```

with a code sequence that uses a `goto there` instruction, which can jump anywhere in the program code space.

```

    btfsc STATUS, C ;bit test, skip if clear
    goto  there     ;not skipped if C=1, use goto statement
next
..some instruction..

```

## I.6 CHAPTER 6

---

1. The address of the next instruction, or  $PC+4 = 0x0104$ .
3. If target address of the `rcall` is further away than  $-1024$  or  $+1023$  instruction words, a `call` instruction must be used.
5. The FSR0 SFR is changed to  $0x024$  by `lfsr FSR0, 0x024`. The `decf INDF0, f` instruction decrements the value of the memory location pointed to by FSR0, so the value of location  $0x024$  is changed from  $0xC7$  to  $0xC6$ .
7. The FSR0 SFR is changed to  $0x024$  by `lfsr FSR0, 0x024`. The `incf POSTINC0, f` instruction increments the value of the memory location pointed to by FSR0, so the value of location  $0x024$  is changed from  $0xC7$  to  $0xC8$ . The `POSTINC0` mode causes FSR0 to be incremented after the `incf` instruction is executed, so the final value of FSR0 is  $0x025$ .
9. Table I.2 shows the memory locations used for the `char a[]` array. The content of location  $0x0102$  is copied to location  $0x0101$  by the statement `*(ptr+1) = *(ptr+2)`.



**TABLE I.2** Solution for Problem 5.9

Initial Contents		Final Value		Comment
0x0100	0x34	0x0100	0x34	
0x0101	0x24	0x0101	0x11	New value is copied from location 0x0102
0x0102	0x11	0x0102	0x11	
0x0103	0xFE	0x0103	0xFE	

11. Table I.3 shows the memory locations used for the `int a[]` array. The hex memory contents are the integer values converted to two's complement 16-bit values stored in little-endian order. Thus,  $-234$  is `0xFF16` stored in little-endian order as `0x16` (location `0x0100`), `0xFF` (location `0x0101`). Array element `a[2]` (locations `0x0104`, `0x105`) is copied to array element `a[1]` (locations `0x0102`, `0x103`) by the statement `*(ptr+1) = *(ptr+2)`. The `ptr` variable is declared as an `int *` type (pointer to an integer), so the address `ptr+1` is computed as  $ptr+1*2 = 0x0100 + 2 = 0x0102$ , as each `int` element is 2 bytes. Similarly, the address `ptr+2` is computed as  $ptr+2*2 = 0x0100 + 4 = 0x0104$ . Thus, the statement `*(ptr+1) = *(ptr+2)` reads as “copy the `int` element starting at location `0x0104` to the `int` element starting at location `0x0102`.”

**TABLE I.3** Solution for Problem 5.11

Initial Contents		Final Value		Comment
0x0100	0x16	0x0100	0x16	
0x0101	0xFF	0x0101	0xFF	
0x0102	0x78	0x0102	<b>0x30</b>	Array value <code>a[1]</code> is replaced by array value <code>a[2]</code> ; two locations are modified.
0x0103	0x00	0x0103	<b>0x75</b>	
0x0104	0x30	0x0104	0x30	
0x0105	0x75	0x0105	0x75	
0x0106	0xE0	0x0106	0xE0	
0x0107	0xB1	0x0107	0xB1	

13. It is assumed pointer values are passed in the s1, s2 locations of the parameter block.

```

CBLOCK ????          ;static allocation for parameters
    s1:2, s2:2, c    ;two bytes needed for each pointer
ENDC
;use FSR0 for accessing s1, FSR1 for accessing s2
movff    s1,FSR0L
movff    s1+1,FSROH      ;FSR0 = s1
movff    s2,FSR1L
movff    s2+1,FSR1H      ;FSR1 = s2
loop_top
movf     INDF0,w          ;test *s1
bz       exit            ;exit if zero
movff    INDF0,c          ;c = *s1
movff    INDF1,INDF0     ;*s1 = *s2
movff    c,INDF2         ;*s2 = c
movf     POSTINC0,w      ;s1++
movf     POSTINC1,w      ;s2++
bra      loop_top
exit
return

```

15. This small C subroutine expands into a large amount of PIC18 code because of the use of long data types, and the need for computing the pointer addresses ptr+i and ptr+j.

```

CBLOCK ????          ;static allocation for parameters
    ptr:2, i, j, k:4; ;need 4 bytes for k, it is a LONG
ENDC

;; multiply i,j both by 4 as they are
;; indexes used with a LONG type pointer, so all pointer
;; indexes has to be multiplied by 4. Assume i,j < 64 so
;; no overflow occurs. *4 is done by shifting right twice
bcf     STATUS,C
rlcf    i,f
bcf     STATUS,C
rlcf    i,f            ; i = i * 4;
bcf     STATUS,C
rlcf    j,f
bcf     STATUS,C
rlcf    j,f            ; j = j * 4;

;compute ptr+i, save in FSR1
movff    ptr,FSR1L
movff    ptr+1,FSR1H    ;FSR1 = ptr
movf     i,w
addwf    FSR1L,f
movlw    0
addwfc   FSR1H,f        ;FSR1 = ptr+i

;now do k = *(ptr+i)
;each operand is 4 bytes long, need to move each byte.
movff    POSTINC1,k     ;k =*(ptr + i) low byte

```

```

movff  POSTINC1,k+1      ;k = *(ptr + i) 2nd byte
movff  POSTINC1,k+2      ;k = *(ptr + i) 3rd byte
movff  POSTINC1,k+3      ;k = *(ptr + i) MSByte

; compute ptr+i, save in FSR1 (old FSR1 destroyed by
; by POSTINC1 address mode used previously
movff  ptr,FSR1L
movff  ptr+1,FSR1H      ;FSR1 = ptr
movf   i,w
addwf  FSR1L,f
movlw  0
addwfc FSR1H,f         ;FSR1 = ptr+i

;compute ptr+j, save in FSR2
movff  ptr,FSR2L
movff  ptr+1,FSR2H      ;FSR2
movf   j,w
addwf  FSR2L,f
movlw  0
addwfc FSR2H,f         ;FSR2 = ptr+j
; now do *(ptr+i) = *(ptr+j)

movff  POSTINC2,POSTINC1 ;*(ptr+i) = *(ptr+j) lowbyte
movff  POSTINC2,POSTINC1 ;*(ptr+i) = *(ptr+j) 2nd byte
movff  POSTINC2,POSTINC1 ;*(ptr+i) = *(ptr+j) 3rd byte
movff  POSTINC2,POSTINC1 ;*(ptr+i) = *(ptr+j) MSbyte

;compute ptr+j, save in FSR2
movff  ptr,FSR2L
movff  ptr+1,FSR2H      ;FSR2 = ptr
movf   j,w
addwf  FSR2L,f
movlw  0
addwfc FSR2H,f         ;FSR2 = ptr+j

; now do *(ptr+j) = k

movff  k,POSTINC2       ;k = *(ptr+j) lowbyte
movff  k+1,POSTINC2     ;k = *(ptr+j) 2nd byte
movff  k+2,POSTINC2     ;k = *(ptr+j) 3rd byte
movff  k+2,POSTINC2     ;k = *(ptr+j) MSbyte

return

```

17.

```

CBLOCK ????           ;static allocation for parameters
    ia:2, ib:2, cnt;   ;two bytes needed for each pointer
ENDC

    movff    ia,FSR0L
    movff    ia+1,FSR0H      ;FSR0 = ia
    movff    ib,FSR1L

```

```

    movff    ib+1,FSR1H        ;FSR1 = ib
loop_top
    movf    cnt,w              ;test cnt
    bz      exit
    movf    POSTINC1,w
    addwf   POSTINC0,f        ;low byte add, ptr increment
    movf    POSTINC1,w
    addwfc  POSTINC0,f        ;high byte add with carry
    ;; don't have to increment ia, ib as they are already
    ;; incremented by use of the POSTINCx addressing mode
    decf   cnt,f
    bra    loop_top
exit
    return

```

19.

```

CBLOCK ????
    s:2, c;
ENDC

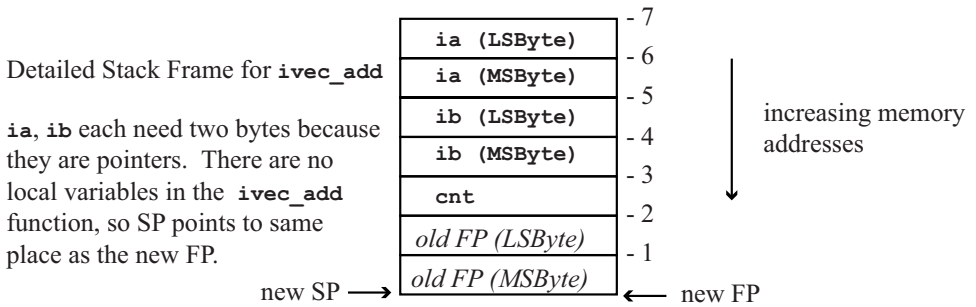
```

```

    movff   s,FSROL
    movff   s+1,FSROH        ;FSRO = s
loop_top
    call    getch
    movwf   c                ; subr. return value in W!
    movff   c,POSTINC0      ;*s = c, s++
    movf    c,w              ;test c
    bnz    loop_top
    return

```

21. Each call to `fib` consumes 4 bytes on the data stack. For  $n = 1$  or  $n = 0$ , no additional calls are made, so the number of data stack bytes needed is 4. For  $n > 1$ , a call is made to `fib(n-1)`, and then a call is made to `fib(n-2)`. If the data stack does not overflow for the call to `fib(n-1)`, it will not overflow for `fib(n-2)`. So, for  $n > 1$ , the number of data bytes needed is  $4 + (n-1)*4 = n*4$ . There are 128 bytes available on the data stack, so the maximum value for  $n$  is  $n \leq 128/4$ , or  $n \leq 32$ . On the return address stack, there is room for 31 return addresses. For  $n = 32$ , the maximum call depth to `fib` is 32 (includes the first call), so the return address stack will overflow before the data stack runs out of space. All of this is somewhat moot, however, because the `char` computations limit  $n$  to value of 13 to keep the result less than 255. Using an `int` type for  $n$  would increase the storage required on the stack, thus reducing the number of stack frames that could be stored.
23. See Figure I.5.



**FIGURE I.5** Stack frame for `ivec_add` (problem 5.23).

25. Assume `FSR1` is the stack pointer. Define a push as  $(FSR1) \leftarrow w$ ;  $FSR1--$ , or `movwf POSTDEC1` pushes `W` on the data stack. This means the location that the stack pointer is initialized to will contain active data after a push is done. Define a pop as  $++FSR1$ ;  $W \leftarrow ((FSR1))$ , or `movf PREINC1, w` pops a value from the stack into `W`. Thus, the stack grows toward decreasing memory locations on pushes, and grows toward increasing memory locations on pops.

## I.7 CHAPTER 7

1.  $0x39 * 0xAD = 57 * 173 = 9861 = 0x2685$
3. An 8-bit \* 16-bit product needs 24 bits to avoid overflow, so a long (32-bit) data type is required.
5.  $0x93AD \div 0xC5 = 37805 \div 197 = 191 (0xBF)$ , remainder is 178 (0xB2).
7. If the divisor is zero, this causes overflow because the MSByte of the dividend is greater than or equal to the divisor. When overflow is detected, the function aborts and returns with the carry flag set to "1". The carry flag is returned cleared if no overflow occurs.
9.  $0xC4$  as a 0.8 fixed-point number is  $0.11000100 = 1*2^{-1} + 1*2^{-2} + 1*2^{-6} = 0.765625$ .
11.  $0x39 + 0x59 = 0x92$  using normal binary addition. As two complement numbers, overflow occurs as the sum of two positive numbers produced a negative number. In signed saturating addition, the result is saturated to the maximum positive value, or  $0x39 + 0x59 = 0x7F$ .
13. First, convert the decimal value 0.15625 to binary.
  - $0.15625 * 2 = 0.3125$ , this is less than 1, so first bit (MSb) = 0
  - $0.3125 * 2 = 0.625$ , this is less than 1, so next bit = 0

$0.625 * 2 = 1.25$ , this is greater than 1, so next bit = 1  
 $1.25 - 1.0 = 0.25$ ,  $0.25 * 2 = 0.5$ , this is less than 1, so next bit = 0  
 $0.5 * 2 = 1.0$ , this is equal to 1, so next bit = 1  
 $- 1.0 = 0$ , remaining bits are 0  
 0.15625 in binary is  $0.00101 * 2^0 = 1.01 * 2^{-3}$  (normalized form)  
 Sign bit is 1 as sign is negative.  
 Exponent field is  $-3 + 127 = 124 = 0x7C = 01111100$ .  
 Significand field is 0100000000000000000000 (23 bits), the leading “1”  
 is dropped as this is understood in normalized form.  
 Complete number is 10111110001000000000000000000000 =  
 0xBE200000.

15. Assume the numbers being compared are  $a$  and  $b$ .
- If a sign bit == b sign bit, go to step C; else go to step B.
  - If a sign bit < b sign bit, then  $a > b$  (a positive, b negative); else  $a < b$  (a negative, b positive) finished.
  - If a exponent field == b exponent field, go to step E; else go to step D.
  - If a exponent field < b exponent field, then  $a < b$  else  $a > b$ , finished.
  - If a significand field != b significand field, go to step F; else numbers are equal and return.
  - If a significand field < b significand field, then  $a < b$  else  $a > b$ , finished.
17. The ten’s complement of 0x58 is  $0x99 - 0x58 + 1 = 0x42$ .
19. Listing 7.4 implements a divide algorithm for a 16-bit dividend and an 8-bit divisor, and returns an 8-bit quotient and 8-bit remainder. The first step of Table 7.9 divides the number by 10 (0x0A). The algorithm of Listing 7.4 overflows if the 8-bit divisor is greater than or equal to the MSByte of the 16-bit dividend, so the largest number that can be converted must be less than or equal to  $0x09FF = 2559$ .

## **I.8 CHAPTER 8**

---

- Your answers may vary depending on the PICC-18 compiler version; this answer was obtained with PICC-18 V8.35PL2. The map file contains:
 

```

_main          text    000058    (location 0x0058 in program memory).
_a_delay      text    00001C    (location 0x001C in program memory).
```

The variable `i` is not in the map file because it is an auto variable. Open a program memory window within MPLAB and look at location `a_delay()` (location `0x001C`) for some code that initializes a data location to 200 (`0x00ca`). You should find the following code fragment a few locations after the start of the `a_delay()` function:

```
movlw    0xc8           ;w = 0xc8
movwf   0xff6,ACCESS   ;initialize low byte of i to 0xc8
clrf    0xff7, ACCESS  ;initialize high byte of i to 0x00
```

So, variable `int i` is located in BANK 0 (because of the ACCESS bit) using locations `0xff6` (LSByte) and `0xff7` (MSByte). This is actually in the Special Function Registers (see Appendix A, “PIC18Fxx2 Architecture, Instruction Set, Register Summary,” for a memory map of the Special Function Registers) with the TBLPTRL register used for the `i` LSByte and the TBLPTRU register used for the `i` MSByte. This is possible because this simple program does not use these SFRs. This is an interesting compiler optimization; the compiler takes advantage of unused SFRs, thus saving RAM space. If you compile with the “`-a200`” flag to produce code compatible with the serial bootloader, then `main()` and `a_delay()` are shifted by `0x0200`:

```
_main      text    000258  (location 0x0258 in program memory).
_a_delay   text    00021C  (location 0x021C in program memory).
```

However, the location of variable `i` is unchanged.

- From the OUTPUT window of MPLAB with full optimization (global optimization level 9) and PICC-18 V8.35PL2:

```
Total ROM used  80 bytes (0.5%)
Total RAM used   0 bytes (0.0%)   Near RAM used      0 bytes (0.0%)
```

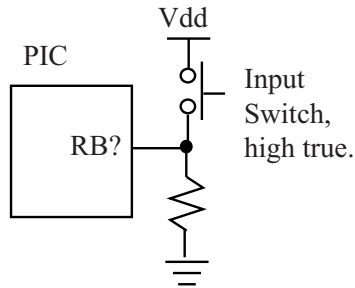
No RAM is used because the auto variables in the `a_delay()` function are allocated to unused SFRs. From the OUTPUT window of MPLAB with optimization turn off and PICC-18 V8.35PL2:

```
Total ROM used  84 bytes (0.5%)
Total RAM used   4 bytes (0.5%)   Near RAM used      0 bytes (0.0%)
```

Observe that 4 bytes of RAM are now needed to hold the auto variables `int i, k` of function `a_delay()`.

- This requires  $[(x \cdot \text{FOSC}/12\text{MHz})] < 256$  as `_dcnt` is a char variable, so  $x < [(256 \cdot 12 \text{ MHz})/\text{FOSC}]$ , or  $x < [(256 \cdot 12)/30] < 102.4$ . If  $x = 102$ , then  $[(x \cdot 30)/12] = 255$ , the maximum value for `_dcnt`.

7. See Figure I.6.



**FIGURE I.6** High true switch (problem 8.7).

9. See Figure I.7.

11. Between 4.5 mA and 5 mA.

13.

```

TRISB4 = 0; // RB4 output
TRISB6 = 1; //RB6 input
while(1) {
    //toggle LED
    if (LB4) RB4 = 0; else RB4 = 1;
    if (!RB6) {
        // switch is pushed, so delay for 1/2 second, every two times
        through
        // loop LED blinks so 1/2 second delay is 1 second blink
        DelayMs(200); DelayMs(200); DelayMs(100);
    } else {
        // switch is not pushed, delay for 1/4 second, every two
        times through
        // loop LED blinks so 1/4 second delay is 1/2 second blink
        DelayMs(200); DelayMs(50);
    } //end if
} // end while

```

15. This code does not include delays for switch debouncing.

```

TRISB = 0xF0; //RB7-RB4 inputs, RB3:RB0 outputs
PORTB = 0; // all LEDs are off
while(1) {
    while(RB4); while(!RB4); // wait for press/release
    RB0 = 1; RB3 = 0; //turn on LED0, turn off LED3
    while(RB4); while(!RB4); // wait for press/release
    RB1 = 1; RB0 = 0; //turn on LED1, turn off LED0
    while(RB4); while(!RB4); // wait for press/release
    RB2 = 1; RB1 = 0; //turn on LED2, turn off LED1
    while(RB4); while(!RB4); // wait for press/release
    RB3 = 1; RB2 = 0; //turn on LED3, turn off LED2
}

```



```

persistent char reset_cnt;

main(void){
    int i;
    char c;

    serial_init(95,1); // 19200 in HSPLL mode, crystal = 7.3728 MHz
    pcrLf();
    if (POR == 0){
        printf("Power-on reset has occurred."); pcrLf();
        POR = 1; // setting to bit to 1 means that will
                // remain a '1' for other reset types
        reset_cnt = 0;
    }
    if (TO == 0) {
        SWDTEN = 0; // disable watchdog timer
        printf("Watchdog timer reset has occurred."); pcrLf();
    }
    if (RI == 0) {
        printf("Software reset has occurred!"); pcrLf(); RI = 1;
    }
    reset_cnt;
    printf("Reset cnt is: %d",i);
    pcrLf();
    reset_cnt++;
    while(1) {
        printf("'1' to enable watchdog timer"); pcrLf();
        printf("'2' for sleep mode"); pcrLf();
        printf("'3 ' for both watchdog timer and sleep mode"); pcrLf();
        printf("'4 ' software reset "); pcrLf();
        printf("Anything else does nothing, enter keypress: ");
        c = getch();
        putchar(c);
        pcrLf();
        if (c == '1') SWDTEN = 1; // enable watchdog timer
        else if (c == '2') asm("sleep");
        else if (c == '3') {
            SWDTEN = 1; // enable watchdog timer
            asm("sleep");
        }
        else if (c == '4') {
            asm("reset");
        }
    }
}

```

Detects software reset.  
Set RI bit back to a "1".

New choice for software reset.

Do software reset

**FIGURE I.7** Detecting a software reset (problem 8.9).

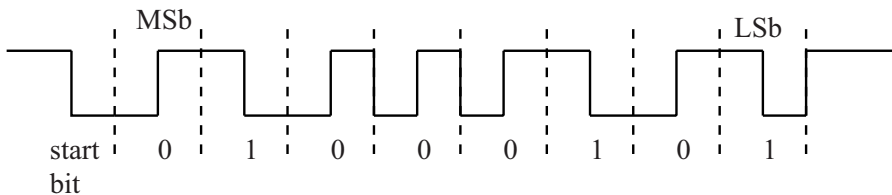
- 17. Current varies linearly with frequency, so expected current draw would be 16 mA/4 ~ 4 mA.
- 19. Reading LA4 returns the last thing written to LA4, while reading RB4 reads the external pin value. A complication is that RA4 is an open-drain output; it can only pull low. After RA4 = 0 is done, the output pin is pulled low, so reading RA4 returns a "0". Reading LA4 also returns a "0". After RA4 = 1 is done, the output pin is now floating, so reading RA4 returns an unpre-

dictable value, it could either read as a “0” or as a “1”. Reading LA4 will return a “1”, as that is the content of the LA4 latch.

## I.9 CHAPTER 9

1. Data transfers happen at a 4 MHz rate (every 2nd clock edge), so 4 bytes (32 bits) are transferred every 0.25  $\mu$ s. Bandwidth is 4 bytes/0.25  $\mu$ s = 16 MB/second.
3.  $1/19200 = 5.21 \text{ e-}5 = 52.1 \mu\text{s}$
5. One bit time at 115,200 baud is  $1/115,200 = 8.68 \mu\text{s}$ . Transmission time for one 8-bit datum is 8 + 1 stop + 1 start = 10 bit times, so  $10 * 8.68$

0x45 = 0b01000101 sent as biphasic data



**FIGURE I.8** Asynchronous serial waveform for 0x38 (problem 9.7).

- us = 86.8  $\mu$ s/byte. Bandwidth is bytes/second, so  $1/(86.6 \mu\text{s/byte}) = 11,520 \text{ bytes/second} = 11,520 \text{ B/s}$ .
7. See Figure I.8.
  9. The 7-bit value 0x38 (0b111000) has an odd number of 1s, so the parity bit value must be “1” to make even parity.
  11. For each bit,  $\pm 5 \text{ clocks} = \pm 5/16 * 100\% = \pm 31.25\%$  error tolerance for 1 bit. Over 16 data + 1 stop + 1 stop bit, the error tolerance becomes  $\pm 31.25\%/18 = 1.74 \%$ .
  13. With a 6 MHz FOSC, supported baud rate range from the listed ranges are 4800 to 38400 (above 38400 error become > 3%).
  15. A framing error occurs when a stop bit is detected as a “0” bit. If the sender’s baud rate is less than the receiver’s baud rate, the sender will still be sending “1”s, “0”s when the receiver is expecting a stop bit. Therefore, a framing error is more likely to occur when the sender baud rate is less than the receiver baud rate. If the sender has a higher baud rate than the re-

ceiver, a framing occurs if the sender begins sending a second data frame when the receiver is expecting a stop bit.

17. RS232 signaling values are  $-3\text{ V}$  to  $-25\text{ V}$  (logic 1) and  $+3\text{ V}$  to  $+25\text{ V}$  for logic 0. The large disparity between a “0” and a “1” provides for greater noise immunity, and hence greater reliability. For the MAX202, minimum output swing is  $\pm 5\text{ V}$ . The minimum receiver low threshold is  $0.8\text{ V}$ , which gives  $\text{abs}(0.8\text{ V} - (-5\text{ V})) = 5.8\text{ V}$  low threshold noise margin. The maximum receiver high threshold is  $2.4\text{ V}$ , which gives  $5\text{ V} - 2.4\text{ V} = 2.6\text{ V}$  high threshold noise margin. By contrast, typical CMOS logic noise margins for a  $5\text{ V}$  supply are  $0.4\text{ V}$  (low threshold) and  $1.1\text{ V}$  (high threshold).
19. You would need to reverse the order of the entries in the search table to search highest to lowest. A problem arises in that a framing error will probably occur for the higher baud rates, as the sender is still sending data bits when the receiver is expecting a stop bit. This means that when a FERR occurs, you do not start back to the beginning of the baud rate array, but just reset the USART module and go to the next lowest baud rate. If the end of the baud rate array is reached and no match is found, loop back to the beginning of the array and keep trying until a carriage return character is found. Because of the FERR problems with the higher baud rates, this approach will probably take longer to find the correct baud rate than the search from low to high baud rates.

## I.10 CHAPTER 10

---

1. Both the `retfie` and `return` instructions pop a return address from the return address stack, and restore BSR, W, and STATUS from the shadow registers if the *s* bit in the `return/retfie` instruction word is a “1”. However, `retfie` performs one additional action. Recall that when an interrupt occurs, the GIE bit is cleared ( $\text{GIE} = 0$ ) by the interrupt hardware so that all interrupts are disabled when ISR is entered. As part of its execution, the `retfie` instruction sets  $\text{GIE} = 1$ , thus re-enabling interrupts when the foreground code is re-entered.
3. Gate *g1* disables priorities if  $\text{IPEN} = 0$  by passing any low priority interrupt to the high priority gating logic. If  $\text{IPEN} = 1$ , the output of gate *g1* is always zero, which has no effect on its destination OR gate.
5. Gate *g3* prevents a low priority interrupt from being generated if a high priority interrupt is pending.

7. One possible method is shown in Listing I.1.

**LISTING I.1** Problem 10.7.

---

```

interrupt_isr () {
    RB3 = 0;
    INTOIF = 0;
}
main () {
    TRISB3 = 0; RB3=0; // RB3 is output, init low, connected to
INT0
    INTEDG0 = 1; // INTO rising edge triggered
    //enable INTO interrupt
    INTOIF=0;INTOIE=1;IPEN=0;PEIE=1;GIE=1;
    while(1) {
        RB3 = 1;
    }
}
}SB

```

The RB3 port is configured as an output that is initially low and is assumed connected to the INT0 interrupt input, which is enabled for a rising edge interrupt. The while(1){} loop continually asserts RB3 high. The first rising edge on INT0 triggers the ISR, which resets RB3 low. When the foreground code is resumed, RB3 is immediately re-asserted high. The code ping-pongs between the while(1){} foreground loop and the ISR, creating a square wave on the RB3 output. The high pulse width of the square wave is equal to the ISR latency minus the instruction execution time it takes to clear the RB3 output to zero within the ISR.

9. If an interrupt occurs while the software delay loop is active, the execution time spent in the ISR is added to the software delay loop, making the delay longer than intended. This can be prevented by disabling all interrupts via the statement `GIE = 0` before entering the software delay loop, but then you run the risk of delaying the servicing of a pending interrupt too long.
11. The change of `IPEN = 0` disabled priorities, so all interrupts are treated as high priority interrupts. The falling edge of the button press causes both INT1F and INT2F flags to be set. The high priority interrupt service routine is executed since all interrupts now vector to the high priority ISR. Both flags are set, so statement A, then B is executed.
13. INT 1 is now rising edge triggered (and a high priority interrupt). The falling edge causes INT2IF to be set, so the low priority ISR is executed as INT2 is a low priority interrupt, which causes statement C to be executed. The ISR exits, and a pushbutton release causes a rising edge to occur—this sets INT1IF, causing the high priority ISR to execute, and statement B is executed.
15. Both interrupts are now low priority interrupts. The falling edge of the button press causes both INT1F and INT2F flags to be set and triggers the

execution of the low priority ISR. Both flags are set, so statement C, then D is executed.

17. The change  $INT2IE = 0$  disables the INT2 interrupt, but does not prevent the INT2IF flag from being set. The falling edge sets the INT1IF flag as well, triggering the high priority ISR. Because both flags are set, statement A is executed, then statement B. If statement B is to be executed only if the INT2 interrupt is enabled, change `if (INT2IF)` to `if (INT2IF && INT2IE)`.
19. All interrupts are disabled when  $GIE = 0$ , so the ISRs are not invoked and none of the target statements is executed.

21. Timer2 interrupt period =  $(PR2+1) * (4/FOSC) * PRE * POST$   
 $(1/2 \text{ kHz}) = (PR2+1) * (4/10 \text{ MHz}) * PRE * POST$   
 $PR2 = [(10\text{MHz}/4) / (2 \text{ KHZ} * PRE * POST)] - 1;$

Choose  $PRE = 4$ , other choices are possible. Then  $PR2 = 312.5/POST - 1$ .

If  $POST = 1$ ,  $PR2 = 311$  (too large,  $> 255$ ); if  $POST = 2$ ,  $PR2 \sim 155$ .

Use  $PRE = 4$ ,  $POST = 2$ ,  $PR2 = 155$ .

23. We need a ratio where:

$$((PR2_{\text{new}}+1) * PRE_{\text{new}} * POST_{\text{new}}) / ((PR2_{\text{old}}+1) * PRE_{\text{old}} * POST_{\text{old}}) = 10.$$

If we set  $PRE_{\text{new}} = 16$ , keep  $POST_{\text{new}} = POST_{\text{old}}$ , we have:

$$((PR2_{\text{new}}+1) * 16 * 5) / (64 * 4 * 5) = 10$$

$$PR2_{\text{new}}+1 = (10 * 64 * 4) / 16 = 160$$

$$PR2_{\text{new}} = 160 - 1 = 159.$$

Thus, the new values are  $POST_{\text{new}}=5$ ,  $PRE_{\text{new}} = 16$ ,  $PR2_{\text{new}} = 159$ .

25. Timer2 Interrupt period max =  $TOSC * 4 * PRE_{\text{max}} * POST_{\text{max}} * 8\text{-bit}$  rollover

$$= TOSC * 4 * 16 * 16 * 256 = 1/25\text{MHz} * 4 * 256 * 256 \sim 10.5 \text{ ms.}$$

27. A solution is shown only for state S0; the other states are handled similarly. This is not a very efficient solution in terms of code requirements, but it is clear.

```
case S0:
    if (last_state == S1) {
        if (count != max) count++;
    } else {
        if (count != min) count--;
    }
    break;
```

## I.11 CHAPTER 11

---

1. The clock is idle low, so  $CKP = 0$ . Data is stable on the rising clock edge, so  $CKE = 1$ .
3. The wiper cannot be read.

5. The MAX5408 supports 32 wiper positions.
7. If zero crossing is enabled, after a “change wiper position” command is executed the potentiometer will wait until the voltage across the potentiometer is zero before changing the wiper. This reduces clicks and pops in an audio application. The potentiometer has an internal timer that will timeout after 50 ms when waiting for the zero crossing condition.
9. For the AT25256A, the total bit capacity is 256K with an organization of 32K x 8.
11. The maximum SCK frequency is 20 MHz @ 5 V.
13. Bit 0 of the STATUS register is a “1” when a write is in progress; use a “read status register” command to return the contents of the STATUS register.
15. The following commands will set the Intersil X9221A wiper position to the value in wpos. This device is unusual in that the LSb of the I<sup>2</sup>C address is not a read/write# bit.

```
i2c_start();           //                A3 A2 A1 A0
i2c_put(0x5F);        // i2c address 0101 1 1 1 1 ,
i2c_put(0xA0);        // write command for altering wiper counter reg #0
i2c_put(wpos);        // write wiper position
i2c_stop();           // halt transaction
```

17. The maximum SCL clock frequency for the Intersil X9221A is 100 kHz.
19. The maximum SCL clock frequency for the Philips PCF8598C-2 is 100 kHz.
21. The device will send a NAK if addressed while a write is in progress.
23. For FOSC = 20 MHz, each instruction takes 4/20 MHz = 0.2 μs. An I<sup>2</sup>C bit time is 1/400 kHz = 2.5 μs. The bytes sent are the I<sup>2</sup>C address, MSB EEPROM address, LSB EEPROM address, and 64 data bytes for a total of 67 bytes. Total bit times are 67\*9 +start+stop = 605 bit times. Each byte is 9 bit times because of the acknowledge bit. The estimated time for data transfer is:  
 [instruction overhead] + [I<sup>2</sup>C transmission time]  
 [67 bytes \* 20 instructions \* instruction time] + [605 bits \* I<sup>2</sup>C bit time]  
 [67 \* 20 \* 0.2 μs] + [605 \* 2.5 μs] = 1780.5 μs.
25. SSPADD = [FOSC/(4\*400 kHz)] - 1 = [30E6/(4\*400e3)] - 1 = 17.75 = 18.

## I.12 CHAPTER 12

---

1. 211 = 2048, 212 = 4096 which is > 4000, so 12 bits are needed.
3. 80 minutes \* 60 = 4800 seconds, and 16 bits = 2 bytes.  
 Each track is 2 bytes \* 44.1E3 \* 4800 = 423,360,000 for each track.  
 Two tracks = 2 \* 423,360,000 = 846, 720, 000 bytes ~ 807 MB (1 MB = 220).

5. Each color has 28 combinations, so  $28 * 28 * 28 = 224 \sim 16.8$  Million (16,777,216).
7. Step 1: Guess is “1000”, so  $V_{ref} = 8/16 * 4 V = 2 V$ .  $V_{in}$  of  $1.8 V < 2 V$ , so  $D[3] = 0$ .  
 Step 2: Guess is “0100”, so  $V_{ref} = 4/16 * 4 V = 1 V$ .  $V_{in}$  of  $1.8 V > 1 V$ , so  $D[2] = 1$ .  
 Step 3: Guess is “0110”, so  $V_{ref} = 6/16 * 4 V = 1.5 V$ .  $V_{in}$  of  $1.8 V > 1.5 V$ , so  $D[1] = 1$ .  
 Step 4: Guess is “0111”, so  $V_{ref} = 7/16 * 4 V = 1.75 V$ .  $V_{in}$  of  $1.8 V > 1.75 V$ , so  $D[0] = 1$ .  
 The final 4-bit conversion returns “0111”.
9. The reference voltages for the seven comparators from the resistor string are  $7/8 * V_{ref}$ ,  $6/8 * V_{ref}$ ,  $5/8 * V_{ref}$ ,  $4/8 * V_{ref}$ ,  $3/8 * V_{ref}$ ,  $2/8 * V_{ref}$ , and  $1/8 * V_{ref}$ . The input voltage of  $2.7 V$  is between  $6/8 * V_{ref} = 3.0 V$  and  $5/8 * V_{ref} = 2.5 V$ , so the outputs of the comparators are “001111”.
11. The PIC18 has 10 bits of resolution, so:  
 $1 \text{ LSB} = 4.096 V / 2^{10} = 4.096 V / 1024 = 0.004 V = 4 \text{ mV}$ .  
 $\pm 0.1\% * 4.096 V = 0.004096 V$ , so  $4.096 \text{ mV} / 4 \text{ mV} * 100 \% = 102.4 \%$  of a LSB.  
 This means that only 9 bits of the PIC18 10-bit result should be used, as the voltage reference is not accurate enough for 10 bits.
13.  $0.449 V / 5 V * 256 \sim 23 = 0x17$ .  $3.91 V / 5 V * 256 \sim 200 = 0xC8$ .
15.  $0x7F = 127$ ;  $127/256 * 5 V = 2.48 V$ ;  $0x4B = 75$ ;  $75/256 * 5 V = 1.46 V$ ;  
 $0xCB = 203$ ;  $203/256 * 5 V = 3.96 V$ .
17. The principle advantage of a flash ADC over a successive approximation ADC is speed. The principle disadvantage is that the number of transistors (and hence silicon area) doubles for a flash ADC with each added bit of precision.
19. The PIC18 has 10-bits of resolution, so:  
 $1 \text{ LSB} = 5 V / 2^{10} = 5 V / 1024 \sim 0.00488 V = 4.88 \text{ mV} \sim 5 \text{ mV}$ .  
 This provides  $1/2$  degree °F resolution, if one degree °F is 10 mV.
21. We need  $120 * 4$  distinct codes; or 480 codes. (the “\*4” is needed because of the 0.25°F resolution for each degree). For 480 codes, we need 9 bits as  $2^9 = 512$ , which is greater than 480.
23. ADC clock choices:  
 $F_{OSC}/16 = 750 \text{ kHz}$ , is a period of  $1.3 \mu\text{s} < 1.6 \mu\text{s}$ .  
 $F_{OSC}/32 = 375 \text{ kHz}$ , is a period of  $2.7 \mu\text{s} > 1.6 \mu\text{s}$ ; use this choice as it is the fastest clock that meets the constraint.  
 Configuration code is:  

```
// AD Configuration, ADCON0 register
ADCS2 = 0;ADCS1 = 1; ADCS0 = 0; //FOSC/32
ADON = 1; //A/D turned on
```

```

ADFM = 0; // left justified
// AN3 is Vref+, VSS is Vref-, AN1, AN0 are analog inputs
PCFG3 = 0; PCFG2 = 0; PCFG1 = 0; PCFG0 = 1;

```

## I.13 CHAPTER 13

---

- Timer1 clock period is  $1/(40/4 \text{ MHz}) = 1.0e-7 \text{ seconds} = 0.1 \mu\text{s}$ .  
 1 Timer1 rollover =  $0.1 \mu\text{s} * 2 \text{ (prescale)} * 2^{16} \text{ (16-bit timer)} = 0.1 \mu\text{s} * 2 * 65536 = 13,107.2 \mu\text{s}$ .
- Timer0 can have a maximum prescale value of 256, and can operate as a 16-bit timer. So,  
 Timer0 Clock period is  $1/(20/4 \text{ MHz}) = 2.0e-7 \text{ seconds} = 0.2 \mu\text{s}$ .  
 1 Timer0 rollover =  $0.2 \mu\text{s} * 256 \text{ (prescale)} * 2^{16} \text{ (16-bit timer)} = 0.2 \mu\text{s} * 256 * 65536 = 3,355,443.2 \mu\text{s} \sim 3.4 \text{ s}$
- Timer1 Clock period is  $1/(30/4 \text{ MHz}) = 1.3e-7 \text{ seconds} = 0.13 \mu\text{s}$ .  
 1 Timer1 tic =  $0.13 \mu\text{s} * 8 \text{ (prescale)} = 1.07 \mu\text{s}$ ; so  $5 \text{ ms}/(1.07 \mu\text{s}) = 4687.5 \text{ tics}$ .  
 Thus, 5 ms contains approximately 4688 Timer1 tics.
- The statement  $\text{delta} = \text{delta} \ll 16$  is used to implement the operation  $\text{delta} * 2^{16}$ . The delta variable is type long, which means that it is 4 bytes. An alternate implementation is to simply move the two low bytes of delta to the two high bytes of delta, and to zero the two low bytes. This can be done via the following code, which could be a more efficient implementation if the compiler does not recognize this optimization:
 

```

char *ptr;
ptr = &delta; //get a pointer to delta
*(ptr+2) = *ptr; // assume little endian, copy LSByte to byte[2]
*(ptr+3) = *(ptr+1) // copy byte[1] to MSByte
*ptr = 0; // zero the LSByte
*(ptr+1) = 0; // zero the byte[1]

```
- See the comments in the code of Listing I.2 for details on this solution.

### LISTING I.2 Problem 13.9.

---

```

// Need a period 2 kHz Timer1 interrupt using compare mode.
// Assume a 25 MHz FOSC, use prescale = 1.
// Each Timer1 tic = 1/(25MHz/4) = 0.16 us
// 1/(2 kHz) = 500 us, so 500 us/0.16 us = 3125 Timer1 tics
// Pulswidth high = 0.25 * 3125 = 781 Timer1 tics
// Pulswidth low = 3125 - 781 = 2344 Timer1 tics

#define PWH 781 // high pulse width in timer tics
#define PWL 2344 // low pulse width in timer tics

```



```

volatile char out_state; // tracks if output is high or low
unsigned int match;

// uses Timer1, compare & toggle mode to generate sq wave
void interrupt timer_isr(void){
    if (CCP1IF) {
        if (!out_state) {
            out_state = 1; // next is high pulse width
            match = match + PWH; // add pulse width high tics
        } else {
            out_state = 0; // next is low pulse width
            match = match + PWL; // add pulse width low tics
        }
        //change match register, write MSB first to avoid false match
        CCP1H = match >> 8;
        CCP1L = match & 0xFF;
        CCP1IF = 0;
    }
}

main(void){
    // initialize timer 1
    T1CKPS1 = 0; T1CKPS0 = 0; // prescale by 1
    // use internal clock, 16 bit read mode
    T1OSCN = 0; TMR1CS = 0; T1SYNC = 0; T1RD16 = 1;
    bitclr(TRISC,2); // set RC2/CCP1 as output
    // initialize CCP1 for compare
    match = PWL; //low pulse width initially
    CCP1 = match;
    CCP1CON = 0x00; // Clear CCP1CON to set CCP1 low initially
    out_state = 0; //track output state
    CCP1CON = 0x02; // toggle mode
    // interrupt enable
    CCP1IF = 0; CCP1IE = 1;
    TMR1ON = 1; // enable timer 1
    IPEN = 0; PEIE = 1; GIE = 1;
    while(1); // interrupt does all work
        // of generating sqwave
}

```

11. See Figure I.9.

13. See Figure I.10.

15. Timer2 PWM period =  $(PR2+1) * (4/FOSC) * PRE$  (POST is NOT used for PWM period)

$(1/3 \text{ kHz}) = (PR2+1) * (4/20 \text{ MHz}) * PRE$

$PR2 = [(20 \text{ MHz}/4) / (3 \text{ KHZ} * PRE)] - 1;$

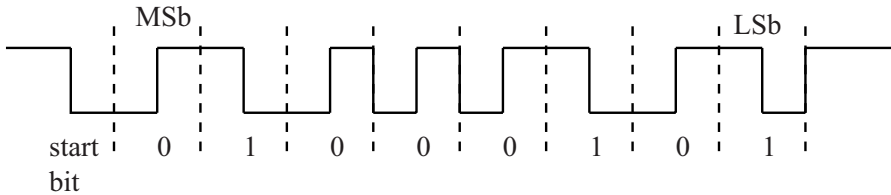
For PRE = 1, PR2 = 1666 (> 255, too large)

For PRE = 4, PR2 = 416 (> 255, too large)

For PRE = 16, PR2 = 103, so use PRE=16, PR2 = 103.

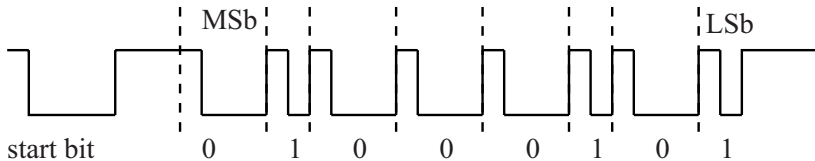
For 30% duty cycle, CCP1 =  $0.3 * (103 + 1) \sim 31.$

0x45 = 0b01000101 sent as biphas data



**FIGURE I.9** Biphas encoding for 0x45 (problem 13.11).

0x45 = 0b01000101 sent as space-width encoded data



**FIGURE I.10** Space-width encoding for 0x45 (problem 13.13).

17. See the comments in the code of Listing I.3 for details on this solution.

**LISTING I.3** Problem 13.17.

---

```

char print_flag;
int edges, tmr1_oflow;

interrupt isr() {
    if (TMR1IF) {
        TMR1IF = 0; //clear iflag
        tmr1_oflow++; // track overflows to detect idle condition
        if (tmr1_oflow > 50) {
            // input is idle, so set print_flag semaphore, finished
            print_flag = 1;
            TMR1IE = 0; CCP1IE = 0; //disable interrupts
        }
    }
    if (CCP1IF) { // edge found!
        CCP1IF = 0; // clear iflag
        edges++; //found an edge, this counter tracks number of edges
        tmr1_oflow = 0; //reset
    }
} //end isr

main (){

```

```

// assume TMR1, serial port configured, code not shown
CCP1CON = 0x04; // look for falling edges
while(1) {
    // clear flags
    print_flag = 0; edges = 0; tmr1_oflow = 0;
    printf("Hit any key."); pcrflf();
    getch(); //wait for use to hit key
    // enable interrupts
    TMR1IF = 0; TMR1IE = 1; CCP1IF=0; CCP1IE = 1;
    IPEN = 0; PIE1 = 1; GIE = 1;
    while(!print_flag); // wait for idle
    printf("Edge count is %d",edges);
    pcrflf();
} // end while
} // end main

```

19. See the comments in the code of Listing I.4 for details on this solution. The Timer3 interrupt and compare mode is used to generate a periodic 10 kHz interrupt. The ISR does all of the work of generating the RB3, RB4 PWM signals. The variable `dc_rb3_cnt` (`dc_rb4_cnt`) keeps track of the period of the RB3 (RB4) PWM signal; when this reaches a value of 10 the RB3 (RB4) output is set to "1". The variable `dc_rb3` (`dc_rb4`) sets the high pulse width (and thus the duty cycle) of the PWM signals; when the `dc_rb3_cnt3` (`dc_rb4_cnt`) variable is equal to `dc_rb3` (`dc_rb4_cnt`) variable, the RB3 (RB4) output is reset to "0". If the duty cycle variable `dc_rb3` (`dc_rb4`) is 0, the RB3 (RB4) output is never set high.

---

**LISTING I.4** Problem 13.19.

```

// Need a period 10 kHz Timer3 interrupt using compare mode.
// Assume a 20 MHz FOSC, use prescale = 1.
// Each Timer3 tic = 1/(20MHz/4) = 0.2 us
// 1/10 kHz = 100 us, so 100 us/0.2 us = 500 Timer3 tics
#define INTPERIOD 500

int dc_rb3, dc_rb4; // sets the duty cycle
char dc_rb3_cnt,dc_rb4_cnt; //tracks the period

unsigned int match;

void interrupt isr(void){
    if (CCP2IF) {
        // change compare register to next match
        match = match + INTPERIOD;
        CCPR2H = match >> 8;
        CCPR2L = match & 0xFF;
        CCP2IF = 0;
        //update the RB3, RB4 outputs
        if (dc_rb3 != 0) {
            dc_rb3_cnt++;

```

```

        // check if high pulse width is finished
        if (dc_rb3_cnt == dc_rb3) RB3 = 0;
        if (dc_rb3_cnt == 10) { // period is finished
            dc_rb3_cnt = 0; RB3 = 1;
        }
    } //end if(dc_rb3...
    if (dc_rb4 != 0) {
        dc_rb4_cnt++;
        // check if high pulse width is finished
        if (dc_rb4_cnt == dc_rb4) RB4 = 0;
        if (dc_rb4_cnt == 10) { // period is finished
            dc_rb4_cnt = 0; RB4 = 0;
        }
    } // end if(dc_rb4...
} // end if(CCP2IF)
} //end isr    main(void){
    serial_init(95,1); // 19200 in HSPLL mode, crystal = 7.3728 MHz
    T3CKPS1 = 0; T3CKPS0 = 0; // initialize timer 3, prescale by 1
    // disable osc, use internal clock, use 16-bit update mode
    T10SCEN = 0; TMR3CS = 0; T3RD16 = 1;
    T3CCP2 = 1; // use Timer3 as source for compare registers
    // RB4, RB3 are outputs
    TRISB4 = 0; TRISB3 = 0;
    RB4 = 0; RB3 = 0;
    // initialize CCPR2 for compare
    match = INTPERIOD;
    CCPR2 = match;
    CCP2CON = 0x0A; // compare mode, CCP2IF interrupt only
    // prompt user for duty cycle
    printf("Enter RB3 duty cycle (0-10)");pchr1f();
    scanf("%d",&dc_rb3);
    printf("Enter RB4 duty cycle (0-10)");pchr1f();
    scanf("%d",&dc_rb4);

    if (dc_rb3) RB3 = 1; // initialize high if non-zero duty cycle
    if (dc_rb4) RB4 = 1; // initialize high if non-zero duty cycle
    // enable interrupts
    CCP2IF = 0; CCP2IE = 1; IPEN = 0; PEIE = 1; GIE = 1;
    TMR3ON = 1; // enable timer 3
    while(1); // interrupt does all work of generating PWM signals
}

```

*This page intentionally left blank*



## About the CD-ROM

This book's companion CD-ROM has the following directories:

**bootldr/:** Contains the Jolt/Colt serial bootloader self-installing executables (Appendix F, "The Jolt/Colt Serial Bootloaders").

**hitech/:** Contains the self-installing HI-TECH PICC-18 C Compiler for PIC18F242 (Appendix C, "HI-TECH PICC-18 C Compiler Demo for the PIC18F242").

**figures/chap??:** Contains the book figures as *.png* files separated by chapters.

**code/chap??:** Contains assembly language and C code source files from book examples separated by chapters. Each file has the figure or listing number that features this example.

**code/common:** Contains the `#include` C code files needed by the book examples.

**code/labs:** Contains all of the files referenced by the lab exercises in Appendix I.

### **J.1 GENERAL SYSTEM REQUIREMENTS**

---

Any personal computer capable of running Windows 2000/XP can run the software included on the companion CD-ROM.

1. Colt Serial Bootloader (V 0.5, Martin Dubuc author, <http://mdubuc.freeshell.org/Colt/>). This is a serial bootloader for the PICmicro family that includes a PC client written in Visual C++ and PICmicro firmware (hex file pro-

vided). Once a PICmicro is programmed with the included firmware, the PIC can be programmed by the PC client through the PC serial port, or with a USB-to-serial port adapter. System requirements are 460 KB of free disk space, Windows XP, and either a serial port or a USB-to-serial adapter (see Appendix F for installation details).

2. Jolt Serial Bootloader (V 1.1, Martin Dubuc author, [http:// mdubuc.freeshell.org/Jolt/](http://mdubuc.freeshell.org/Jolt/)). This is a serial bootloader for the PICmicro family that includes a PC client written in Java and PICmicro firmware (hex file provided). Once a PICmicro is programmed with the included firmware, the PIC can be programmed by the PC client through the PC serial port, or with a USB-to-serial port adapter. Jolt itself requires 300 KB of free disk space, Windows 2000/XP, and either a serial port or a USB-to-serial adapter. Jolt also requires the installation of the Java Runtime Environment (JRE Version 5.0 or later) from Sun Microsystems ([http:// java.sun.com](http://java.sun.com)), which requires 100 MB of free disk space, and Windows 2000/XP (see Appendix F for installation details).

Both Jolt and Colt are provided to give the user flexibility in choosing a preferred bootloader environment. The differences between Jolt and Colt are explained in Appendix F.

3. HI-TECH PICC-18 C Compiler (HI-TECH PICC-18 Compiler Special Demo Version 8.35 PL2, HI-TECH Software, [www.htsoft.com](http://www.htsoft.com)). System requirements are 13 MB of free disk space, and Windows 2000/XP. This is a C compiler demo from HI-TECH software that has all the capabilities of the full featured compiler except for the limitations noted in Appendix C. See Appendix C for installation details.

# Index

- A**
- access banks, 57
  - accessing table data from program memory, 160–162
  - ADCs. *See* analog-to-digital converters
  - adder logic circuit, 18–20
  - addition
    - BCD arithmetic, 197–199
    - binary and hex, 6–7
    - floating-point, 195–196
    - extended precision operations, 107–108
  - addresses
    - return, subroutines, 136
    - target, 63
  - addressing
    - bus enumeration, 526
    - indirect, 138
    - modes, 57
  - addwfc instruction, 107–108
  - algebra, Boolean, 10–13
  - Algorithmic State Machines (ASMs)
  - algorithms
    - Booth's algorithm, 181–182
    - division, 183–188
    - multiplication, 176–178
  - Alternating Current (AC), 613
  - ALUs (arithmetic/logic units)
    - adder/subtractor component, 120–121
    - described, 54
    - operations implemented in processor data path, 78
  - American Standard Code for Information Interchange. *See* ASCII
  - analog-to-digital conversion
    - generally, 373–382
    - PIC18Fxx2 converter, 382–391
  - analog-to-digital converters (ADCs)
    - counter ramp ADC, 376–377
    - described, 32, 406
    - flash ADC, 380–382
    - PIC18Fxx2, 382–391
    - successive approximation ADC, 377–380
  - AND function, Boolean logic, 10–13
  - AND operator
    - bitwise operations, 78–81
    - logical, 86
  - answers to review problems, 625–653
  - application notes, 224
  - application-specific integrated circuit (ASIC), 529
  - applications
    - audio record/playback, ISR code, 461–463
    - autonomous robot, 494–504
      - square root (fig.), 289
    - arbitration
      - bus, in I2C, 516–518
      - mechanisms, 346
    - architecture
      - ADC, 376
      - instruction set, PIC16 vs. PIC18 (table), 515
      - PIC18Fxx2 (fig.), 54
      - PIC18Fxx2 instruction set, 537–548
      - R-2R resistor ladder flash DAC (fig.), 396
      - resistor string flash DAC (fig.), 394
    - arithmetic
      - See also* mathematics
      - basic, and control instructions, 61–64
      - Binary Coded Decimal (BCD), 197–199, 204
      - binary to hex, 6–10
      - fixed-point and saturating, 188–192
      - floating-point, 195–196
      - operators in C (table), 78
      - pointer, 147
      - vs. logic, right shift (fig.), 122
    - arithmetic/logic units. *See* ALUs
    - arithmetic operations
      - See also* operations
      - unsigned vs. signed (table), 121
    - array multipliers, 179, 204
    - arrays
      - in assembly language, 152–156
      - and pointers in C language, 146–151
    - ASCII code
      - data conversion, 199–204
      - described, 27–28
    - ASIC (application-specific integrated circuit), 529
    - assembler directives, 68
    - assembler programs, 67
    - assembly described, 42
    - assembly language
      - advanced, for higher math, 175–204
      - arrays and pointers in, 152–156
      - implementing subroutines in, 141–145
      - PIC18 program, 64–72
      - and stored program machines, 39–43
      - unoptimized, 210
      - vs. high-level languages, 208–210
    - assignment of extended precision variables, 106
    - asynchronous
      - communication, RS232 standard, 270–273
      - DRAM, 509
      - input and D Flip-Flop (DFF), 23
      - serial data input, 287–291
      - serial IO, 259–263
    - audio record/playback system
      - design and implementation, 456–465
      - implementing, 594–595
    - autonomous robot, designing and implementing, 494–504
- B**
- bandwidth
    - audio record/playback system, 457–459
    - communication channel, 254
    - increasing IO channel, 254
  - Bank Select Register (BSR), 56, 59, 70
  - banks, memory, 55
  - Basic Stamp processors, 530
  - baud rate, 261, 267
  - BCD arithmetic, 197–199
  - binary
    - to ASCII conversion, 199–204
    - Binary Coded Decimal (BCD)
      - arithmetic, 197–199
      - data, encoding, 2–5
      - and hex addition, 6–7
      - subtraction, 8–10
      - values, shift operations, 9–10
    - Binary Coded Decimal (BCD) arithmetic, 197–199, 204
  - biphase encoding, 434–436
  - bit definitions
    - CCP1CON register (table), 423
    - Timer1 control register T1CON (table), 420
  - bit operations, 78–83
  - bit stuffing, destuffing, 522–523
  - bit variables and special function registers, 212–214
  - BIT\_CAPTURE, BITCHANGE states, 439
  - bits
    - configuration, 221–222
    - described, 2
    - least, most significant bit (LSb, MSb), 3, 260
    - parity, 260
    - status, 264
  - bitwise complements, 11
  - bitwise logical operations, 78–83, 106–107
  - block diagrams
    - finite state machines (fig.), 35
    - PIC18Fxx2 (fig.), 538
    - watchdog timer (fig.), 226
  - BLOCKAGE\_THRESHOLD, 500–501
  - bnz instruction, 110–111



- book, this
  - for hobbyists, xxi
  - CD-ROM. See CD-ROM
  - using in academic environment, xvii–xxi
- Boolean algebra, 10–13
- Booth's algorithm, 181–182
- branches
  - described, 40
  - signed, 125
- branch instructions
  - described, 87–88
  - encoding, 127–128
- BSR (Bank Select Register), 56, 59, 70, 83, 145
- buffer overruns, 291
- buffers
  - CMOS (fig.), 15
  - double buffering for interrupt-driven writes, 364–366
  - first-in, first-out (FIFO), 265
  - trace, 320
  - two-transistors, 14–15
- building blocks
  - adder/subtractor, 120–121
  - combinational, 17–20
  - sequential, 25–27
- bulk USB transactions, 525
- bus arbitration in I2C bus, 516–518
- busses
  - Controller Area Network (CAN), 518–523
  - external memory, 511
  - hardware design for number sequencing program, 43
  - I2C, 345–348
  - Universal Serial Bus (USB), 523–527
- C**
- C programming language
  - accessing table data from program memory, 160–162
  - arrays and pointers, 146–151
  - bitwise complements, 11
  - C++ programmer notes, 559–561
  - code operations on 16-bit registers, 413–415
  - conditional tests in, 85–94
  - formatted IO, 557–559
  - functions, translating to subroutine in PIC18, 142
  - machine code listing (listing), 69–70, 71
  - multiplication, implementing, 176–183
  - number sequencing task, 39–40
  - programs in program, compilation, 64–66
  - shift operations, 9–10
  - switch statements, 89–90
- C programs
  - data transfer, arithmetic operations, 64–66
  - machine code listing for simple (listing), 69
  - machine code listing, variables in bank 1 (listing), 71
  - MPLAB assembler source code for simple (listing), 67–68
  - and PICC-18 C compiler, 553
- cabling, RS232 standard, 270
- cache controllers, 529
- cache memory, 508
- call graphs, 214
- call instruction, machine code format for (fig.), 141
- call/return instructions, 136–141
- caller, callee, subroutines, 134
- CAN (Controller Area Network), 259, 518–523
- capacitors
  - decoupling, 218
  - described, 618–619
  - pF included in parts kit, 598–599
- capstone projects
  - audio record/playback system, 456–465
  - home monitoring system, 466–468
  - suggested project modifications, 505–506
- capture mode
  - pulse width measurement using, 422–428
  - using for frequency measurement, 447–450
  - using for infrared decoding, 433–441
- Carrier Sense Multiple Access/Collision Detection (CSMA/CD) arbitration (fig.), 516–518
- Carry (C) flag, 84–85, 112, 119–120, 123–125
- case blocks in switch statements, 90–91
- CD-ROM
  - about, 655–656
  - HI-TECH software PICC-18 C compiler demo, 553–556
  - Microchip MPLAB integrated design environment, 549
- central processing unit (CPU), 52
- channels
  - asynchronous data rate, 261
  - parallel port IO, basics, 254–259
- char data type, unsigned, 65, 104
- character data, encoding, 27–29
- checklist, hardware debugging, 595–597
- chip select signals, 509
- chips
  - described, 52
  - transceiver, 519
- chipsets described, 53
- circuits
  - building with combinational blocks, 17–20
  - integrated, 52
  - introduction to, 613–619
  - sequential logic, 21–23
- circular buffers, 291
- clearing bits, using BCF (listing), 81
- clock cycles, SPM vs. FMS (table), 46
- clock signal
  - clock generation for PIC18, 219
  - and instruction execution, 73
  - in sequential logic circuit, 21–23
  - synchronous serial IO, 257–259
- clock source for Timer1/Timer3, 420
- CMOS (Complementary Metal Oxide Semiconductor)
  - and DeMorgan's Law, 12
  - logic gate implementations, 13–16
  - power consumption, 227–228
  - code optimization, 144
  - Colt serial bootloader, 601–611, 655–656
- command sets, DS1621 Digital Thermometer (table), 471
- common-mode noise rejection, 519–520
- compare mode, Timer1/Timer3, 428–432
- comparisons
  - unsigned literal, 93–94
  - unsigned, using cpfseq, cpfsgt, cpfslt, 92–93
  - unsigned vs. signed (fig.), 124
- Complementary Metal Oxide Semiconductor. See CMOS
- compilers
  - described, 41
  - dynamic vs. static allocation, 144
  - HI-TECH software PICC-18 C compiler, 249, 553–556
  - PICC-18 runtime code, 214–215
- compiling programs generally, 65
- complements
  - in BCD arithmetic, 197–199
  - bitwise, 11
  - logical vs. bitwise operators, 86
  - and output, input, 10
  - and shift operations, 9–10
  - in signed number representation, 114–118
- computers
  - and number sequencing computers, 47
  - and stored program machines, 39
- conditional inputs in ASM, 34
- conditional tests
  - in C, 85–87
  - equality, inequality, 89–90
  - zero, nonzero, 87–88
- conductors, 613
- configuration bits
  - settings, 221–222
  - in USART hardware system, 264
- constants (literals), 59
- control instructions, basic arithmetic and, 61–64
- control registers/bits for asynchronous configuration (table), 268
- Controller Area Network (CAN)
  - bus described, 518–523
  - serial transmission standards, synchronization, 259
- controllers
  - described, 32
  - implementing as finite state machine, 34–39
  - implementing as stored program machine, 39–47
- conversion
  - See also data conversion
  - analog-to-digital converters (ADCs), 32
  - ASCII data, 199–204
  - binary-to-decimal, hex-to-decimal, 5–6
  - digital-to-analog, 391–406
  - converting
  - 8-bit unsigned to 16-bit unsigned, 118
  - binary to ASCII-decimal representation, 200–201
  - binary to ASCII-hex format, 199–200
  - fixed-point binary numbers to decimals, 189–190
  - unsigned decimals to fixed-point representation, 188–189
  - upper- to lower-case characters, 156–157
- counter ramp ADC, 376–377
- counters
  - described, using, 26

- loop, 96
- cpfsq, cpfsqt, cpfst instructions, 92–93
- CPU (central processing unit)
  - described, 52
  - and real-time operating systems, 531–532
- CSMA/CD (Carrier Sense Multiple Access/ Collision Detection) arbitration (fig.), 516–518
- current
  - measuring, 598
  - voltage and resistance, 613
- current mode signaling, 255
- Cypress Semiconductor processors, 530
- Cypress Semiconductor SRAM, 509–512

**D**

- D Flip-Flop (DFF), 23–24
- DACs. See digital-to-analog converters
- Dallas Semiconductor DS32KHz, 421
- data
  - arrays of, 146
  - binary, 2–5
  - collisions, 517
  - conversion. See data conversion
  - encoding character, 27–29
  - frames, CAN (fig.), 521
  - storage. See data storage
  - transfer. See data transfer
- data conversion
  - analog-to-digital conversion, 373–382
  - basics of, 372–373
  - digital-to-analog conversion, 391–406
  - experimenting with, 585–587
  - PIC18Fxx2 analog-to-digital converter, 382–391
- Data EEPROM memory, 474–478
- data memory location interchangeable, 57
- data memory organization, PIC18Fxx2, 55–58
- data sheet for PIC18Fxx2, 223–225
- data stacks, 162, 169
- data storage
  - audio, 457
  - nonvolatile storage on PIC18Fxx2, 475–482
  - serial EEPROM, 338
- Data Terminal Equipment (DTE), 270
- data transfer
  - in bulk transaction (fig.), 527
  - data packets, 525
  - infrared (IR) transmit and receive, 433–441
  - I2C bus (fig.), 347
  - parallel and serial IO, 277–278
  - PIC18Fxx2, 55–58
  - serial, synchronous, 254
  - streaming data, 364–366
- data types
  - C extended precision integers (table), 104
  - signed, unsigned, 65
- datasheet for PIC18Fxx2, 210
- debouncing
  - pushbutton inputs, 492
  - switch, using timers, 307–309
- debugger, MPLAB SIM, 551–552
- debugging
  - hardware problems, 595–597

- ISRs (interrupt service routines), 319–321
  - serial ports, 275–277
- Decimal Carry (DC) flag, 84
- decimal numbers, 3–4
- declarations, variable, 559–561
- decoding infrared, 433–441
- decoupling capacitors, 218
- decrement instructions, 62
- defensive programming, 350
- DelayMsKill () function, 301–302
- DeMorgan's Law, 12–13, 16
- designing
  - audio record/playback system, 456–459
  - autonomous robot, 494–504
  - finite state machines (FSMs), 34–39
  - hardware for number sequencing
    - program, 43–46
- DFF (D Flip-Flop), 23–24
- DFF toggle frequency, 304
- differential signaling, 519
- digital
  - logic, 2
  - multimeter (DMM), 598
  - processing generally, 32–33
  - potentiometer, 334–337
- digital-to-analog conversion, 391–400
- digital-to-analog converters (DACs)
  - described, 32–33, 406
  - flash DACs, 393
  - MAX518, 400–406
  - a PWM, 445–447
  - R-2R resistor ladder flash DAC, 395–400
  - resistor string flash DACs, 393–395
- diode circuits, 617–618
- Direct Current (DC), 613
- directives, assembler, 68
- disabled interrupts, 283
- disassembly described, 42
- display, LCD. See liquid crystal display
- division, implementing in PIC18 assembly
  - language, 183–188
- DMM (digital multimeter), 598
- do-while {} loop structure, 95
- do\_config () function, 487–488
- do\_ircap () function, 437–439
- do\_settime () function, 488–490
- double buffering for interrupt-driven
  - writes, 364–366
- DOUT, output values referenced to states
  - (table), 37
- drain, MOS transistors, 13
- DRAM (Dynamic Random Access Memory)
  - DS1621 Digital Thermometer, 469–474
  - dual-inline package (DIP), 216
  - duplex communications, 255
  - dynamic memory allocation, 142–144, 169
  - Dynamic Random Access Memory (DRAM), 508–510

**E**

- EECON1 registers (table), 476
- EEPROM (electrically erasable programmable ROM), 55, 337–344, 356
- EIA-RS232 standard, 270–273
- electrically erasable programmable ROM (EEPROM), 55
- embedded systems, 209, 494

- encoding
  - binary data, 2–5
  - branch instructions, 127–128
  - character data, 27–29
  - data strobe, 258
  - IEEE 754 floating-point, 192–195
  - mechanical shaft rotation, 309–315
  - opcode, 40–41
  - signed magnitude, 114–115
  - space-width, biphasic, 434–436
- epulse () function, 246
- equality, inequality conditional tests
  - 8-bit, 89–90
  - 16-bit, 110–111
- events
  - computing elapsed timer tics between
    - two, 424–427
  - waveform, 22
- exercises
  - review questions. See review questions
  - suggested laboratory, 563–600
- extended precision integers, operations, 104–114
- external memory interfacing, 508–513

**F**

- falling edge and clock signal, 21
- fast call/return mode select bit, 141
- fetch/execute actions and stored program machines, 39
- Fibonacci numbers, 166
- field programmable gate arrays (FPGAs), 529
- FIFO (first-in, first-out) buffers, 265, 291
- file registers
  - and compiling programs, 65
  - described, 57
- filters, reconstruction, 399
- finite state machines (FSMs), 33–39, 46
- first-in, first-out buffers (FIFO), 265, 289
- fixed-point arithmetic, 188–192
- flags
  - read busy, 245
  - setting and clearing, 84
- flash ADC, 380–382
- flash DACs, 393
- Flash program memory read/write, 478–482
- flash programmable memory, 55, 474
- flashing LED, implementing, 222–223
- floating-point (FP) representation, 192–196
- floc operand, 57, 62
- FOSC (clock frequency), 73
- FPGAs (field programmable gate arrays), 529
- frame pointers (FPs), 163
- frames, data transmission, 521
- FREE bit, 478, 480
- free-running code, 301
- FreeRTOS, 533
- frequency
  - clock, 21
  - common units (table), 22
  - using capture mode for measurement, 447–450
- FSMs (finite state machines), 33–39, 46
- FSR0, FSR1, FSR2 registers, 152
- function set command, 246
- functions
  - combinational logic, 10–13
  - in USB networks, 523

- G**
- gate, MOS transistors, 13
  - general-purpose register (GPS) in data memory organization, 56
  - getche () function, 289
  - GO/DONE# bit, 387
  - goto
    - described, 40
    - instruction machine code format (fig.), 63
    - statements, using in assembly language, 69
  - GPS (general-purpose register) in data memory organization, 56
  - greater than (>), greater-than-or-equal (>=)
    - conditional tests, 16-bit, 110–111
    - conditional tests, 8-bit, 90–92
    - operations on signed data, 123–125
  - ground, system, in CMOS transistors, 14
- H**
- half-duplex communications, 255
  - handshakes, 525
  - hardware
    - debugging checklist, 595–597
    - design for stored program machines, 43–46
    - PIC18 subsystems, 264–270
    - PIC18F242 startup, experiment, 575–578
  - Harvard architecture, 54
  - hexadecimal
    - ASCII to binary conversion, 201–202
    - and binary arithmetic, 6–7
    - numbers, 3
    - subtraction, 8–10
  - HI-TECH software PICC-18 C compiler, 249, 553–556, 656
  - high-level languages vs. assembly languages, 208–210, 249
  - high pulse width, 21
  - home monitoring system design, 466–494
  - hubs in USB networks, 523
  - HyperTerminal program (Windows), 271, 273
  - hysteresis, 234
- I**
- IBM processors, little- and big-endian processors, 105
  - IEEE 754 floating-point encoding, 192–195
  - if-else statements in C (fig.), 87
  - if {} statements
    - equality, inequality conditional tests, 89–90
    - in while {} loops, 94–97
  - implementing
    - audio record/playback system, 459–465
    - autonomous robot, 494–504
    - finite state machines (FSMs), 34–39
    - flashing LED, 222–223
    - home monitoring system, 483–494
    - multiplication operations, 176–183
    - subroutines in assembly language, 141–145
  - include file paths, 554
  - INCLUDE statements as assembler directives, 68
  - increment instructions, 61–62
  - incremental encoders, 310
  - incrementer, building, 18–19
  - indirect addressing, 138
  - inequality, equality conditional tests
    - 8-bit, 89–90
    - 16-bit, 110–111
  - infrared (IR)
    - decoding, 433–441, 593–594
    - mappings for robot control (table), 503
  - inline assemblies, 230
  - inputs, conditional and unconditional, 34
  - installing
    - Java Communications API, 605–607
    - Jolt/Colt firmware, 605–607
  - instruction execution, and the clock, 73
  - instruction classes
    - arithmetic, control, 61–64
    - movf, 57–58
  - Instruction Pointer (IP), 44
  - Instruction Set Architecture (ISA) and PIC18, 53
  - instruction sets
    - design, and assembly language, 39–43
    - PIC18Fxx2, 542–543
  - instruction word, 53
  - instructions
    - affects on flags, 84
    - branch, 87–88, 127–128
  - int data type, unsigned ranges, C (table), 104
  - integers
    - extended precision, 104–105
    - signed, 8
  - integrated circuits, invention of, 52
  - Integrated Design Environment (IDE), 67
  - Intel
    - early chips, 52
    - x86 processors, little- and big-endian processors, 105
  - Inter I<sup>2</sup>C bus
    - bus arbitration in, 516–518
    - C function for initialing I2C master mode, 354–355
    - described, 345–348
    - experimenting with, 583–584
    - on the PIC18Fxx2, 348–355
  - interfaces
    - 25LC640 serial EEPROM, 337–344
    - numeric keypad, 315–319
    - PIC18 to 24LC515 I2C, 356–364
    - PIC18 to LCD (fig.), 244
    - rotary encoder, 309–315
    - SPI (Serial Peripheral Interface), 331–344
  - interrupt-driven
    - asynchronous data transmit, 294–296
    - asynchronous serial data input, 287–291
    - IO, 282–283
    - IO, state machine programming for, 299–303
    - IO, using software FIFO with, 291–294
    - writes, double buffering, 364–366
  - interrupt-on-change feature, 296
  - interrupt service routine (ISR), 282, 317
  - interrupt vectors, 69
  - interrupts
    - described, interrupt-driven IO, 282–283
    - experimenting with, 580–583
    - PIC18, details about, 284–287
    - summary on, 321–322
  - inverters, 14
  - IO
    - channel basics, 254–259
    - polled IO, 282
    - serial. *See* serial IO
  - isochronous USB transactions, 525
  - ISRs (interrupt service routines)
    - described, 282
    - writing, debugging, 319–321
- J**
- Java Communications API, installing, 605–607
  - Java Runtime Environment (JRE), 605
  - JK Flip-Flop (JKFF), 24
  - Jolt/Colt firmware
    - programming, installing, running, 601–611
    - system requirements, 655–656
  - Jolt serial bootloader, troubleshooting, 597
  - jump described, 40
- K**
- keyboard scans, 317
  - Kilby, Jack, 52
  - KxN memory device, 19–20
- L**
- laboratory exercises
    - data conversion, 585–587
    - debugging hardware checklist, 597
    - extended precision and signed operations, 572–573
    - hardware setup, 575–578
    - instrumentation and prototyping hints, 598–600
    - interrupts, 580–583
    - lab setup, 563–566
    - LED/switch IO, asynchronous serial IO, 578–580
    - PIC18Fxx2 introduction, 567–569
    - pointers and subroutines, 573–575
    - stored program machine experiment, 566–567
    - time measurement, IR waveform decoding, 593–594
    - timers, waveform generation, 587–592
    - unsigned 8-bit operations, 569–571
  - languages, programming. *See* programming languages
  - last-in, first-out (LIFO) data structures, 137
  - lcase () function, 156–157
  - LCD. *See* liquid crystal display
  - lcd\_write () function, 246
  - least significant bit (LSb), 3, 260
  - least significant byte (LSB), 104
  - least significant digit (LSD), 3
  - light emitting diode (LED)
    - described, 617
    - included in parts kit, 598
    - interrupt-driven, 299–303
    - program for PIC18F242 startup, 220–223
    - schematic for, 217–218
    - switch IO, 237–242, 578–580
  - liquid crystal display (LCD)

- described, 242
- interfacing with module, 243–248
- screen formats (fig.), 468
- literals (constants), 59
- little- and big-endian processors, 105
- LM340T5 voltage regulator, 217
- LM386 audio amplifier, 459–460
- logic
  - binary, 2
  - circuits, 17–20
  - combinational functions, 10–13
  - sequential, 21–24
  - wired, 235
- logic elements, D Flip-Flop (DFF), 23–24
- logic gates, 11, 13–16
- logical
  - operators in C (table), 78
  - vs. arithmetic, right shift (fig.), 122
- long data type, unsigned ranges, C (table), 104
- long vs. float operations, 195–196
- looping
  - long vs. float operations, 195–196
  - while {} loop structures, 94–97
- loops
  - loop counters, 96
  - phase locked loop (PLL), 258–259
  - polling, 245
  - software delay, 222
- low pulse width, 21
- LSb (least significant bit), 3
- LSB (least significant byte), 104

**M**

- machine code
  - described, 42
  - for simple C program (listing), 69–70
- magnetic encoders, 310
- magnitude, signed, encoding, 114–115
- main () function
  - assembly implementation for Fibonacci C code (fig.), 167
  - C program entry point, 65
  - calling vlshtft () C function, 145
  - home monitoring system (figs.), 486–487
  - home monitoring system flowchart (fig.), 484
  - robot application (fig.), 498
- map files, 214
- masked interrupts, 283
- mathematics
  - ASCII data conversion, 199–204
  - BCD arithmetic, 197–199
  - fixed-point and saturating arithmetic, 188–192
  - floating-point number representation, 192–196
  - implementing integer division in PIC18 assembly language, 183–188
  - implementing integer multiplication in PIC18 assembly language, 175–183
- MAX518 dual 8-bit DAC, 400–406
- Maxim Integrated Products MAX518, 400–406
- MCP41xxx digital potentiometer, 334–337
- measuring
  - frequency using capture mode, 447–450
  - pulse width, 415–419
  - voltage, resistance, current, 598

- memory
  - 18Fxx2 sizes (table), 55
  - accessing table data from program, 160–162
  - Data EEPROM, 475–478
  - dynamic allocation, 142, 162–169
  - EEPROM (electrically erasable programmable ROM), 55
  - external, interfacing, 508–513
  - flash programmable, 55, 474
  - FSM vs. stored program machine, 39
  - KxN device, 19–20
  - PIC18Fxx2, 54–55
  - read-only (ROM), 55
  - SRAM and DRAM, 508–510
  - subroutine assignments, 146–151
  - virtual, 529
- memory management units (MMUs), 529
- Microchip MPASM tool suite, 554
- Microchip MPLAB integrated design environment, 549–552
- Microchip Technology
  - IDE environment, 67
  - PIC microcontroller families, 513–516
  - PIC18Fxx2 data sheet, 210
- microcontrollers
  - high-level languages vs. assembly languages, 208–210
  - introduction to, 52–53
  - non-PIC, 527–531
  - PIC family, 513–516
  - PIC18FXX2, 53–55
- microprocessors
  - component datasheets, 223–225
  - high-level languages vs. assembly languages, 208–210
  - introduction to, 52–53
  - numeric keypad interface, 315–319
  - registers, 25
- MMUs (memory management units), 529
- monitoring system, home, 466–494
- monotonicity, 395
- Moore's Law, 372
- MOS transistors and CMOS transistors, 13
- most significant bit (MsB), 3, 260
- most significant byte (MSB), 104
- most significant digit (MSD), 3
- Motorola
  - little- and big-endian processors, 105
  - Serial Peripheral Interface, 331
  - 68XXX family of processors, 529
- movf instruction, 57–58
- movff instruction, 60–61
- movlb instruction, 59
- movlw instruction (listing), 63
- movwf, movff instructions, 60–61
- MPASM tool suite, 554
- MPLAB assembler
  - machine code listing (listing), 69
  - source code for C language example (listing), 67–68
- MSb (most significant bit), 3
- MSB (most significant byte), 104
- multi-master capability, 346, 516
- multiplexers
  - described, 17–18
  - time domain multiplexing (TDM), 518
- multiplication, implementing in PIC18 assembly language, 175–183
- mux described, 17–18

**N**

- N-bit registers, 25
- N (negative) flag, 119–120, 123–125
- NAK (not-acknowledge), 347
- NAND function, Boolean logic, 10–13
- negation operator, logical, 86
- negative (N) flag, 119–120
- nested subroutines, 139
- networks
  - of logic gates, 11
  - USB, topology, 523–525
- Non Return to Zero Invert (NRZI) data encoding, 525
- non-return-to-zero (NRZ)
  - asynchronous transmission, 520
  - format, 258
- nonvolatile
  - memory, 54
  - storage on PIC18Fxx2, 474–482
- nonzero conditional tests
  - 8-bit, 87–88
  - 16-bit, 108–110
- nop instruction, using, 71–72
- NOR function, Boolean logic, 10–13
- not-acknowledge (NAK), 347
- NOT function, Boolean logic, 10–13
- NTZ (non-return-to-zero) asynchronous transmission, 520
- number sequencing computer (NSC), 47
- number sequencing program, 39–43
  - hardware design (fig.), 46
  - PIC18 assembly program for (listing), 82
- numbers
  - binary, 3
  - unsigned, representation, 114–118
- numeric keypad interface, 315–319

**O**

- Ohm's Law, 614
- one-hot coding, 3, 36
- one-time programmable memory, 55
- opcode, 40
- open-drain output, 235–236
- operand, 40
- operating systems, real-time, 531–533
- operations
  - 25LC640 read, write (figs.), 338–339
  - bitwise logical, bit, 78–83
  - CMOS inverter (fig.), 15
  - compare mode. See compare mode
  - extended precision and signed, experiment, 572–573
  - floating-point operations, 195–196
  - mathematical. See arithmetic, mathematics
  - PIC18Fxx2 register, control, literal table read/write, 539–541
  - push, 136
  - push/pop to data stack (fig.), 163
  - shift, 9–10
  - shifts and rotates, 97–100
  - on signed data, 120–127
  - tristate buffer, 232, 234
  - unsigned 8-bit, experiment, 569–571
  - weak pullup, 233–234
- operators
  - arithmetic and logical in C (table), 78
  - conditional tests in C (table), 86
  - pointer dereference, 147

- optimization
    - code, 144
    - and compilation, 210–211
  - OR operator
    - bitwise operations, 78–81
    - logical, 10–13, 86
  - oscillator options, PIC18, 219
  - oscillator start-up timer (OST), 225
  - OST (oscillator start-up timer), 225
  - overflow (OV) flag, 84–85, 119–120
  - overflow, stack, 140
  - overflow, unsigned, 8
  - overflow (V) flag, 123–125
  - overrun errors, 266
  - overruns, buffer, 291
- P**
- packet types/formats in USB (fig.), 526
  - parallel port IO
    - channel basics, 254–259
    - PIC18Fxx2, 231–237, 249–251
  - parity bits, 260
  - PC board, robot application prototype (fig.), 496
  - performance, CMOS power consumption, 227–228
  - period
    - clock, 21
    - registers, 304
  - peripheral interrupts, 285
  - phase locked loop (PLL), 258–259
  - PIC/CAN system (fig.), 519
  - PIC family of microcontrollers, 513–515
  - PIC programmers, 601–604
  - PIC18
    - audio record/playback system resources (table), 460
    - design of audio record/playback system, 456–465
    - home monitoring system application (table), 483
    - interrupts, 282–287
    - robot application resources (table), 497
    - sleep mode, 296–299
    - synchronous IO on, 366–367
    - and synchronous serial IO, 328
    - timer summary, 451–452
    - timer2 subsystem, 304–307
  - PIC18C801 external bus operation (fig.), 512
  - PIC18F242 microprocessor
    - flashing LED, C program for startup, 220–223
    - startup schematic, 216–219
    - as stored program machine, 32
  - PIC18Fxx2 microprocessors
    - analog-to-digital converter, 382–391
    - architecture, instruction set, register summary, 537–548
    - data sheet, 210, 223–225
    - external memory interfacing, 508–510
    - I2C bus on, 348–355
    - parallel port IO, 231–237
    - reset sources, 225–228
    - system startup, 207–231
    - USART hardware system, 264–270
    - using nonvolatile storage, 475–482
  - PIC18FXX2 microcontroller, 53–55
  - PICC-18 runtime code, 214–215
  - PICC-18 tool suite, 554
  - pin functions
    - parallel port IO, 231–237
    - PIC18Fxx2, 216–219
  - PLAB integrated design environment, 549–552
  - playback mode, audio record/playback application, 464–465
  - PLUSWn addressing mode, 155
  - pointers
    - and arrays in C language, 146–151
    - in assembly language, 152–156
    - frame (FPs), 163
    - PIC18, experiment, 573–575
    - stack (SPs), 137, 161, 162, 169
    - subroutine with, 156–160
    - variable containing memory addresses of other variables, 146
  - polarization, circuit, 617
  - polled IO, 282, 289
  - polling loop, 245
  - pop operations, 162–163
  - ports
    - parallel, operation, 231–237
    - serial, debugging, 275–277
  - POSTINCn, POSTDECn addressing modes, 153–154
  - postscalar, on timer, 226, 304
  - power consumption, CMOS (Complementary Metal Oxide Semiconductor), 227–228
  - power-on-reset (POR), 218, 228
  - power-up timer (PWRT), 225
  - powers of two, common (table), 4
  - PREINCn addressing mode, 153–154
  - prescaler, on timer, 304
  - printf statements, and C language, 553, 557–559
  - Program Counter (PC), 44, 73
  - program memory
    - accessing table data from, 160–162
    - Flash, read/write, 478–482
  - Programmable System-on-a-Chip (PSoC), 530
  - programming
    - defensive, 350
    - Jolt/Colt firmware, 601–604
    - state machine, for interrupt-driven IO, 299–303
  - programming languages
    - assembly vs. high-level, 208–210, 249
    - C. *See* C programming language
  - programs
    - See also specific program*
    - assembly language, 41
    - compiling C, 64–66
    - described, 39
    - interrupting normal flow. *See* interrupts
    - stored program machines. *See* stored program machines
  - propagation delay, 24
  - prototyping hints, 598–600
  - pullup resistor, 218
  - pullup, weak, 233–234
  - pulse width measurement
    - described, 415–419
    - using swdetov.c, 593–594
    - with capture mode, 422–428
  - pulse width modulation (PWM), Timer2 and, 442–447
  - push operations, 136, 162
  - push/pop operations to data stack (fig.), 163
  - putc () function, 295
  - PWRT (power-up timer), 225
- Q**
- Quadrature Encoder Interface (QEI), 515
  - qualifiers
    - interrupt, 287
    - variable, 214–215
  - quantization errors, 375
- R**
- R-2R resistor ladder flash DAC, 395–400
  - Rabbit processors, 530
  - RAM (random access memory)
    - described, 55
    - LCD data display (fig.), 244
  - recall instruction, machine code format for (fig.), 141
  - RCON register, 228–229
  - read busy flag, 245
  - read-only memory (ROM), 55
  - real-time operating systems (RTOS), 531–533
  - receivers, infrared, 434
  - reconstruction filters, 399
  - record mode, audio record/playback application, 463–464
  - recursive function calls, dynamic vs. static allocation (fig.), 143
  - register stacks, 141
  - registers
    - See also specific registers*
    - C code operations on 16-bit, 413
    - data and control in USART subsystem, 264
    - EECON1 (table), 476
    - file, 57
    - N-bit, 25
    - period, 304
    - PIC18 pointer, 152
    - PIC18Fxx2 summary, 537–548
    - shadow, 141
    - shift, 26
    - special function register (SFR), 212–214
    - STATUS. *See* STATUS register
    - STKPTR, 138
    - W. *See* WREG register
  - repeated start condition, 348, 351
  - representation
    - fixed-point arithmetic, 188–192
    - floating-point numbers, 192–196
    - unsigned numbers, 114–118
  - reset sources for PIC18Fxx2, 225–228
  - reset conditions, PIC18, 228–231
  - reset vectors, 69
  - resistance, measuring, 598, 613
  - resistor string flash DACs, 393–395
  - resistors, 218, 614–617
  - retlw instruction, machine code format for (fig.), 141
  - return address, subroutines, 136
  - review problems
    - advanced assembly language, higher math, 204–205
    - answers to, 625–653
    - asynchronous serial IO, 278–279
    - data conversion, 407–409

- extended precision, signed operations, 129–131
  - interrupts, timers, 322–325
  - number systems, digital logic review, 29–30
  - PIC18Fxx2 introduction, 73
  - PIC18Fxx2 system startup, parallel port IO, 249–251
  - stored program machines, 47–49
  - subroutines and pointers, 170–174
  - synchronous serial IO, 367–369
  - timers, 452–453
  - unsigned 8-bit arithmetic, logical, conditional operations, 101–102
  - rising edge, and clock signal, 21
  - robot, design and implementation of
    - autonomous, 494–504
    - rotary encoder interface, 309–315
    - rotate instructions, 97–100
    - routes, subroutines, 134–136
    - RS232 standard, 270–273
    - RTOS (real-time operating systems), 531–533
- S**
- sampling period, 456
  - saturating operations, 188–192
  - scanf () function, 557–559
  - scanning keyboard presses, 317
  - scheduler tasks, 531
  - schematics
    - audio record/playback system (fig.), 459
    - for LED, 217–218
    - home monitoring system (fig.), 467
    - for PIC18F242 microprocessor, 216–219
    - robot application, 495
  - Schmitt triggers, 233
  - school, using this book in, xvii–xxi
  - self-reset, 218
  - sequential logic element, 21–24
  - serial data transfer, 254
  - serial EEPROM, experimenting with, 583–584
  - serial IO
    - examples, 273–277
    - and shift registers, 26
    - synchronous, 257–259
  - Serial Peripheral Interface. *See* SPI
  - serial ports, verification, 576
  - serial ports, debugging, 275–277
  - SFR (special function register)
    - and bit variables, 212–214
    - in data memory organization, 56
  - shadow registers, 141
  - shift/add technique for multiplication, 177–182
  - shift operations, 9–10, 97–100, 112–113, 122–123
  - shift registers, 25–26
  - shifter combinational block, 19
  - short data type, unsigned ranges, C (table), 104
  - shorts
    - described, 218
    - in transistors, 15–16
  - sign extension
    - signed data operations, 125–127
    - signed number representation, 118
  - signaling
    - current mode, 255
    - differential, 519
    - USB electrical (fig.), 524
  - signed integers, 8
  - signed branches, 125
  - signed comparisons vs. unsigned comparisons (fig.), 124
  - signed data
    - greater than (>), greater-than-or-equal (>=) operations on, 123–125
    - operations on, 120–123
    - shift operations on, 122–123
  - signed data types, 65
  - simplex communications, 255
  - sleep mode, PIC18 hardware, 296–299
  - software delay loop, 222, 239, 307–309
  - space-width encoding, 434
  - sparkles, 381
  - special function register (SFR) in data memory organization, 56
  - SPI (Serial Peripheral Interface), 331–344
  - sprintf () function, 557–559
  - square root application (fig.), 289
  - square wave generator, 447–450
  - SRAM (Static Random Access Memory), 508–510
  - scanf () function, 557–559
  - stack and call/return, 136–141
  - stack, data, 162
  - stack frames
    - steps in constructing (fig.), 165
    - and subroutines, 162–169
  - stack pointers (SPs), 137, 138, 162
  - startup
    - flashing LED, C code (fig.), 220
    - schematic for PIC18F242 microprocessor, 216–219
  - state assignment, 36
  - state machine programming for interrupt-driven IO, 299–303
  - state machines, IO programming, 240–242
  - static memory allocation, 142–144
  - Static Random Access Memory. *See* SRAM
  - status bits, 264
  - STATUS register
    - high, low priority interrupt service routines, 286
    - in PIC18, 83–85
    - subroutine calls, 145
    - two's complement overflow, 119–120
  - STKPTR register, 138
  - stored program machines
    - components, instruction set design and assembly language, 39–43
    - described, 33
    - hardware design, 43–46
    - vs. FMS clock cycles (table), 46
  - streaming data, capturing, 364–366
  - string functions, convert to lowercase, 156–157
  - subroutines
    - described, using, 134–136, 169
    - implementing in assembly language, 141–145
    - PIC18, experiment, 573–575
    - with pointers, 156–160
    - return values (table), 165
    - stack and call/return, 136–141
    - and stack frames, 162–169
  - subsystems. *See specific subsystem*
  - subtraction
    - /addition, extended precision operations, 107–108
    - BCD arithmetic, 197–199
    - binary and hex, 8–10
  - successive approximation ADC, 377–380
  - survey topics, 534–536
  - switch bounce
    - in mechanical encoders (fig.), 310
  - switch debouncing using timers, 307–309
  - switch statements in C, 89–90
  - switches, LED, 237–242
  - synchronous DRAM, 509
  - synchronous input and D Flip-Flop (DFF), 23
  - synchronous serial IO
    - introduction to, 257–259, 366–367
    - PIC18 and, 328
    - USART synchronous mode, 329–331
  - system requirements, CD-ROM, 655–656
  - systems, embedded, 209
- T**
- T Flip-Flop (TFF), 24
  - TABLAT register, 160–162
  - table reads instructions, 160
  - target addresses, 63
  - task switching, 531–532
  - TBLPTR register, 160
  - TDM (time domain multiplexing), 518
  - testing
    - DS1621 C functions (fig.), 473
    - PIC18F242 functionality, 217–218
  - tests
    - unsigned conditional, 85–94
    - zero, nonzero conditional, 108–110
  - thermometer, DS1621 Digital, 469–474
  - time, common units (table), 22
  - time domain multiplexing (TDM), 518
  - timers
    - 16-bit, Timer1 and Timer3, 419–422
    - introduction, and waveform generation, 587–592
    - PIC18 timer2, 304–307
    - power-up, 225
    - pulse width measurement using capture mode, 422–428
    - summary on, 321–322
    - summary on PIC18, 451–452
    - switch debouncing using, 307–309
    - Timer0 subsystem, 412–418
    - Timer1, Timer3 subsystems, 419–421, 428–432
    - Timer2, and pulse width modulation, 442–447
    - using capture mode for infrared decoding, 433–441
  - TINI (Tiny InterNet Interfaces) modules, 533
  - token packets, 525
  - top of the stack (TOS), 136
  - topics, suggested survey, 534–536
  - topologies, USB (fig.), 524
  - trace buffers, 320
  - transceiver chips, 519
  - transistors
    - CMOS. *See* CMOS
    - invention of, 52
  - transition density, 259
  - tristate buffer, 232, 234

truth tables and logic operations, 11  
 24LC515 serial EEPROM, 356–364, 457  
 25LC640 serial EEPROM interface, 337–344

**U**

unconditional inputs in ASM, 34  
 underflow  
     stack, 139  
     unsigned, 8  
 Unicode, 27–28  
 Universal Serial Bus (USB)  
     standard, 271  
     synchronization, 259  
     topologies, signaling, transactions,  
         523–527  
 Universal Synchronous Asynchronous  
     Receiver Transmitter. See USART  
 unsigned  
     char, 65  
     comparisons, 92–93  
     conditional tests, 85–94  
     data types, 65  
     8-bit operations, experiment, 569–571  
     literal comparisons, 93–94  
     number conversion, 5–6  
     number representation, 114–118  
     overflow, underflow, 7, 8  
     vs. signed comparisons (fig.), 124  
 USART (Universal Synchronous Asynchro-  
     nous Receiver Transmitter)

hardware subsystem described,  
     264–270, 277  
 and interrupt sources, 296–297  
     synchronous mode, 329–331  
 USB (Universal Serial Bus), 259, 271,  
     523–527

**V**

vacuum tubes, 52  
 variable declarations in C language,  
     559–561  
 variables  
     assignment of extended precision, 106  
     bit, and special function registers,  
         212–214  
     global, 142  
     memory addresses of, 146  
     variable qualifiers (table), 215  
 vectors, interrupt and reset, 69  
 verifying serial ports, 576  
 virtual memory, 529  
 vishift () subroutine, 136  
 vlshtft () C function  
     in assembly language (fig.), 144  
     calling from main () (fig.), 145  
 voice recorder, digital, 32–33  
 voltage, measuring, 598, 613  
 voltage comparator, 375  
 voltage regulators, 217  
 Von Neumann machines, 33

**W**

W register  
     in PIC18, 83  
     and push operations, 162  
     subroutine calls, 145  
 wakeup support, 296–297  
 watchdog timer (WDT), 226, 228–231, 249,  
     296  
 waveform events, 22  
 waveform generation, 587–592  
 waveforms, clock, 21  
 WDT (watchdog timer), 226, 228–231, 249,  
     296  
 weak pullup, 233–234  
 while {} loops, using, 94–97  
 wire wrapping, 599–600  
 wired logic, 235  
 WREG instruction, using, 71

**X**

XOR function, Boolean logic, 11  
 XOR operator, bitwise operations, 78–81

**Z**

zero conditional tests  
     8-bit, 87–88  
     16-bit, 108–110  
 zero (Z) flag, 84–85