

PROGRAMMING AND CUSTOMIZING THE

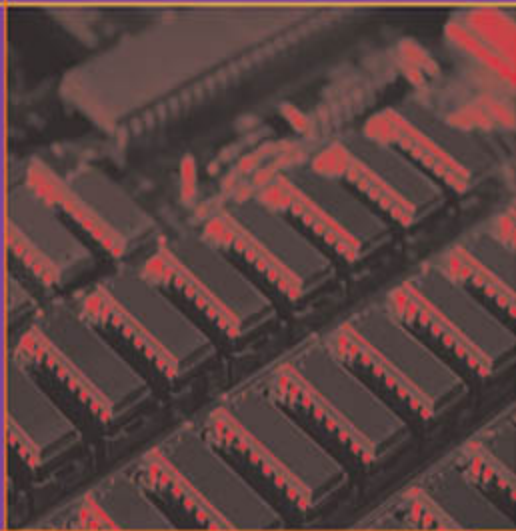
50,000
COPIES
SOLD!

PIC[®]

3rd EDITION

Microcontroller

- **FOCUS:** 3 major PIC MCU families and their roles in the applications that use them
- **LANGUAGES:** Assembly, BASIC, and C—the most popular for PIC programming
- **INSTRUCTIONS:** Set up your own PIC microcontroller development lab
- **APPLICATIONS:** Work on 100 complete experiments, and customize to your own specifications



Includes valuable coupons
for creating your own PIC[®]
microcontroller development lab



Myke Predko

PROGRAMMING AND CUSTOMIZING THE PIC[®] MICROCONTROLLER

About the Author

A resident of Toronto, Canada, Myke Predko is the best-selling author of 13 McGraw-Hill electronics and engineering titles, including *Digital Electronics Demystified* and *123 Robotics Experiments for the Evil Genius*. He holds a B.S.E.E. from the University of Waterloo, and is the Electrical Engineering/Firmware Development Manager for Logitech's Harmony Remote Control Business Unit.

PROGRAMMING AND CUSTOMIZING THE PIC[®] MICROCONTROLLER

MYKE PREDKO

Third Edition



New York Chicago San Francisco Lisbon London Madrid
Mexico City Milan New Delhi San Juan Seoul
Singapore Sydney Toronto

Copyright © 2008, 2002, 1997 by The McGraw-Hill Companies, Inc. All rights reserved. Manufactured in the United States of America. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

0-07-151087-7

The material in this eBook also appears in the print version of this title: 0-07-147287-8.

All trademarks are trademarks of their respective owners. Rather than put a trademark symbol after every occurrence of a trademarked name, we use names in an editorial fashion only, and to the benefit of the trademark owner, with no intention of infringement of the trademark. Where such designations appear in this book, they have been printed with initial caps.

McGraw-Hill eBooks are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. For more information, please contact George Hoare, Special Sales, at george_hoare@mcgraw-hill.com or (212) 904-4069.

TERMS OF USE

This is a copyrighted work and The McGraw-Hill Companies, Inc. ("McGraw-Hill") and its licensors reserve all rights in and to the work. Use of this work is subject to these terms. Except as permitted under the Copyright Act of 1976 and the right to store and retrieve one copy of the work, you may not decompile, disassemble, reverse engineer, reproduce, modify, create derivative works based upon, transmit, distribute, disseminate, sell, publish or sublicense the work or any part of it without McGraw-Hill's prior consent. You may use the work for your own noncommercial and personal use; any other use of the work is strictly prohibited. Your right to use the work may be terminated if you fail to comply with these terms.

THE WORK IS PROVIDED "AS IS." McGRAW-HILL AND ITS LICENSORS MAKE NO GUARANTEES OR WARRANTIES AS TO THE ACCURACY, ADEQUACY OR COMPLETENESS OF OR RESULTS TO BE OBTAINED FROM USING THE WORK, INCLUDING ANY INFORMATION THAT CAN BE ACCESSED THROUGH THE WORK VIA HYPERLINK OR OTHERWISE, AND EXPRESSLY DISCLAIM ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. McGraw-Hill and its licensors do not warrant or guarantee that the functions contained in the work will meet your requirements or that its operation will be uninterrupted or error free. Neither McGraw-Hill nor its licensors shall be liable to you or anyone else for any inaccuracy, error or omission, regardless of cause, in the work or for any damages resulting therefrom. McGraw-Hill has no responsibility for the content of any information accessed through the work. Under no circumstances shall McGraw-Hill and/or its licensors be liable for any indirect, incidental, special, punitive, consequential or similar damages that result from the use of or inability to use the work, even if any of them has been advised of the possibility of such damages. This limitation of liability shall apply to any claim or cause whatsoever whether such claim or cause arises in contract, tort or otherwise.

DOI: 10.1036/0071472878



Professional



Want to learn more?

We hope you enjoy this McGraw-Hill eBook! If you'd like more information about this book, its author, or related books and websites, please [click here](#).

CONTENTS

Introduction	<i>xi</i>
Acknowledgments	<i>xxi</i>
 Chapter 1 Embedded Microcontrollers	 1
Microcontroller Types	1
Internal Hardware	2
Applications	5
Processor Architectures	7
Instructions and Software	12
Peripheral Functions	17
Memory Types	21
Microcontroller Communication	28
Device Packaging	35
Application Development Tools	39
 Chapter 2 The Microchip PIC Microcontroller	 43
Accessing the Microchip Web Site	43
PIC Microcontroller Feature Summary	48
Features Unique to the PIC Microcontroller	54
PIC Microcontroller Families	59
 Chapter 3 Software Development Tools	 63
Tools Overview	65
High Level Languages	83
Microchip MPLAB IDE	103
 Chapter 4 Programming PIC Microcontrollers	 155
Hex File Format	156
Code Protection	158
Parallel Programming	159
PIC ICSP Programmer Interface	166
Microchip Programmers	178
My Programmers	181
Third-Party Programmers	204
 Chapter 5 Emulators and Debuggers	 207
MPLAB ICE-2000	210
MPLAB REAL ICE	213

MPLAB ICD 2 Debugger	214
The Emu-II	219
Other Emulators	241

Chapter 6 The Microchip PIC MCU Processor Architecture 243

The CPU	244
Hardware and File Registers	248
The PIC Microcontroller's ALU	254
Data Movement	260
The Program Counter and Stack	264
Reset	268
Interrupts	271
Architecture Differences	273

Chapter 7 Using the PIC MCU Instruction Set 293

Setting Up the MPLAB IDE Simulator with a Test Template	294
PIC MCU Instruction Types	297
The Mid-Range Instruction Set	303
Low-End PIC Microcontroller Instruction Set	348
PIC18 Instruction Set	356

Chapter 8 Assembly-Language Software Techniques 373

Sample Template	374
Labels, Addresses, and Flags	376
Subroutines with Parameter Passing	381
Subtraction, Comparing and Negation	385
Bit AND and OR	389
16-Bit Operations	390
MULDiv, Constant Multiplication and Division	392
Delays	400
Patch Space	405
Structures, Pointers, and Arrays	407
Sorting Data	414
Interrupts	419
Reentrant Subroutines	423
Simulating Logic	423
Event-Driven Programming	426
State Machine Programming	429
Porting Code Between PIC Microcontroller Device Architectures	430
Optimizing PIC Microcontroller Applications	438
A Baker's Dozen Rules to Follow That Will Help to Avoid Application Software Problems	443

Chapter 9 Basic Operating Features 445

Power Input and Decoupling	446
Configuration Fuses	451
OPTION Register	470
TMR0	478
Interrupt Operation	483
The Right PIC Microcontroller to Learn On	485

Chapter 10 Macro Development	489
PIC Microcontroller Assembly-Language Macros	489
The Difference Between Defines and Macros	492
The Assembler Calculator	494
Multiline C Macros	499
Conditional Assembly/Compilation	500
Using Defines and Conditional Assembly for Application Debug	507
Debugging Macros	509
Structured Programming Macros	513
Chapter 11 Building and Linking	519
Creating Linked Applications	519
Chapter 12 Bootloaders	527
Bootloader Requirements	528
Mid-Range Bootloaders	530
PIC18 Bootloaders	535
Chapter 13 Real-Time Operating Systems	537
Low-End and Mid-Range RTOSs	541
PIC18 RTOS Design	542
Chapter 14 Debugging Your Applications	565
Document the Expected State	566
Characterize the Problem	567
Hypothesize and Test Your Hypothesis	569
Propose Corrective Actions	571
Test Fixes	572
Release Your Solution	576
Debug: An Application to Test Your Debug Skills	577
Chapter 15 PIC Microcontroller Application Design and Hardware Interfacing	589
Requirements Definition	590
PIC Microcontroller Resource Allocation	595
Effective User Interfacing	597
Project Management	599
Power Management	603
Reset	608
Interfacing to External Devices	611
Chapter 16 PIC MCU Optional Hardware Features	617
Mid-Range Built-in EEPROM/Flash Access	618
TMR1	624
TMR2	626
Compare/Capture/PWM (CCP) Module	628
Serial I/O	633
Analog I/O	649
Parallel Slave Port (PSP)	657
In-Circuit Serial Programming (ICSP)	659

Chapter 17 PIC MCU Input and Output Device Interfacing	661
LEDs	661
Switch Bounce	665
Matrix Keypads	668
LCDs	672
Analog I/O	682
Audio Output	690
Relays and Solenoids	692
Asynchronous (NRZ) Serial Interfaces	693
Synchronous Serial Interfaces	704
Chapter 18 Motor Control	711
Dc Motors	711
Stepper Motors	724
R/C Servo Control	733
Chapter 19 Practical PC Interfacing	739
PC Software Application Development Tools	740
Serial Port	742
Parallel Port	749
Chapter 20 PIC Microcontroller Application Basics	755
Jumping Around	755
Some Basic Functions	771
Analog Input/Output	798
I/O with Interrupts	810
Serial I/O	832
Chapter 21 Projects	853
Low-End Devices	853
Mid-Range Devices	878
PIC18 Devices	953
Appendix A Resources	965
Microchip	965
Books to Help You Learn More about the PIC Microcontroller	966
Useful Books	967
Recommended PIC Microcontroller Websites	970
Periodicals	971
Other Websites of Interest	972
Part Suppliers	973
Appendix B PIC Microcontroller Summary	977
Feature to Part Number Table	977
Instruction Sets	977
I/O Register Addresses	1016
Device Pinouts	1030

Appendix C Useful Tables and Data	1061
Electrical Engineering Formulas	1063
Mathematical Formulas	1065
Mathematical Conversions	1066
ASCII	1067
Appendix D Miscellaneous Electronic Reference Information	1073
Basic Electronic Components and Their Symbols	1073
Test Equipment	1080
Appendix E Basic Programming Language	1089
PICBASIC	1091
Appendix F C Programming Language	1123
Common Library Functions	1130
PICC Library Functions	1133
Microchip C18 Library Functions	1138
Appendix G Reuse, Return, and Recycle	1149
Useful Snippets	1150
Mykerns.inc	1160
Sixteen-Bit Numbers	1200
Glossary	1213
Index	1229

This page intentionally left blank

INTRODUCTION

In a time when digital electronics is becoming more complex and less accessible to students and low-end project and product developers, microcontrollers have become the tools of choice for learning about electronics and programming as well as providing the capabilities needed to create sophisticated applications cheaply and easily. If you were to look through any electronics magazine, you would discover that almost every example application uses a microcontroller (often abbreviated to just MCU) to provide a user interface, sequence operations, and respond to changing inputs. These chips are inexpensive, have a surprisingly high level of performance, and are easy to integrate into an application. Microcontrollers have reversed the trend of modern electronics and provide an easy and effective way for students, hobbyists, and professionals to create applications.

In the introduction to the first edition of this book, I explained my fascination with the Intel 8048 microcontroller. I first discovered this device when I was looking at the first IBM Personal Computer's schematics. While the PC's schematic itself took up 20 pages, the keyboard's simply consisted of a single 8048 chips which provided a "bridge" between the keys on the keyboard and the PC system unit. I got a copy of the 8048's datasheet and was amazed at the features available, making the single chip device very analogous to a complete computer system with a processor, application storage, variable storage, timers, processor interrupt capability, and Input/Output (I/O). This single chip gave designers the capability of developing highly sophisticated applications in one simple component that could be easily wired into the overall product.

One of the most popular and easy to use microcontroller families available in the market today is the Microchip "PIC microcontroller." Originally known as the PIC (for Peripheral Interface Controller), the PIC microcontroller *MCU* consists of over 400 variations (or Part Numbers), each designed to be optimal in different applications. These variations consist of a number of memory configurations, different I/O pin arrangements, amount of support hardware required, packaging, and available peripheral functions. This wide range of device options is not unique to the PIC microcontroller; many other microcontrollers can boast a similar menu of part numbers with different options for the designer.

Since writing the first edition of this book, Microchip has released over 250 new versions of PIC microcontrollers (with each version known as a new Part Number). Each part number is built from a specific processor family and has unique program, register and data memory types and capacities as well as I/O features that are designed to simplify the task of designing an application. The parts are very well documented and when you look at Microchip's web site (<http://www.microchip.com>) you will discover that there are literally thousands of Adobe Acrobat (pdf) documents, including

datasheets, application notes, and manuals that can be used to help you understand the most efficient way to use the PIC microcontrollers in your applications. Microchip has gone a step further than most other part suppliers and made available the full featured MPLAB integrated development environment which is designed to support and automate every step of the application development process. The unimaginable work that the more than 250 PIC microcontroller part numbers as well as the support tools and documentation produced by Microchip is the reason why developers turn to the PIC microcontroller as their first choice.

For new application developers, this mountain of material is daunting and choosing and effectively utilizing the PIC can be intimidating.

If you were to look through magazines, books or do a search on the Internet of different sample applications that are designed around the PIC MCU, you would discover that the vast majority are designed around two or three part numbers (the PIC16F84, PIC16C54, and perhaps the PIC16F877). This is unfortunate because there are a plethora of different PIC microcontrollers that you can choose from to build your applications around and chances are there will be a part number that has exactly the features that will allow you to simply design the circuit and efficiently develop the required code.

I must confess that in the previous editions of this book I have been guilty of the same practice; I have focused on a single part number or family. In this edition, I have worked at exposing you to more of the complete range of PIC microcontrollers along with the confidence to select the best device for your needs from the hundreds available to choose from. In doing this, I am focusing on the different functions built into the different PIC microcontroller chips instead of specific part numbers and the peripheral functions built into them.

As well as changing my focus for this book, I would also like to point out that Microchip has made many differences to the PIC microcontroller line up since the second edition. The most significant differences that you will see is the wider voltage range available throughout the line; previously when PIC microcontrollers were needed for low-voltage applications, the developer was required to use the LC or LF types of parts. The most recent PIC microcontrollers are normally designed across a wide operating voltage (usually from 2.0 to 6.0 volts), simplifying the design of the power supply required for the application. It isn't unusual to see many products designed to use a couple of alkaline AA batteries for power or a single lithium "button" or "hearing aid" battery without any other components other than a couple of capacitors and a switch. In many cases, the switch isn't included because of the excellent low-power capabilities of the PIC microcontroller.

I should also point out that there are many more Flash program memory based parts than when the last edition of this book was released. These parts make it easier to learn about the PIC microcontroller, allowing you to leave them in circuit and not have to pull them out to erase them and then reprogram them. They also give a new dimension to products allowing them to be reprogrammed in the field.

I should point out that when I wrote the second edition, I recommended that for commercial developers create and debug their applications choosing Flash based parts

that could be substituted with on time programmable (OTP) EPROM PIC microcontrollers when the code was complete and qualified because there was very little likelihood that a product's firmware would be reprogrammed in the field. As fate would have it, I am now responsible for the hardware and software design of PIC microcontroller based products (Logitech's Harmony remote controls) that is designed to have its firmware updated in the field. From the time I have spent, I can see the advantages of being able to have customers update their firmware when new functions become available although I cringe anytime we release a fix to the basic firmware to be downloaded by our customers.

Having the ability to reprogram the Flash in the field should never be used as a "catch" for poor product design and qualification.

With the different capabilities of the three main PIC microcontroller family architectures, I wanted to point out that I would characterize applications for the different parts as follows:

The low-end devices (formerly referred to as the PIC16C5x but are now identified by their 12 bit instruction size) should be used for simple applications including implementation of simple logic functions. Their limited addressing capabilities make them poorly suited, especially compared to the other architecture families, to applications which require sophisticated user I/O or connectivity with other devices. Somebody may point out that the Parallax Stamp products use the low-end PIC microcontrollers, but at the time the products were designed, these were the cheapest chips available; today the mid-range and PIC18 devices provide much more capabilities at a lower price point than the low-end did when the Stamp was first conceived.

The mid-range (14 bit instruction size) PIC microcontrollers are excellent choices for applications which require advanced I/O capabilities and significant user interfacing. As I write this, there are over 200 mid-range part numbers, each with differing I/O capabilities meaning that it is very unlikely that you will not be able to find a part with the features that you require eliminating the need for any additional I/O peripheral chips.

I don't like to call the PIC18 the "high end" of the PIC microcontroller line due to the recent introduction of the PIC24 16 bit data word size MCUs and the PIC17 which has a 16 bit instruction size like the PIC18. The PIC18 architecture is your best choice for developing applications that need to communicate with external devices or your PC (many of the chips have built in USB ports which do not require any external interface chips). The PIC18 also offers the largest program memory space and best performance of the different PIC family devices allowing you to follow design techniques rather trying to maximize your program memory usage and implement the fastest running code possible. Microchip is working at making sure the PIC18 is a very cost effective device and, when I do the fourth edition of this book, I can see it displacing the other two architectures.

The final point I want to make about this book is that I am emphasizing the use of high level languages (C specifically with some BASIC). This is due to the improvement in compiler technology for the PIC microcontroller architecture as well as the PIC18 architecture which is well suited for the operation of code from high level

languages. I never would have imagined it when I wrote the previous editions of this book but I can honestly say that it is now possible to create PIC microcontroller applications without having to learn assembly language for the architecture of the device that you are going to use.

In the previous edition of this book, while focusing on the knowledgeable user, I provided a great deal of introductory electronics and programming information. This made the book more cumbersome for the targeted reader and did not provide a satisfactory experience for the beginner. For absolute beginners, I have written *123 Robotics Experiments for the Evil Genius* (ISBN 0071413588) which will provide you with the basics of electronics and programming as well as some guidance on how to create your own robots. For developers with some knowledge of programming and electronics I have written *123 PIC Microcontroller Experiments for the Evil Genius* (ISBN 0071451420) which will provide a more comprehensive introduction to microcontrollers and designing them into application circuits.

Three types of applications have been included in this book. *Experiments* are simple applications that, for the most part, do not do anything useful other than help explain how the PIC microcontroller is programmed and used in circuits. These applications along with the code and circuitry used with them can be used in your own applications, avoiding the need for you to come up with them on your own. The *Projects* are complete applications that demonstrate how the PIC microcontroller can interface with different devices. While some of the applications are quite complex, I have worked at keeping them all as simple as possible and design them so they can be built and tested in one evening. The applications have also been designed to avoid high speed execution and AC transmission line issues however possible to make prototype builds as robust as possible. The last type of application presented here are the various developer's tools used for application software development. In this book, I have included the design for two different types of PIC microcontroller programmers and a device emulator that can be used to help with your own PIC microcontroller application development. By studying and working through these different application types you will gain a strong insight into the operation of the PIC microcontroller and help to understand how you can develop your applications using the different part numbers available to you.

PIC Microcontroller Resources and Tools

Unlike the previous edition there is no CD-ROM included with this book and there is no PCB for the user to build their own PIC microcontroller programmer (although the design for the PIC microcontroller programmer is discussed in the body of the book). The decision not to include these features was quite easy when I looked at the current situation and what is being offered with this book.

The primary purpose of the CD-ROM included in the previous editions of the book was to provide the source code for the experiments and projects as well as a source for

the development tools and the datasheets. When I look at the internet and what it offers, I feel like the CD-ROM is redundant and does not allow you to get the latest versions of the code (I find I end up going back and improving the code) and it is a nightmare for keeping track of the latest versions of the development tools and datasheets. Microchip, much more than the average chip manufacturer, works very hard at modernizing their tools—it isn't unusual to see four or five releases of MPLAB IDE over a year. In the time it has taken to copyedit and set the pages for this book, two versions of MPLAB IDE have been released. Similarly, datasheets are also kept under constant review and updating.

To get the latest Microchip tools and datasheets, I recommend that you download them from:

<http://www.microchip.com>

For the source code for the experiments and projects in this book, you can find them at:

<http://www.books.mcgraw-hill.com/authors/predkopacpic>

I had an extremely difficult time supporting the El-Cheapo, especially with the rapid change in PIC microcontroller part numbers, new requirements from users, the withdrawal of support of VB6 by Microsoft, the changes to the Windows operating system, and the changes made to standard PCs. Instead, I am very pleased that Microchip is able to offer a series of coupons with this book to allow you to order their development tools at significantly reduced prices. These tools are well maintained by Microchip including being integrated with MPLAB IDE, are usable in Vista PCs (which the El-Cheapo is not), and are very reliable.

When you start out, I recommend buying the MPLAB ICD 2 which will provide you with the capability of programming Flash based parts as well as a single stepping, breakpoint, and memory access debug capability on many of the chips. The debug capability could be considered a “poor man’s” emulator and will give you the ability to more quickly debug and qualify your applications. In the text, I will point out some low-cost adapters that you may want to purchase to simplify programming and interfacing to your applications.

As you become more familiar with the PIC microcontroller families and are developing more complex applications, you will want to look at the MPLAB Real ICD or MPLAB ICE 2000 which will give you full in circuit emulator capabilities. The MPLAB Real ICD can use the MPLAB ICD 2 interfaces or a custom one on its own while the MPLAB ICE 2000 provides a “pod” which replaces the microcontroller in the application circuit.

Finally, for programming PIC microcontrollers for wiring in applications, the best programmer on the market is the MPLAB PM3. This device “production” programs (rather than “development” programs) all PIC microcontroller part numbers and can operate as a stand-alone device or connected to MPLAB IDE as part of your development lab.

Conventions Used in This Book

TABLE I.1 CONVENTIONS USED IN THIS BOOK	
SYMBOL	MEANING
Ω	Unit of resistance—ohms
k	Thousands of Ohms resistance
M Ω	Millions of Ohms resistance
μ F	microFarads—1/1,000,000 Farads
pF	picoFarads—1/1,000,000,000,000 Farads
s	Seconds
ms	Milliseconds—1/1,000 seconds
μ s	microseconds—1/1,000,000 seconds
ns	Nanoseconds—1/1,000,000,000 seconds
Hz	Hertz (number of cycles per second)
kHz	kiloHertz—1,000 Hertz
MHz	megaHertz—1,000,000 Hertz
GHz	gigahertz—1,000,000,000 Hertz
####	Decimal number (# is 0 to 9)
-####	Negative decimal number
0x0 #####	Hexadecimal number (“#” is “0” to “9,” “A,” “B,” “C,” “D,” “E,” “F.”)
0b0 #####	Binary number (“#” is “0” or “1”)
{ }	Optional information or text
	Either/or parameters
Label# _Label	Negatively active signal or bit
Register.bit	Specific bit in a register
Register.bit:bit	Range of bits in a register
Monospace font	Example code
//	Text is comment information
: ...	“And so on.” Text is repeated or continued

(Continued)

TABLE I.1 CONVENTIONS USED IN THIS BOOK (CONTINUED)		
SYMBOL	MEANING	
&	—Two input bitwise AND	
	—Truth Table:	
	Inputs Output	
	A B	

	0 0	0
	0 1	0
	1 0	0
	1 1	1
AND &&	Logical AND	
	—Two input bitwise OR	
	—Truth Table:	
	Inputs Output	
	A B	

	0 0	0
	0 1	1
	1 0	1
	1 1	1
OR	Logical OR	
^	—Two Input bitwise XOR	
	—Truth Table:	
	Inputs Output	
	A B	

	0 0	0
	0 1	1
	1 0	1
	1 1	0

(Continued)

TABLE I.1 CONVENTIONS USED IN THIS BOOK (CONTINUED)											
SYMBOL	MEANING										
XOR	Logical XOR										
!	—Single Input Bitwise Inversion —Truth Table: <table><tr><td>Input</td><td>Output</td></tr><tr><td>A</td><td></td></tr><tr><td>-----</td><td>-----</td></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	Input	Output	A		-----	-----	0	1	1	0
Input	Output										
A											
-----	-----										
0	1										
1	0										
NOT	Logical inversion										
+	Addition										
–	Subtraction or negation or a decimal										
*	Multiplication										
/	Division										
%	Modulus of two numbers or remainder of integer division										
<< #	Shift value to the left “#” bits										
>> #	Shift value to the left “#” bits										

Along with defining Units there are a few terms and expressions I should define here to make sure you are clear on what I am saying in the text. These terms are often used in electronics and programming, although my use of them is specific to microcontrollers and the PIC microcontroller.

Application	The hardware circuit and programming code used to make up a microcontroller project. Both are required for the microcontroller to work properly.
Source Code	The human-readable instructions used in an application that are converted by a compiler or assembler into instructions that the microcontroller’s processor can execute directly.
Software	I use the generic term <i>Software</i> for the application’s code. You may have seen the term replaced with <i>Firmware</i> in some references.

What’s New in This Edition

While much of the material from the second edition has been retained for this one, there have been some significant changes and additions for this edition.

They are:

- 1** Some of the information was given in the first edition before prerequisite information was presented. Some chapters have been reordered and changed to eliminate this from being a problem in the second edition, and this has been continued in the third edition.
- 2** All pseudo-code examples are written in “C.” C is the most popular high-level language for PIC microcontroller application development (as well as most technical programming). I have followed C conventions in all areas of the book when presenting information or data, wherever possible.
- 3** A .zip file of the source code used in this book can be found at: <http://www.books.mcgraw-hill.com/authors/predkopapic>
- 4** A table format for register definitions has been used in this edition to help make finding specific information easier. Bits are defined from the most significant to least significant to make translating the bit numbers to values simpler.
- 5** A glossary of terms used in the book has been included. This glossary has been optimized for the PIC microcontroller and the concepts required for it.
- 6** “Holes” in information and data have been eliminated.
- 7** Most of the references to the PIC17 (including sample projects) have been removed. Microchip does not have any plans for new PIC17 architecture parts and the ones available do not take advantage of modern device features like Flash program memory.
- 8** As noted above, this book has been written for readers with a knowledge of programming and electronics. The introductory electronics and programming information that was found in the CD-ROM that accompanied this book have been removed; however, some of the information pertinent to the PIC microcontroller has been retained.
- 9** More glossary/appendix reference data has been provided.
- 10** The “Conventions Used in This Book” section of the introduction has been expanded to include all mathematical operators and symbols used in the text and formulas.
- 11** The example experiments, projects, and tools have been enhanced.
- 12** The experiments, projects, and tools have been relabeled to avoid confusion regarding the order in which information is presented. In the original edition, the applications were labeled according to the order in which they were developed. In this edition, the experiments and projects have been labeled according to what category of application they come under and the order in which they appear.
- 13** There are a number of new experiments, projects, and tools added to this book. These additions are used to demonstrate new functions or clarify areas that were ambiguous.
- 14** Complete schematics and bills of material are available for all the applications that are presented in this book.
- 15** The El Cheapo programmer PCB that was included with the book has not been included in this edition due to the difficulty in creating a common interface to the PC that does not require preprogrammed parts. Instead, there are web coupons available for you to order Microchip development tools.

- 16** PC Interface code has been tested on a variety of PCs. While I cannot guarantee that the code will work on all PCs, it should be robust enough to work on most without problems. I have tried to include both MS-DOS as well as Microsoft Windows code for the projects.
- 17** All parts specified in this book are available easily from a variety of sources. Where there can be confusion with regards to the parts, I have listed distributor part numbers in the text.
- 18** The latest PIC microcontroller devices and features are presented. The eight and fourteen pin PIC microcontrollers along with the latest EEPROM/Flash and PIC18 microcontroller parts and their features have been added to this book. I realize that between the time when this was written and when the book comes to print even more parts will be added. Please consult the Microchip web site for the latest list of available PIC microcontroller part numbers.
- 19** With the description of each interface, I have included sample code that can be placed directly into your applications. These “snippets” of code are written with constants or variables that are described in the accompanying text.
- 20** To help you with your application development I have pulled out many of the experiments that dealt with specific interfaces and added a chapter on DC and stepper motor control.
- 21** New chapters on assembly language and macro programming have been added to help you understand how optimal code is developed and how it is measured. The measurements that I introduce may be considered somewhat unusual, but I believe they are appropriate for real-time microcontroller applications.

Copyrights and Trademarks

Microchip is the owner of the following trademarks: PIC, PIC microcontroller, REAL ICE, ICSP, KEELOQ, MPLAB, PICSTART, PRO MATE and PICMASTER. PICC and PICC Lite are owned by HI-TECH Software. microEngineering Labs, Inc. is the owner of PicBasic. Microsoft is the owner of Windows/95, Windows/98, Windows/NT, Windows/2000, and Visual Basic. All other copyrights and trademarks not listed are the property of their respective manufacturers and owners.

ACKNOWLEDGMENTS

This edition (as well as the first two) would not have been possible without the generous help of a multitude of people and companies. While my name is on the cover, this book wouldn't have been possible without their efforts and suggestions—the list of people that I feel I must recognize grows substantially with each new edition.

The first “thank you” goes to everyone on MIT's PICList. The two thousand or so individuals subscribed to this list server have made the PIC microcontroller probably the best supported and most interesting chips available in the market today. While I could probably fill several pages of names listing everyone who has answered my questions and made suggestions on how this second edition could be better, I am going to refrain in fear that I will miss someone.

This book wouldn't have been possible except for the patience and enthusiasm of my editor at McGraw-Hill, Judy Bass. During the development of this book, I took on a new job and built a new home which made it difficult for me to focus as much attention as I should have on the book and the manuscript was subsequently very late. Judy was exceedingly understanding and helpful in getting this book on track and ready for publication.

Ben Wirz has been an invaluable resource on this book, helping me to better understand the control of motors and basic robotics concepts; Ben has also been my partner with the TAB Electronics Build Your Own Robot kits and those products as well as this book would not have been possible without all his hard work. I really appreciated his critiques of the materials in the book as well as his suggestions on what the book needed to make it better for everyone.

Along with Ben, I would like to thank Don McKenzie, Kalle Pihlajasaari, Mick Gulovsen, John Peatman, and Philippe Techer for your suggestions and ideas. A lot of the projects in this book wouldn't exist without their help, ideas, or the SimmStick.

I have never seen a quote pointing out the irony that the greatest opportunity to learn is by teaching others. I want to thank Blair Clarkson of the Ontario Science Centre for his tireless energy in running the OSC/Celestica robot workshops along with his suggestions and ideas for robots and opportunities for the community at large. I would also like to recognize Brad North at Rick Hansen Secondary School in Mississauga, Ontario, and thank him for the opportunity to spend time in the classroom meeting with his students and helping them learn more about electronics, programming, and the PIC microcontroller. In both these situations, I believe I have walked away with a lot more than what I was able to give and I want to thank both of these devoted individuals for the opportunities to work with them.

I am pleased to say that Microchip has been behind me every step of the way for this book project. Along with (early) part samples, tool references, and information,

I appreciate the fast response to questions and the help with making sure I had the correct information. A big thank you goes out to Fadi Atallah, Andre Nemat, Len Chiella, and Greg Anderson of the local (Toronto) Microchip offices as well as Carol Popovich, Al Lovrich, Kris Aman, Elizabeth Hancock, and Eric Sells for the time spent on the phone, the many emails, graphics, parts, and suggestions. I know that supporting authors is not in any of their job descriptions and I appreciate the time they were able to devote to me.

Along with the efforts of the Microchip employees, I would like to thank Dave Cochran of Pipe-Thompson Technologies who made sure that I always had everything I needed and all my questions were answered. Dave, I also appreciated the lunches at Grazie with you, Len and Greg where not only did we agree on what should be in the book, but also on what to order.

Jeff Schmoyer of microEngineering Labs, Inc. was an excellent resource for me to understand how “PicBasic” worked and was always enthusiastic and helpful for all the questions that I had. PicBasic and the “EPIC” programmer are outstanding tools that I recommend to both new PIC microcontroller MCU developers and experienced application designers alike.

I learned more about compiler operation from Walter Banks of Bytecraft Limited in a few hours of telephone conversations than I did in my two senior years at university. While much of this information came after I had finished this book, the time spent allowed me to go back over the experiments and applications presented in this book with a much better eye toward making the code more efficient.

There are five other companies that I have grown to rely on an awful lot for creating books as well as doing my own home projects. I recognized two of these companies in the first edition and I felt I should include three others for their excellent service in the Toronto area.

Since writing the first edition of this book, Digi-Key has continued their excellent customer support and improved upon it with their web pages and overnight home delivery to Canada. AP Circuits are still the best quick turn PCB prototyping house in the business and I recommend that you use them for all your projects.

For the first two editions, I have relied upon M & A Cameras and LightLabs here in Toronto for equipment rentals, photofinishing, and advice. I realize that M & A also rent equipment to the professional photographers in movie industry, but they have always taken the time to answer my questions and help me become a better photographer. LightLabs has always done their level best to ensure the poor pictures I have taken come out as clear and scanner ready as possible. I know I can still do a lot better, but both these companies have done a lot to hide my mistakes. Lastly, I want to thank the people at Supremetronic on Queen Street in Toronto for their unbelievably well stocked shelves of all the “little stuff” that I need for developing circuits and applications along with the time spent helping me find (and count) the parts that I have needed.

Professionally, I have been blessed with remarkable places to work, develop, and learn. I started out in IBM, which was then spun off into “Celestica” and now I am proud to be working for Logitech in the Harmony Remote Control Business Unit. In each of these companies, I have been amazed at the diverse and rich talent that these

companies have been able to attract. There are many people I would like to thank for answering my questions and helping me to understand the PIC microcontroller from different perspectives and while I am reluctant to try and name everyone that has helped me over the years, I would like to recognize Karim Osman, John Scharkov, and Jules Varenikic for the time they have spent with me talking about PIC microcontroller and robotics projects and helping me with creating them.

To my children, Joel, Elliot, Marya, and Talitha (our family is continually growing), thank you for recognizing the notes, parts, and projects left lying around the house are not to be touched and when I'm mumbling about strange things, I'm probably not listening to how your day went. The four of you would be absolutely perfect if you would just finish your homework before it was due. This book is something that you should be proud of as well.

Finally, the biggest "thank you" has to go to my aptly named wife, Patience. Thank you for letting me spend all those hours in front of my PC and then spending a similar number of hours helping me out by keying in the never ending pages of scrawl that was written in airport bars, hotel rooms, and cramped airline seats. Thank you for putting up with the incessant FedEx, Purolator, and UPS couriers, organizing the sale of the old house and being part of the creation of a completely new one. Writing something like this book is an unbelievably arduous task and it never would have been possible without your love and support.

Let's go and enjoy our new home.

Myke Predko
Toronto, Canada
August 2007

This page intentionally left blank

PROGRAMMING AND CUSTOMIZING THE PIC[®] MICROCONTROLLER

This page intentionally left blank

EMBEDDED MICROCONTROLLERS

The primary role of the Microchip PIC® and other embedded microcontrollers is to provide inexpensive, programmable logic control and interfacing to external devices. This means they typically are not required to provide highly complex functions—they can't replace the Opteron processor in your ISP's server. They are well suited to monitoring a variety of inputs, including digital signals, button presses, and analog inputs, and responding to them using the preprogrammed instructions that are executed by the built-in computer processor. An embedded microcontroller can respond to these inputs with a wide variety of outputs that are appropriate for different devices. These capabilities are available to you at a very reasonable cost without a lot of effort.

This chapter will introduce you to the functions and features that you should look for when choosing a microcontroller for a specific target application. While keeping the information as general as possible, I have put in pointers to specific PIC MCU features to help you understand what makes the PIC family of microcontrollers unique and which applications they are best suited for. You will probably find it useful to return to this chapter as you work through the book if a specific feature or aspect of the design of the PIC microcontrollers seems strange or illogical. There is probably a reason for the way something was done and if you can fully understand what it is doing, you will be best able to take advantage of it in your own applications.

Microcontroller Types

If you were to look at different manufacturer's products, you would probably be bewildered at the number of different devices that are out there and all their features and capabilities. I find it useful to think of the microcontroller marketplace having the three major subheadings:

- Embedded (self-contained) microcontrollers
- Microcontrollers with external support
- Digital signal processors

There is quite a wide range of embedded (self-contained) devices available. An embedded microcontroller has all the necessary resources—clocking, reset, input, and output (referred to as I/O)—available in a very low cost chip. In your application circuit, you don't have to provide much more than power (and this can be as simple as a couple of AA cells). The software for the computer processor built into the microcontroller is stored in nonvolatile (always available) memory that is also built into the chip. If you were to look at hobbyist and relatively simple electronic products designed in the 1970s and 1980s, you would discover a number of standard chips such as the 555 timer chip, whereas if you were to look at more modern designs, you would discover that they are based almost entirely on embedded microcontrollers. Embedded microcontrollers have become the new standard for these applications.

When you look at some of the more powerful microcontrollers, you might be confused as to the difference between them and microprocessors. There are a number of chips that are called “microcontrollers” (with typically 32-bit data and address paths) that require external memory and interface circuitry added to them so they can be used in applications. These chips are typically called microcontrollers because they have some of the built-in features of the embedded microcontrollers, such as a clock generator or serial interface, or because they have built-in interface circuitry to specific types of memory. Microcontrollers tend to require support circuitry for clocking and can have a very wide range of external interface and memory devices wired to them.

Digital signal processors (DSPs) are essentially very powerful calculators that execute a predetermined set of mathematical operations on incoming data. They may have built-in memory and interfaces, like the embedded microcontroller, or they may require a substantial amount of external circuitry. DSPs do not have the ability to efficiently execute conditionally; they are designed to run through the calculations needed for processing the formula needed to process an analog signal very quickly instead of responding to changing inputs. These formulas are developed from digital control theory and can require a lot of effort to develop for specific applications. There are DSPs that are completely self-contained, like an embedded microcontroller, or they may require external support chips.

If you were to look at the microcontroller applications contained within your PC, you would find that the embedded MCUs are used for relatively simple applications such as controlling the circuitry in the mouse. The disc drives use the more powerful microcontrollers, which can access large amounts of memory for data caching as well as have interfaces to the disc drive motors and read/write circuitry. The sound input and output probably pass through DSPs to provide tone equalization or break down speech input. If you look at other electronic devices around your house (such as your TV and stereo), you can probably guess which type of microcontroller is used for the different functions.

Internal Hardware

If you were to pull off the plastic packaging (called encapsulant) around a microcontroller to see the chip inside, you would see a rectangle of silicon similar to the one in Fig. 1.1, with each of the functions provided within the chip being visibly different from

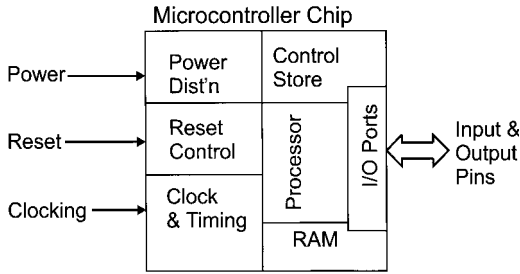


Figure 1.1 Block diagram with the basic features that can be expected in an embedded microcontroller.

the surrounding circuitry. The reason why you would be able to tell the function of each block is due to the specific circuitry used for each block; random processor logic looks different from neat arrays of memory circuits, and it looks different from the large transistors used for providing large current I/O functions.

Along with the basic circuitry presented in the block diagram of Fig. 1.1, most modern microcontrollers have many of following features built into the chips:

- Nonvolatile (available on power-up) program memory programming circuitry
- Interrupt capability (from a variety of sources)
- Analog input and output (I/O), both PWM and variable direct current (DC) I/O
- Serial I/O (synchronous and asynchronous data transfers)
- Bus/external memory interfaces (for RAM and ROM)
- Built-in monitor/debugger program

All these features increase the flexibility of the device considerably and not only make developing all applications easier, but allow the creation of applications that might not be possible otherwise. Most of these options enhance the function of different I/O pins and do not affect their basic operation, and they can usually be disabled, restoring the I/O pins function to straight digital input and output.

Most modern devices are fabricated using CMOS technology, which decreases the current chip's size and the power requirements considerably over early devices' reliance on NMOS or HexMOS technologies. For most modern microcontrollers, the current required is anywhere from a few microamperes (uA) in Sleep mode to up to about a milliampere (mA) for a microcontroller running at 20 MHz. A smaller chip size means that along with less power being required for the chip, more chips can be built on a single wafer. The more chips that are built on a wafer, the lower the unit price is.

Note that in CMOS circuitry, positive power is labeled "Vdd" and negative power or ground" is "Vss." This corresponds to TTL's "Vcc" and "Gnd" connections. This can be confusing to people new to electronics; in this book, I will be indicating power as being either positive (+) or at ground level and use the manufacturer's power pin labels in the schematics.

Maximum speeds for the devices are typically in the low tens of megahertz (MHz), with the primary limiting factor the access time of the memory built onto the chips. For the typical embedded microcontroller application, this is usually not an issue. What is

an issue is the ability to provide relatively complex interfaces for applications using simple microcontroller inputs and outputs. The execution cycles and the delay for software routines limit the MCU's ability to process complex input and output waveforms. Later in the book, I will discuss the advanced PIC microcontroller hardware features that provide interfacing functions as well as "bit-banging" algorithms for simulating the interfaces while still leaving enough processor cycles to provide the other application operations required.

Despite the tremendous advantages that a microcontroller has with built-in program storage and internal variable RAM, there are times (and applications) where you will want to add external (both program and variable) memory to your microcontroller. There are three basic ways of doing this. The first is to add memory devices to the microcontroller as if it were a microprocessor. Many microcontrollers are designed with built-in hardware to access external devices like a microprocessor (with the memory interface circuitry added to the chip as shown in Fig. 1.2) with the classic example of this being the Intel 8051. A typical application for a microcontroller with external memory is as a hard disk cache/buffer that buffers and distributes large amounts of data. The 8051's bus designs of the 8051 allows the addition of up to 64K as well as 64K variable RAM. An interesting feature of the 8051 is that internal nonvolatile memory can be disabled, allowing the 8051 chip to be used even if it was programmed with incorrect or "downlevel" programs.

The second method of adding external memory is to simulate microprocessor bus operations with the chip's I/O pins. This method tends to be much slower than having a microcontroller that can access external devices directly, like the 8051. While it is not recommended to simulate a microprocessor bus for memory devices, isn't unusual to see a microcontroller simulating a microprocessor bus to allow access to a specialized peripheral I/O chip. There are cases where a specific chip will provide exactly the function needed and it is designed to be controlled by a microprocessor.

The last method is to use a bus protocol that has been designed to provide additional memory and I/O capabilities to microcontrollers. The two wire "inter-inter computer"

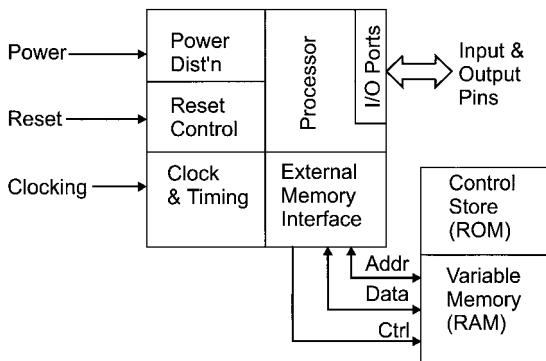


Figure 1.2 Block diagram of a microcontroller with built-in circuitry to access external memory devices.

(I²C) protocol is a very commonly used bus standard that provides this capability. This standard allows I/O devices and multiple microcontrollers to communicate with each other without complex bus protocols.

Applications

In this book, I use the term “application” to collectively describe the hardware circuitry and software required to develop a microcontroller-based circuit. I think it is important to note that a microcontroller project is based on multiple development efforts (for circuitry and software) and not the result of a single discipline. In this section, I will introduce you to the five elements of a microcontroller project and explain some of the terms and concepts relating to them.

The five aspects of every microcontroller project are:

- Microcontroller and support circuitry
- Project power
- Application software
- User interface (UI)
- Device input/output (I/O)

These elements are shown working together in Fig. 1.3.

The microcontroller with its internal features (processor, clocking, variable memory, reset/support, and application program memory) is simply the complete embedded microcontroller chip. Other than the chip itself, most microcontroller circuitry just requires power along with a decoupling capacitor and often a reset circuit and an oscillator to run. The design of the PIC MCU (as with most other microcontrollers) makes the specification of power and external parts almost trivial; chances are, other than power and a decoupling capacitor, you will not require any other parts to support the embedded microcontroller in the application.

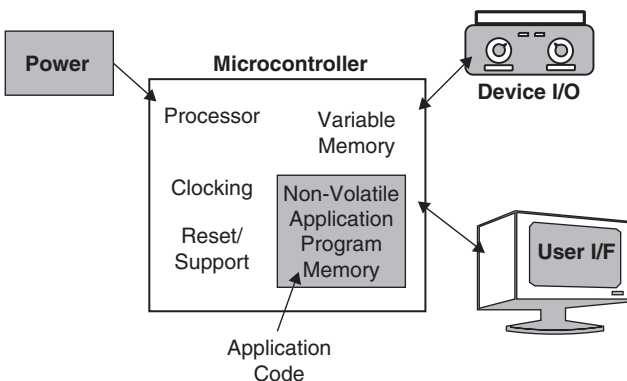


Figure 1.3 Embedded microcontroller application block diagram showing five development project aspects.

In the second edition of this book, I took a fair amount of effort to ensure that the voltage levels of the power applied to the PIC MCUs were within relatively narrow ranges. Most new PIC MCUs (as well as other manufacturers' chips) are now able to run within a surprisingly wide range of voltages (from 2 to 6 volts), which will allow you to use simple alkaline batteries and dispense with voltage regulators for most applications.

A decoupling capacitor—usually 0.01 μF to 0.1 μF connected across positive power (Vdd) and ground (Vss)—should always be wired to the power connection of each chip in your application circuitry, with one pin as close to the positive power input pin as possible. Decoupling capacitors are used to minimize the effects on the chips of rapid changes in power levels and current availability caused by other chips in the circuit switching and drawing more power. A decoupling capacitor can be thought of as a filter that smoothes out the rough spots of the power supply and provides additional current for high-load situations on the part. As I will show later in the book, having a decoupling capacitor is critical with the PIC MCU and should never be left out of an application's circuit.

The purpose of the reset circuit is to hold the processor within the microcontroller until it can be reliably assumed that the input power has reached an acceptable level for the chip to run and any initial oscillations have completed. Many embedded microcontrollers (including different PIC MCU part numbers) provide the reset circuitry internally or they can be as simple as just a “pull-up” (resistor connected to positive power). The reset circuitry can become more complex, providing the capability of holding the microcontroller reset if power “drips” below a certain point (often called “brown out”). For most applications, the reset circuitry of an embedded microcontroller can be very simple, but when the operation of the device is critical, care must be taken to ensure the microcontroller will only operate when power and other conditions are within specific parameters.

For any computer processor to run, it requires a clock to provide timing for each instruction operation. This clock is provided by an oscillator built into the PICmicro, which uses a crystal, ceramic resonator, or an RC oscillator to provide the time base of the PICmicro's clocks circuitry. Many modern microcontrollers have built-in RC oscillators to provide the basic clock signal for the application. When you are first starting to learn about embedded microcontrollers, a nice feature is the built-in oscillator, as adding a crystal or ceramic resonator can be a bit finicky and will give you an additional variable to check if your circuit doesn't seem to be running.

The user interface is critical to the success of a microcontroller application. In this book, I will be showing you number of ways of passing data between a user and a PIC microcontroller. Some of these methods may seem frivolous or trivial, but having an easy to use interface between your application and the user is a differentiator in today's marketplace. Along with information on working with different user I/O circuitry and devices, I will also be giving you some of my thoughts on the philosophy of what is appropriate for users.

Device I/O is really what microcontroller applications are all about. The I/O pins can be interfaces to strictly logic devices, analog signals, or complex device interfaces.

Looking over the “Projects” chapter, you should get the idea that there is a myriad of devices that microcontrollers can interface with to control or monitor. I have tried to present a good sampling of devices to show different methods of interfacing to the PICmicro that can be used in your own applications.

Within the microcontroller is the application code stored in application program memory, which is the computer program used to control the operation of the application. The word “code” is often used as a synonym for “program.” While this is one-fifth of the elements that make up a microcontroller application, it will seem like it requires six-fifths of the work. Microcontroller application software development is more an art than a science, and I will present information in this book that should give you a good basis for developing your own applications. In addition, you will find code snippets that you can add to your own applications and methodologies for finding and fixing problems in the application code.

Processor Architectures

Here’s a hint when you are inviting computer scientists to dinner: make sure they all agree on what is the best type of computer architecture. There are a variety of strong points for supporting the options that are available in computer architectures. While RISC is in vogue right now, many people feel that CISC has been unfairly maligned. This is also true for proponents of Harvard over “Princeton” computer architectures and whether a processor’s instructions should be hard-coded or microcoded. Trust me when I say if you don’t type your guests properly, you will have a dinner with lots of shouting, name calling, and bun throwing.

The following sections will give you some background on the various processor types, explain feature advantages and disadvantages, and help you understand why engineers made some choices over others when specifying and designing a microcontroller’s processor. They are not meant to provide you with a complete understanding of computer processor architecture design, but should help explain the concepts behind the buzz words used in microcontroller marketing materials.

CISC VERSUS RISC

Many processors are called RISC (reduced instruction set computers, pronounced “risk”), as there is a perception that RISC is faster than CISC (complex instruction set computers) because the instructions they execute are small and tailored to specific tasks required by the application. CISC instructions tend to be large and perform functions that the processor designer believes will be best suited for the applications they will be used for. When choosing a microcontroller for a specific application, you will be given the choice between RISC, RISC-like, and CISC processors.

There is no definitive correct answer to the question of which is better. There are applications in which either one of the design methodologies is more efficient. A well-designed

RISC processor has a small instruction set, which can be very easy to memorize. A CISC instruction set provides high level functions that are easy to implement and do not require the programmer to be intimately familiar with the processor's architecture. In terms of high level language compilers, there are equally sophisticated tools available on the market for either one. Both allow complex applications to be written for them. For new programmers, a CISC processor will be easier to code, but for an experienced programmer, a RISC processor will actually be easier to create complex code. Proponents of the methodologies will push different advantages, but when you get right down to it neither is substantially better than the other.

Personally, I prefer a RISC processor with the ability to access all the registers in a single instruction. This ability to access all the registers in the processor as if they were the same is known as orthogonality and provides some unexpectedly powerful and flexible capabilities to applications. The PIC microcontroller's processors are orthogonal, and as I go through the PICmicro architecture, instructions, and applications in the following chapters, you will see that fast data processing operations within the processor can be very easily implemented in a surprisingly small instruction set.

HARVARD VERSUS PRINCETON

In the 1940s, the United States government asked Harvard and Princeton Universities to come up with a computer architecture to be used in computing tables of naval artillery shell distances for varying elevations and environmental conditions. Princeton's response was for a computer that had common memory for storing the control program as well as variables and other data structures. It was best known by the chief scientist's name, John Von Neumann. Fig. 1.4 is a block diagram of the architecture. In contrast, Harvard's response was a design (shown in Fig. 1.5) that used separate memory banks for program storage, the processor stack, and variable RAM. The Princeton architecture won the competition because it was better suited to the technology of the time; a single

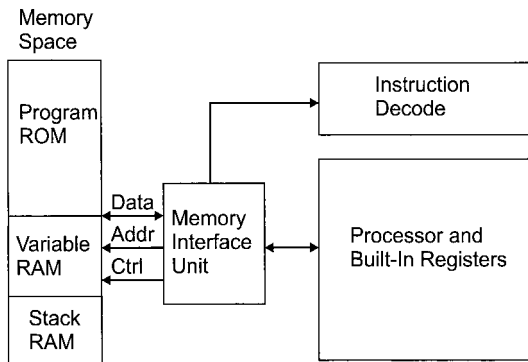


Figure 1.4 Princeton computer architecture block diagram.

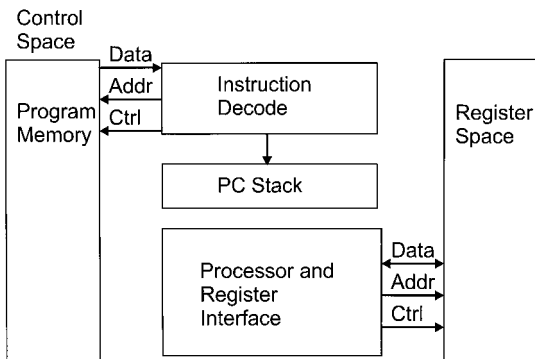


Figure 1.5 Harvard computer architecture block diagram.

memory space was preferable because of the unreliability of the current electronics (this was before transistors were even invented) and the simpler interface would have fewer parts that could fail.

The Princeton architecture's memory interface unit is responsible for arbitrating access to the memory space between reading instructions and passing data back and forth to the processor. This hardware is something of a bottleneck between the processor's instruction processing hardware and the memory accessing hardware. In many Princeton-architected processors, the delay is reduced because much of the time required to execute an instruction is normally used to fetch the next instruction (this is known as pre-fetching). Other processors (most notably the Pentium processor in your PC) have separate program and data caches that pass data directly to the appropriate area of the processor while external memory accesses are taking place.

The Harvard architecture was largely ignored until the late 1970s when microcontroller manufacturers realized that the architecture did not have the instruction/data bottleneck of the Princeton architecture-based computers. The dual data paths give Harvard architecture computers the ability to execute instructions in fewer instruction cycles than the Princeton architecture due to the instruction parallelism possible in the Harvard architecture. Parallelism means that instruction fetches can take place during previous instruction execution and not wait for either a dead cycle of the instruction's execution or have to stop the processor's operation while the next instruction is being fetched.

After reading this description of how data is transferred in the two architectures, you probably feel that a Harvard-architected microcontroller is the only way to go. But the Harvard architecture lacks the flexibility of the Princeton in the software required for some applications that are typically found in high-end systems such as servers and workstations. The Harvard architecture is really best for processors that do not process large amounts of memory from different sources (which is what the Von Neumann architecture is best at) and have to access this small amount of memory very quickly. This feature of the Harvard architecture (used in the PIC microcontroller's processor) makes it well suited for microcontroller applications.

MICROCODED VERSUS HARDWIRED PROCESSORS

Once the processor's architecture has been decided upon, the design of the architecture goes to the engineers responsible for implementing the design in silicon. Most of these details are left under the covers and do not affect how the application designer interfaces with the application. There is one detail that can have a big effect on how applications execute and that is whether the processor is a hardwired or microcoded device. The decision between the two types of processor implementations can have significant implications as to the ease of design of the processor, when it is available, and its ability to catch and fix mistakes.

Each processor instruction is in fact a series of instructions that are executed to carry out the larger, basic instruction. For example, to load the accumulator in a processor, the following steps need to be taken:

- 1 Output address in instruction to the data memory address bus drivers.
- 2 Configure internal bus for data memory value to be stored in accumulator.
- 3 Enable bus read.
- 4 Compare data values read from memory to zero or any other important conditions and set bits in the STATUS register.
- 5 Disable bus read.

Each of these steps must be executed in order to carry out the basic instruction's function. To execute these steps, the processor is designed to either fetch this series of instructions from a memory or execute a set of logic functions unique to the instruction.

A microcoded processor is really a processor within a processor. In a microcoded processor, a state machine executes each instruction as the address to a subroutine of instructions. When an instruction is loaded into the instruction holding register, certain bits of the instruction are used to point to the start of the instruction routine (or microcode) and the μ Code instruction decode and processor logic executes the microcode instructions until an instruction end is encountered as shown in Fig. 1.6.

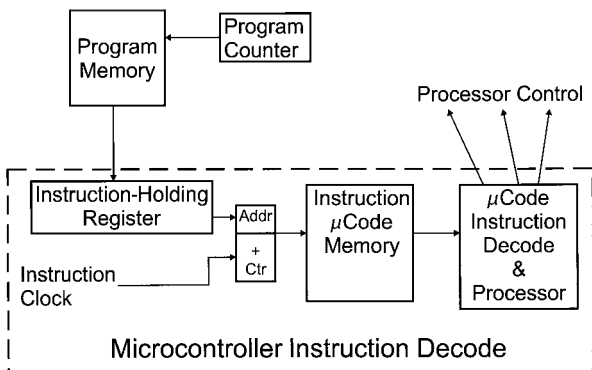


Figure 1.6 Microcoded processor with memory storing individual instruction steps.

I should point out that having the instruction holding register wider than the program memory is not a mistake. In some processors, the program memory is only 8 bits wide although the full instruction may be some multiple of this (for example, in the 8051 most instructions are 16 bits wide). In this case, multiple program memory reads take place to load the instruction holding register before the instruction can be executed.

The width of the program memory and the speed with which the instruction holding register can be loaded into is a factor in the speed of execution of the processor. In Harvard-architected processors, like the PICmicro, the program memory is the width of the instruction word and the instruction holding register can be loaded in one cycle. In most Princeton-architected processors, which have an 8-bit data bus, the instruction holding register is loaded through multiple data reads.

A hardwired processor uses the bit pattern of the instruction to access specific logic gates (possibly unique to the instruction) that are executed as a combinatorial circuit to carry out the instruction. Fig. 1.7 shows how the instruction loaded into the instruction holding register is used to initiate a specific portion of the execution logic that carries out all the functions of the instruction.

Each of the two methods offers advantages over the other. A microcoded process is usually simpler than a hardwired one to design and can be implemented faster with less chance of having problems at specific conditions. If problems are found, revised stepings of the silicon can be made with a relatively small amount of design effort. An example of the quick and easy changes that microcoded processors allow was a number of years ago when IBM wanted to have a microprocessor that could run 370 assembly language instructions. Before IBM began to design their own microprocessor, they looked around at existing designs and noticed that the Motorola 68000 had the same hardware architecture as the 370 (although the instructions were completely different). IBM ended up paying Motorola to rewrite the microcode for the 68000 and came up with a new microprocessor that was able to run 370 instructions much more quickly and at a small fraction of the cost of developing a new chip.

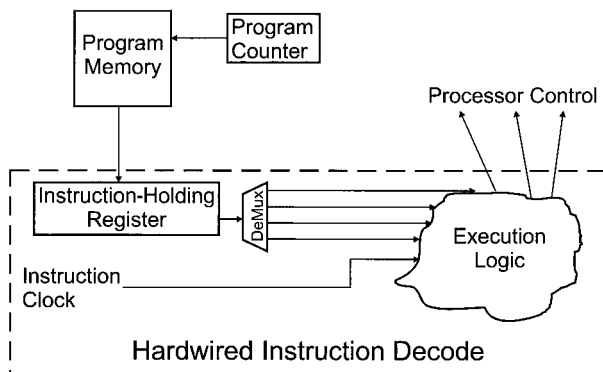


Figure 1.7 The hardwired processor generates each individual instruction step from execution logic arrays.

A hardwired processor is usually a lot more complex because the same functions have to be repeated over and over again in hardware—how many times do you think a register read or write function has to be repeated for each type of instruction? This means the processor design will probably be harder to debug and less flexible than a micro-coded design, but instructions will execute in fewer clock cycles.

This brings up a point you are probably not aware of. In most processors, each instruction executes in a set number of clock cycles. This set number of clock cycles is known as the processor's instruction cycle. Each instruction cycle in the PIC microcontroller family of devices takes four clock cycles. This means that a PIC MCU running at 4 MHz is executing the instructions at a rate of 1 million instructions per second.

Using a hardwired over microcoded processor can result in some significant performance gains. For example, the original 8051 was designed to execute one instruction in 12 cycles. This large number of cycles requires a 12 MHz clock to execute code at a rate of 1 MIPS (million instructions per second) whereas a PIC microcontroller with a 4 MHz clock gets the same performance.

Instructions and Software

It is amazing that, in a tiny plastic package, there is a chip that can perform basic input and output functions, with a full computer processor along with memory storing the full application code and variable data areas built on it as well. (In the next chapter, you will get an idea of what “tiny” means when the different PIC microcontroller chip packages are described.) The microcontroller's computer processor has essentially all the capabilities of the processor in your desktop PC, although it cannot handle as much or as large data as the PC. The microcontroller's processor executes a series of basic instructions that make up the application software, which controls the circuitry of the application.

When a computer processor executes each individual program instruction, it is reading a set of bits from program memory and decoding them to carry out specific functions. Each instruction bit set carries out a different function in the processor. A collection of instructions is known as a program. The program instructions are stored in memory at incrementing addresses and are referenced using a program counter to pull them out sequentially. After each instruction is executed, the program counter is incremented to point to the next instruction in program memory.

There are four types of instructions:

- Data movement
- Data processing
- Execution change
- Processor control

The data movement instructions move data or constants to and from processor registers, variable memory and program memory (which in some processors are the same thing), and peripheral I/O ports. There can be many types of data movement instructions

based on the processor architecture, number of internal addressing modes, and the organization of the I/O ports.

The five basic addressing modes (which are available in the PIC microcontroller and will be explained in greater detail in later chapters) move data to or from the registers or program memory. If you are familiar with the Intel processors in PCs, you will know that there are two memory areas: data and registers. The data area stores program instructions and variable data, while the register area is designed to be used for I/O registers. The addressing modes available to a processor are designed to efficiently transfer data between the different memory locations within the computer system.

In the PIC microcontroller's processor (and other microcontrollers that use Harvard-architected processors) there are also two memory areas, but they are somewhat different from that of a PC and consist of program memory and registers. The program memory is loaded exclusively with the program instructions and, except in certain circumstances, cannot be accessed by the processor. The registers consist of the processor and I/O function registers along with the microcontroller's variable data (which are called file registers in the PIC microcontroller). The five addressing modes available in the PIC MCU allow data to be transferred between registers only. They are:

- Immediate (or literal) values stored in the accumulator register
- Register contents stored in the accumulator register
- Indexed address register contents stored in the accumulator register
- Accumulator register contents stored in a register
- Accumulator register contents stored in an indexed address register

These five addressing modes are very basic and when you research other processor architectures, you will find that many devices can have more than a dozen ways of accessing data within the memory spaces. The five methods above are a good base for a processor and can provide virtually any function that is required of an application. The most significant missing addressing mode is the ability to access data in the program counter stack. This addressing mode, along with the other five, is available in the high-end PIC microcontroller chips.

The data processing instructions are the arithmetic and bitwise data manipulation operations available in the processor's arithmetic/logic unit. A typical processor will have the following data processing instructions:

- Addition
- Subtraction
- Incrementing
- Decrementing
- Bitwise AND
- Bitwise OR
- Bitwise XOR
- Bitwise negation

These instructions work the number of bits that is the data word size (the PIC MCU has an 8-bit word size). Many processors are capable of carrying out multiplication, division and comparison operations on data types of varying sizes, as well as logarithmic and trigonometric operations. For most microcontrollers, such as the PIC microcontroller, the word size is 8 bits and advanced data processing operations are not available.

Execution change instructions include branches, gotos, skips, calls, and interrupts. For branches and gotos, the new address is specified as part of the instruction. Branches and gotos are similar except that branches are used for short jumps that cannot access the entire program memory and are used because they take up less memory and execute in fewer instruction cycles. Gotos give a program the ability to jump to a new location anywhere in the processor's instruction address space.

Branches and gotos are generally known as "nonconditional" because they are always executed when encountered by the processor. There can be conditional branches or gotos and in some processors conditional skips are available. Skips are instructions that will skip over the following instruction when a specific condition is met. The condition used to determine whether or not a branch, goto, or skip is to execute is often based on a specific status condition.

If you have developed applications on other processors, you may interpret the word "status" to mean the bits built into the ALU STATUS register. These bits are set after an arithmetic or bitwise logical instruction to indicate such things as whether or not the result was equal to zero, was negative, or caused an overflow. These status bits are available in the PIC microcontroller, but are supplemented by all the other bits in the processor, each of which can be accessed and tested individually. This provides a great deal of additional capabilities in the PIC MCU that is not present in many other devices and allows some amazing improvements in processor performance.

An example of using conditionally executing status bits is shown in the 16-bit variable increment example below. After incrementing the lower 8 bits, if the processor's zero flag is not set (which indicates that the incremented register's contents have changed from 0xFF to 0x00), then the increment of the higher 8 bits is skipped. But if the result of the lower 8-bit increment is equal to zero, then the skip instruction doesn't execute and the upper 8-bit increment is executed.

```

Increment                LowEightBits
SkipIfNotZero
  Increment              HighEightBits

```

The skip is used in the PIC microcontroller to provide conditional execution, which is why it is described in detail here. The skip instructions can access *every* bit in the PIC MCU's register space, making it a very powerful instruction, as will be described below.

Other execution change instructions include call and interrupt, which causes execution to jump to a routine and return back to the instruction after the call/interrupt instruction. A call is similar to a branch or goto and has the address of the routine to jump to included in the instruction. The address after the call instruction is saved and when a return instruction is encountered, this address is used to return execution to the software that executed the original call instruction.

There are two types of interrupts. Hardware interrupts are explained in more detail in the next section. Software interrupts are instructions that are similar to subroutine calls, but instead of jumping to a specific address, they make calls to predefined interrupt handler routines. The advantage of software interrupts over subroutine calls is their ability to provide systemwide subroutine functions without having to provide the addresses to subroutines to all the programs that can run within it. Software interrupts are not often used in smaller microcontrollers, but they are used to advantage in the IBM PC.

Rather than providing instructions that immediately change the program counter (and where the program is executing), it can be advantageous to be able to arithmetically create a new address and load new values directly into the program counter registers. In most processors, the program counter cannot be accessed directly, to jump or call arbitrary addresses in program memory. The PIC microcontroller architecture is one of the few that does allow the program to access the program counter's registers and change them during program execution. This capability adds a great deal of flexibility and efficiency in programming (which will be discussed later in the book), however, care must be taken when updating the processor's PC to make sure the correct address is calculated before it is updated.

Processor control instructions are specific and control the operation of the processor. One common processor control instruction is sleep, which puts the processor (and microcontroller) into a low-power mode. Another processor control instruction is the interrupt mask, which stops hardware interrupt requests from being processed. These instructions are often very device specific and cannot be counted upon to be present when you move to a new microcontroller family.

HARDWARE INTERRUPTS

Properly used, hardware interrupts can greatly improve the efficiency of your applications as well as simplify your application code. Despite these potential advantages, they are seldom used and often avoided as much as possible. For many application developers, interrupts are perceived as being difficult to work with and something that complicates the application code and its execution. This perception isn't accurate if you follow the basic rules that will be discussed in this book.

Hardware interrupts in computer systems are analogous to interrupts in your everyday life. As the computer processor is executing application code, a hardware event may occur that will request the processor to stop executing and respond to or handle the hardware event. Once the processor has responded to the event, the regular program execution can continue where it was stopped. The hardware event requesting the interrupt can be a timer overflow, a serial character received (or finished sending), a user pressing a button, and so on. There are many different hardware events that will cause an interrupt to take place—similar to you getting a phone call or other distraction while working. Like a phone call giving you new information, the application code often uses the information provided by the interrupt as new data to consider during execution.

Possible hardware interrupt requests that you will have to consider responding to in your microcontroller applications include such situations as changing digital inputs, the completion of an analog-to-digital conversion, the receipt of a serial character, and so

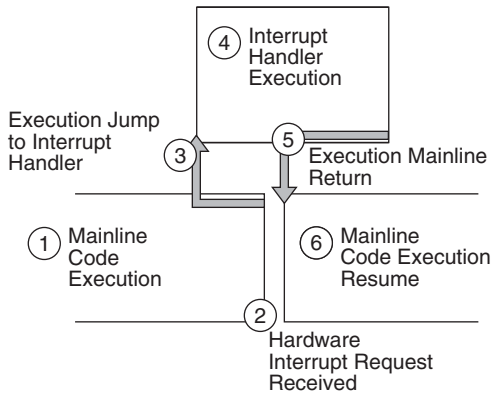


Figure 1.8 The steps taken when a hardware event requests that the execution of the application is interrupted to respond to it.

on. When sending a string of data, you may use interrupts to load in the next bit or byte to be output without affecting the primary application’s execution. In any case, it is important to quickly respond to these requests and store the new information as quickly as possible to avoid negatively affecting how the application runs.

A good rule of thumb is to code your applications so the data provided by hardware interrupts is in as simple a form as possible and reading it is as simple as reading a byte or a bit.

The process of responding to a hardware interrupt request follows the six distinct steps outlined in Fig. 1.8. If a hardware interrupt request is received while the primary application (or mainline) code is executing (1 in Fig. 1.8), the processor continues executing the current instruction and then tests to see if interrupt requests are allowed. Hardware interrupt requests do not have to be responded to immediately or at all. This is an important point because an application may ignore interrupt requests if time sensitive or high priority code is being executed. If the request is ignored, the hardware will continue requesting until the application code enables the processor circuitry that responds to interrupts. This is analogous to you ignoring a phone call and listening to a message later because you were doing something that you considered more important.

If the processor can respond to a hardware interrupt request, execution of the mainline code is stopped (2 in Fig. 1.8) and the current program counter and other important data is saved until the interrupt response has completed and execution returns to where it was stopped. The important data is often called the context data or context information, and consists of the contents of the registers that were being used by the mainline code when it was interrupted. This context information may be saved automatically by the processor or require special code to save and retrieve it. The PIC microcontroller requires special code to save and retrieve the context data. With the return address saved, the processor then changes the program counter to the interrupt handler vector (3 in Fig. 1.8). The interrupt handler (4 in Fig. 1.8) is the subroutine-like code that processes the data from the interrupting hardware and stores it for later use. You may see terms like “interrupt service routine” in some references instead of interrupt handler, but they both mean the same thing.

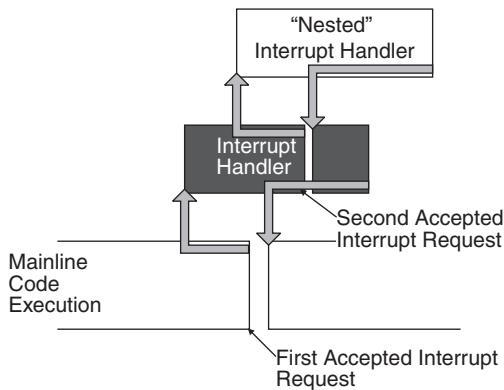


Figure 1.9 Nested interrupt requests occur when interrupts are responded to while other interrupt handlers are active.

The interrupt handler vector is a program memory address that points to the start of the interrupt handler. After the interrupt handler code is finished (5 in Fig. 1.8), the hardware interrupt has been acknowledged and the hardware reset to request another interrupt when the condition happens again. The mainline's context information is restored, the saved address where the mainline was interrupted is loaded into the program counter, and the mainline code execution resumes just as if nothing had happened.

Most high level language compilers (such as HI-TECH PICC-Lite, discussed in this book) provide the ability to create interrupt handlers that are based on a subroutine model and eliminate the need for you to fully understand the mechanics of creating an interrupt handler. The interrupt handler routines produced take care of the interrupt handler vector and context information storage so you can concentrate on designing the interrupt handler.

In some processors, you have the ability to acknowledge a new interrupt while still handling another one. This is known as nesting interrupts (Fig. 1.9) and is generally only done when there are hardware interrupts of such high priority that they supersede the response to *other* interrupts. Creating application code that allows response to nested interrupts is generally not trivial, and in the PIC microcontroller architectures it is very difficult to implement successfully.

Peripheral Functions

All microcontrollers have built-in I/O pins that allow the microcontroller to access external or peripheral devices. The hardware built into these pins can range from I/O pins consisting of just a pull-up resistor and a transistor to full Ethernet interfaces or video on-screen display functions that require just a few high level commands. The capabilities of the I/O pins define the peripheral functions the microcontroller can perform and what applications a manufacturer's part or a specific part number is best suited for. Along with memory size, the peripheral functions of a microcontroller are the most important characteristics used to select a device for a specific application.

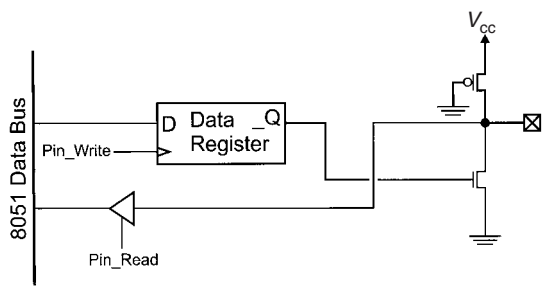


Figure 1.10 The 8051 bidirectional input/output pin consisting of a pull-up resistor and transistor.

I wasn't being facetious when I said an I/O pin could be as simple as a transistor and a pull-up resistor. The Intel 8051 uses an I/O pin that is this simple, as is shown in Fig. 1.10. This pin design is somewhat austere and is designed to be used as an input when the output is set high so another driver on the pin can change the pin's state to high or low easily against the high impedance pull-up. When used as an output, this design of I/O pin can only sink (pass to ground) current effectively, it cannot be used to source (pass current from positive power) current.

A more typical I/O pin is shown in Fig. 1.11 and provides "tristatable" output from the control register. This pin can be used for digital input as well (with the output driver turned off). When the output driver is enabled, the pin can both sink and source current to external devices. This design of I/O pin is used in the PIC microcontroller; later in the book I will explain the operation of the I/O pins in greater detail.

A microcontroller may also have more advanced peripheral functions built into its I/O pins, such as the ability to send and receive serial I/O that will allow communication with a PC via RS-232. These peripheral functions are designed to simplify the interfacing to other devices. How functions are programmed in a microcontroller is half the battle in understanding how they are used; along with changing the function of an I/O pin, they may also require other features (such as a timer or the microcontroller's interrupt

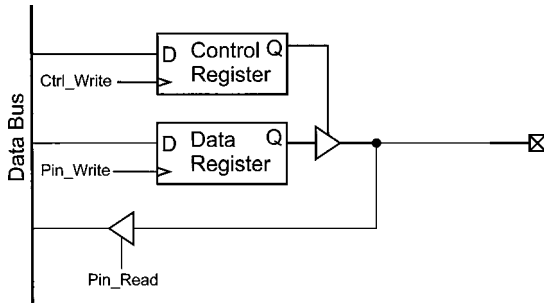


Figure 1.11 The tristate driver on this I/O pin can sink and source current as well as work as a digital input.

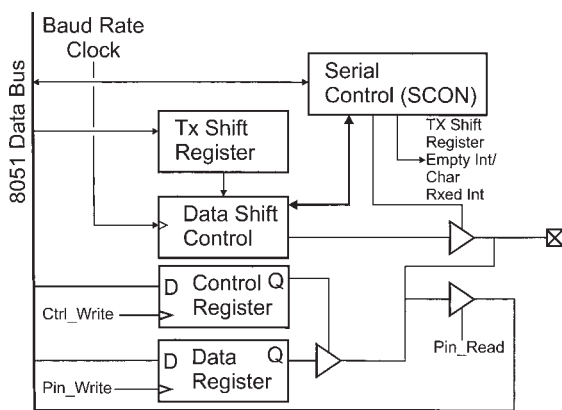


Figure 1.12 An I/O pin that can be used for sending serial data is not only more complex but requires other resources within the microcontroller.

controller). Fig. 1.12 shows the block diagram of an I/O port that can be used for digital I/O as well as transmitting serial data—note that there are a number of external resources required to implement this function.

While most peripheral functions can issue hardware interrupt requests, you don't have to use this feature in your applications. Often a single flag bit can be read or polled in the mainline to determine whether the peripheral function has new information for the application to respond to. Along with hardware interrupts, advanced peripheral functions built into a microcontroller's I/O pins provide you with additional options for your applications.

BIT-BANGING I/O

Despite the plethora of peripheral features available in microcontrollers, there will be situations where you want to use peripheral functions that are not available or the built-in features are not designed to work with the specific hardware you want to use in the application. These functions can be provided by writing code that executes the desired I/O operations using the I/O pins of the microcontroller. This is known as "bit-banging," and the practice is very common for microcontroller application development.

There are two philosophies behind the methods used to provide bit-banging peripheral functions. The first is to carry out the operation directly in the execution line of the code and suspend all other operations. An example of this for a serial receiver is shown below:

```
Int SerRXWait() {                               // Wait for and Return the
Next                                                  // Asynchronous Character
int i;
int OutByte;

while (High == IP_Bit); // Wait for the "Start" Bit
```

```
HalfBitDelay();           // Delay to Middle of Bit

for (i = 0; i < 8; i++) { // Read the 8 Bits of Data

    BitDelay();           // Delay one Bit Period

    OutByte = (OutByte >> 1) + (IP_Bit << 7);

}

return OutByte;           // Return the Byte Read In
} // End SerRXWait
```

The advantage of this method is that it is relatively easy to code, but the downside is that it requires all other operations in the microcontroller to stop. The serial receive function above waits literally forever for data to come in. While the function is waiting or receiving a character, nothing else can execute in the microcontroller.

The other method of providing bit-banging functions is to periodically interrupt the main-line execution to provide the peripheral function. To do this, the timing relationships of the peripheral function have to be well understood. For the serial receive function, a bit-banging interface could be implemented using a timer interrupt at three times the incoming bit speed. This code will start reading the incoming data when a start bit is detected. After the data byte has been received, it is stored for later use by the mainline code, just as a byte that was received in a specialized serial receiver pin function would be.

```
Interrupt IntSerRX() {

    Reset(Timer);           // Reset Interrupt
    Requesting H/W
    if (startRX != rSTATUS) // Is Something Being
    Received?
        If (IP_Bit == Low) { // No - Check for Start Bit
            rSTATUS = startRX; // Start Bit Found
            dlayCount = 4;     // Wait four Timer
            Days to middle    // of 2nd Bit
                                // Eight Bits are Read
            bitCount = 8;
        } else;
    else // Reading a Byte
        if (--dlayCount == 0) { // If Bit Delay is
    Finished
        OutByte = (OutByte >> 1) + (IP_Bit << 7);
        dlayCount = 3;

        if (--BitCount == 0) // Read all 8 Bits?
            rSTATUS = byteRX;
        }

} // End Serial RX Interrupt Handler
```

While this function is operating as a periodic interrupt, it is taking processor cycles away from the mainline code. But the overall percentage of lost cycles is very low—it will probably only use 1 or 2 percent of the total execution cycles available in the microcontroller. For this reason, I prefer it to the inline bit-banging peripheral functions.

The timer serial interrupt handler code probably seems quite complex and at first glance is just about impossible to understand exactly how it works. I will explain the theory behind the function and how it is implemented in the PIC microcontroller in more detail later in the book. What I wanted to show now was a bit-banging function that does not prevent other microcontroller operations from being carried out while it is operating.

Memory Types

Memory is probably not something you normally think about when you create applications for a personal computer. The memory available for an application in a modern Microsoft Windows PC can be up to 4.3 gigabytes (GB) in size and can be swapped in and out of memory as required. Few people have PCs with this much memory, and even if they did, they would find that all the potential programs they could run on it would take up more than this amount of space. Fortunately, in a PC you can store programs and data on a disk drive and access them as required. This eliminates the need to manage how software and data are stored and accessed on the computer and makes it easy for the casual user to work with a PC.

A small embedded microcontroller, like the ones discussed in this book, does not have the capability to control a disk drive or the user interface to load and execute applications. When you create an application for an embedded microcontroller, you will have to know how much memory (of different types) is available in the microcontroller and how the program and data are to be stored on the chip. For the most part, this is not difficult, but you will encounter circumstances where you find that you are running out of memory and either have to redesign your application or select another device to put the application on. While it may seem to be a bit of a burden when you start working with microcontrollers, it will very quickly become second nature and allow you to further customize your application to best suit the device you have chosen.

There are two or three types of memory that are provided in embedded microcontrollers:

- Nonvolatile program memory
- Volatile variable memory
- Optional nonvolatile data memory

Program memory is known by a number of different names, including control store and firmware (as well as some permutations of these names). The name really isn't important, as long as you understand that this memory space is used to store the application software. The adjective "nonvolatile" describes the ability of memory to retain the information stored in it even when power is removed. This is important because each time power is applied to the microcontroller, the application code should start working. The program memory space is the maximum size of application that can be loaded into

the microcontroller and contains all the code that is executed in an application along with the initial values for the variables used in the application. Program memory is not generally changed during program execution, and the application code is stored in it using custom chip programming equipment.

The variable memory available in an embedded microcontroller consists of a fairly small amount of RAM (random-access memory), which is used for the temporary storage of data. Variable memory is volatile, which means that its values will be lost when power is removed from the microcontroller. When the processor addressing modes were discussed earlier, they were primarily referring to accessing the variable memory of a microcontroller. It is important to remember that application execution does not take place in variable memory. While in Princeton-architected microcontrollers, it is possible there is no simple way of loading the memory with code when the device starts up other than having software in the main program write initial values to the variables.

The nonvolatile data memory provides long-term storage of information even when power is lost. Typical information stored in this memory includes data logging information for later transmittal to another device, calibration data for different peripherals, and IP address information for networked devices.

With an idea of how applications execute in an embedded microcontroller, you can look at how it is actually implemented on the chip. The nonvolatile program memory will probably be some flavor of read-only memory (ROM), called this because during execution the processor can only read from this memory, not write new information into it. In the PIC microcontroller, there are four types of program memory available in devices and applications: none (external ROM), mask ROM, EPROM, and EEPROM/Flash. While these four types of nonvolatile memory options all provide the same function—memory for the processor to read and execute—they each have different characteristics and are best suited for different purposes.

“None” probably seems like a strange option, but in the high-end PICmicros running in microprocessor mode, it is a very legitimate one. With no internal program memory, the device has to be connected to an external ROM chip, as can be seen in Fig. 1.13. The external ROM feature is primarily used when more application program memory is required or applications and data are to be loaded into RAM while the application is running.

There are microcontrollers available with the traditional type of read-only memory program memory although they are becoming increasingly rare. This type of read-only memory consists of memory cells that can be configured as either a one or a zero by not

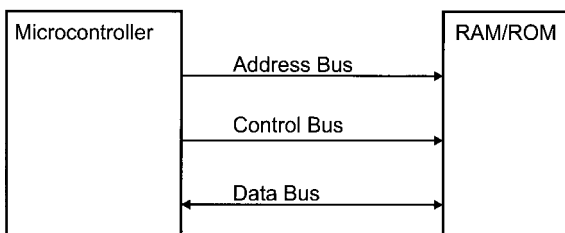


Figure 1.13 External memory connections to a microcontroller are wired similarly to that of a microprocessor.

etching the last metal layer during the wafer manufacturing process. When an order comes in for a batch of microcontrollers with a ROM with a customer-specified application, these wafers are pulled from stock and the last metal layer is exposed to a custom mask made from the customer-supplied software program, which makes the connections to the memory cells that turns them into ones or zeros. This is known as mask ROM programming. With the program put into the chip, the customer will have a device they can use in their product without having to load a program into it later. ROM contents typically cannot be read out of the microcontroller to thwart others trying to pirate or reverse engineer the product.

There are some significant downsides to buying microcontrollers with mask ROM. The first two are the cost and lead time required to have the customized chips built. While the actual piece price of a ROM program memory chip is less than a device with a customer (or field) programmable program memory, the nonrecurring expense (NRE) costs of getting the mask made makes this process cost effective in lot sizes of 10,000 or more chips. The lead time for getting mask ROM devices built is typically on the order of six to ten weeks. For certain applications, such as for the automotive market, the downsides of mask ROM microcontrollers do not take away from the cost advantages; here, the parts are ordered well in advance and with one or more per vehicle, a large guaranteed order is assured.

It should be obvious that going straight to mask ROM for a product or project is not an efficient method of finding out whether the program works. To provide a method of loading a program into a device outside the factory in short order, programmable read-only memory (PROM) was invented. The most popular form of PROM is known as fuseable link, in which high current is optionally passed through small metal connections to burn them out and cause the memory cell they are associated with to be programmed to a one or zero. These chips fell out of favor for two reasons: the part can only be used once and cannot be reprogrammed, and after a period of time some of the links will “regrow” back, changing the value of the cell (and ruining the program contained within the chip).

Erasable PROM (EPROM) program memory quickly eclipsed PROM-based memory because it was reprogrammable. The microcontrollers using this type of program memory became available in the late 1970s. EPROM uses ultraviolet light to erase its memory cells, which consist of a transistor that can be set to always on or off. Fig. 1.14 shows the side view of the EPROM transistor.

The EPROM transistor is a MOSFET-like transistor with a floating gate surrounded by silicon dioxide above the substrate of the device. To program the floating gate, the

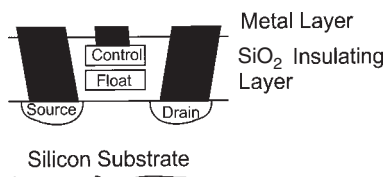


Figure 1.14 EPROM memory, which is programmed when the control gate forces a charge onto the floating gate.

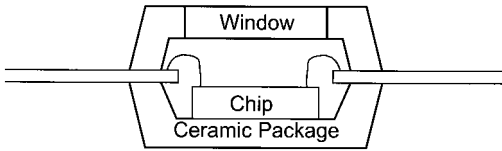


Figure 1.15 The quartz window on a ceramic EPROM chip package allows ultra-violet light through to erase the chip.

control gate above the floating gate is raised to a high voltage potential that causes the silicon dioxide surrounding it to break down and allow a charge to pass into the floating gate. With a charge in the floating gate, the transistor is turned on at all times. Before programming, all the floating gates of all the cells are uncharged. The act of programming the program memory will load a charge into some of the floating gates of these cells. By convention, the memory cell acts as a switch to a pulled-up bit. If an unprogrammed memory cell is read, a 1 will be returned because the switch is off. After the cell is programmed and pulls the line to ground, a 0 is returned.

To erase a programmed EPROM cell, ultraviolet (UV) light energizes the trapped electrons in the floating gate to an energy level where they can escape the silicon oxide barrier. In some manufacturer's devices, you find that some EPROM cells are protected from UV light by a metal layer over them. The purpose of this metal layer is to prevent the cell from being erased. This is often done in memory protection schemes in which critical bits, if erased, will allow reading out of the software in the device. By placing the metal shield over the bit, UV light targeted to just the code protection bit cannot reach the floating gate and the programmed cell cannot be erased.

This may seem like an unreliable method of storing data, but EPROM memories are normally rated as being able to keep their contents without any bits changing state for 30 years or more. This specification is based on the probability of the charge in one of the cells leaking away enough in 30 years to change the state of the transistor from on to off.

Microcontrollers with EPROM program memory can be placed in two types of packages. If you've worked with EPROM before, you probably have seen the ceramic packages with a small window built in for erasing the device (Fig. 1.15). EPROM microcontrollers are also available in black plastic packages that do not have a window, known as one-time programmable (OTP, see Fig. 1.16).

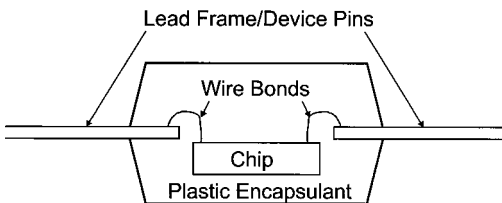


Figure 1.16 The plastic encapsulant of the OTP package does not allow ultraviolet light through to the EPROM chip inside.

The reason for producing OTP devices is probably not obvious when you consider that the advantage of the EPROM is its ability to be erased and reprogrammed using ultraviolet light, which cannot pass through opaque plastic. It seems to make more sense to go with a mask ROM or fusible link PROM device. OTP devices actually fill a large market niche, as windowed ceramic packages cost roughly ten times what a plastic package costs, the EPROM memory is more reliable than PROM, and in most microcontroller applications and products, the device will never be reprogrammed. So, by using OTP packaging, the part can be programmed at the product assembly site, will be electrically identical to the part used to develop the application, and is very cost effective for quantities less than the break-even point for mask ROM.

When you look at some manufacturers' and distributors' catalogs, you will discover that the term "OTP" is used in situations that don't match what is described here (that is, for chips that do not have EPROM memory). In the past few years, "OTP" has come to mean any programmable part that is not in a windowed package. Remember that if you are ever confused as to the appearance of a packaged part, you can look it up in the manufacturer's datasheets.

An improvement over UV erasable EPROM technology is electrically erasable PROM (EEPROM). This type of nonvolatile memory is built with the same technology as EPROM, but the floating gate's charge can be removed by circuits on the chip and no UV light is required. There are two types of EEPROM in use in microcontrollers. The first type is simply known as EEPROM and allows each bit (and sometimes byte) of the program memory array to be reprogrammed without affecting any other cells in the array. This type of memory first became available in the early 1980s and found its way into microcontrollers in the early 1990s. EEPROM has been very successful when implemented in small, easy-to-access packages.

In the late 1980s, Intel introduced a modification to EEPROM that was called Flash. The difference between Flash and EEPROM is Flash's use of a bussed circuit for erasing the cells' floating gates rather than making each cell independent. This reduced the cost of the EEPROM memory and speeded up the time required to program a device (rather than having to erase each cell in the EEPROM individually, the Flash erase cycle erases all the memory in the array, which takes as long as for 1 byte).

If you've spent some time programming PC applications, you've probably never worried about the space variables and data structures take up. Most modern PC languages will allow effectively unlimited direct storage. If you looked at a PIC microcontroller datasheet before reading this book, you would probably have been shocked to see tens to a few hundreds of bytes of variable memory in the file registers area of the chip, and you probably wondered how complex applications could be written for the device.

Creating complex applications with limited variable RAM in microcontrollers is not difficult, although large arrays cannot be implemented without external memory. Throughout this book, I will present some very substantial applications without requiring any external memory. These applications also include sophisticated text-based user interfaces that take advantage of the PIC microcontroller's ability to read program memory for text output data.

Variable storage in the microcontrollers is implemented as static random-access memory (SRAM), which will retain the current contents only as long as power is supplied

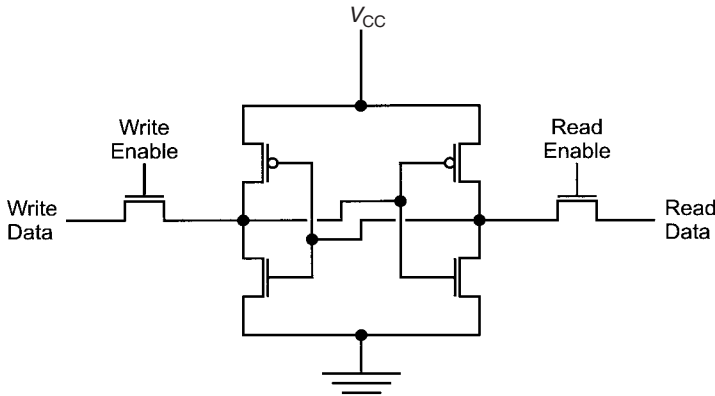


Figure 1.17 Six transistor CMOS SRAM memory cell circuitry.

to it and hence is referred to as volatile memory. Each bit in an SRAM memory array is made up of the six transistor memory cells shown in Fig. 1.17. This memory cell circuitry (probably known to you as a “flip-flop”) will stay in one state until the write enable transistor is enabled and the write data is used to set the state of the SRAM cell.

The P-channel/N-channel transistor pair on the write side of the flip-flop will hold this value as a voltage level because it will cause the P-channel/N-channel transistor pair on the read side to output the complemented value. This complemented value will then be fed back to the write side’s transistors, which complements the value again, resulting in the actual value that had been set in the flip-flop. This circuit is really a pair of inverters feeding back to each other, as I’ve shown in Fig. 1.18.

Reading data is accomplished by asserting the read enable line and inverting the value output (because the read side contains the inverted write side’s data). The driver to the SRAM cell must be able to overpower the output of the inverter in order for it to change the cell’s state. Once a value has been set in the inverters’ feedback loop, it will stay there until changed. This method of implementing a SRAM cell is well suited to a microcontroller

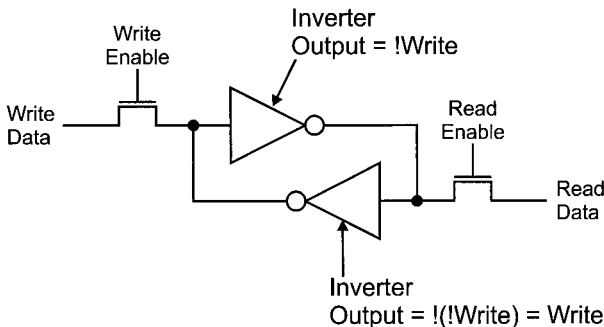


Figure 1.18 Conceptual diagram of a CMOS SRAM memory cell.

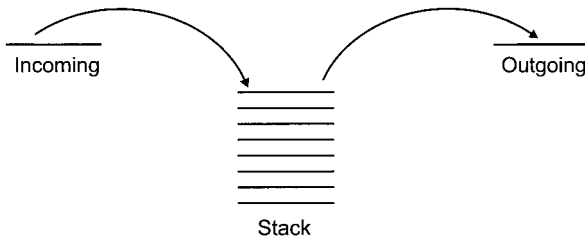


Figure 1.19 Stacks store data with the last item stored to be the first item retrieved.

because it uses very little power (current only flows when the state is changed) and is quite fast. It is not very efficient in terms of silicon space, as the six transistors required for each memory cell actually take up quite a bit of silicon surface area (or “real estate”).

Along with being able to access memory via absolute addresses, microcontrollers provide stack memory, which can be used for saving context information before a subroutine call or as part of the start of an interrupt handler. Instead of storing data at specific addresses, stacks (see Fig. 1.19) save data in a processor the same way you save papers on your desk, with the item on top of the pile being the first that you look at. A stack is known as a “last in/first out” (LIFO) memory. This should be pretty obvious—the first work item on the stack of paper will be last one you get to.

In a computer processor, a stack works in exactly the same manner as the stacked paper example. Data most recently put onto the stack (this is known as a push) is the first item pulled off the stack (this is a pop). The operation could be modeled with array variables using the pseudocode below:

```
push(data) {

    SP++;                      // Point to the Next Address
    in Memory
    Stack[SP] = data;          // Store the Data

} // End push

int pop() {

    int i;

    i = Stack[SP];              // Get Data Pointed to by
    the SP
    SP--;                       // Decrement the SP

    return i;

} // End pop
```

The PIC microcontrollers do not have data stacks available to you, but there are other ways of storing data that I will go through in more detail later in the book.

Microcontroller Communication

The capability of microcontrollers to communicate with other devices has become very important in the past few years. In the previous editions of this book, communication was discussed more as an afterthought. With the explosion of the Internet, the number of applications that require microcontrollers to be able to communicate with other devices have grown significantly. In this book, there will be more emphasis placed on enabling PIC microcontroller communication with other devices, both directly in point-to-point communication and in networked environments.

The term “point-to-point communication” describes connecting a microcontroller to devices that have known addresses. The term may seem confusing as it encompasses busses of multiple devices as well as communication paths that link two devices together. Several memory and peripheral chips connected to a microcontroller would be accessed using point-to-point communication techniques even if they could be removed during operation of the application (like a device on a network) because their addresses remain constant. In networked communications, the interface hardware and software allow for changing addresses, even if practically speaking, the same hardware (and the same addresses) are available throughout the life of the application. An important feature of networked devices is that they can generally operate even when the network is not available to the device. This differentiation between point-to-point communication and network communication may seem subtle, but as you will see in the following sections, there are significant differences in the way the communication schemes are implemented.

POINT-TO-POINT COMMUNICATION

Point-to-point communication between two devices in an application is typically implemented using serial connections to provide a basic data transfer capability. Many new developers will connect the devices in parallel (all bits transferred simultaneously on individual connections) with a full bus (consisting of address, data, and control lines) as this is something they are most comfortable with, having seen how chips are wired in microprocessor circuits. Embedded microcontrollers do not have the number of pins available to a typical microprocessor. They have just a few pins available, resulting in point-to-point communication being implemented using a serial data format. The serial protocols used are generally quite simple to implement although there can be some tricks to coding them. In this section, I will present the serial data transfer protocols used in point-to-point communication.

Applications that have multiple devices communicating with a microcontroller are not limited to just two chips; there are many applications with multiple chips connected to the central MCU. An obvious way of connecting these devices is to provide an individual connection from the microcontroller as shown in Fig. 1.20. This method obviously can only be used when there are sufficient I/O pins built into the microcontroller to allow connections between each device. If there are not enough pins, a bussed connection (Fig. 1.21) will have to be implemented using a point-to-point communication method that allows multiple devices to be connected together and not interfere with each other's operation. There

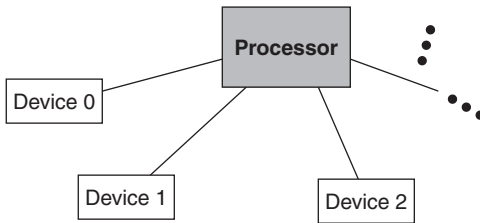


Figure 1.20 Multiple devices can be wired to a microcontroller using individual connections with a wiring scheme that looks like a star.

are a number of synchronous serial communication protocols (such as I²C and Microwire) that allow point-to-point communication devices to be wired in the common bus format.

Synchronous serial is the most basic method of transferring data serially and requires two lines, one for transferring data and another to clock the data to indicate when the transfer is taking place. A sample 8-bit data transfer is shown in Fig. 1.22 with the clock line pulsing when the data bits are valid. There is always a master device that initiates the data transfers and provides the clock for the data transfer (even if it is receiving data from another device). The slave device waits for the clock signal to either receive or send data. While the term “clock” is typically used to indicate a signal that occurs at a regular interval, the clock in synchronous serial communications is considerably more flexible and generally does not have to be timed precisely for the data transfer to take place.

When setting up synchronous serial communications between devices there are a number of things to be aware of. The first is which bit comes first in the serial data transfer—is it the least significant bit or the most significant bit? Next, you have to be aware of when the transfer takes place—is it during the rising or falling edge of the clock or some time when the clock is high or low? Along with the bit numbering and data valid for the clock, you should also understand whether the data bits are tristate or open collector to allow bidirectional data transfers or allow multiple devices on a common bus.

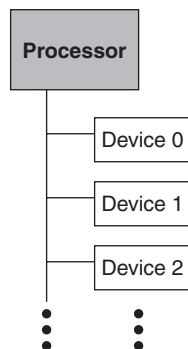


Figure 1.21 A single bus scheme can be used to connect multiple devices to a single microcontroller.

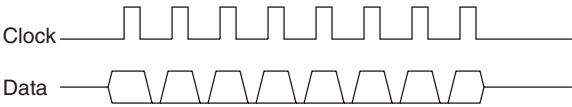


Figure 1.22 Synchronous serial data uses a clock to indicate when the data bits are valid.

While there are some interface standards for synchronous serial communications designed to allow multiple devices on a common synchronous serial bus, you will discover that there really isn't a standard way of implementing synchronous serial communications. Before starting your application development, make sure you have read the various device datasheets and understand how data transfers work.

Single data lines can be used for data communications and there are a number of protocols for sending data without a separate clock line. In all of these protocols, the timing of the data bits must be known and the receiver must be able to determine when a bit is coming and whether it is valid. To illustrate the operation of a single data line communications protocol, take a look at Fig. 1.23, which is a diagram of a 5-bit non-return to zero (NRZ) data packet. Each bit is a constant amount of time, with the start bit being used to indicate to the receiver that the data is coming with the data bits following (least significant bit first). After the data has been sent, there is a single parity bit, which is a simple error detection code, and a stop bit. The receiver continually polls (reads) the data line and when the signal goes low, it determines the middle of the bit and then waits a full bit period before reading the data. Hardware NRZ receivers built into microcontrollers are very common and provide the start bit detection and bit polling automatically, enabling the processor to simply read the data value from a register. The receiver must be provided with the period of each bit so it can successfully decode the incoming data packet.

Before moving on to some of the other single line point-to-point communication protocols, there are few things that you should be aware of about NRZ data packets. First, it is the data protocol used for RS-232 or asynchronous serial communications used in your PC; note that it is not the *electrical* protocol as this will not be what you expect and requires some specialized circuitry to implement. When used with the PC (and, indeed, most modern applications) the data packets are described as "8-N-1" which means 8 data bits, no parity, and 1 stop bit. The parity bit, as indicated above, is a simple error detection bit and when used will indicate whether the sum of the other bits is odd or even. It is rarely used because modern data transmission protocols provide more elaborate error detection and correction functions, which require less overhead than the

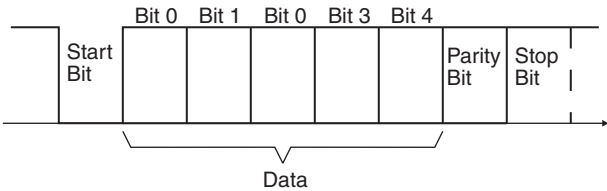


Figure 1.23 Data bits with a constant period can be sent together with a leading start bit and stop bit.

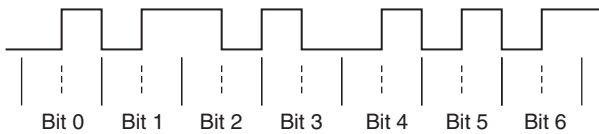


Figure 1.24 Transition in the middle of the bit period indicates its value.

simple parity check bit. The stop bit is some “dead air” in which the receiver can process the incoming byte and the transmitter can prepare the next one. Finally, the NRZ data packet can be used in common bus point-to-point communication when open collector drivers are used. These points may make NRZ sound like it is quite a complex communications methodology, but in reality it is simple to work with.

The Manchester data encoding scheme does not use a voltage level to indicate a bit value, but instead uses the direction of the transition of the incoming data line. Like NRZ, Manchester encoding has a constant bit period (Fig. 1.24), but in the middle of the bit (the dashed line) there is always a level change and, depending on the implementation, a low to high could mean a 0 and a high to low could mean a 1. The need to always have a transition can make a stream of Manchester data hard to interpret when you look at it—I find it to be nonintuitive. When moving from one bit value to another, there is no transition at the bit boundaries, and when two bits are the same value, there is a transition at the point between the two. Manchester encoding is often used in networking protocols where the receive synchs to the incoming signal using a phase locked loop and the level transition is used to toggle in a bit value. Though this sounds complex, the changing logic levels are quite easy to implement in hardware and do not require any timing resources on the part of the receiving processor.

The last method of providing point-to-point serial communication on a single line is the pulse-coded data format shown in Fig. 1.25, in which data is indicated by the length of time a signal is active. Whereas NRZ bit values are determined as logic levels at a specific time and Manchester bit values are logic changes at a specific time, the length of time a pulse-coded signal varies along with the entire data packet the bit is in. This change in packet timing makes pulse-coded data difficult to design traditional logic circuitry that reads the incoming data, and generally code must be written to read the incoming data. These two attributes make pulse coding of data to be inefficient in transmission

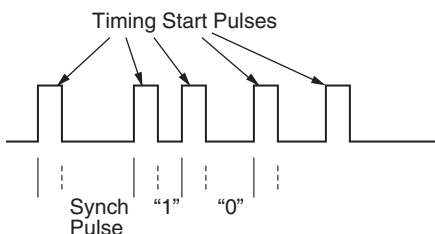


Figure 1.25 The length of time the signal is active (low) indicates the value of the bit.

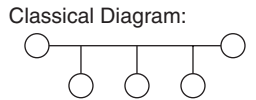
and in the amount of resources required to read the data. For these reasons, pulse-coded data is only used for small amounts of information that is occasionally transmitted. A popular application for pulse-coded data is TV remote controls, which are commonly used for controlling robots and other microcontroller applications.

NETWORK COMMUNICATION

It has only been in the past few years that microcontrollers have been considered legitimate network devices. The drop in cost in powerful microcontrollers and network interface chips for both wired and wireless applications has had a lot to do with the boom in MCU-based network applications. When wireless networks are noted, the Bluetooth and ZigBee protocols should also be considered along with WiFi (802.11) as potential network mediums. Similarly, home Ethernet networks allow for the addition of networked sensors and control devices throughout the house; the old joke of the Internet-enabled toaster has never been closer to being a reality. When the second edition of this book was written, there were just a few tentative steps toward creating microcontroller-based network devices, but in this edition more space will be devoted to showing you how to create networked applications using the PIC MCU.

Understanding how a network is laid out and wired will tell you a lot about the various applications that run on it and their characteristics. Rather than use the term “layout,” computer scientists use the term “topology” to describe how a network is organized and how different devices (usually referred to as “nodes”) are wired to each other. As a simple rule, the more connections nodes have with other nodes within the network, the higher performance (and higher reliability) the network will be. As will be shown, multiple networks of different types can be connected together using nodes known as bridges, which have network interfaces for the different network types. In fact, the Internet is really just a collection of networks that have been networked together at different points. The network topologies shown in this section are really for your edification and to familiarize you with some of the terms that will be presented later in the book when networking is discussed. Fully understanding the characteristics of network topologies is really in the realm of computer scientists who are designing networks for specific applications.

Fig. 1.26 shows the prototypical network, the bus or single media network in which a single connection is used to link all the nodes in the network. While “bus” is the more commonly accepted term, I prefer “single media” because it’s more descriptive and avoids confusion with point-to-point communication using a bus. I also prefer to visualize it using the diagram at the bottom of Fig. 1.26, which looks like a blob (the communications medium) with nodes sprinkled within it. If you visualize the blob of the single media network as air and the connection between the nodes as radio waves, you can see that there can only be one node transmitting at any given time. If you have a WiFi network at home, you should appreciate this model because only one computer can transmit at any time or the messages collide and become garbled. Part of the single media network hardware must be a receiver that monitors the outgoing messages to ensure that messages do not collide. If they do, the transmitter will wait a random amount of time before retrying to send the message. When the nodes are not transmitting, the network



Single Media Perspective:

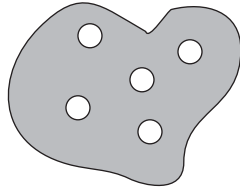


Figure 1.26 A bus or single media network can be drawn out as either a simple bus or a number of devices within a single common communications medium.

receivers must continually monitor the messages going on around it and record anything that is addressed for the node. This method of networking probably seems unnecessarily complex, but as noted, it was the first type of computer network developed and its deficiencies were targeted when other types of networks were developed.

The star network (Fig. 1.27) improves on the basic single-media network by linking all the nodes to a single switch (“Sw” in Fig. 1.27), which is responsible for directing messages between nodes (directly from the transmitter to the receiver). The addition of the switch allows multiple messages to be passed simultaneously between nodes. If you access a cabled Ethernet network, this is the topology that is used. Many networks use a router instead of a switch for passing messages back and forth, and the difference between the two is that a router will determine the best path for a message (including passing it along to other networks). If you have a cable modem at home, you probably have a router to pass data between computers in your home as well as out to the Internet if it is required. By providing a direction connection from each node to the switch, the star network allows much faster data transmission along with less unnecessary traffic to the various nodes compared to the single-media network.

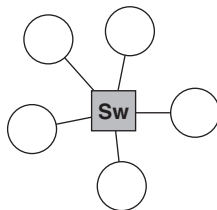


Figure 1.27 A switch linking each node directs messages directly from the transmitter to the appropriate receiver.

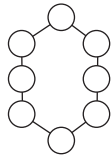


Figure 1.28 A ring of nodes passes a token that is used to control the movement of messages between computers.

The ring network (Fig. 1.28) endeavors to avoid the collision issue by passing a token, which can have a message attached to it. Computers only transmit onto the ring network if the token is empty when it comes to them. If there is a message attached to the token, the computer must strip it and transmit an empty token in its place. Message transmission is quite efficient although there is a major drawback to this network topology: if one node fails, the entire network is brought down. Despite this disadvantage, ring networks are used in some telecommunications applications.

Supercomputers made up of multiple processors and high performance server farms provide multiple network connections from each node to, ideally, every other node in the network, as shown in Fig. 1.29. This topology is known as a mesh network. It avoids the issues of possible message collisions by having each node specify where each message is going. The obvious disadvantages of this topology are the cost of having multiple network adapters on each node as well as the wiring complexity between each node. It is impossible to provide a connection between each node when there are more than four or five nodes in a mesh, so there has been a great deal of research done on designing the optimal mesh configurations (often consisting of meshes of meshes) for supercomputers or servers built from thousands of microprocessors. There really is no need for mesh networking when working with embedded microcontrollers.

Large networks, such as the Internet, are usually called hybrid networks because they are built from smaller networks of varying topologies as shown in Fig. 1.30. The interconnections between the smaller networks and the large hybrid are usually through bridges (connections between two dissimilar network connections) or routers (when the connections are the same).

Finally, I want to discuss universal serial bus (USB) adapters, as they are common applications for embedded microcontrollers like the PIC MCU. While you may be most

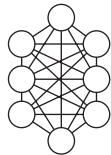


Figure 1.29 The mesh of connections provides a direct path to every computer in the network, allowing extremely fast data transfers.

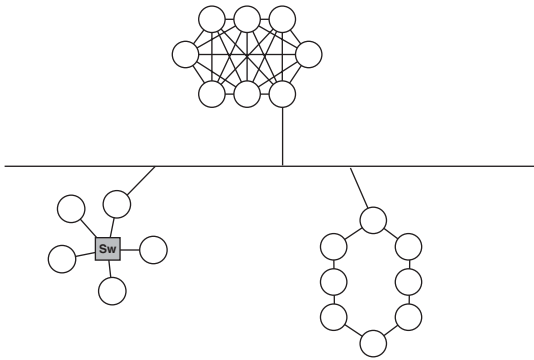


Figure 1.30 A large network built from a number of smaller networks is usually a hybrid network, consisting of several different topologies as shown here.

familiar with them being used as peripheral devices for PCs and as point-to-point communication devices, providing I/O functions for the computer, I think of them as network devices. When they are plugged into a PC, they go through an enumeration process similar to that of new nodes being connected to a network. They can be designed to operate independently of the PC, using the PC's USB port as a method of transferring data from the device to a PC application. This may be a rather unconventional way of looking at USB devices, but I find that developing applications for them is much easier if this perspective is taken.

This discussion on networks is quite simplistic and there are many topics I have not discussed in fear of getting bogged down in minutiae when the purpose of this chapter is to give you an introduction to the various concepts that are involved with creating embedded microcontroller applications. Later in the book, I will discuss the various issues required for designing network applications for the PIC microcontroller.

Device Packaging

There are quite a few packaging options you should be aware of that will affect the cost, size, robustness, and performance of the final application. When I use the term “device packaging,” I am describing the methodology used to protect a chip and the interconnect technology used to connect it electrically to the printed circuit board (PCB, which I call the raw card) and other circuitry used in the application. Knowing which options are available will allow you to come up with a final circuit that best meets the requirements of the application.

The term “encapsulation” is used to describe the method of protecting the chip from the environment. There are two primary types used: plastic and ceramic. Plastic encapsulants are the most prevalent and use an epoxy potting compound to provide effective

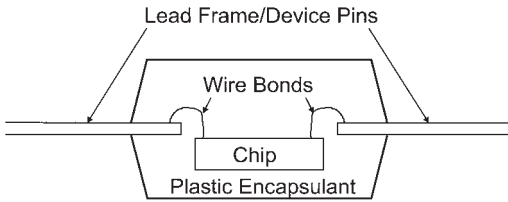


Figure 1.31 OTP packaging completely surrounds a chip with an impervious epoxy encapsulant.

and inexpensive protection. Ceramic encapsulants are considerably more expensive (driving chip costs up as much as ten times over that of plastic encapsulated chips) and should only be considered when thermal or other considerations make its use mandatory. In the second edition of this book, the current technology required that PIC microcontroller ceramic chip packages be used for various applications. With the advent of Flash program memory, every application presented in this edition can be built using chips in plastic packages.

Before the chip is encapsulated or packaged, it must be connected to a lead frame. The lead frame is a copper form that becomes the pins that come out of the package and connects the chip to the other circuitry in the application. The copper lead frame is wired to the chip via very thin aluminum wires ultrasonically bonded to both the chip and the lead frame. During encapsulation, the lead frame is used to handle the chip and the encapsulant while it is hardening.

Plastic packaging is often referred to as one-time programmable (OTP, see Fig. 1.31) packaging due to its historic relationship with EPROM program memory. Earlier in the chapter, I described EPROM program memory as being erasable when exposed to ultraviolet light. To allow a chip to be erased, a package with a window to allow the light to pass through has to be used, but there are applications where EPROM is the best choice and there is no need to reprogram it. The chip was therefore put into a plastic encapsulation and labeled “one-time programmable” because once the EPROM was programmed, it could never be changed. This term has continued to be used with all PIC microcontrollers built with plastic packaging (and many other chips besides).

The primary purpose of putting an embedded microcontroller into a ceramic package is to allow a quartz window to be made available for the purpose of erasing the EPROM program memory (Fig. 1.32). When a ceramic package is used, the chip is glued

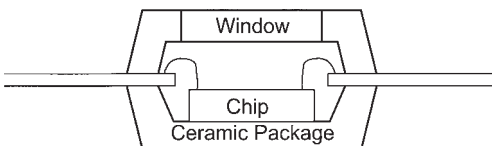


Figure 1.32 Windowed ceramic packaging allows a chip to be exposed to ultraviolet light to erase its EPROM program memory.

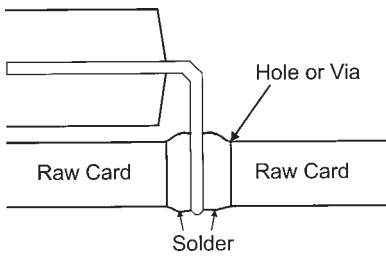


Figure 1.33 Pin-through-hole connection from a chip to a printed circuit board.

to the bottom half and wired to the lead frame. Ceramic packaging is normally only available as a pin-through-hole (PTH) device, whereas plastic packages have a wide range of card attachment technologies. The added cost of the ceramic packaging makes it typically only appropriate for prototyping circuits where the embedded microcontroller will be erased and reprogrammed.

The technology used to attach chips to PCBs has changed dramatically over the past 20 years. In the early 1980s, most devices were only available in PTH technology packages (shown in Fig. 1.33) in which the lead frame pins are soldered into holes in the raw card. This type of attach technology has the advantage that it is very easy to work with—little specialized knowledge or equipment is required to manufacture or rework boards built with PTH chips. The primary disadvantage of PTH is the amount of space required to put the hole in the PCB as well as the size of the chip's pins. Surface-mount technology (SMT) eliminates this disadvantage by soldering the chip's pins to the surface of the PCB using one of the two pin types shown in Fig. 1.34. The two SMT pin types offer advantages in certain situations. The gull wing pin allows for hand assembly of parts and easier inspection of the solder joints. The J lead reduces the size of the part's overall footprint.

The shift from PTH to SMT is due to the reduction in PCB space required by SMT for a given number of I/O pins. The size difference of an SMT versus PTH component is shown in Fig. 1.35. Pin-through-hole components are normally built with pins 0.100in (100 thousandths of an inch) between pin centers. The measurement between lead centers is a critical one for electronics because it is directly related to how

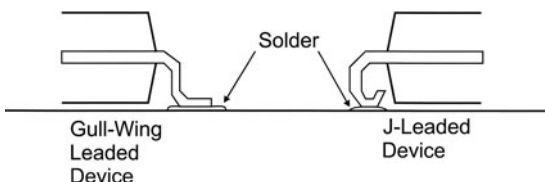


Figure 1.34 The different types of surface-mount technology pins used to solder a chip to a PCB.

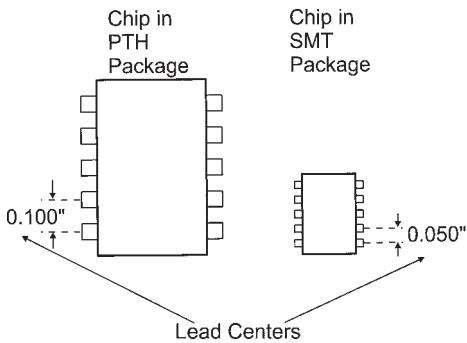


Figure 1.35 Comparison of PTH and SMT components sizes for the same number of pins.

densely a board can be populated with electronic components. The SMT component shown in Fig. 1.35 has lead centers of 0.050in and the amount of board space is about 25 percent of the PTH component. This may seem like a considerable improvement, but consider that SMT components can be placed on *both* sides of a printed circuit board—which means that eight SMT chips use up the same amount of PCB real estate as one PTH chip!

The 0.050in lead centers quoted above are actually very large for SMT components. Modern SMT components can have lead centers as small as 0.16in (0.4mm). SMT parts with small lead centers are known as fine pitch parts. Further reducing board sizes for a given set of pins is the SMT technology known as ball grid array, in which the SMT pins are built from a two-dimensional array of small solder balls. Amazing as it sounds, there are chip packages with over 2,000 leads that take up no more than a square 1.25in (30mm) on a side.

Assembly and rework of SMT parts are actually easier in a manufacturing setting than PTH. When the components are first assembled onto PCBs, a solder/flux mixture called solder paste is printed onto the SMT pads of the boards using a metal stencil with holes cut in the locations where the solder paste is to be put. A squeegee-like device spreads the paste over the stencil and deposits it on the card where there are holes. Once the paste has been deposited, the parts are placed onto the paste and then run through an oven to melt the solder paste, soldering the parts to the board. To rework a component, hot air (or nitrogen gas) is flowed over the solder joints to melt the solder, allowing the part to be pulled off. SMT is easier to work with in a manufacturing setting; it is a lot more difficult for the hobbyist or developer.

Chip-on-board (COB) or glob top packaging is just what the name describes—a chip is literally bonded to a printed circuit board with just a drop of encapsulant over it. COB parts are useful in applications that require a very small form factor for the final product; because the chip is used directly, there is no overhead of a package for the application. Typical applications for COB include telephone smart cards and satellite or cable TV descramblers. There are two methods of COB attachment currently in use. The

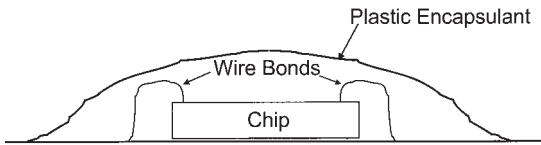


Figure 1.36 Chip-on-board packaging consisting of a chip wire bonded directly to a PCB with a glob top of encapsulant placed over it for protection.

first method is to place the chip on the card and wire the pads of the chip to the pads on the card using the same technology as wiring a chip to its lead frame inside a package. This is done using small aluminum wires ultrasonically welded to the chip and raw card. Fig. 1.36 gives you an idea of what this looks like.

The other method of COB is known as C4 and is actually very similar to the SMT process described above but with the chip being soldered directly to the PCB rather than a plastic package. Pads on the chips have small solder balls (called bumps) attached to them that are soldered to matching pads on the PCB. This technology was originally developed by IBM for attaching chips to ceramic substrates or backplanes without having to go through a wire-bonding step. It requires a significant investment in tools for placement as it is a specialized process used in applications where PCB real estate is at a premium and the application is not expected to experience significant thermal extremes.

Application Development Tools

When the first computers were created over 60 years ago, machine instruction programs were written by hand, converted to instruction bits, and manually loaded into the computer ROM. ROM in the first computers consisted of large banks of switches, and setting them correctly for a specific program was a pretty onerous task. The application development process was tedious and had to be done very carefully to prevent an error creeping that would take hours to recognize, find, and fix. Today, there is a certain irony that computers have taken over many of these tasks and in doing so have eliminated many problems, making the application development process much faster and easier. The applications that provide these functions were originally designed as stand-alone programs but have recently been packaged together into a single program known as an integrated development environment (IDE), which eliminates almost all of the manual effort needed to develop a computer application.

The first tool you will have to be familiar with is an editor, which will allow you to create the program source code. The source code is the human-readable file that contains the instructions (either as assembly or high level language) that are to be processed into the bits executed by the processor. There is a plethora of different editors available, ranging from simple line editors (such as the MS-DOS Edlin program) to very sophisticated programmer's editors that automatically insert programming text in specific

formats for you and will check the code over for errors. Personally, I believe that the choice of editor isn't that important, although I would suggest that you use one with the same "Ctrl" codes as standard Windows editors to minimize the possible confusion between cut-copy-paste-undo in different programs.

With the program stored in a text file from the editor, it must be converted or built into an object file that can be loaded into the processor. The most basic conversion program is the assembler, which converts processor instructions into the bits required to execute by the processor. Even though I used the term "most basic" to describe the assembler, the program has more sophisticated features and capabilities than you would expect from something that works with the basic machine instructions. The input to the assembler is a text file and it produces a number of files, including a listing file (containing a printout of the program and system generated comments and the instruction codes) and an object code file (which contains the straight program bit information for loading into the processor). There are also files to help with program debugging or loading into the processor. Most assemblers are described as being "two pass," which means they read through the program once to check the instructions and identify labels with addresses and again to associate the addresses with the different instructions.

Compilers convert high level languages into object code files in a similar manner to the assembler. Like the assembler, the compiler also outputs object, listing, debug, and loading files. Compilers may also output an assembler file, which will allow you to look at the instructions that are needed to implement a given high level statement—an excellent way to see how assembly language programming is done to accomplish specific tasks.

There are two types of object code files produced by compilers and assemblers for use in embedded microcontrollers. The .hex file is an application file that has the complete data to be loaded into the microcontroller. The .obj file will have to be linked to other files or libraries (collections of standard functions) to create the .hex application file. The .hex file is the final form of the object code and is used by the programmer that loads the microcontroller's program memory.

There are three ways to test the operation of the application code in an embedded microcontroller: a simulator, a debugger, or an emulator. The simulator is a program that simulates the operation of the microcontroller and its I/O on a PC so you can test your application before loading it into a chip. A debugger consists of devoted I/O pins on an embedded microcontroller chip that interfaces to PC-based software that allows execution monitoring and control. The debugger differs from an emulator, which makes the entire microcontroller accessible for monitoring program execution without devoting any I/O pins to interface with the controlling PC program. Each of these application test tools requires additional files from the assembler and compiler to provide a source code view of the program's execution to allow you to understand exactly where the program is executing and what it is doing.

To summarize, the full collection of tools that are used to create a microcontroller application are:

- Editor
- Assembler

- Compiler
- Linker
- Simulator
- Debugger
- Emulator
- Programmer

In the not so distant past, all these tools were separate, and part of the effort of creating a lab for developing embedded microcontroller applications was to select, install, and learn each of them. This was quite a bit of work, both to select the tools and to make sure that they were compatible. When the first edition of this book was written, I suggested that the reader use the Windows WordPad editor, a command line (seriously!) assembler, a separate command line simulator from Microchip, and a Windows-based programmer. At the time, there was quite a bit of frustration in finding an editor that was compatible with the assembler and making sure the assembler output was compatible with the chosen programmer.

This situation was not unique to the embedded microcontroller; selecting, installing, and integrating software for many computer platforms was a difficult task. To ease this burden, tool vendors produced integrated development environments (IDEs), which brought together all the tools necessary to write, build, and test software in one package. For the PC, Microsoft's Visual Studio has set the standard for complete and easy to use tools for application software development.

In the late 1990s, Microchip introduced their MPLAB IDE, which integrates all these functions into one easy to use desktop tool. Now, to create a software development lab for PIC microcontrollers, the only time and effort required is to download and install the software. Once installed, you have a software development lab that is complete on its own or can be upgraded with additional compilers and debuggers, emulators and programmers. Over the years, MPLAB IDE has been continually refined to make it the standard for embedded microcontroller development tools. It allows you to efficiently create, build, and test the highest quality code, free of the translation errors that were rife in the early computer applications, where programs were assembled by hand and programmed manually into switch-based memory.

This page intentionally left blank

THE MICROCHIP PIC MICROCONTROLLER

If you were to look at a complete list of the different types (or part numbers) of PIC[®] microcontrollers, you would probably be overwhelmed by the number of choices, each with unique features and configurations. As I write this, there are more than 275 PIC MCU part numbers available—a result of Microchip designing and releasing thirty or more part numbers each year for the past six or seven years. Despite the best efforts of Microchip, you cannot divine the features of a PIC microcontroller by its part number. The most efficient way to find a PIC MCU with specific features or to understand what features a part number has is to use the search functions built into Microchip's Web page.

The purpose of this chapter is to introduce you to the high-level features of the plethora of PIC microcontrollers that you have to choose from. It will also help you start thinking of the complete catalog of parts, rather than just one or two specific chips that can be used in a variety of applications, although not truly optimized for any of them. We'll begin with Microchip's Web page, an invaluable resource with many megabytes of information that will make your device selection, circuit design, and application coding much easier. Next, we'll take a look at the characteristics and features of the PIC microcontroller part numbers and the electrical and mechanical features of the chips. Next, I will explain some of the unique features of the PIC MCUs that provide them with powerful capabilities not available on other microcontrollers. Finally, I will discuss the PIC microcontroller processor architectures that allow you to tailor the “number crunching” capabilities to the application. After reading through this chapter, you will have a good idea of how to take best advantage of the many PIC microcontroller part numbers at your disposal.

Accessing the Microchip Web Site

The most important resource at your disposal is Microchip's Web site at www.microchip.com.

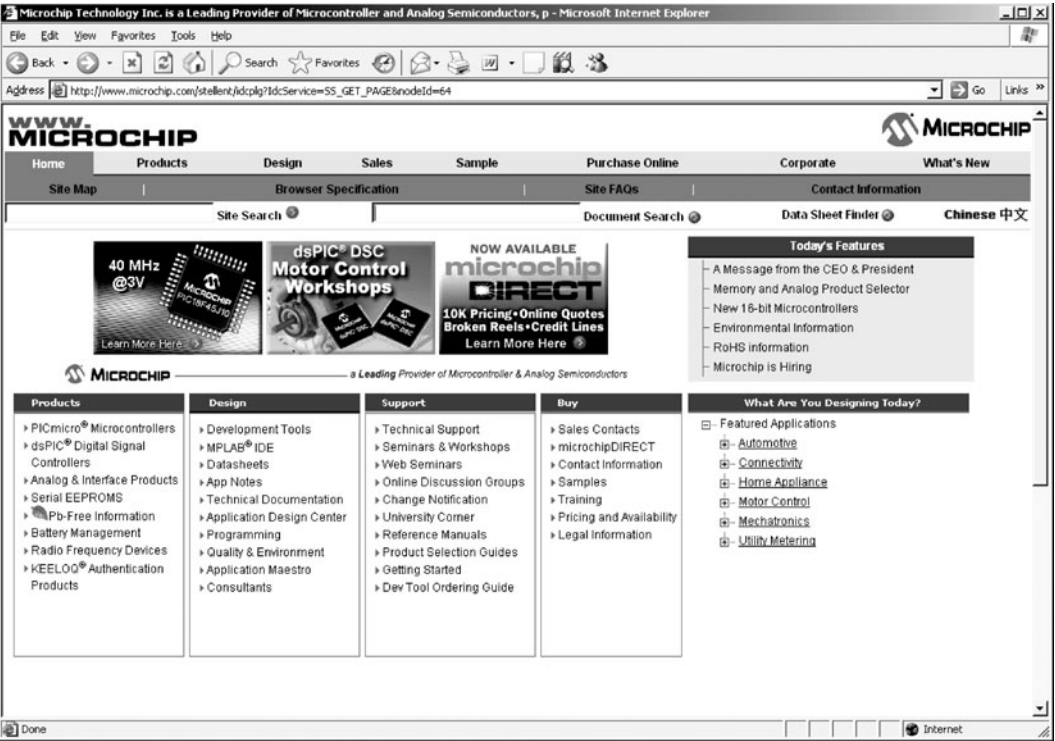


Figure 2.1 Microchip’s home page (<http://www.microchip.com>).

The home page (Fig. 2.1) provides you with links to information on each of the PIC microcontroller part numbers, sample applications (known as application notes or app notes), downloadable tools, and other resources you can take advantage of. There is a great deal of information available at the Microchip Web site, and you should take some time looking through it and familiarizing yourself with the resources, as it can be overwhelming to find specific information quickly when you first start to work with the PIC microcontroller. Please note that microchip homepage is continuously changing.

The site is broken up into a number of high level headers, with PICmicro Microcontrollers being the topic that you will probably spend the most time with initially. By clicking on this link, you will be shown groups of devices to choose from listed by number of pins, program memory, processor architecture, special I/O features, and more. As you become more knowledgeable about the PIC microcontrollers and understand the development process better, you will be able to find the category (say, 14-pin chips) and select the PIC MCU that is best for your application based on the list of parts you are given.

There are a number of other high level headers, such as Development Tools, MPLAB® IDE, Technical Support, and so on, that you can click to find more information or to have specific questions answered. The pages that are linked to these headers may seem to be

focusing on devices other than the ones you are interested in or may have a bewildering series of options to choose from. Initially, I would recommend that you use the Site Search feature to find more information on a specific PIC microcontroller part number and only click on the headers that are specific to the development tools you are using. Choose the MPLAB® IDE link for application development questions and Technical Support for technical questions. This will help you stay focused in your search for information and won't give you too many choices to search through.

I find the most efficient way of finding information for a specific PIC microcontroller is to put the cursor at Site Search (which is a Google-powered search engine) and enter the information I am looking for. There is an important difference in how I use this search engine as opposed to the generic Google search page, and that is I tend to keep my search strings centered around just the part number I am working with or want to get more information for. In Fig. 2.2, I have done a screen capture of the search results for "PIC18F2550," showing you the resources available for this part, including the summary information about it, data sheet, application notes, programming information, development tools, errata, and so on. These pages have all the essential information you will require to develop an application using a specific part number.

REFERENCING THE DATASHEETS

As you work through this book, to fully understand how the examples and applications work, you will probably want to download the datasheet of the part being referenced. Microchip has done a wonderful job of creating accurate and detailed datasheets for the complete set of PIC microcontroller part numbers (as well as development tools and systems) that you will need to develop your own applications. These datasheets are very large and can be overwhelming to new developers. I have a few suggestions for finding and referencing the information quickly. Each datasheet is arranged with the following information:

- Device summary pages, showing part pin out and listing important features of the device (including processor architecture and I/O features). This summary will help you decide which part number is best suited for your application.
- Family part number differences (primarily in program and data memory size and pin outs for packaging options)
- Table of contents
- Basic operating characteristics
- Processor architecture overview
- Data and special function register (SFR) organization and access
- I/O, peripheral and internal feature operation, and register descriptions
- Instruction set summary
- Configuration and operating mode description
- Operating electrical characteristics

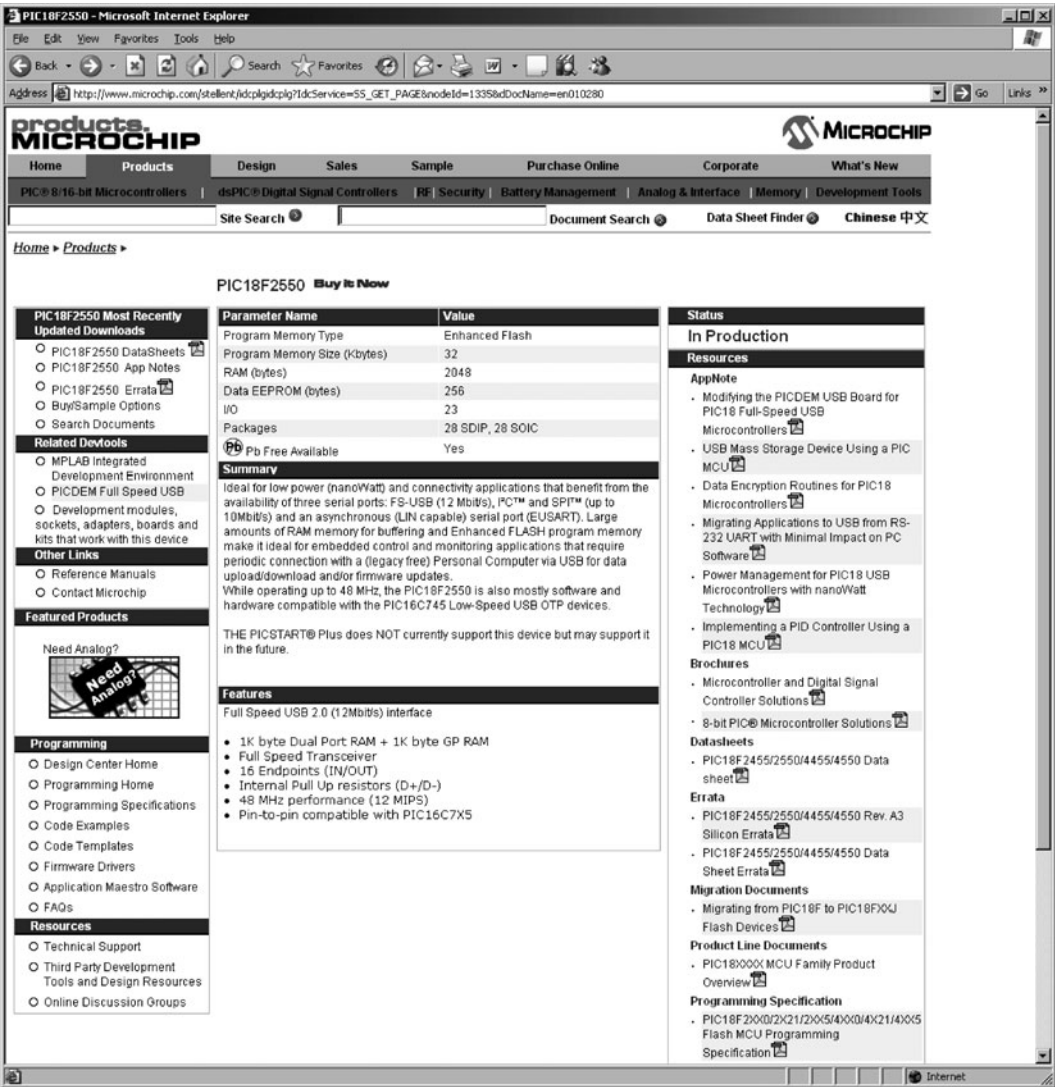


Figure 2.2 Microchip’s information page for the PIC18F2550, listing various information resources available for the device.

- Packaging information
- Miscellaneous information
- Microchip sales and support contact information

This list of information is consistent for all PIC microcontroller datasheets, but you will find some variances for different part numbers. These variances could be a result

of the datasheets being older (and being in a slightly different format) or because the chip has special features, which require the addition of extra sections.

The most important page when figuring out which PIC MCU to select is the part summary at the start of the datasheet. This page should have enough information for you to decide whether the specific part number will work in your planned application. Much of this data is available at Microchip's Web site, both in a page view (like in the datasheet) for you to review the part number's features as well as in selection tables to allow you to compare part numbers to identify which parts best meet your requirements. Throughout this book, I will reference the information in the other sections of the datasheet and point out what you should be looking for in the datasheets, but to begin you should just be comfortable with the summary page.

When you start working with PIC microcontrollers, I recommend that you have a complete datasheet available for your use. If you have a Microchip representative you can call upon, you can probably get printed and bound copies of the datasheets but most likely you will have to print your own. When I print out datasheets, I normally print them out on two sides of a piece of paper using the technique shown in Fig. 2.3. First, the even pages are printed in reverse order (this can be selected in your printing option windows) followed by putting the pages back into the printer with the blank side ready for printing (and orientated so the tops of the two sides of the page are the same). Once this is done, you can put the pages into a binder or use some other method of keeping them together. I find it takes between 15 to 30 minutes to print out a 150-page datasheet on both sides of the paper.

Microchip PIC microcontroller datasheets are stored in Adobe Acrobat (pdf) format files and while you can view them online, I recommend that you save them on your PC's hard disk before attempting to print them out. Chances are you will want to go back to them at some point in the future and having them on the hard disk will let you retrieve

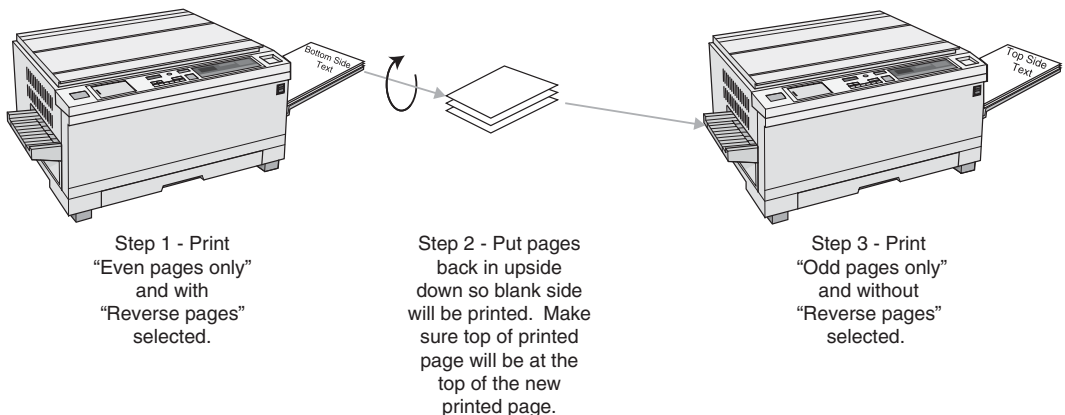


Figure 2.3 To create two-sided documents, first print the back (even page number) side in reverse order, put the pages back into the printer, and print the front (odd page number) side in ascending order.

them much more quickly than looking them up on Microchip's Web site. It will also minimize your frustration when printing out the datasheet if some kind of problem occurs that requires you to reboot your PC.

I have a fairly cheap (and old) laser printer and it isn't unusual for pages to get stuck together or jammed in the printer's mechanism. To minimize the problems, I usually keep the size of the printing to 20 to 40 double-sided pages (10 to 20 pieces of paper) at a time. I keep the page counts small because I find that the printer does not cancel jobs effectively due to the page buffer in the printer, and if a hard stop (i.e., pulling the printer's plug) is made, the whole system may have to be rebooted. Keeping page count low minimizes the number of ruined pages if there is a problem—if printing fouls up while printing a large datasheet (some can be more than 300 pages), I will have wasted most of a package of paper. With a bit of experimentation, you will learn how to best print out, bind, and use the PIC microcontroller (and other chip) datasheets most effectively.

PIC Microcontroller Feature Summary

Choosing one 8-bit PIC microcontroller part number over another for a specific application is not as difficult as you may think. The chips are based on four processor architectures which provide different capabilities while still requiring many of the same thinking skills and tricks required to create an application. Code development is greatly simplified by the use of Microchip's MPLAB IDE (integrated development environment), which can be used for all the PIC microcontroller part numbers. The input/output (I/O) pins and peripheral features are designed to easily interface with devices wired to the PIC MCU using standard interfaces or bit-banging general purpose pins. The real secret to designing PIC microcontroller applications is not to learn one part inside and out before looking at other devices, but to understand the fundamentals of programming the devices, the interfacing options available to you, and working with MPLAB IDE. By taking this approach, you will be able to work with a great many PIC MCUs with very little learning specific to individual part numbers.

PIC MICROCONTROLLER FEATURES AND PERIPHERALS

In Table 2.1, I have summarized the electrical features that make up different PIC microcontrollers and can be searched on to find the part number that best meets your application's requirements. Table 2.1 does not summarize the packaging options that are available for the PIC MCU part numbers—all part numbers have multiple packages (the plastic or ceramic case that protects the chip from the elements and allows it to be attached to a printed circuit board)—but this information is presented in detail later in this chapter.

One parameter not included in Table 2.1 is the specification between commercial (standard temperature range), industrial (extended temperature range), or automotive (extreme temperature range) parts. Parts specified to work in one of the extended

TABLE 2.1 MICROCHIP PIC MICROCONTROLLER DISTINGUISHING CHARACTERISTIC AND FEATURE LIST

FEATURE	OPTIONS	COMMENTS
Processor architecture	Low-end, mid-range, PIC17, and PIC18	The computer portion of the microcontroller that executes programs. The processor architecture is often referred to as the "family" the PIC microcontroller belongs in. Interrupt capability is a function of the processor architecture.
Operating voltage	Standard parts: 4.5 to 5.5 volts; L or low voltage parts run from 2.0 to 6.0 volts	Most new parts run at what was once considered the low voltage range.
Debugging options	MPLAB ICD built-in debugger hardware	While there is emulator hardware available from Microchip for all PIC microcontroller part numbers, some have built-in debugger capabilities, which allows for low cost in circuit application debugging.
Program memory type	Mask ROM, EPROM, Flash	The program memory contains the application software used by the PIC microcontroller and is known as nonvolatile because it is not lost when power is taken away from the PIC MCU.
Program memory size	Ranges from hundreds of instructions to millions	There are many similar PIC microcontroller part numbers in which the only distinguishing characteristic is the amount of program memory.
Programming method	Low-end parallel, in-circuit serial programming (ICSP), PIC17 parallel	The most popular method of programming is ICSP, which uses the same I/O pins as the MPLAB ICD debugger.
Oscillator type	Watch (32.768 kHz) crystal, high speed crystal (1 MHz and above), RC oscillators, external resistor oscillators, internal oscillators, clock multiplier PLL, dual operation	Many clocking options are available for PIC microcontrollers. Selection can affect application performance, accuracy, and cost.
Reset operation	External, optional internal, brownout reset, watchdog timer, power up delay timer	Options available to help control the reliable operation of the application.

(Continued)

TABLE 2.1 MICROCHIP PIC MICROCONTROLLER DISTINGUISHING CHARACTERISTIC AND FEATURE LIST (CONTINUED)

FEATURE	OPTIONS	COMMENTS
File register size	From tens of bytes to thousands	File registers are the PIC microcontroller's variable memory. Register organization is specific to processor architecture.
Data EEPROM size	Optional in some devices from tens of bytes to hundreds	Long term or nonvolatile variable storage.
Timers	One to three timers available; timers are 8 and 16 bits in size	Timers can be used for software timers as well as counting/timing external events.
I/O pins	Number, type (totem pole, open collector, analog I/O), internal pull-ups	I/O pins are available in quantities of fewer than 10 to more than 60. Analog inputs are either summing ADC or comparators.
Interfaces	USART, SPI, I2C, USB, CAN, parallel slave port, LCD	I/O pins can either be used as general purpose interfaces or devoted to specific interface functions.
Event sensors	Compare/capture/PWM module, pin state change interrupts	Monitor and flag processors of external events.
Operating capabilities	Bootloader, real time operating system	Features to allow advanced software operations.

temperature ranges are usually the standard temperature range parts, which have been tested for some period of time at the required temperatures to ensure they will work reliably.

PIC MICROCONTROLLER PACKAGING OPTIONS

PIC MCUs are available in a variety of packaging, as can be seen in the following figures, which illustrate the packages as well as the letter codes that specify them. Fig. 2.4 shows the one-time programmable (OTP) packages that you will be using when you are first learning about the PIC microcontroller. The term “OTP” became popular when programmable parts (which are normally in windowed ceramic packages, like the ones shown in Fig. 2.5) were built in solid plastic packages. The reason for doing this was the much lower cost of the plastic packaging and the expectation that the part would never be reprogrammed (which requires the availability of a window for ultraviolet light to erase the EPROM memory on the chip). PIC microcontrollers with EEPROM and Flash program memory that are put into solid plastic packages,

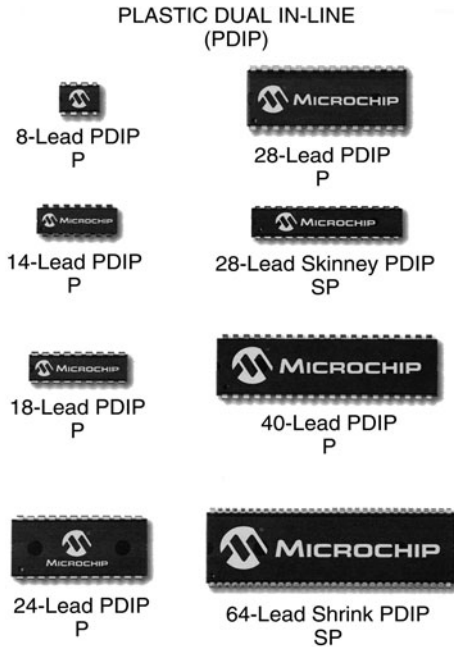


Figure 2.4 Plastic dual in-line (DIP) OTP packages.

like the ones shown in Fig. 2.4, do not require ultraviolet light to be erased and are still identified as OTP.

While you will learn how to wire into an application circuit using pin-through-hole (PTH) parts like the ones shown in Figures 2.4 and 2.5, you will probably never use these

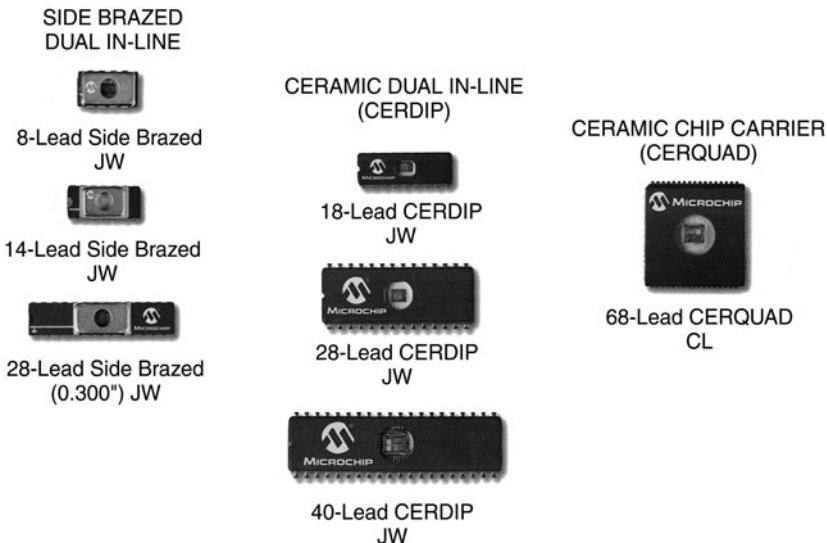


Figure 2.5 Windowed ceramic PIC microcontroller packages.

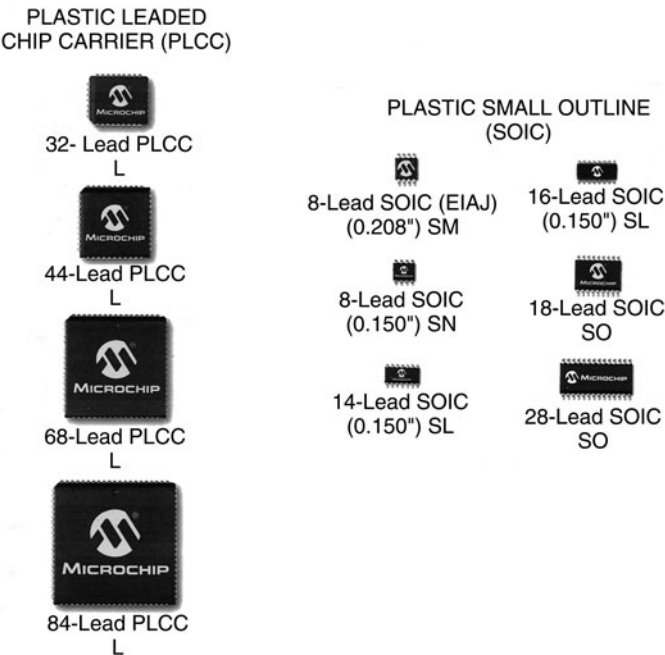


Figure 2.6 Standard plastic surface mount packages.

parts in products because of their size compared to surface mount technology (SMT) parts and the requirement for a solder wave to attach them to a PCB. Standard SMT packages are shown in Fig. 2.6 and the fine pitch lead devices are shown in Fig. 2.7.

Unpackaged chips are available for some parts from Microchip for use in chip-on-board (COB) applications. In these cases, the chips are said to be available in “waffle pack” for automated chip pickup and placement.

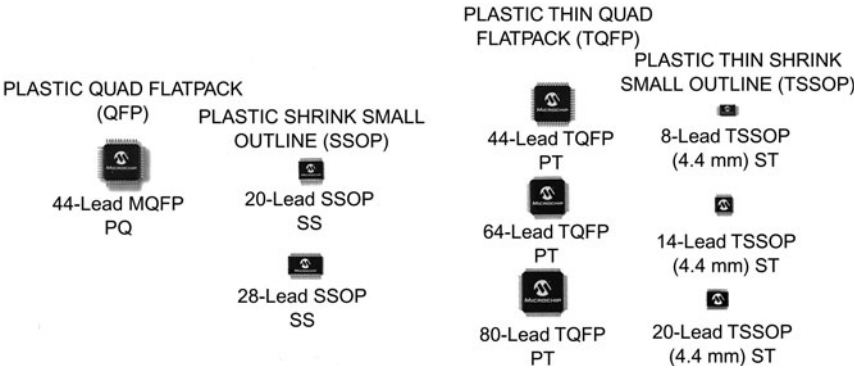


Figure 2.7 Fine pitch plastic surface mount packages.

Each package has a one- or two-letter code that is put at the end of the part number to describe which package the chip is in. These codes are listed in Figures 2.4 to 2.7, along with the package description. For example, the 16F84 is normally sold as a PIC16F84P, which indicates the 16F84 has a plastic, dual in-line package (DIP).

PART NUMBER CONVENTIONS AND ORDERING

Determining which part number to specify when ordering a PICmicro for an application or product is very consistent across the entire line. Fig. 2.8 shows the conventions for how the part numbers are specified by Microchip.

The PIC microcontroller chips described in this book are all 8-bit parts and will start with the five-letter groups PIC10, PIC12, PIC14, PIC16, PIC17, and PIC18. When you look at Microchip catalogs, you will also see dsPIC and PIC30 prefixes for microcontrollers. These parts are digital signal processors and 16-bit MCUs and will not be described in this book.

Note that Fig. 2.8 probably does not have all the possible part number options for parts released after publication of this book. The datasheet for the part number will have specific information about the packages the chip is available in, including dimensional information and PCB footprint information for some parts.

LETTER SUFFIXES

When you look for a specific PICmicro part number, you may discover that you have more than one part to choose from. For example, there are three versions of the 16C73 available: the PIC16C73, the PIC16C73A, and the PIC16C73B. The letter suffixes indicate

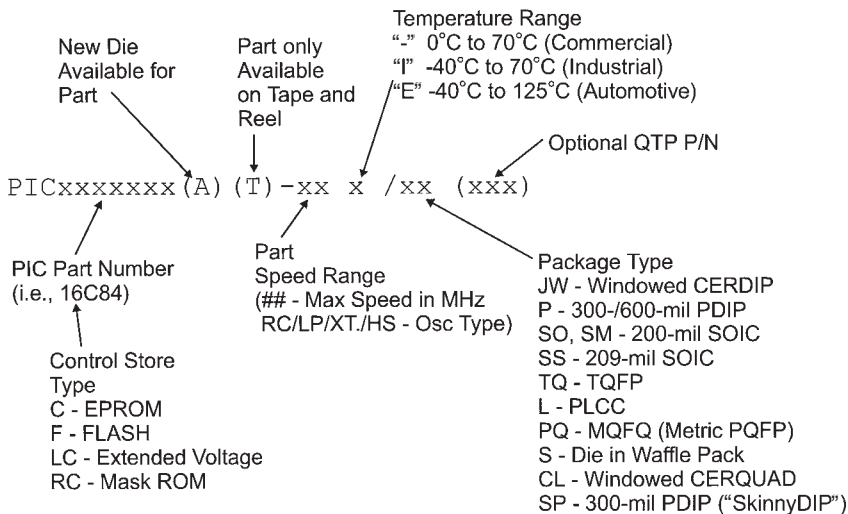


Figure 2.8 PIC microcontroller part number definition.

different versions of the part, but documentation on the differences is often very sketchy and will seem incomplete.

Microchip, like many other integrated circuit manufacturers, continually tracks the quality of their products as well as their conformance to specifications. They are also continually replacing their manufacturing equipment with newer tools that are capable of producing better quality chips with smaller device dimensions. The quality information and manufacturing process improvements make the updating of parts attractive in a variety of situations. These updates are the new letter suffixes that you will see in catalogs.

These suffixes represent an entirely new chip design (often referred to as a respin). Microchip continually updates their parts to use smaller chips as well as eliminate the use of circuits that have proved to be unreliable in manufacturing. The function (speed and features) of the part is never changed in these revisions, lest compatibility with previous versions of the device is lost. For the most part, different letter codes of the same PICmicro part number will work in an application, regardless of what its suffix letter is.

Even though I've mentioned product quality as a driver in implementing a respin, it is not the main driver. The main reason for carrying out a respin is to provide smaller chips that perform the same function. Smaller chips bring two advantages to Microchip: power reduction and part cost reduction.

Power is reduced as the part is shrunk. The longer the path an electron has to travel, the greater the overall resistance it will encounter. By reducing the distances electrical currents have to travel on the chips, the resistances (which are the cause of power dissipation within the chip) are reduced.

It may be surprising that redesigning a part reduces its cost, but even a small reduction in part size can have huge cost advantages for a chip manufacturer. In chip manufacturing, cost is directly related to the number of wafers required to build a specific number of chips. By increasing the number of chips on a wafer (by decreasing their size), the number of chips produced by the manufacturing process will increase without a significant increase in cost.

Features Unique to the PIC Microcontroller

In terms of traditional microcontroller measurements used to evaluate different chips against each other, the Microchip PIC MCUs will appear to be competitive, but not significantly more so than other devices. With a 4 MHz clock, the basic PIC microcontroller processor is capable of 1 MIPS (1 million instructions per second), which is somewhere in the middle for MCUs. PIC MCUs can have anywhere from 4 to 60 I/O pins, which again is in the range of most devices. Finally, if you were to compare the peripheral functions (which can be described as enhancements to the basic operation of the microcontroller) available in the PIC MCU to those of other microcontrollers, you would see that the PIC MCU probably has a slightly larger range of peripherals, but not so many as to make the devices unique.

There are five features of the Microchip PIC microcontrollers that do make them unique—and superior, in many ways, to other microcontrollers. The ability to select operating features of the microcontroller before boot-up allows you a great deal of flexibility in the operation of the PIC MCU and allows you to customize it for different applications. The PIC microcontroller has a variety of clocking methodologies available to you, including accurate internal clocks, which means that the PIC MCU can be wired into your application circuitry as simply as a TTL chip. The serial programming and in-circuit debugger available in many PIC microcontroller part numbers give you flexibility in how you design and debug your applications in ways that are not available in virtually all other microcontroller devices. The processor architecture will seem strange when you first start working with it, but as you become more familiar with it, it offers flexibility and the opportunity for clever coding that can't be found in traditional processor designs. Finally, the MPLAB IDE (integrated development environment) is simply the finest development suite available for any microcontroller family. Being free of charge makes it an especially useful consideration when looking at which device to use in your application. I consider these five features to be the defining reasons why the PIC MCU is superior in many ways to other microcontrollers and why customers look at this device first when choosing a microcontroller for their applications.

Elsewhere in this book, I will explain the operation of MPLAB IDE and the processor architecture of the various PIC MCU families as well as how highly optimized code can be written for it. In the following sections, I want to introduce you to some of the most important features of the PIC microcontroller, many of which are poorly understood by new developers.

CONFIGURATION FUSE REGISTER

The configuration fuse register is probably the PIC microcontroller feature that is least understood by new developers. One reason is that it is not available on other devices and another reason is due to its placement in the PIC MCU datasheets (usually in the last few pages before the electrical information and the part package dimensions). This is unfortunate, because I believe that it is probably the first feature of the PIC microcontroller that should be explained as it is part of the program memory and on power-up its contents are used to specify the initial electrical operating parameters of the chip. Many new developers do not understand how the configuration register works and often will just select parameters that allow their applications to run, without understanding exactly what they do or what advantages they can bring to an application.

The configuration fuse register (which may be referred to as configuration fuses, configuration fuse bits, configuration bits, or configuration register) is a word in program memory (at a different address for each PIC MCU processor family) in which each bit controls a different aspect of the PIC microcontroller's operation. Some of the more common ones are described in Table 2.2.

Many programmers have the capability to set these values manually, but I strongly recommend that you never take advantage of this feature. Instead, I would recommend

TABLE 2.2 COMMON PIC MICROCONTROLLER OPERATIONS CONTROLLED BY THE CONFIGURATION FUSE REGISTER

OPERATING CHARACTERISTIC	COMMON OPTIONS	COMMENTS
System clock/oscillator	External RC network, internal RC circuit, external crystal/ceramic resonator, high-speed crystal	The PIC microcontroller processor requires a clock or oscillator to run. This configuration fuse parameter helps you specify which one from a selection of different options.
WDT enable	Enable or disable the watchdog timer	The watchdog timer (WDT) is used to reset the PIC microcontroller periodically to keep control if basic operation is affected by electrical noise or electrical surges.
Reset options, including BOD enable	Internal/external reset used for the PIC microcontroller and if the brownout detect (BOD) circuit is to be used in the application	Brownout detect will force the microcontroller's reset active if incoming voltage falls below either a preset level or a programmed level.
Power-up delay	A 70 ms delay can be added to the power-up sequence	This option is used to ensure the power supply and PIC MCU clock have stabilized before releasing reset and allowing the microcontroller to execute.
Program memory protection	Prevent updating and readback of program memory's contents	In some devices, varying amounts of program memory can be protected.
Data memory protection	Prevent programmer updating and readback of data EEPROM contents	
Low voltage programming	Allow logic level programming	Normally 13 volts is required to place the PIC MCU into programming mode. When LVP is enabled, the microcontroller can be programmed without the high voltage.
MPLAB ICD	Allow pins and internal hardware to be dedicated to the MPLAB ICD debugger	The MPLAB ICD debugger is external hardware that will allow you to monitor the execution of the microcontroller, set breakpoints, and read/modify register contents.
External memory bus options	Select different execution options, including using external memory	

that you specify these values in your source code. New PIC microcontroller application developers rarely use this feature because it is not well documented or explained in Microchip datasheets.

In assembly and C language include files, there are a number of constant values that are declared as labels for each of the different configuration fuse options. To select the option, use the label with a configuration fuse directive like `__CONFIG`, which is used in assembly language.

The trick to using this directive is recognizing that the constant values are specified to only affect the appropriate bits of the configuration fuses. The bits that do not relate to the specified configuration fuse register bits are set to 1 in the constant. This way, when you want to specify a number of configuration fuse settings, you simply AND the labels together like this:

```
__CONFIG _OSC_RC & _INT_MCLR & _WDT_OFF
```

In this case, the configuration fuse settings will select an RC oscillator, use an internal reset, and disable the watchdog timer. I recommend that all configuration fuse features be listed in the `__CONFIG` directive even if you aren't planning on using them (and you will be selecting `off`) because the default value may be contrary to what you want and the PIC microcontroller will not behave as expected. In an example I will show later in the book, in most devices the default value for the watchdog timer is off, and if the `off` constant label is not selected in the `__CONFIG` statement, you will end up resetting your application once every two and a half seconds.

As application code is presented later in the book, I will discuss the configuration fuse register in greater detail for specific parts and the characteristics of options available to various chips.

INTERNAL OSCILLATOR CALIBRATION

Many microcontrollers have internal clocks. These circuits are typically implemented out of a ring oscillator, consisting of an RC network providing a delay to a ring oscillator like the one in Fig. 2.9. Typically these circuits run with an accuracy error of 20 percent or more and allow the microcontroller to be used in applications such as logic replacement or in cases where the clock speed is not critical. Using a microcontroller

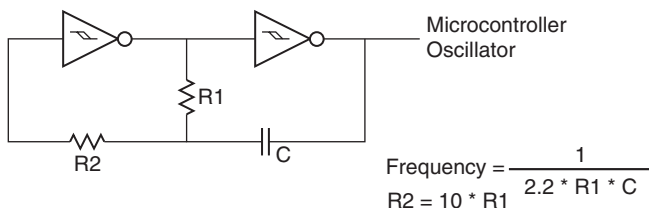


Figure 2.9 Relaxation oscillator used as an internal microcontroller clock. R1 is variable in the PIC microcontrollers with built-in oscillators.

in these cases is often cheaper and easier than wiring the circuit out of TTL logic or designing a programmable logic device for the task.

In the PIC microcontrollers that have internal clocks, Microchip has gone a step further and exchanged the fixed resistor with a programmable one. This allows the calibrating of the operating frequency to 1 or 2 percent, allowing the development of more applications that communicate with other intelligent devices or require a repeating signal within a set range. I have used this feature to create a simple circuit (consisting of a battery, a PIC microcontroller, and a few transistors and resistors) that can communicate with a PC via RS-232.

For the parts that have internal oscillators, the calibration is stored in program memory at a specific location determined by Microchip and then loaded into the PIC microcontroller's calibration registers by the application code. This calibration value is normally programmed into the PIC microcontroller by Microchip when the part is built and tested—the code to do this varies among the different PIC microcontroller families. The code for reading and storing the calibration value is discussed and demonstrated later in the book.

When you are programming an EPROM PIC microcontroller that has an internal oscillator, you have to make sure that the programmer can either read the calibration value after which you must record it down and load it into the programmer when you are reprogramming the PIC microcontroller when the application is burned into it. When I am developing applications using windowed devices, I will usually read the calibration value and write it on the underside of the chip using a Sharpie or other kind of indelible marker.

If you have a Flash-based PIC microcontroller, you will discover that many programmers will read the calibration value before erasing the device and then reprogram it with the rest of the application. Obviously, you should make sure that you understand the operation of the programmer you are using when working with Flash-based parts to know if you have to record the configuration value before erasing and reprogramming it.

IN-CIRCUIT SERIAL PROGRAMMING AND BUILT-IN DEBUGGER

Several microcontrollers have simple serial programming interfaces that allow them to be programmed using simple circuitry as well as when they are in the application circuit. Microchip has been a leader in this area as one of the first to provide a simple programming interface. Many recently released have interface hardware built in for a debugger that allows you to monitor the status of your application or stop at a predefined point and examine the contents of memory. The debugger interface minimizes the need for expensive in-circuit emulators (ICEs) while providing much of the same functionality. Both these features have been instrumental in allowing students and hobbyists the ability to work with the PIC microcontroller by lowering the cost and the technical complexity of working with microcontrollers.

Microchip's in-circuit serial programming (ICSP) protocol is a fairly conventional synchronous communications protocol with the ability to control the state of the microcontroller using a high voltage (12 to 15 V) signal on the reset pin (known as MCLR) of the PIC MCU. Once the microcontroller is in programming mode, data is shifted into

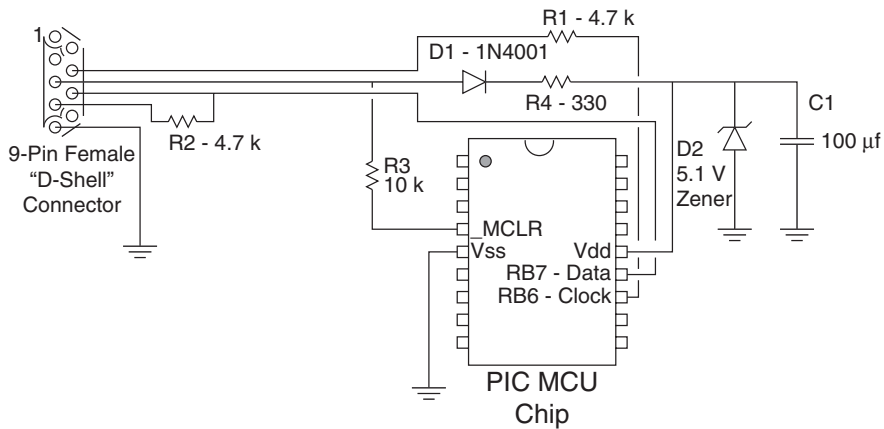


Figure 2.10 PIC MCU programmers can be as simple as this collection of discrete components connected to a PC's serial port.

it using a simple communications protocol, which is discussed in the next section. The simplicity of the programming interface allowed the creation of simple "no part" programmers (like the one shown in Fig. 2.10) for PIC microcontrollers with Flash program memory. The programmer shown in the next section follows many of the same conventions of the programmer shown in Fig. 2.10, but has additional circuitry to ensure voltage levels and avoid the need to create PC code for the synchronous serial interface.

The simple programming port also allows the use of a similar interface for debugging an application. The circuitry connected to the programming pins shown in Fig. 2.10 is quite a bit more complex but carries out essentially the same function. Microchip provides the MPLAB ICD 2 tool, which interfaces to the programming interfaces and along with programming controls the operation of the microcontroller to allow you to understand how your program executes.

The simple programming interface shown here is not actually the first programming interface that the designers of the PIC microcontroller came up with. The low-end and PIC17 parts (introduced below) have parallel programming interfaces. ICSP became available with the mid-range parts and is also used for the PIC18 family. The PIC microcontroller really became noticed by hobbyists when Microchip introduced the first of the chips with program memory based on EEPROM, which could be programmed in circuit using very simple programmers without having to go through an ultraviolet light deletion step.

PIC Microcontroller Families

There are four PIC MCU families of 8-bit microcontrollers that you can choose from for your applications. The most important difference between the families is primarily due to processor architecture built into the chip—the peripheral functions available on

the chips are largely a function of the sophistication of the processor. The more capable the processor, the more likely that sophisticated peripheral functions are available in chips built using this processor.

LOW-END PIC MCUs

When the PIC microcontroller first became available from General Instruments in the early 1980s, the device consisted of a very simple processor executing 12-bit wide instructions with basic I/O functions. Variable RAM in these devices consisted of a few tens of bytes. Some early PIC part numbers were able to work with external program memory, while others had built-in programmable ROM on board. The chips themselves were built using a number of manufacturing processes and were quite inefficient in terms of power utilization. Over the years, these microcontrollers were improved by redesigning them with CMOS technology, providing better program memory that can be easily programmed (or burned) in the field, as well as additional features, and became the PIC families of microcontrollers.

Despite these improvements, the original PIC microcontroller architecture became the low-end of the modern PIC MCU families. The devices do not have many of the features of the other PIC families, which makes them less attractive to work with for many applications. This is not to say the low-end devices are not useful and should not be considered when planning an application. Instead, they should be considered for specific application niches, chosen with the following application requirements in mind:

- Simple interface functions
- Limited variable memory
- Simple digital interfaces

The low-end PIC microcontrollers do not have a lot of program memory and cannot execute involved application code with sophisticated mathematical operations. Many devices only have program memories capable of storing 512 instructions available, with the maximum for the architecture being only 2048 (2K). This doesn't mean complex applications and code cannot be implemented, just that code with complex interface functions (and complex text interfaces) should not be implemented with low-end PIC MCUs.

When the low-end PIC microcontrollers first came out, they were given their own unique register names and conventions that differed from the mid-range devices. These variances caused some confusion for people transitioning between the parts. To help alleviate this problem, over the past few years Microchip has been changing the low-end device register names and documentation to better match the mid-ranges. This has resulted in some confusion for people who have worked with the low-end devices or who have older documentation.

I recommend that you only use the latest low-end documentation and stick with the register and resource names that match the mid-range devices to simplify the effort in moving (or porting) applications between the two architectures. With these changes to the documentation, the low-end devices have become much more like the mid-range,

although with fewer features. For this reason, I tend to call the low-end architecture a subset of the mid-range architecture.

When I first started working with PIC microcontrollers, I didn't feel that the low-end architectures were that useful, due to the limited program and variable memory, no interrupts, no advanced peripheral features, and no serial programming. This conclusion could be felt even more strongly due to the availability of the low cost mid-range parts that do not have these limitations. Microchip has kept the low-end architecture viable with the release of Flash-based products that are ideally suited for simple, digital interfacing applications. These parts are extremely low cost and can be used to replace common clocking (such as the 555 timer) and logic chips.

MID-RANGE PIC MCUs

When you look at a list of PIC microcontroller part numbers that cross-references to the processor architecture built into them, you will discover that the mid-range processor architecture is used in an overwhelming majority of the microcontrollers that Microchip makes. The mid-range PIC MCUs also have the widest range of peripheral enhancements available to any of the other PIC microcontroller families. They have amazing diversity in peripherals, memory sizes, and features, allowing you to find the solution PIC microcontroller part number for your application needs. Depending on your experience with other microcontrollers, you may be taking this statement with a grain of salt; other manufacturers have had problems supplying all the parts they advertise or only make certain part numbers available to low-volume customers. Microchip has gone to great efforts to ensure that all PICmicros are available and virtually all package types from distributors. The only exception to this would be bare dies in waffle pack shipping containers that are only available for high-volume customers ordering directly from Microchip.

If you are familiar with the classic Von Neumann architecture, all the PIC microcontroller families (not just the mid-range) will seem pretty strange to you. In this book, I present the mid-range architecture from a block diagram perspective and try to explain each aspect of it so you can understand how the instructions execute and what makes them attractive to optimizing the application. This point is probably the most important: when you are comfortable with the PIC microcontroller's architecture, you will be amazed at what you can come up with.

The mid-range PIC MCU architecture is a super-set of the low-end PIC microcontroller architecture and enhances it with the ability to access many more registers as well as be interrupted. These capabilities are taken advantage of with the addition of a number of advanced I/O peripheral devices that are available to the mid-range chips over the low-end.

PIC17 DEVICES

The PIC17Cxx PICmicro is the most different from the other three PICmicro processor architectures presented in this book. The PIC17 has the ability to interface with 8- and 16-bit parallel bus devices as well as having good built-in serial (asynchronous and synchronous) interfaces. Besides the built-in parallel bus and serial interfaces, the PIC17

also has a number of timers that have good support for pulse generation and measurement.

Development work on new PIC17 microcontrollers has slowed in recent years, and while these part numbers are still available, no new chips are planned for the future. The PIC17 can boot from either internal program memory or external memory, but in 16-bit words. Along with this, the register organization is substantially different from the other PIC MCU families, making writing code for them similar, but also making it difficult to easily port between families. For these reasons, the PIC17 is not presented with the same amount of information as the other three PIC microcontroller families.

PIC18 DEVICES

The PIC18 family of microcontrollers is well positioned to be the PIC architecture family of choice, having peripheral features similar to that of the mid-range family with an enhanced processor that has significantly increased capabilities.

The PIC18 processor offers the following advantages:

- Up to one megabyte of instructions can be addressed in the program memory
- Up to 4K of file and hardware registers
- A software accessible stack
- Extended oscillator options
- Enhanced ICD capabilities, including multiple breakpoints

The PIC18 has a 16-bit instruction word and instruction set that is “source code” compatible with the mid-range devices. This level of compatibility makes it quite easy to work with the low-end, mid-range, and PIC18 families of devices and share code and programming tricks between the architectures. While increased memory access and more oscillator options are useful advantages of the PIC18 over the other PIC microcontroller families, the ability to read and write to the stack is very exciting. As I will show later in the book, this feature can be used to implement a true real-time operating system—an important application difference between the PIC18 and the other PIC architecture families.

SOFTWARE DEVELOPMENT TOOLS

When I first started working with the PIC[®] microcontroller, Microchip provided MS-DOS command-line tools that performed the task of converting assembly language source files into hex files as well as simulating the operation of the program files as if they were working in actual hardware. The MS-DOS command-line tools were competent, but quite clunky, and it was difficult to see the full picture of what was happening during application execution. Fig. 3.1 shows a screen shot of MPSIM (the MS-DOS command-line simulator) executing. This tool is a program simulator that requires a user to provide a configuration file (known as the MPSIM-INI), which specifies which registers are to be monitored by the user as the application executes. Stimulus files could be added to the command-line simulator to avoid the need for the user to add specific inputs manually. Despite working well enough to be recommended as the development tools for the readers of the first edition of this book, the command-line tools required a bit of learning and effort to provide the necessary capabilities for the PIC microcontroller application developer.

After releasing the manuscript of the first edition of this book, Microchip released version 3.00 of the MPLAB integrated development environment (or IDE, shown in Fig. 3.2). This Microsoft Windows application eliminated a lot of the difficulty in developing and simulating a PIC microcontroller application that was inherent in the MS-DOS command-line environment. MPLAB IDE provide the user with a tool that combined an editor, assembler, compiler, linker, simulator, and emulator as well as a programmer interface that allowed the user to burn the application code into a PIC MCU chip. This all-in-one tool is excellent for new PIC microcontroller developers and is a great way to see your applications executing and responding to various input conditions. This book focuses on the MPLAB IDE and uses it exclusively for PIC microcontroller application development because of the ease with which applications can be created with it.

In this chapter, I will introduce you to the tools available for PIC MCU application development, from editors to assemblers, compilers, simulators, and emulators. Although I focus on the MPLAB IDE in this book, in this chapter I will also present some other

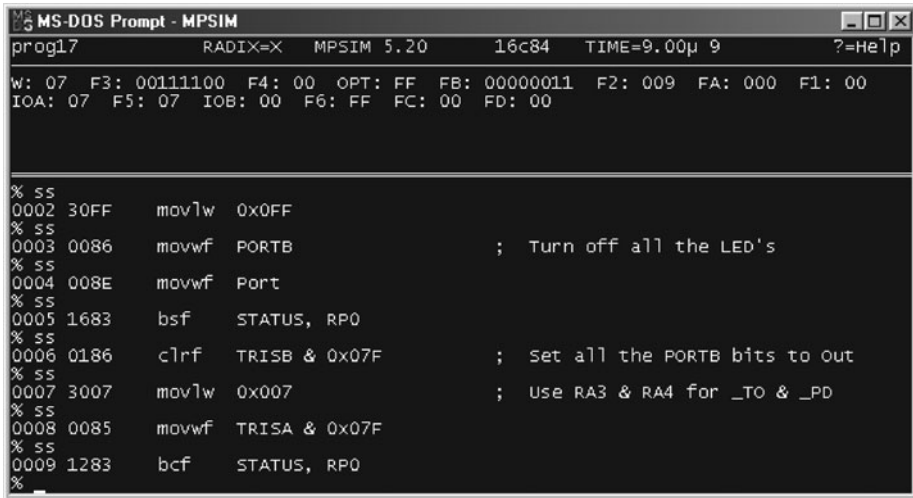


Figure 3.1 The first PIC microcontroller development tools were MS-DOS command-line programs like the MPSIM.

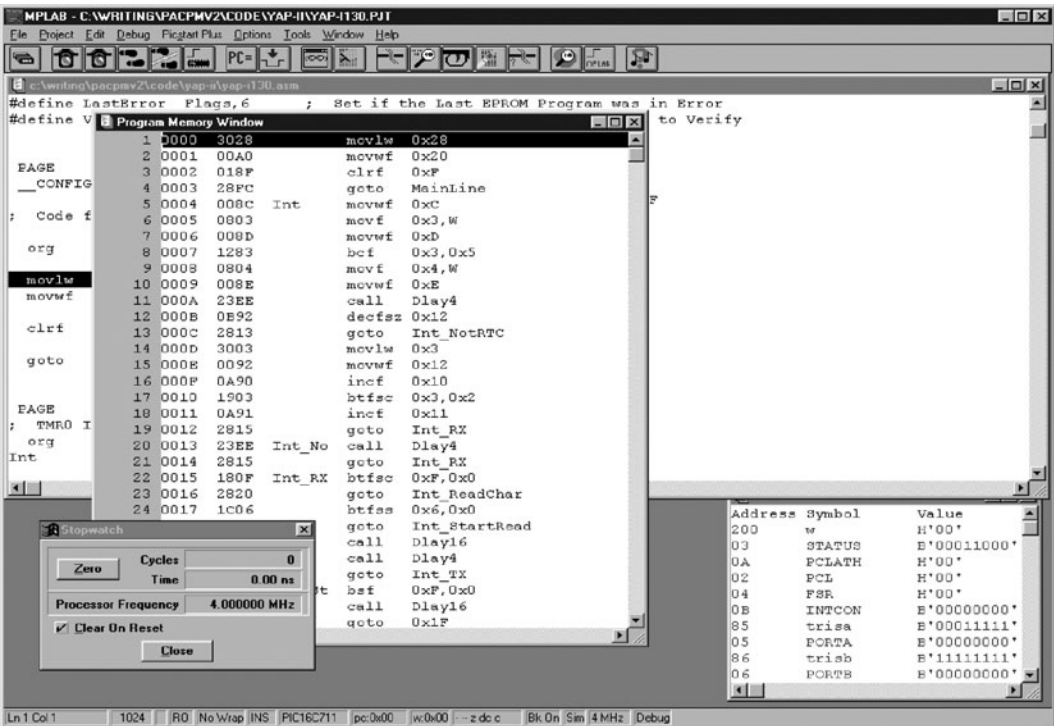


Figure 3.2 Microchip's MPLAB integrated development environment simplifies the task of developing and configuring a PIC microcontroller development system.

tools that you can use for PIC microcontroller development. The tools available for creating PIC microcontroller application run the gamut from freeware and shareware to GNU tools and commercial products.

Tools Overview

Regardless of the software programs that you use to develop your PIC microcontroller application software, they will all provide the same basic functions. To develop your source code, you will need an editor program that allows you to enter program statements as well as modify and save them. Once this is done, you will have to convert them into object hex files using an assembler or compiler. You may wish to combine converted code from different sources and build them into the final application code using a linker. Once you have built your application, you will want to test them in a simulator before programming the application into either a part or an emulator, which will allow you to follow the progress of the application (as part of qualifying or debugging the code). These tools go by various names and may be integrated together into single, larger programs, but in every development system, these functions are necessary to create microcontroller applications.

EDITORS

An editor is an application program that runs on a PC or workstation to allow a “human-readable” file to be created or changed. The editor can also be used for reviewing data located in files (which you may refer to as “browsing”). Over the years, I have probably tried a hundred or more different editors for developing and modifying application code, browsing and changing hex files, creating text (like this book or web page HTML), and sending email. Most of these trials were made at the suggestion of someone else because they had found a “wonderful new editor.” Of all the editors I’ve tried, I’ve only found two that I’ve liked and used for any length of time. For standard editing requirements, I just use the standard Microsoft Windows WordPad, Notepad, and Word editors. I find that I’m very picky in what I consider to be an outstanding editor and for the most part, I rely on the Microsoft data entry standards with basic text editing features. The editor in Microchip’s MPLAB IDE uses the standard Microsoft editing conventions and allows for easy editing of PIC microcontroller source code files.

WordPad (Fig. 3.3) is a Word-based editor. The cursor, which is the vertical bar where characters will be placed, is moved onto the window by either the arrow keys or by using the mouse and setting its position with a left click. When I’m editing a file, I very rarely use the mouse; instead I use the arrow keys, Home, End, Page Up, and Page Down almost exclusively.

To delete and move text, I use the keystroke operations listed in Table 3.1. To select text to cut and paste, I mark the text first pressing the Shift key while moving the cursor to highlight the text to be relocated. If you have marked text incorrectly, simply move the cursor without the Shift key pressed to delete the marked text. Pressing your mouse’s left button and moving the mouse across the desired text will also mark it. Left-clicking

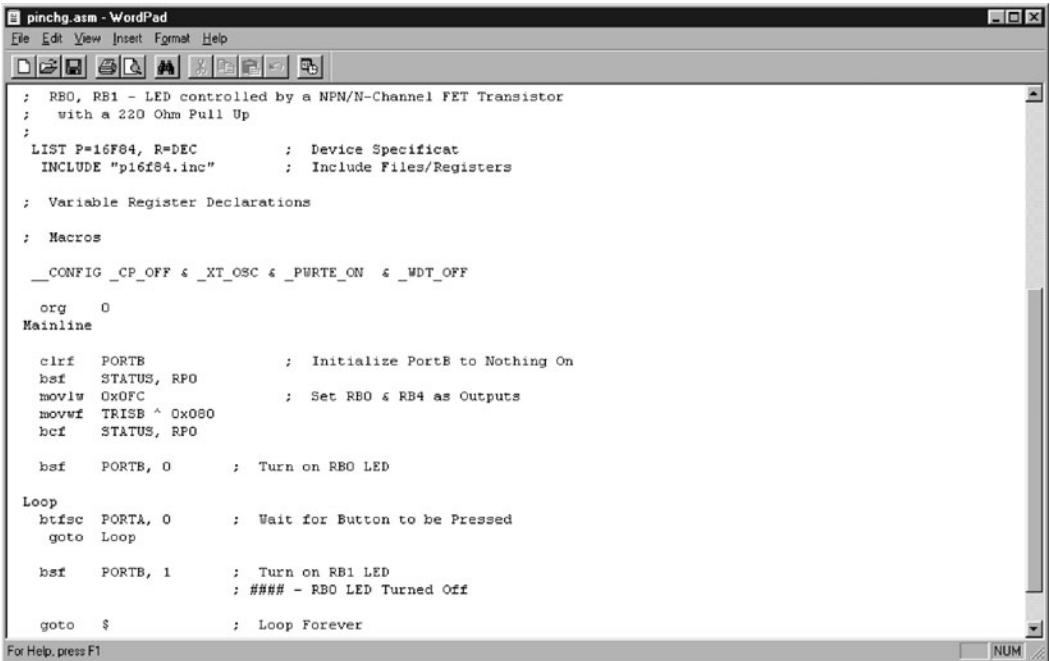


Figure 3.3 Microsoft's WordPad editor allows easy reading and changing of data in text files.

on another part of the screen will move the cursor there and eliminate the highlighting for the text. Next, the keystrokes Ctrl-X are used; this removes the marked text from the file and places it into the Windows clipboard. Ctrl-C copies the marked text into the clipboard and doesn't delete it. To put the text at a specific location within a file after the current cursor location, Ctrl-V is used. Instead of these keys, you can click on the pull-down Edit menu and select Cut, Copy, or Paste to perform these functions.

Note that I do not use Delete or Insert. Deleting marked text destroys it completely whereas Ctrl-C saves it in the clipboard so it can be restored if you made a mistake. The Insert key toggles the editor between data Insert and Replace modes. Normally when an editor boots up, it is in Insert mode, which means any keystrokes are placed. This is the preferable mode to be in.

Some early PC editors were line-based rather than paragraph-based like standard Microsoft compatible editors. In a line-based editor, a CR/LF character combination is saved at the end of each line displayed on the screen. In a Microsoft compatible editor, the CR/LF is used to separate paragraphs and is not inserted automatically. If you were to look at a paragraph produced by a Microsoft compatible editor on a line editor, you would find that the paragraph would be one line and most of it would not be displayed because it was past the right edge of the editor's text window. An advantage of a line-based editor over the Microsoft compatible editor is the ease with which blocks of data can be moved.

When selecting a text editor, you may wish to choose one that is programmable and will enhance the programming experience. An example of this is an editor that responds

TABLE 3.1 STANDARD MICROSOFT EDITOR KEYSTROKE OPERATIONS

KEYSTROKES	OPERATION
Up arrow	Move cursor up one line
Down arrow	Move cursor down one line
Left arrow	Move cursor left one character
Right arrow	Move cursor right one line
Page Up	Move viewed window up
Page Down	Move viewed window down
Ctrl-left arrow	Jump to start of word
Ctrl-right arrow	Jump to start of next word
Ctrl-Page Up	Move cursor to top of viewed window
Ctrl-Page Down	Move cursor to bottom of viewed window
Home	Move cursor to start of line
End	Move cursor to end of line
Ctrl-Home	Jump to start of file
Ctrl-End	Jump to end of file
Shift-left arrow	Increase the marked block by one character to the left
Shift-right arrow	Increase the marked block by one character to the right
Shift-up arrow	Increase the marked block by one line up
Shift-down arrow	Increase the marked block by one line down
Ctrl/Shift-left arrow	Increase the marked block by one word to the left
Ctrl/Shift-right arrow	Increase the marked block by one word to the right

to specific keystroke sequences and enters additional text for you in a specific format. An example of this type of function in a programmable editor that has been customized is an enhancement that responds to a basic key sequence like:

```
if
```

by inserting the additional text:

```
if ( ) {
} else {
} // endif
```

I also keep a bit editor handy, in case I want to look at a binary file. A bit editor is tool that shows each byte as two nybbles. This tool can be useful for patching data files

although I do not recommend patching PIC MCU hex files. Bit editors usually have a difference function, which allows you to compare two hex files to find differences between them. This function is similar to diff utilities, which also allow you to compare two hex files and have the differences flagged between them. A useful function of a bit editor is the ability to display data in different formats; for example, if you wanted to find a specific string in a hex file, the character display can make this effort much simpler.

One thing you must watch out with older editors (which you may download from the Internet to try out) is that they may put a 0x1A character at the end of a file. The 0x1A character was required by MS-DOS 1.x to indicate a file's end. For some reason, even though MS-DOS and Windows no longer require this file end delimiter and have become more sophisticated in their file handling, this file end indicator has not completely disappeared. In editors that do not place the character at the end of the file, this character is treated as part of the file and displayed as a small square box at the end of the file. In general, it can be deleted without problem. MPLAB and most other tools usually do not have a problem with this character placed at the end of the file but some other tools do. If you are using a tool where this is a problem, you can simply delete the character and resave the file before passing it again to the tool.

ASSEMBLERS

The most popular way to program the PIC microcontroller is to use the Microchip MPASM assembler program (usually just called an assembler), which can be downloaded free of charge from Microchip's Web site and used with and without the MPLAB integrated development environment. An assembler converts a set of human-readable assembler instructions (often called mnemonics) into bit patterns that will be programmed into the PIC microcontroller and executed by the processor.

For the example assembler instructions:

```
movlw    7
Loop:
    addlw    0 - 1
    btfsc    STATUS, Z
    goto     Loop
```

An assembler performs the following operations. First, it determines the program counter address for the first assembler instruction that is encountered (`movlw 7`). Next, it reads through the first parameter of the assembler instruction and if it is a label, it marks it for later translation into its numeric value. Along with the instruction's code, it also identifies the expected parameters. If the label is not in the instruction table, the assembler assumes the string is an address label and stores the value in an address label. This operation is known as a pass and processes the entire source file before going on.

If the example code above started at address 0x0123, the first pass of the assembler would produce the information listed in Table 3.2.

Once this pass is complete, a second pass is executed in which the parameter values for each instruction are evaluated and added to the instruction's numeric. When the

TABLE 3.2 MPASM FIRST PASS ASSEMBLY OF A BLOCK OF CODE

CODE	ADDRESS	NUMERIC	EXPECTED PARAMETERS	LABEL VALUE
movlw 7	0x123	0x3000	8-bit value	
Loop:				Loop = 124
addlw 0 – 1	0x124	0x3E00	8-bit value	
btfsc STATUS, Z	0x125	0x1C00	Register, Bit#	
goto Loop	0x126	0x2800	11-bit value	

parameters are evaluated, any labels that are found are checked against a file register/address label table. The second pass produces complete instruction values as shown in Table 3.3.

In the parameter evaluation, MPASM produces a 32-bit value that is used from the least significant bits upwards according to the instruction format. This is done by what I call the assembler calculator, which is discussed in detail in Chap. 10—“Macro Development.” In the third line (`addlw 0 – 1`), the 32-bit value for negative one (–1) is:

0x0FFFFFFF

even though the least significant 8 bits are used with the instruction. Note that when the instruction is first decoded, the parameter values are left zeroed. When the parameters are evaluated, the correct values can be simply added to the instruction numeric in the second pass to create a correct instruction for the PIC microcontroller’s processor.

The MPASM assembler works almost identically to the description above but with a few differences. The first is the inclusion of a macro processor, which inserts macros into the source code. The MPASM assembler also accesses one of four instruction types to numeric and parameter type tables based on the type of PIC microcontroller the source is written for. Lastly, MPASM imbeds included files into the source code before the passes begin. Once these operations are complete, the two-pass assembler can be invoked

TABLE 3.3 RESULTS AFTER SECOND PASS OF THE ASSEMBLER

CODE	ADDRESS	NUMERIC
movlw 7	0x123	0x3007
Loop:		
addlw 0 – 1	0x124	0x3EFF
btfsc STATUS, Z	0x125	0x1D03
goto Loop	0x126	0x2924

to convert the resulting source into instruction bit pattern for the PIC microcontroller's processor. This initial step and operations are also very common for many assemblers.

Errors in the MPASM assembler can be flagged in one of three categories: error, warning, or message. Errors indicate that there are significant problems with the source code that will have to be corrected before the application can be programmed into a PIC microcontroller. Warnings indicate that the code does not follow the Microchip recommended format and may not work as you expect. Messages are used to indicate that something looks funny and while the code will probably run, you should make sure you understand what the problem is.

As I will say throughout this book, you should never attempt to program a PIC microcontroller and expect it to work properly if there are any errors, warnings, or messages. In some of the examples here, I will show what kind of messages you can get and then show what happens when the assembler messages are ignored. The errors, warnings, and messages are well thought out and if something comes up, you should heed them. Chances are, they will save you from having to debug a problem on the PIC microcontroller.

This also goes for suppressing messages. In the first edition, I noted that leaving off the `, f` when an operations destination is put back into the source register could be done with only a message being displayed. In the three years since writing that book, I've discovered cases where I have had problems with an application because I left off the `, w` parameter from the instruction. Now I very rarely suppress any messages or warnings and only program a PIC microcontroller if the application assembles cleanly (without any errors, warnings, or messages).

The only time I do suppress a message or warning is when I expect it. In these cases, I suppress the message or warning generation, place the instruction, and then enable the message or warning generation. This way, I acknowledge the expected message or warning and am notified if it comes up again elsewhere (and unexpectedly) in the application source code.

COMPILERS

When you first see the results of a compiler converting high level source code into assembly language, you will probably feel like the process of changing high level language source code into processor assembly code is more a result of magic than a series of mathematical operations executed by a computer. As well as converting the source code into assembly language, modern compilers also look for opportunities to simplify the code that is output, resulting in smaller and more efficient applications. If you are a beginner with PIC microcontroller assembly language development, you should not be surprised to discover that modern compilers can produce more efficient assembly code than *you* can. The low-end and mid-range PIC microcontrollers may seem like they are poorly designed for compilers to develop efficient code for them, but efficient compilers can be created for them reasonably easily if the compiler is well thought out beforehand.

Throughout this book, you will see statements like:

```
A = B + (C * D);
```

TABLE 3.4 THE FIVE BASIC TYPES OF HIGH LEVEL PROGRAMMING STATEMENTS

STATEMENT FORMAT	TYPE
variable = . . .	Assignment statement
if (. . .)	Conditional test statement
label (. . .)	Subroutine/function call
type variable [= constant]	Variable declaration
type label(variable, . . .)	Subroutine/function declaration

To convert this statement into assembly language, a number of steps have to be carried out. Most compilers take advantage of a processor data stack for working through these types of instructions and keeping track of temporary values. When processing the statements, the values are pushed onto the stack in the reverse order that they are required. When the statement is executed, this data is “popped” off the stack and processed. Because most of the PIC microcontroller architectures do not have a built-in data stack, the compiler developer will have to decide how to “push” and “pop” the data. To explain how compilers work, I will introduce you to their operation using a stack and then discuss the options available in the PIC microcontroller.

The first thing a compiler does is determine the type of statement it has to work on next. In most high level languages, there are the five types of statements listed in Table 3.4. In the first types (assignment statement, conditional execution, and subroutine/function call), the part or the statement shown as . . . can be considered as the statement. This is the data that will be put on the data stack and then executed. Putting the statement in a heap stored in a postfix order (the least significant operations given the highest priority) does this.

For the operation in the assignment statement:

A = B + (C * D);

The postfix heap is shown in Fig. 3.4.

Next, the order of operations is determined by pulling the data from the heap in a prefix (left to right) order. In this operation, the lowest operation on the left is pushed onto the stack followed by the lowest on the right and then the values required for the operations

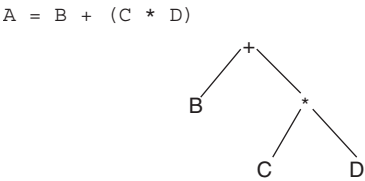


Figure 3.4 Compiler postfix heap for a high level language statement.

are pushed onto the stack. When the operator executes, it pops the previous two (or one) stack elements and then pushes the result onto the stack. The stack operations for:

`A = B + (C * D);`

are:

Push	D
Push	C
Execute	*
Push	B
Execute	+
Pop	A

In this sequence of stack operations, D, followed by C, is pushed onto the stack. Next, they are popped off the stack and multiplied together and the product is pushed onto the stack. The product of `C * D` is popped off the stack along with B and added together with the result pushed onto the stack. To finish off the instruction, the final result—`B+ (C*D)`—is popped from the stack and stored in A.

Depending on your age, you may remember Hewlett-Packard calculators that worked this way. Data entry took the form of the stack instructions listed above and were known as reverse polish notation (RPN). It took a bit of getting used to, but once you were able to think in RPN it actually was easier working through complex problems because you didn't have to remember how deep the parentheses of the expression were. In universities and colleges all over the world, the truly cool people could think in RPN and not use a pencil and paper to plan out how they were going to enter statements into their calculators.

Leaving a result on the stack is important for the other two types of statements. In the `if` statement, if the value on the top of the stack is not equal to zero, then the condition is determined to be true. As will be discussed below, for the subroutine/function call statement, the parameters passed to the subroutine/function are accessed from the stack by the subroutine/function code according to their position relative to the top of the stack. Array elements are also stack values that are popped off when an element is to be accessed.

In the PIC microcontroller, a data stack can be implemented using the FSR register. To push an element onto the stack of the mid-range processors, you could use the code:

```
incf    FSR, f
movwf   INDF
```

and to pop a value, the code

```
movf    INDF
decf    FSR, f
```

could be used. The push snippet increments the FSR before writing to the stack to ensure there is no way the stack values can be corrupted if the pop operation is interrupted halfway through. The biggest problem with this method is that the FSR index register is dedicated

to the compiled code and is not available to any linked-in user written assembly code. In this case, the contents of FSR could be stored in a temporary variable before it is changed into assembly language code and restored upon leaving the assembly language code.

Another way the PIC microcontroller can implement a data stack is to use temporary variables and access them directly. For the assignment statement example above, the operations would be:

```
Temp1 = C;
Temp2 = D;
Temp3 = B;

Temp1 = Temp1 * Temp2;
Temp1 = Temp1 + Temp3;

A = temp 1
```

This method gets very complex when statements that have more than one stack entry are left on the stack during execution. The two types of statements that best come to mind are ones that use array elements and those that call subroutines or functions that require more than one input parameter. In these cases, the data is evaluated and left on the stack for later operation.

Adding an array element read to the example statement:

```
A = B + (C[4] * D);
```

would result in the postfix heap shown in Fig. 3.5 and the stack operations:

Push	D
Push	4
Push	C[
Execute	*
Push	B
Execute	+

In this example, the order of operations would be to push D onto the stack, followed by 4. When the Push C[operation was encountered, the previous element (the 4)

```
A = B + (C[4] * D)
```

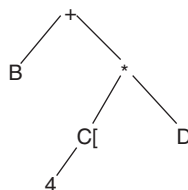


Figure 3.5 Postfix heap with array element pushed onto the stack.

would be popped off the stack and used as the index into array C. Once C[4] was evaluated, it would be pushed onto the stack and the operation would continue as before with the result left on the stack.

Many subroutines and functions have multiple parameters passed to them. For the function:

```
int Func(int varA, int varB)
```

a calling assignment statement could be:

```
A = B + Func(C[4], D);
```

which would generate the postfix heap shown in Fig. 3.6 and the stack loading:

```
Push      D
Push      4
Push      C [
Call       Func
Push      B
Execute    +
```

The stack is executed through similarly to the previous example, but when the `Func` call is encountered, the two previous values are left on the stack and then `Func` is called with them as local variable arguments. In `Func`, the two arguments are referenced according to their position relative to the top of the stack; the first parameter is one position below the stack top while the second parameter is the stack top.

If `Func` was:

```
Int Func(int varA, int varB)
{
    varA = varA + 1;
    return varA * varB;
} // end Func
```

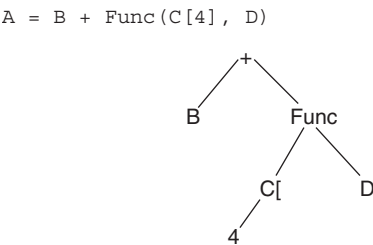


Figure 3.6 Postfix heap showing a function call with an array element as an argument.

the actual compiled code for the function could be:

```
Func:
; varA = varA + 1
  Push      StackTop - 1      ; Push varA as the new stack top
  Push      1                  ; Push 1 onto the stack
  Execute   +                  ; Pop varA and 1, add, push result
  Pop       StackTop - 1      ; Pop stack top and store in varA
; return varA * varB
  Push      StackTop - 1      ; Push varA as the new stack top
  Push      StackTop          ; Push varB as the new stack top
  Execute   *                  ; Pop varA and varB, multiply, push
                               ; result
  return
```

In the calling code after the initial call statement, the first stack item would be popped off and saved and the next two would be popped off and discarded. The saved value would then be pushed onto the stack and execution would continue.

Another way of doing this would be to pop the new top value off the stack and place it as the first stack element before the call. Once this is done, any other values could be popped off the stack and discarded. The code making this call and popping off the unneeded stack elements would look like this:

```
Push      varA
Push      varB
Call      Func
Pop       StackTop - 2        ; Put result in the first parameter
                               ; position
Pop       BitBucket          ; Get rid of second parameter (varB)
; Result of Func is the top stack element
```

When you look at actual code produced by a compiler, you will probably see deviations from the strict stack operations outlined in this section. Optimizing code in the compiler causes these deviations. For example, statements like:

```
A = B + (C * (4 * 2));
```

should be executed as:

```
A = B + (C * 8);
```

with the stack operations looking like:

```
Push      8
Push      C
Execute   *
Push      B
Execute   +
Pop       A
```

In this case, the data stack may not be required at all by the compiler. Instead the operations could be carried out using temporary registers. The only time a stack would be needed is if the assignment statement executed in a subroutine that was called by the main line code and the interrupt handler, or if it was inside a recursive subroutine.

There is one issue regarding compiled code that you should be aware of when working with the PIC microcontroller and that is to make sure that subtraction is handled correctly. Addition (and bitwise) operators can be handled in the order in which they were received. The value being subtracted has to be stored in *w* before the subtraction operation executes, in which the value in *w* is subtracted from the subtraction instruction value. The unusual operation of the subtraction instruction complicates the operation of the post-fix heap and stack operations for subtraction as well as comparison operations.

LINKERS

Object files (which normally end in *.obj*) can be produced by a compiler or assembler instead of *.hex* files (which are complete applications). Object files are portions of an application that are linked together to create a complete application. Linker programs are very complex and not only put multiple object files together but also provide address references between the object files. They can also be used in conjunction with make files, which are used to specify how the final application is put together. In this book, I have concentrated on just presenting autonomous applications that can be built and then burned into a PIC MCU, but if you are working on large programs or ones that have large shared portions, you may want to consider building smaller pieces into *.obj* files and linking them together.

An important function of the linker is to provide address references between *.obj* files. Object files are very similar to hex files except for labels that are going to be referenced outside the application are flagged and not deleted. References to addresses that are outside the current object file are also flagged. One of the purposes of a linker is to resolve the references to external labels with the public labels in the object files and correlate the addresses to the instructions that use them.

Another function of the linker is to provide addresses for the various object files. To do this, the code must be written so that it is relocatable or able to be located anywhere within PIC microcontroller program memory and not tied to any one location. For high level languages, this is usually not a problem, although it can make assembly language development more difficult. To show how relocatable codes are used in a linked application, consider the case where three object files have to be linked together with their parameters, as listed in Table 3.5.

TABLE 3.5 LINKER FILE LENGTHS AND POSITIONING		
FILE	LENGTH	COMMENTS
ObjectA	100 instructions	Contains application header; this file must <i>always</i> be first
ObjectB	200 instructions	
ObjectC	14 instructions	

In this example, ObjectB or ObjectC can be put in any order after ObjectA. This means that ObjectB can start at address 100 or address 250 depending on whether or not ObjectC is between it and ObjectA. Once the linker has calculated the addresses for each object file, it will calculate the addresses for the labels that are accessed outside of each object file and put the addresses into the code.

It may be possible to convert object files into libraries. A library differs from an object file in its ability to store pieces of the code within it rather than the entire file. When you write applications in the C programming language, when the application is built, it is linked to a library of standard C functions (which are discussed in the appendices), but only functions required by the application are linked into it. The full C function library is probably larger than the amount of available program memory, so it is important that only the functions that are required to run the application are linked into the final hex file. Before converting an object file into a library, remember to consult with the linker's technical reference to make sure any dependencies or restrictions (such as not being able to call other functions and subroutines in the library) are not violated.

The Microchip linker currently does not have a make capability, but many other linkers do. A make file is a program that specifies how an application is to be built. The make utility executes the make program and selects files for assembly, compiling, and linking with options. The make file can be very complex, with different options specified for conditional debug or partial linking. Because the make utility can also assemble and compile source files, parameters can be passed from it to the assembler and linker as well.

Linked applications make a lot of sense when:

- Multiple languages are involved (such as putting assembly and C programming language source files together in one application)
- There is a previously built library file that is required for the object code used within the application
- There are multiple people involved with the application development
- The source application is very long (longer than 10,000 lines)

In these cases, linking object files together will make the application easier to understand and probably faster to create and debug.

SIMULATORS

Throughout this book, I stress the importance of simulators when developing PIC microcontroller applications. A simulator is a software tool in which the operation of an application can be observed executing, allowing you to find and fix problems much easier than if you program a PIC microcontroller and try out some code after writing it. For inexperienced developers, I consider the simulator to be the most useful tool in your arsenal to test your application and verify that it will work as you expect. The simulator that is built into MPLAB IDE is an outstanding tool and one you can quickly learn.

A simulator consists of a software model of the PIC microcontroller processor, which can be controlled along with the ability to pass basic I/O signals back and forth with

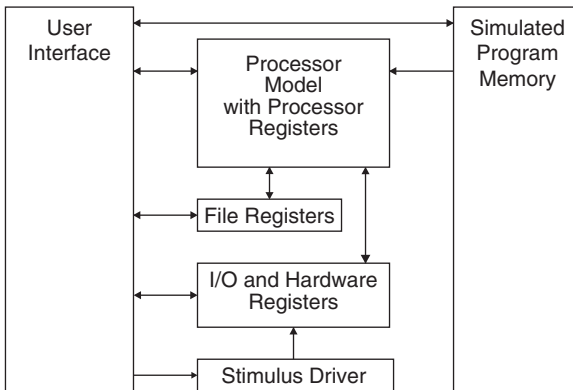


Figure 3.7 The software simulator consists of a number of program modules that perform the basic functions of the PIC microcontroller hardware.

the application. It is important to remember that the processor and I/O in a simulator are virtual, and there will be situations that can come up in the simulator in which the actual hardware is not fully or properly simulated. The MPLAB IDE simulator is an excellent tool for testing out application code and observing the execution of algorithms, but it is somewhat limited in its ability to model advanced peripheral I/O ports.

I tend to think of a simulator as a collection of “black boxes,” which are controlled by the simulator host software as shown in Fig. 3.7. I drew the simulator this way because it allows boxes to be swapped in and out to make up different part numbers without significant effort required to create simulators for different functions. There are probably more actual simulator modules used in MPLAB IDE, but for the purposes of this discussion, the five presented in Fig. 3.7 are adequate.

The program memory block is loaded with the hex file that will be programmed into the PIC microcontroller. The processor model will pull data from this simulated program memory as required. The file registers are similar, but the processor model can read and write to the file registers. The I/O and hardware registers block provides some kind of model of the I/O pins. For basic digital I/O, the MPLAB IDE simulator provides a good model of the I/O hardware, but for advanced peripherals (the USART, MSSP, and other peripherals), the simulator model does little more than just accept input from the user interface, which is stored in the registers.

To help you debug your applications, the I/O pins have the capability of being driven with external inputs. This is the stimulus box shown in Fig. 3.7. It can be directly used in changing I/O register bits or can run a stimulus script, which can be either a user file or generated from the MPLAB IDE user interface. A stimulus script is created with I/O information that can be processed by the simulator processor without intervention by the user. This allows the user to create a test routine to examine why code is failing as well as test changes to the application to see if the problem has been fixed. I believe that being able to create stimulus information for your applications is critical to being able to test and verify their operation before burning them into a PIC microcontroller.

The topic of stimulus files brings up an important point. When debugging an application, the simulator must be set up so that it will always run the same way. This philosophical point is important and one that I want to make sure you follow. If the simulated application runs differently each time, you will have a hard time trying to work through problems and fixing them. When you debug your application using a simulator, you should focus on fixing the problem, not repeatedly setting up the simulator and stimulus for testing a piece of code.

The processor block is obviously the heart of the simulator, with the user interface commanding it to execute, execute to breakpoint, single step, or stop. The processor model is an extremely complex piece of software. Not only does it have to fetch and execute instructions as well as access registers, but it also has to manage such peripheral functions as TMRO interrupts and the watchdog timer. Making the design of this module even more complex, the execution of the simulated instructions must be as fast as possible.

The user interface is the primary window into the application code and as such should be as configurable as possible to allow for user preferences as well as displaying variables and I/O registers based on the user's preferences. Along with displaying customizable registers, the simulator should work through the code source, rather than the simple instructions.

In the first edition of this book, I provided Microchip's MPSIM MS-DOS command-line simulator on the diskette that came with the book. This simulator provided all the features I have discussed in this section except for a full source file display. The ability to see how code executes from a source code view is critical for me and made working with MPSIM difficult. Having this capability in the MPLAB IDE simulator makes debugging application software much easier and more efficient.

EMULATORS

The next step up from a simulator is a chip emulator, which replaces the PIC microcontroller in your hardware application and provides a connection to your PC, allowing you to monitor the execution of the application in hardware. The emulator block diagram looks like Fig. 3.8 and shows that the emulator consists of a piece of hardware plugged into the application to replace the PIC microcontroller. The MCU replacement

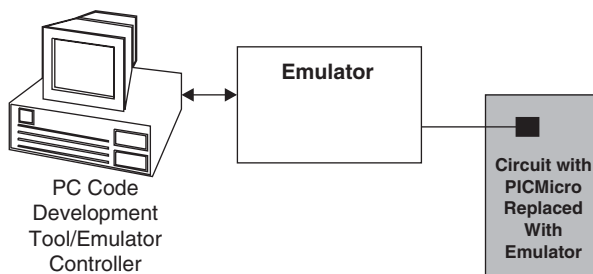


Figure 3.8 A microcontroller emulator consists of a chip replacement connected to external hardware that is controlled by a PC.

hardware is usually connected to a block of hardware that provides control and operational monitoring of the emulator chip. To ensure that there are no critical time delays between the application circuitry and the controlling hardware, the cable connecting the two is usually quite short, often less than 6 inches (15 cm). An advantage of an emulator over the simulator is that actual pin I/O signals can be observed, both from the processor's perspective as well as from the circuits. As well, the emulator often has the same hardware as the actual device, so there are no missing peripheral interface functions.

The best method of providing an emulator is for the manufacturer to provide a "bond-out" chip. This chip is built from the same silicon as the actual microcontroller but has connections to memory and the chip execution control hardware, which allows an external device to control and monitor its operation. Microchip designs the PIC microcontroller chips with emulation in mind; the chips all contain the emulator functions, but when used in a typical application, the pads on the chip providing these functions are normally left unconnected. The bond-out devices are the same chips that are used in the applications so the operation of the bond-out chip in the application will be identical to that of the packaged chip that will be used in the final application.

There are two disadvantages to working with an emulator. The first one is cost. Emulators generally cost \$2,000 or more and require separate pods for each device being emulated (which makes sense because each device is the unique chip). This cost is not significant for many companies, but for small companies and individuals, it can be. To help offset the costs, there are options such as Microchip's MPLAB ICD 2 debugger (discussed in the next section). The other problem with emulators is the difficulty in connecting them to a circuit and the unreliability of the connection once it is in the circuit. Microchip's MPLAB-2000 emulator provides a small tripod to hold the emulator control hardware close to the target circuit without placing any strain on the cable connection.

HARDWARE DEBUGGERS

If you have been involved in the low-level development of high performance microcontroller or processor applications, you are probably familiar with the use of JTAG (Joint Test Action Group standard 1149.1) debugging hardware, which allows a developer to take over execution of the processor. By taking over the execution of the processor, the application code can be tested in the actual hardware, allowing the developer to see what values are being returned from external hardware or if there are events taking place in the application that are not expected. The four-pin JTAG interface takes up little space in the application circuit and is quite fast. Not only is it used for controlling the operation of the processor, but it can be used to download applications or upload data for analysis on a host PC. The JTAG interface provides many of the functions of the emulator without the cost or the restrictions of a bond-out chip wired into an application.

For many PIC microcontrollers, Microchip has provided much of the same functionality as the JTAG port via the MPLAB ICD 2 debugger. The base function of this relatively inexpensive interface is the ability to program a PIC MCU with application

code in circuit, without the need for a separate programmer. It also gives you the capability to single-step through your applications, stopping either at a breakpoint or at some arbitrary point in time, and allows you to read back data or change the register values of the microcontroller. Like the JTAG interface, it only requires a few pins and is very easy to implement an ICD connector in your application, giving you the capability to program and debug your application code in the actual circuit. If you are interested in having a stand-alone programmer, there are ICD PCBs available with ZIF sockets that will allow you to program chips separately. For new PIC MCU developers, the MPLAB ICD 2 is a very wise investment. It will aid you in learning about the PIC MCU and you will find it invaluable in the development of commercial grade applications.

It is not usually recognized, but the MPLAB ICD 2 hardware has a significant advantage over an emulator: it cannot be easily damaged if there is a problem with your circuitry. The emulator bond-out chip can be damaged just like any other chip by applying high voltages to the I/O pins, but the ICD module connects to the PIC MCU through dedicated pins, which provide a measure of protection to the module. In the unlikely case that the module is damaged, replacing it costs approximately as much as a single bond-out chip. The built-in power supply of the ICD module cannot provide a great deal of power, but it will allow you to drive a few LEDs, avoiding the need for a separate power solution when you are developing your first, basic applications.

As good as the MPLAB ICD 2 is, there are a few deficiencies that you should be aware of. The first is that it can be painfully slow to respond to commands such as single-stepping. This is a function of the serial communications with the chip and the USB connection to the host PC. For small pin count chips, you will have to buy a PCB module, which consists of a similar chip with extra I/O pins that provide the connection to the ICD connector—this module provides similar capabilities to the emulator bond-out chip, although at a fraction of the cost. During single-stepping, some hardware functions will stop functioning; for example, the analog-to-digital converter available to many PIC MCU I/O pins will not operate if code execution is single-stepped. You will have to set a breakpoint after these functions and then examine the data values when execution has stopped. Finally, this feature is not available on all PIC microcontrollers, although it is available on many of the most popular chips.

INTEGRATED DEVELOPMENT ENVIRONMENTS

When I'm developing software, I always find that I'm the happiest (most productive, debug fastest) using an integrated development environment (IDE). This extends to PC programming, where I always liked the original Borland Turbo Pascal and the modern Microsoft Visual Basic and Visual C++ development tools (part of Microsoft's Visual Development Studio integrated development environment). I don't believe that I am alone in feeling this way, and the Microchip PIC microcontroller MPLAB integrated development environment (shown in Fig. 3.9) has done a lot to make the PIC microcontroller popular with people looking for a place to learn about microcontrollers.

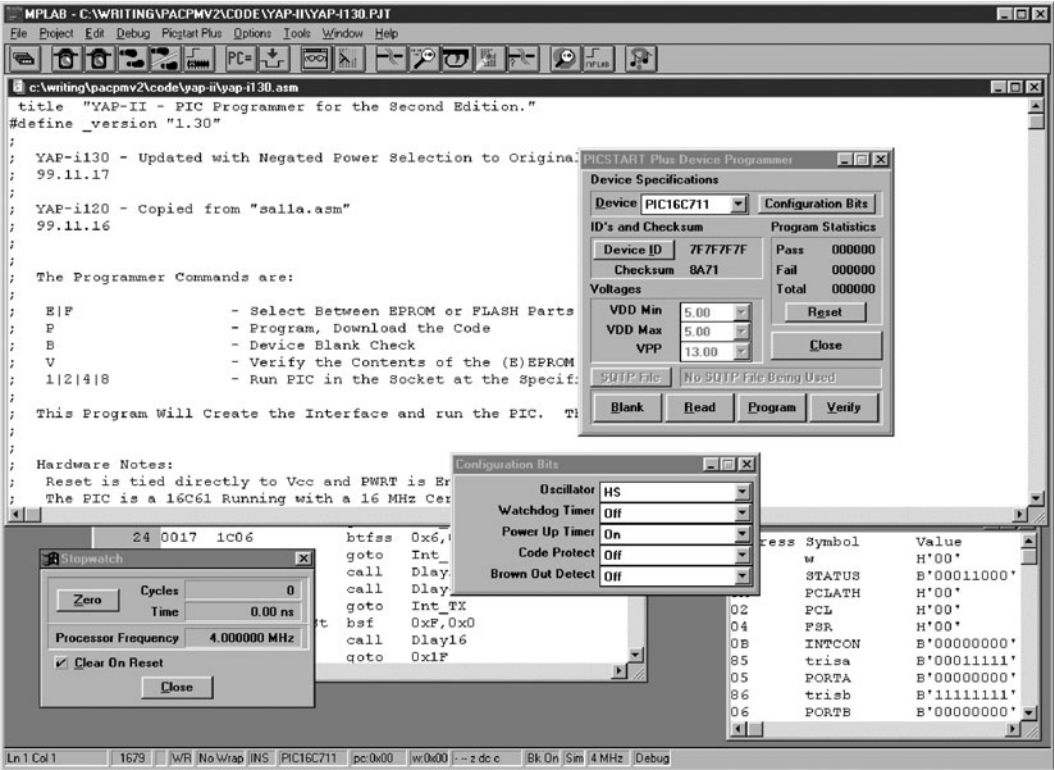


Figure 3.9 The Microchip MPLAB integrated development environment (IDE) is the first tool you should turn to when learning to create PIC microcontroller applications.

An integrated development environment integrates all the software development tools I’ve described in the previous sections. A microcontroller integrated development environment brings the following tools together:

- Editor
- Assembler
- Compiler
- Linker
- Simulator
- Emulator
- Programmer

The purpose of an integrated development environment is to provide the data in a system that allows it to be shared seamlessly as far as the user is concerned and does not require any special input. When I first started working with the PIC microcontroller with MS-DOS command-line tools, to change some source code, assemble, and program a part, I had to go through the following steps:

- 1** Invoke a source code editor.
- 2** Change/enter application code.
- 3** Save and exit editor.
- 4** Invoke the assembler.
- 5** Select the source file.
- 6** Invoke the programmer.
- 7** Select the correct configuration bits.
- 8** Program the PIC microcontroller.

In MPLAB IDE, the steps for changing source code and programming a part are reduced to:

- 1** Start up MPLAB IDE.
- 2** Change/enter application code.
- 3** Click on Project | Build.
- 4** Click on Enable MPLAB ICD 2.
- 5** Click on Start.

Not only is this process faster and less prone to errors, problems with the source code can be fixed immediately in the editor and the application rebuilt without having to go back to the editor and assembler.

MPLAB IDE's projects are used to record the user's preferences for an application, the source code files currently edited, the PIC microcontroller part number along with its execution parameters (such as the clock speed used with it), and the user's preferences in terms of window placement on the MPLAB IDE desktop. In later sections of this book, I will describe the features of MPLAB IDE and how to work with it when developing your own applications.

When learning a new programming language or microcontroller, you should only look at devices that have a well supported integrated development environment available. This will make the process of learning the new device easier for you as well as minimize the possible mistakes you will experience transferring data between the editor, assembler, simulator, and programmer. In this book, I will work with MPLAB IDE exclusively because of its ease in integrating the PIC microcontroller application development steps with application code development, simulation, and programming.

High Level Languages

The PIC microcontroller has been around long enough for there to be a number of languages (and versions of each language) to choose from. These languages are surprisingly efficient—I say “surprisingly” because the low-end and mid-range PIC microcontroller architectures are not well suited for implementing compilers because the limited program stack, the inability to push and pop data, and the limited register space all prevent traditional compiler code designs to be used to create compilers for

them. The code that is generated is usually not as efficient as that somebody who is very familiar with the processor cores writing assembly language can create; the code can be counted on being at least 25 percent less efficient in terms of execution speed and memory usage. If this were a PC or workstation processor, I would say that this difference is enough to make the languages unattractive. But for a microcontroller this is a reasonable difference and will allow you to choose the language and compiler that will best meet your requirements. It will also allow you to learn how to work with the PIC microcontroller peripheral hardware without the tedium of learning the internal operation of the PIC MCU processor.

There are a few features of language implementation that you should be aware of when choosing a compiler. The first has to do with memory. Few microcontrollers (and the PIC microcontrollers in particular) are blessed with the essentially unlimited memory available to the personal computer. The language and implementation you use should be very frugal with their use of the memory resources. One of the things I have found in developing PIC microcontroller applications is that a well designed application does not require a lot of memory. Although in several of the applications I present in this book I do use up most of the available resources, in none of them am I totally hamstrung with meeting the application's requirements. In many cases, I am trying to use the PIC microcontroller with the smallest amount of memory required for the application to minimize the total cost. Languages can use a lot of memory, especially if they aren't optimized. Before investing in a compiler, make sure you understand what type of code is produced and the maximum number of lines you can expect to be able to write for your application. Efficiency is measured in terms of execution speed and program memory requirements for an application—a good PIC microcontroller application uses very little memory and can respond faster than is required by the application requirements.

The next important aspect of the language is the data types used. Native PIC microcontrollers only run in 8-bit code; you should make sure the compiler you use gives you a variety of data types (16-bit variables at a minimum can be extremely useful). I've found there are many times when I would like to use more than 8 bits for counters and such. Many of the programs presented here use 16-bit variables, and in the appendices I've included a number of 16-bit mathematical algorithms for your use when programming assembly language applications.

Hardware support and initializations are important aspects to look at when evaluating PIC microcontroller compilers. It's important to know what the compiler's initial code does before starting the application code (that is, does it set certain features, such as timers and I/O ports in specific states that may cause problems later?). If the compiler uses resources that you will want to use in assembly language (such as the FSR register), you may have to change the resources used by the compiler or use a different approach in your assembly language programming.

Applicability across the whole PIC microcontroller line is another consideration when looking at different languages. The language that you choose should produce code for all the devices in the PIC microcontroller lineup. This is a very important concern because you may be putting your application on a PIC16F84 for development and

debugging, but your ultimate application may use a PIC16F54 (which is much cheaper although it has a different PIC MCU architecture). Using a compiler capable of producing code means that new code doesn't have to be written when porting functions (or even whole applications) to various members of the PIC microcontroller family.

Optimization of the produced code is an important aspect of the compiler. The compiler should be able to review the operation of the code and look for ways to implement it as efficiently as possible. The other question you should ask in terms of optimization is: how well does the compiler do at including only the library code that is required for the program? I like to have all my subroutines available in one file. Only the subroutines that are required by a particular application's calls should be included in the final object file. This may be called "optimized linking." Even simple optimization routines can improve the size and speed of the compiled code by orders of magnitude and make a high level language application approximate the performance (speed and code size) of assembly language code written by an expert.

An important feature is a compiler that can be used "natively" with MPLAB IDE, which is to say that code can be written in MPLAB IDE and compiled with debugging capabilities directly inside the IDE to allow you to take full advantage of the tool. Some early compilers written for PIC microcontrollers did not have the capability to plug in to MPLAB IDE and the lack of capability was very noticeable. The compilers presented in this book can interface natively with the MPLAB IDE and, if you have experience with other systems, I believe that you will be able to recognize the advantages in efficient application development and debugging that a high level language compiler working with MPLAB IDE brings to you.

The last (and probably most important) piece of advice that I have about languages is don't pick one that hides the features of the microcontroller. You should have direct access to all the PIC microcontroller registers. The reason for this, as paradoxical as it seems, is simplicity. If the language controls the interface to the hardware, you must learn how to use the language controls. This means on top of understanding the PIC microcontroller, you also have to learn the language and its hardware interface. To make matters worse, chances are the built in interface was written for the general case, not by somebody that has your application in mind.

For example, to print "Hello World" and start a new line:

```
printf( "Hello World\n" );           // In "C"
```

The `printf` routine depends on specific hardware to function properly. If interface functions (like this one) are required for different hardware, it is important that you are able to access the required resources so that the necessary functions can be coded easily.

High level code development for microcontrollers like the PIC MCU offer significant advantages in terms of development effort and speed, and writing the code in a high level language does prevent a lot of the typos and confusion endemic in assembler coding. Care should be taken, however, to ensure that the compiler and language chosen do not limit what type of applications you can create and instead simplify them and enhance the development process.

GLOBAL AND LOCAL VARIABLES

When working with procedural high level languages (languages that have subroutines, such as C), there are two types of variables that can be used in applications: global and local. Global variables can be accessed anywhere in an application, whereas local variables can only be accessed within the routine they are declared in. Deciding which type of variable to use will affect the size, operation, and efficiency of your PIC microcontroller application. Proper use will help you avoid errors in your application code and make your code easier to write and understand. You might assume that local variables can only be used in high level languages such as C, but they can be used in assembler subroutines as well.

In C, global variables are defined outside of any functions, like `row` and `col` in the example below. Local variables are defined as being inside functions (`temp`) and include the parameters to the functions (`i` and `j`).

```
int row, col;           // Global variable declaration

int videoPos(int i int j) { // Passed parameters are local
    variables

    int temp;           // Local variable declaration

    i = i * 80;
    temp = I + j;

    Return temp;

} // End videoPos Function

main() {                // Example Mainline

    for (row = 0; row < 25; row++ )
        for (col = 0; col < 80; col++ )
            VideoMemory[videoPos(row , col)] = ' ';

} // End Example Mainline
```

The variables `row` and `col` are global and can be accessed by any function in the application. The variables `i`, `j`, and `temp`, are local to the `videoPos` function and can only be accessed within the function. Attempts to access `i`, `j`, or `temp` outside of `videoPos` will result in error statements indicating that the variables are not recognized. This error can be confusing to new programmers as the variables can be seen by reading through the code.

Local variable names can be reused in different functions and there will not be any confusion or invalidly passed values by the compiler. For example, in the mainline (`main`), `row` and `col` could be replaced by local variables `i` and `j` and execution would not be any different nor would there be any confusion between the values of `i` and `j` in `main` and the values inside `videoPos`, even if `videoPos` changed the values of `i` and `j` inside its function. The reason the values wouldn't change is because `i` and `j` in `main` are physically different variables in `videoPos`, even though their names are identical. In many high performance processors (for example, your PC's processor), the local variables are offsets to the data stack, and global variables are placed at static (unchanging) locations in memory.

You should avoid having global variables with the same names as local variables. In some compiler implementations the global variables will be used unless they are in a function with local variables with the same name. In others, the global variables will take precedence and the local variables will be ignored. In still others, an error will be generated if there are local variables with the same names as global variables. To be on the safe side, make sure that you never create code with global and local variable names in common.

When local variables are stored in and referenced from the data stack the memory required for them is allocated dynamically by the active subroutines as required. The input parameters of a subroutine (`i` and `j` in `videoPos` above) are explicitly pushed onto the stack to reserve (or allocate) space for them when a subroutine is called. When the routine is finished, the values are popped off the stack to free up the space that was used by them. If you are familiar with the concept of garbage collection (freeing up memory no longer required by subroutines), you will realize that this is quite an efficient way of executing; once the memory is no longer required for local variables, it is returned to the stack automatically at the end of the subroutine without the need for any other operations.

Local variables inside a function are relative to the current stack position, and when they are to be accessed, the offset to the data stack must be calculated. The operation of `videoPos` can be illustrated using pseudoassembler code:

```

Main:
    :

; VideoMemory[videoPos(row , col)] = ' ';
push row                ; Parameters Pushed in Same Order as in
push col                ; Subroutines
call videoPos
pop  _____         ; Restore the stack to the precall value
pop  _____

; videoPos(int i, int j)

; int temp;
push #0                 ; Define "temp"

; i = i * 80
move accumulator, (sp - 3); Get first Parameter ("i") from stack
mul  accumulator, #80

```

```
    move (sp - 3), accumulator    ; Store the Result back onto the
stack                               ;
                                   ; (in "i")

; temp = i + j

    move accumulator, (sp - 3)    ; Load Accumulator with "I"
    add  accumulator, (sp - 2)    ; Add "j" to Accumulator
    move (sp - 1), accumulator    ; Save the Result in "temp"

; return temp

    pop  accumulator    ; Restore the Stack to its Original Value and
                        ; load the Accumulator with "temp"

    return              ; End of "videoPos"
```

In this example, the parameter and local variables are pushed onto the stack before the subroutine's code is executed. By doing this, there are unique memory blocks allocated for local variables each time the subroutine is called.

The low-end and mid-range PIC microcontroller architectures do not have a data stack with push, pop, and reference instructions, which makes the operation more complex. One solution to this problem is for a compiler to look at the call paths possible in the application and use the same locations for local variables of subroutines that will never execute as part of the same execution branch. This isn't a bad method and will work well for any execution type except for recursive subroutines. The PIC18 has the ability to implement a data stack that can be accessed with an offset, allowing for the implementation of local variables in the same method as high-end processors. It should be noted that recursive subroutines should not be implemented in the PIC microcontroller architectures because of the limited stack depths, even for the PIC18.

BASIC AND PICBASIC

As designed and intended to be used, BASIC (Beginner's All-purpose Symbolic Instruction Code) is not very well suited for the PIC microcontroller because it is an interpreted language meant to provide a simple development environment for users to enter, execute, and debug code. An interpreted language is one in which the source code is saved directly into the computer and a separate program (called an interpreter) reads through the code and executes the instructions it encounters. The first personal computer BASICs were built using this model—arguably, the most famous one being the interpreter shipped with the Apple II. The BASIC used in this computer was hand assembled; Steve Wozniak, the lead designer of the Apple II and the BASIC developer, never stored the BASIC interpreter source code in a computer. Instead, he kept it in a notebook in which the assembly language instructions were converted by hand into hex codes and manually burned into ROMs, which were installed in the computer. This BASIC was many people's (including myself) introduction to computer programming because it was easy to use and took advantage of

the console built into the Apple II. The BASIC language is not well suited to the PIC microcontroller because the lack of a console normally available to the chip precludes many of the interactive features of BASIC, and the need for storing the program as source code is difficult on the chips because of their small program memories.

The first BASIC that was generally available for the PIC microcontroller was Parallax's PBASIC, which is used for the BASIC Stamp and initially had a rudimentary compiler for the low-end (PIC16C54) PIC MCUs. This language is a very simple BASIC that provides all the basic features, but has some things to watch out for. In App. E, I give a detailed description of microEngineering Lab's PICBASIC (a derivative of Parallax's original PBASIC), which has gone on to become the most popular BASIC available for the PIC microcontroller. Despite not having the sophistication of some of the C programming language offerings available, it is an easy way to start with the PIC microcontroller, especially for developers who started with the Parallax BASIC Stamp.

PICBASIC works quite well, but there are a few points you should be aware of when you are developing applications using this compiler. The first is how variables are handled. In "true" BASICs, variables are implicitly declared. This means they are not declared and no space is reserved for them until they are referenced in the source code. The type of variable is defined by the postfix to the label (no postfix means the variable is a floating value, a # postfix means an integer, and a \$ means a text string). Specific type variables (including arrays) are defined using the DIM statement.

In PBASIC, 8-bit variables are predefined and are given the labels B0 to B15 to maintain compatibility with the BASIC Stamp. These variable types can be concatenated together to form 16-bit integers or broken up into individual bit variables. Arbitrary label names can be assigned to these variables as a define.

Assignment statements are quite usual in PBASIC. Instead of working through an order of operations, assignments are executed right to left. For example, the statement:

$$A = (B * C) + D$$

would be expected to execute in the following order:

- 1** Multiply the contents of B with the contents of C.
- 2** Add D to the product of B and C.
- 3** Store the result in A.

In PBASIC, the addition would execute first, followed by the multiplication. Written out, this would be:

- 1** Add the contents of C to the contents of D.
- 2** Multiply B with the sum of C and D.
- 3** Store the result in A.

This can be a problem for people who have experience with other languages. To avoid this problem when I work with PBASIC, I avoid compound expressions like the one above in the assignment statement above, instead breaking the assignment statement up into its constituent parts and saving any intermediate results in temporary variables.

This also avoids issues with BASIC compilers that do not have the order of execution limitation that PICBASIC does and avoids differences in application execution if you port the application between the two compilers.

The statement above ($A = (B * C) + D$) could be broken up into the multiple assignment statements listed below:

```
Temp = B * C
A = Temp + D
```

When writing out compound statements like this, make sure that the intermediate values are not stored in the final destination. Storing intermediate values in the final destination could cause them to be used by peripheral hardware in the PIC microcontroller, causing invalid operation of the application.

PBASIC can access hardware resources directly, but you may wish to use assembly language drivers (which are discussed later in the book). Built-in driver and interface code for PICBASIC as well as all high level languages will tend to be generic. It will either not do exactly what you want or will be all-encompassing, which means the library will take up more space than can be afforded in the application and provide more features than are required. When you are first starting out, you will probably want to consider using the built-in I/O functions, but as you become more familiar with the PIC microcontroller and its peripheral functions, you will probably prefer to develop your own interfaces that are tailored to the application.

BASIC87X

As a bit of a lark, I tried my hand at creating a simple BASIC interpreter that provided its own simple user interface that could be accessed via RS-232 on an ASCII terminal emulator built into a PC. The resulting program, which I called BASIC87x, was written to run on a PIC16F877(A) with the internal Flash program memory used for storage of both the interpreter (with user interface) as well as the execution code. Later in the book, I will show how a PIC microcontroller can be used as either a stand-alone device or as an application development tool—a great way to introduce new developers to microcontroller programming and interfacing as well as provide a view into what it was like in the early days of personal computers.

This language provided by BASIC87x is an interpreted and standard BASIC with the following characteristics:

- Very rudimentary implementation of BASIC.
- The program memory write function of the PIC16F876/PIC16F877 is used to store the application code.
- Maximum application size is 8,192 7-bit ASCII characters.
 - Eight bit characters (bit 7 set) will be converted to “.”
 - Tabs will be converted to blanks.
 - Any ASCII control characters (ASCII code less than 0x020 or 32 decimal) other than BS, TAB, and CR will be ignored.

- *Almost* all of the internal features of the PIC16F87x are available to the user.
- Maximum of 57 7-bit ASCII characters to a line.
- Rudimentary compression (double characters per instruction location along with additional black compression) of the program statements to save memory as well as simplify statement parsing.
- 96 bytes of variable name and variable storage.
- Reformatting of statements to no spaces between operators, constants, and variables. One space between constants and keywords. The formatting code has a parameter for the number of spaces from the left and has a parameter for the column number for the comment.
- Controlled serially at comfortable speed for processor USART (1,200 bps at 4 MHz).
- The interpreter can be expected to run at 600 statements per second average when simulated on a 4 MHz PIC microcontroller. The actual application speed will vary according to the amount of Flash accesses and serial I/O operations performed.
- All application source text (not comments) will be in uppercase to simplify label comparison operations.
- Arithmetic statements are simple one or two parameter operations.
- Multiple statements can be put on a single line, separated by a single colon (:) character.
 - Statements cannot be extended past the next line.
- If there is an application already present in program memory, it will start after a 10 second countdown in which a serial `Ctrl-C` halts the operation and initiates the user interface.
 - During operation, input keystrokes will be buffered and passed to the application via the `INPUT` statement.
 - `Ctrl-C` can be used at any time to stop application execution, but will only be invoked at the end of the current line.

Unlike early versions of BASIC, BASIC87x is not line number based. Instead, destinations are specified by labels. As well, only simple assignment statements (with a maximum of one arithmetic operation) are allowed. Blank lines (whitespace) are not available; instead a commented line will have to be used to separate code and comments for readability. BASIC87x was written to natively execute 16-bit values, although 8-bit values are available. The language is reasonably complete in terms of data types, both 16- and 8-bit variables can be declared as arrays, and registers can be declared like variables for direct access.

Assignment statements use the traditional programming language format of:

```
Destination = [MonatomicOperator] Parameter1 [Operator Parameter2]
```

where the destination is a variable (or array element) and the parameters can be either a variable (or array element) or a decimal value. Hexadecimal or binary values are not allowed. Array elements are checked to be within declared boundaries and 0 (zero) is the first element of an array.

The arithmetic operators available in the assignment statement are listed in Table 3.6, and Table 3.7 lists all the statement types available to the application developer.

TABLE 3.6 BASIC87X ARITHMETIC OPERATORS	
OPERATOR	FUNCTION
+	Addition
—	Subtraction (possible monatomic)
*	Multiplication
/	Division
//	Modulus division (return remainder)
!	Bitwise complement (monatomic)
&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
<<	Bitwise shift left
>>	Bitwise shift right

The format for the condition of the IF statement is:

```
Parameter1 ConditionalOperator Parameter2
```

where the `ConditionalOperator` is listed in Table 3.8.
Multiple comparisons can be implemented in the IF statement like this:

```
IF Parameter1 > I AND Parameter2 < I THEN Label
```

Return addresses for `FOR` and `GOSUB` are stored on a stack with the maximum number of nested `FOR/GOSUB` statements being 24. Note that working to this maximum will cause problems, as this area is used for temporary variables for different statement operations.

To edit a line, first the line to be changed has to be made current (using the “Jump to Specified Line” command listed in Table 3.9). Next, the line can be displayed (using the `L` or `List` command) before it is deleted (`D` command). Finally, the new source line is entered in and Enter is pressed (a “Carriage Return” character or ASCII 0x00D is sent) to save the line in the PIC MCU’s program storage.

Changing the line changes where the application is executing. Entered statements will be saved in memory, but the variable display command (`?`) can be used to show the contents of a variable and will prompt the user to enter in a new value.

More information about BASIC87x can be found on my web page (www.myke.com).

C FOR THE PIC MICROCONTROLLER

Currently, the most popular high level language application development tool is C. This language has been used in application development for virtually every computer

TABLE 3.7 AVAILABLE BASIC87X STATEMENTS

KEYWORD	OPERATIONAL FORMAT	COMMENTS
Comment	'	Everything to the right (and including) the single quote (') will be ignored.
Label	Label	Used for destination of GOTO/GOSUB/IF statements. Differentiated from a line continuation by the label not being a valid statement or keyword.
Line Continuation	:	Single colon (:) after statement will continue execution on the current line.
Internal Data Define	DATA Constant[, ...]	Define constants or constant strings for READING by the application. Data can be numeric or within a constant string (ASCII string enclosed by double quotes). If data is less than 16 bits, then upper bits returned are zero.
Declare Variables or Registers	DIM Variable[(Size)] [AS INT BYTE IREG (Address)][, ...]	Define a variable or register accessed by the application. A variable (INTeger or BYTE) can be an array of constant size. Total size for variable names and their contents cannot exceed 95 bytes.
Access Data EEPROM	EEPROM(Index)	Any location within the 256-byte EEPROM data memory can be read or written from with any variable statement using the EEPROM keyword. Note that program memory Flash (EEPROM) cannot be accessed.
Stop Execution	END	Stop application execution and display the user prompt. Execution also stops when the last statement in the source code is executed.
Repeat Execution	FOR Variable = Constant Variable TO Constant Constant Variable [STEP Variable] ... NEXT [Variable]	Loop a set number of times, ending when Variable equals End. Each execution of NEXT causes Variable to be incremented STEP number of times or, if the parameter is not present, then Variable is incremented by 1.
Change Execution	GOTO Label	Change execution to the statement following Label.
Call a Subroutine	GOSUB Label	Save the location of the next statement and change execution to the statement following Label.
Conditionally Change Execution	IF Condition THEN Label	If Condition is true, then jump to the statement following Label. Condition is defined below.

(Continued)

TABLE 3.7 AVAILABLE BASIC87X STATEMENTS (CONTINUED)

KEYWORD	OPERATIONAL FORMAT	COMMENTS
Get Byte from RS-232 Host	INPUT Variable	Return next byte received by the PICmicro MCU's UART. If there is no byte received to be passed to the application, 0x0FFFF is returned.
Reserved	INTERRUPT	This word is reserved for future implementations of BASIC87x.
Output Data to RS-232 Host	PRINT Constant[@]Variable [,!; , . . !;]	Output data to RS-232 either as constant string or variable contents. If @ is placed before the variable, the least significant 8 bits of the variable are transmitted as an ASCII character. If a comma (,) is encountered, then an ASCII horizontal tab (0x009) is transmitted. If a semi-colon (;) is encountered, then nothing is sent. An ASCII carriage return/line (0x00D/0x00A) character combination is sent at the end of the line if the PRINT statement does not end in a period or a semi-colon.
READ Internal Data	READ Variable	Read the data defined by the DATA keyword. Variable will equal 0x0FFFF if there are no DATA statements or if the read took place past the end of the DATA. The DATA pointer cannot be reset during execution.
Return from Subroutine	RETURN	Return to the statement following the last GOSUB.

TABLE 3.8 CONDITIONAL OPERATORS AVAILABLE IN BASIC87X

CONDITIONAL OPERATOR	COMPLEMENT CONDITIONAL OPERATOR	FUNCTION
=	<>	Compare and jump if equal
<>	=	Compare and jump if <i>not</i> equal
>	<=	Compare and jump if greater than
>=	<	Compare and jump if greater than or Equal
<	>=	Compare and jump if less than
<=	>	Compare and jump if less than or equal

TABLE 3.9 BASIC87X USER INTERFACE COMMANDS

COMMAND	FORMAT	COMMENTS
Add New Statement to Saved Application	Statement <Enter>	Statement placed before current line and becomes the new current line. A statement is assumed to be any line that does not follow the format of any of the commands listed in Table 3-7.
Single Step Current Line	<Enter>	Execute line and stop.
Jump to Specified Line	# <Enter>	Change the current line to the specified one from the start of the application (Line numbers start at one, not zero).
Display the Contents of the Specified Variable	? Label <Enter>	Return the current contents of the variable label that is specified. If an array element is specified, then just display that element, otherwise display all the array elements if the element is not specified. If a single element is specified, then prompt the user to change it. If nothing is entered, the value stays the same.
Display Complete Application	A <Enter>	
Toggle Breakpoint at Current Line	B <Enter>	
Return the Current Line Number	C <Enter>	
Delete Current Line	D <Enter>	
Erase Application	E <Enter>	
Set Current Line to Application End	F <Enter>	Used to point to area where lines can be added.
Execute Application	G <Enter>	Executes starting at the current line. Stops at END statement, the end of the application, or at a breakpoint.
Skip Over Subroutine Call	J <Enter>	If no subroutine call, then this command acts just like a "Skip to Next Line." If there is a call (GOSUB) statement, then the interpreter executes until a return statement is finished executing.
Display Current Line	L <Enter>	
Ping BASIC87x Interface	P <Enter>	BASIC87x displays a new prompt.
Reset the Application	R <Enter>	Reset the application, the interpreter finds all the DIM statements and Processes the variables and finds the first statement to execute.
Display and List the Labels and Contents of All the Variables	W <Enter>	Line by line display of each variable (to allow array elements to be displayed).

available on the market for the past 20 years or so. The reasons for its popularity are the logical design of the language's statements, advanced features such as data structures, and efficient processor assembly language code that can be generated. C's popularity, like many things, has fed on itself. As the language has become more popular, more courses have been taught in it, resulting in more C programmers. The Internet has allowed people to share source code (which can be used in a variety of systems), which further increases C's popularity and usefulness. This popularity makes C an important language to be comfortable with for anyone interested in developing systems applications.

The low-end and mid-range PIC microcontroller architectures are not well suited for procedural-based languages like C, although the higher-end PIC microcontrollers are better suited and can handle the high level language statements reasonably efficiently in most cases. In this section, I want to discuss the various PIC microcontroller devices and the features that make them poorly or well suited for high level languages like C, as well as the features you should look for when choosing a PIC microcontroller C compiler.

The largest point of concern for implementing a high level language like C is the need for a data stack that can save intermediate values in the high level language statements. This is shown in the traditional methods used to program complex statements like:

```
A = B + (C * D);
```

where the statement parameters are pushed onto a stack and processed from there. For the statement above, the stack operations would be:

```
push B
push C
push D
mul
add
pop A
```

where the `mul` and `add` statements pop the two top elements off the stack, perform the operations, and then push the result back onto the stack.

The problem with the low-end and mid-range PIC microcontrollers is the lack of a data stack. This stack can be simulated using the macros:

```
Push Macro
    incf    FSR, f
    movwf  INDF
endm

and

Pop Macro
    movf   INDF, w
    decf   FSR, f
endm
```


The first macro will push the contents of the accumulator (w) onto the stack after incrementing the data stack pointer (FSR). I increment first because if there is an interrupt, which also uses the stack, and the data was written to before the increment, there is the opportunity that the value put onto the stack is overwritten during the interrupt handler.

The PIC18 processor architecture does not have this concern, as indirect addressing operations have the capability of incrementing and decrementing the FSR register during data transfers. Ideally, in a C language, the preincrement and postdecrement should be used to provide stack functions that are compatible with the statements above. Even though the PIC18 has the FSR preincrement and postdecrement operations, they do not have an offset add/subtract capability, which would make accessing stack data within subroutines easier. For example, the call and subroutine code:

```
Subr(int YAxis, int ZAxis)
{
    :

    Address = 4 + (YAxis * Width) + (ZAxis * Length * Width);

    :
} // End Subr

:

A = Subr(j, k);
```

requires accessing data on the stack at arbitrary locations. Before Subr is called, both the j and k variables are pushed onto the stack and accessed by the statements within Subr. For the Address assignment statement, these stack values have to be pulled off the stack along with pushing temporary values onto the stack. Assuming the preincrement and postdecrement push and pop operations described above, this statement would become:

```
; Address = 4 + (YAxis * Width) + (ZAxis * Length * Width);
push Stack - 1                ; Push "ZAxis" Stack Parameter
push Length
mul
push Width
mul
push Stack - 3                ; Push "YAxis" Stack Parameter
push Width
mul
add                            ; Add the Two Products Together
push 4
add
pop Address
```

The problem is the `Stack - 1` and `Stack - 3` operations. In many other processors, an offset can be added to the index for carrying out an access. This is possible in the PIC18 with its ability to add the offset to the FSR register and the `movff` (move from file register to file register) instruction.

In the low-end and mid-range PIC microcontroller architectures, the `push Stack - n` (which I call an offset push) operation would be executed as:

```
bcf    INTCON, GIE        ; Disable Interrupts in the Mid-Range
                                ; PIC microcontrollers
movlw  n                    ; Decrement the Stack to the Correct
subwf  FSR, f              ; Position
movf   INDF, w
movwf  PushPopTemp
movlw  n + 1                ; Increment to the Current Position + 1
addwf  FSR, f              ; for Push
bsf    INTCON, GIE        ; Enable Interrupts in the Mid-Range
                                ; PIC microcontrollers
movf   PushPopTemp, w      ; Save the Data on the Stack
movwf  INDF
```

This is obviously quite a complex operation and it uses up a lot of execution cycles and space in the PIC microcontroller's program memory.

In the PIC18, the data stack pushes and pops should use preincrements and post-decrements to make previous stack values be above the current position. By doing this, the code required to execute the `Address` assignment is:

```
movlw  1                    ; Push "ZAxis" onto the Stack
movff  PLUSW0, POSTDEC0     ; Push the Previous Stack Element Again
                                ; onto Stack
movff  Length, POSTDEC0    ; Push "Length" onto the Stack
call   mul                 ; Multiply the Two Top Stack Elements
                                ; Together
movff  Width, POSTDEC0 ; Push "Width" onto the Stack
call   mul
movlw  3                    ; Push "YAxis" onto the Stack
movff  PLUSW0, POSTDEC0
movff  Width
call   mul
call   add                 ; Add the Two Parameters together and
                                ; put the Result back onto the Stack
movlw  4
movwf  POSTDEC0            ; Push 4 Onto the Stack
call   add
movff  PREINC0, Address
```

Compared to the ideal case shown above of 12 instructions, the PIC18 can execute the same statement in 15 instructions, for an increase of 25 percent—which is actually

very good for any processor. For the mid-range PIC microcontroller architecture, using the push, pop, and offset push code described above would require 36 instructions, or 200 percent more.

The other issue with high level languages like C that the low-end and mid-range PIC microcontrollers are not well suited for is pointers. As I've indicated elsewhere in this book, pointers are difficult to implement and, for most people, this is not a bad thing.

The PIC18 has multiple FSR registers, which means that the stack pointer FSR can be left alone while another FSR register is used to provide the pointer function. This microcontroller processor architecture, with its large (up to 4,096 bytes) contiguous data memory space, allows for pointer use quite easily and efficiently.

Structures and unions in C require an additional FSR as well for accessing specific elements of the data types. Again, I would recommend avoiding the low-end and mid-range PIC microcontrollers for C applications that have these programming constructs.

In typical C development environments, a large amount of heap space is made available. This is typically used for the statement data stack (which I have described above), but it is also used for providing space for automatic variables, including structures and unions. In all the PIC microcontroller architectures, this space is very limited and space should not be allocated, to avoid problems with the data stack running out of space.

If you have worked with C and are familiar with how it works, you should be comfortable with the concept that it is almost impossible to use without pointers. Simple statements like `printf`, as in the following example:

```
printf("Failure, Expect 0x0%04X Actual 0x0%40X", Expected, Actual);
```

use pointers in them even though they are not explicitly noted. In the `printf` example above, everything within the double quotes (") is stored in memory and the pointer to this string is passed to the `printf` library routine. In a typical C implementation for a Harvard processor, the string in quotes is copied from program memory and stored into register memory and a pointer to it returned. Implementing this type of statement in a PIC microcontroller causes a double whammy to the application developer. The string takes up quite a bit of space in both the program memory and register memory, neither of which has much to spare.

In PIC microcontroller compilers, this code should never use a pointer, although it may be reasonable to do it in the PIC18Cxx because of its large, "flat" register space. Instead, the string in quotes should be kept in program memory, and a pointer to the string passed to the `printf` function should indicate that the string is not in register memory.

When selecting a C compiler for the PIC microcontroller, make sure that it provides a basic interrupt handler procedure header. In this book, you will see me use the `interrupt` data type, which indicates that the interrupt handler code is within the procedure.

```
interrupt InterruptHandler()
{
    :                               // Put in Interrupt Handler Code Here
} // End InterruptHandler
```

The start of the interrupt handler procedure should save all the possible context registers by pushing them onto the stack (and popping them off). Pushing the context registers onto the stack will allow nested interrupt handlers or, in the case of the PIC18, multiple handlers to execute.

Note that when implementing an interrupt handler in the mid-range chips, the `w` and `STATUS` register contents will have to be stored in a temporary register first to avoid problems with the stack. A sample mid-range PIC microcontroller C interrupt handler starting code that pushes the context registers onto the stack could be:

InterruptHandler

```
movwf  _w           ; Save "w" and STATUS before placing on stack
movf   STATUS, w
bcf    STATUS, RP0  ; Make Sure Execution in Bank 0
movwf  _status
movf   _w           ; Now, Save "w" onto the stack
incf   FSR, f
movwf  INDF
movf   _status      ; Save the Status Register onto the stack
incf   FSR, f
movwf  INDF
movf   PCLATH, w    ; Save the PCLATH Register
incf   FSR, f
movwf  INDF
clrf   PCLATH       ; Reset PCLATH to Page 0 for the Interrupt
                        ; Handler
```

In the sample code above, notice that I reset the `RP0` flag (because in the mid-range devices, I recommend that all arrays and other data structures that are accessed by the `FSR` register are placed in bank 1) and that I reset `PCLATH`. This code is the truly general case, and while it takes a few more cycles than what could be considered best, it should always be used to ensure proper operation in all cases.

After the header above is executed, interrupts can be enabled again, allowing for nested interrupts because the saved `w` and `STATUS` register values are saved on the data stack. The interrupt entry and exit code can be used exactly as shown in assembly language applications to provide nested interrupt handlers. This is the only case where I feel it is acceptable to allow nested interrupts in the mid-range PIC microcontroller.

To return from an interrupt in the mid-range PIC microcontroller, the following code should be used:

```
movf   INDF, w      ; Restore PCLATH
decf   FSR, f
movwf  PCLATH
movf   INDF, w
decf   FSR, f
bcf    INTCON, GIE  ; Disable Interrupts until operation is
                        ; complete
```

```

movwf  _status           ; Save the Status Register Values
movf   INDF, w
movwf  _w                 ; Save the "w" Register
movf   _status, w        ; Restore STATUS
movwf  STATUS
swapf  _w, f             ; Restore "w"
swapf  _w, w
retfie

```

A true C (ANSI standard) has a number of library routines associated with it (the standard ones are listed in App. F). These routines provide the capability of different types of data (such as floating point numbers), finding trigonometric and logarithmic values and providing standard I/O to consoles (`getf` and `printf`). There are many standard routines and often a lot of custom ones for the environment (processor and operating system) and hardware. Just as an example, for the C that I use for the IBM PC, the library routines take up 286 K of space on the hard drive. I would expect that a full-featured PIC microcontroller C compiler would have a similarly sized library.

This makes the presence of an application linker that only includes needed functions from the application code and library very important. This capability is provided with MPLINK, but you should check for it in other tool sets before buying.

When using functions, understanding how they work is even more critical in the PIC microcontroller than in a PC or workstation. As I discussed earlier, the PIC microcontroller has limited heap space, but different PIC microcontrollers have different hardware features. Standard I/O functions (`getf` and `printf`) may be designed to work with PIC microcontrollers that have built-in USARTs, but will not work in lower-end PIC microcontrollers that do not have this capability.

VERSION SUPPORT TOOLS

Version support is a ten-dollar phrase for keeping track of your source code and making sure that the appropriate versions of the files are used in development builds despite having multiple people working on the project. Depending on the number of people working together to develop the application, the number of modules involved in the development of the application, and the number of files that go into the build of the application, you have various options for controlling the source code to ensure that the final executable file was created from the correct files. For large projects with many people involved in the development of the application, there are sophisticated tools for ensuring that only the correct files are included in the build, while allowing developers to create new code in updated source code files that are not included in the overall application build. If you are working by yourself and have an application that consists of many files linked together, you can probably control the module versions by keeping the latest code for the build in a separate file folder. Finally, when you are starting to develop applications, you can control the level by modifying the file name (including a version number or the date) and making sure that you create MPLAB IDE projects that just build these files. In this section, I will discuss some of the techniques for controlling the software

level for applications that do not have a large number of source files or more than just one or two developers working on them. These techniques will work for large applications, but you (and the other people working on them) will find them cumbersome and error-prone to implement.

It should be obvious, but very few new developers do this: every application should be given its own folder on the PC that the application is built from. Personally, I like to have a set of folders inside the primary application folder, with each one holding the source and MPLAB IDE files for a specific version of the code. Having the MPLAB IDE files in the same folder as the source code (and hex files) is important because they specify which files are used in the project. By creating a unique project for each version of the application, when you build the source into the application, you can be assured that only the files in the subdirectory relating to the version are used rather than having to manually change the source files when you are building a new version of the application.

The folders that are mentioned in this section are most likely located on a central server with the build files directed to the locations of these files. The files should be backed up onto CD-ROM or DVD regularly (every week or each time there is a major update to the code base) to ensure that hard drive crashes or inadvertent changes do not erase many hours of work. It may be useful to write-protect the source code folders to users once the code is completed for the current level. There will be times when a new build does not work, and one of the techniques used to determine why the code doesn't work is to compare pieces of source code and the object files produced to previous versions of the application. Storing the source code files and write-protecting them on a central server once the code has been released will provide the team with a single location for storing files for application release builds, ensuring that the correct code is always used in application development.

With the folder, project, and name of the file unique, I note in the source code comments where this version is to be used and whether or not it is to be released. This is to ensure that the source code matches the desired version. If there is a display or RS-232 connection, I always output the version number of the application. This can be done simply by putting a `define` at the start of the application with a string defining what the version is. This usually looks like:

```
#define _version "1.00"
```

In the code itself, I can then insert this string anywhere the version information is required. If I want to have the version number displayed from a table read in an assembler file, I can insert the `define` label simply into a text string statement. The `define` string can not only be used in any output messages but it can often be inserted into the listing page headers to allow you to see the source code level at a glance.

As rudimentary and obvious as these procedures seem, by taking some time to establish how you are going to keep track of which version of your code you are going to be working on, how you are going to save previous versions of the application, and where you can find the code, you are meeting the requirements of a sophisticated version control tool. By following these procedures, you will find that simple mistakes that you

probably make every day with respect to selecting which source code files are to be used in a new version of your application—or trying to go back to a previous version of the code, only to find that it was overwritten—will become a thing of the past. MPLAB helps facilitate version control through its use of projects. Later in this chapter I will spend some time explaining how to set up projects in MPLAB and how they can simplify your application development as well as ensure that you are only working with the code specific to the current version of the application.

Microchip MPLAB IDE

Although there are a number of companies that have created IDEs and development tools for the PIC microcontroller, I believe that Microchip's MPLAB IDE is the best, both for beginners as well as experienced developers. MPLAB IDE offers the basic development functions, including an editor, assembler/linker/builder interface, and a simulator, but it also includes more advanced features such as programmer, emulator and debugger interfaces that eliminate the need for learning new tools. MPLAB IDE also provides the facilities for cross-referencing machine code and hard addresses back to the source files to allow the editor to indicate the current line or update the data watch windows. The software is professionally written and you will probably not experience any problems with it, which will make the process of learning the tool and how to develop applications for the PIC microcontroller more efficient. MPLAB is fairly unique because Microchip provides this tool free of charge to download off the web. Not having this type of development tool would make learning the PIC microcontroller much more difficult.

MPLAB IDE is a very complete integrated development environment for all the PIC microcontroller families and runs under Microsoft's Windows operating system. The tool is regularly updated, so you should check Microchip's web site periodically to see if new versions of the tool are available for download and if there are new chip support, features, or bug fixes that you will require. Along with this tool, Microchip provides a set of forums where you can post questions or help others learn about PIC MCU applications or MPLAB IDE operation. Microchip, like many other vendors, has embraced the Internet and provides software and datasheets on their web site, which you can access to help you develop your PIC microcontroller applications.

INSTALLING MPLAB IDE

In the previous editions of this book, I included a diskette or CD-ROM with copies of the Microchip development tools (the first edition included MS-DOS command-line tools while the second edition included a copy of the Windows version). The problem with doing this is how quickly the copies became out of date; Microchip continually updates MPLAB IDE with new features and new chip support, which is why I recommend that you download the code from the web and install it directly on your PC. In this section, I will guide you through the process of installing MPLAB IDE, followed by showing

you the process of creating a project and installing an introductory C compiler that will give you quite a bit of flexibility in creating your own applications.

Before starting the installation process, there are a few things for you to do to ensure a trouble-free installation. First, you must be prepared to spend an hour for the installation—the process is definitely not “fire and forget,” and you will have to be present throughout the download and installation. The installation requires a lot of your PC’s resources, so all other applications, except for folder windows and an Internet browser, should be closed before you start downloading and definitely must be closed when you start the installation process. Finally, some antivirus tools may attempt to block the download; if necessary you should temporarily enable this capability if it is disabled. Downloading and installing MPLAB IDE is a significant operation and one that you should be prepared for to ensure that it goes smoothly.

The first step is to go to the Microchip web site (Fig. 3.10) and click on MPLAB® IDE in the Design box. This will bring you to a new page devoted to the MPLAB IDE from which you can review the features of the software and download the application

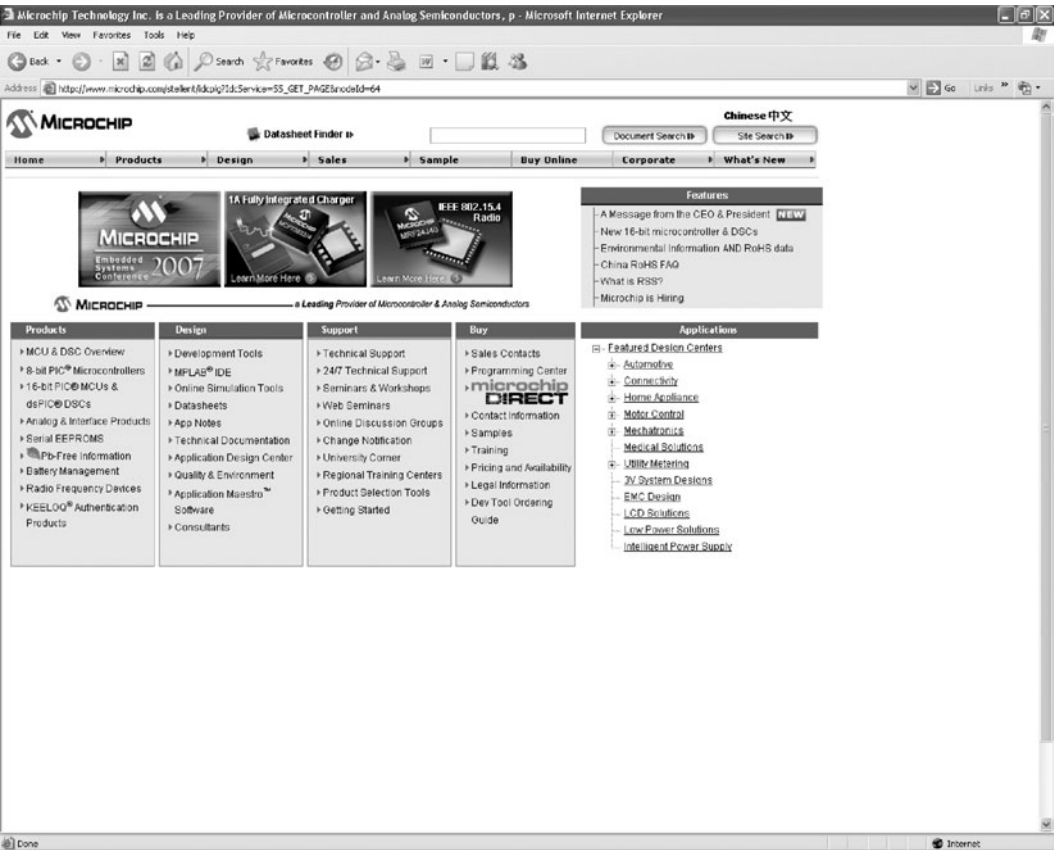


Figure 3.10 Microchip’s web site can be found at www.microchip.com.

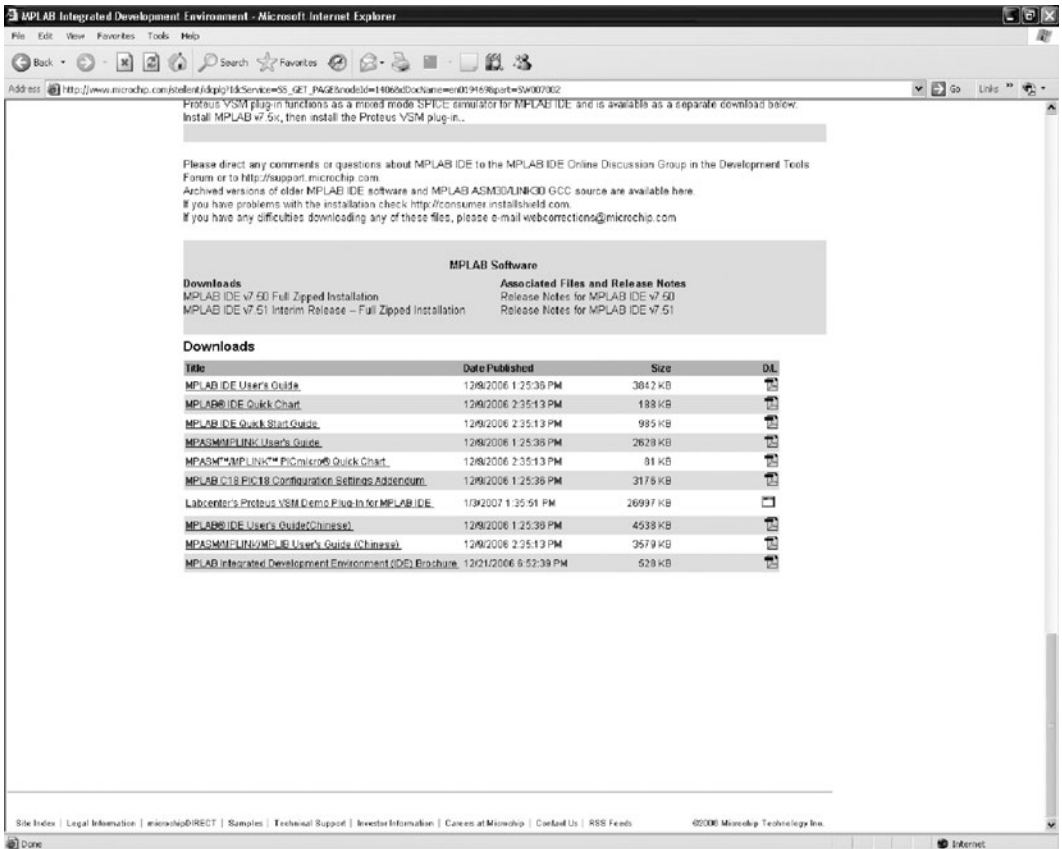


Figure 3.11 Click on Download Full Zipped Installation to get the most recent fully tested release of MPLAB IDE.

or datasheets. Scroll down to the bottom of the page (shown in Fig. 3.11) and click on the appropriate Download option. I usually select the Full Zipped Installation rather than the Interim Release to ensure that the software and its features are fully tested.

When you are given the option of where to put the .zip file, select a temporary folder such as C:\temp. Once the program has downloaded (depending on your Internet connection, this can range from five minutes to over an hour; at the time this was written, the full MPLAB IDE package is greater than 30 megabytes), run an unzip program, and store (extract) the contents into the C:\temp folder. Figure 3.12 shows the .zip file with the extracted contents; the application installation files ready to load MPLAB IDE onto your PC.

With the full .zip program downloaded and the MPLAB IDE setup files extracted from it, double-click on MPLAB IDE v#.#.# Install (where "#.#.#" is the version number of



Figure 3.12 Extract the MPLAB IDE installation files from the downloaded .zip file and store in a temporary folder.

MPLAB IDE) and follow the instructions to install the program onto your PC (Fig. 3.13). Before the installation completes, you should have the opportunity to display the readme files for the different functions. I recommend that you do so, to help familiarize yourself with the program and its capabilities.

When the installation is complete, you can delete the contents of the C:\temp folder. MPLAB IDE is constantly updated and improved. As I noted at the start of this section, I didn't feel that it would be helpful to provide you with a copy of the tool on a CD-ROM and I do not believe there is a reason for you to save a copy either—if you have a disk crash or want to migrate to a new PC, you should download a new copy of MPLAB IDE and install it (using the procedures outlined in this section). Similarly, every couple of months or so, you may want to check to see if there is a new version of the program available for download. If there is, you will want to install it to take advantage of the latest changes. You do not have to remove the current copy of MPLAB IDE, the installation program will do that for you. There really is no reason to save the installation file that you have downloaded and unzipped.

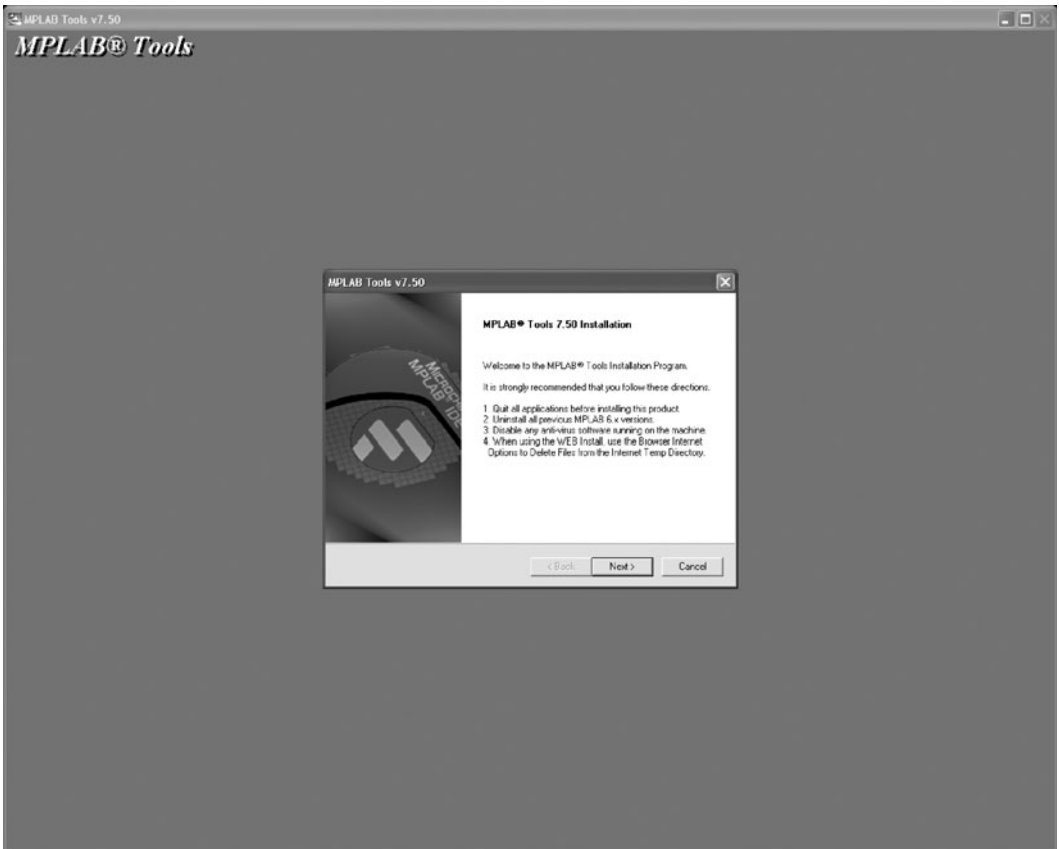


Figure 3.13 After unzipping the MPLAB File, click on MPLAB IDE v#.#.# Install and follow the instructions to install MPLAB IDE for a complete installation.

Now you should have a desktop icon which you can click to start up MPLAB IDE. The first time you click on the icon, don't be surprised if it takes up to five minutes for the program to configure itself and do final setups before you can use it. Later program invocations will be much faster. Once the MPLAB IDE desktop is up and running (Fig. 3.14) you are able to create your first application.

MPLAB IDE FILE TYPES

While MPLAB IDE does a lot to shield you from the various files that are required and produced when you are creating an application, you should be aware of the different files involved in the creation of an application. When you are creating an application, along with the source code you should be aware of the need for include files as well as linker control files. When assembly or compilation is complete, the process will produce a number of files which, along with the .hex file (which is programmed into a PIC microcontroller) are resources for the linker and the debugger or emulator tools. You

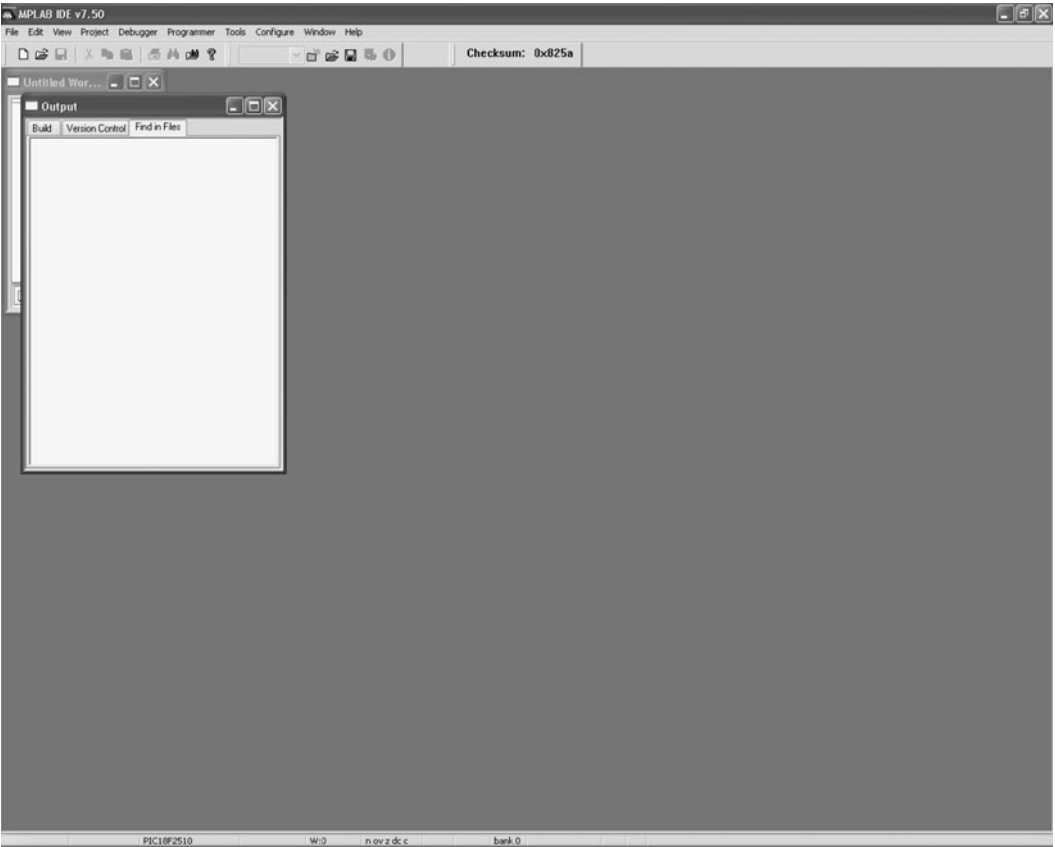


Figure 3.14 Finished installing MPLAB IDE—the desktop is up and running.

do not need to concern yourself with the various MPLAB IDE project management files (such as the files that end in .mcp, .mcs, .mcw, etc.) because these files will change with different versions of MPLAB IDE as well as the project and the tools used to build the application. I realize that when you are starting out with a new device, the number of things to learn is overwhelming; the purpose of reviewing the file types is to help you understand the process and choose the path that will allow you to most efficiently learn how to program the PIC microcontroller.

In Fig. 3.15, I have drawn the build process as a large box encompassing the assembler/compiler and the linker process step as well as showing the files required and produced for the two process steps. For your first application, you should be aware of the process and the purpose of the ten file types shown here. As you become more familiar with the PIC microcontroller, MPLAB IDE and the development tools, you will be able to customize these files to make your application development more sophisticated.

The need for the source files should be self-explanatory—the .asm, .bas, and .c files that are created as the program is run through the assembler/compiler are entered into

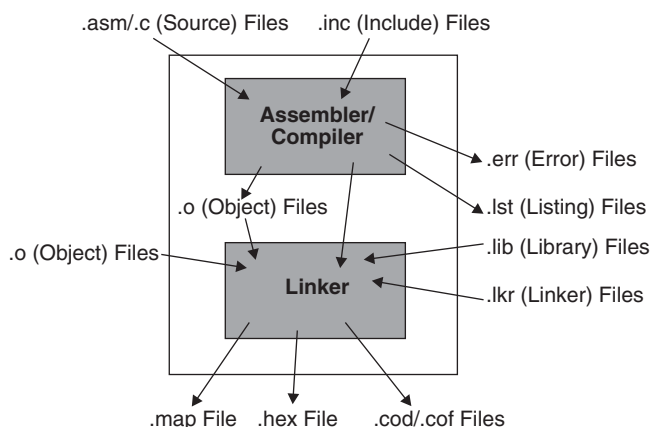


Figure 3.15 The application build process requires and produces the various files shown here.

the MPLAB IDE editor and then passed through the build process. Later in the book, I will comment on how these files should be written and give basic templates for your use, but for now, I want to identify them as files that are the basis of the application.

Often, one of the first things I see when new PIC microcontroller programmers have problems with their initial applications is that they have defined the hardware registers they are going to use in their applications. The programmer is mystified why the code doesn't work or, in some cases, assemble correctly. Almost invariably, the problems with the application are a result of a typo or transposition of a register address or label. To fix the problems, I will tell them to delete the hardware register declarations in their source code and use the `include` directive to load the Microchip or compiler vendor written `.inc` register definition files into their applications. These files were written to provide the application developer with the addresses of the PIC microcontroller hardware registers, along with some other parameters, in the same format as the documentation. Usually, when the programmer-defined hardware register declarations are deleted and the `.inc` file added to the source, the application problems disappear.

When working with the MPASM assembler, there is an `.inc` file for every PIC microcontroller part number in the format:

```
p<I>PIC_MCU_P/N</I>.inc
```

where `<I>PIC_MCU_P/N</I>` is the PIC microcontroller part number. For example, the include file for the PIC16F84A is `p16f84a.inc` and the include file for the PIC12C508 is `p12c508.inc`. This is true for all the PIC microcontroller devices except for the original, low-end (PIC16C5x) parts. For these devices, the include file is `p16c5x.inc`.

The file below is `p12c508.inc`, which is relatively small, but has all the elements that you should look for in the include files.

```
LIST
; P12C508.INC    Standard Header File, Version 1.02      Microchip
Technology, Inc.
NOLIST

; This header file defines configurations, registers, and other useful bits
of
; information for the PIC12C508 microcontroller. These names are taken
to match
; the data sheets as closely as possible.

; Note that the processor must be selected before this file is
; included. The processor may be selected the following ways:

;      1. Command line switch:
;          C:\ MPASM MYFILE.ASM /P12C508
;      2. LIST directive in the source file
;          LIST    P=12C508
;      3. Processor Type entry in the MPASM full-screen interface

;=====
;
;      Revision History
;
;=====

;Rev:   Date:   Reason:

;1.02   05/12/97 Correct STATUS and OPTION register bits
;1.01   08/21/96 Removed VCLMP fuse, corrected oscillators
;1.00   04/10/96 Initial Release

;=====
;
;      Verify Processor
;
;=====

        IFNDEF __12C508
            MESSG "Processor-header file mismatch.  Verify selected
processor."
        ENDIF

;=====
;
;      Register Definitions
;
;=====
```

```

W          EQU    H'0000'
F          EQU    H'0001'

```

```

;--- Register Files -----

```

```

INDF          EQU    H'0000'
TMR0          EQU    H'0001'
PCL           EQU    H'0002'
STATUS        EQU    H'0003'
FSR           EQU    H'0004'
OSCCAL        EQU    H'0005'
GPIO          EQU    H'0006'

```

```

;--- STATUS Bits -----

```

```

GPWUF          EQU    H'0007'
PA0            EQU    H'0005'
NOT_TO         EQU    H'0004'
NOT_PD         EQU    H'0003'
Z              EQU    H'0002'
DC             EQU    H'0001'
C              EQU    H'0000'

```

```

;--- OPTION Bits -----

```

```

NOT_GPWU       EQU    H'0007'
NOT_GPPU       EQU    H'0006'
T0CS           EQU    H'0005'
T0SE           EQU    H'0004'
PSA            EQU    H'0003'
PS2            EQU    H'0002'
PS1            EQU    H'0001'
PS0            EQU    H'0000'

```

```

;=====
;
;    RAM Definition
;
;=====

```

```

    __MAXRAM H'1F'

```

```

;=====
;
;    Configuration Bits
;
;=====

```

<code>_MCLRE_ON</code>	<code>EQU</code>	<code>H'0FFF'</code>
<code>_MCLRE_OFF</code>	<code>EQU</code>	<code>H'0FEF'</code>
<code>_CP_ON</code>	<code>EQU</code>	<code>H'0FF7'</code>
<code>_CP_OFF</code>	<code>EQU</code>	<code>H'0FFF'</code>
<code>_WDT_ON</code>	<code>EQU</code>	<code>H'0FFF'</code>
<code>_WDT_OFF</code>	<code>EQU</code>	<code>H'0FFB'</code>
<code>_LP_OSC</code>	<code>EQU</code>	<code>H'0FFC'</code>
<code>_XT_OSC</code>	<code>EQU</code>	<code>H'0FFD'</code>
<code>_IntRC_OSC</code>	<code>EQU</code>	<code>H'0FFE'</code>
<code>_ExtRC_OSC</code>	<code>EQU</code>	<code>H'0FFF'</code>

LIST

At the start of the file, the PIC microcontroller specified within MPLAB is checked against the file to make sure they match. When MPLAB has a PIC microcontroller selected, the part number label with two underscore characters (__) is defined when the assembler is invoked. For the PIC12C508, this label is `__12C508`, for the PIC16F84, it is `__16F84`, and so on. This label can be used in conditionally assembled code to access hardware appropriately instead of having to define multiple source files for different devices. Once the PIC microcontroller type is verified, then the hardware register addresses (under Register Files) are defined. The registers are given the same labels as the Microchip documentation and have their addresses specified with them. Following the hardware register address definitions, the bit definitions for hardware registers that have unique, accessible bits are defined.

After the hardware register files are defined, then the file registers are defined. The `__MAXRAM` and `__BADRAM` directives are used to indicate the valid addresses for variables. One thing lacking with these directives is that the addresses are not given labels (a label indicating the start of the file registers would be useful) and the registers “shadowed” across banks are not defined. This information could make application development somewhat easier and avoid the necessity of looking up the file register address ranges from the data books.

Lastly, the configuration fuse bits are defined. When I start working with a new PIC microcontroller, one of the first things I always do is to open up the .inc file and look at the configuration fuses. As I will discuss in other areas of the book, a very common mistake is to forget one of the configuration fuse options, which causes your PIC microcontroller application to not work as expected. I want to make sure I access each configuration fuse option, either enabling or disabling it to make sure I don't have any unexpected problems.

The .inc values are defined with the `NOLIST` parameter specified. This means the actual definitions will not be seen in the listing file, but will show up in the symbol table at the end of the listing file.

Compiler include files can provide the same register and address information for an application as the assembly language .inc files and they can also be used to define language functions and features. When you are working with C, to properly access specific library subroutines and functions there are include files that provide function prototypes, constants, and structures required for proper operation of the language.

In early versions of MPLAB IDE, when the tool is invoked and a project is loaded with a source file in the editor, a .\$\$\$ file is created. This file contains the source code before any changes by the MPLAB editor. Typically this file is not required unless the source is corrupted in some way (which often means that you have done something you didn't mean to). This is the same for the .bkx file, which is a backup of the hex file created by MPLAB for the project when it was invoked previously. This file is no longer produced, so it is up to you to ensure that the source file is created correctly.

One of the outputs of the assembler or compiler is the object (.o) file, which is a conversion of the source code statements into assembly code but without the full address information. When the application is linked together, the various object files are attached using the .lkr file, which is produced manually or by the Project Wizard in MPLAB IDE. The object file contains reference information for other object files if label addresses and data objects are local to it or null pointers to addresses and objects if they are external to the object file. The only time the object file will have all the correct references is if the application is written in assembler and there are no references to other object files or libraries.

The listing (.lst) file is another output of the assembler or compiler. Its purpose is to provide error messages where they are encountered in the text, show expanded defines as well as macros, and, if possible, show instruction and list label addresses. Addresses are generally only possible with single assembly source code projects. The following listing file was taken from one of the experiments from the second edition and shows the elements of the listing file. I will go through them to explain what is being displayed. To make the file easier to read, I have truncated the lines to the end of the page and deleted anything that would be wrapped around to the next line. As well, I have taken away the page breaks (except for the one at the start of the application) in order to save space in the book.

```
MPASM 02.30 Released                      ADC.ASM    12-27-1999   14:26:06
PAGE 1
```

LOC	OBJECT	CODE	LINE	SOURCE	TEXT
	VALUE				
			00001	title	"ADC - Reading a Resistor Value with
			00002	;	
			00003	;	This Program Uses the ADC built into a
			00004	;	Reads an ADC Value and displays it on
			00005	;	
			00006	;	Hardware Notes:
			00007	;	PIC16C711 running at 4 MHz
			00008	;	Reset is tied directly to Vcc and PWRT is
			00009	;	A 10K Pot Wired as a Voltage Divider on
			00010	;	A 220 Ohm Resistor and LED is attached to
			00011	;	
			00012	;	Myke Predko
			00013	;	99.12.27
			00014	;	
			00015	LIST	P=16C711, R=DEC

```
00016    INCLUDE "p16c711.inc"
00001        LIST
00002 ; P16C711.INC    Standard Header File, Version
00151        LIST
00017
00018 ;    Registers
00019
2007 3FF1    00020    __CONFIG __CP_OFF&_WDT_OFF&_XT_OSC&_PWRTE_ON
00021
00022    PAGE
00023 ;    Mainline of ADC
00024
0000        00025    org        0
00026
0000 30FF    00027    movlw    0x0FF
0001 0086    00028    movwf    PORTB                ;    Turn off
0002 0185    00029    clrf     PORTA                ;    Use PORTA
00030
0003 1683    00031    bsf     STATUS, RP0                ;    Have to go
0004 0186    00032    clrf     TRISB & 0x07F        ;    Set all
0005 0188    00033    clrf     ADCON1 ^ 0x080        ;    Make RA0
0006 1283    00034    bcf     STATUS, RP0                ;    Go back to
00035
0007 3081    00036    movlw    0x081                ;    Setup
0008 0088    00037    movwf    ADCON0                ;
00038                ;
00039                ;    CHS1:CHS0
00040                ;    Go/_Done
00041                ;    ADIF - 0
00042                ;    ADON - 1
00043
0009        00044    Loop
00045
0009 3003    00046    movlw    3                ;    Wait 12
000A 3EFF    00047    addlw    0x0FF                ;    Take One
000B 1D03    00048    btfss    STATUS, Z
000C 280A    00049    goto     $ - 2
00050
000D 1508    00051    bsf     ADCON0, GO                ;    Turn on
000E 1908    00052    btfsc    ADCON0, GO                ;    Wait for
000F 280E    00053    goto     $ - 1
00054
0010 1683    00055    bsf     STATUS, RP0
0011 0909    00056    comf     ADRES, w                ;    Get the
0012 1283    00057    bcf     STATUS, RP0
0013 0086    00058    movwf    PORTB
00059
0014 2809    00060    goto     Loop                ;    Get
```

```

00061
00062
00063  end

```

SYMBOL TABLE

LABEL	VALUE
ADCON0	00000008
ADCON1	00000088
ADCS0	00000006
ADCS1	00000007
ADIE	00000006
ADIF	00000001
ADON	00000000
ADRES	00000009
C	00000000
CHS0	00000003
CHS1	00000004
DC	00000001
F	00000001
FSR	00000004
GIE	00000007
GO	00000002
GO_DONE	00000002
INDF	00000000
INTCON	0000000B
INTE	00000004
INTEDG	00000006
INTF	00000001
IRP	00000007
Loop	00000009
NOT_BO	00000000
NOT_BOR	00000000
NOT_DONE	00000002
NOT_PD	00000003
NOT_POR	00000001
NOT_RBPU	00000007
NOT_TO	00000004
OPTION_REG	00000081
PCFG0	00000000
PCFG1	00000001
PCL	00000002
PCLATH	0000000A
PCON	00000087
PORTA	00000005
PORTB	00000006
PS0	00000000
PS1	00000001

PS2	00000002
PSA	00000003
RBIE	00000003
RBIF	00000000
RP0	00000005
RP1	00000006
STATUS	00000003
T0CS	00000005
T0IE	00000005
T0IF	00000002
T0SE	00000004
TMR0	00000001
TRISA	00000085
TRISB	00000086
W	00000000
Z	00000002
_BODEN_OFF	00003FBF
_BODEN_ON	00003FFF
_CP_OFF	00003FFF
_CP_ON	0000004F
_HS_OSC	00003FFE
_LP_OSC	00003FFC
_PWRTT_OFF	00003FFF
_PWRTT_ON	00003FF7
_RC_OSC	00003FFF
_WDT_OFF	00003FFB
_WDT_ON	00003FFF
_XT_OSC	00003FFD
__16C711	00000001

MEMORY USAGE MAP ('X' = Used, '-' = Unused)

0000 : XXXXXXXXXXXXXXXX XXXXX=====

2000 : ====X=====

All other memory blocks unused.

Program Memory Words Used: 21

Program Memory Words Free: 1003

Errors : 0

Warnings : 0 reported, 0 suppressed

Messages : 0 reported, 0 suppressed

There are three separate areas in the listing file that you should be aware of. The first is the source code area in which the object code (the hex instruction value) is given to the left of the source file line along with the address where it is located. Each line is repeated with its line number in the source file listed. With this information, instructions

can be found either by their address within the PIC microcontroller's program memory space or by the line they are found on in the source code.

The next section in the listing file is a list of all the labels in the application and what their values are. Note that along with hardware register addresses, bit numbers, labels, and variable file register addresses are all included in this section, listed in alphabetical order. If you are familiar with other assemblers, you may expect that the label types and references to them are also included. In MPLAB just the label and its value at the end of the application are listed.

The last section is a summary of the addresses used by the application along with a total of any errors, warnings, or messages. The program memory address summary can be very useful when you are using a large portion of the address space in the PIC microcontroller and you want to get an idea of what is available.

The last output file you should be concerned with is the .err or error file. This file consists of a list of source files and line numbers of the errors encountered in the assembly/compile step. The information is identical to what is in the .lst file, but it can be easier to see the messages because only the errors are listed in this file. When a source file has been converted successfully into an object file, the .err file length should be zero. This is a bit of a philosophical point because some people will release an application with warnings and messages, which will not prevent the object file from being created (whereas an error will), but I like to make sure that there are no messages of any kind in my applications before going forward. Warnings and messages are usually an indicator of semantic errors in the logic of the program and not syntactic errors, which tend to be typos. This is the reason I do not suppress any warnings or messages and insist that the .err file is zero bytes long before proceeding and programming the code into the PIC microcontroller.

There are a couple of things I don't like about how errors are reported in MPLAB. The first is that the error descriptions can be somewhat terse and vague and may not be fully understood by new PIC microcontroller application developers. If you get an error and don't understand what it means, don't feel bad about it. Instead, jump to the line that is referenced (by double-clicking on the line in the error window displayed in MPLAB) and see if you can figure out what the problem is. The second thing I don't like is how errors with macros are reported. Macro errors are referenced back to the invoking line, not the line in the macro. This can make debugging macros a challenge, especially if they are very complex. The safest way to ensure that there aren't any problems is to only attempt to program a PIC microcontroller if the .err file is zero bytes long. This ensures that there are no misunderstood errors (in the form of messages and warnings) that could cause problems with your application later.

Once all the source files have been assembled or compiled into object files, the linker combines them all together into a single .hex file, which is specified by the linker (.lkr) file. I tend to think of the linker file as a make file for PIC applications. It specifies which files are included in the application and when it comes time to "Build All," it pulls together all the files to be included, and assembles/compiles them, and links their object files.

An important file for the linker is any library (.lib) file that is included in an application. The libraries are collections of functions and subroutines used to provide basic functions for compiled code. An example of a library subroutine is C's `printf`, which formats and passes data to a console. The user simply specifies the include file that has

the `printf` subroutine prototypes, and during linking, the necessary library subroutines are added to the final application. The library file differs from an object file containing all the possibly required subroutines and functions because only the required subroutines and functions are passed into the linker and included as part of the final application. This keeps the size of the final application as small as possible with just the required library subroutines and functions included in the application.

The `.cod` and/or `.cof` files are the label reference tables that allow MPLAB to run the simulator, emulator, or debugger and put pointers in code to the correct line of the source. These files are not human-readable, and though their data is documented, it is quite difficult to work through and understand. `.cod` files differ from `.cof` files by the limited size of the source file path they can work with (64 bytes versus 256), which means you have to keep the paths and file names to the project folders and files as short as possible. The MPASM compiler can only produce `.cod` files, whereas some C compilers can produce and work with `.cof` files, which allow you to have longer path and file names. These two files should never be deleted and should always be included with the source, object, and hex files to ensure that the simulator, debugger, or emulator functions of MPLAB IDE are always available.

The `.map` file provides a list of the application addresses of labels and data structures in the PIC microcontroller and is produced by the linker when multiple object files are linked together. This file is a good reference to understand how the program is put together and whether there are potential problems such as a subroutine going over a code page boundary or an incorrectly defined array overwriting other memory objects.

The hex file, which is explained in more detail in Chap. 4, is the result of the MPLAB build operation and is the code (ones and zeros) to be programmed into the PIC microcontroller. This file is in human-readable format, although the first time you look at it, it will be somewhat confusing. When you understand the organization of PIC microcontrollers better, you will see how the information produced by the assembler, compiler, and linker is stored in the file, ready to be programmed into the chip.

MPLAB IDE DESKTOP

The Microchip MPLAB IDE has continually evolved since its inception in the late 1990s. The tool has become more powerful, in terms of what it can do, as well as becoming easier and more intuitive to work with. When it first became available, it was a good single user tool with limited capabilities for multiple objects being linked together from a limited number of compilers and assemblers. The capabilities have expanded to allow many development applications to be linked in and it easily accommodates linking multiple object files together. The early versions of the program required a certain amount of customization to be useable whereas the current versions are very straightforward to use. I have no doubt that the continued improvements to the tool will make it easier to develop PIC microcontroller applications in the future while retaining many of the features that are described here.

The basic desktop of MPLAB IDE is shown in Fig. 3.16 with all the commands available from the pull-down menu toolbar, specific and commonly used functions available on the IDE toolbar, with the file window and build status window providing you with information about the application while providing source file editor windows.

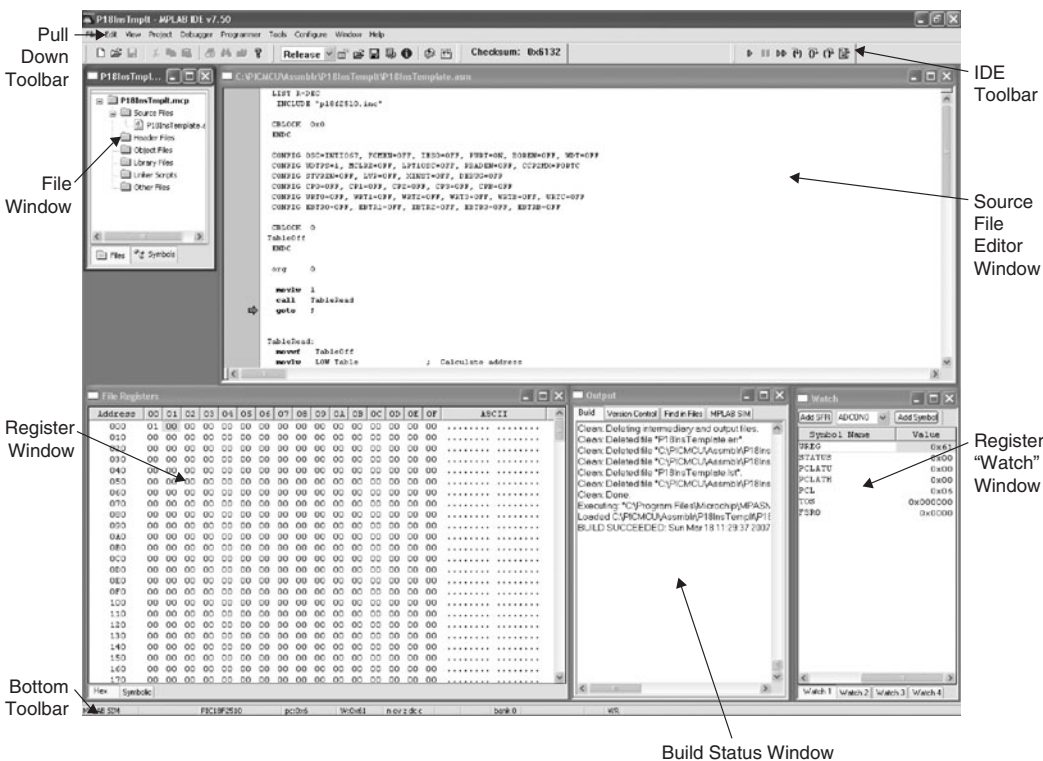


Figure 3.16 All the elements required to create, build, and test an application are available on the MPLAB IDE desktop.

You can monitor the execution of the application in the register window, register watch window, and the bottom toolbar. These functions work together to provide you with the ability to create, build, and test your application all in one program.

The pull-down menu toolbar and IDE toolbar provide you with the basic controls for the operations available to you in MPLAB IDE. Both these toolbars are dynamically configured and the functions available on them will change according to the project (an application including source files and build instructions), the PIC microcontroller it is to be programmed into, as well as the tools available to debug the application and program the chip. In earlier versions of MPLAB IDE, the IDE toolbar could be selected from a series of toolbars with predefined or user selected functions; the more recent IDE toolbars contain the basic functions required to create an application. When you are ready to start testing and debugging an application, you will have the opportunity to select the debugger tool (simulator, emulator ICD 2, etc.) and the programmer, which will add or change icons and pull-down options available to you. The constant updating of MPLAB IDE has created a control interface and paradigm that is quite efficient and easy to learn.

The file window and build status window will list the files specified in an application along with their status as part of the build process. The files that make up a project can be selected using the Project Wizard and automatically added to the project. The file

locations can also be specified manually or have their paths changed manually. This feature allows source files from different PCs and servers to be included in the build, creating an application that is the result of several persons' efforts.

You can have more than one source file editor window active in a project. Even if I am only using a single source file in my application, I will often load up the device .inc and other reference information files to application build so that they can be easily displayed on the MPLAB IDE desktop. The editor is Microsoft compatible, which means it works exactly like text editors you are probably familiar with, such as WordPad, and you can cut, copy, and paste using the PC's clipboard. When multiple editor windows are active, you will have to tile them or order them so you can find the necessary information quickly; unfortunately, there is no tabbing of the windows, which would make the search for specific information faster. The important thing to remember when you have multiple editor windows open is to keep track of which window has which file.

Monitoring the status of the application as it is being simulated or debugged is quite easy with the various windows and the bottom toolbar available to you. The bottom toolbar is the only method discussed here which is not optional. It is always available with the PIC part number selected, the current program counter, the editor operating mode, and the WREG and STATUS register contents. The other windows are discussed in more detail throughout the book and provide you with the ability to monitor the changes in the registers of the chip as well as change their contents.

In Fig. 3.16, I have arranged the various windows the way that I feel most comfortable working with MPLAB IDE. I like to have all the relevant information available to me at all times and I only use overlapping windows for the source file editor—all others have their own location on the desktop that does not interfere with any other windows. The larger and higher pixel count display that you have, the more data you will be able to add to your MPLAB IDE desktop, providing you with all the information and interfaces required to develop your own PIC microcontroller applications.

MPLAB IDE APPLICATION BUILD TOOLS

The build tools (assembler, compilers, and linker) that I discuss in this book are probably the most popular tools available for the PIC microcontrollers and each of them integrate well with MPLAB IDE. The nuts and bolts of this integration were discussed earlier in the chapter with the discussion of the files used or produced in the build process. Each of the tools discussed in the following sections can utilize these files, even though in the case of source code and include files the formats will be different for the different tools. There are other tools available for the PIC microcontroller and many of them provide the same functions as the ones listed here.

Microchip's MPASM Assembler Microchip's "MPASM" is a full-featured macro assembler that can produce object and hex files for any PIC microcontroller processor architecture. The assembler can work with macros and defines to simplify programming along with having the ability to create data structures. Errors and messages are passed directly to MPLAB IDE, and when you click on them in the build status window the cursor will jump directly to the error. The assembler is designed to work on more

than just assembly language source code, it can also process and format table data and configuration fuse values. The assembler can produce object code for linking with other programs or pass its output directly to the linker for creation of a .hex file that can be programmed directly into a PIC microcontroller.

This is the default tool for developing application code, and when I wrote the second edition of this book, I considered assembly language to be the basic method of PIC microcontroller programming. Over time, I have seen the efficiency of high level compilers improve and I would say that the need for understanding and using the assembler has lessened considerably. That said, a good basic understanding of the various PIC microcontroller processor architectures and their configuration fuses and other features is necessary to successfully develop efficient applications.

If you have looked ahead at later chapters in which I have provided application code, you would probably be surprised to find that only two types of statements are required for a PIC microcontroller application. This will be hard to reconcile because the applications in the book seem to be just *full* of various types of statements, each one seeming to provide a different feature to the PIC microcontroller. Actually, all these statement types are meaningless to the assembler: instead it just looks through the application code for instructions and an indication of the end of the code.

The most basic application source I could come up with is called `minimum.asm`, which can be found in the `code\minimum` subdirectory of the `PICDownload` folder. This code clears `PORTB` and then clears the `TRISB` register, which enables all 8 bits for output. Once this has completed, the application goes into an endless loop. The code that does this is simply:

```
clrf    6
bsf     3, 5
clrf    6
bcf     3, 5
goto    4
end
```

Comparing this source file to what I have produced in Chap. 21, you will feel like something is missing. I can say that nothing is missing from the perspective of what the assembler needs to convert the source code to a hex file that can be programmed into the application. The reason why this source code looks so different is that different statements have been added to the MPASM assembler to make applications easier for you to write. In this section, I will go through the various aspects of the source file and explain what the statements are and why you might like to use them.

The two statement types that are required for an application are the PIC microcontroller instructions and the directives. The instructions are the application itself, and the end directive is a command to stop the assembler. The only requirement of these two statements is that they cannot start in the first column of the file. Directives are instructions to an assembler. In the next section, I will list all the directives that are recognized by the MPASM assembler and what they do. In later chapters, I will discuss various types of directives (such as macros) in more detail and how they can be used to simplify application development. In this section, I will just introduce you to the basic directives needed to develop a readable PIC microcontroller application.

Just using these two statements will certainly make your application efficient, but almost impossible for other people (and probably yourself) to read. By adding different types of statements, the readability of the MPASM source is improved considerably and the ease with which you develop applications will be improved as well.

When you look at `minimum.asm`, the first problem you will have with it is that you don't have any idea what the instructions are pointing to. Labels and defines are added to applications that allow you to reference addresses and certain constants with text strings that should make understanding the code somewhat easier. By taking `minimum.asm` and adding the register name labels (from the documentation), you can improve the readability of the application considerably:

```
clrf    PORTB
bsf     STATUS, 5
clrf    TRISB ^ 0x080
bcf     STATUS, 5
goto    4
end
```

The bit labels given in the documentation can also be used to further enhance the readability of the application source code:

```
clrf    PORTB
bsf     STATUS, RP0
clrf    TRISB ^ 80
bcf     STATUS, RP0
goto    4
end
```

The XORing `TRISB` with 80 clears the most significant bit of the address.

When MPASM starts executing, the default numbering system (or radix) is hexadecimal. This means that the 80 that is XORed with the address of `TRISB` is actually 128 decimal.

The register and bit labels are not available automatically to the assembler; they must be loaded in from the Microchip include files (`.inc`). As will be discussed later in this chapter, the include files have all the labels in the documentation as well as other information required by the application. The `include` directive is used to copy a text file (such as the `.inc` file) into the source file.

```
include "p16F84.inc"
clrf    PORTB
bsf     STATUS, RP0
clrf    TRISB ^ 80
bcf     STATUS, RP0
goto    4
end
```

For this application, I have assumed that PIC16F84 is the PIC microcontroller used in the application and loaded its `.inc` file using the `include` directive.

Labels can also be used as addresses within the application and are located in the first column of the application. This avoids having to keep track of absolute or relative addresses. In `minimum.asm`, I can add the `forever` label to eliminate the need to count the number of instructions and explicitly put in the address to jump to.

```
include "p16F84.inc"
    clrf    PORTB
    bsf     STATUS, RP0
    clrf    TRISB ^ 80
    bcf     STATUS, RP0
forever:
    goto    forever
end
```

In the PIC microcontroller assembler, a colon character (:) is not absolutely needed to identify a label, but it should always be used to avoid any ambiguity for either the human reader or the assembler. The label should be in the first column to indicate that it is not an instruction or directive. When a label definition, such as the `forever` line above, is encountered, the label (`forever` in this case) is assigned the value of the current address.

Another way of doing the same thing in this case is to use the `$` directive as the destination of the `goto` instruction. The `$` directive returns the address of the current instruction.

```
include "p16F84.inc"
    clrf    PORTB
    bsf     STATUS, RP0
    clrf    TRISB ^ 80
    bcf     STATUS, RP0
    goto    $
end
```

In this case, the `goto $` instruction statement puts the PIC microcontroller processor into an endless loop. The `$` can be used with arithmetic operations to jump to an address that is relative to the current one. For example, `$ - 1` will place the address of the previous instruction into the source code.

Labels can be used for variables that are defined as file registers. The recommended method of doing this is to use the `CBLOCK` directive, which has the single parameter as the start of the register block. Following the `CBLOCK` and starting address statement, the variables are listed. If more than one byte is required for a variable, a colon (:) followed by the number of bytes is specified. Once all the variables have been included, the `ENDC` directive is used. The variable declaration looks like:

```
CBLOCK 0x020
i                               ; 8 Bit Variable
j:2                             ; 16 Bit Variable
k:4                             ; 32 Bit Variable
ENDC
```

After each variable declaration, a counter initialized to the starting address (the parameter of the CBLOCK statement) is incremented by the number of bytes of the variables. For the example above, *i* is at address 0x020, *j* is at address 0x021, and *k* is at address 0x023.

Accessing multibyte variables is accomplished by creating small structures using the CBLOCK directive and using the offsets of the structure elements to access the different bytes of the variable like this:

```
CBLOCK 0                ; Structure to Define a 16 Bit Number
LowByte                 ; Least Significant 8 Bits
HighByte                ; Most Significant 8 bits
ENDC
```

Using the structure, the 16-bit variable *j* can be accessed like this:

```
movlw   High 1234        ; Load "j" with Decimal 4660
movwf   HighByte + j     ; High Byte Loaded with 0x012
movlw   LOW 1234         ;
movwf   LowByte + j      ; Low Byte Loaded with 0x034
```

LOW always returns the least significant byte of the constant, HIGH returns the second least significant byte of the constant, and UPPER returns the most significant byte. For variables larger than 16-bit, HIGH and UPPER can be a problem because they do not limit the returned value to 8 bits. Instead, I use the assembler calculator, as in the example below, to load the 4 bytes of *k* (with the offsets specified by *byte#*) with a 32-bit constant:

```
movlw   LOW 0x012345678   ; Load "k" with the 32 Bit Constant
movwf   k + byte0         ; Load Byte 0 of "k" with 0x078
movlw   (0x012345678 >> 8) & 0xFF
movwf   k + byte1         ; Load Byte 1 of "k" with 0x056
movlw   (0x012345678 >> 16) & 0xFF
movwf   k + byte2         ; Load Byte 2 of "k" with 0x034
movlw   (0x012345678 >> 24) & 0xFF
movwf   k + byte3         ; Load Byte 3 of "k" with 0x012
```

The second way of defining variables is to define their addresses as constants. Constants are text labels that have been assigned a numeric value using the EQU directive and may be referred to as "equates." For example, the statement:

```
PORTB_REG EQU 6          ; Define a different value
```

is used to assign the value 6 to the string PORTB_REG. Each time PORTB_REG is encountered in the application code, the MPASM assembler substitutes the string for the constant 6.

Constants can be set to immediate values, as shown above, or they can be set to an arithmetic value that is calculated when the assembler encounters the statement. The

reason for this caveat will be explained below. An example of a constant declaration using an arithmetic statement is shown here:

```
TRISB_REG EQU PORTB_REG + 0x080
```

In the second `EQU` statement, the `TRISB` register is assigned the offset of the `PORTB` register plus `0x080` to indicate that it is in Bank 1.

I do not recommend using equates for variable definitions. The `CBLOCK` directive is somewhat simpler (and requires fewer keystrokes) and keeps track of variable addresses for you if you add or delete variables.

The address of code can be set explicitly with the `org` directive. This directive sets the starting location of the assembly programming. Normally, the start of a PIC microcontroller application is given the `org 0` statement to ensure that code is placed at the beginning of the application:

```
include "p16F84.inc"
org      0
  clrf   PORTB
  bsf    STATUS, RP0
  clrf   TRISB ^ 80
  bcf    STATUS, RP0
  goto   $
end
```

This is not absolutely required for this application as the assembler is reset to zero before it starts executing. It is a good idea to do it, however, to make sure someone reading the code will understand where it begins.

For your initial PIC microcontroller applications the only time you will not use the `org 0` statement is when you are specifying the address of the PIC microcontroller's interrupt vector (which is at address `0x0004`). A typical application that uses interrupts will have initial statements like:

```
org      0
goto     Mainline

Int
org      4
:
Mainline
:
; Interrupt Handler
; Mainline Code
```

One of the biggest differences between the PIC microcontroller and other microcontrollers is the configuration fuse register. This register is defined differently for each PIC microcontroller part number and contains operating information, including:

- Program memory (code) protection
- Oscillator type

- Watchdog timer enable
- Power-up wait timer enable
- Emulator mode enable

These fuses are specified in the source file using the `__CONFIG` directive. This directive takes the bit value of its single parameter and stores it at the configuration fuse register address. For the mid-range devices, this is address 0x02007. So the statement:

```
__CONFIG 0x01234
```

stores the value 0x01234 at address 0x02007. This statement is equivalent to:

```
org    0x02007
dw     0x01234
```

The fuse values and states are defined in the PIC microcontroller include (.inc) file. As I have indicated elsewhere, when you begin working with a PIC microcontroller device you should understand what the configuration options are and make sure that you include all of these options in your `__CONFIG` statement.

When specifying configuration fuse values from the include file, each parameter should be ANDed together. This way any reset bits will be combined to produce the value that is loaded into the configuration fuse register.

In the minimum application, which uses the PIC16F84, there are four configuration fuses you should be aware of:

- Oscillator
- Watchdog timer
- Power-up timer
- Program memory code protection

In this application, I want to use some fairly typical settings: the crystal oscillator (`_XT_OSC`), no watchdog timer enabled (`_WDT_OFF`), the power-up timer enabled (`_PWRTE_ON`), and no program memory code protection (`_CP_OFF`). To combine these settings into a single value for the configuration fuses, I add the statement:

```
__CONFIG _XT_OSC & _WDT_OFF & _PWRTE_ON & _CP_OFF
```

to the application code, which changes it to:

```
include "p16F84.inc"
__CONFIG _XT_OSC & _WDT_OFF & _PWRTE_ON & _CP_OFF
org    0
clrf   PORTB
bsf    STATUS, RP0
clrf   TRISB ^ 80
bcf    STATUS, RP0
```


This adds a lot to help understand what is happening in the application. Note that not every line has a comment; I have tried to only comment the instructions which change the contents of registers, not the currently executing page, to allow the programmer who will be working with this code to try and understand what is happening here better.

It probably seems a bit tight. To alleviate this, blank lines (whitespace) are added to break up functional blocks of code and make the code easier to understand.

```
; minimum.asm — A simple Application that turns on the LEDs that are
;   connected to all the PORTB Pins.
; Author: Myke Predko
; 00.01.06

LIST R=DEC

include "pl6F84.inc"

_CONFIG _XT_OSC & _WDT_OFF & _PWRTE_ON & _CP_OFF

org      0

    clrf   PORTB                ; LED is ON when Port Bit is Low

    bsf    STATUS, RP0
    clrf   TRISB ^ 0x080        ; Enable all of PORTB for Output
    bcf    STATUS, RP0

    goto   $                    ; When done, Loop forever

end
```

Now the application is in a format that should be reasonably easy to understand and see what is happening. The header comments are in different format from what I will use in the book, but you should get an idea of what each line is responsible for. Using labels, comments, and whitespace, you will greatly enhance the readability of your application.

Assembler Directives The MPASM assembler is rich with directives— assembler instructions that can be used to make your application programming easier. The directives are executed by the assembler to modify, enhance, or format the source file before assembling the source code into an object or hex file. Directives are *not* placed in the first column of a source file. To help me differentiate them, I place them on the second column of the source file with only labels starting in the first column. I place source code at the third column.

Table 3.10 lists all the assembler directives used in MPLAB along with examples for their use and any comments I have on using them. For directives that can only be used with another directive, I have provided a notation to the prerequisite directive.

TABLE 3.10 MPLAB IDE ASSEMBLER DIRECTIVES

DIRECTIVE	USAGE EXAMPLE	COMMENTS
<code>__BADRAM</code>	<code>__BADRAM Start, End</code>	Flag a range of file registers that are unimplemented.
<code>BANKISEL</code>	<code>BANKISEL <label></code>	Update the IRP bit of the STATUS register before the FSR register is used to access a register indirectly. This directive is normally used with linked source files.
<code>BANKSEL</code>	<code>BANKSEL Label</code>	Update the RPx bits of the STATUS register before accessing a file register directly. This directive is not available for the low-end devices (for these devices, the FSR register should be used to access the specific address indirectly). This directive is also not available for the high-end PIC microcontrollers, which should use the <code>movlb</code> instruction.
<code>CBLOCK</code>	<code>CBLOCK Address Var1, Var2 VarA:2 : ENDC</code>	Define a starting address for variables or constants that require increasing values. To declare multiple byte variables or constants that increment by more than one, a colon (:) is placed after the label and before the number to increment by. This is shown for VarA in the usage example. The <code>ENDC</code> directive is required to turn off the <code>CBLOCK</code> operation.
<code>CODE</code>	<code>CODE Address</code>	Used with an object file to define the start of application code in the source file. A label can be specified before the directive to give a specific label to the object file block of code. If no address is specified, MPLINK will calculate the appropriate address for the <code>CODE</code> statement and the instructions that follow it.
<code>__CONFIG</code>	<code>__CONFIG Value</code>	Set the PIC microcontroller's configuration bits to a specific value. <code>__CONFIG</code> automatically sets the correct address for the specific PIC microcontroller. The value is made up of constants declared in the PIC microcontroller's .inc file.
<code>CONSTANT/ = EQU</code>	<code>CONSTANT Label = Value <or> Label = Value <or> Label ≡ Value</code>	Define a constant using one of the three formatting methods shown in the usage example. The constant value references to the label and is evaluated when the label is defined. For replacing a label with a string, use <code>#DEFINE</code> .

(Continued)

TABLE 3.10 MPLAB IDE ASSEMBLER DIRECTIVES (CONTINUED)		
DIRECTIVE	USAGE EXAMPLE	COMMENTS
DA/DATA/ DB	DA Value[“string” <or> DATA Value[“string” <or> DB Value[“string”	Set program memory words with the specified data values. If a string is defined, then each byte is put into its own word. The DW directive is recommended to be used DB instead of DATA or DB because its operation is less ambiguous when it comes to how the data is stored. Note that DATA/DB/DW do <i>not</i> store the data as part of a retlw instruction. For the retlw instruction to be included with the data, the DT directive must be used. These directives are best suited for use in serial EEPROM source files.
DE	ORG 0x02100 DE Value[“string”	Save initialization data for the PIC microcontroller’s built-in data EEPROM. Note that an org 0x02100 statement has to precede the DE directive to ensure that the PIC microcontroller’s program counter will be at the correct address for programming.
#DEFINE	#DEFINE Label [string]	Specify that any time Label is encountered, it is replaced by the string. Note that string is optional and the defined label can be used for conditional assembly. If Label is to be replaced by a constant, then one of the CONSTANT declarations should be used. This directive is placed in the first column of the source file.
DT	DT Value [,Value. . .] “string”	Place the value in a retlw statement. If DT’s parameter is part of a string, then each byte of the string is given its own retlw statement. This directive is used for implementing read-only tables in the PIC microcontroller.
DW	DW Value[,Value. . .]	Reserve program memory for the specified value. This value will be placed in a full program memory word.
ELSE		Used in conjunction with IF, IFDEF, or IFNDEF to provide an alternative path for conditional assembly. Look at these directives for examples of how ELSE is used.
END	END	End the program block. This directive is required at the end of all application source files.
ENDC		End the CBLOCK label constant value, saving and updating. See CBLOCK for an example of how this directive is used.

TABLE 3.10 MPLAB IDE ASSEMBLER DIRECTIVES (CONTINUED)

DIRECTIVE	USAGE EXAMPLE	COMMENTS
ENDIF		End an IF statement conditional code block. See IF, IFDEF, or IFNDEF for an example of how this directive is used.
ENDM		End the MACRO definition. See CBLOCK for an example of how this directive is used.
ENDW		End the block of code repeated by the WHILE conditional loop instruction. See WHILE for an example of how this directive is used.
ERROR	ERROR "string"	Force an ERROR into the code with the string message inserted into the listing/error files.
ERRORLEVEL	ERRORLEVEL 0 1 2, +# -#	Suppress the assembler's response to the specific error (2), warning (1), or message (0) number (#). Specifying a hyphen (-) before the number will cause any occurrences of the error, warning, or message to be ignored by the assembler and not reported. Specifying + before the number will cause any occurrences of the error, warning, or message to be output by the assembler.
EXITM		For use within a macro to force the stopping of the macro expansion. Using this directive is not recommended except in the case where the macro's execution is in error and should not continue until the error has been fixed. Using EXITM in the body of the macro could result in phase errors, which can be very hard to find.
EXPAND	EXPAND	Enable printing macro expansions in the listing file after they have been disabled by the NOEXPAND directive. Printing of macro expansions is the default in MPLAB.
EXTERN	EXTERN Label	Make a program memory label in an object file available to other object files.
FILL	FILL Value, Count	Put in Value for Count words. If Value is surrounded by parentheses, then an instruction can be put in, such as (goto 0). In earlier versions of MPLAB, FILL did not have a Count parameter and replaced any program memory address that did not have an instruction assigned to it or areas that were not reserved (using RES) with the value.

(Continued)

TABLE 3.10 MPLAB IDE ASSEMBLER DIRECTIVES (CONTINUED)																				
DIRECTIVE	USAGE EXAMPLE	COMMENTS																		
GLOBAL	GLOBAL Label	Specify a label within an object file that can be accessed by other object files. GLOBAL is different from EXTERN as it can only be put into the source <i>after</i> the label is defined.																		
IDATA	IDATA [Address]	Used to specify a data area within an object file. If no address is specified, then the assembler calculates the address. A label can be used with IDATA for referencing it.																		
__IDLOCS	__IDLOCS Value	Set the four ID locations of the PIC microcontroller with the four nybbles of Value. This directive is not available for the 17Cxx devices.																		
IF	IF Parm1 COND Parm2 ; "True" Code [ELSE ; "False" Code] ENDIF	If Parm1 COND Parm2 is true, then insert and assemble the True code, else insert and assemble the optional "False" code. The ELSE directive and False codes are optional.																		
IFDEF	IFDEF Label ; "True" Code [ELSE ; "False" Code] ENDIF	If the label has been defined (using #DEFINE), then insert and assemble the True code, else insert and assemble the optional False code.																		
IFNDEF	IFNDEF Label ; "True" Code [ELSE ; "False" Code] ENDIF	If the label has not been defined (using #DEFINE), then insert and assemble the True code, else insert and assemble the optional False code.																		
INCLUDE	INCLUDE "FileName.Ext"	Load FileName.Ext at the current location within the source code.																		
LIST	LIST option[, . . .]	Define the assembler options for the source file. The available options are: <table><tr><th>Option</th><th>Default</th><th>Description</th></tr><tr><td>b=nnn</td><td>8</td><td>Set tab spaces.</td></tr><tr><td>c=nnn</td><td>132</td><td>Set column width.</td></tr><tr><td>f=format</td><td>INHX8M</td><td>Set the output hex file type.</td></tr><tr><td>free</td><td>FIXED</td><td>Use free-format parser.</td></tr><tr><td>fixed</td><td>FIXED</td><td>Use fixed-format parser.</td></tr></table>	Option	Default	Description	b=nnn	8	Set tab spaces.	c=nnn	132	Set column width.	f=format	INHX8M	Set the output hex file type.	free	FIXED	Use free-format parser.	fixed	FIXED	Use fixed-format parser.
Option	Default	Description																		
b=nnn	8	Set tab spaces.																		
c=nnn	132	Set column width.																		
f=format	INHX8M	Set the output hex file type.																		
free	FIXED	Use free-format parser.																		
fixed	FIXED	Use fixed-format parser.																		

TABLE 3.10 MPLAB IDE ASSEMBLER DIRECTIVES (CONTINUED)

DIRECTIVE	USAGE EXAMPLE	COMMENTS
		mm=ONIOFF ON Print memory map in list file. n=nnn 60 Set lines per page. p=type None Set PIC microcontroller type. r=radix HEX Set default radix (HEX, DEC, or OCT available) st=ONIOFF ON Print symbol table in list t=ONIOFF OFF Truncate listing lines. w=0112 0 Set the message level. x=ONIOFF ON Turn macro expansion on or off.
LOCAL	Fillup MACRO Size Local i i = 0 WHILE (i < Size) DW 0x015AA i = i + 1 ENDW ENDM	Define a variable that is local to a macro and cannot be accessed outside of the macro.
MACRO	Label MACRO [Parm[, . . .]] : ENDM	Define a block of code that will replace the label every time it is encountered. The optional parameters will replace the parameters in the macro itself.
__MAXRAM	__MAXRAM End	Define the last file register address in a PIC microcontroller that can be used.
MESSG	MESSG "string"	Cause "string" to be inserted into the source file at the MESSG statement. No errors or warnings are generated for this instruction.
NOEXPAND	NOEXPAND	Turn off macro expansion in the listing file.
NOLIST	NOLIST	Turn off source code listing output in the listingfile.
ORG	ORG Address	Set the starting address for the following code to be placed at.
PAGE	PAGE	Insert a page break before the PAGE directive.
PAGESEL	PAGESEL Label	Insert the instruction page of a goto label before jumping to that label or calling the subroutine at it.

(Continued)

TABLE 3.10 MPLAB IDE ASSEMBLER DIRECTIVES (CONTINUED)

DIRECTIVE	USAGE EXAMPLE	COMMENTS
PROCESSOR	PROCESSOR type	This directive is available for commonality with earlier Microchip PIC microcontroller assemblers. The processor option of the LIST directive should be used instead.
RADIX	RADIX Radix	This directive is available for commonality with earlier Microchip PIC microcontroller assemblers. Available options are HEX, DEC, and OCT. The default radix should be selected in the LIST directive instead.
RES	RES MemorySize	Reserve a block of program memory in an object file for use by another. A label may be placed before the RES directive to save what the value is.
SET	Label SET Value	SET is similar to the CONSTANT, EQU, and = directives, except that the label can be changed later in the code with another SET directive statement.
SPACE	SPACE Value	Insert a set number of blank lines into a listing file.
SUBTITLE	SUBTITLE "string"	Insert "string" on the line following the TITLE string on each page of a listing file.
TITLE	TITLE "string"	Insert "string" on the top line on each page of a listing file.
UDATA	UDATA [Address] Label1 RES 1 Label2 RES 2	Declare the beginning of an uninitialized data section. RES labels should follow to mark variables in the uninitialized data space. This command is designed for serial EEPROMs.
UDATA_ACS	UDATA_ACS [Address] Label1 RES 1 Label2 RES 2	Declare the beginning of an uninitialized data section in a PIC18 microcontroller. RES labels should follow to mark variables in the uninitialized data space.
UDATA_OVR	UDATA_OVR [Address] Label1 RES 1 Label2 RES 2	Declare the beginning of an uninitialized data section that can be overwritten by other files (as an overlay). RES labels should follow to mark variables in the uninitialized data space. This command is designed for serial EEPROMs.
UDATA_SHR	UDATA_SHR [Address] Label1 RES 1	Declare the beginning of data memory that is shared across all the register banks.
#UNDEFINE	#UNDEFINE Label	Delete a label that was #DEFINED.

TABLE 3.10 MPLAB IDE ASSEMBLER DIRECTIVES (CONTINUED)

DIRECTIVE	USAGE EXAMPLE	COMMENTS
VARIABLE	VARIABLE Label [= Value]	Declare an assembly-time variable that can be updated within the code using a simple assignment statement.
WHILE	WHILE Parm1 COND Parm2 : ENDW	Execute code within the WHILE/ENDW directives while the Parm1 COND Parm2 test is true. Note that in the listing file, the code will appear as if the code within the WHILE/WEND directives was repeated a number of times.

Microchip's C18 Microchip has an ANSI (1989) compatible C compiler for MPLAB IDE that can be used for developing PIC18 applications. The compiler will provide the source level debugging files (.cod and .cof) required for the MPLAB IDE simulator, ICD 2 debugger, and emulators. The generated code is very efficient and continually updated by Microchip to ensure that any bugs are fixed and available to users as soon as possible. The compiler easily integrates into MPLAB IDE and provides a fast path to working with the PIC18 microcontroller chips.

The object files produced by the compiler are location and device independent, allowing object code to be reused in a variety of applications as well as simplifying the development of applications. Some of the features you should be aware of include the ability to add inline assembly code, easy reads and writes to external memory, and a very large library of standard hardware interface drivers that you can take advantage of in your applications (these library functions are listed in App. F). The compiler generated code is highly optimized with the ability to work with a number of different pointer types, allowing you to take advantage of all the capabilities of the PIC18 architecture without having to be familiar with assembly language programming.

The compiler itself is relatively expensive, but Microchip does offer a Student Edition/Demo version, available for download from the Microchip web site. The downloaded tool has all the capabilities of the commercial compiler, except that support for some optimizations (in the area of procedural abstractions) and the extended PIC18 instruction will be disabled after 60 days. The code generated by the compiler after this time will still function normally, but you may notice that it is larger than what was originally produced. I highly recommend downloading a copy of the Student Edition/Demo C18 and installing it into your MPLAB IDE to help you start working quickly with the PIC microcontroller.

Documentation for the compiler can be found on the Microchip web site along with the Student Edition/Demo. The compiler itself is quite large (at the time of writing it is more than 24 MB), so as when installing MPLAB IDE, make sure you leave a block of time to download and install it. The documentation is up to Microchip's standard and it is well worth your while to download all the available .pdfs and print out the "Getting Started" guide.

HI-TECH Software's PICC Compiler HI-TECH Software's PICC compiler is probably the most popular compiler for the low-end and mid-range PIC microcontroller products. The current product is the result of many years of improvements and optimizations, which is critical because of the difficulty involved in developing a compiler for these PIC microcontroller architectures. The lack of a data stack, the register banks, and the minimal instruction set make it a challenge to create a compiler that produces efficient code for these chips. I have used the assembler output option and found the generated code to be very similar in quality to my code and automatically inserts many of the optimizations I know to put in my assembler code.

Like Microchip's C18, PICC is ANSI compliant and produces object code that can be linked with assembly language code to allow you to create applications very easily. Local variables are not stack based, but instead are selected from a common file register area, with the addresses used for different local variables selected from the call path the functions take. PICC can work with floating point numbers, although calculations involving them are quite slow and the library functions that use them are very large. HI-TECH Software supports the complete PIC microcontroller low-end and mid-range list of part numbers and has .inc files available for each part with the register and bit values predefined for you.

Along with the ability to be integrated easily into MPLAB IDE, HI-TECH Software has their own integrated development environment (called HI-TECH Integrated Development Environment or HI-TIDE), which can be used for developing PIC microcontroller applications. I realize that a lot of work has gone into this tool, but I would not recommend using it unless you are working with other microcontrollers that use a HI-TECH Software compiler. In that case, you would want to use HI-TIDE because it would be a common development tool for all the devices.

Installing HI-TECH Software's PICC-Lite Compiler Like C18, PICC is fairly expensive, although, like Microchip, HI-TECH Software has a version of PICC (known as PICC-Lite) that you can download and start using the compiler. The PICC-Lite compiler supports the mid-range devices listed in Table 3.11. Note that for a number of these parts, the full memory (both program memory and file registers) are not available to the compiler. Despite this limitation, PICC-Lite is an excellent tool and a highly recommended way to start programming PIC microcontrollers in C.

To install PICC-Lite, you will have to work through the following procedure:

- 1** Go to the HI-TECH Software web site, www.htsoft.com (Fig. 3.17).
- 2** Click on Demos & Free Software under the Downloads pull-down menu.
- 3** Select HI-TECH PICC-Lite to get to the introductory screen shown in Fig. 3.18.

Before you can download PICC-Lite, you will have to register with HI-TECH Software. This process requires you to give your email address, and you will be given a registration number that you can use to download later versions of PICC-Lite.

Once you have the registration information, proceed to the download page where you can download the program. After loading it into a temporary folder (such as C:\temp), you can follow the instructions on the web page, execute the program, and add the PICC-Lite compiler to your MPLAB IDE build tool options.

TABLE 3.11 PIC MICROCONTROLLERS SUPPORTED BY PICC-LITE AND ANY PROGRAM SIZE LIMITATIONS

PIC MICROCONTROLLER PART NUMBER	LIMITATIONS
PIC12F629	None
PIC12F675	None
PIC16C84	None
PIC16F627	2 RAM banks
PIC16F627A	2 RAM banks
PIC16F684	1 RAM bank, 1K program memory
PIC16F690	2 RAM banks, 2K program memory
PIC16F84A	None
PIC16F877	2 RAM banks, 2K program memory
PIC16F877A	2 RAM banks, 2K program memory
PIC16F887	2 RAM banks, 2K program memory
PIC16F917	2 RAM banks, 2K program memory

microEngineering Labs' PICBASIC For something less than C18 or PICC, microEngineering Labs' (also known as melabs) PICBASIC PRO compiler is an excellent tool for new developers to become introduced to PIC microcontrollers. The language is based on Parallax's PBASIC for the BASIC Stamp (indeed, this is how melabs got their start, by selling a compiler that produced a BASIC compiler that would compile PBASIC into PIC machine code) with a number of extensions—the language is detailed in App. E. Unlike the two C programming language compiler options listed above, PICBASIC PRO will produce code for the low-end, mid-range, and PIC18 architectures. While BASIC is inherently more limited than C, and PIC BASIC PRO does not have procedures with local variables, it is a “gentler” introduction to PIC microcontroller application programming than going directly into C or assembler.

PICBASIC PRO is not a true version of BASIC. Whereas C18 and PICC strive for ANSI compatibility, PICBASIC PRO has been optimized for PIC microcontroller applications programming. Along with the PBASIC statement set, melabs has done quite a bit to enhance the language in terms of providing additional functionality for such features as LCDs and commonly used I/O busses. The language also has structured programming statements (such as if/then/else/endif) and the ability to have assembly language inserted as part of the BASIC program files. These changes to the BASIC programming language specification have resulted in a language that is quite easy to learn and work with.

One unique feature of the language is the ability to specify the clock speed used with the PIC microcontroller. This specification is used to create internal calculations for timer delays and serial communications routines. This feature frees the programmer from

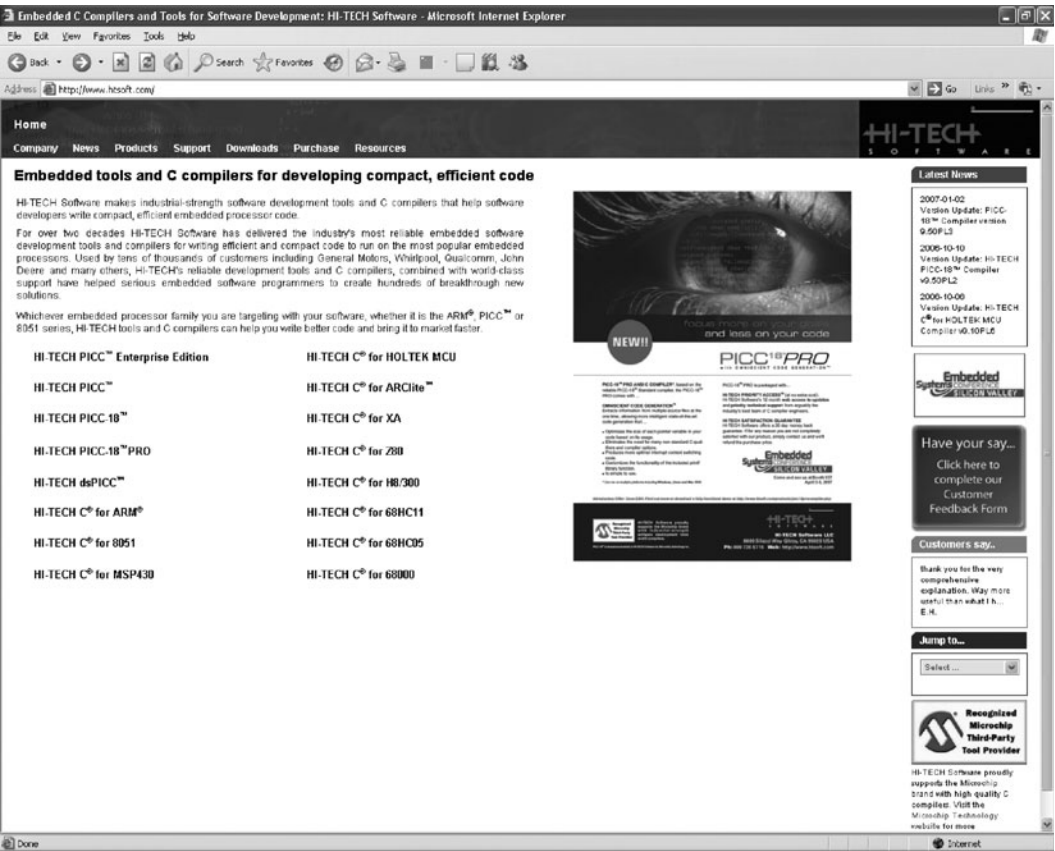


Figure 3.17 To get a copy of the PICC-Lite compiler, go to www.htsoft.com.

having to manually calculate the number of cycles and develop code to produce a specific delay or timing required for an interface function.

Along with PICBASIC PRO, melabs also offers the lower priced PICBASIC compiler, which only produces code for the mid-range devices and does not have many of the structured programming statements and advanced features of the PICBASIC PRO compiler. If you choose to purchase the PICBASIC compiler, you can upgrade to PICBASIC PRO at a later time. Both PICBASIC PRO and PICBASIC can be fully integrated with MPLAB IDE, although, at time of writing, the process is quite involved (it is fully explained on the melabs web page). I expect that in the future, the products will have better installation software, which will consist of simply running an executable.

Experimenting with CompileSpot.com If you go to the melabs web site (www.melabs.com), you will discover that there is a free version of the PICBASIC compiler available for download, similar to the free tools offered by Microchip and HI-TECH Software. Unfortunately, the free tools are limited to 31 lines of source code, which

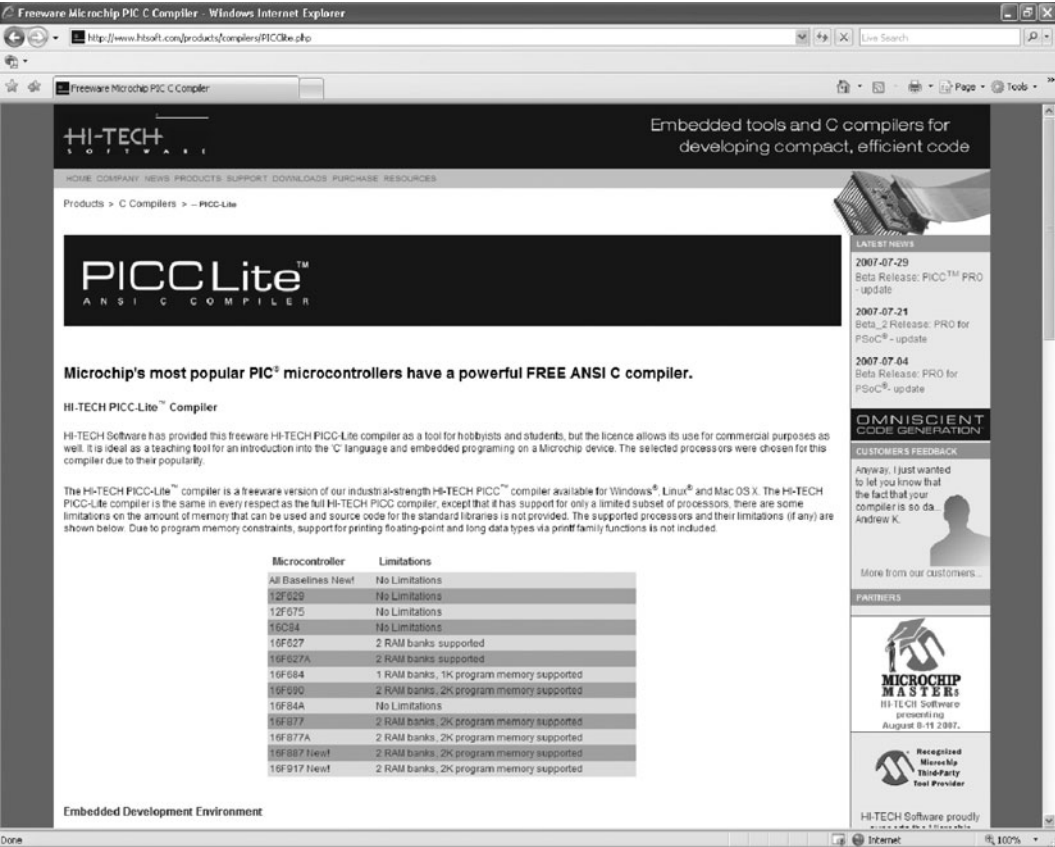


Figure 3.18 The PICC-Lite information/download page.

restricts the complexity of the projects you can develop. There is another option and that is to go to another melabs' web page, <http://compilespt.com>, which provides the free compilers (along with server access to the full compilers for a fee) and where you can develop and test simple applications (which you can download into MPLAB IDE). Later in this chapter, I will demonstrate a simple application that was created on the compilespt.com servers.

MPLINK The last tool I want to introduce is the MPLAB IDE linker tool, MPLINK. This tool is available as part of the MPLAB IDE package and allows you to combine object files produced by either the assembler or one of the compilers listed above into a single .hex file that can be programmed into a PIC microcontroller. The operation of MPLINK is controlled by a programmer-developed script file, which lists the object files to be added to a project. Linking object files together may seem somewhat scary and overwhelming, but it is actually quite simple and will allow you to take advantage of previously developed and built code instead of relying on having to include new files in your application and building them all together.

There are sample linker scripts for all of the PIC microcontrollers available in the C:\Program Files\Microchip\MPASM Suite\LKR folder of your PC. To build a program, you will start with a sample linker file like the following (which was written for the PIC16F877A):

```
// Sample linker command file for 16F877a and 876a
// $Id: 16f877a.lkr,v 1.2.16.1 2005/11/30 15:15:29 curtiss Exp $

LIBPATH .

CODEPAGE    NAME=vectors    START=0x0000    END=0x0004    PROTECTED
CODEPAGE    NAME=page0      START=0x0005    END=0x07FF
CODEPAGE    NAME=page1      START=0x0800    END=0x0FFF
CODEPAGE    NAME=page2      START=0x1000    END=0x17FF
CODEPAGE    NAME=page3      START=0x1800    END=0x1FFF
CODEPAGE    NAME=.idlocs    START=0x2000    END=0x2003    PROTECTED
CODEPAGE    NAME=.config    START=0x2007    END=0x2007    PROTECTED
CODEPAGE    NAME=eedata     START=0x2100    END=0x21FF    PROTECTED

DATABANK    NAME=sfr0       START=0x0        END=0x1F      PROTECTED
DATABANK    NAME=sfr1       START=0x80       END=0x9F      PROTECTED
DATABANK    NAME=sfr2       START=0x100      END=0x10F     PROTECTED
DATABANK    NAME=sfr3       START=0x180      END=0x18F     PROTECTED

DATABANK    NAME=gpr0       START=0x20       END=0x6F
DATABANK    NAME=gpr1       START=0xA0       END=0xEF
DATABANK    NAME=gpr2       START=0x110      END=0x16F
DATABANK    NAME=gpr3       START=0x190      END=0x1EF

SHAREBANK    NAME=gprnbnk   START=0x70       END=0x7F
SHAREBANK    NAME=gprnbnk   START=0xF0       END=0xFF
SHAREBANK    NAME=gprnbnk   START=0x170      END=0x17F
SHAREBANK    NAME=gprnbnk   START=0x1F0      END=0x1FF

SECTION     NAME=STARTUP    ROM=vectors      // Reset and interrupt
vectors
SECTION     NAME=PROG1      ROM=page0        // ROM code space - page0
SECTION     NAME=PROG2      ROM=page1        // ROM code space - page1
SECTION     NAME=PROG3      ROM=page2        // ROM code space - page2
SECTION     NAME=PROG4      ROM=page3        // ROM code space - page3
SECTION     NAME=IDLOCS     ROM=.idlocs      // ID locations
SECTION     NAME=DEEPROM    ROM=eedata       // Data EEPROM
```

and then modify it according to the needs of the application. The CODEPAGE directives are used to define areas in program memory, with the PROTECTED areas only available for object files that have code that is to execute there specifically. The DATABANK areas list the areas in each bank according to special purpose registers (which are PROTECTED

because the registers are going to be accessed explicitly) or file registers. The `SHARE-BANK` directives indicate the 16 bytes at the top of each register bank that are common across all four banks. Finally, the `SECTION` directive is used to specify the regions code can reside in.

When MPLINK executes, it locates code and data spaces for the object files and their variable areas. The ideal situation is to have an object code that can be located anywhere in memory as it makes the work of MPLINK easier. Once the object code locations are specified, MPLINK works to resolve addresses between objects and ensure that all memory objects are accessible for the different object files. When this is done, MPLINK produces the executable `.hex` file along with the `.cod` and `.cof` files.

Using MPLINK may seem like a lot of work (especially when you are first learning the PIC microcontroller), but it will help you manage your source code, optionally with built object files or putting multiple functions together as a library. It will also take care of many housekeeping functions you may have to perform, which include making sure there aren't any named segments or blocks that aren't being accessed as well as providing your applications with a single location for memory objects.

SIMULATING APPLICATIONS

I place a high value on the MPLAB IDE simulator and its ability to help you understand how the PIC microcontroller is configured at startup as well as how the application runs before you go through the effort of building the circuitry and programming the chip. I have found that spending a few minutes verifying that the application works on the simulator can save literally hours (or even days if you are new to the PIC microcontroller) thrashing around to find what the problem is. While not perfect, and unable to simulate all the various peripheral devices in a PIC microcontroller, the MPLAB IDE simulator can give you over 90 percent confidence that any application will work before you apply power to the chip. This confidence translates into PIC MCU applications that will almost always start running when installed in circuit, and if the application does not work, you can then concentrate on hardware causes for the problem rather than software issues. This capability has saved me thousands of hours over the years and allowed me to find and fix problems quickly, which makes simulation a very valuable commodity for me.

I am surprised at the number of new developers I meet who do not understand the value of simulation; they often write their code, build the applications, program the PIC microcontrollers, plug them into the application circuit, and then don't know where to begin when the application doesn't work. The simulator could have avoided much of these problems by allowing them to test the application code before trying out actual hardware. I've found that, when asked to help new developers when they encounter problems, by simply asking "have you simulated the application?" I can find the problem in just a few moments—and win a convert who will now first simulate the application before trying it out in actual hardware.

An example of the importance of using the simulator before burning an application into a PIC microcontroller can be shown in the C18 example listed at the end of this chapter. The application itself is very simple: just enable `PORTB.0` as an output, delay for some period of time, and then toggle the state of `PORTB.0` before looping around

again to the delay. I've done this program many times before, although not on the PIC18F1320 that I use in this chapter. The basic program (found in the C18\example\SimApp folder) is:

```
#include <p18f1320.h>
// SimApp - Initial PIC18 Simulator Application
//
// 07.03.21 - myke predko

int i;

void main(void)
{

    TRISBbits.TRISB0 = 0;

    while (1 == 1) {
        for (i = 0; i < 32000; i++);
        PORTBbits.RB0 ^= 1;
    } // endwhile

} // End
```

This is a typical C program for toggling the state of an LED driver pin and I expected it to work without any problems. The use of the `TRISB` and `PORTB` bits are a bit unusual. Before writing the program, I checked the `p18f1320.h` file and got the pin definitions from there. To make sure I wouldn't have any surprises, I created the project shown in Fig. 3.19, set a breakpoint at `PORTBbits.RB0`, clicked the Run button (shown in Fig. 3.20), and expected the RB0 pin to toggle each time through the loop—unfortunately, this didn't happen.

Before I explain what I did to fix the problem, I want to explain how the simulator was enabled and what the buttons shown in Fig. 3.20 do, and show you how to set breakpoints in the program. Once an MPLAB IDE project has been created (which is described for C18 projects at the end of the chapter), to enable the simulator, click on the Debugger pull-down menu, click on Select Tool, followed by MPLAB SIM. When this is done, the simulator buttons shown in Fig. 3.20 will appear on the IDE toolbar. When the buttons are displayed, you are now ready to simulate (or, if you selected one of the other tools such as ICD 2, debug or run an emulator).

The simulator buttons are the basic controls needed to reset, execute, single-step, or step over or out of the current subroutine. The Run button does just that, it executes the program and will continue to until it encounters a breakpoint, the Pause button is pressed, or an execution error (such as a stack overflow) is encountered. In early versions of MPLAB IDE, there was a problem with running the application—either a “speed up” program would have to be running in the background or the user would have to move the mouse continually to get full speed out of the simulator. I'm mentioning this because you may see some references to this requirement in some PIC microcontroller resources and it is no longer required in the latest versions of the program.

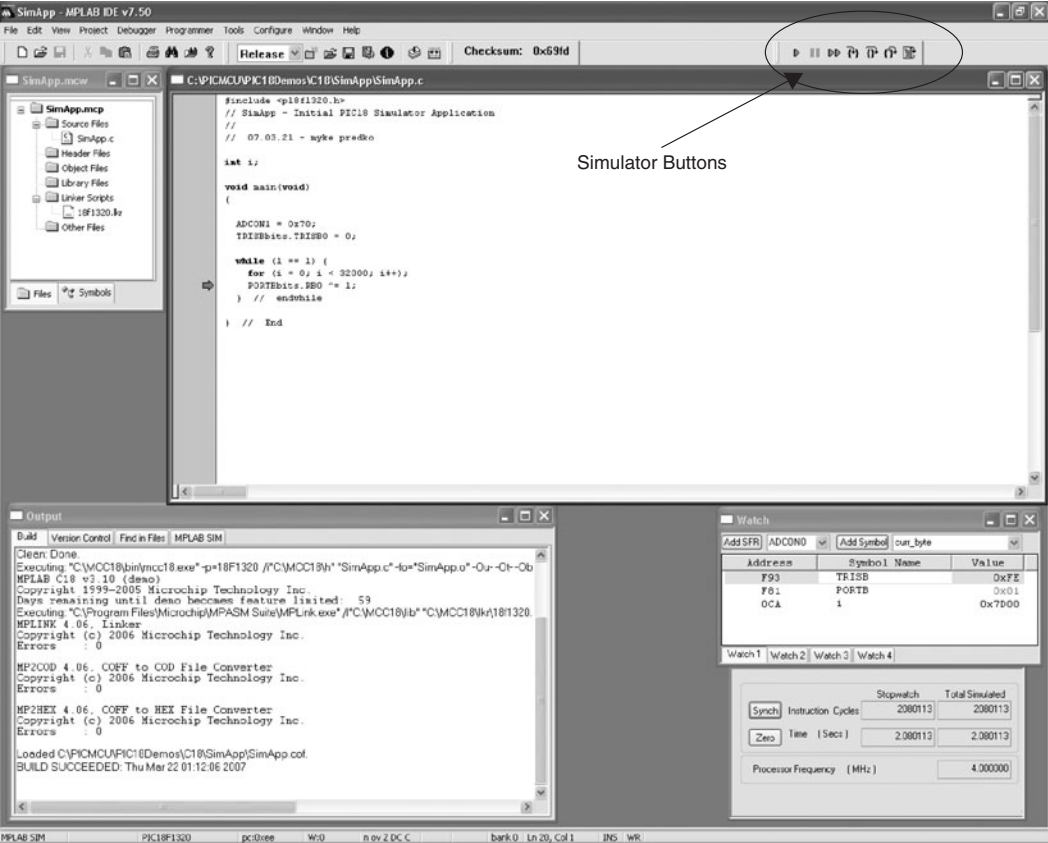


Figure 3.19 The MPLAB IDE desktop with the simulator enabled.

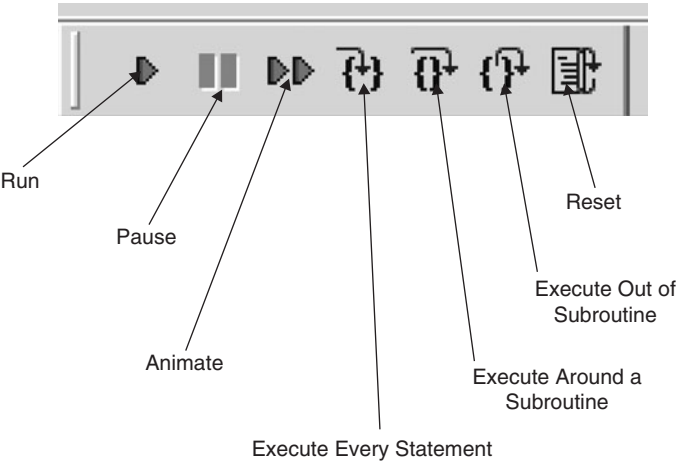


Figure 3.20 The MPLAB IDE simulator control buttons.

Along with running the application at full speed, you can also “animate” it, which will allow the program to single-step through at a speed that should be observable by a human (the speeds are selectable from the Debugger pull-down menu, selecting the Settings dialog box). The Debugger menu provides you with a number of parameters for the basic functions as well as some additional useful features that are useful that are not available from the seven basic buttons. When you are running the application, you can stop it at any time by clicking on the Pause button. This button just pauses the execution of the program and does not reset it or start again from the beginning; you can resume execution right at the point where the program paused.

There are three single-step execution options which consist of basic single-stepping and single-stepping until a subroutine call is encountered and then execute through the subroutine and stop at the instruction after the “call” instruction as well as executing at full speed until the next subroutine “return” instruction is executed. These buttons allow you to work through the application code surprisingly quickly and efficiently. Finally, there is a Reset button, which returns the simulated PIC microcontroller to its power on state and startup vector address. For the most part, you will not require any of the additional capabilities available from the Debugger menu.

To set a breakpoint in the program, simply double-click the space to the left of the source code line. When you do this, a stop sign icon will appear in the space. Now the application execution will stop when a breakpoint is encountered. To remove the breakpoint, simply right-click on the stop sign and then click on Remove Breakpoint.

As a final note, if you change any of the source code, you will not be able to continue with the simulation (or debugging or emulation). To resume these operations, you will have to rebuild the application and then restart it from the beginning. MPLAB IDE is actually pretty good at keeping track of breakpoints, so when you add or delete lines, you will see that the breakpoints will follow the program statements they were associated to, not be locked to their line numbers.

Going back to the SimApp.c C18 application, when I simulated it I found that the RB0 bit would not toggle—it always stayed at 0. By reading the datasheet, I discovered that the RB0 bit (along with some others) is initially set to be an analog input; to change the pins operation to digital I/O, I had to add the line `ADCON1 = 0x70` before the setting of `TRISB` pin zero. If I had not used the simulator to find this problem, I would have been stuck at first guessing at whether or not the PIC microcontroller was executing (which means checking clocks, power, and reset) followed by guessing at what the problem is. By using the simulator, I could see that the RB0 pin never changed state so I could concentrate my research on this pin and I discovered that the problem was actually that I was attempting to write a digital value to an analog input pin—the `ADCON1 = 0x70` statement changes RB0, RB1, and RB4 to digital I/O pins.

The three operations, enabling the simulator, using the seven simulator buttons on the IDE toolbar, and adding or removing breakpoints are all you have to know to start simulating your PIC microcontroller applications. This is not to say that you will be efficient at finding bugs right from the start, but you will be quite a bit faster than if you were trying to figure out the problem from the behavior of the application.

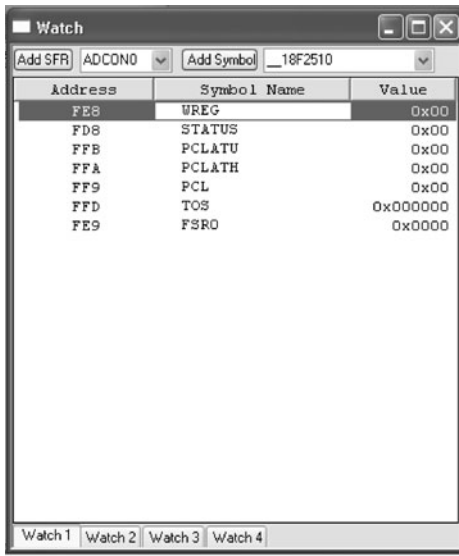


Figure 3.21 The Watch window allows you to observe and change the contents of various registers (and variables) during program simulation, debugging, or emulation.

Watch Window Files MPLAB has the capability of displaying specific register and bit contents in the PIC microcontroller. The Watch windows, such as the one shown in Fig. 3.21, allow you to select the registers to monitor and optionally update. The format of the data displayed in the Watch window can be specified to best illustrate the contents of the register or variable. Along with the Watch window, you could specify the File Registers window, but this one takes the guesswork out of figuring out which register you want to look at and the actual value of its contents. By clicking on the value of the register, a cursor will appear and you can change the contents of the register in the Watch window. The Watch window is a very useful tool when you are debugging an application and will help you understand exactly what is going on in the PIC microcontroller when your code is running.

Creating a Watch window is very simple: click on the View pull-down menu and then select Watch. Once you have the window up and placed on your MPLAB IDE desktop, you can add registers by simply selecting them from the two lists (next to Add SFR and Add Symbol) and then clicking on the buttons to their left. Don't worry if you don't get them in the order you are comfortable in—you can drag and drop the register entries in the Watch window to rearrange them. Similarly, if you don't like the data format used for the register or variable, you can change it by right-clicking on the symbol and clicking on Properties, which will give you the dialog box shown in Fig. 3.22.

Watch windows should only be started *after* the application has assembled without any errors, warnings, or messages. If there are errors when the Watch window is created, the



Figure 3.22 Changing the data format of a Watch window register is accomplished by right-clicking on the symbol and clicking on Properties.

file register information is not available to MPLAB IDE, and the list of registers available is restricted to the basic set available to the device and will not include any of the registers and variables defined in the application.

Stimulus There are very few computer applications of any type, not just PIC microcontroller ones, that can run without input of some kind and the ones that do really aren't that interesting. Responding to external events is what makes microcontrollers so important. To demonstrate how an application responds to an external event, you must provide stimulus to the simulated part while the application is running. Unfortunately, when simulating an application, it probably seems overwhelming to learn how to add various inputs, and if you have worked with preversion 7.50 of MPLAB IDE, it probably seemed quite difficult. After 7.50, though, it is simple and easy to work with. Regardless of the difficulty in creating stimulus for PIC microcontrollers, it's a good idea to do everything in your power to ensure that your applications work correctly before you burn them into a chip.

There are four methods of providing stimulus to the PIC microcontroller in MPLAB IDE:

- Asynchronous: Setting a pin, a set of pins, or a register to be driven with a specific value at an arbitrary time, initiated by something like a mouse click.
- Synchronous (known as pin/register access): Pins and registers are driven at a specific value starting at a specific point in time (or cycles) for a specific number of cycles

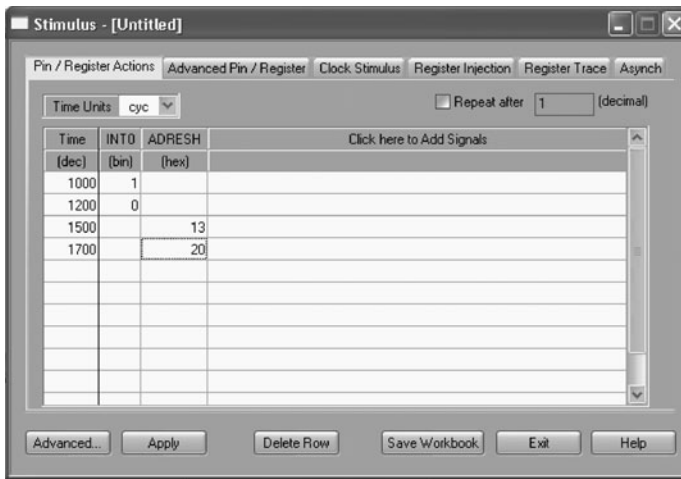


Figure 3.24 Synchronous stimulus always provides the same input.

me to use the same test case over and over (Fig. 3.24). The test case that you come up with can be saved (and later merged back into this or another application) by clicking on the Advanced button.

If you are keeping to a cycle-based measurement, you will have to calculate the instruction count using the formula:

$$\text{Instruction Count} = \text{Time Delay} * \text{Frequency} / 4$$

To get the instruction count for a 15 ms delay in a 3.58 MHz PIC microcontroller, the formula would return:

```
Instruction Count = Time Delay * Frequency / 4  
                 = 15 ms * 3.58 MHz / 4  
                 = 15(10**-3) seconds * 3.58(10**6) cycles/  
                   second / 4  
                 = 13,425 Cycles
```

In this example, the cycle step count at 15 ms is 13,462. In the synchronous Stimulus dialog box, this value would be put into the Time entry point. The step counts are absolute, so the cycle count should be added to the step values after the data pattern has been determined.

Clocks can be specified along with their frequency and the length of time the clock is high or low. The clock input is a useful tool for testing the response of a program to a single input without having to repeatedly push a button in the asynchronous Stimulus dialog box or putting in a large number of Set High and Set Low events at specific times in the synchronous Stimulus dialog box.

The last stimulus function listed above is to create an ASCII file, with each line containing a hex value that will be used as a read of an SFR to allow you to simulate the operation of the hardware device and test your code with different values. This, along with the other options of the Stimulus dialog box is quite advanced, and though not that difficult, requires a fairly sophisticated knowledge of the PIC microcontroller hardware and the application circuitry to ensure that you create the correct input for the functions and understand what the results mean.

In earlier versions of MPLAB IDE, creating stimulus for applications was tedious and inconsistent for different functions. In the latest versions of the IDE, creating stimulus files is quite easy and can be done very quickly before the application is burned onto a PIC microcontroller, allowing you to discover beforehand that the application doesn't work. A few moments spent at the start of the application will save hours scratching your head and trying to figure out why the PIC microcontroller isn't doing what you want it to.

Your First Application While you may feel like you only have a cursory introduction to the PIC microcontroller and the development process, you do have enough knowledge, as the saying goes, to be dangerous. You should have enough background to create your first simple application. The application I have chosen is to use a mid-range PIC microcontroller to flash an LED. Fig. 3.25 shows the schematic (with the bill of materials in Table 3.12) and Fig. 3.26 shows a photograph of my first prototype. The circuit should take you less than five minutes to build and will give you the opportunity to see a PIC microcontroller actually running.

The PIC16F684 was chosen for a number of reasons. From a technical perspective, it is an inexpensive Flash-based part, is available in a 14-pin PTH part, is ICSP programmable, and has a built-in oscillator. These technical specifications allow for a very simple circuit; all it needs is power and the LED/resistor output circuitry to run, and there are enough leftover pins to allow the ICSP connection to be implemented without affecting the operation of the application. Most importantly, I had one sitting on the table next to my desk so I didn't have to go very far to find a device to try out.

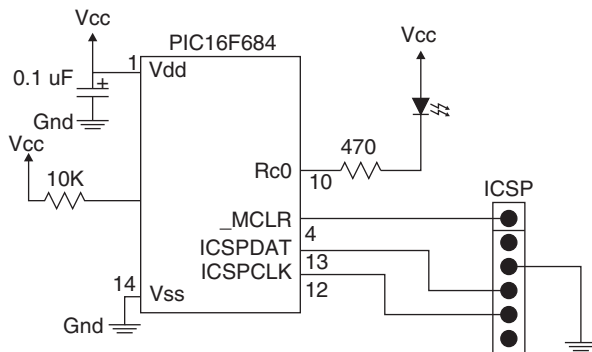


Figure 3.25 A simple circuit to turn on an LED and make it flash.

TABLE 3.12 MATERIALS NEEDED TO CREATE THE FLASHING LED CIRCUIT	
PART	DESCRIPTION
PIC16F684	PIC16F684-I/P, 14-pin PIC microcontroller
0.1 uF	0.1 uF capacitor, any type
470	470 ohm, 1/4 watt resistor
LED	Any visible light LED
ICSP	Six-pin ICSP connector in AC164110
Misc.	Breadboard, wiring, three-AA battery clip, 3x AA alkaline batteries

This last point may seem a bit facetious, but there is a certain amount of seriousness in it. The PIC16F684, like many other PIC microcontroller part numbers, has all the built-in features I needed to allow me to create this simple application very quickly. The only feature it doesn't have that I would have liked to take advantage of is the ICD hardware to let me single-step through the application.

The high level operation of the program is quite simple and could be blocked out as:

- 1 Turn off the comparators.
- 2 Turn off the ADC inputs.
- 3 Set RC0 as an output.

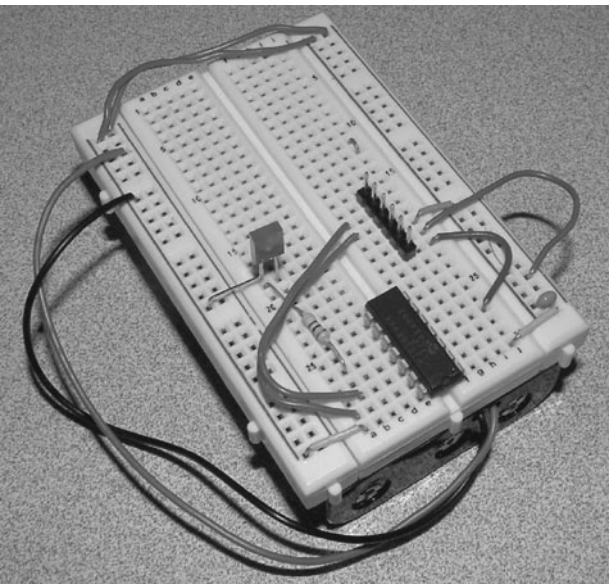


Figure 3.26 The prototype LED-flashing circuit built on a breadboard.

- 4 Delay some period of time.
- 5 Toggle RC0's state.
- 6 Go to step 4.

The need to turn off the comparators and the ADC inputs was found by reading through the datasheet and looking at the initial power-up state of the chip. I recommend reading through the datasheet first because chances are a part like this in which the I/O pins can perform multiple functions will not power up in the state you expect. As a rule of thumb, if you see ADC inputs on a PIC microcontroller pin, the part will start up with these pins defined as analog inputs and you will have to turn off this function to allow them to operate as digital I/O.

To program the PIC microcontroller, I used the MPLAB ICD 2 and the AC164110 ICD 2 to ISCP adapter with the six-pin connector that comes with the AC164110 kit. This avoided the need for me to remove the PIC microcontroller chip to program it when I was debugging the application code or testing out new parameters. I highly recommend that you look for cases in which you can leave the PIC microcontroller in circuit while you program it—this is much more convenient than removing the microcontroller, putting it into a programmer socket, and then reinserting it again into the application circuit. It is also much easier on the part itself and less likely to result in bent pins, which will render the chip useless.

I used a very small breadboard for the circuit, to which I attached, using the breadboard's two-sided tape, a three-AA battery clip. This is a very convenient way of combining the breadboard with its power supply to allow it to be moved and stored easily and safely.

PICC-Lite Flashing LED Once I had the circuit built, I created the following C program for the HI-TECH Software PICC-Lite compiler:

```
#include <pic.h>

/*  cFlash.c - Simple C Program to Flash an LED on a PIC16F684
    RC0 - LED Negative Connection

    myke predko
    07.04.01

*/

__CONFIG(INTIO & WDTRDIS & PWRTEEN & MCLRDIS & UNPROTECT \
    & UNPROTECT & BORDIS & IESODIS & FCMDIS);

int i, j;

main()
{
```

```
PORTC = 0;
CMCON0 = 7;           // Turn off Comparators
ANSEL = 0;           // Turn off ADC
TRISC0 = 0;           // Make RC0 Output

while(1 == 1)         // Loop Forever
{
    for (i = 0; i < 255; i++) // Simple 500ms Delay
        for (j = 0; j < 32; j++);

    RC0 ^= 1;          // Toggle LED
} // elihw
} // End C684Flash
```

This program can be found in the PICDwnld\C folder and is called C684Flash.c.

The MPLAB IDE project was built using the Project Wizard (found in the Project pull-down menu) and consisted of me selecting the PIC16F684, followed by the HI-TECH Software PICC-Lite compiler and then the C684Flash.c program. When I had created the project, I then tried building the code followed by programming the part.

When I specified how the MPLAB ICD 2 was to operate in circuit, I specified that the PIC microcontroller should have its own power supply—as you can see above, I specified three AA alkaline batteries, which produced 4.5 volts for programming the chip. When I attempted to program the part, I received an error message indicating that the 4.5 volts was not enough (5.0 volts were required) to program the part. In response to this, I disconnected the power to the breadboard and selected the MPLAB ICD 2 programming power option and the programming proceeded without any problems.

It was interesting to see the program work with the MPLAB ICD 2 still connected. The debugger hardware provides more than enough current to drive the PIC microcontroller and the LED, and once the programming operation was complete, the MPLAB ICD 2 MCLR# driver was disabled, allowing the application to start running.

To test the program, I changed the final value of *j* in the second `for` statement. When I originally wrote the program, I finished the loop when *j* was 132. To see if reducing this number would speed up the flashing of the LED, I changed it to 32, the value that you see in the source code above. I would recommend that you attempt this type of change when you create your first applications to get an idea of exactly how they work and how the code operates in the PIC microcontroller.

Assembler Flashing LED Once I had the PICC-Lite program running on circuit, I created PIC684Flash.asm, which can be found in the PICDwnld\Assmblr\PIC684Flash folder:

```
title "asmFlash - PIC16F684 Flashing LED"
;
;
```



```

; Hardware Notes:
; PIC16F684 running at 4 MHz Using the Internal Clock
; Internal Reset is Used
; RC0 - LED Control
;
;
; Myke Predko
; 07.04.01
;
LIST R=DEC
INCLUDE "p16f684.inc"

__CONFIG _FCMEN_OFF & _IESO_OFF & _BOD_OFF & _CPD_OFF & _CP_OFF &
_MCLR_ON & _PWRTE_ON & _WDT_OFF & _INTOSCIO

; Variables
CBLOCK 0x20
Dlay:2
ENDC

PAGE
; Mainline

org      0

nop                                ; For ICD Debug

clrf     PORTC                     ; Initialize I/O Bit to Off
movlw    7                         ; Turn off Comparators
movwf    CMCON0
bsf      STATUS, RP0               ; Execute out of Bank 1
clrf     ANSEL ^ 0x080             ; All Bits are Digital
bcf      TRISC ^ 0x080, 0
bcf      STATUS, RP0               ; Return Execution to Bank 0

Loop:                                         ; Return Here after D0 Toggle
clrf     Dlay + 1                   ; High 8 Bits for Delay
clrf     Dlay                       ; Low 8 Bits for Delay
DlayLoop:
goto     $ + 1                      ; Three Cycle Delay
nop
movlw    1                         ; Decrement the Inside Loop
subwf    Dlay, f
btfss    STATUS, Z                 ; Skip if Zero Flag is Set
goto     DlayLoop                  ; Else, Loop Again
decf     Dlay + 1, f               ; Decrement the High Byte Until
btfss    STATUS, Z                 ; It Equals Zero
goto     DlayLoop

```

```
movlw    1                ; Toggle RC0
xorwf    PORTC, f

goto     Loop             ; Repeat

end
```

A project was created for this application in exactly the same fashion as the C program (but with MPASM selected as the build tool) and was programmed exactly the same way—and when I finished, the chip just sat there and did not execute the same way the C version did.

After a bit of experimenting, I decided to remove the MPLAB ICD 2 and power the application circuit from the three AA batteries to see what would happen. The LED started flashing. It seems that the MPLAB ICD 2's MCLR# driver did not go to a high impedance state, like it did in the PICC-Lite version of the application. I'm mentioning this because you should remember that the hardware will not always work in exactly the same way for different projects and software build tools.

PROGRAMMING PIC MICROCONTROLLERS

When Microchip published the datasheets and other technical information on their mid-range products, they took the unusual (for the time) step of publishing the programming specifications without requiring a nondisclosure agreement (NDA). The programming interface and connections, which are now known as ICSP (for in-circuit serial programming), are quite simple and can be implemented easily with standard personal computer interfaces. With this information available, many hobbyists (as well as smaller programmer vendors) started producing programmer designs that allowed students, other hobbyists, and professionals to buy or create their own PIC® microcontroller development tools for modest amounts of money. Along with allowing others to develop programmers for their parts, Microchip was also one of the first manufacturers to incorporate electrically erasable programmable read-only memory (EEPROM, as well as Flash memory which is related to EEPROM) for program memory that does not require windowed ceramic packages, and UV erasers to erase the chips so new programs can be loaded into them. This strategy made the PIC microcontroller the choice of many people getting into microcontrollers for the first time—and it's why I can offer this book with a PCB with which you can build your own programmer for very little cost.

In this chapter, I will introduce to you the files used to store program data for loading into the microcontrollers and the algorithms used to program various PIC® MCU families. For additional information, I recommend that you download Microchip's datasheets explaining the important points for programming PIC microcontrollers. Along with the theory, I will also discuss some approaches used for programmer designs before going on to the next chapter, in which I will show you how to create your own programmer using the PCB that comes with this book.

Hex File Format

The purpose of assemblers and compilers is to convert application source code into a data format that can be used by a programmer to load the application into a PIC microcontroller. The most popular format (used by Microchip and most other programmers, including the two presented in this chapter) is the Intel 8-bit hex file format.

When an application is built (assembled or compiled), a hex file is generated. It may seem unnecessary to explain this, but the file is referred to as a “hex” file because that is the filename extension given to the generated file. For example, a simple application hex file could look like:

```
:10000000FF308600831686018312A001A101A00B98
:0A0010000728A10B07288603072824
:02400E00F13F80
:00000001FF
```

Each line consists of a starting address and data to be placed starting at this address. The offsets of each line have their own functions, which are explained in Table 4.1.

Each pair of characters makes up an ASCII byte, with the most significant nybble coming first, followed by the least significant nybble. Some of the data is represented

TABLE 4.1 THE FUNCTION OF THE OFFSETS ON EACH LINE OF A HEX FILE	
OFFSET FROM START OF LINE	FUNCTION
0	Always : and used to indicate the start of a new line.
1–2	Two times the number of 2-byte instructions on the line in hexadecimal with most significant digit first. There can be up to eight instructions (for a value of 16 or 10 hexadecimal).
3–6	Two times the starting address for the instructions on the line. The address has the most significant digit first and the least significant digit last.
7–8	The line type (00 = data, 01 = end).
10–13	The first instruction to be programmed into the PIC microcontroller. The data format is loaded with the first 2 bytes representing the 2 least significant nybbles of the instruction and the next 2 bytes being the 2 most significant nybbles representing the most significant nybbles.
14–17, ...	Additional instructions on the line.
Last 2	The checksum of the contents of the line.

by 4 bytes—which will translate to 2 bytes (16 bits) of actual data—with each pair of bytes used to make up a byte of data or address.

The next 4 bytes (characters) indicate twice the starting address of the data on the line. If there was a break in the code, say an instruction at address 0 and a break until address 4, the hex file would look something like:

```
:020000000728CF
:0800080029150B1109008316F4
```

After each instruction is loaded into the PIC microcontroller's program memory, an internal counter is incremented. When a line is finished, this counter is usually at the correct value for the next line, but if it is not, it is incremented until it is the same as the line's address. This means that if there are gaps in the application, the addresses will be left unprogrammed.

Note that the second line ends at address 8 boundary (the next line of data will start at address 0x008, the following one at 0x010, and so on). This is not necessary, but a convention used by the MPASM assembler.

The next 2 bytes specify the line type. Normally, this is 00, indicating that the line is data, but when it is 01, it indicates that the line is the end of the file.

The instruction data bytes follow the data type bytes. Each 4 bytes represents the instruction that is to be loaded into the PIC microcontroller's program memory. Depending on the PIC microcontroller architecture used, 12 or 14 bytes are required for the instruction, but 16 bits will always be used to store the instruction, with the top 4 or 2 bits, respectively, being zeros. Unlike the address bytes, the instruction bytes are saved in Intel format, which means the first 2 bytes are the least significant bytes of the instruction. The instruction bytes are not multiplied by two.

The last 2 bytes of each line of the hex file are the checksum of the line. This value is used to confirm the contents of the line and ensure that when all bytes of the line are summed the least significant 8 bits are equal to 0x000. This value is calculated by taking the least significant 8 bits of the sum of the line and subtracting it from 0x0100.

Using the second line of the example hex file above:

```
:0A0010000728A10B07288603072824
```

The sum of all the bytes (except for the checksum bytes is):

```
0A
00
10
00
07
28
A1
0B
07
28
```

```
86
03
07
+ 28
-----
1DC
```

The least significant 8 bits (0x0DC) are taken away from 0x0100 to get the checksum:

```
0x0100
- 0x00DC
-----
0x0024
```

This calculated checksum value of 0x024 is the same as the last 2 bytes of the original line.

While I've called the 2 checksum bytes the end of each line in the hex file, each line in the file is actually terminated by an ASCII carriage return (0x0100) and line feed (0x0100) combination. This is important for homegrown programmers: because of the different way files can be read, the line feed character may or may not be present. This caused me quite a few problems with the YAP programmer, as I will detail later in the chapter.

Code Protection

In all PIC microcontrollers, one or more code protect bits are included in the configuration fuse register. These bits are used to hinder unauthorized copying or downloading of the hex file of your application once you have completed and released an application. Once the code protection bit is set for a section (or all) of program memory, program memory reads in a typical programmer returns all zeros. In some older devices, program memory data can still be read out, but it is XORed with the adjacent words to allow for verifying the contents of program memory while still making the contents unreadable. In either case, you may find it preferable to burn the application code into program memory, read it back, and then program the code protect bits before finishing the programming operation.

If you are working with EPROM program memory based PIC MCUs, the EPROM cells of the configuration word code protection bits are often covered by an opaque layer of aluminum, as shown in Fig. 4.1. This is to prevent the code protect bits from being

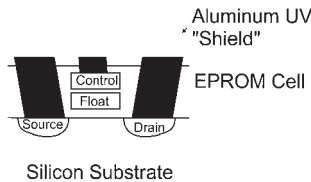


Figure 4.1 EPROM configuration fuse register cell with aluminum layer preventing selective erasure.

TABLE 4.2 CODE PROTECTION MODES AND BIT SELECTION FOR THE PIC16F877

BITS	CODE PROTECTION OPERATION
00	All program memory protected
01	The last 4K (upper half) instructions of program memory are protected
10	The last 256 instructions of program memory are protected
11	No program memory is protected

selectively erased (normally in the PIC microcontroller, when code protection is disabled, these cells are left unprogrammed) allowing the rest of program memory to be read straight back. The metal layer prevents ultraviolet erasing light from reaching the EPROM cell and effectively prevents it from ever being reprogrammed.

For this reason, I recommend that you *never* enable code protection in EPROM-based PIC microcontrollers unless you are absolutely sure of what you are doing. While some people have reported that a “deep” erase cycle of several hours to several days will clear code protect bits with the layer of aluminum over them, most have ended up with an interesting (and expensive) piece of abstract art or jewelry.

The EEPROM and Flash program memory based PIC MCU code protection is designed so that if it is set, a complete erase of the part is required before it can be reused. This will ensure that all the contents of the chip are cleared before allowing subsequent writes or reading back.

There are a lot of options for code protection in many of the PIC microcontrollers. Table 4.2 lists the four ways of specifying code protection in the PIC16F877. This allows you some interesting options and protection for your application.

While the PIC microcontroller’s code protection hardware is well designed to protect the contents of the PIC microcontroller’s program memory, it is not infallible. There are many companies that advertise the capabilities of reading code protected memory (ostensibly for legitimate companies that have lost the source code to a part). The techniques used are somewhat esoteric, but can be accomplished on lab equipment such as scanning electron microscopes, which is available in many chip-making facilities around the world.

Parallel Programming

The first part numbers of General Instrument’s PIC (peripheral interface controllers) were programmed using a parallel algorithm: an entire instruction word was presented to the microcontroller and then latched in. This method was reliable and fast but had two major drawbacks. The first was that the device had to have enough I/O pins to allow a

full instruction word as well as some handshaking, programming voltage, and control bits for the programming operation. This restriction meant that for the early low-end products, there had to be at least 15 I/O pins (which is why when you look at some of the older PIC microcontroller part numbers, like the PIC16F54, there are 18 pins—15 for programming and 2 for power—the last pin is the clock input which isn’t accessed during the programming operation) and precluded the development of smaller, low pin count products. The second issue was the added complexity of the programmer circuitry. Generally, it is easier and cheaper to create products that transfer data serially than it is to do it in parallel. By having parts that required parallel programming circuits, the PIC microcontrollers would be less attractive to students and hobbyists. All PIC microcontrollers designed after 2000 use the ICSP programming interface (described later in this chapter), but many of the earlier chips in the low-end and PIC17 families use parallel programming algorithms, which are described in the following sections.

LOW-END PROGRAMMING

The low-end PIC microcontroller requires at least 17 pins for programming, which are listed in Table 4.3. Figure 4.2 shows the block diagram for a programmer circuit that could be used in the low-end PIC microcontrollers. In this circuit, there are multiple single shots to ensure that the specified timing is achieved to program the PIC microcontroller for normal programming. A 100 μ s pulse is required, but for the configuration word, the timing is 10 ms, which is why I show the separate single shot.

When a low-end PIC microcontroller is to be programmed, the `_MCLR/Vpp` line is pulled up to 13V, while `T0CK1` is held high and `OSC1` is pulled low. The PIC MCU’s internal program counter (which is used for keeping track of the address) is initialized to 0x0100, which is the configuration fuse address.

To program a memory location, the following procedure is used:

- 1 The new word is driven onto RA0-RA3 and RB0-RB7.
- 2 The `prog` single shot sends a 100 μ s programming pulse to the PIC microcontroller.

TABLE 4.3 LOW-END PIC MICROCONTROLLER PINS AND PROGRAMMING FUNCTION	
PINS	PROGRAMMING FUNCTION
RA3-RA0	D3-D0 of instruction word
RB0-RB7	D11-D4 of instruction word
T0CK1	Program/verify clock
OSC1	Program counter input
_MCLR/Vpp	Programming voltage
Vdd	PIC microcontroller power
Vss	PIC microcontroller ground (Gnd)

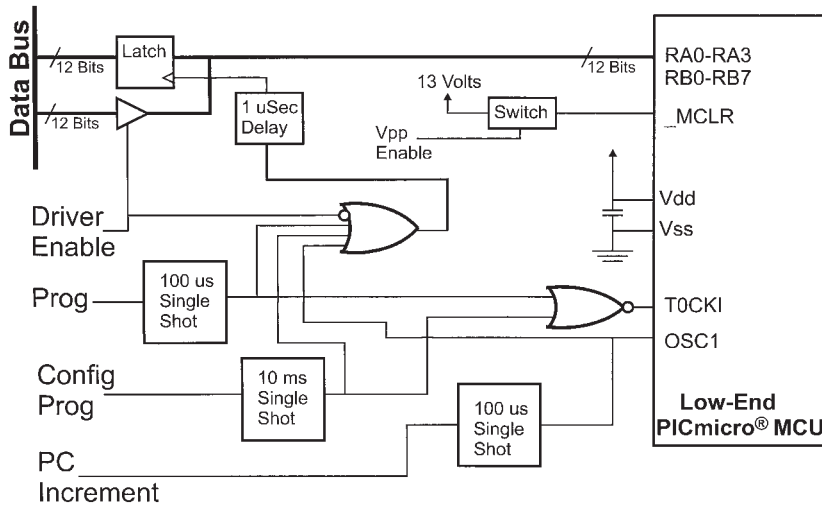


Figure 4.2 Low-end PIC microcontroller programmer circuit block diagram.

- 3** The data word driver (driver enable) is turned off.
- 4** A programming pulse is driven, which reads back the word address to confirm the programming was correct. In Fig. 4.2, the read back latch is loaded on the falling edge of the “on” gate to get the data driven by the PIC MCU.
- 5** Steps 2 through 4 are repeated a maximum of 25 times or until the data stored in the latch is correct.
- 6** Steps 1 through 4 are repeated three times more than are required to get the correct data out from the PIC microcontroller. This “overprogramming” is used to ensure the data is programmed in reliably.
- 7** OSC1 is pulsed to increment to the next address. This operation also causes the PIC microcontroller to drive out the data at the current address before incrementing the program counter (which happens on the falling edge of OSC1).

Looking at the circuit in Fig. 4.2, you are probably thinking that it is needlessly complex, and I would tend to agree with you if you were thinking of programming a low-end device using a dedicated intelligent programmer where timing pulse durations can be algorithmically produced. If you were going to use a programmer based on a PC’s serial port, then the programmable single shot chips shown in Fig. 4.2 are definitely required.

In Fig. 4.3, the programming steps 1 to 4 listed above are shown along with the latch clock signal. When programming, there must always be two T0CK1 pulses, the first being the programming pulse (10 ms or 100 μ s) and the readback. The program data word must be valid for one μ s before the T0CKI programming pulse is driven into the PIC microcontroller, and data out is available 250 ms after the falling edge of T0CKI. Note that using the circuit shown in Fig. 4.2 will result in data being driven into the readback latch because T0CKI is used for the programming pulse.

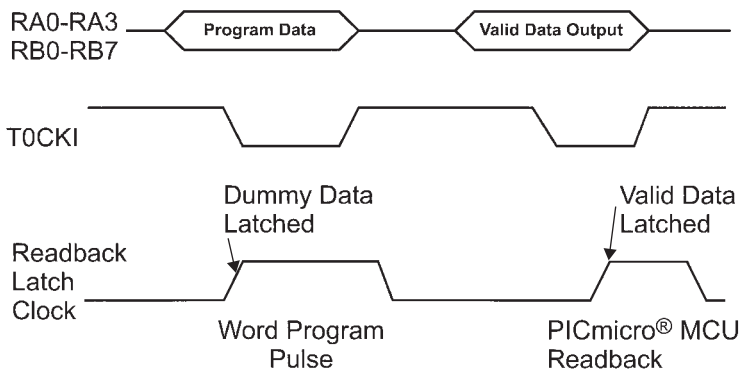


Figure 4.3 Low-end PIC microcontroller programming waveform.

There are four things to note about low-end PIC microcontroller programming. The first is that when `_MCLR` is active at 13V, the program counter is initially set to the configuration fuse register—this is different from the mid-range devices. The configuration register also requires a considerably longer pulse to program than the standard addresses.

Secondly, just pulsing the OSC1 pin can be used to implement a fast verify, as shown in Fig. 4.4. As noted above, each time OSC1 is pulsed, data at the current address will be output and then increment the PIC MCU’s program counter. Figure 4.4 shows the fast verify right from the start with the configuration fuse output first to be verified before the contents of the program memory.

Past the end of the low-end PIC microcontroller’s program memory are 4 bytes of EPROM words that can be used for serial number or application code version information. These four words cannot be accessed by the PIC microcontroller during application execution and are known as the ID location or IDLOCS.

The last point to make is that the configuration fuse register should *always* be programmed last. This means that the configuration information is skipped over when

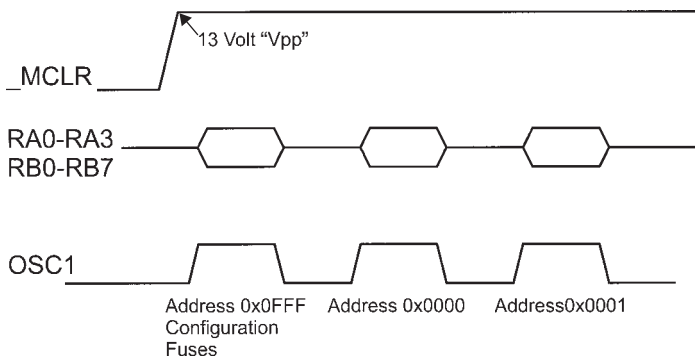


Figure 4.4 Low-end PIC microcontroller fast verifying waveform.

burning the program memory and when finished `_MCLR` is pulled low and cycled high again with the configuration fuse register programmed with its final value.

The reason for programming the configuration fuse register last is to make sure the code protect bit of the configuration register is not reset (enabled) during program memory programming. If code protection is enabled, then data read back will be scrambled during programming, which makes verification of the code impossible.

PIC17 PROGRAMMING

A PIC17 microcontroller programmer connects to the chip as shown in Fig. 4.5. Note that `PORTB` and `PORTC` are used for transferring data 16 bits at a time and `PORTA` is used for the control bits that control the operation of the programmer. The `_MCLR` pin is pulled high to 13V as would be expected to put the PIC microcontroller into programming mode.

While the programming of the PIC17Cxx is described as being in parallel, a special boot ROM routine executes within the PIC microcontroller and this accepts data from the I/O ports and programs the code into the PIC microcontroller. To help facilitate this, the `TEST` line, which is normally tied low, is pulled high during application execution to make sure that the programming functions can be accessed. The clock, which can be any value from 4 MHz to 10 MHz, is used to execute the boot ROM code for the programming operations to execute.

To put the PIC microcontroller into programming mode, the `TEST` line is made active before `_MCLR` is pulled to V_{pp} and then 0x0E1 is driven on `PORTB` to command the boot code to enter the programmer routine (this sequence is shown in Fig. 4.6). To end programming mode, `_MCLR` must be pulled to ground 10 ms or more before power is taken away from the PIC microcontroller. `TEST` should be deasserted after `_MCLR` is pulled low.

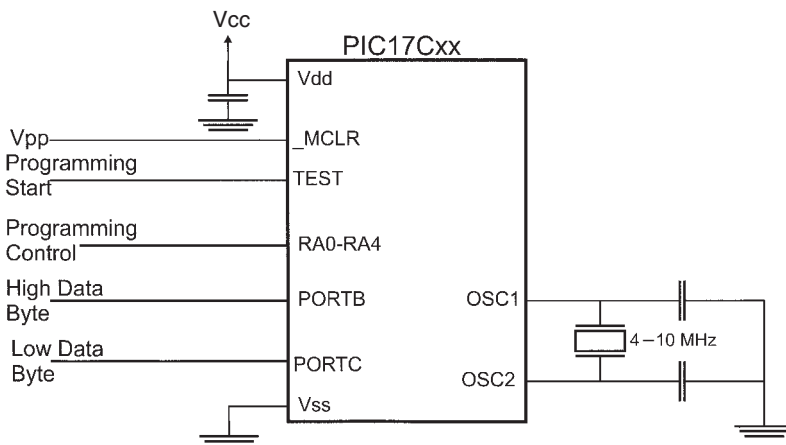


Figure 4.5 PIC17 parallel programmer connections.

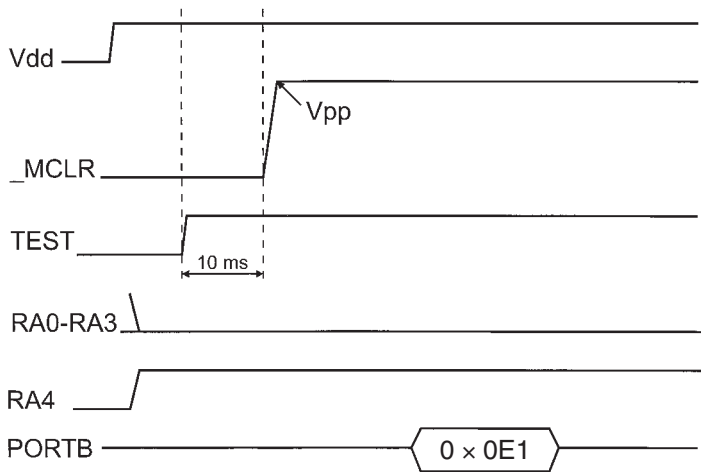


Figure 4.6 PIC17 parallel programming startup.

When programming, the RA0 pin is pulsed high for at least 10 instruction cycles (10 μ s for the PIC microcontroller running at 4 MHz) to load in the instruction address followed by the PIC microcontroller latching out the data (so that it can be verified). After the data has been verified, RA0 is pulsed high for 100 μ s to program the data. If RA1 is low during the RA0 pulse, the PIC microcontroller program counter will be incremented. If it goes high during the pulse, the internal program counter will not be incremented and the instruction word contents can be read back in the next RA1 cycles without having to load in a new address. The latter operation is preferred and looks like the waveforms shown in Fig. 4.7.

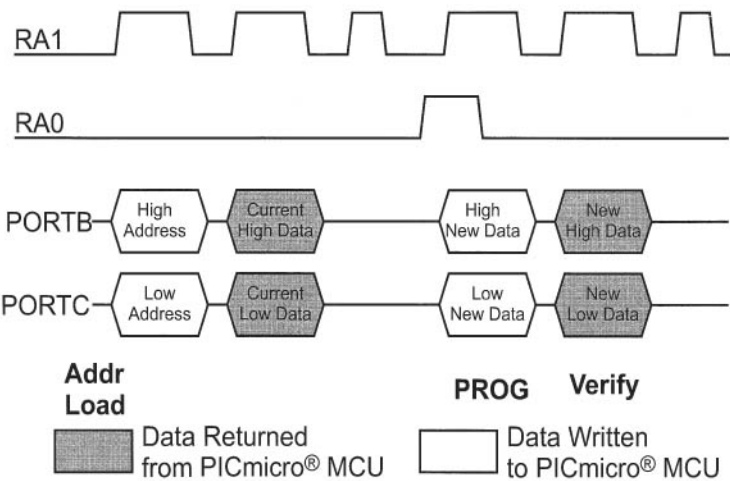


Figure 4.7 PIC17 parallel programming waveform.

This waveform should be repeated until the data is loaded or up to 25 times. Once it is programmed in, then three times the number of programming cycles must be used to lock and overprogram the data in. This process is similar to that of the other EPROM parts.

Writing to the specified addresses between 0x0FE00 and 0x0FE0F programs and verifies the configuration word. To program (make 0) one of the configuration bits, its register is written to. Reading back the configuration word uses the first three RA1 cycles of Fig. 4.7 at either 0x0FE00 or 0x0FE08. Reading 0x0FE00 will return the low byte of the configuration word in PORTC (0x0FF will be in PORTB) and reading 0x0FE08 will return the high byte in PORTC.

When writing PIC17 configuration fuse register bits, the addresses written to must be in ascending order. Programming the bit in nonregister ascending order can result in unpredictable programming of the configuration word as the processor mode changes to a code protected mode before the data is loaded in completely. This issue is important to watch out for in all PIC microcontroller programming; the configuration fuses must be programmed last, with any code protection programmed into the PIC microcontroller as the last possible programming operation.

In some Microchip documentation, you will see comments that imply that the PIC17 has some ICSP or serial programming capability. This is not entirely correct as software called a bootloader (described later in the book) is used to save data passed to the PIC17 using the ability of the chip to write to its own EPROM program memory. This software can be used to provide a rudimentary in-circuit programming capability that can be exploited in your applications.

The capability of a PIC17Cxx application to write to program memory is enabled when the `_MCLR` is driven by more than 13V and a `tablwt` instruction is executed. When `tablwt` is executed, the data loaded into the table latch (`TABLATH` and `TABLATL`) registers is programmed into the memory locations addressed by the table pointer registers (`TBLPTRH` and `TBLPTRL`). This instruction keeps executing until it is terminated by an interrupt request or `_MCLR` reset.

To perform a word write, the following bootloader code execution sequence would be used:

- 1** Disable TMRO interrupts.
- 2** Load `TABPTRH` and `TABPTRL` with the address.
- 3** Load `TABLATH` or `TABLATL` with the data to be stored.
- 4** Enable a 1,000 μ s TMRO delay interrupt (initialize TMRO and enable TMRO interrupt).
- 5** Execute `tablwt` instruction with the missing half of data.
- 6** Disable TMRO interrupts.
- 7** Read back data; check for match.
- 8** If no match, return error.

To enable internal programming, `_MCLR` has to be switched from 5V (V_{dd}) to 13V. The Microchip circuit that is recommended is shown in Fig. 4.8 and will drive the PIC17's `_MCLR` pin at 5V until RA2 is pulled low. When RA2 is pulled low, the voltage driven in to `_MCLR` will become 13V (or V_{pp}). The programming current at 13V is a minimum of 30 mA.

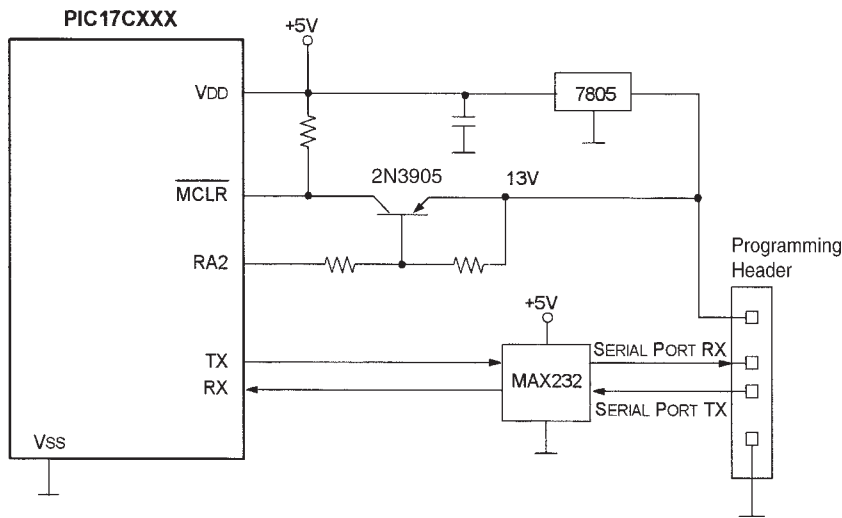


Figure 4.8 PIC17 in-circuit serial programming schematic.

Typical bootloader code for a PIC17 microcontroller would execute following the procedures:

- 1** Establish communication with programming host.
- 2** If no communication link established jump to application code.
- 3** Enable Vpp (RA2 = 0)
- 4** Wait for host to send instruction word address.
- 5** Program in the word.
- 6** Confirm word programmed correctly.
- 7** Loop back to 4.

In this process, you will probably want to program as few instruction locations as possible. This is due to the need for programming in the bootloader initially. If this code has to be programmed in, then you might as well program in the application at the same time, leaving the bootloader for programming serial numbers or calibration values. This is really the optimal use of the self-program capabilities of the PIC17 devices.

PIC ICSP Programmer Interface

The PIC microcontroller's in-circuit serial programming (ICSP) capability provides a significant advantage for developers, hobbyists, and manufacturers. The ICSP features of the PIC microcontroller allow for the use of simple programmers; the El Cheapo, presented later in this chapter, is an example of a very basic PIC MCU programmer that you can build inexpensively. This feature allows you to program PIC MCUs after they

have been assembled into the application circuit, which eliminates one manufacturing step or eliminates the need for buying specialized sockets and handling equipment for different devices. The ICSP interface has also been enhanced for a number of chips to allow debugging of the application while it is in circuit. In-circuit serial programming is one of the three reasons why the PIC microcontroller is as popular as it is (the other two are MPLAB and the wide availability of PIC microcontroller part numbers and features from a number of sources).

In this section, I want to introduce in-circuit serial programming and discuss how ICSP is implemented for the mid-range PIC microcontrollers and how programming works. In the following sections of this chapter, I will discuss some aspects of ICSP and how it is implemented for various devices as well as review some ICSP programmers that are available to you or that you could build yourself.

ICSP is a synchronous serial communications protocol in which instructions for program memory are downloaded into a PIC microcontroller when the master reset (also known as `_MCLR`) is raised about 13V. Table 4.4 reviews the wiring for various pin count PIC microcontroller devices.

To program and read data, the PIC microcontroller must be put into programming mode by raising the `_MCLR` pin to 13–14V, and pulling the data and clock lines low for several milliseconds. Once the PIC microcontroller is in programming mode, data can then be shifted in and out using the clock line. There is also a low voltage programming (LVP) mode available in some devices that doesn't require 13V V_{pp} or 5V V_{dd} —for simplicity I have just referenced the requirements for standard ICSP programming. If you are using a device that has LVP capabilities, consult the datasheet for the proper voltage levels and pin control algorithms. I do want to point out that if LVP is selected, an additional pin (often labeled `LVP`) is used to indicate when programming operations are about to take place.

When the programming voltage is applied to the `_MCLR` pin, it is important to remember that up to 50 mA has to be supplied on the V_{pp} circuit to ensure that EPROM parts will program properly (this is not an issue for Flash-based parts, which just require a few mA of current from the V_{pp} line). This 50 mA is relatively high and a relatively easy way to produce this voltage is using a 78L12 regulator with two silicon diodes used

TABLE 4.4 PIN SELECTIONS FOR PIC MICROCONTROLLER DEVICES

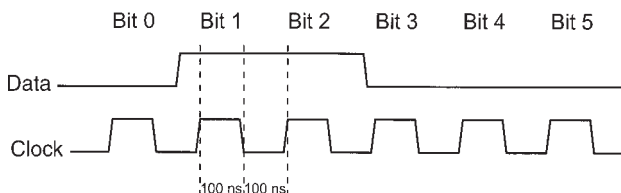
PIN	8-PIN DIP	14-PIN DIP	18-PIN DIP	28-PIN DIP	40-PIN DIP
V_{pp}	4- <code>_MCLR</code>	4- <code>_MCLR</code>	4- <code>_MCLR</code>	1- <code>_MCLR</code>	1- <code>_MCLR</code>
V_{dd} (+ Voltage)	1	1	14	26	11, 32
V_{ss} (Ground)	8	14	5	8, 21	12, 31
Data	7-GP0	13-RB0	13-RB7	28-RB7	40-RB7
Clock	6-GP1	12-RB1	12-RB6	27-RB6	39-RB6

TABLE 4.5 EPROM PROGRAM MEMORY ICSP PROGRAMMING COMMANDS

COMMAND	BIT PATTERN	DATA	COMMENTS
LoadData	0b000010	0, 14 bits, 0	Load word for programming. Multiply data value by 2 before sending to PIC MCU. Low-end architectures (12 bits) also multiply the instruction code by 2 and leave the top 3 bits zeroed out.
BeginProgramming	0b001000	None	Start programming cycle.
EndProgramming	0b001110	None	End programming cycle after preset period of time.
IncrementAddress	0b000110	None	Increment the PIC microcontroller's PC.
ReadData	0b000100	0, 14 bits, 0	Read instruction from PIC MCU's program memory. Read "Load Data" comments for low-end instructions.
LoadConfig	0b000000	0x7FFE	Set the mid-range device's program counter to 0x2000.

Data is shifted in and out of the PIC microcontroller using a synchronous protocol. Data is shifted out least significant bit first on the falling edge of the clock line. The minimum period for the clock is 200 ns with the data bit centered as shown in Fig. 4.11, which is sending an **IncrementAddress** command. When data is to be transferred, the same protocol is used, but a 16-bit transfer (LSB first) follows after 1 μ s has passed since the transmission of the command. The 16 bits consist of the instruction word shifted to the left by one. This means the first and last bits of the data transfer are always 0.

Before programming of a PIC microcontroller with EPROM program memory can start, the program memory should be checked to make sure it is blank. This is accomplished by simply reading the program memory (**ReadData** command listed in Table 4.5) and comparing the data returned to 0x07FFE. After every compare, the PIC

**Figure 4.11** ICSP programmer 6-bit command.

microcontroller's program counter is incremented (using the `IncrementAddress` command) to the size of the device's program memory. Once the program memory is checked, the program counter is jumped to 0x02000 (using the `LoadConfiguration` command) and then the next eight words are checked for 0x07FFE.

To program an EPROM program memory instruction word, a `LoadData` command (followed by the instruction value) is sent to the PIC microcontroller followed by a `BeginProgramming` command. After at least 100 ms has passed, an `EndProgramming` command is sent. This sequence is known as a programming cycle. After each programming cycle, the contents are read back and compared to the expected value.

This process is repeated up to 25 times or until the program memory is correct. If the program memory is correct, then three times the number of programming cycles needed to get the correct value are executed to ensure the instruction word is not marginally programmed. This will be a bit confusing; consider, for example, a PIC microcontroller program memory word that requires four programming cycles before the correct data is returned. After the correct data has been returned, an additional 12 programming cycles (three times the four cycles) are sent to the PIC MCU.

Once a memory location has been correctly programmed, the PIC microcontroller's program counter can be incremented. If there is nothing to program at a memory location, or the value is 0x03FFF, then you can simply send an "IncrementAddress" command to skip to the next address and ignore programming the instruction completely.

The configuration registers and ID locations are programmed the same way after sending a `LoadConfiguration` command after the program memory has been loaded and its contents verified against the expected program. By doing this, if there are any protection bits enabled in the configuration fuses, they won't affect the programming of the chip.

The process for burning a PIC microcontroller's program memory could be blocked out with the pseudocode:

```
ICSPPProgram()          // Program to be burned in is in an array of
{                        // addresses and data

int PC = 0;              // PIC microcontroller's program counter
int i, i j k;
int retvalue = 0;

    for (i = 0; (i - PGMSize) && (retvalue == 0); I++) {

        if (PC != address[i]) {
            if ((address[I] >= 0x02000) && (PC < 0x02000)) {
                LoadConfiguration(0x07FFE);
                PC = 0x02000;
            }
        }

        for (; PC < address[i]; PC++)
            IncrementAddress();
    }
```

```

for (i = 0; (i < 25) && (retvalue != data[I]); I++) {
    LoadData(ins[i] << 1);          // Programming Cycle
    BeginProgramming();
    Dlay(100usec);
    EndProgramming();

    Retvalue = ReadData();
}

if (i == 25)
    retvalue = -1;          // Programming Error
else {
    retvalue = 0;          // Okay, Repeat Programming Cycle 3x
    for (k = 0; k < (j * 3); k++){
        LoadData(ins[i] << 1);
        BeginProgramming();
        Dlay(100usec);
        EndProgramming();
    } // endif
} // endif
} // endfor
} // end ICSPProgram

```

After the program memory has been loaded with the application code, Vpp should be cycled off and on and the PIC microcontrollers program memory read out and compared against the expected contents. When this verify is executed, Vpp should be cycled again with Vdd a minimum voltage (4.5V) and then repeated again with Vdd at a maximum voltage (5V) value.

When this verify is executed at voltage margins, the PIC microcontroller is said to be production programmed. If the margins are not checked, programming operation is said to be prototype programmed. Most hobbyist programmers (including the ones presented in this book and the Microchip PICSTART Plus) are prototype programmers because they cannot margin Vdd when the chip is programmed.

Microchip uses a modified version of this programming algorithm for PIC microcontrollers that have Flash program memory. The actual programming algorithm is quite a bit simpler and programmers designed for just Flash parts are often a few basic electronic parts. What makes Flash programming different from ICSP programming is the need to erase the contents of program memory before starting to program (this operation is not required for EPROM parts, which have their contents erased using ultraviolet light).

The same connections to the PIC microcontroller are used (Fig. 4.12) for programming Flash-based chips as EPROM-based ones. Electrically, the programming voltages are basically the same as required for the mid-range devices. There is the difference (noted earlier in this section) in the voltage and current required for Vpp. For PIC microcontrollers with EPROM program memory, up to 50 mA are required for EPROM programming. The PIC MCUs with Flash-based program memory have a built-in VPP generator that provides adequate voltage and current to program while requiring very little current from the

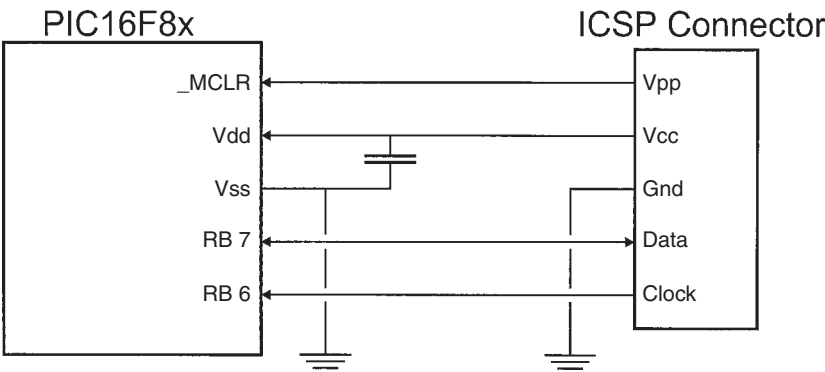


Figure 4.12 ICSP programmer connections.

programmer. The same data packet format is used for the Flash-based PIC microcontrollers, but the commands and how they work are slightly different (as shown in Table 4.6).

The data, as in the EPROM parts, is always 16 bits with the first and last bit always equal to zero. Data is always transferred LSB first using the same timings as specified earlier in the chapter for the mid-range parts. When I have designed PIC microcontroller programmers, I multiply the data word by 2 (or shift it to the left by one) to provide a 16-bit word with the first and last bit equal to zero and the data word in between.

The programming cycle for the Flash-based PIC microcontrollers is:

- 1** Load data command (000010 + data word x 2).
- 2** Begin programming command (001000).
- 3** Wait 10 ms.

There is no EndProgramming command required, but the 10 ms delay makes programming Flash parts somewhat slower. There are no multiple program/verify steps.

TABLE 4.6 FLASH PROGRAM MEMORY ICSP PROGRAMMING COMMANDS		
COMMAND	BITS	DATA
Load Configuration	0b000000	0x7FFE
Load Data for Program Memory	0b000010	0, data, 0
Load Data for Data Memory	0b000011	0, byte, 0
Read Data from Program Memory	0b000100	0, data, 0
Read Data from Data Memory	0b000101	0, byte, 0
Increment Program Counter	0b000110	None
Begin Programming	0b001000	None
Bulk Erase Program Memory	0b001001	None
BulkEraseDataMemory	0b001011	None

The only issue left to discuss is how to erase the program memory before programming. Like the EPROM devices, the Flash program memory, when erased, converts specific 1s in memory to 0s. As with the EPROM device's ultraviolet erase, the Flash erase step loads 1s in all the memory locations.

The erase operation could be accomplished using the bulk erase commands listed in Table 4.6, but I prefer to use the Microchip specified erase for specific devices. These operations will erase all Flash and EEPROM memory in the PIC microcontroller device, even if code protection is enabled. In the general case, the instruction sequence is:

- 1 Apply Vpp.
- 2 Execute load configuration (0b00000000 + 0x07FFE).
- 3 Increment the PC to the configuration register word (send 0b0000110 seven times).
- 4 Send command 0b0000001 to the PIC microcontroller.
- 5 Send command 0b0000111 to the PIC microcontroller.
- 6 Send "begin programming" (0b0001000) to the PIC microcontroller.
- 7 Wait ten ms.
- 8 Send command 0b0000001.
- 9 Send command 0b0000111.

Note that there are two commands that aren't listed in Table 4.6 (0b0000001 and 0b0000111). These two commands, Bulk Erase 1 and Bulk Erase 2, respectively, are special commands used to ensure that all Flash program memory is erased at the end of this sequence.

PIC18 PROGRAMMING

Like the PIC17, the PIC18 has the capability to "self program" using the table read and write instructions. In the PIC18, this capability is not only available within applications, but is used to program the device right from the start without the need for specialized boot ROM or ICSP interface code, unlike the PIC17. Programming the PIC18 microcontrollers can be accomplished by applying either a high or low voltage on the `_MCLR` pin (and, optionally, the `LVP` pin) and a clock and data line as in mid-range ICSP programming. The programming sequence is actually quite a bit more complicated than in the other device families, but it allows you to reprogram blocks of code fairly easily, something that the other families do not provide.

To program the PIC18, instructions are downloaded into the PIC microcontroller after setting the `_MCLR` pin to Vpp (13–14V, as in the other EPROM PIC microcontrollers). Passing instructions (which contain the program data) to the PIC microcontroller is accomplished by first sending a 4-bit "special instruction" followed by an optional 16-bit instruction. The 4-bit special instruction is sent most significant bit first and can either specify that an instruction follows or that it is a mnemonic for a `TBLRD` or `TBLWT` instruction as shown in Table 4.7.

The data transmission looks like Fig. 4.13 with a 4-bit `nop` instruction operation code transmitted first, followed by the 16-bit instruction, which is then executed. If the operation is to be a table read or write operation, then the Operation Instruction code is used

TABLE 4.7 PIC18 PROGRAMMING MNEMONICS			
SPECIAL INSTRUCTION	MNEMONIC	INSTRUCTION OPERATION	CYCLES
0000	nop	Shift in and execute next instruction	1
0010		Shift out TABLAT register	
1000	TBLRD *	Read table	2
1001	TBLRD *+	Read table, increment TBLPTR	2
1010	TBLRD *-	Read table, decrement TBLPTR	2
1011	TBLRD +*	Increment TBLPTR, read table	2
1100	TBLWT *	Write table	2
1101	TBLWT *+	Write table, Increment TBLPTR	2
1110	TBLWT *-	Write table, decrement TBLPTR	2
1111	TBLWT *, PROG	Write table, start programming	2

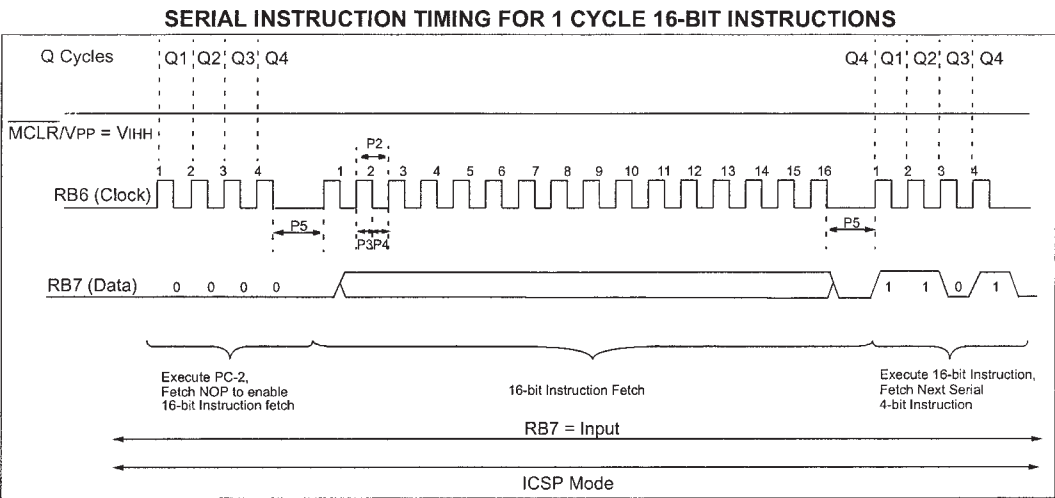


Figure 4.13 The serial instruction timing for sending one instruction that will be executed in the PIC18 microcontroller core.

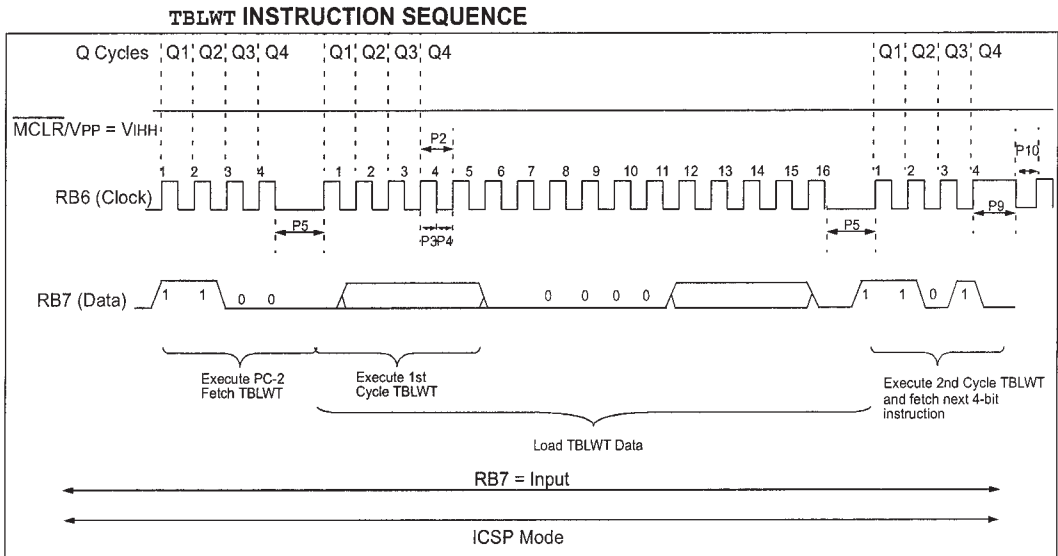


Figure 4.14 The TBLWT (table write) instruction sequence.

instead of the `nop` to simplify the data transfer; in the case of a table write, the 16 bits of the word to be burned into program memory are sent as shown in Fig. 4.14. If a read-back of the table contents is required, the data is shifted out of the PIC microcontroller.

To set up a table read or write in Flash devices, the internal PIC programming registers must be initialized. The programming write operation is controlled using the `EECON1` register and the `EEPGD` and `CFGS` bits using the following two instructions:

Mnemonic	Instruction/Data
<code>nop</code>	<code>bsf EECON1, EEPGD</code>
<code>nop</code>	<code>bsf EECON1, CFGS</code>

These two instructions only have to be executed before the start of the programming operation. They do not have to be repeated each time data is written sequentially to the PIC18 microcontroller.

Next the `TBLPTR` has to be initialized. This is done using standard `movlw` and `movwf` instruction. For example, to program address `0x12345` with the value `0x6789`, the following data sequence is written to the PIC18:

Mnemonic	Instruction/Data
<code>nop</code>	<code>movlw UPPER 0x12345</code>
<code>nop</code>	<code>movwf TBLPTRU</code>
<code>nop</code>	<code>movlw (0x12345 >> 8) & 0xFF</code>
<code>nop</code>	<code>movwf TBLPTRH</code>
<code>nop</code>	<code>movlw LOW 0x12345</code>
<code>nop</code>	<code>movwf TBLPTL</code>
<code>tblwt *</code>	<code>0x6789</code>
<code>nop</code>	<code>CLOCKHIGH</code>

The final `nop` mnemonic and `CLOCKHIGH` are used to program in the data. After sending the 4-bit `nop` mnemonic, the programming clock line is held high for 1 ms (known in the datasheets as the P9 programming time).

Up to four instructions (8 bytes in total), which is known as a “panel,” can be written at a time before the programming instruction `TABWT` is executed. Normally, this is accomplished using the `TABWT ++` mnemonic in which the table pointer is incremented by two after the 16 bits are written. At the end of the sequence the `TABWT *` mnemonic is executed, which starts the writing sequence:

```
Mnemonic      Instruction/Data
nop            movlw  UPPER StartAddress
nop            movwf  TBLPTRU
nop            movlw  (StartAddress >> 8) & 0xFF
nop            movwf  TBLPTRH
nop            movlw  LOW StartAddress
nop            movwf  TBLPTRL
tblwt ++      Word0
tblwt ++      Word1
tblwt ++      Word2
tblwt *        Word3
nop            CLOCKHIGH
```

Erasing the entire program and data memory of the chip is accomplished using the erase options listed in Table 4.8 and sending the data byte to address 0x3C0004 using the programming sequence:

```
Mnemonic      Instruction/Data
nop            movlw  UPPER 0x3C0004
nop            movwf  TBLPTRU
nop            movlw  (0x3C0004 >> 8) & 0xFF
```

TABLE 4.8 BULK ERASE OPTIONS AVAILABLE IN THE FLASH-BASED PIC18 MICROCONTROLLERS	
OPTION DESCRIPTION	DATA
Chip Erase	0x80
Erase Data EEPROM	0x81
Erase Boot Block	0x83
Erase Block 0	0x88
Erase Block 1	0x89
Erase Block 2	0x8A
Erase Block 3	0x8B


```

nop          movwf  TBLPTRH
nop          movlw  LOW 0x3C0004
nop          movwf  TBLPTRL
tblwt *      0x0080
nop          nop
nop          DATALOW

```

The last nop DATALOW sequence is the 4 bits of the nop operation instruction followed by holding the data line low for P11 or 5 ms.

The final operation that can be performed is erasing a panel. This is accomplished by using the EEPROM write capability of the PIC microcontroller with the instruction sequence:

Mnemonic	Instruction/Data
nop	bsf EECON1, EEPCD
nop	bcf EECON1, CFGS
nop	movlw UPPER PanelStart
nop	movwf TBLPTRU
nop	movlw (PanelStart >> 8) & 0xFF
nop	movwf TBLPTRH
nop	movlw LOW PanelStart
nop	movwf TBLPTRL
nop	bsf EECON1, WREN
nop	bsf EECON1, FREE
nop	movlw 0x55
nop	movwf EECON2
nop	molw 0xAA
nop	movwf EECON2
nop	bsf EECON1, WR
nop	nop
	; Wait 10 ms
nop	bcf EECON1, WREN
tblwt **	Word0
tblwt **	Word1
tblwt **	Word2
tblwt *	Word3
nop	CLOCKHIGH

It must be pointed out that the PIC18 programming commands are for just programming. It may seem like they have the capability of being used for single-stepping through program instructions to debug it, but there really is no simple mechanism for returning the value of the WREG and no guarantee that the special function registers, other than EECON1, will work. The ICD interface is well suited for this task and uses the same pin resources so the circuit does not need to be modified between the two. At the end of the chapter, I discuss how the two programming interface specifications are related and what this means to the electrical connections.

Microchip Programmers

Microchip, as part of its developer support, offers a number of programmers and programming options for its PIC microcontroller and serial EPROM products. These programmers and options are reasonably priced and integrate seamlessly to MPLAB for direct application programming, eliminating possible problems with moving hex files between applications.

Microchip programmers interface easily with the MPLAB IDE as shown in Fig. 4.15. When Picstart Plus | Enable is selected from MPLAB IDE's top pull-down line, the memory contents are displayed along with a PICSTART Plus control box and a dialog box showing how the configuration fuses will be set. The configuration values are set automatically from the values specified by the `__config` statement in the assembler source code. To burn an application into a part, normally all that has to be done at this stage is to click on Program. This level of integration has been made available for all Microchip programmers as well as debuggers and emulators and allows you to simply select the tools you would like to use with your project and click on the buttons that appear on the the MPLAB IDE desktop.

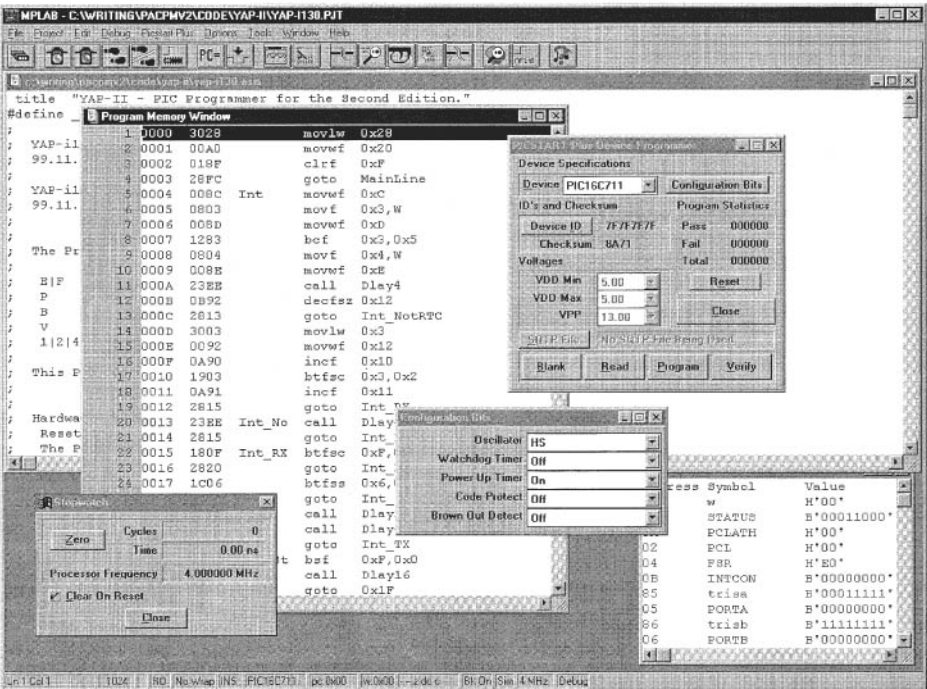


Figure 4.15 The Microchip PICSTART Plus control dialog boxes.

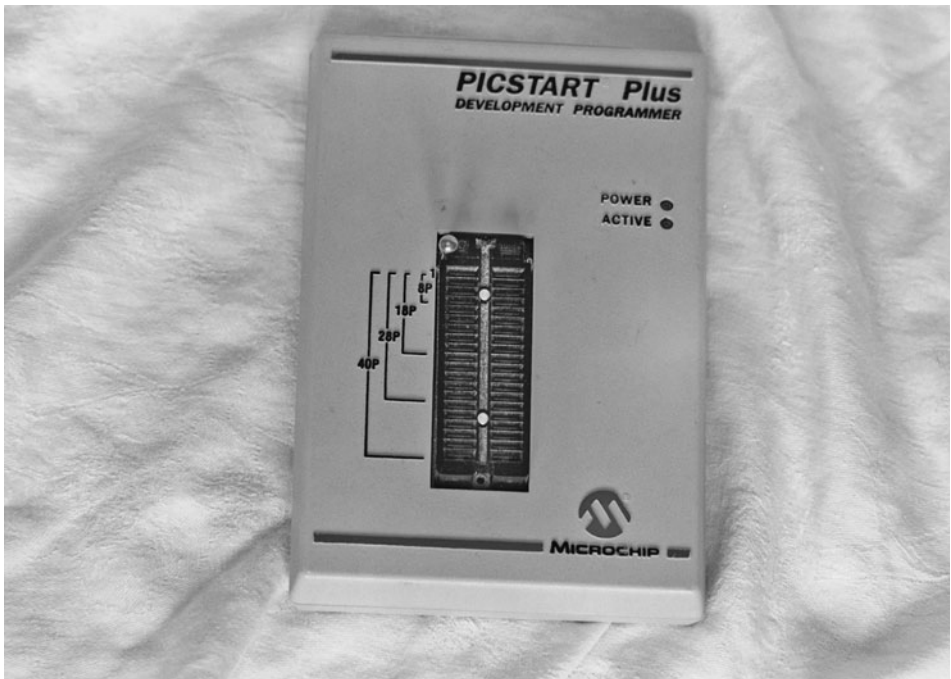


Figure 4.16 The Microchip PICSTART Plus has a 40-pin ZIF socket and is integrated with the MPLAB IDE.

THE PICSTART PLUS

The basic Microchip programmer is PICSTART Plus (Fig. 4.16) and I have owned and used one since they first came out. The PICSTART Plus, which can be referred to as PSP or PS+, is a development programmer that connects to a PC via an RS-232 cable. The programmer itself consists of a small box with a ZIF (zero insertion force) socket for programming all the various DIP PIC microcontrollers and can have its firmware updated when new parts with new programming algorithms come out. While it is still an excellent tool, I believe that there are better programming options available from Microchip that allow programming of SMT package PIC microcontrollers as well as chips that have been soldered into a circuit and need to be reprogrammed.

With the PICSTART Plus, there are a few things that you should be aware of. The first is that, as designed, it will only program DIP parts. This isn't a problem for hobbyists and PTH prototypes, but it can be a problem for SMT parts. An ICSP cable could be created from a PTH pinned socket, which would be inserted into the PICSTART Plus's ZIF socket along with some wire and a six-pin header, which would solve this dilemma. The header could either be attached to an SMT socket adapter or into a connector designed in circuit. The only problem is that the pinout of the ICSP signals on the PICSTART Plus's ZIF socket could change with the part.

Another issue you should be aware of, especially if you are buying a used PICSTART Plus, is firmware revisions. The early programmers had a PIC17C44, which had to be erased (using ultraviolet light) and then programmed with new firmware periodically. This operation was accomplished using a PIC17C44 with the old firmware inside the programmer, programming an erased PIC17C44, and then the two were swapped. The PIC17C44 was changed to a PIC18 Flash-based device that doesn't need to be erased outside of the programmer, but there were some programmers that were too old to take the Flash-based part and are now obsolete.

The PICSTART Plus is a development programmer and as such does not check the contents of a programmed part at low voltage for prototyping operations. The full programming algorithm, as specified by Microchip, includes a verification step at 4.5V. The PICSTART Plus, with its V_{dd} at a nominal 5V, cannot provide this function. Microchip will not consider a PIC microcontroller to be production programmed by the PICSTART Plus and will not respond to field problems with chips that are having problems, if the PICSTART Plus is used to burn the programs onto them. The 5V-only programming will be a problem for some newer parts that are not designed to work with V_{dd} above 3.6V, and applying 5V will damage them.

The PICSTART Plus package consists of the PICSTART Plus, a power supply, an RS-232 nine-pin "straight through" with male to female cable connectors. A sample 16F84 and CD-ROM containing data sheets, MPLAB, and applications notes are also included to help you get started.

MPLAB PM3 UNIVERSAL DEVICE PROGRAMMER

If you require a production level PIC microcontroller programmer, or need the ability to program a surface mount device, then you should look at the MPLAB PM3 Universal Device Programmer from Microchip (Fig. 4.17). This product is the follow-up to the Promate II and can work with the earlier programmer's adapter modules if you have already invested in this programmer. The MPLAB PM3 is much more flexible than the PICSTART Plus and offers the following additional features:

- Executes from MPLAB IDE or the MS-DOS command line
- Implements complete programming specification for all PIC microcontrollers
- Interfaces to the PC through RS-232 or USB
- Provides production low voltage verify
- Has a very fast programming time
- Has three operating modes:
 - PC Host mode with MPLAB IDE control
 - Safe mode for secure data
 - Stand-alone mode
- Interchangeable sockets for PTM, SMT, and ICSP cabling as well as an interface for Promate II sockets
- Also programs Microchip serial EPROMS
- Can serialize parts

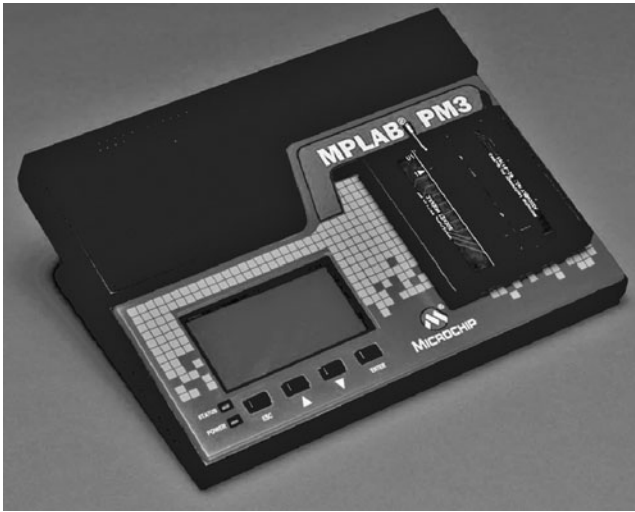


Figure 4.17 The MPLAB PM3 Universal Device Programmer can be used for all Microchip microcontrollers regardless of package type and operating voltages.

- SD/MMC sockets for storing hex data files
- Loud audible alarm for noisy manufacturing environments

These additional capabilities come at a price, however; the MPLAB PM3 is about \$1,000 (USD), which may make it less attractive for hobbyists or companies that want to see what the PIC microcontroller is all about before making substantial investments.

My Programmers

I must confess that for many years, I had the desire to come up with the perfect “universal” hobbyist programmer, and I have created a number of programmers, which actually worked quite well for specific PIC microcontroller part numbers and specific PC hosts. When these programmers were first developed, there were fewer PIC microcontroller part numbers available, with a small fraction of them being EEPROM or Flash-based and able to be electrically reprogrammed. There are several hundred PIC microcontroller part numbers available today with many subtle variations on the ICSP programming algorithms existing, which means that each device has to have its programming algorithm and parameters specified uniquely and not part of a “class” of parts. Similarly, PCs were much less sophisticated than they are now, with parallel ports consisting of the same interface circuitry designed into the PC and having operating systems that allowed applications to read and write I/O ports. Today’s PCs are much more complex and have substantial protections built in to prevent errant and malicious applications from overwriting data and hardware registers. I believe that the goal of a “universal” hobbyist programmer for PIC microcontrollers is really not

attainable because of the complexity of modern PC hardware and the plethora of programming options possible in PIC microcontrollers.

In the next two sections, I present a couple of the programmer projects that I have embarked upon. I'm including them here because I think there are some useful functions that you may want to incorporate in your own designs and because I have a hope that one day somebody will come up with a perfect hobbyist programmer that can be built cheaply and easily, allowing hobbyists easy access to the PIC microcontroller without making a substantial investment in money or time.

THE YAP-II

When I wrote the first edition of this book, I ended up spending an unreasonable amount of time trying to come up with a programmer for it. The goal was to create a PIC16F84 programmer that would work with virtually all PCs, with a simple programming interface. As I was working through the book, I came up with three different programmers, using both the PC's parallel port and serial port, each one with some strengths and weaknesses. Usually the programmers were very inexpensive, but none of them was able to run on a reasonably wide variety of PCs. My final solution to the problem was the YAP (Yet Another Programmer). This programmer used a PIC16C61 with an RS-232 interface that took a downloaded hex file and programmed it into the target PIC microcontroller as the file was downloaded. The programmer worked quite well though it only runs at 1200 bps, somewhat slower than other devices out there. The reason for 1200 bps was to make sure there would be enough time for the worst case programming of EPROM parts. Fig. 4.18 shows the assembled YAP-II programmer.

The slowness of the operation was to ensure that data did not come in faster than the PIC MCU could program into the target device. The YAP approached the problem from the perspective of using the PC's RS-232 ports as the basic interface not only to the programmer but for programming timing as well. This had three advantages over the other methods tried. The first was the use of an I/O port with standard timings—while the communication voltages would have to be translated from RS-232 protocol to CMOS/TTL, the incoming

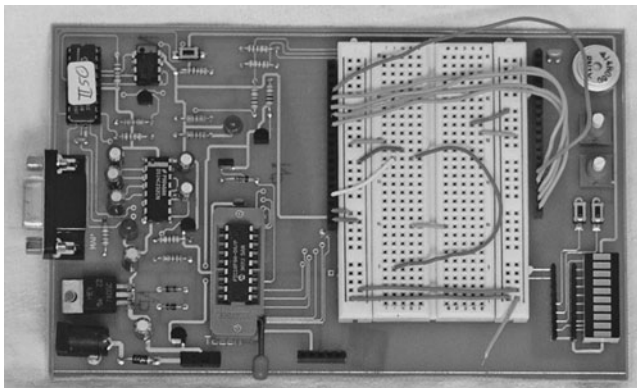


Figure 4.18 The assembled YAP-II programmer.

data rate could be used to time data going into the PIC microcontroller. The second advantage was the ability of PIC microcontroller applications to interface to standard ASCII terminal emulators and not require custom PC software that would have to be debugged in parallel with the PIC microcontroller application. Lastly, many people who wanted to learn about the PIC microcontroller but were not running an MS-DOS or Microsoft Windows PC could run the YAP on their hardware to develop their own applications.

Once I had this concept, I created the following specifications for the YAP programmer:

- Able to program all mid-range parts (EPROM and Flash-based program memory)
- Able to take the programming signals and use them in ICSP applications
- Allow any RS-232 equipped host PC or workstation to program PIC microcontrollers
- Allow serial communications between the host PC or workstation and the executing application for debugging applications on the fly

The result was the YAP, which really wasn't a bad design, but fell short in a number of areas. These included problems with the reset circuit that made programming EPROM parts unreliable, selecting parts that were difficult for people to find or expensive, and creating a form factor that made building sample applications more difficult than it should have been. It did have some positive points, however, in terms of its ease of use and reliability for different PCs and workstations. I also created a Visual Basic interface, which makes using the YAP much easier than running it from a basic terminal emulator. Once the problem with the line-ending characters was resolved, the programmer itself was downloaded and built by a number of people, and Wirz Electronics has sold a large number of built and tested units with very few complaints.

Taking this base, the programmer was enhanced into the YAP-II with the following features:

- More reliable programming for EPROM parts
- Ability to program PIC12C5xx and PIC16C505 PIC microcontrollers as well as 28- and 40-pin parts
- Eliminates some of the difficult-to-find/expensive parts
- Provides a better form factor for hobbyists and people learning the PIC microcontroller

The YAP-II consists of a simplified circuit and the actual PCB has been laid out to include a built-in breadboard and a set of sample devices that you can interface to a PIC microcontroller in order to test out applications simply.

The YAP-II really is a PIC microcontroller application, in which the PIC microcontroller communicates with a host system via RS-232 and provides some interesting interfaces to other devices (including a second PIC microcontroller with a synchronous serial interface and a high-voltage, moderate—up to 50 mA—current control). The schematic for the YAP-II is shown in Fig. 4.19 and is the basic application circuit. Attached to it on the PCB that I have designed for it is a set of I/O accessories, which are shown in Fig. 4.20.

The parts for building the YAP-II are quite straightforward and are listed in the bill of materials given in Table 4.9. As this book is written, the PIC16C711 is available for sale

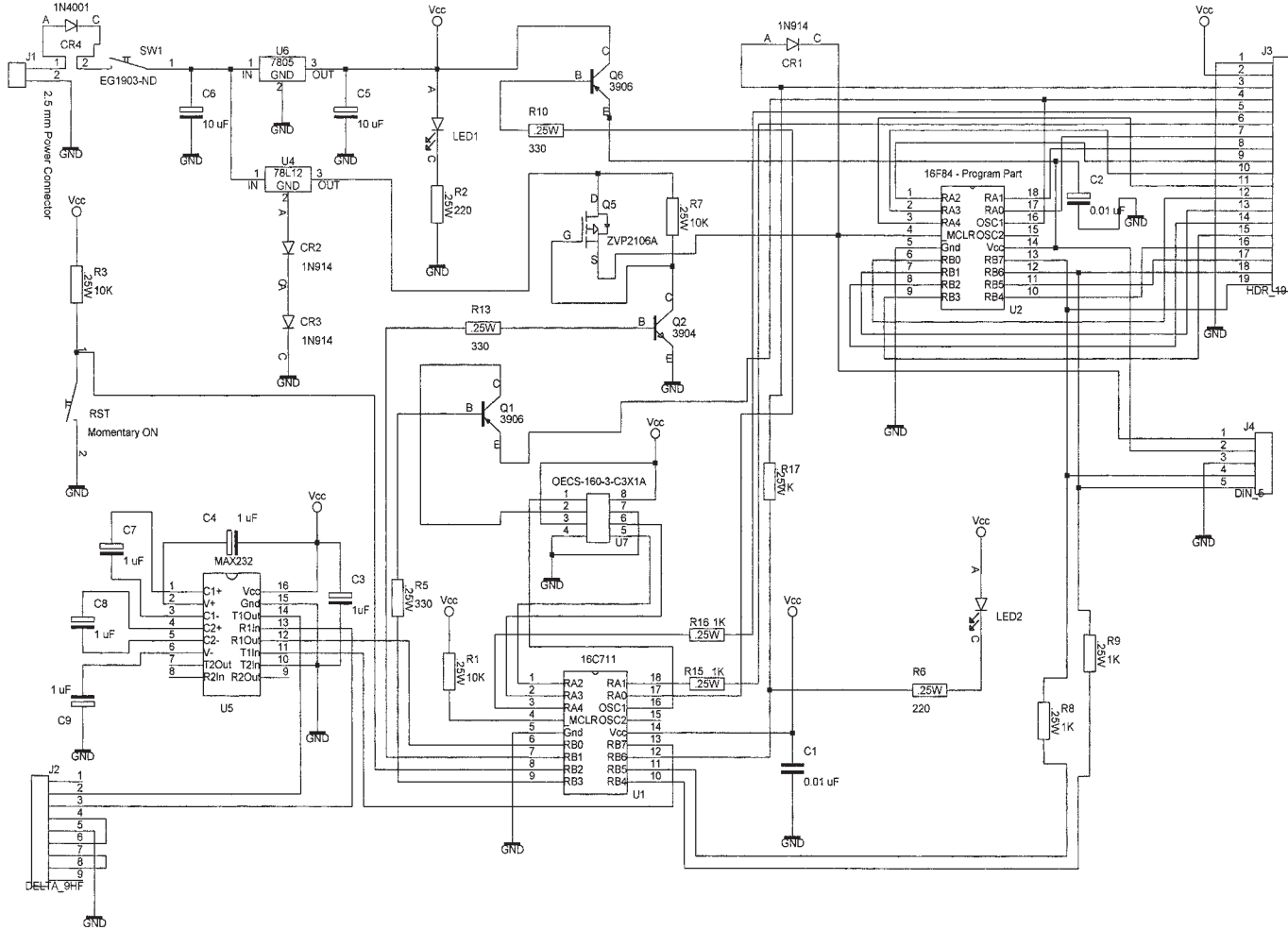


Figure 4.19 First page of the YAP-II schematic.

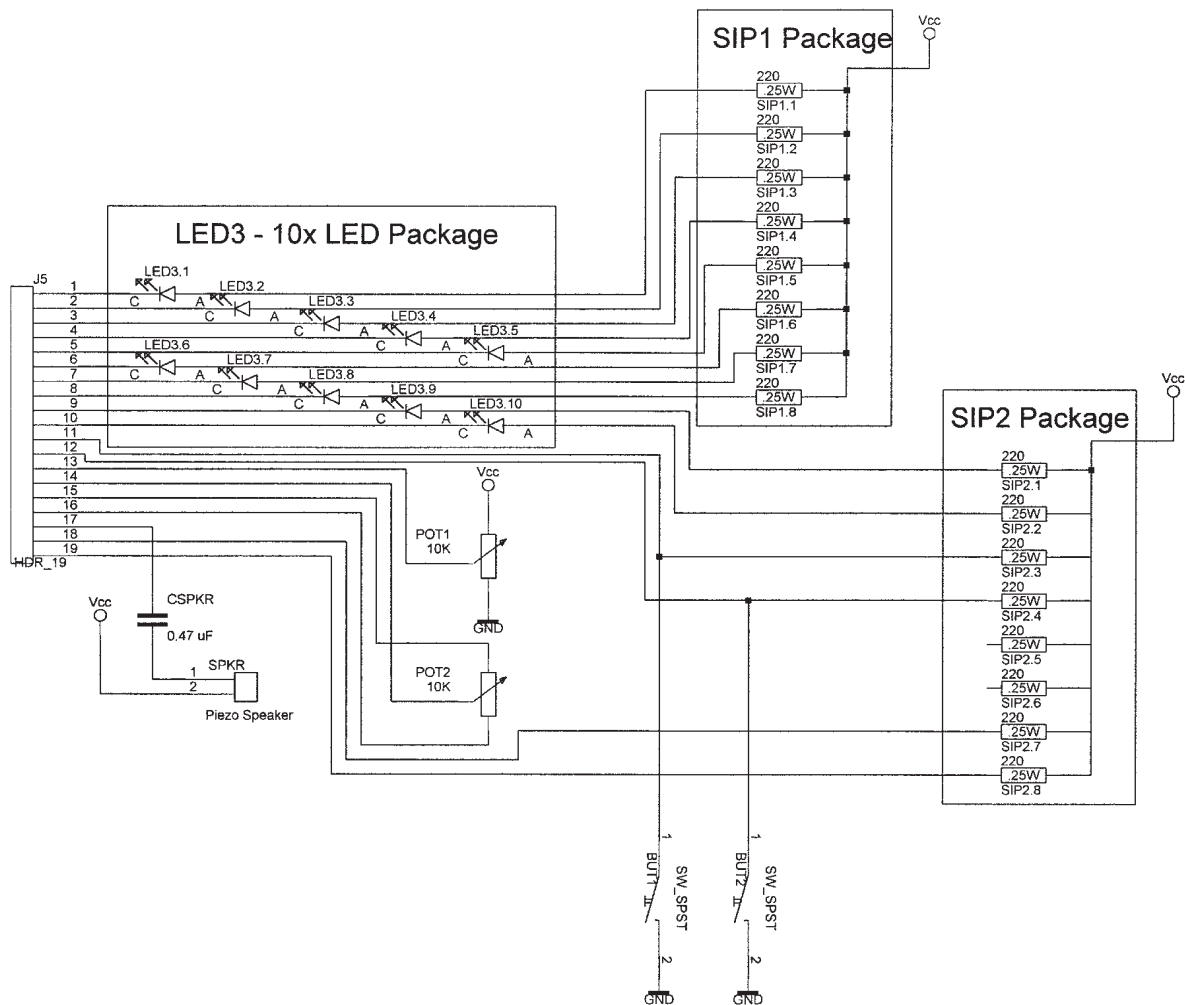
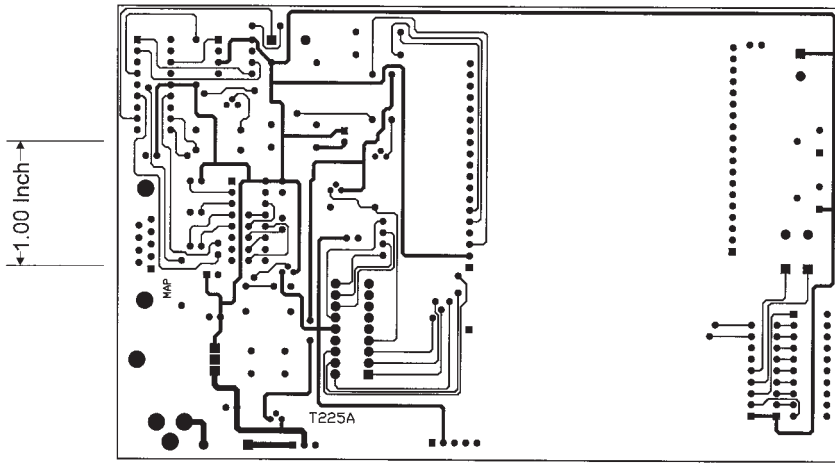


Figure 4.20 Second page of the YAP-II schematic, showing the available accessories.

TABLE 4.9 YAP-II BILL OF MATERIALS

REFERENCE DESIGNATOR	PART NUMBER
U1	PIC16C711-20/P preprogrammed with YAP-II software
U2	18-pin socket/ZIF socket
U4	78L12
U5	MAX232
U6	7805
U7	ECS programmable oscillator—part number ECS-160-3-C3X1A
CR1-CR3	1N914 silicon diode
CR4	1N4001 silicon diode
LED1-LED2	5mm red LED with 0.100in lead spacing
LED3	10x red LED bar graph display
Q1, Q6	2N3906 PNP bipolar transistor
Q2	2N3904 NPN bipolar transistor
Q6	2106A P-channel MOSFET
R1, R3, R7	10K, 1/4 watt
R2, R6	220 Ω , 1/4 watt
R5, R10, R13	330 Ω , 1/4 watt
R8, R9, R15-R17	1K, 1/4 watt
POT1-POT2	10K, single turn PCB mount POT
SIP1-SIP2	220 Ω x9 common pin SIP
C1-C2	0.01 μ F, any type
C3-C4, C7-C9	1 μ F, any type
C5-C6	10 μ F electrolytic
CSPKR	0.47 tantalum
SPKR	Piezo speaker
J1	SPDT PCB mount switch
J2	9-pin female PCB mount D-shell
J3, J5	19x1 PCB mount socket strip
J4	5x1 PCB mount socket strip
RST, BUT1-BUT2	Momentary on PCB mount switch
Misc.	PCB board, serial cable, power supply



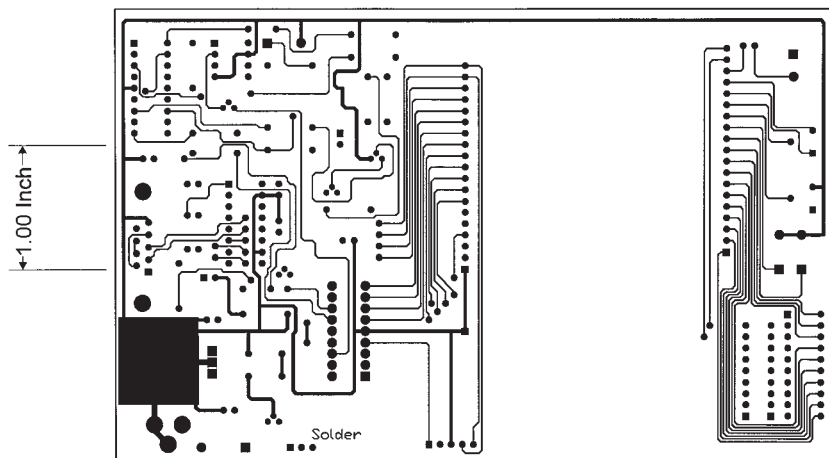
T225A Top Layer

Figure 4.21 YAP-II top layer design.

as both a ceramic windowed part and as an all-plastic one-time programmable (OTP) device. The only part that you may have difficulty getting is the 2106A P-channel MOSFET. This device can be substituted for other P-channel MOSFETs—its critical parameters are its I_d (On maximum current) of 280 mA and low internal resistance (R_{ds}) of 5Ω .

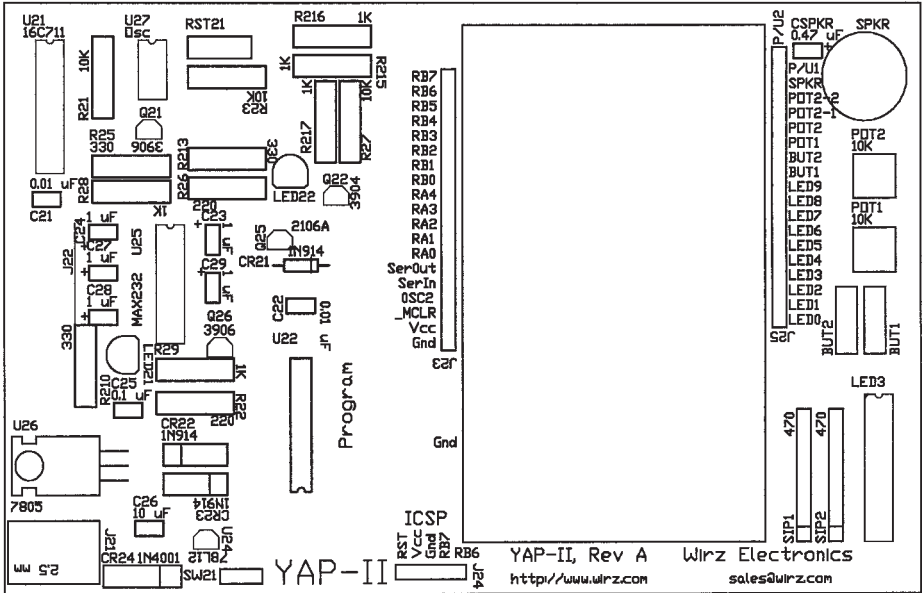
The source code for the YAP-II (`yap-ii50.asm`) can be found in the `PICD\wnld\YAP-II` folder. The operation of this code is described next.

The PCB designed for this circuit is a two-layer board; the top and bottom layers are shown in Fig. 4.21 and Fig. 4.22, respectively. The silkscreen overlay information is shown in Fig. 4.23.



T225A Bottom Layer

Figure 4.22 YAP-II bottom layer design.



T226A □ Overlay

Figure 4.23 **YAP-II silkscreen overlay.**

Note that in the overlay layer, the part number references have a 2 added to them (for example, R8 in Fig. 4.19 is R28 in Fig. 4.23). This change is due to my placing multiple PCB images on one card and the PCB design system not having the capabilities to allow multiple parts with the same part number onto the PCB.

The basic circuit of the YAP-II is a PIC16C711 running at 16 MHz (from the dual-output programmable oscillator) communicating to a host system via an RS-232 interface. As I will discuss elsewhere, I hate making up my own cables, so the circuit is designed to be used with a standard straight-through cable. While in my circuit I have used a 9-pin D-shell, you can use whatever method of connections you are most comfortable with. The RS-232 interface is essentially a three-wire RS-232 connection. The RS-232 interface application code executing in the PIC microcontroller uses TMR0 to provide an interrupt at three times the incoming data rate. The RS-232 interface code is designed to buffer the incoming serial data and indicate when the current byte being sent has completed. This interface is used to allow programming operations to take place in the foreground while serial I/O is taking place in the background. When data is being programmed, a new programming operation is initiated every four instructions.

The power supply is quite straightforward with a 7805 providing up to 1A of current at 5V, and a 78L12 and two 1N914 diodes providing 13.4V at up to 100 mA. In the power supply circuit, note that I have included a 1N4001 diode to make sure negative voltages cannot damage the circuit. For the power source, use a wall-mounted

AC/DC converter with an output of at least 14V and 500 mA. Wall power adapters with these specifications can usually be bought from discount stores for as little as two dollars.

The programming interface circuit consists of three transistors, one diode, and seven resistors. Transistor Q6 (along with R10) provides a switched power supply to U2, or the part to be programmed. Transistors Q2 and Q5 (along with R7 and R13) provide a control to the 13.4V power supply to the Vpp pin. The reset circuit is also driven by U1's RB6 pin that provides reset voltages for when the programmed part is run in a circuit.

Resistors R8 and R9 provide the data and clock interfaces to the part being programmed. The resistors are used to provide protection to U1's pins. To initiate a programming operation, the U1 pins on R8 and R9 (RB5 and RB4, respectively) are pulled low and then +13.4V are applied to the PIC microcontroller in the socket at U2 or connected to the ICSP port at J4. Once the programming voltage has stabilized, instructions are sent to the PIC microcontroller being programmed.

The programmed parts reset can also be controlled by U1's RB6 for allowing the PIC microcontroller in the socket to execute. When the PIC microcontroller in the U2 socket is to execute, U1's RB3 is pulled low, turning on the gate to the programmable oscillator.

The U2 socket is designed to provide a method of programming the PIC microcontroller and allowing it to execute freely (clocked by the programmable clock, U7). The U2 socket itself is connected to a 19-pin interface (J3) that can be connected to circuits on the breadboard attached to the YAP-II PCB or to the second 19-pin socket, which provides the built-in accessory interface.

The PIC microcontroller 19-pin interface is defined in Table 4.10.

Note that pins 18 and 19 are directly connected to U2 and are used for programming the PIC microcontroller in U2 from U1. There are 1K resistors between the U2 and U1 pins, but there should never be an active driver on J3's pin 18 and 19 when you are trying to program the part in the U2 socket. If there is an active driver (and this can be an LED on a pull-up), then U1 will be unable to overpower it because of the 1K resistors on the ICSP clock and data lines.

Along with the circuit necessary to program the PIC microcontroller in the U2 socket, I have also included an ICSP compatible connector at J4. This connector is defined in Table 4.11.

This J4 connector can be used with J3 to program PIC microcontrollers that are different from 18 pins. A 28-pin device could be programmed by wiring it into the YAP-II's breadboard and providing Vpp from the ICSP connector.

Along with the PIC microcontroller interface, I have also included a set of accessories to allow new users to try out new applications very quickly, without having to find parts and figure out how to wire them in. As you will see, these features greatly simplify the wiring of the experiments.

J5 is a 19-pin connector, like J3, and provides an interface to LEDs, buttons, potentiometers, a speaker, and some pull-ups. Ten LEDs are built into a bar graph display, which is soldered into the board. These LEDs are pulled up by 220 Ω resistors and to turn them on, they have to be pulled to ground. Two pulled-up buttons are also available along with a potentiometer that acts like a voltage divider. The second potentiometer has all three

TABLE 4.10 YAP-II 19-PIN INTERFACE			
PIN	FUNCTION	U1 CONNECTION	U2 CONNECTION
1	Gnd		
2	Vcc		
3	_Reset	RB6/1K resistor	No direct connect
4	YAP-II oscillator		
5	U1 serial in	RA4/1K resistor	
6	U1 serial out	RA1/1K resistor	
7	U2 RA0		RA0
8	U2 RA1		RA1
9	U2 RA2		RA2
10	U2 RA3		RA3
11	U2 RA4		RA4
12	U2 RB0		RB0
13	U2 RB1		RB1
14	U2 RB2		RB2
15	U2 RB3		RB3
16	U2 RB4		RB4
17	U2 RB5		RB5
18	U2 RB6—programming clock	RB4/R9	RB6
19	U2 RB7—programming data	RB5/R8	RB7

connections passed to the J5 connector, so different circuits can be built with it. The piezo speaker is connected to J5 through a 0.47 uF capacitor so that the driver is isolated from the speaker (and any transients coming from it). Finally, there are two pull-ups for convenience’s sake. The pinout of J5 is listed in Table 4.12.

TABLE 4.11 YAP-II ICSP CONNECTOR PIN DEFINITION	
PIN	FUNCTION
1	Vpp—Connected to PIC MCU MCLR# pin
2	Vdd
3	Vss
4	ICSP data
5	ICSP clock

TABLE 4.12 YAP-II ACCESSORY CONNECTOR PIN DEFINITION

PIN	FUNCTION
1	LED1
2	LED2
3	LED3
4	LED4
5	LED5
6	LED6
7	LED7
8	LED8
9	LED9
10	LED10
11	BUT1
12	BUT2
13	POT1
14	POT2 Wiper
15	POT2—Connection 1
16	POT2—Connection 2
17	Speaker
18	Pull-Up 1
19	Pull-Up 2

The original YAP was well designed for the PIC16F84 and PIC16Cx(x)1 part numbers, but not very many others. The YAP-II is designed for a wider range of PIC microcontrollers with varying program memory sizes.

In the YAP-II, I've further simplified the command set so that a single character is sent as a command, followed by a carriage return. The 13 commands are listed in Table 4.13.

The interface itself is designed to run at 1200 bps. This speed was chosen as the fastest standard speed for the time it takes to receive 4 bytes (from a hex file) giving an instruction for programming and perform a programming operation in parallel. Running the interface at 1200 bps makes the YAP somewhat slower than other PIC microcontroller programmers, but the alternative would be to add an external buffer memory, which would add to the cost of the device.

When demonstrating how the commands work, I have provided screen shots of HyperTerminal operating with the data shown on the display. HyperTerminal operation

TABLE 4.13 YAP-II COMMANDS	
COMMAND	OPERATION
A	Ping—Return nothing but carriage return/line feed
B	Reset the program counter to 0
C	Clear the contents of Flash memory
D	Dump 256 instructions increment “Read” program counter by 256
E	EPROM part programming—“Text Send” hex file
F	Flash part programming—“Text Send” hex file
G	Get 8 instructions starting at address 0x02000
1, 2, 4, 8	Run PIC microcontroller at the specified speed

is explained elsewhere, but to connect to the YAP-II, HyperTerminal should be set up with a direct connect to the YAP-II at 1200 bps with an 8-N-1 data format.

The interface will convert lowercase ASCII to uppercase and ignores all characters except for the ones listed in the table above and ASCII Backspace (0x008) and Enter (0x00D).

The ping command is designed for advanced interfaces (like the Visual Basic interface that is presented below) to check to see if the YAP-II is connected and working properly. After sending ASCII A (0x041), followed by an ASCII Enter (0x00D), the YAP-II returns a carriage return/line feed string. This instruction is simply used for checking the interface without having to parse the

`"<== Invalid"`

message that is returned for invalid commands (everything other than the 13 commands listed in Table 4.13).

During programming, the PIC microcontroller uses its built-in program counter for keeping track of where operations are taking place. This program counter is “shadowed” within the YAP-II to keep track of where it is executing. I use the shadowed program counter to keep track of the offset of the 256 instructions last returned by the YAP-II during the Dump instruction. To reset it, the B command is used.

The C command clears the contents of Flash program memory using the Microchip specified “All Clear” instruction. This is the same process as was discussed above.

- 1 Apply Vpp.
- 2 Execute load configuration (0b0000000 + 0x07FFE).
- 3 Increment the PC to the configuration register word (send 0b0000110 seven times).

designed to only work with PIC microcontrollers that had 1,024 instructions of program memory space. By downloading data 256 instructions at a time, different program memory sizes can be supported as well as the PIC12C5xx and PIC16C505, which are low-end devices that use the same programming protocol as the mid-range devices but program the configuration fuses as the first byte.

Programming the PIC microcontroller is accomplished by downloading the MPASM produced hex file into the YAP-II after specifying either the **E** (EPROM) or **F** (Flash) command. The YAP-II will decode the hex file format and check that the results of the programming operation are correct for each instruction. When the file has finished being transmitted, if there was an error in the programming, the first instance of problems will be returned with the expected (E) and actual (A) values displayed and the address where the error occurred.

The programming operations wait for the data file for one minute before timing out. If you don't want to go through with the programming operation, then sending a 0x003 (Ctrl-C) character to the YAP-II can stop it.

The programming operations are identical to what is described above except that there is no final verify operation (other than using the dump command). When each line of the hex file comes in, the address to program is compared against the current value of the shadowed program counter. If there is a difference between the two values, the PIC microcontroller's program counter is incremented while the next 6 bytes are coming in. These 6 bytes take 50 ms to come in, which is just enough to increment the PIC microcontroller's program counter 50 times. If the difference between the current PIC microcontroller program counter and the specified address is greater than 50, then a jump error will be flagged by the YAP-II and programming will stop. Generally, this is not a major concern as most applications are written with instruction addresses written consecutively and no space left in between the modules in the application.

When the configuration fuses are programmed (at address 0x02007), the "**Load Configuration**" command will be used, which changes the internal program counter to 0x02000 and avoids the need to repetitively increment the PIC microcontroller's program counter.

This is not true for data EEPROM initialization. This data, which is located at address 0x02100 of the hex file, requires a different programming algorithm and cannot be accessed. If the **de** directive is used in your source file, the application will return a jump error.

The configuration memory (at 0x02007) is read for verify by using the **G** command. The 8 bytes starting at 0x02000 are dumped onto the screen (in the same format as the **D** command), showing you not only the configuration fuses, but the ID locations as well.

The last four commands specify the operating speed for the PIC microcontroller. On the card is an ECS 16 MHz dual output programmable oscillator. The primary, 16 MHz output is passed to the PIC16C711, which controls the operation of the YAP-II. The secondary output, which has a built-in programmable divisor, is passed to the part in the programming socket when one of these commands is executed. When 1, 2, 4, or 8 is sent to the YAP-II from the controlling PC, the PIC microcontroller in the programming

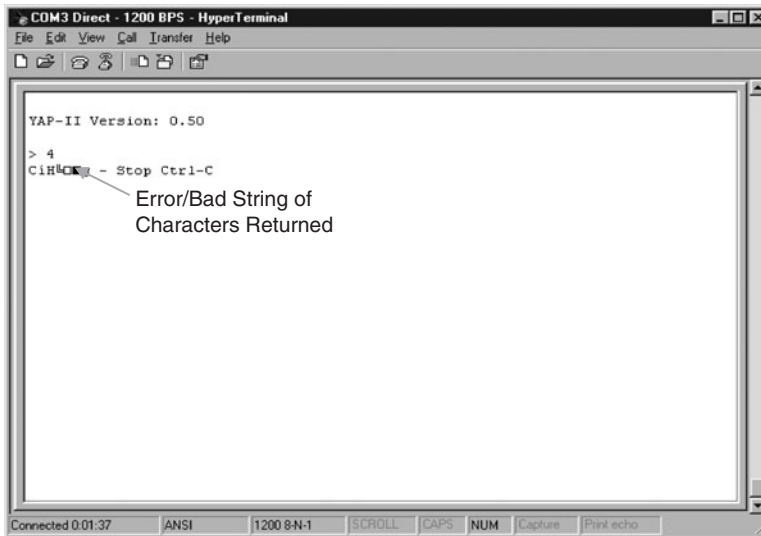


Figure 4.25 YAP-II stopped, showing string of invalid characters.

socket has power applied to it along with the programmable oscillator output, and finally, the MCLR# pin goes high. During execution, serial data can be passed to and from the PIC microcontroller with the application stopped when a Ctrl-C (0x003) character is received by the YAP-II.

When the Ctrl-C character is received, you will see a funny string of characters overwriting the “Running” message as is shown in Fig. 4.25. This is caused by invalid serial line data being inadvertently sent by the serial pass-through option (described in the previous paragraph) when the programmed PIC microcontroller’s `_MCLR` line becomes active.

This is a by-product of how the PIC16C711 YAP-II controller operates and there is no way to prevent it except to eliminate the pass-through option. When the Ctrl-C is received, the serial interface changes mode with the value of the `SerIn` line being passed to the host without a full byte being sent. I have chosen to leave this in because I normally use the YAP-II with a Visual Basic front end and I can mask the invalid characters there.

The PIC16C711 software used in the YAP-II is quite complex and though it is based on the original YAP application, there have been substantial changes to the operation of the code. Also in the YAP-II application code, because I was using a PIC16C711 (which has a lot more file registers than the YAP’s PIC16C61), I was not as restricted in the number of variables that I used in the application. This makes the code somewhat easier to read and follow.

The Visual Basic front end (see Fig. 4.26) provides a graphical interface for you to operate the YAP-II. There is an installation package in the `PICDwnLd\code\YAP-II\YAP-II VB Package` folder.

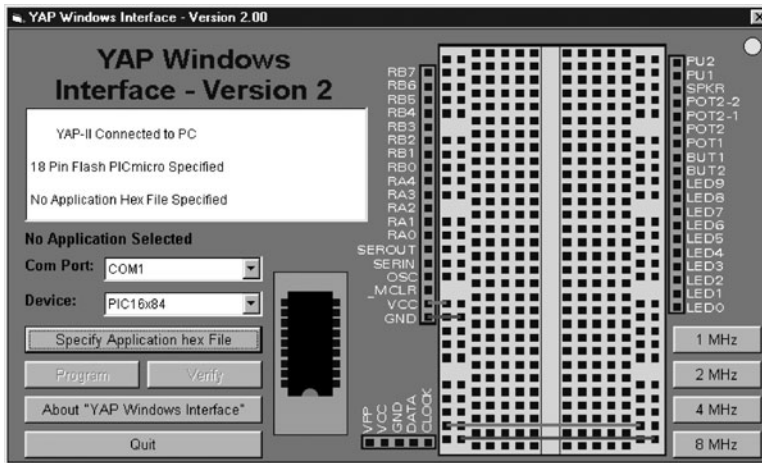


Figure 4.26 The YAP-II Visual Basic windows interface.

The interface port (COM1, COM2, or COM3) can be selected from the front window, as well as the PIC microcontroller to work with and the hex file to program into it. The Visual Basic front end continually pings for a YAP-II on the specified interface port and indicates when one is available. To simplify the Visual Basic front end, the primary parameters, COM port, hex file, and PIC microcontroller target are saved on your PC's hard disk so that when you start up again, you do not have to re-enter these parameters.

When the application is executing, a separate terminal emulator display will allow you to send and receive executing data from your application.

When you are working with the YAP-II, I recommend that you initially wire the two top and bottom rails with Vcc and ground as I've shown in the Windows interface (Fig. 4.26). This will give you convenient power connections as you build a test application circuit.

THE EL CHEAPO

Another programmer project I spent a considerable amount of time on was a simple PC parallel port programmer designed to work on a variety of PCs and for both Flash as well as EPROM program memory based PIC microcontrollers. The result of four redesigns was the El Cheapo, which, at the time of the second edition, could have been used in virtually any PC available at that time. Fig. 4.27 shows an assembled El Cheapo. The programmer allowed people to program a variety of ICSP PIC microcontrollers with the exception of the PIC17Cxx and PIC18. The project ultimately collapsed due to the weight of supporting all the different PIC microcontroller part numbers (I gave up when there were more than 175 low-end and mid-range ICSP programmable PIC microcontrollers) and the change in PC hardware from parallel ports that consisted of processor accessible registers to IEEE 1284 compatible ports that no longer provided the basic

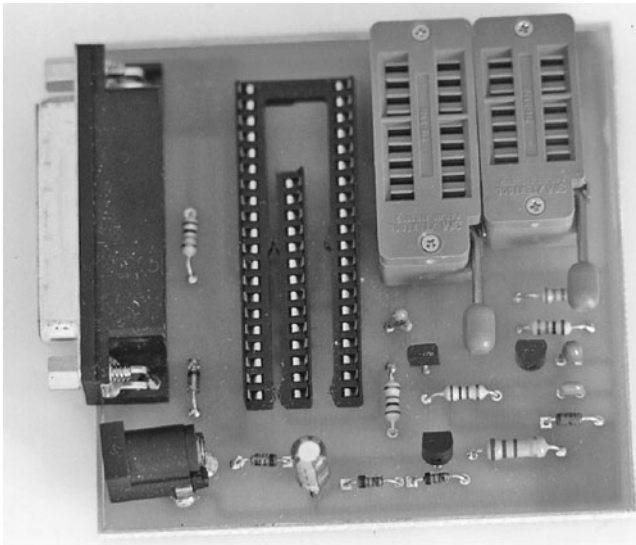


Figure 4.27 Photograph of an assembled El Cheapo.

register and bit access. The programmer is an interesting example of how simply a PIC microcontroller programmer circuit can be designed and implemented.

To give you an idea of how PC systems have changed over the years, consider this paragraph from the second edition (written in 1999):

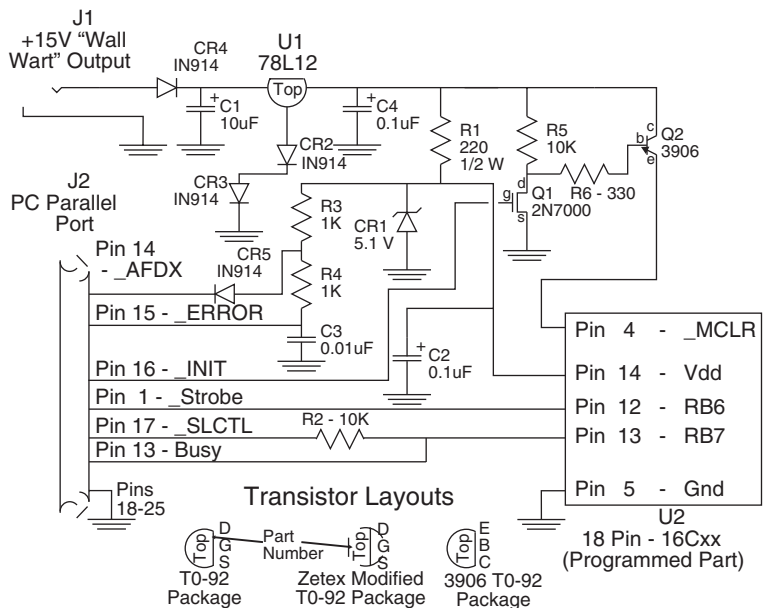
The programmer's speed should not be considered a major consideration because, depending on how the programmer is built, it may work very quickly with one host PC and not at all in another. This is why I went through several revisions of the El Cheapo and YAP-II to make sure that the programmers were as device independent as possible. As I write this, the first 800 MHz PCs are becoming available with 1 GHz PCs not far away. Coupled with many sub-\$500 PCs with Celeron processors running under 400 MHz, the performance range on modern systems is staggering . . .

My current system is a 2.8 GHz dual core Pentium with my printer on my home's local area network on another PC and connected to it via USB. Modern PCs are simply not systems that can be used for connecting simple programmers like the El Cheapo.

This circuit was designed to handle 8-, 14-, 18-, 28-, and 40-pin low-end and mid-range PIC microcontrollers that can be programmed using the two-wire ICSP synchronous data protocol that I discussed previously in this chapter. The basic circuit design is shown in Fig. 4.28, which shows how an 18-pin PIC microcontroller is wired into the circuit.

The bill of materials for the programmer is listed in Table 4.14. Note that for the 2.5 mm power connector (J1), I have included a Digi-Key part number to make it easier for you to find.

The circuit can be broken up into four major subsystems. The first is the power supply. The input DC power is passed through the 78L12 to provide the programming voltage (V_{pp}) for the PIC microcontrollers. This voltage must be at least 13V. To shift



the voltage output of the 78L12 from 12V, I have added two silicon diodes to the ground reference of the 78L12. Each silicon diode will bump up the ground reference by 0.7V, the two of them raising the ground reference to 1.4V and with it, the 78L12's output to 13.4V above the El Cheapo's ground reference.

To provide the 5V Vdd required by the PIC microcontroller, I used a Zener diode and resistor to regulate down the 13.4V to 5.1V. When I checked the programming specifications of the various PIC microcontroller devices that I wanted this programmer to handle, I found that the maximum Vdd during programming was 40 mA. This value determined what resistor was going to be used in the circuit. For a voltage drop of 8.3V when 40 mA is used, a 207 Ω resistor would be best. I used a 220 Ω resistor simply because I have a lot of them around. The 8.3V drop at 40 mA dissipates 0.33W of power, requiring a 0.5W resistor. If there is no load on the 5.1V power supply, these 40 mA are passed through the Zener diode. Doing the calculation again, the total power dissipated by the Zener diode is 0.24W. To be on the safe side, this diode should be rated for at least half a watt along with the resistor.

The 5V power supply is designed so that if it is shorted out, only 40 mA will be supplied by it. This allows the PIC microcontroller to be put and pulled out without switching or disconnecting the power supply. The power supply circuit makes some people uncomfortable, but I ask that you do not change or modify this circuit as it works very well and will protect the circuit, the PIC microcontroller being programmed as well as your PC against anything bad happening. I know of people who have used bench supplies

TABLE 4.14 EL CHEAPO BILL OF MATERIALS

PART	DESCRIPTION	COMMENTS
U1	78L12	
Q1	2N7000 N-Channel MOSFET	Note Zetex Modified TO-92 package output in Fig. 4.28
Q2	2N3906	Labeled 3906 in Fig. 4.28
CR1	5.1V, 0.5W Zener	
CR2-CR5	1N914 silicon diodes	Any silicon diode can be used in this circuit
C1	10 uF electrolytic	16+V
C2, C4	0.1 uF	Any type of capacitor can be used
C3	0.01 uF	Any type of capacitor can be used
R1	220 Ω , 0.5W	
R2, R5	10K, 0.25W	
R3-R4	1K, 0.25W	
R6	330 Ω , 0.25W	
J1	2.5 mm power socket	Digi-Key part number: SX1152-ND
J2	DB25-F PCB mount socket	
U2	18-pin DIP socket	Can be a ZIF socket—Use 3M Textool P/N 218-3341-00—0602R
U3	14-pin DIP socket	Can be a ZIF socket—Use 3M Textool P/N 214-3341-00—0602R
P28	14-pin SIP socket	Can be cut down from a 28-pin DIP socket
P40	40-pin DIP socket	See text regarding socketing options
PCB		
Power	14+V AC/DC	The power supply must source at least 250 mA; output must match J1
PC interface cable DB-25M to DB-25M		Straight-through parallel port switch cable; see text

and one poor soul who used his PC's +5V and +12V power supplies and ended up burning them out because he didn't know what he was doing. The circuit I've shown here can take a lot of abuse, and if too much current is drawn, the 78L12 will shut down to protect you and the circuitry it is connected to.

The second subsystem in this programmer is the V_{pp} control circuit consisting of the dual transistor switch of Q1 and Q2. Q1 controls a zero to a 13.4V signal that controls the PNP transistor at Q2. This circuit probably seems a bit unwieldy and unnecessarily complex, but V_{pp} will require up to a 50 mA source to program EPROM parts. This circuit will switch the 13.4V of regulated voltage and allow the maximum current supplied by the 78L12 to be passed to the PIC microcontroller being programmed.

The RC delay circuit (consisting of R3, R4, C3, and CR5) is used to provide an external delay to the PC for programming the PIC microcontroller. The delay itself is on the order of 100 μ s. Five volts are provided by the El Cheapo's power supply and are controlled by pin 14 of the PC's parallel port. The RC network will delay the action and allow the PC to poll the error (pin 15) line of the parallel port to get a (relatively) constant delay that is independent of the PC's operating speed and internal architecture.

I found that the RC network did perform its designed function but the (relatively) long RC delay made it impractical for use in timing the programming operation of the chip. Similarly, the circuit could not be used to practically time the number of cycles the PC's processor could execute in a given amount of time due to different operating system requirements and variances in timing of the processor execution. As I gained more experience with the circuit and more people tried it out, I found that it really couldn't be used for providing a stable timing reference for the programmer and I stopped using it.

The last subsystem in the El Cheapo is the PIC microcontroller socket and programming data pins. In Fig. 4.28, I showed just an 18-pin PIC microcontroller socket. As can be seen in Fig. 4.29, there are actually three sockets, which allow the programming of five different pin-through-hole (PTH) PIC microcontroller configurations.

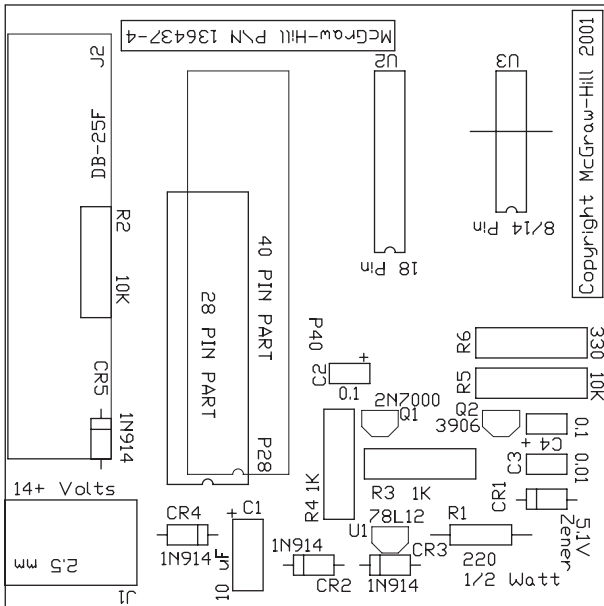


Figure 4.29

Silkscreen overlay for the El Cheapo PCB.

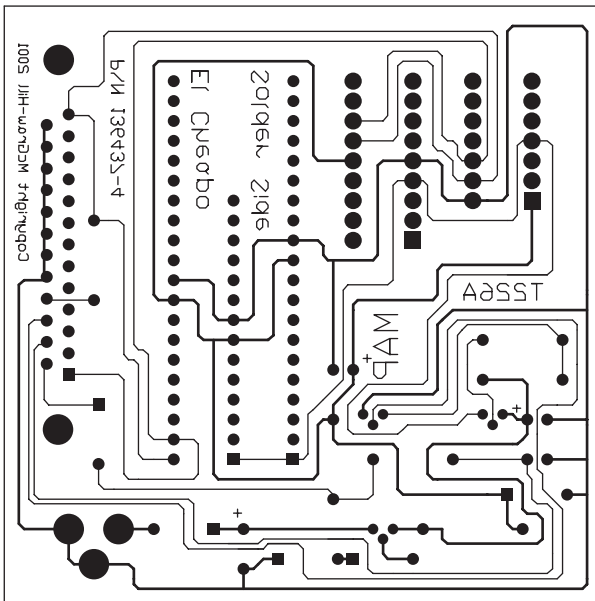


Figure 4.30 Bottom side copper pattern for the El Cheapo PCB.

The PCB design is shown in Fig. 4.29 and Fig. 4.30. Figure 4.29 is the overlay, indicating where parts are to be placed on the board and their orientation. Figure 4.30 is the bottom side copper pattern used on the board, and the lettering on it is the mirror image of what is going to be displayed, because the stencils made for the PCB are made from the top down. When you look at the actual PCB that came with the book, all the lettering on it will be readable (without using a mirror).

The circuit is quite easy to assemble on the PCB, but there are a few comments that I want to make. First off, this is a single-sided board. In order to layout the board without any jumpers on the top side, there are some traces between 0.100in parts and between the transistors' outside pins. When you are soldering in the components be careful that the traces between their pins are not shorted to the pins.

Next, there are three or five polarized components. Make sure the transistors and the 78L12 regulator are inserted properly. If you buy Zetex Modified TO-92 package transistors, remember that the flat side with the labeling should match the flat side on the PCB's overlay silkscreen.

Note that the parallel port socket is female and the cable connecting the El Cheapo to your PC is a DB-25 male to DB-25 male straight-through cable. Do not use a male socket on the El Cheapo and a DB-25 male to DB-25 female cable—this will reverse the pins going into the programmer and it will not work.

The sockets used for the 8, 14, and 18 PIC microcontroller positions can either be standard DIP sockets or ZIF (zero insertion force) sockets. The holes in the PCB are large enough to support ZIF sockets and I recommend that you use them. I realize that

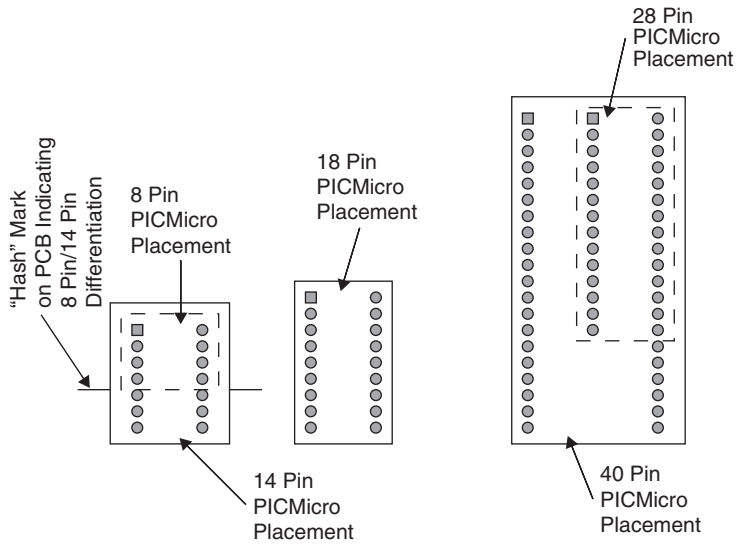


Figure 4.31 The PTH PIC microcontroller El Cheapo installation options.

a single ZIF socket will cost three or four times what the other sockets on the board cost, but they will make using the programmer a lot easier.

The 14-pin socket is used to program both 8- and 14-pin devices, just the same way as the 40-pin socket programs both as does the 40- and 28-pin chips. They are combined as shown in Fig. 4.31.

While a simple 14-pin DIP socket can be used for the 8/14-pin PIC microcontroller devices, the 28- and 40-pin PIC microcontrollers use a slightly different arrangement. The 28-pin PIC microcontrollers are built as 0.300in “skinny DIPs.” The combined socket uses a cut-up 40-pin socket along with a SIP connector that was cut from another socket. This socket is not wired for a ZIF socket.

When building the El Cheapo, I suggest that you do it in the order that I have listed below and stop and check after each point as I indicate. This process is also given in the Windows software under the Build/Test option.

- 1** First test the power connector (J1) with the AC/DC power supply that you have selected. This is done by plugging the power connector into the power supply and plugging the power supply into a wall socket followed by measuring the voltage at the connector. The two end pins of the connector should be checked with the terminal away from the connector hole being positive. The output voltage must be at least 14.5V for the El Cheapo to work properly. The actual output voltage of the power supply should be between 14.5V and 16V for the power supply to work properly. Too high and you will find the 78L12 (U1) will get very hot during programming and could shut down.
- 2** Next, solder the connector onto the board along with the CR4 diode. This diode is used to rectify the power coming in and make sure that the power is positive. This diode cannot be depended upon to provide rectified AC output. After soldering J1 and

CR4 to the board, check the voltage at C1 to make sure that the input voltage is greater than 13.75V. If the voltage is less than 13.75V, you will have to find another AC/DC power supply.

- 3** Wire in the 13.4V power supply. In doing this, solder in C1, U1, CR2, CR3, and C4. When finished, check the voltage output with the AC/DC power supply connected to the El Cheapo board. The output should be between 13V and 14V. The two diodes, CR2 and CR3, are used to boost the voltage output from 12V to more than 13V to ensure that EPROM parts will be properly programmed.
- 4** Now wire in the 5.1V power supply by soldering in R1, C2, and CR1. Note that R1 and CR1 should be capable of dissipating 0.5W of power. When there is no PIC microcontroller in any of the sockets, 40 mA will flow through CR1 and R1. This translates to 320 mW or more of power that requires the larger parts to dissipate the heat. The output should be checked with the digital multimeter checking pin 1 and pin 14 of the 14-pin socket.
- 5** Solder in J2. This will be required for the following build steps. As I noted above, make sure the connector is female (with holes for accepting pins).
- 6** With J2 in, the reset control circuit will be soldered in. Solder in Q1, Q2, R5, and R6 making sure that you get the transistor polarities correct before soldering in the parts. To test the reset control circuit, connect the El Cheapo to the PC and power and follow option 2 (Reset) of the El Debug program described below.
- 7** Next, solder in R3, R4, C3, and CR5. This provides a hardware delay that will be used by the programmer software. This circuit is tested using option 3 (RC Delay) of El Debug.
- 8** The last electronic component to install is R2. Once this is done, check the operation of the clock and data pins using El Cheapo.
- 9** Finally, install the PIC microcontroller sockets. If ZIF sockets are to be used for 14- and 18-pin parts, make sure the parts are open when they are soldered in to avoid any problems with the pins being soldered into a position where they are stressed and the socket pin cannot open properly. The 28- and 40-pin socket is created by cutting the top strut from a 40-pin socket to allow the 14-pin SIP socket to be installed inside it with the same level for pin 1. The 14-pin SIP socket is cut down from a 28- or 40-pin socket and soldered in between. When installing the 14-pin socket, make sure the pins are oriented in such a way that the PIC microcontroller will be fully seated when it is installed.

If you have problems with the El Cheapo, I suggest you carry out the following steps before contacting me (because I'll just ask you to carry them out before I will respond to you):

- 1** Make sure that the parts are installed and soldered in the correct orientation.
- 2** Check the parallel cable and make sure that it is straight-through and male to male, and that J2 is a female PCB mount DB-25 connector.
- 3** Use El Debug and ensure that each aspect of the programmer is working.
- 4** Check for any error messages (such as "Unable to Access Port Registers") and check that you have administrator access rights to the PC.
- 5** Test the programmer with a known good Flash PIC microcontroller (ideally a PIC16F84) with it placed in and out of the socket.

Third-Party Programmers

In the first edition of this book, I stated that a real cottage industry has grown up around creating PIC microcontroller programmers. There are literally hundreds of designs and software interfaces, which makes choosing a programmer difficult. Choosing the right programmer for you can be a frustrating experience, especially if you try various designs only to find that they have problems.

There are two types of programmers on the market today that can be used for the PIC microcontroller. The first is the commercial, “professional” programmer. These programmers can usually program a lot more than just a PIC microcontroller and will program the devices exactly according to the Microchip specifications. The downside to these programmers is their price; they are often \$500 or more for the base unit and additional money must be spent for adapters designed for specific parts. These devices are best suited for manufacturing sites or commercial development sites where designs using more than one microcontroller device type are produced.

For the hobbyist or the small business that is interested in looking at the PIC microcontroller for the first time and does not want to invest a lot of money into equipment that may not be required, there are a plethora of low cost programmers. The El Cheapo and YAP-II presented here fit into this category as well as the Microchip PIC-START Plus. The microEngineering Labs EPIC programmer (shown in Fig. 4.32) also belongs here.

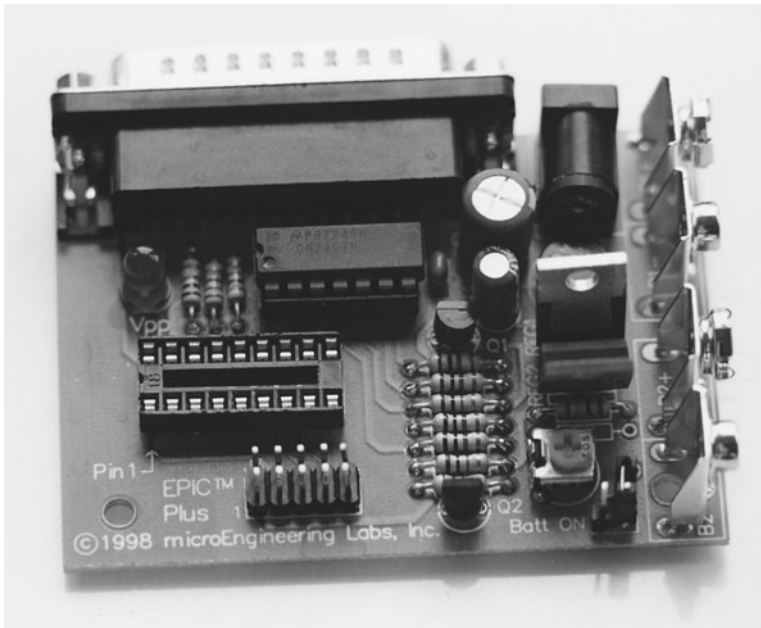


Figure 4.32 The microEngineering Labs EPIC programmer.

When you look at a low-cost programmer, you should consider the following characteristics:

- What are the supported PIC microcontroller devices?
- What is the interface and how is the application timed?
- How are the configuration fuses programmed?
- What operating system does it run under?

Notice that I did not list cost or programming time as characteristics you should consider when buying PIC microcontroller programmer. “Cost” is a very subjective term and can be very misleading. A low cost programmer can become very costly if all the bells and whistles are added to it. As I was writing the second edition of this book and working through the El Cheapo, I looked at listing the costs of all the parts needed to build the El Cheapo and run through the experiments. Calculating retail prices for the El Cheapo with all the parts on its board (not including ZIF sockets), the cables, a wall-mounted AC/DC adapter, 5V power supply, a sample PIC16F84, a breadboard, and the parts needed to work through the experiments, the cost would be approximately \$100—adding the three ZIF sockets would add another \$60. It makes you feel like the El Cheapo doesn’t live up to its name.

You should be aware of the various PIC microcontroller devices that are supported by the programmer you are looking at. I tend to design my projects with only the ICSP enabled devices. This simplifies the types of programmers needed to a very small set. Note that the 17Cxx ICSP is not compatible with the other devices’ ICSP and programming of the boot ROM is required before the application can be loaded into the PIC microcontroller.

One issue that isn’t often addressed is whether EPROM as well as Flash devices can be programmed. Many people assume that because the ICSP pins and packet protocol for the two different program memory types is the same, a programmer that can do Flash can also do EPROM. This is not the case because of the EPROM’s 50 mA Vpp source requirements. For the El Cheapo and the YAP-II, the Vpp circuits are seemingly more complex than they should be in order to supply the 50 mA Vpp for EPROM programming. Very few low cost devices have the capability of programming EPROM devices. The El Cheapo, YAP-II, and microEngineering Lab’s EPIC are among the few that do have the Vpp current drive capabilities.

If you are new to working with the PIC microcontroller, I highly recommend not buying a programmer that you have to set the configuration fuses on. Chances are that when you first start working with the PIC microcontroller, you will have a lot to learn and a lot to remember. Having to remember fuse settings is not something you should consider as you will invariably forget and end up with a PIC microcontroller that doesn’t run after a simple application tweak and you will have no way to find out what the problem is.

This problem invariably happens when you are under stress already with an assignment or project deadline due. Not having to worry about what the configuration fuses are set to (other than in the source code) eliminates one variable and potential problem for you.

Along with PC speeds getting faster and faster, you also have to consider the operating system that is used on your PC. Many programmers are designed to run under

MS-DOS. While there is an MS-DOS prompt under Windows/3.1x/95/98/NT/2000, you cannot count on it being able to access all the hardware in the system. This is of considerable concern with programmers that use the parallel port; if you do not have administrator access in Windows/NT/2000, you will probably find that you cannot use the programmer without your network administrator granting a session special access. Linux offers MS-DOS support, but this may be limited in the I/O ports that are accessible from it.

The safest bet is to only use a programmer that accesses the PC via the RS-232 serial port. RS-232 support and drivers are available under all operating systems without requiring special access rights.

If you are going to use a programmer that accesses to your PC via the parallel port, I suggest that you buy a second parallel port adapter if you are using the primary parallel port for a printer. Doubling up the functions on a PC's parallel port is time consuming and probably won't work that well with different software applications claiming the hardware as their own. A USB, ISA, or PCI parallel port adapter card can be purchased new for less than \$15 and you will feel it is worth it after just a few minutes of working with it instead of sharing a port with other devices.

The information provided in this section should be available from the programmer manufacturer. If it isn't, then look at another programmer. Over the years, I have had enough problems with balky PC adapters that I recommend you avoid any potential problems and find a programmer that will be well supported by an established company or has hardware interfaces that do not require software upgrades as time goes on.

The programmer should have a 5-pin ICSP cable adapter that can be bought separately. This cable provides power, V_{pp} , and programming clock and data signals to a connector built onto a product. The connector consists of a 5x1 connector with pins spaced 0.100in apart and would provide a connection to the target PIC microcontroller with the features discussed in Chap. 5.

EMULATORS AND DEBUGGERS

The purpose of the PIC[®] microcontroller emulators and debuggers is to provide you with an interface into the executing part, allowing you to monitor actual data, external I/O pin signals and stop at specific locations in the application. Emulators replace the entire circuit whereas debuggers involve specialized hardware built into the MCU to allow you to monitor the execution of the device in the application. In the debugging process, emulators and debuggers can be invaluable for identifying the state of the device and the circuitry it is connected to as part of the characterization and verification steps of failure analysis. Unfortunately, they are often not used for this purpose, instead they are used as an integral part of the development process. The danger of using an emulator or debugger as a development tool is that the possible resulting application is not fully thought out and can have numerous problems that are hard to find.

There is a fine line between PIC microcontroller emulators and debuggers. The reason for the difficulty distinguishing between the two is the similar operation of the two types of devices. The emulator replaces the microcontroller chip with a special chip that consists of a microcontroller with a high speed interface to an emulator, allowing the operation of the chip to be controlled and monitored. A debugger consists of a PIC MCU chip with specialized hardware that allows control of the processor and the ability to sample different registers. The primary difference between the two consists of the debugger's need to allocate two or three I/O pins to the controller connection (the emulator does not sacrifice any I/O pins for this purpose), and communication of the debugger to the control hardware is relatively slow (especially when compared to the emulator). If you are familiar with either the emulator or debugger, you won't have any trouble working with the other.

There are four types of emulators that are available for the PIC microcontroller. While there are a number of low-cost emulators available (including the Emu-II presented later in this chapter), the full device emulator (like the MPLAB ICE-2000) is the best device on the market to use. The first type of emulator is a simple hardware interface

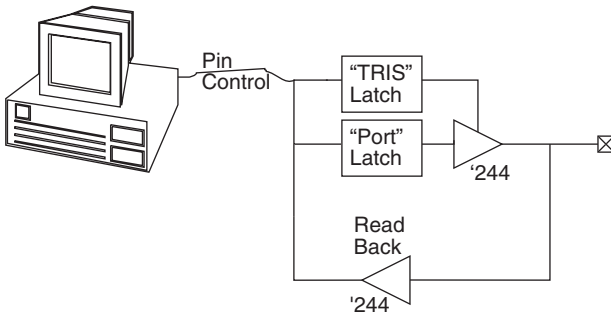


Figure 5.1 The most basic emulator consists of I/O functions that can be controlled by a PC.

emulator that is controlled by a PC or workstation (see Fig. 5.1). In this type of emulator, simulator code in the PC runs the PIC microcontroller application and accesses the emulator pod's I/O pins. This type of emulator is very inexpensive, but probably is the least accurate of the three types in terms of timing and electrical behavior of the I/O pins. Using a PC with a PIII processor and a high speed interface, the actual speed of the application will be approximately the same as what the PIC microcontroller would produce although there would probably be some significant differences in edge to edge timing.

The pin operation can be difficult to properly simulate using discrete chips, but a CMOS PLD could be designed to accurately model the PIC microcontroller's pin behavior. This problem could be eliminated by writing a small PIC microcontroller application that allows a PC to interface with it and remotely control the operation of the I/O ports as is shown in Fig. 5.2. This emulator is the same type as the circuit presented later in this chapter (the Emu-II) and takes advantage of the PIC16F87x family of PIC microcontrollers, which can read and write internal program memory. The Emu-II uses this feature to load and execute code in the PIC microcontroller as is shown in Fig. 5.3. This type of emulator is often called a downloader and can be created quite inexpensively, but the limited program memory of the PIC microcontroller means that only relatively simple operations can be implemented.

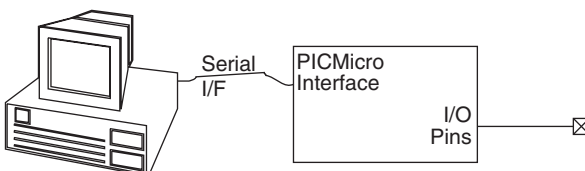


Figure 5.2 A PIC microcontroller can be used to provide the I/O pins required for an emulator controlled by an external PC.

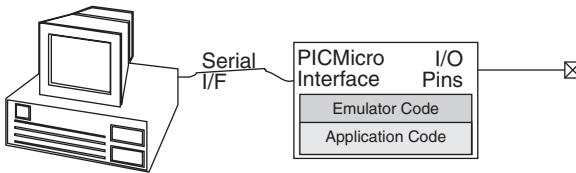


Figure 5.3 A PIC microcontroller executing the emulator functions by running the application code within its program memory and communicating with an external controller.

The emulator concept presented in Fig. 5.3 is extended to be built into many different PIC microcontroller devices that have ICD 2 debugger functionality built into them (Fig. 5.4). The 16F87x, PIC18, and many other PIC microcontroller part numbers have built-in serial port features that allow custom hardware to a synchronous serial port to access the different registers and functions of the PIC microcontroller via the MPLAB ICD 2 Puck, which provides a USB interface to the PC and a synchronous serial interface to the emulated (debugged) chip. As noted above, a selection of I/O pins (often MCLR#, RB3, RB6, and RB7) is required for the emulator functions and using these pins for this purpose reduces the total number of pins available to the application. For some devices, like the 18-pin parts, these pins are part of the only full 8-bit port available, which means that often the application cannot run the full, unchanged application with the ICD 2 debugger active.

The last type of emulator is the most expensive, but it does address all the issues raised by the other methods that have been presented. Built into every PIC microcontroller chip are the I/O pins that can be used by external hardware to control the operation of the PIC microcontroller as well as provide a separate memory to execute out of. To provide access to these I/O pins, a special type of package (known as a bondout chip) is used. The bondout chip package is used to interface to the application and the emulator so that if any damaging voltages or currents are driven into the emulator by the application, this is the only part that has to be replaced. The cost for a bondout chip is on the order of \$500, which, though expensive, is a lot cheaper than replacing a \$2,000 emulator. The block for this type of emulator is shown in Fig. 5.5.

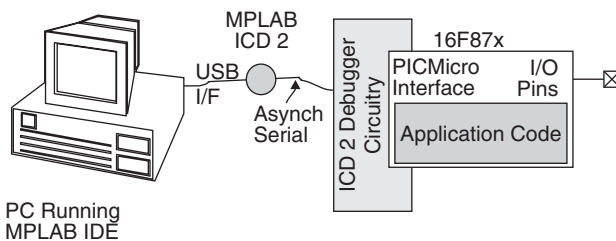


Figure 5.4 The ICD 2 debugger interface allows the PIC microcontroller to work as if it were an emulator.

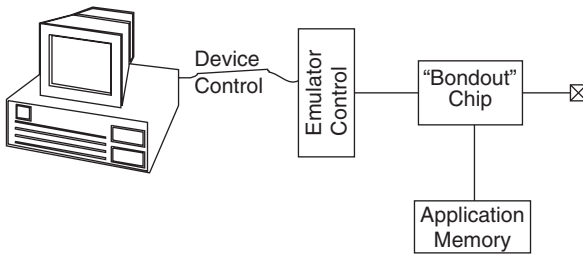


Figure 5.5 PIC microcontroller emulator implemented using a bondout chip that is connected to custom hardware controlled by a PC running MPLAB IDE.

MPLAB ICE-2000

Microchip's MPLAB ICE 2000 emulator (Fig. 5.6) is close to the ultimate tool for understanding the operation of a PIC microcontroller application. The MPLAB ICE 2000 consists of a roughly 7in by 6in box (pod) that is about 0.75in deep (it is surprisingly small) and connects to an unregulated DC power source, a PC's parallel port, and a processor module. The processor module is similar to a PCMCIA (PC-card) circuit, with a 15in ribbon cable leading from it to a product connector that is connected to a device adapter or transition socket. This hardware organization is shown in Fig. 5.7 and should give you an idea of how simple the connections are. To make it easier to interface to the target system, the MPLAB ICE 2000 includes a small tripod

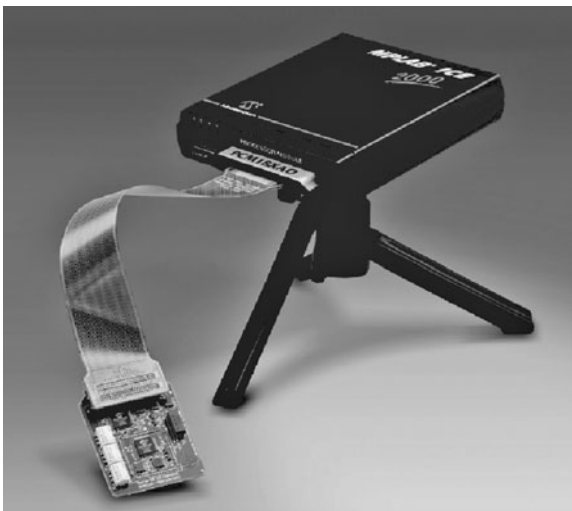


Figure 5.6 The Microchip MPLAB ICE 2000 emulator provides you with the ability to truly see inside your executing application.

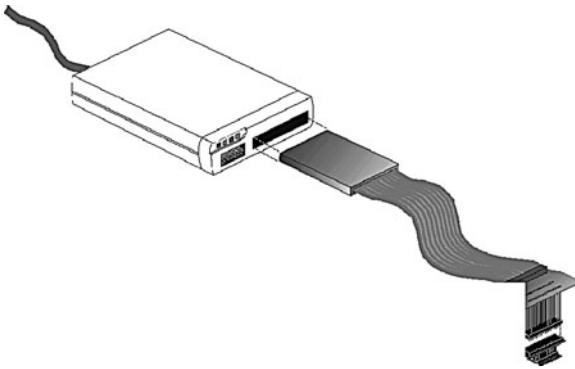


Figure 5.7 The device adapter plugs into the MPLAB ICE 2000 pod as shown here.

to raise the pod over the circuit under test and minimize the mechanical stress on the device adapter cable and the PIC microcontroller socket, resulting in a reliable electrical connection. The MPLAB ICE 2000 is the second generation emulator from Microchip; the earlier tool was much bulkier, more expensive, and had a large, heavy, and inflexible cable linking the emulated PIC microcontroller to the control hardware.

The MPLAB-ICE 2000 has the following list of impressive features and can emulate the entire PIC microcontroller line, including some of the latest PIC18 devices:

- Programmable internal clock able to provide clocks for the emulated devices running from 32 KHz to 40 MHz
- Full 2V to 5.5V operating voltage operation
- Breakpoints on execution address, internal device address, register contents, or external events
- Complex breakpoints can be built out of four different breakpoint sources
- 32K of trace memory
- Oscilloscope/logic analyzer input/output
- Processor modules/device adapters/transition sockets for all classes of PIC microcontrollers

The programmable clock is a very nice feature to have because it gives you “what if” capability to look at how the application could be implemented for various clocking schemes. This can be especially useful for looking at different ways of implementing applications.

The various breakpoint options bear a few comments and their usefulness is probably not readily apparent when you first look at the capability. As you work through applications, you will find events that you want to understand happening “in the middle” of the application and that can be very hard to trigger on.

An example of this would be a PIC microcontroller application that is communicating with another device and seems to have a problem in the fourth communications

packet. In a traditional system, to trigger on this, you would either need a very long delay or you would have to develop some hardware to look for the specific event (I've done both over the years). With the MPLAB ICE 2000, you can set up where you want to trigger algorithmically and then wait for the event to happen to look at where the problem is.

Many interfaces cannot stand to have the processor stopped for a human to single-step through to see what the problem is. The reasons for this not being possible can be the specific timing of the interface or the timeouts built into the communication protocol. To avoid these issues, you could single-step both communicating devices or you could use the trace buffer in the PIC microcontroller to follow through what happened at a specific point of time. I really like the trace buffer feature because it allows me to see exactly what has happened and I can identify very specifically what the problem is and how it is manifested. When I make my proposed changes to the code to fix the problem, I can run the changes through the trace buffer and see how the changes affect the operation of the application.

Further enhancing your ability to observe problems are the logical analyzer and oscilloscope interfaces that are available on the front of the MPLAB ICE 2000. These connectors (which are not discernable in the photographs used for this section) allow you to either trigger the emulator's trace buffer remotely or use the complex triggering capability of the MPLAB ICE 2000 to trigger other pieces of test equipment. The connectors can be used for external devices to trigger the MPLAB-ICE 2000 to see what is happening in the code at a specific external event.

The last point about processor modules, device adapters, and transition sockets being available for every class of PIC microcontroller is important in order to understand exactly what I mean. As I have discussed elsewhere in the book, there are literally hundreds of different PIC microcontroller devices. To provide a processor module for each and every PIC microcontroller part number would be economically unfeasible for Microchip to produce, for distributors to keep in stock, and for you to buy. Instead, you should be looking for the supported device that best fits your requirements. As I write this, the supported devices include:

- PIC12C5xx
- PIC12C67x
- PIC14000
- PIC16C505
- PIC16C52, PIC16C54, PIC16C55, PIC16C56, PIC16C57, and PIC16C58
- PIC16C55x
- PIC16C62x
- PIC16C6x
- PIC16C6x2
- PIC16F648A
- PIC16C71x, PIC16C72, PIC16C73A, and PIC16C74A
- PIC16C770, PIC16C
- PIC16C773, PIC16C774, and PIC16C777
- PIC16F877

- PIC16C92x
- PIC17C4x
- PIC17C5x
- PIC18C658, PIC16C858
- PIC18F6680, PIC18F8680

Looking at this list and comparing it to the devices that I use in this book, you will notice some missing parts—for example, the PIC16F84A. To emulate a PIC16F84A application, I would probably use a PIC16F848A processor module and create the code in such a way that can be easily ported between the emulated device and the actual hardware. This is what I mean when I say that the MPLAB ICE 2000 supports classes of PIC microcontrollers. You should be able to find a supported part that is very close to the part that you actually want to use in your application.

Looking over the list again, you should see that the overwhelming device technology type emulated by the MPLAB ICE 2000 is EPROM (C technology type) parts. These parts do not have the electrically reprogrammable Flash memory of the F technology type parts. This means that, without an emulator, you will have to either buy EPROM parts with an ultraviolet erasing window or resign yourself to throwing away parts as you debug the application. As will be discussed later in this chapter, many of the Flash-based parts can connect to a debugger, mitigating the need for the emulator or, at the very worst, can be reprogrammed in circuit. The emulator, with its built-in reprogrammable memory, allows you to develop applications for EPROM-based PIC microcontrollers without having to pull, erase, and reprogram parts (or worse, program and discard them).

MPLAB REAL ICE

As this book was going to press, Microchip made its MPLAB REAL ICE emulator available for sale. This tool has many of the same features as the MPLAB ICD 2 but is much faster and provides some emulator features, including execution trace logging, while providing a similar user interface as the other MPLAB IDE tools discussed in this book. To provide the bandwidth necessary for these functions, a USB 2.0 connection is provided between the MPLAB REAL ICE emulator pod and the controlling PC. Along with the improved connections between the emulator and the PC, the tool also takes advantage of the ICD (described below) connection built into many PIC microcontroller part numbers, simplifying the connection between the emulator and the in-circuit PIC microcontroller target device.

It is interesting to see the evolution of in-PIC microcontroller emulators over the years. The original emulator, the PICMASTER, was several thousand dollars, was extremely large, and had a very delicate connection between it and the target device while using a nonstandard ISA bus interface to the PC. The MPLAB ICE 2000 cost around \$1,000, had an improved interface between the target device and the emulator pod, but had a nonstandard parallel port interface. Finally, the MPLAB REAL ICE costs

a few hundred dollars, uses the simple and reliable RJ11 connector of the ICD 2, and communicates with the host PC using USB 2.0. The one thing all the emulators have in common is their integration with MPLAB IDE, which means that once a developer is used to working with an emulator in MPLAB IDE, he or she should be able to work with other ones with a minimum of learning.

By using the ICD interface built onto the PIC microcontrollers, the MPLAB REAL ICE allows very fast and easy connection into the circuit not only for application bring-up and debug but for debugging of defective products. It also avoids the need for Microchip to develop the special chips and PCB interfaces of the MPLAB ICE 2000, which allows inclusion of new part numbers much faster, allowing customers to implement new PIC microcontroller chips into their designs much more quickly. The only downside to using the ICD interface is the requirement to devote the pins used for clocking and data to the MPLAB REAL ICE, while in the ICD the pins can be shared with other functions in the application. This can be a significant issue for small, limited pin count PIC microcontroller part numbers. Overall, I believe the advantages of MPLAB REAL ICE using the ICD interface make it the best choice for PIC microcontroller emulators.

The MPLAB REAL ICE provides a much faster interface than the standard MPLAB ICD 2 and provides additional functionality in terms of a trace buffer. The trace buffer allows the instruction and data stream encountered by the PIC microcontroller to be recorded and then played back at a later time. Along with the ICD 2's 6-pin connector, the MPLAB REAL ICE can interface to the target PIC microcontroller via a high speed, low voltage differential pair interface. A further enhancement is an 8-pin logic input or output interface built into the MPLAB REAL ICE pod, giving you the ability to create logic signal traces of the application or inject your own signals at a specific time. MPLAB IDE running REAL ICE responds almost as quickly as it would if it were just running a device emulator.

The other, useful features available in MPLAB REAL ICE are:

- Real-time watch and stopwatch
- Full program execution control with breakpoints, single-stepping, and the ability to review or change the values of registers and variables
- Full PIC microcontroller voltage range operation (2V to 6V logic swings)

Finally, it should be noted that MPLAB REAL ICE also has the ability to program your chips in circuit. This makes the MPLAB REAL ICE a step up from the MPLAB ICD 2 while retaining its ability to program and debug your applications in circuit.

MPLAB ICD 2 Debugger

The best all-around tool for beginning developers working with the PIC microcontroller is the MPLAB ICD 2 debugger (Fig. 5.8). The MPLAB ICD 2 takes advantage of the debugger features built into most of the recently released PIC microcontroller part numbers and is designed to integrate with MPLAB IDE and the development circuit to provide you with

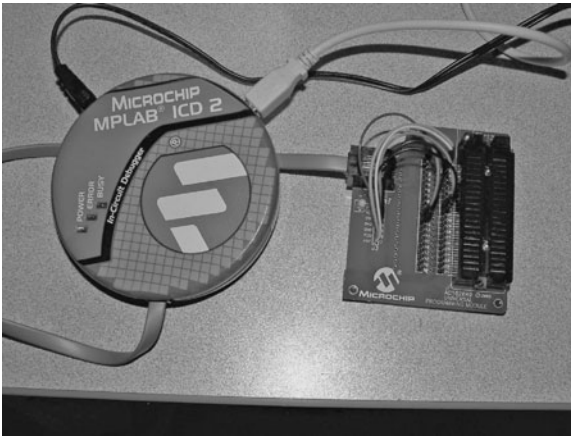


Figure 5.8 The MPLAB ICD 2 is shown here with Microchip's Universal Programming Adapter.

the ability of stopping at specific breakpoints and reading/updating file and hardware registers, just like a true in-circuit emulator. The ICD interface uses the same I/O pins as the ICSP programming interface and the ICD can implement both functions. If you were to look back at the MPLAB REAL ICE, you would see the same methodology for connecting to the chip in circuit. I am pleased to be able to include a coupon in this book for you to buy the MPLAB ICD 2 directly from Microchip because I believe it is the best tool available for developing PIC microcontroller applications and learning about the device.

There are a number of MPLAB ICD 2 kits that you can purchase. Figure 5.8 shows an MPLAB ICD 2 (which is available by itself with an AC/DC power adapter, USB cable, and six-conductor ICD cable) along with the Microchip Universal Programming Adapter, which can be used to program any PTH PIC microcontrollers by changing the jumper wires on the PCB. Along with the Universal Programming Adapter, there is the ICD-to-ICSP adapter that was discussed in Chap. 3 as well as a number of development kits that take advantage of the ICD interface. I should also point out that there are a number of ICD adapters, consisting of a small PCB containing an ICD enabled part that can be inserted in circuit instead of a basic PIC microcontroller, to provide you with an inexpensive in-circuit emulator for your application. The list of development kits and adapters is continually growing and when you create your application, I recommend that you spend some time researching what is available from Microchip to make sure that you have the right options and kits.

Enabling the MPLAB ICD 2 debugger from MPLAB IDE is as simple as clicking on the Debugger pull-down menu, followed by Select Tool and then MPLAB ICD 2. Similarly, if you want to use the MPLAB ICD 2 for programming a device, you simply click on the Programmer pull-down menu in MPLAB IDE and then Select Programmer and MPLAB ICD 2. This level of integration with MPLAB IDE makes using MPLAB ICD 2 with your application, either as a debugger or programmer, very simple and intuitive.

It is important to remember that MPLAB ICD 2 is a debugger and a programmer with the programmer function being invoked anytime the source code is modified (requiring

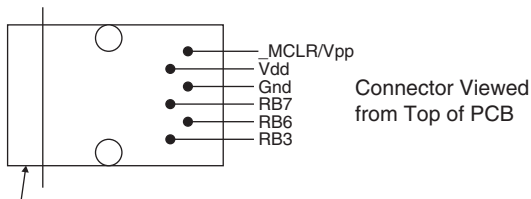
a rebuild of the application followed by programming the target PIC microcontroller with the updated application). When the programmer function is used at this time, the ICD interface code is loaded in the last 256 instructions in memory. When you are ready to release the code, you must reprogram the chip using the MPLAB ICD 2 as a programmer to ensure that the hooks put into the application are not left in the chip when you have finished application development.

When you are using MPLAB ICD 2, you will find that a number of functions that execute quite quickly in the MPLAB simulator (or MPLAB-ICE 2000) are mind-numbingly slow in MPLAB ICD 2. These functions include building the project and single-stepping. Building the project involves programming the PIC microcontroller with the application (the programming operations can be observed in the MPLAB ICD 2 Debugger dialog box). Single-stepping involves returning the contents of all the hardware I/O registers as well as the contents of the file registers. This operation cannot be limited to just the registers that are changed because there is no space to store the previous contents of the registers for a comparison to see what has been updated. After using MPLAB ICD 2 for a while, you'll discover that you rely on strategically placed breakpoints more than single-stepping to monitor the execution of an application to avoid the slow response of the tool.

To perform the MPLAB ICD 2 functions, a Flash program memory equipped PIC microcontroller is used with the MPLAB ICD 2 module board. There are many mid-range and PIC18 part numbers that have the ICD functions built into them.

If you are porting an application to another PIC microcontroller, you will find that some PIC microcontroller part numbers will make it necessary to keep track of the pins and file registers that are available in the device that you want to use. My first suggestion is that you do not use PORTB at all except for some simple digital I/O. I/O pins RB3, RB6, and RB7 interface directly to the MPLAB ICD 2 and prevent PORTB's use as an 8-pin I/O register. When I have tested applications on MPLAB ICD 2, I specify PORTC or PORTD (depending on the requirements of advanced I/O) for the 8-bit wide port bits.

One of the interesting aspects of MPLAB ICD 2 is the ability to simply add the RJ-45 connector that is used with the MPLAB ICD 2 module. The connector is a standard AMP part and the pinout is shown in Fig. 5.9. Note that the MPLAB-ICD module card is



AMP 5-554710-3, Plug, 6-6 Strand Round Modular Plug
Digi-Key Part Number: A9117-ND

Figure 5.9 The ICD RJ-45 connector can be mounted to the product PCB to allow direct programming and debug connection to the PIC microcontroller.

powered from the application; an extra 70 mA at 5V should be available for the module card's power requirements.

MPLAB ICD 2 PROGRAMMING

The software changes required to allow MPLAB ICD 2 to be used to help debug an application are surprisingly minimal and will probably not affect the overall operation of the PIC in the application or require conditional builds.

The single change that must be made to support ICD is the use of a `nop` instruction at the reset address (0). Rather than worry about when the `nop` should be used and when it can be left out, I simply put it into all my application code, regardless of whether or not the PIC microcontroller part number selected for the application supports it. I would even go so far as to say that the `nop` instruction should be put in low-end PIC microcontroller applications to ensure that you never forget to put it in.

The ICD interface code that runs in the PIC microcontroller is installed automatically in the last 256 instructions of the chip. In earlier versions of MPLAB IDE, this memory would have to be reserved, but the current versions of the integrated development environment will give you error messages if you have code or data that extends into this region.

Finally, you have to make sure that the configuration fuses will not enable the code protection or disable the Flash self-write features of the PIC microcontroller. To ensure that there will be no problems, the following four options should be selected in the application's configuration fuses:

- No code protection
- Debug interface enabled
- Internal program memory writes enabled
- Watchdog timer disabled

All other configuration fuse values, including clocking and reset options, can be used with MPLAB ICD 2. If any of the options above are required for your application, you will have to implement some kind of conditional build in which you can specify the configuration values when required.

ICSP VERSUS ICD

One set of definitions that confuses many people is the difference between ICSP (in-circuit serial programming) and ICD (in-circuit debugger). The functions seem similar and in the cases of the MPLAB ICD 2 and MPLAB REAL ICE the tools can be used for both functions.

The primary difference that you should be aware of is the connectors used. ICD uses the RJ-45 connector, shown in Fig. 5.9. ICSP uses the 6-pin interface shown in Fig. 5.10. The ICSP connector can be easily plugged into a breadboard or other PTH chip prototyping system, whereas the ICD connector requires a PCB with the RJ-45 connector footprint built in to provide the same capabilities.

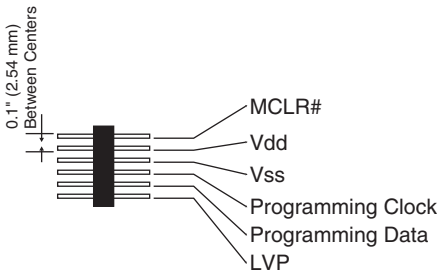


Figure 5.10 The ICSP connector has the same pinout as the ICD connector, but places the pins 0.1in apart for easy use with PTH chips and boards.

Fortunately, as will be shown in the next section, the schematic wiring of the two interfaces can be identical and the two functions (programming and debugging) can be combined into one interface using a small PCB like Microchip’s ICD 2 to ICSP adapter (Microchip part number AC164110), shown in Fig. 5.11. This adapter allows the ICD 2 to be used to program and debug PIC microcontrollers wired into breadboards and other PTH development tools easily and fairly inexpensively.

HARDWARE DESIGN FOR ICD AND ICSP

When I first started working with ICD (and ICSP), I liked the idea of creating circuitry that would allow the I/O pins used for ICD and programming to be shared with the



Figure 5.11 The Microchip ICD 2 to ICSP adapter (part number AC164110) is an easy way to use the MPLAB ICD 2 debugger with breadboarded applications.

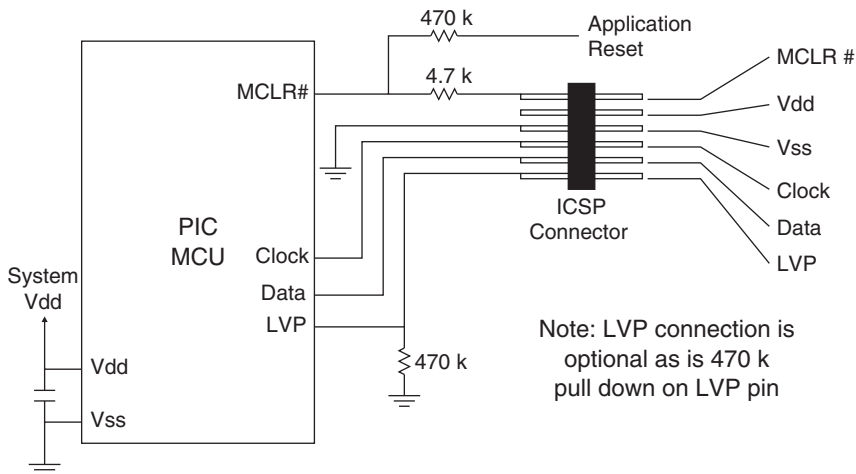


Figure 5.12 This circuit should be implemented when using the MPLAB ICD 2 or an ICSP programmer with the PIC microcontroller in the application circuit.

application. After working through a few applications in which ICD is used to debug the hardware, I have come to the conclusion that this approach was in error. Sharing I/O pins, even through current-limiting resistors, is a dangerous practice and one in which it is difficult to have a set of rules that will work in all applications. Similarly, care must be taken with PIC and system power and I have found it much less likely to have power problems if the PIC MCU is powered by the application rather than through the MPLAB ICD 2. The culmination of these rules is displayed in the schematic shown in Fig. 5.12 and will allow you to create a PIC microcontroller application that can be plugged into an MPLAB ICD 2 or an ICSP programmer at any time.

The EMU-II

When I first heard about the (original) MPLAB ICD and PIC16F877 Flash self-write capabilities, I was quite excited as I thought there was a way to use these capabilities to make my own simple in-circuit emulator. In the first edition of the book, I created a simple emulator (called the Emu), which stored a PIC microcontroller hex file in a serial EEPROM and executed this data using an instruction interpreter that I wrote. Unfortunately, the best operating speed this home-built emulator could do was 10,000 instructions per second, which was two orders of magnitude less than the execution speed of the PIC microcontroller in the application. To come up with a better emulator, I wanted to use the internal debug features of the PIC16F877 for my own emulator to see if I could improve upon the MPLAB ICD implementation and create a product I would be comfortable with. The final result is definitely not what I initially envisioned when I started the project, but it did turn into one of the most interesting applications in this book and something that I probably learned the most on.

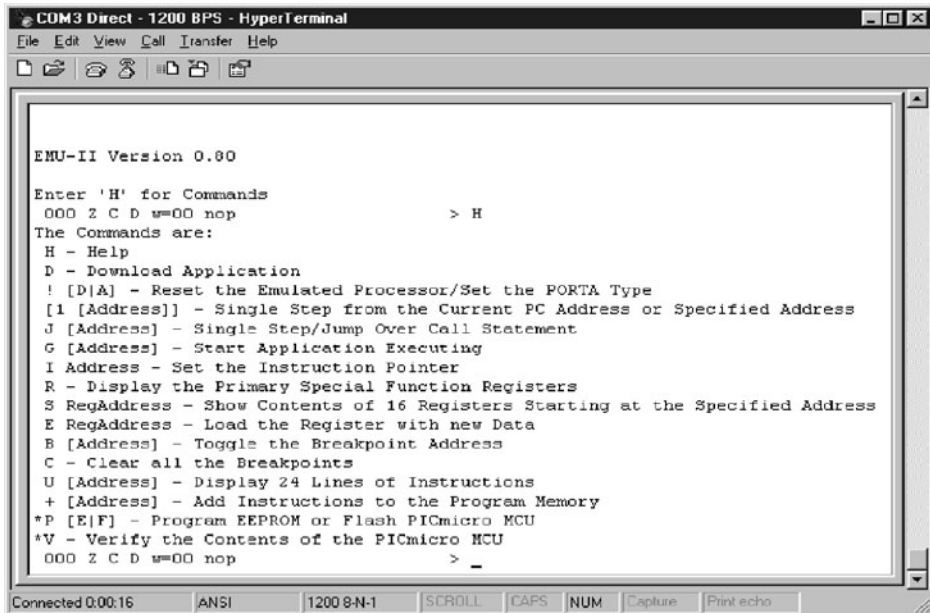


Figure 5.13 The execution options available to the Emu-II.

Even if you are not going to build this application, I urge you to read through it as there will be information you will find interesting and useful. As well, I have provided some macros that could be useful in your own applications.

The issues I had with MPLAB ICD that I wanted to improve upon were:

- Slow response to reset and single-step
- The single breakpoint
- No way of changing the program counter from the MPLAB IDE interface

The resulting application worked similarly to the YAP-II in that I provided an intelligent RS-232 interface to a terminal emulator (as well as a Visual Basic front end) that is completely self contained. The commands available to the Emu-II are listed in Fig. 5.13. Like the YAP-II, the Emu-II runs at 1200 bps and takes applications as hex files. The maximum application size is up to 1,792 (0x0700) instructions.

The EMU-II provides the same functions as higher cost emulators with the added features:

- The EMU-II is not device or operating system specific. Any PC, Mac, or workstation with a terminal emulator program and a serial port capable of running at 1200 bps can be used with the EMU-II.
- Pins 6 and 7 of PORTB are useable, allowing 18-pin PIC microcontroller MCU applications to be emulated and programmed directly into parts without having to debug on PORTC or splitting up the PORTB functions.
- Provisions for a built-in PIC microcontroller programmer are included in the application.
- Up to eight breakpoints can be specified during application execution.

Unsupported features and potential limitations of the EMU-II to be aware of include:

- Applications must start with a nop to allow an emulator execution breakpoint (this is the same as what MPLAB ICD requires).
- The PIC16F877 reset pin is passed to the application but consists of just an unbounced 10K pull-up and pull-down switch
- The 4 MHz clock available to the EMU-II is not distributed to the application circuit.
- It is not recommended that power provided on the board be distributed to the application circuit.
- Built-in assembly of instructions is not supported. There is a patch (+) command, however.
- There is no protected program memory for the application. For this reason, writes to Flash/data EEPROM are not recommended except using the built-in functions described below.
- Applications to be emulated are limited to 1,792 instructions in size.
- The EMU-II cannot single-step from a `return`, `retlw`, or `retfie` instruction.
- Variable memory in banks 2 and 3 should not be accessed.

The first order of business was to come up with a circuit to start from. The initial circuit was very similar to the YAP-II's and I designed a PCB that was based on the YAP-II's. I jokingly said that the circuit was almost identical to the YAP-II—the two ends are the same, only the middle was changed. This is actually quite true for the PCB that I designed for the board (the Gerber files of which can be found on the McGraw-Hill website).

The final circuit I came up with is shown in Fig. 5.14. What is not shown is that either a PIC16F877 or PIC16F876 can be used for driving the EMU-II. To allow either part to be used in the emulator, I avoided using PORTD and PORTE (which are not available in the 28 pin PIC16F876) for control pins. The PCB, which is presented below, is designed to accept either a PIC16F877 or PIC16F876. The reason for doing this was some initial problems with getting PIC16F876 parts when I was specifying the application.

The bill of material for the Emu-II is listed in Table 5.1.

There is a 19-pin connector that includes ten LEDs, two buttons, two potentiometers, a speaker, and some pull-ups that are wired exactly the same way as the YAP-II's built-in I/O devices.

When you look at this circuit, you should notice five important differences between it and the YAP-II. They are:

- The use of a PIC16F877/PIC16F876 instead of a PIC16C711.
- A 4 MHz ceramic resonator is used instead of a programmable oscillator.
- The processor signals passed to the breadboard area are different.
- The addition of U4, which is labeled 16F84 and is a socket in which 18-pin Scotchflex DIP connectors can be used to pass the emulated signals from the EMU-II to a development board.
- A number of PIC16F877/PIC16F876 I/O pins are unavailable in the Emu-II.

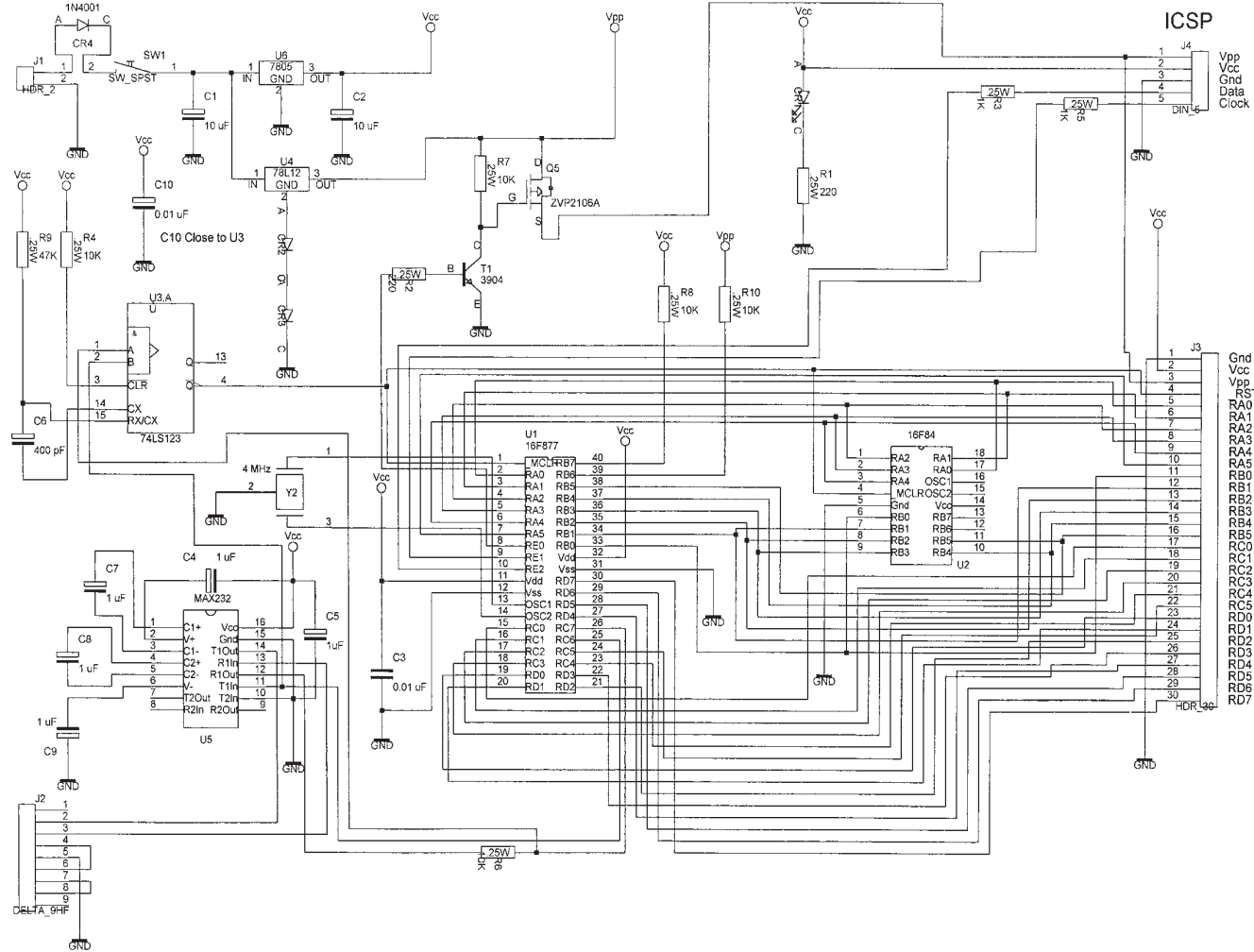


Figure 5.14 The EMU II schematic.

TABLE 5.1 BILL OF MATERIAL FOR THE EMU-II

PART	DESCRIPTION
U1	PIC16F877-04/P or PIC16F876-04/SP
U2	18-pin (ZIF) DIP socket
U3	74LS123 single shot
U4	78L12 +12 volt regulator in TO-92 package
U5	MAX232 RS-232 interface
U6	7805 +5 volt regulator in TO-220 package
T1	2N3904 NPN transistor
Q5	ZVP2106A P-channel MOSFET transistor
CR1, CR6	Red LED
CR2, CR3, CR4, CR5, CR7	1N1414 silicon transistors
CR4	1N4001 silicon transistor
Y1	4 MHz ceramic resonator with internal capacitors
R1	10K, 1/4 watt resistor
R2, R7	220 Ω , 1/4 watt resistor
R3	330 Ω , 1/4 watt resistor
C1 – C2	10 μ F, 35V electrolytic capacitors
C3	0.01 μ F tantalum capacitor
C4 – C8	1.0 μ F capacitor, any type
SW1	SPST switch
SW2	Momentary on" SPST switch
J1	2.5mm power socket
J2	9-pin female D-shell connector
J3	19 $\times$ 1 socket
J4, J6	5 $\times$ 1 ICSP connector
Misc.	PCB board, wiring

The EMU-II is designed to emulate an 18-pin PIC microcontroller as closely as possible. The I/O signals from the PIC16F877/PIC16F876 are available to the breadboard in exactly the same fashion as the YAP-II. This has resulted in some restrictions in how the PIC16F877/PIC16F876 is used:

- The USART port is dedicated to the EMU-II function. The RCSTA, TXSTA, RCREG, TXREG, and SPBRG registers should not be accessed in any way from an application.

- Bringing RC3–RC5 to the 19-pin product connector means that the MSSP hardware can be used with the application. But SPI slave mode cannot be implemented because RA5 (the `_SS` pin) is not passed to the emulated part.
- Other than MSSP functions, no other PORTC functions are to be accessed from the application. PORTC controls the operation of the programming reset circuitry and inadvertent writes to the hardware could potentially damage the EMU-II or the application circuit it is connected to.
- Note that none of the connections from the PIC16F877/PIC16F876 to the application circuit are protected. The reason for this is to allow the EMU-II to interface directly without any unexpected impedances that will affect the operation of the application.

The reason for the single frequency option for this board was to avoid the need for changing the delay constants (or timers) when the user selected a different operating frequency. Four MHz was chosen as the constant frequency because it is the one chosen for most applications.

U4 is left unpopulated and is in fact a socket for wiring an 18-pin emulated PIC microcontroller into the application's circuit. This is done by using an 18-pin Scotchflex ribbon cable between two 18-pin DIP connectors. Note that the `Vcc`, `OSC1`, and `OSC2` pins are left floating (unconnected) on the U4 connector to avoid contention problems with the Emu-II. `_MCLR` will be driven high by the EMU-II and will be toggled low when the `reset` command is initiated.

Once the EMU-II is built, it is connected to a PC (of any type) via an RS-232 cable and powered by 14+V from a wall-mounted AC/DC power converter. To interface with the EMU-II, a terminal emulator set to 1200 bps, 8-N-1 data protocol is used. No hardware handshaking or XON/XOFF control protocol is required for the interface.

The commands sent to the EMU-II are similar to those of the YAP-II but allow for hexadecimal address and data parameters as well as alphanumeric register parameters. Table 5.2 lists the commands available from the EMU-II.

Note that parameters listed in Table 5.2 that are in square brackets ([]) are optional. All numeric data is in hexadecimal, and register addresses can either be a hex address or a register name.

The operation of the application is quite straightforward with the only nonintuitive operation being the application download operation. To load a new application into the EMU-II's program memory, carry out the following steps:

- 1 Enter `D/CR` (or press Enter if a PC is being used). The EMU-II will return with a message saying that program memory is being cleared.
- 2 When the EMU-II requests the application to be downloaded, use the terminal emulator to send a `Text File Hex File`. This operation will be picked up by the EMU-II and the application code will be stored into the program memory devoted to the operation.

The download operation will not typically have any feedback as to the status of the operation. This can make the operation worrisome but note that after the application hex file has been sent, the EMU-II will poll incoming serial data for five seconds to ensure the host

TABLE 5.2 COMMAND SET FOR THE EMU-II

COMMAND	PARAMETERS	COMMENTS
H		Display list of commands
D		Download application hex file
!	[D A]	Reset emulated part with A (analog) or D (digital) PORTA I/O pins
1	[Address]	Single-step starting at current program counter or specified address
		If Enter pressed/carriage return sent without a command, single-step from current program counter is invoked
J	[Address]	Single-step starting at current program counter or specified address
		If instruction to single-step is a subroutine call, then execution resumes at the instruction after the call instruction
		If breakpoint is encountered within the subroutine, then execution will stop at that breakpoint
G	[Address]	Start executing at current program counter or specified address
		If breakpoint is encountered, then execution will stop at that breakpoint
I	[Address]	Set new program counter value
R		Display the primary special function registers in the PIC microcontroller MCU
S	Register	Display the contents of 16 registers starting at Register address
E	Register	Display and optionally change the contents of the specified register
B	[Address]	Toggle a breakpoint at either the specified address or the current program counter
C		Clear all the breakpoints in the emulator
U	[Address]	Disassemble the 22 instructions starting at either the current program counter or the specified address
+	[Address]	Load hex values that are to be entered into program memory either starting at the current program counter or the specified address
P	[E F]	Program an EPROM or Flash PIC microcontroller MCU from the contents of the EMU-II's program memory
V		Verify the contents of a PIC microcontroller MCU with the contents of the EMU-II's program memory

system (PC) download was not preempted by another task. No characters should be entered in the PC keyboard until the EMU-II prompt (which will be at address 0) has been displayed.

The register names listed in Table 5.3 have been built into the EMU-II to allow for some symbolic application debugging.

TABLE 5.3 REGISTERS SUPPORTED BY THE EMU-II			
BANK 0		BANK 1	
ADDRESS	REGISTER NAME	ADDRESS	REGISTER NAME
0x000	INDF	0x080	INDF
0x001	TMR0	0x081	OPTION (2)
0x002	PCL	0x082	PCL
0x003	STATUS	0x083	STATUS
0x004	FSR	0x084	FSR
0x005	PORTA	0x085	TRISA
0x006	PORTB	0x086	TRISB
0x007	PORTC	0x087	TRISC
0x008	PORTD (1)	0x088	TRISD (1)
0x009	PORTE (1)	0x089	TRISE (1)
0x00A	PCLATH	0x08A	PCLATH
0x00B	INTCON	0x08B	INTCON
0x00C	PIR1	0x08C	PIE1
0x00D	PIR2	0x08D	PIE2
0x00E	TMR1L	0x08E	PCON
0x00F	TMR1H	0x08F	Zero (3)
0x010	TCON1	0x090	Zero (3)
0x011	TMR2	0x091	SSPCON2
0x012	TCON2	0x092	PR2
0x013	SSPBUF	0x093	SSPADD
0x014	SSPCON	0x094	SSPSTAT
0x015	CCPR1L	0x095	Zero (3)
0x016	CCPR1H	0x096	Zero (3)
0x017	CCP1CON	0x097	Zero (3)
0x018	RCSTA (4)	0x098	TXSTA (4)
0x019	TXREG (4)	0x099	SPBRG (4)

TABLE 5.3 REGISTERS SUPPORTED BY THE EMU-II (CONTINUED)

BANK 0		BANK 1	
ADDRESS	REGISTER NAME	ADDRESS	REGISTER NAME
0x01A	RCREG (4)	0x09A	Zero (3)
0x01B	CCPR2L	0x09B	Zero (3)
0x01C	CCPR2H	0x09C	Zero (3)
0x01D	CCP2CON	0x09D	Zero (3)
0x01E	ADRESH	0x09E	ADRESL
0x01F	ADCON0	0x09F	ADCON1

1. Registers and I/O ports are not available in the PIC16F876.
2. Register is known as `OPTION_REG` in Microchip documentation/tools.
3. No special functions are devoted to these registers, 0x000 always returned upon register read.
4. Registers used by the EMU-II. These registers along with the EEPROM registers (EEDATA, EEDATH, EEADR, EEADRH, EECON1, and EECON2) should never be accessed except using the functions listed below.

As indicated in the table notes, the USART and EEPROM registers should never be accessed by an application. Instead, the functions listed in Table 5.4 should be used for serial communications with the PC host and for accessing the data EEPROM. The program memory EEPROM must never be accessed. The serial port registers are enabled for noninterrupt communications at 1200-8-N-1 and should not be modified in any way.

Applications written for generic PIC microcontroller MCU applications can be debugged using the EMU-II with very little modification. The EMU-II was designed

TABLE 5.4 BUILT-IN INTERFACE FUNCTIONS

ADDRESS	FUNCTION NAME	DESCRIPTION
0x07B0	SerialPoll	If character received and not yet read, return with the carry flag set
0x07C0	SerialRead	Wait until a character has been received and return it in w
0x07D0	SerialWrite	Send the character in w out serially to the host
0x07E0	EERead	Read the data EEPROM at the address specified in FSR
0x07F0	EEWrite	Write the data EEPROM with the value in w at the address specified in FSR

to minimize the need for developing applications that had to be modified for both EMU-II operation and actual application operation.

To create applications that can be debugged on the EMU-II, the following rules must be followed:

- `nop` as the first instruction at address 0x0000.
- The maximum size of the application is 1,791 (0x06FF) instructions.
- Variables should start at 0x020 rather than 0x00C as is possible in some devices.
- Use the USART and EEPROM functions listed above and do not access the special function registers that control these functions directly.
- For variables that are accessed from either Bank 0 or Bank 1, use address range 0x070 to 0x07E.

Ideally, applications should not access any registers in Bank 2 or Bank 3 as the EMU-II state variables are stored in these banks along with the EEPROM access control registers.

The `nop` instruction at the start of the application is used to allow a `goto EmuIIReset` instruction, which jumps to the EMU-II emulator application when the EMU-II first boots. This `goto` instruction is tested for in the EMU-II software and is treated like a `nop` during application execution.

As noted above, applications cannot be larger than 1,791 (0x06FF) instructions. The reason for this size is that the 256 instructions at addresses 0x0700 to 0x07FF is used by the EMU-II to provide an interface for the breakpoint handlers as well as the USART/data EEPROM functions listed above. As will be discussed below, when a breakpoint is set in the application, it jumps to the handler address. By placing the handler vectors in the first page, a single `goto` instruction is required and the PCLATH registers do not have to be modified by the breakpoint operation.

It is recommended that all applications specify the configuration flags to be used by the final PIC microcontroller application using the `__CONFIG` statement in the application source code. The `config` and `__IDLOCS` data will be stored within the EMU-II and will be programmed into a PIC microcontroller using the `Program` and `Verify` commands.

I originally wanted this application to take advantage of the built-in debug features demonstrated by MPLAB ICD. The MPLAB ICD's operation and the PIC16F87x built-in debug features are not documented by Microchip so I did a bit of hacking to find out what was going on inside them and how they worked.

Actually, the "hacking" just consisted of taking a PIC16F877 that had been programmed by the MPLAB-ICD and looking at what was put in it. I found that code that was very similar to an interrupt handler was placed at addresses 0x01F00 to 0x01FFF. This matched what I expected based on the requirements for the MPLAB ICD – in the specifications Microchip indicates that no application code can be placed in the PIC microcontroller at addresses 0x01F00 to 0x01FFF. (Along with leaving 0x01F00 to 0x01FFF alone, address 0 must have a `nop` instruction.)

To look at this code, I used my PICSTART Plus to read the contents of the PIC16F877 and then dumped the unassembled code into a text file. Once this was done, I spent some time labeling the code and making sure it could be reassembled properly (i.e., loaded back into a PIC16F877 and tested with MPLAB ICD). The resulting application code

is ICD. ASM and can be found in the PICDwnld\code\ICD subdirectory the PIC microcontroller directory.

This code is copyrighted by Microchip and cannot be distributed in any products. If you are planning on using this code in your own product you must contact Microchip first and get their permission. I have reproduced it here so you can see what is going on inside the PIC microcontroller controlled by the MPLAB ICD card. I should point out that the source code for the hex file which I have reverse engineered can be found on the Microchip web site as part of an application note explaining the operation of the ICD interface.

When you look at this code, you will see that it has the entry point of 0x01F00. I will call this the “debug interface” as that is what it is; it provides an interface between the executing application in the PIC microcontroller and the MPLAB-ICD. Execution jumps to this address any time the PIC microcontroller is reset or the previous command has completed executing. When execution jumps to this address, the context registers (the w, STATUS, FSR, and PCLATH registers) are saved and the contents of 0x018E and 0x018F are saved.

Registers 0x018E and 0x018F are reserved and used for passing information between the executing program and the debug interface. Upon debug interface entry at 0x01F00, these registers normally contain the address execution stopped at. Upon return to the executing application, these registers are loaded with a command, the context registers are restored and a `return` instruction is executed.

The debug interface communicates using a synchronous serial protocol with the MPLAB ICD. RB7 is a bidirectional I/O pin and RB6 is a clock provided by the MPLAB-ICD. The first clock cycle is a synching bit and is followed by a 32-bit data transfer. The data transfer is unusual in that when the clock line rises, the debug interface drives a bit from 0x018E and 0x018F onto the data line, and when the clock line falls, a new bit for 0x018E and 0x018F is read from the data line. Using this method, 16 bits of data are input and output using one clock.

There are four instructions passed from the MPLAB-ICD to the PIC microcontroller via this interface, listed in Table 5.5.

The “run to breakpoint address” command executes from the current program counter value and stops after it executes the specified instruction.

TABLE 5.5 MPLAB-ICD RESERVED ADDRESS INSTRUCTIONS

0x018E	0x018F	COMMAND
0x000	0x000	Reset the PIC microcontroller
0x040	0x000	Run to breakpoint address added to the 0x04000 instruction value
0x05F	0x000	Run freely
0x07F	0x000	Single-step

If an instruction to the executing hardware instead of the debug hardware is to be passed, then a command that does not have bit 6 of 0x018E is passed and this instruction is jumped to a handler address where the command (such as “read the contents of w”) is executed.

In observing this code and experimenting with my own, I was able to make the following observations about the built-in hardware debugger of the PIC16F877:

- Execution jumps to 0x01F00 after reset or after a command has finished executing.
- The first time `return` is executed, the debug interface ends and the application code executes again using the command in 0x018E and 0x018F.
- Hardware resources may or may not be available in the PIC microcontroller. I found that most resources are not available except for:
 - EEPROM read/write interface
 - PORTB (via Bank 2 and Bank 3)
- In the application code, RB6 and RB7 are not available for I/O.
- Changes to hardware registers take effect after the `return` is executed.
- Reset is required to enable the internal debug hardware.

This set of rules was developed using the MPLAB-ICD and monitoring the data being passed back and forth with an oscilloscope as well as experimenting with the Emu-II hardware.

My original plan was to implement the EMU-II using the built-in debug functions of the PIC16F87x and providing protected EMU-II application program memory from the address range 0x01000 to 0x01FFF. Though I was somewhat successful, I discovered that there were three problems with this approach. They were:

- 1 No hardware resources of the PIC microcontroller (i.e., the USART) except for the EEPROM/Flash read/write registers could be accessed while the PIC microcontroller was in debug mode. Serial communications relied upon bit-banging functions.
- 2 Only one breakpoint is available in the application. If I were to implement multiple breakpoints, I would have to put in code like:

```
movwf    _w                ; Save the "w" Register
swapf    PCLATH, w         ; Save the current PCLATH Register
movwf    _pclath
movlw    HIGH BreakpointVector
movwf    PCLATH            ; Setup the Jump to the Next page
goto     (BreakpointVector & 0x07FF) | ($ & 0x01800)
```

This would make setting single-step stop points at skip instructions (which have two possible stop points) impossible as well as making it impossible to set breakpoints for `goto` loops that were less than six instructions from their destinations.

- 3 At the end of an operation, the PIC microcontroller has to be reset (to put it back into internal debug mode). Using a 74123 single shot to toggle reset, I was able to accomplish this, but I felt the circuit was kludgy.

After experimenting with the built-in debugging mode of the PIC16F87x, I settled on the design that I am presenting in this section, which does not utilize the built-in debugging features of the PIC microcontroller. The resulting application is surprisingly fast (especially considering it works at 1200 bps) and very useful for new application developers.

This application is probably the most complex of anything presented in the book (as well as being far and away the longest). Despite having 6,400 instructions to work with, I found that I had to come up with some new ways of doing things in order to fit in all the functions that I wanted. Ideally, I would like to implement a line assembler as part of the EMU-II package—but for that, I would require 1,000 or more free instructions in program memory. There are four aspects of this application that I would like to bring to your attention, as I believe they are noteworthy and may help you out in your own applications.

The first is how the text table reads are implemented in the EMU-II application. For most other applications, I pass a text message number to a `SendMsg` subroutine, which reads through a large table of ASCIIZ messages to find the specific ASCIIZ message. When the selected message is found, it is output via the serial port until the ending NUL (0x000) character is encountered.

For the EMU-II, I had planned on implementing a large number of text messages to help guide the user (I wasn't expecting to have an application like the YAP-II Windows Interface to help the user). The message table that was developed was 2,888 ASCII characters long. If this table was used directly in the application, it would take up 45 percent of the total I had available.

The solution to this problem was to compress the tables so that two 7-bit ASCII characters could be placed in a single 14-bit instruction location. This would eliminate the two `retlw` instructions and take advantage of the PIC16F87x to read its own program memory.

To do this, I started with a typical table declaration (the `compress.asm` file in the `code\EMU-II` subdirectory of the PIC microcontroller directory) like this:

```
MsgTable                ; List of Messages
Msg0                    ; Start with \r\n (New Line)
    dt      0x00D, 0x00A, 0
Msg1                    ; Put in the "Prompt" Message:
    dt      " > ", 0
Msg2                    ; Backspace
    dt      8, " ", 8, 0
Msg3                    ; Introductory Message
    dt      "Enter 'H' for Commands", 0x00D, 0x00A, 0
Msg4                    ; "Help" Message
    dt      "The Commands are:", 0x00D, 0x00A
    dt      " H - Help", 0x00D, 0x00A
    dt      " D - Download Application", 0x00D, 0x00A
    dt      " ! [D|A] - Reset the Emulated Processor/Set the PORTA Type"
    dt      0x00D, 0x00A
    dt      " [1 [Address]] - Single Step from the Current PC Address or"
```

```
dt      " Specified Address", 0x00D, 0x00A
dt      " J [Address] - Single Step/Jump Over Call Statement", 0x00D
dt      0x00A
dt      " G [Address] - Start Application Executing", 0x00D, 0x00A
dt      " I Address - Set the Instruction Pointer ", 0x00D, 0x00A
dt      " R - Display the Primary Special Function Registers", 0x00D
dt      0x00A
dt      " S RegAddress - Show Contents of 16 Registers Starting at "
dt      "the Specified Address", 0x00D, 0x00A
dt      " E RegAddress - Load the Register with new Data", 0x00D
dt      0x00A
dt      " B [Address] - Toggle the Breakpoint Address", 0x00D, 0x00A
dt      " C - Clear all the Breakpoints", 0x00D, 0x00A
dt      " U [Address] - Display 24 Lines of Instructions", 0x00D
dt      0x00A
dt      " + [Address] - Add Instructions to the Program Memory"
dt      0x00D, 0x00A
dt      " P [E|F] - Program EEPROM or Flash PIC microcontroller MCU",
0x00D
dt      0x00A
dt      " V - Verify the Contents of the PIC microcontroller MCU",
0x00D, 0x00A
dt      0
```

and added the two statements:

```
;#CompStart
```

and

```
;#CompEnd
```

before and after the table. These keywords were used to indicate that the table between them is to be compressed with the first (least significant address) character being used as the least significant 7 bits of the instruction. The second (most significant address) character is being used as the most significant 7 bits of the instruction.

I then wrote the Visual Basic application DT Compress (which can be found in the PICDwnld\code\DT Compress folder), which converts the table between the ;#CompStart and ;#CompEnd statements to the compressed code similar to what's shown here:

```
;#CompStart
MsgTable          ; List of Messages
Msg0              ; Start with \r\n (New Line)
    dw 0x050D
Msg1              ; Put in the "Prompt" Message:
    dw 0x01000,0x0103E
Msg2              ; Backspace
    dw 0x0400,0x0420
```



```

Msg3                                     ; Introductory Message
    dw 0x02280,0x03A6E,0x03965,0x013A0,0x013C8,0x03320,0x0396F,0x021A0
    dw 0x036EF,0x030ED,0x0326E,0x06F3,0x0A
Msg4                                     ; "Help" Message
    dw 0x03454,0x01065,0x037C3,0x036ED,0x03761,0x039E4,0x030A0,0x032F2
    dw 0x06BA
    dw 0x0100A,0x01048,0x0102D,0x032C8,0x0386C,0x050D
    dw 0x02220,0x016A0,0x02220,0x03BEF,0x0366E,0x030EF,0x01064,0x03841
    dw 0x03670,0x031E9,0x03A61,0x037E9,0x06EE
    dw 0x0100A,0x01021,0x0225B,0x020FC,0x0105D,0x0102D,0x032D2,0x032F3
    dw 0x01074,0x03474,0x01065,0x036C5,0x03675,0x03A61,0x03265,0x02820
    dw 0x037F2,0x032E3,0x039F3,0x0396F,0x029AF,0x03A65,0x03A20,0x032E8
    dw 0x02820,0x0294F,0x020D4,0x02A20,0x03879,0x06E5
    dw 0x0100A,0x018DB,0x02DA0,0x03241,0x03964,0x039E5,0x02EF3,0x0105D
    dw 0x0102D,0x034D3,0x033EE,0x032EC,0x029A0,0x032F4,0x01070,0x03966
    dw 0x036EF,0x03A20,0x032E8,0x021A0,0x03975,0x032F2,0x03A6E,0x02820
    dw 0x01043,0x03241,0x03964,0x039E5,0x01073,0x0396F,0x029A0,0x032F0
    dw 0x034E3,0x034E6,0x03265,0x020A0,0x03264,0x032F2,0x039F3,0x050D
    dw 0x02520,0x02DA0,0x03241,0x03964,0x039E5,0x02EF3,0x016A0,0x029A0
    dw 0x03769,0x03667,0x01065,0x03A53,0x03865,0x0252F,0x036F5,0x01070
    dw 0x03B4F,0x03965,0x021A0,0x03661,0x0106C,0x03A53,0x03A61,0x036E5
    dw 0x03765,0x06F4
    dw 0x0100A,0x01047,0x020DB,0x03264,0x032F2,0x039F3,0x0105D,0x0102D
    dw 0x03A53,0x03961,0x01074,0x03841,0x03670,0x031E9,0x03A61,0x037E9
    dw 0x0106E,0x03C45,0x031E5,0x03A75,0x03769,0x06E7
    dw 0x0100A,0x01049,0x03241,0x03964,0x039E5,0x01073,0x0102D,0x032D3
    dw 0x01074,0x03474,0x01065,0x03749,0x03A73,0x03AF2,0x03A63,0x037E9
    dw 0x0106E,0x037D0,0x03769,0x032F4,0x01072,0x050D
    dw 0x02920,0x016A0,0x02220,0x039E9,0x03670,0x03CE1,0x03A20,0x032E8
    dw 0x02820,0x034F2,0x030ED,0x03CF2,0x029A0,0x032F0,0x034E3,0x03661
    dw 0x02320,0x03775,0x03A63,0x037E9,0x0106E,0x032D2,0x034E7,0x03A73
    dw 0x03965,0x06F3
    dw 0x0100A,0x01053,0x032D2,0x020E7,0x03264,0x032F2,0x039F3,0x016A0
    dw 0x029A0,0x037E8,0x01077,0x037C3,0x03A6E,0x03765,0x039F4,0x037A0
    dw 0x01066,0x01B31,0x02920,0x033E5,0x039E9,0x032F4,0x039F2,0x029A0
    dw 0x030F4,0x03A72,0x03769,0x01067,0x03A61,0x03A20,0x032E8,0x029A0
    dw 0x032F0,0x034E3,0x034E6,0x03265,0x020A0,0x03264,0x032F2,0x039F3
    dw 0x050D
    dw 0x022A0,0x02920,0x033E5,0x03241,0x03964,0x039E5,0x01073,0x0102D
    dw 0x037CC,0x03261,0x03A20,0x032E8,0x02920,0x033E5,0x039E9,0x032F4
    dw 0x01072,0x034F7,0x03474,0x03720,0x03BE5,0x02220,0x03A61,0x06E1
    dw 0x0100A,0x01042,0x020DB,0x03264,0x032F2,0x039F3,0x0105D,0x0102D
    dw 0x037D4,0x033E7,0x032EC,0x03A20,0x032E8,0x02120,0x032F2,0x035E1
    dw 0x037F0,0x03769,0x01074,0x03241,0x03964,0x039E5,0x06F3
    dw 0x0100A,0x01043,0x0102D,0x03643,0x030E5,0x01072,0x03661,0x0106C
    dw 0x03474,0x01065,0x03942,0x030E5,0x0386B,0x034EF,0x03A6E,0x06F3
    dw 0x0100A,0x01055,0x020DB,0x03264,0x032F2,0x039F3,0x0105D,0x0102D
    dw 0x034C4,0x03873,0x030EC,0x01079,0x01A32,0x02620,0x03769,0x039E5
    dw 0x037A0,0x01066,0x03749,0x03A73,0x03AF2,0x03A63,0x037E9,0x039EE

```



```

    rlf      EEDATA ^ 0x0100, w      ; Start with the Odd Byte
    rlf      EEDATH ^ 0x0100, w
    btfss    MsgOffset ^ 0x0100, 0   ; Odd or Even Byte?
    movf     EEDATA ^ 0x0100, w      ; Even Byte
    andlw    0x07F                   ; Convert to ASCII 7 Bits
    btfsc    STATUS, Z
    decf     MsgTemp ^ 0x0100, f     ; Decrement the Value Count
    incf     MsgOffset ^ 0x0100, f   ; Point to the Next Byte in the String
    btfsc    STATUS, Z
    incf     (MsgOffset + 1) ^ 0x0100, f
    goto     MT_Loop1

MT_Loop2                                     ; Have Correct Offset, Now, Display the
                                           ; Message
    bcf      STATUS, C               ; Calculate Address of Next Word to
    rrf      (MsgOffset + 1) ^ 0x0100, w ; Display
    addlw    HIGH MsgTable
    movwf    EEADRH ^ 0x0100
    rrf      (MsgOffset + 1) ^ 0x0100, w ; Setup Carry Correctly for
    rrf      MsgOffset ^ 0x0100, w ; Increment
    addlw    LOW MsgTable
    movwf    EEADR ^ 0x0100
    btfsc    STATUS, C
    incf     EEADRH ^ 0x0100, f      ; If Carry Set, Increment to Next Page
EEPReadMacro                               ; Now, Do the Program Memory Read
    rlf      EEDATA ^ 0x0100, w      ; Start with the Odd Byte
    rlf      EEDATH ^ 0x0100, w
    btfss    MsgOffset ^ 0x0100, 0   ; Odd or Even Byte?
    movf     EEDATA ^ 0x0100, w      ; Even Byte
    andlw    0x07F                   ; Convert to ASCII 7 Bits
    btfsc    STATUS, Z
    goto     MT_End                  ; Zero, Yes
SendCharMacro
    incf     MsgOffset ^ 0x0100, f   ; Point to the Next Byte in the String
    btfsc    STATUS, Z
    incf     (MsgOffset + 1) ^ 0x0100, f
    incf     SMCount ^ 0x0100, f
    goto     MT_Loop2

MT_End                                     ; Finished sending out the Table Data
    movf     SMCount ^ 0x0100, w     ; Return the Number of Bytes Sent
    EmuReturn

```

In this code there are three macros. The `EEPReadMacro` (along with the `EEPWriteMacro`) is used to access the Flash program memory of the PIC16F87x. `SendCharMacro` is used to poll the UART transmit holding register empty interrupt request flag (TXIF of PIR1) and send the byte when the holding register is open. The macro code is:

```
SendCharMacro Macro
    bcf      STATUS, RP1
    ifndef Debug
        btfss PIR1, TXIF
        goto  $ - 1
    else
        goto  $ + 3                ; Put in a Skip over the "nop" to save
        nop                        ; a Mouse Click
    endif
    movwf    TXREG                 ; Send the Byte
    bcf      PIR1, TXIF           ; Reset the Interrupt Request Flag
    bsf      STATUS, RP1
endm
```

and it should be noted that if the label `Debug` is defined, the polling loop is ignored because in MPLAB, the USART hardware is not simulated and execution will never fall out of this loop.

There are two other things to notice about this macro. The first is in the code that executes when the `Debug` label is defined. I kept two instructions to match the `btfss/goto` instructions of the polling loop, but I jump over the second one to save a mouse click when I'm single-stepping through the application. This might seem like a petty place to save a mouse click or two, but `SendCharMacro` is used a lot in this application, and when single-stepping through the application, skipping over the instructions seems to reduce the number of mouse clicks significantly.

The second point to notice about this macro is that it changes the operating bank from 2 to 0 and then back to 2. The EMU-II application has all its variables in Banks 2 and 3 of the PIC microcontroller. This allows the user to access almost all the registers (except for the USART specific ones) in Banks 0 and 1 without affecting the operation of the EMU-II in any way.

Instead of using the `call` and `return` instructions in the EMU-II, I used two macros, `EmuCall` and `EmuReturn`, which I wrote to implement a subroutine call that does not access the built-in program counter stack. The reason for writing these subroutines was to avoid the possibility that the subroutine calls in the EMU-II application code would affect the emulated application.

The `EmuCall` and `EmuReturn` macros are:

```
EmuCall Macro Address                ; Stackless Emulator Call
    local ReturnAddr
    movwf    tempw ^ 0x0100          ; Save the Call Value in "w"
    incf     FSR, f
    movlw    LOW ReturnAddr          ; Setup the Return Address
    movwf    INDF
    incf     FSR, f
    movlw    HIGH ReturnAddr
    movwf    INDF
    movlw    HIGH Address            ; Jump to the Specified Address
    movwf    PCLATH
```

```

    movf    tempw ^ 0x0100, w          ; Restore "w" before doing it
    goto    (Address & 0x07FF)        | ($ & 0x01800)
ReturnAddr
    movf    tempw ^ 0x0100, w          ; restore the "w" from the Subroutine
    endm

EmuReturn Macro                        ; Return from the Macro Call
    movwf   tempw ^ 0x0100            ; Save the Temporary "w" Register
    movf    INDF, w                   ; Get the Pointer to the Return Address
    movwf   PCLATH
    decf    FSR, f                    ; Point to the Low Byte of the
                                      ; Return Address

    movf    INDF, w
    decf    FSR, f
    movwf   PCL                       ; Jump to the Return Address
    endm

```

This will save the return address in a data stack implemented with the FSR register. This address is then used to return to the calling section of code.

These macros are reasonably efficient, but it should be noted that they do affect the state of the zero flag in the STATUS register and they do take up a number of instructions. The number of instructions taken up by the subroutine calls is why I created other macros, like `EEPReadMacro` and `SendCharMacro`, which actually require fewer instructions to implement the required function than the `EmuCall` macro.

The last aspect of the application that I would like to bring to your attention is how I implemented the breakpoints for application single-stepping and breakpoints. As I pointed out above, if I were to use multiple instructions for breakpoints, then code like:

```

    btfss   PIR1, TXIF                ; Poll until USART Free to Send a
    goto    $ - 1                     ; Character
    movwf   TXREG                     ; Output the character in "w"

```

will not be able to be stepped through.

The approach I took was to create a single-step (and breakpoint) mechanism that would not have problems with these situations. By limiting application size to one page, I can use a single `goto` instruction for implementing the return to the EMU-II application code.

For example, if I was single-stepping at the `btfss` instruction in the example above, the EMU-II code would put in the breakpoints shown below:

```

    btfss   PIR1, TXIF                ; Poll until USART Free to Send a
    goto    NextStep                 ; Was "goto $ - 1"
    goto    SecondStep               ; Was "movwf TXREG"

```

Now, depending on the value of `TXIF`, execution will return to the EMU-II code via the `goto NextStep` or `goto SecondStep`, which in either case is located in the instruction code area `0x700` to `0x7FF`. `NextStep` and `SecondStep` are separate

from each other in order for the correct new program counter value to be noted and recorded by the EMU-II application.

The `NextStep`, `SecondStep`, and breakpoint code is similarly designed and uses the following instructions:

```

Step #                ; "#" is from 0 to 8 for Breakpoints
    movwf    _w        ; Save the "w" Register
    movf     STATUS, w  ; Save STATUS and Change to Bank 2
    bsf      STATUS, RP1 ; Execute from Page 2 in EMU-II
    bcf      STATUS, RP0
    movwf    _status ^ 0x100
    movf     PCLATH, w   ; Save the PCLATH Registers
    movwf    _pclath ^ 0x100
    movlw    # * 2       ; Save the Breakpoint Number
    gotom    StopPoint
AddressIns #
    dw       0x3FFF      ; Address of the Breakpoint/Single Step
    dw       0x3FFF      ; Instruction at Breakpoint/Single Step

```

The two words at the end of the breakpoint are used to save the address where the breakpoint was stored and the original instruction. The breakpoint address is used to update the EMU-II's program counter along with replacing the `goto Step #` instruction with the application's original instruction.

Before any type of execution, all the enabled breakpoints are put into the application program memory. Upon completion of execution, the application program memory is scrubbed for all cases of breakpoint `gotos` and they are restored with the original instructions.

Note that when setting up single-step breakpoints, the next instruction or destination of a `goto` or `call` is given the `goto NextStep` and `goto SecondStep` breakpoints. This is possible for all instructions instead of `return`, `retlw`, and `retfie`. The reason for these three instructions to get an error message during single-stepping is that the destination cannot be taken from the processor stack. Instead of putting a breakpoint at potentially every address in the application, I decided to simply prevent single-stepping at these instructions.

As applications may be halted by pressing the Reset button in the application, when the EMU-II first boots up, the scrub operation takes place to ensure that there are not any invalid `gotos` left in the application.

There are a few things to watch out for with breakpoints and interrupts. For most application debugging, I do not recommend setting breakpoints within the interrupt handler. The reason for this is to avoid any missed interrupt requests or having multiple requests queued up in such a way that the application's mainline never returns. I originally thought that this was a limitation of the EMU-II, but I tried some experiments with MPLAB-ICD and found that it also has similar issues. Interrupt handlers should always be debugged as thoroughly as possible using a simulator so as to not miss or overflow on any interrupt events and requests.

This is not to say that simple applications (such as just waiting for a `TMR0` overflow) cannot be used with the EMU-II. In testing out the application, I did work through a

```

001 zdc 00 movlw 0x0FF > U
movlw 0x0FF
movwf PORTB
clrf PORTA
bsf STATUS, RP0
clrf PORTB
movlw 0x0D2
movwf TMRO
bcf STATUS, RP0
movlw 0x085
movwf FSR
bsf PORTA, 0
bcf INDF, 0
movlw 0x0F6
clrf TMRO
movf TMRO, w
btfss STATUS, Z
goto 0x000F
bsf INDF, 0
clrf TMRO
btfsc PORTA, 0
goto 0x0014
comf TMRO, w
001 zdc 00 movlw 0x0FF > _

```

Note "TMRO"
- Should be "OPTION_REG"

Figure 5.15 EMU-II unassembly display showing how a Bank 1 register is displayed with a Bank 0 label.

number of experiments with interrupt handlers without problems. There is one point I should make clear to you: breakpoints should *never* be enabled in both an interrupt handler and mainline code. If an interrupt breakpoint is executed while a mainline breakpoint is being handled by the EMU-II, the mainline breakpoint context registers will be changed to the values of the interrupt handler. The execution of the application may also become erratic.

If you are debugging an application that requires breakpoints in both the interrupt handler and mainline code, I recommend setting only one at a time and using the C (breakpoint all clear) command before setting the next breakpoint.

The Emu-II includes a simple disassembler for reviewing the source code. A typical unassembled application is shown in Fig. 5.15, and there are two things I want to point out about the disassembled function.

The first point to make about the disassembled code is the lack of useful labels. If you were to look at the disassembled code you would see that constant and variable names are not output which makes it much more difficult to read.

```

movlw 0x0FF
movwf PORTB           ; Turn off all the LED's
clrf  PORTA           ; Use PORTA as the Input

bsf   STATUS, RP0     ; Have to go to Page 0 to set Port
                        ; Direction

```

```

    clrf    TRISB & 0x07F          ; Set all the PORTB bits to Output
    movlw   0x0D2                  ; Setup the Timer to fast count
                                      ; Put in Divide by 8 Prescaler for 4x

    movwf   OPTION_REG & 0x07F    ; Clock
    bcf     STATUS, RP0            ; Go back to Page 0

    movlw   TRISA                  ; Have to Set/Read PORTA.0
    movwf   FSR

Loop:

    bsf     PORTA, 0               ; Charge Cap on PORTA.0
    bcf     INDF, 0                ; Make PORTA.0 an Output
    movlw   0x0100 - 10           ; Charge the Cap
    clrf    TMR0                  ; Now, Wait for the Cap to Charge
Sub_Loop1:                          ; Wait for the Timer to Reach 10
    movf    TMR0, w               ; Get the Timer Value
    btfss   STATUS, Z             ; Has the Timer Overflowed?
    goto    Sub_Loop1            ; No, Loop Around again

    bsf     INDF, 0               ; Now, Wait for the Cap to Discharge
    clrf    TMR0                  ; and Time it.
Sub_Loop2:                          ; Just wait for PORTA.1 to go Low
    btfsc   PORTA, 0
    goto    Sub_Loop2

    comf    TMR0, w               ; Get the Timer Value

```

This is an excellent example of why I prefer only using source code enabled development tools. Trying to get the function of the application from Fig. 5.15 is just about impossible, but when you look at the source, the function that it implements—potentiometer measuring code with an RC delay circuit—is quite obvious.

The second problem with what is pointed out in Fig. 5.15 is that the disassembler doesn't know what bank is currently executing. In Fig. 5.15, you should see that the TRISB register, when it is enabled for all output, is referenced as PORTB (line 5 of the code). This is not a big problem and one that I can usually work my way through without any problems.

What I find to be very confusing in Fig. 5.15 is the identification of TMR0 when I want OPTION_REG (or OPTION as it is displayed by the EMU-II). As you step through the application, you will discover that the instruction on the prompt line will display the instruction based on the state of the RP0 bit of the emulated device's STATUS register.

While application code can be downloaded to the Emu-II, it cannot be uploaded into a PC. This was specifically not implemented to discourage the practice of modifying an application in the emulator and then uploading the hex file into the host PC and replicating the application from this source. This is a very dangerous practice and should be avoided at all costs to prevent the proliferation of executable code without supporting application code.

The Emu-II is probably the most involved application that you will find in this book. I am pleased with the way it came out and it is a very interesting project and tool to have while learning about the PIC microcontroller. I don't think that it is adequate as a professional development tool due to the lack of a source code interface, but for very simple applications this emulator can be an invaluable tool for you to learn about the PIC microcontroller.

Other Emulators

While there are a number of very good PIC microcontroller emulators designed and marketed by third parties, there hasn't been the same explosion of designs as with PIC microcontroller programmers by hobbyists. The reason for this really comes down to the complexity of work required to develop an emulator circuit along with the difficulty of developing a user interface for them. The Emu-II is a very simple example of how an emulator could work with a very basic user interface that does not contain many of the features I would consider critical for using an emulator in a professional environment.

A professional product would require a bondout chip and a source/listing file interface for the user to use it effectively. Other features would include providing a high-speed interface to avoid the time required to download application hex files into the Emu-II.

If you are interested in designing your own full emulator, you will have to talk to your local Microchip representative to find out about entering into a nondisclosure agreement (NDA) with them to learn more about the options Microchip has for developing emulators for their products. Microchip does make bondout chips available for all of the PIC microcontroller part numbers, but technical information about them is considered proprietary and not for general release.

Partial emulators (like the Emu-II) are still a lot of work to get running, but designing them gives you a much better appreciation of how the PIC microcontroller works. If you are interested in designing your own, please take a look at the code in the Emu-II to see how the various issues of executing an application from within a PIC microcontroller monitor are handled.

Having said there are few commercial emulators available, I should point out that there are a number of products you can buy. These commercial emulators provide wide ranges of services and can be bought for a few hundred dollars up to \$1,000 or more for a full bondout chip-based complete system.

This page intentionally left blank

THE MICROCHIP PIC MCU PROCESSOR ARCHITECTURE

When starting to research a new device, whether it is a microcontroller such as one of the many PIC® MCU part numbers or a simple logic gate, one of the first things that you have for reference information is the block diagram. A well-drawn block diagram, such as the one taken from the PIC16C61 microcontroller datasheet in Fig. 6.1, actually has all the information that you need to understand and start working with the chip. Unfortunately, block diagrams can be intimidating and confusing when you look at them for the first time. I dare say that many people will skip right over them without spending any time trying to understand how data flows and what features are available on the chip, and this is unfortunate because there is a wealth of information that will help you to visually understand and remember how the chip works and allow you to follow the flow of data through the chip.

The purpose of this chapter is to help you to understand how programs execute and manipulate data in the PIC microcontroller's processor. To do this, I am going to rely on the block diagram that is available in each of the PIC MCU datasheets. I have found that this is a very useful way of going about the task of learning about the processor and how it works because the diagram is an architectural drawing of its inner workings. When you understand the block diagram, you will understand how each instruction executes, and this will give you some insights into how to structure your assembly-language programs so that they are as efficient as possible.

The processor block diagrams are very similar for each of the PIC microcontroller processor families (consisting of the low-end, mid-range, PIC17, and PIC18 architectures), and for this reason, I will explain the mid-range devices in detail and then review the architectural differences in the other PIC MCU families. These differences mainly center on how data is accessed in different register banks, how intrapage jumps and calls are executed, and how data is indexed and stored in stacks. Even without these sections explaining the differences in the mid-range PIC microcontroller architecture, you should be able to understand how these processors work by reviewing the block diagram at the start of the datasheets.

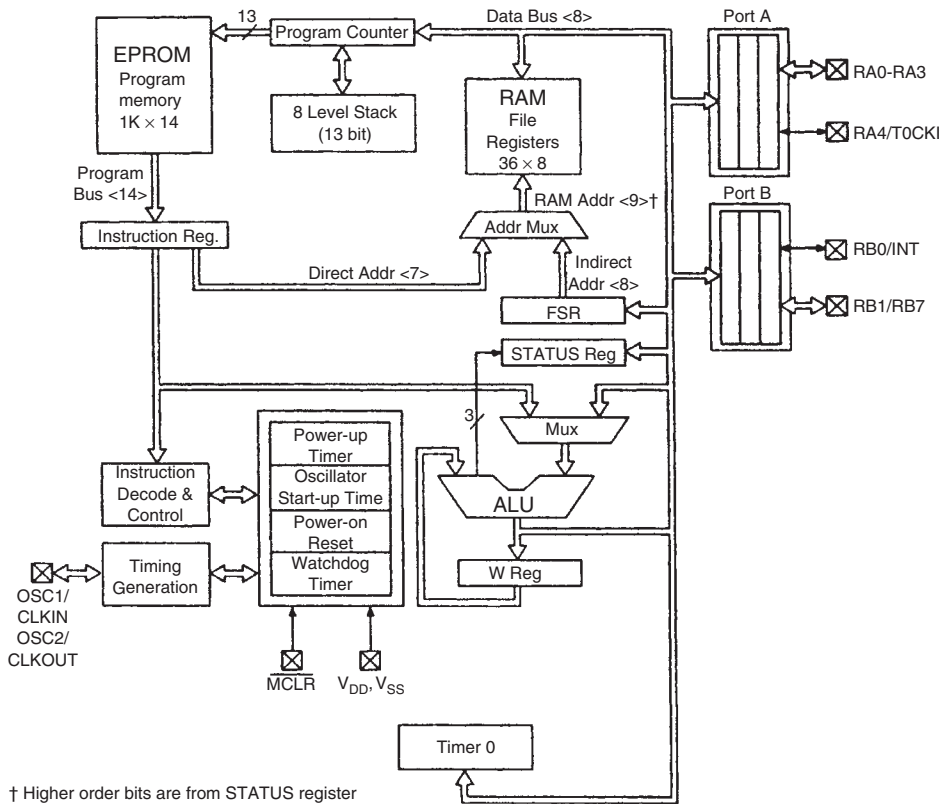


Figure 6.1 PIC16C61 block diagram.

The CPU

In the microchip datasheets you will find that the PIC microcontroller's processor is described as a "RISC-like architecture . . . separate instruction and data memory (Harvard architecture)." In this chapter I want to explain what this means for people who do not have Ph.D.s in computer architectures, as well as help explain how application code executes in the PIC MCU processor. The processor may seem to be very complex and different from other devices you've worked with before, but I believe that it is very intelligently designed and works in a very logical manner. Despite the complex written description of the processor, you will discover that it is actually quite straightforward and designed to simplify the implementation of many complex applications and programming algorithms.

The PIC microcontroller processor can be thought of as being built around the *arithmetic/logic unit* (ALU), which provides basic arithmetic and bitwise operations for the processor. There are a number of specific-use registers that control operation of the CPU as well as input/output (I/O) registers and data-storage (RAM) registers. In this book I call the

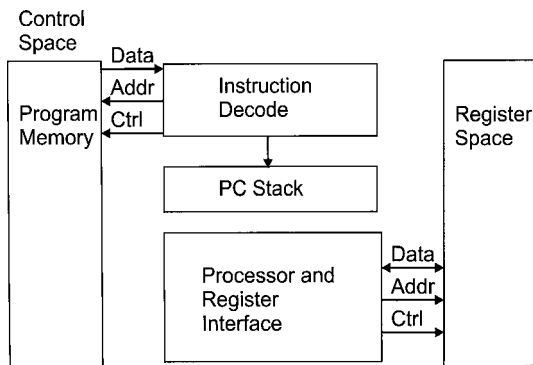


Figure 6.2 Harvard architecture block diagram.

specific-use registers *hardware registers* or *I/O registers* depending on the function they perform. The hardware registers also allow direct manipulation of functions that usually are invisible to the programmer, such as the program counter, to allow for advanced program functions. Data-storage (RAM or variable) registers are called *file registers* by Microchip.

The registers are completely separate from the program memory and are said to be in their own “spaces.” This is known as *Harvard architecture* and is shown in Fig. 6.2. In the figure, note that the program memory and the hardware to which it is connected are completely separate from the register space. This allows program memory reads for instructions to take place while the processor is accessing data and processing it. This capability allows the PIC microcontroller to execute software faster than many of its contemporaries.

Instruction execution takes place over four clock cycles, as shown in Fig. 6.3. During an instruction execution cycle, the next instruction to be executed is fetched from program memory. When the next instruction is executing, the processor is fetching the next instruction after it. After an instruction has been fetched and is latched in a

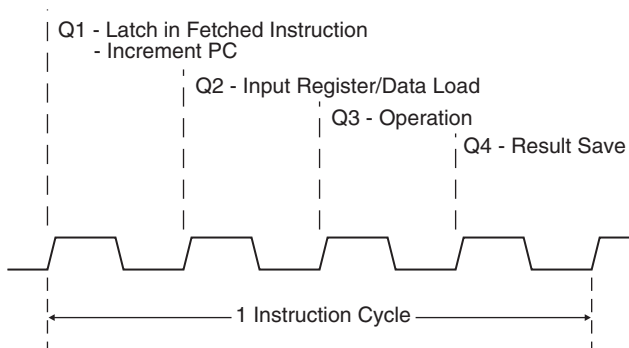


Figure 6.3 Four clock cycles, each performing its own task, make up a single instruction cycle.

holding/decode register, the program counter (used to address which instruction is being executed) is incremented. This is known as Q1. Next (Q2), data to be processed (often with the data in the accumulator or “working” register, which will be described below) is read and put into temporary buffers. During Q3, the data-processing operation takes place. Finally, the resulting data value is stored during Q4, after which the process repeats itself for the next instruction (which is put into the holding register while the current instruction is executing). These four cycles that take place with each “tick” of the clock are known collectively as an *instruction cycle*.

Since the instruction cycle is made up of the four Q cycles, which is equivalent to four clock cycles, the instruction execution speed is said to be one-quarter the clock speed. For example, an application that has a 4-MHz clock would be running 1 million instruction cycles per second (MIPS). In the PIC18 processors, there is a built-in phased-locked loop circuitry that multiplies the external clock’s speed four times. This means that for PIC18 chips with the phased-locked loop active, the instruction cycle is equal to the chip’s clock.

There are three primary methods of accessing data in the PIC microcontroller. *Direct* addressing means that the register address within the register bank (explained below) is specified in the instruction. If a constant is going to be specified, then it is specified *immediately* in the instruction. The last method of addressing is to use an *index* register that points to the address of the register to be accessed. *Indexed addressing* is used because the address to be accessed can be changed arithmetically. In other processors, there are additional methods of addressing data, but in the PIC microcontroller, these are the only three.

When accessing registers in the mid-range PIC microcontrollers directly, 7 address bits are explicitly defined as part of the instruction. These 7 bits result in the ability to specify up to 128 addresses in an instruction, as shown in Fig. 6.4.

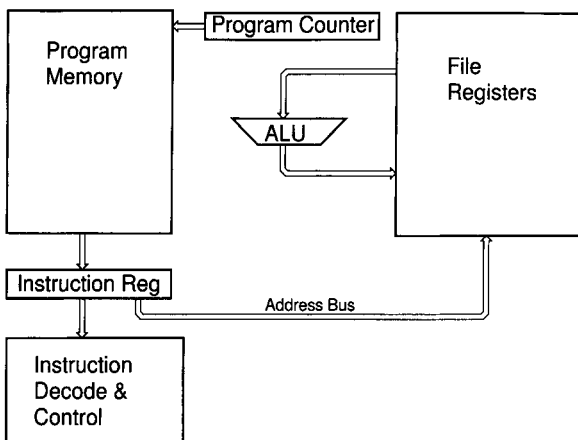


Figure 6.4 Basic PIC microcontroller processor architecture.

These 128 register addresses are known as a *bank*. To expand the register space beyond 128 addresses for hardware and variable registers, Microchip has added the capability of accessing multiple banks of registers, each capable of registering 128 addresses in the mid-range PIC microcontrollers. The low-end PIC microcontrollers can access 32 registers per bank, also with the opportunity of having four banks accessible by processor for up to 128 register addresses in total. This will be explained later in this chapter, along with how register addressing is implemented for the PIC17C and PIC18C processors.

The ALU shown in Fig. 6.4 is an acronym for the *arithmetic/logic unit*. This circuit is responsible for doing all the arithmetic and bitwise operations, as well as the conditional instruction skips implemented in the PIC microcontroller's instruction set. Every microprocessor available today has an ALU that integrates these functions into one block of circuits. The ALU will be discussed later in this chapter.

The *program counter* maintains the current program instruction address in the *program memory* (which contains the instructions for the PIC microcontroller processor, each one of which is read out in sequence and stored in the *instruction reg* and then decoded by the *instruction decode and control* circuitry).

The program memory contains the code that is executed as the PIC microcontroller application. The contents of the program memory consist of the full instruction at each address (which is 12 bits for the low-end, 14 bits for the mid-range and 16 bits for both the PIC17 and PIC18 devices). This differs from many other microcontrollers in which the program memory is only 8 bits wide, and instructions that are larger than 8 bits are read in subsequent reads. Providing the full instruction in program memory and reading it at the same time result in the PIC microcontroller being somewhat faster in instruction fetches than other microcontrollers.

The block diagram in Fig. 6.4, while having 80 percent or more of the circuits needed for the PIC microcontroller's processor is not a viable processor design in itself. As drawn in Fig. 6.4, there is no way to pass data to the program memory for immediate addressing, and there is no way to modify the program counter. As I work through this chapter, I will be fleshing out Fig. 6.4 until it is a complete processor that can execute PIC microcontroller instructions.

To implement two-argument operations, a temporary holding register, often known as an *accumulator*, is required to save a temporary value while the instruction fetches data from another register or is passed a constant value from the instruction. In the PIC microcontroller, the accumulator is known as the *working register* or, more commonly, as the *w register*. The w register really cannot be accessed directly as a register address in itself in the low-end and mid-range PIC microcontrollers. Instead, the contents must be moved to other registers that can be accessed directly. The w register can be accessed as an addressed register in the PIC17 and PIC18 devices. Every arithmetic operation that takes place in the PIC microcontroller uses the w register. If you want to add the contents of two registers together, you would first move the contents of one register into the w register and then add the contents of the second to it.

The PIC microcontroller architecture is very powerful from the perspective that the result of this operation can be stored either in the w register or the source of the data.

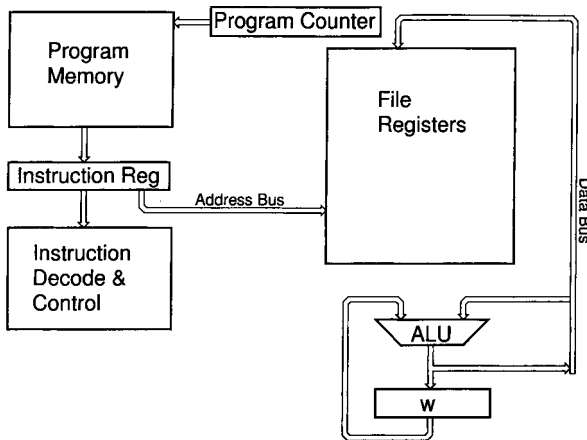


Figure 6.5 PIC microcontroller processor architecture with the w register and file registers as source and destination for ALU operations.

Storing the result back into the source effectively eliminates the need for an additional instruction for saving the result of the operation. There is a lot of flexibility in how instructions are executed to provide arithmetic and bitwise operations for an application.

Adding the w register changes how the ALU is wired in the PIC microcontroller processor block diagram, as shown in Fig. 6.5. Note that the ALU has changed to a device with two inputs (which is the case in the actual PIC microcontroller's ALU) and that the contents of the w register are used as one of the inputs. You also should note that when a result is passed from the ALU, it could either be stored into the w register or in one of the file registers. This is a bit of foreshadowing of one of the most important features of the PIC microcontroller architecture and how instructions execute.

Figure 6.5 shows the PIC microcontroller at its simplest level. This simple circuit can execute well over half the PIC microcontroller's instructions.

Hardware and File Registers

If you have worked with other processors and computer systems, you probably will be surprised by the close coupling and shared memory space of the PIC microcontroller's processor's registers, hardware I/O registers, and variable RAM. This is a result of the small (5-bit addressing for low-end devices and 7-bit addressing for mid-range devices) register space accessible to the processors. Despite being somewhat unusual, this close coupling of registers for both variable storage and hardware I/O registers provides you with a common means of accessing, processing, and updating the contents of registers, regardless of their function, using a single set of tools.

In the mid-range PIC microcontroller, each instruction that accesses a register contains the addresses within the given bank with a maximum bank size of 7 bits, which allows up to 128 different addresses. In each bank, the registers fall within four distinct groups:

TABLE 6.1 BASE REGISTER ADDRESSES BY PIC MICROCONTROLLER ARCHITECTURE FAMILY

REGISTER	LOW-END	MID-RANGE	PIC17	PIC18
WREG	Not accessible	Not accessible	0x0A	0xFE8
STATUS	0x03	0x03	0x04/0x06	0xFD8
PCL	0x02	0x02	0x02	0xFF9
PCLATH	Page bits in STATUS	0x0A	0x0e	0xFFB/0xFFA
FSR	0x04	0x04	0x01/0x09	0xFEA-0xFE9
INDF	0x00	0x00	0x00/0x08	0xFE7

- Processor registers
- I/O hardware registers
- Variable memory
- Shared or “shadowed” variable memory

The processor registers consist of STATUS, PCL, PCLATH (from mid-range devices), FSR, INDF, and WREG (for high-end devices). These registers are always at the same addresses within the different PIC microcontroller families. These addresses are listed in Table 6.1. These registers can be accessed from within any of the register banks.

The I/O hardware registers consist of the OPTION, TMRO, PORT, I/O PINS and enable registers, INTCON, and other interrupt control and flag registers, along with any other hardware features built into the particular PIC microcontroller. The important difference between these registers and processor registers is that except for INTCON, these registers are bank-specific, and while some conventions are used for the placement of these functions, for part numbers, and for specific functions, the registers are located in different addresses. The registers with conventions are listed in Table 6.2.

As time goes on and more features become standard, you’ll probably see the mid-range PIC microcontrollers standardize on a 32-byte processor and I/O hardware register block (also known as the *special function registers*, or SFRs) at the start of each bank.

Above the processor and I/O hardware registers, are the *file registers*, or variable memory. This memory can be bank-specific or shared between banks. In all PIC microcontrollers, there are a number of bytes that are always available (shared, or what I call *shadowed*) across all the register banks. This memory is used to pass data between the banks or, as I prefer to use them, to provide a common variable for sharing context register data during interrupts without having to change the bank specification in the status register. The shared memory is PIC microcontroller part number-specific and can be common across all banks or pairs of banks.

In the low-end PIC microcontrollers, many devices have multiple banks, but these multiple banks are strictly for providing additional file registers. Normally in these

TABLE 6.2 I/O REGISTER ADDRESSES BY PIC MICROCONTROLLER ARCHITECTURE FAMILY				
REGISTER	LOW-END	MID-RANGE	PIC17	PIC18
OPTION	Uses OPTION Instruction	0x81	0x05	0xFD0
TMR0	0x01	0x01	0x0B/0x0C	0xFD7/0xFD6
PORTC-PORTA	0x07–0x05	0x07–0x05	Varies by part number	0xF82–0xF80
TRISC-TRISA	Uses TRIS port Instruction	0x87–0x85	Varies by part number	0xFD4–0xFD2
PORTD/TRISD	Not available	0x08/0x88	Varies	0xF83/0xFD5
PORTE/TRISE	Not available	0x09/0x89	Varies	0xF84/0xFD6
INTCON	Not available	0x0B	0x07	0xFF2
OSCCAL	0x05	Varies by part number	Not available	Varies by part number

PIC MCUs, the first 16 addresses of each bank (address 0 to 0x00F) are common, with the upper 16 bytes of each bank having file registers that are specific to them.

BANK ADDRESSING

One of the most difficult concepts for most people to understand when they first start working with PIC microcontrollers is the register banks used in the different PIC microcontroller architectures. The number of registers available for direct addressing in the PIC microcontroller is limited to the number of address bits in the instruction that are devoted to specifying register access. In low-end PIC microcontrollers there are only 5 bits (for a total of 32 registers per bank), whereas in mid-range PIC microcontrollers there are 7 bits available for a total of 128 registers per bank. The PIC18 can access 256 register addresses, but each bank is 128 registers in size.

In order to provide additional register addresses, Microchip has introduced the concept of *banks* for the registers. Each bank consists of an address space consisting of the maximum size allowable by the number of bits provided for the address. When a mid-range application is executing, it is executing out of a specific bank, with the 128 registers devoted to the bank directly accessible.

In each PIC microcontroller, a number of common hardware registers are available across all the banks. For mid-range devices, these registers are INDF and FSR, STATUS, INTCON (presented later), PCL, and PCLATH (also discussed later). These registers can be accessed regardless of the bank that has been selected. Other hardware registers may be common across all or some of the banks as well. In all mid-range PIC microcontrollers there are common file registers that are common across banks to allow data to be transferred across them.

Bank 0	Bank 1
Addr - Reg	Addr - Reg
00 - INDF	80 - INDF
01 - TMRO	81 - OPTION
02 - PCL	82 - PCL
03 - STATUS	83 - STATUS
04 - FSR	84 - FSR
05 - PORTA	85 - TRISA
06 - PORTB	86 - TRISB
07 -	87 -
08 - EEDATA	88 - EECON1
09 - EEADR	89 - EECON2
0A - PCLATH	8A - PCLATH
0B - INTCON	8B - INTCON
0C - 4F Shared - File Regs	8C - CF Shared - File Regs
50 - 7F - Unused	D0 - FF - Unused



Shaded areas indicate
unused registers
- 0x000 Returned
when these
registers are
read

Figure 6.6 PIC16F84 register map.

In Fig. 6.6, the PIC16C84's register space is shown for bank 0 and bank 1. When execution has selected bank 0, the PORTA and PORTB registers can be addressed directly. When bank 1 is selected, the TRISA and TRISB registers are accessed at the same address as PORTA and PORTB when bank 0 is selected.

To change the current bank out of which the mid-range application is executing, the RPx bits of the STATUS register are changed. To change between bank 0 and bank 1 or bank 2 and bank 3, RP0 is modified. Another way of looking at RP0 is that it selects between odd and even banks. RP1 selects between the upper (bank 2 and bank 3) and lower (bank 0 and bank 1) bank pairs. For most of the basic mid-range PIC microcontroller applications presented in this book, you will only be concerned with bank 0 and bank 1 and RP0.

At the risk of getting ahead of myself, the TRIS registers are used to specify the input or output operation of the I/O port bits. When one of the TRIS register bits is set, the corresponding PORT bit is in *input mode*. When the TRIS bit is reset, then the PORT bit is in *output mode*. To access the PORT bits, bank 0 must be selected, and to access the TRIS bits, bank 1 must be selected.

For example, to set PORTB bit 0 as an output and load it with a 1, the PIC microcontroller code would execute as

```
PORTB.Bit0 = 1;           // Load PORTB.Bit0 with a "1"
STATUS.RP0 = 1;           // Start Executing out of
Bank 1
TRISB.Bit0 = 0;           // Make PORTB.Bit0 Output
STATUS.RP0 = 0;           // Resume Execution in Bank 0
```

Microchip specifies that bank 1 registers are defined with the same address as bank 0 registers but with bit 7 set in their address specification. This means that for the mid-range PIC microcontrollers, bank 0 register addresses are in the range of 0 to 0x7F, whereas

bank 1 register addresses are in the range of 0x80 to 0xFF. Once the RP0 bit is set to select the appropriate bank, the least significant 7 bits of the address are used to access a specific register. This can be very confusing—the reason for having this specification is the FSR (index pointer) register, which is 8 bits in size. The FSR can access registers in both banks transparently. The Microchip TRISB register has the address value 0x86, which has bit 7 set and is in bank 1. PORTB has an address value of 0x006 and can only be accessed when bank 0 is selected.

When you start working with more complex mid-range PIC microcontrollers, which use all four banks, you will see registers with address bit 8 set, which indicates that the registers are in banks 2 and 3. These registers are accessed directly using the RP1 bit (along with RP0), and the least significant 7 bits of the Microchip-specified address are used as the address.

Specifying an address with bit 7 (or 8) set will result in the following message:

Register in operand not in bank 0. Ensure that bank bits are correct.

This indicates that an invalid register address has been specified and to make sure that execution is in the correct bits. Most people clear bits 7 and 8 of the defined register address to avoid this message. This can be done by simply ANDing the address with 0x7F to clear bit 7, but a somewhat more sophisticated operation normally is performed on the address to make sure that the register is accessed from the correct bank. Instead of ANDing with 0x7F to clear bit 7 for bank 1, the address is XORed with 0x80. By doing this, if the register is supposed to be in bank 1 (bit 7 of the address is set), then it will be cleared. If the register can only be accessed in bank 0 (bit 7 of the address is reset), then this operation will result in bit 7 being set and will cause the preceding message to be given. This is a nice way to ensure that you are not accessing registers that are not in the currently selected bank.

Using the XOR operation, the preceding example becomes

```
PORTB.Bit0 = 1;           // Load PORTB.Bit0 with a "1"
STATUS.RP0 = 1;           // Start Executing out of
                           // Start Bank 1
(TRISB ^ 0x080).Bit0 = 0; // Make PORTB.Bit0 Output
STATUS.RP0 = 0;           // Resume Execution in Bank 0
```

This is also true for banks 2 and 3, which have address bit 8 set. In Table 6.3 I have listed the value of the XOR registers for specific banks. If the error message comes out of the register access, then you will know that you are accessing a register in the wrong bank. Note that the INDF, PCL, STATUS, FSR, PCLATH, and INTCON registers are common across all the banks and do not have to have their addresses XORed with a constant value to be accessed correctly.

Direct bank addressing is a very confusing concept and, unfortunately, very important to PIC microcontroller application development. I realize that it probably will be difficult for you to understand exactly what I am saying here, but it will become clearer as you work through the example application code.

TABLE 6.3 BANK ADDRESS TO “RPX” BIT SETTINGS

BANK	RP1	RP0	ADDRESS RANGE	XOR VALUE
0	0	0	0x0–0x7F	None
1	0	1	0x80–0xFF	0x80
2	1	0	0x100–0x17F	0x100
3	1	1	0x180–0x1FF	0x180

The index register (FSR), as I indicated earlier, is 8 bits in size, and its bit 7 is used to select between the odd and even banks (bank 0 and bank 2 versus bank 1 and bank 3). Put another way, if bit 7 of the FSR is set, then the register being pointed to is in the odd register bank. This straddling of the banks makes it very easy to access different banks without changing the RP0 bit. For the preceding example, if I were to use the FSR register to point to TRISB instead of accessing it directly, I could use the code

```
PORTB.Bit0 = 1;           // Load PORTB.Bit0 with a “1”
FSR = TRISB;             // FSR Points to TRISB
INDF.Bit0 = 0;           // Make PORTB.Bit0 Output
```

This ability of the mid-range FSR register to access both banks 0 and 1 is why I recommend that for many applications array variables should be placed in odd banks, and single-element variables should be placed in even banks. Of course, this is only possible if the entire file register range is not “shadowed” across the banks as in the PIC16F84 and other simple mid-range PIC microcontrollers that are used in introductory applications.

To select between the high and low banks with the FSR, the IRP bit of the STATUS register is used. This bit is analogous to the RP1 bit for direct addressing. Having separate bits for selecting between the high and low bank pairs means that data can be transferred between banks using direct and index addressing without having to change the bank-select bits for either case.

There is one thing that I have to note with regard to the FSR register and indirect addressing. Even though the FSR register can access 256 different register addresses across two banks, it *cannot* be used to access more than 128 file registers contiguously (or all in a row). The reason for this is the control registers contained at the first few addresses of each bank. If you try to wrap around a 128-byte bank, you will corrupt the PIC microcontroller’s control registers with disastrous results.

ZERO REGISTERS

I don’t really know if this qualifies as a feature, but unused registers in a PIC microcontroller’s register map will return 0 (0x00) when they are read. This capability can be useful in some applications. *Zero registers* (undefined registers that return 0 when

read) are normally defined in the Microchip documentation as *shaded* addresses in the device register map documentation.

In Fig. 6.6, the PIC16F84's register map is shown with addresses 7 (PORTC registers) in each bank shaded, indicating that they return 0 when read. Of course, when these registers are written to, their values are lost and not stored in the register. (One might say the information has gone to the "great bit bucket in the sky.")

I am hesitant to recommend using the zero registers when programming. It is important to note that in different PIC microcontrollers, the zero registers are at different locations. Because of this, if code is transferred directly from one application to another and the zero register chosen is not available in the PIC MCU destination (e.g., a valid file or hardware register is at this location), then the code will not work correctly. Instead of using a hardware zero register, I would recommend that a file register be defined and cleared for the purpose of always returning 0.

The PIC Microcontroller's ALU

The *arithmetic/logic unit*, which is labeled *ALU* in the PIC microcontroller block diagrams, performs arithmetic, bitwise, and shifting operations on 1 or 2 bytes of data at a time. These three simple functions have been optimized to maximize the performance of the PIC microcontroller and minimize the cost of building the MCUs. An in-depth understanding of the ALU's function is not critical to developing applications for the PIC microcontroller; however, having an idea of the tradeoffs that were made in designing the ALU will give you a better idea of how PIC microcontroller instructions execute and what is the best way to create your applications. In this discussion of how the PIC microcontroller's ALU operates and is designed, I have been able to encompass 27 of the 37 instructions available in the mid-range PIC microcontroller processor. Twenty-five years ago, when the PIC microcontroller was first developed, any savings in circuits used in the ALU (or anywhere else in the device) paid huge dividends in the final cost of manufacturing the device. This philosophy has been embraced in the ALUs used in the different PIC microcontroller processor architectures.

I tend to think of the ALU as a number of processor operations that execute in parallel with a single multiplexer that is used to select which result is to be used by the application. Graphically, this looks like the block diagram shown in Fig. 6.7. The STATUS register stores the results of the operations and will be described in more detail in the next section. The ALU is the primary modifier of the STATUS bits that are used to record the result of operations, as well as providing input to the data shift instructions.

The circuit shown in the block diagram Fig. 6.7 certainly would work as drawn, but it would require a large number of redundant circuits. Many of these functions could be combined into a single circuit by looking for opportunities such as noting that an **Increment** is addition by one and combining the two functions. A list of arithmetic and bitwise functions available within the PIC microcontroller, along with the combinations necessary to provide the full range of arithmetic operations, can be found in Table 6.4.

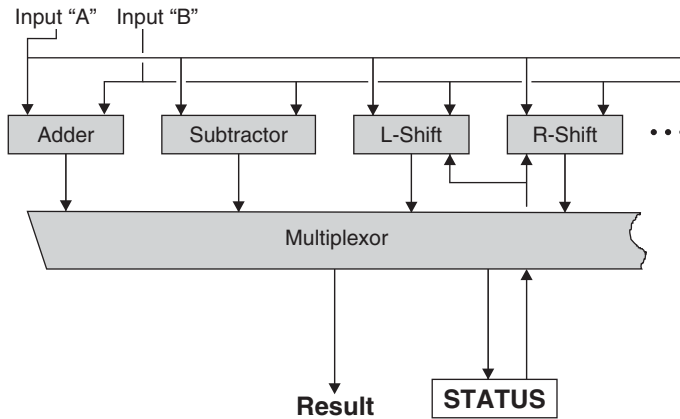


Figure 6.7 Multiplexor used to select arithmetic/bitwise operation result to output from the PIC microcontroller processor ALU.

As can be seen in this table, the 12 operations could be reduced to 6 basic operations with the constants 1 and 0xFF provided as extra inputs along with immediate and register data. Note that the basic bitwise operations (AND, OR, XOR, Shift left, and Shift right) do not have equivalencies, but this is not a problem because they are usually simple functions to implement in logic. This is not true for the arithmetic operations. For example, instead of providing a separate subtractor, the ALU's adder

TABLE 6.4 AVAILABLE PIC MICROCONTROLLER ALU OPERATIONS

OPERATION	EQUIVALENT OPERATION
Move	AND with 0xFF
Addition	None
Subtraction	Addition to value XORed with 0xFF and incremented
Negation	XOR with 0xFF and increment
Increment	Addition with one
Decrement	Addition with 0xFF
AND	None
OR	None
XOR	None
Complement	XOR with 0xFF
Shift left	None
Shift right	None

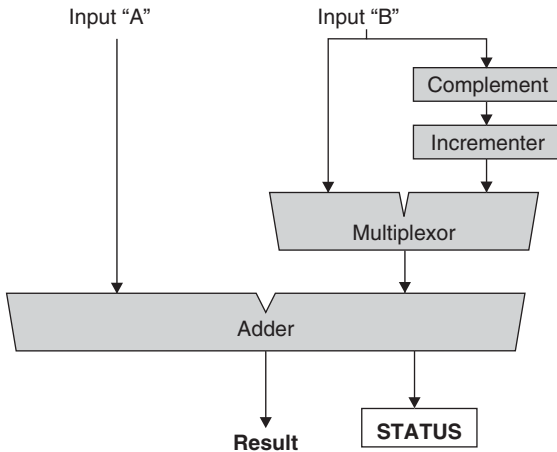


Figure 6.8 Combining operations to provide addition, subtraction, incrementing, and decrementing with a single adder.

could be used with the addition of some simple circuits to provide an addition and subtraction capability, as shown in Fig. 6.8.

I have used **Subtract** as an example here because it is an instruction that you probably will learn to hate as you start working with the PIC microcontroller. The reason for the problems with subtraction is because the result of the operation probably won't make sense to you unless you look at how the operation is carried out and how the hardware is implemented, as shown in Fig. 6.7. I will go through the subtraction instructions in more detail in the next chapter, but to introduce subtraction and help show how the PIC microcontroller's ALU works, I wanted to show how an adder with a few additional circuits could be used to provide addition and subtraction instructions using only an incrementer and a selectable negation circuit. The other instructions in the PIC microcontroller work as you would expect, and optimization of the ALU does not result in any other nonconventional instruction execution.

The circuit in Fig. 6.7 could be enhanced further by selecting between 0 (0x00) and the basic ALU Input B selection. The circuit then looks like Fig. 6.9, which can do addition, subtraction, incrementing, and decrementing. Incrementing and decrementing are carried out by selecting the 0 input and then either incrementing it (to add one to the Input A) or decrementing it (adding 0xFF or -1 to Input A). When Microchip engineers designed the PIC microcontroller's ALU, they used tricks such as this to avoid having to add redundant circuitry to the chip.

As with many microcontrollers, the PIC microcontroller instruction set has the capability of modifying and testing individual bits in registers. These instructions are not as clever as you may think and are, in fact, implemented with the base hardware I've described in this section. A bit **Set** instruction simply ORs a register with a value that has the appropriate bit set. A bit **Clear** (or **Reset**) instruction ANDs the contents of a register with a byte that has all the bits set except for the one to be cleared. I'm mentioning

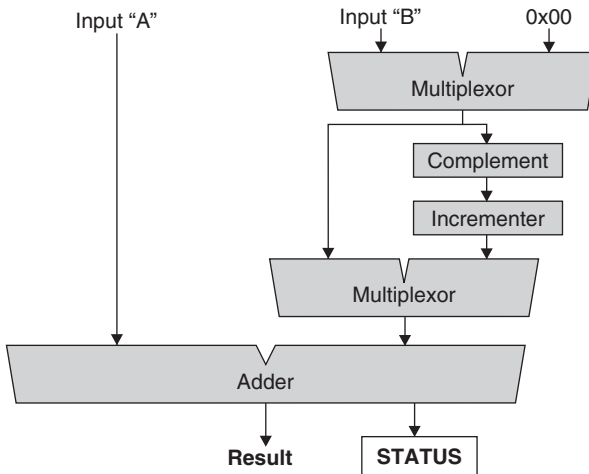


Figure 6.9 Adding multiple inputs to the Input B of the ALU adder/subtractor circuit adds the ability to increment and decrement Input A.

this here because it is important to realize that entire registers are read in, modified by the AND/OR functions, and then written back to the register. As I will show later in this book, not being aware of the method used in the PIC microcontroller for setting and clearing bits can result in some vexing problems when some applications execute.

THE STATUS REGISTER

The STATUS register is the primary central processing unit (CPU) execution control register used for recording the results of arithmetic and bitwise operations and allowing the use of this data to control the execution of application code. The operation results bit register is common to all computer processors, but the PIC microcontroller is somewhat unique in that it makes the data available to the application code, and it is used directly (not indirectly in some instructions) for program execution control. The STATUS register's organization is different in each of the PIC microcontroller architectures, but they all have the same 3 bits of data after arithmetic and bitwise operations. The 3 bits (or *flags*) that are set or reset depending on the result of the arithmetic or bitwise operation are the *carry*, *digit carry*, and *zero* bits. These bits are often referred to as the *execution status flags* (Z, DC, and C).

The zero flag (Z) is set when the result of an operation is 0. For example, ANDing 0x5A with 0xA5 is

$$\begin{aligned}
 0x05A \text{ AND } 0x0A5 &= 0b001011010 \text{ \& } 0b010100101 \\
 &= 0b000000000
 \end{aligned}$$

which will set the zero flag.

Adding 0 and 0 together obviously will produce a 0 result, but so will the addition of two values that add up to 0x100 (256). This case, that is,

$$\begin{aligned} 0x80 + 0x80 &= 0b10000000 + 0b10000000 \\ &= 0b100000000 \end{aligned}$$

produces the 9-bit result 0x100. Since all processor-accessible registers in the PIC microcontroller are only 8 bits in size, only the least significant 8 bits will be stored in the destination. These least significant 8 bits are all zeros, so the zero flag will be set as well. This may seem like I have simplified the operation of the ALU and the zero flag, but it really is as simple as to define the zero flag operation as being set if the stored result is 0 and reset if the stored result is not 0.

The carry flag (C) is set when the result of an operation is greater than 255 (0xFF), and this is meant to indicate that any higher-order bytes should be updated as well. In the preceding example (0x80 + 0x80), the result was 0x100, which stored 0x00 in the destination and set the zero flag. In this case, the ninth bit of the result (the 1) would be stored in the carry flag. If the sum were less than 0x100, then the carry flag would have been reset. Along with being used for addition, the carry flag is used for subtraction and shift instructions. The operation of the carry flag is a bit unusual for subtraction and will be discussed in more detail in later chapters, but I wanted to show its operation as it related to the carry flag to introduce you to the operation of the PIC microcontroller **Subtract** instruction.

In the preceding section I noted that subtraction actually was negative addition. For example, 1 taken away from 2 would be

$$2 - 1 = 2 + (-1)$$

For the negative number, the two's complement equivalent is calculated, which is

$$-1 = (1 \wedge 0xFF) + 1$$

Putting this value back into the preceding formula, subtraction becomes

$$\begin{aligned} 2 - 1 &= 2 + (-1) \\ &= 2 + (1 \wedge 0xFF) + 1 \\ &= 2 + 0xFE + 1 \\ &= 0x101 \end{aligned}$$

This value stored into the (8-bit) destination is 0x01 (because the register can only store 8 bits), but the ninth bit, which is used as the carry flag, is set. This means that the actual subtraction result will set the carry flag. This is different from most other processors, in which a positive (or 0) result from a subtraction operation resets the carry flag and sets it if the result is less than 0. In these processors, the carry flag becomes a *borrow* flag and indicates when a value has to be borrowed from a higher-order byte.

In the PIC microcontroller, the carry flag is really a positive flag when it comes to subtraction. If the carry flag is set, then the result is 0 or positive. If the carry flag is reset, then the result is negative. This difference from other processors can make it difficult

to port assembly-language applications directly from other processors to the PIC microcontroller. In the latest Microchip documentation, the carry flag is referred to as a *negative borrow flag* with respect to subtraction. This is a reasonable way of looking at the execution of the instruction because it is reset when a borrow from the next significant byte is required.

The digit carry flag is set when the least significant nybble (4 bits) of the result is greater than 15 after an arithmetic operation (add or subtract). It behaves identically to the carry flag, except that it is changed only by the result of the least significant 4 bits instead of the whole byte.

For example, in the operation

$$0x0A + 0x0A = 0x14$$

in the PIC microcontroller, the digit carry flag will be set (and the zero and carry flags reset). The digit carry flag may seem to be unnecessary, but as you understand the PIC microcontroller more and more, you will find opportunities where it is very useful. Later in this book I will show some examples of how it can be used and the functions that it can provide for you.

The execution status bits and how different instructions change them will be explained in more detail in Chapter 7. I should note that to change any of the three arithmetic STATUS bits from your application, a new value must be explicitly written into them (using the `movwf`, `bcf`, or `bsf` instruction). If the STATUS register is the destination of an arithmetic or bitwise operation, these bits will contain the bit values of the result of the operation, not the value expected to be stored in them.

The STATUS register can be added to the PIC microcontroller architecture block diagram to show how the results from the ALU are stored in it. Figure 6.10 shows the PIC microcontroller processor with the STATUS register being written to by the ALU.

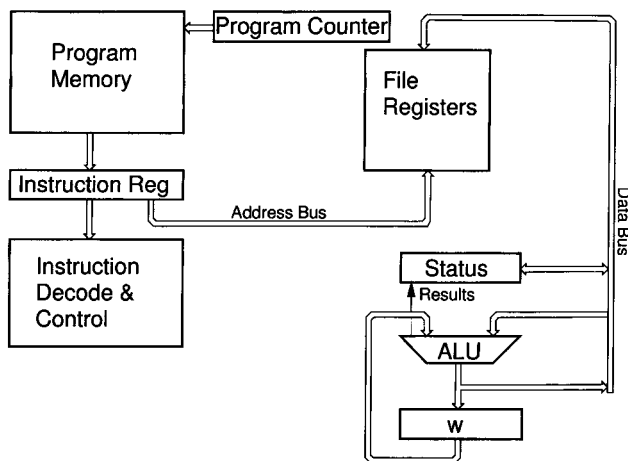


Figure 6.10 PIC microcontroller processor block diagram with the STATUS register added.

Data Movement

Earlier in this chapter I indicated that there are three methods of accessing or addressing data within the PIC microcontroller application. These three methods correspond to the traditional addressing modes presented in introductory assembly-language programming classes. While these three modes are available to you, there are features built into the PIC microcontroller that actually make the different data addressing modes much richer and will help you to create complex but efficient applications. In the following sections I want to discuss the different addressing modes and how the PIC microcontroller architecture has been designed to give much more flexibility to instruction execution than you might first suspect looking at the architecture or the instruction set.

DIRECT ADDRESSING: REGISTER READS AND RESULT SAVING

When the register address within the current bank is specified within an instruction, it is known as *direct addressing*. These instructions can be used for loading or storing data to and from, respectively, the w register, but they allow you to implement arithmetic and bitwise operations that take up less space and run in fewer cycles than similar instructions available in other processors. These arithmetic and bitwise instructions allow the result of the operation to be stored in either the w register or the source register, which often eliminates the need for an extra instruction used to store the result in the appropriate location.

Earlier in this chapter I introduced this capability as something to note in the architecture block diagrams. Looking at Fig. 6.10, you can see that the result from the ALU can be stored either back into the file registers or into the w register. When storing the result back into the file registers, the same address as the source is used for the destination. This capability gives you the option of performing an operation without changing the value saved in either the w register or the source register. The obvious use of this feature is to subtract two values without saving the result and to place the important parts of the result (the arithmetic flag registers) into the STATUS bits and ignore the result of the subtraction operation by leaving it in the w register, where it can be overwritten later.

To select where the result of an operation is saved, the last argument of a register's arithmetic or bitwise assembly-language instruction statement is either a 0 or a 1 (or w or f, respectively), as is shown in the `addwf` instruction:

```
addwf register, w|f
```

In this instruction, the contents of the w register are added to the contents of `register`. If w (or 0) is specified as the destination, then the result is stored in the w register. If f (or 1) is specified, then the result of the addition instruction is stored in `register`.

This is one of the most confusing and powerful concepts of the PIC microcontroller and can be a problem for many new PIC microcontroller programmers. The ability to immediately store an arithmetic operation's result is unusual in 8-bit processors and is not described in most beginner courses in assembly-language programming.

This feature will make applications more efficient and often simpler than what could be written in less radical processor architectures. For example, if you had to implement the statement

```
A = A + 4
```

in a typical processor, the instructions would be

```
Accumulator = 4
Accumulator = Accumulator + 4
A = Accumulator
```

If the register destination option in the PIC microcontroller is used, then the code could be simplified to

```
Accumulator = 4
A = A + Accumulator
```

If you are familiar with the C programming language, you could think of this instruction sequence as the statement

```
A += 4;      // Add 4 to the value of "A"
```

In this example, by simply storing the addition result back into the source register, I decreased the space and cycles required for implementing the `A = A + 4` statement in the PIC microcontroller assembler by one-third over what would be expected in other devices. When I write PIC microcontroller assembly language, I continually look for opportunities to save the result in one of the parameters instead of saving it temporarily in the `w` register and then providing an explicit `Store` instruction.

IMMEDIATE DATA VALUES

If you are new to microcontroller programming, you might have asked yourself how exactly are constants loaded into an application. In a PC program, you could load memory addresses with the value to be used in an operation and then read them back during program execution. It is also possible to use this method in some microcontrollers in which the program memory can be accessed by the ALU during execution. The PIC microcontroller's processor does not have the ability to read directly from its program memory, which means that the only method for using constant values in a program is to include them as part of an arithmetic or Boolean operation instruction. Providing a constant value in an instruction is known as *immediate addressing*. To provide immediate addressing in the PIC microcontroller architecture, a multiplexor is placed before the ALU to select the data source from either the 8 least significant bits of the instruction or the registers of the PIC microcontroller (Fig. 6.11).

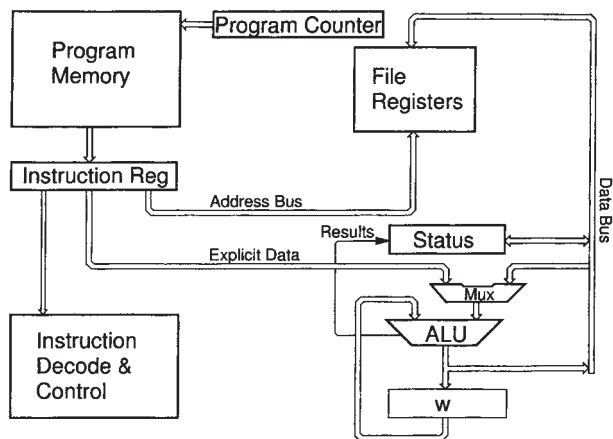


Figure 6.11 PIC microcontroller processor block diagram modified to allow immediate data values that are part of the instruction.

INDEXED ADDRESSING

There will be instances in applications where the ability to address a register directly or to specify a constant value immediately will not be sufficient, and some method of arithmetically specifying an address will be required. To do this, the processor has to calculate the address of the register to be accessed; in the PIC microcontroller, this *indexed addressing* is carried out by loading the FSR register with the address you want to access. This 8-bit register has some bank considerations for data movement.

The contents of the FSR register are multiplexed with the 7 immediate address bits, as shown in Figure 6.12. The format for using indexed addressing is somewhat

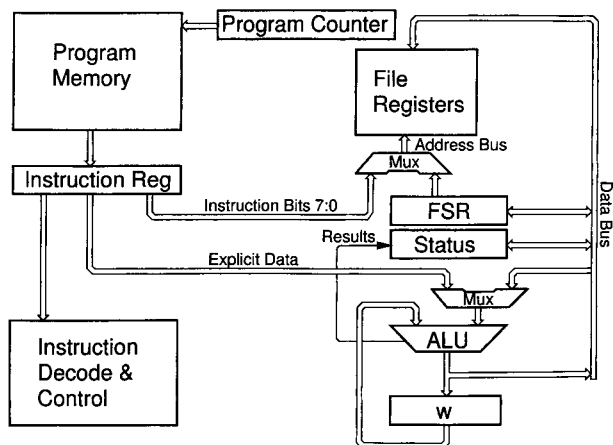


Figure 6.12 Allowing the FSR register to be used in selecting the address of the file register is known as *indexed addressing*.

different from that in other processors with which you may be familiar. In other processors, accessing the address pointed to by the index register is implemented by modifying the index register (the register that contains the address to be accessed) such as enclosing it in parenthesis or brackets. For example, you will see instructions like

```
Move Accumulator, (Index)      ; Load the Accumulator
with the                        ; data at the address pointed to
; data at the address pointed to ; by "Index"
```

Specifying indexed addressing in the PIC microcontroller is accomplished by accessing the INDF register, which is a *phantom register* and does not have any physical hardware. Instead, when the INDF address is accessed, the index or FSR register is selected to provide the address into the register space, as shown in Fig. 6.13. The INDF/FSR mechanism is used in all the different PIC microcontroller processor architectures, even though the memory spaces they are accessing are different.

Indexed addressing typically is described in high level languages as specifying the index to an *array* variable. This method of addressing may be called *array addressing* because the array variable simply may be known as an *Array*. Adding 1 to an array variable could be written out as

```
Array[ Index ] = Array[ Index ] + 1;
```

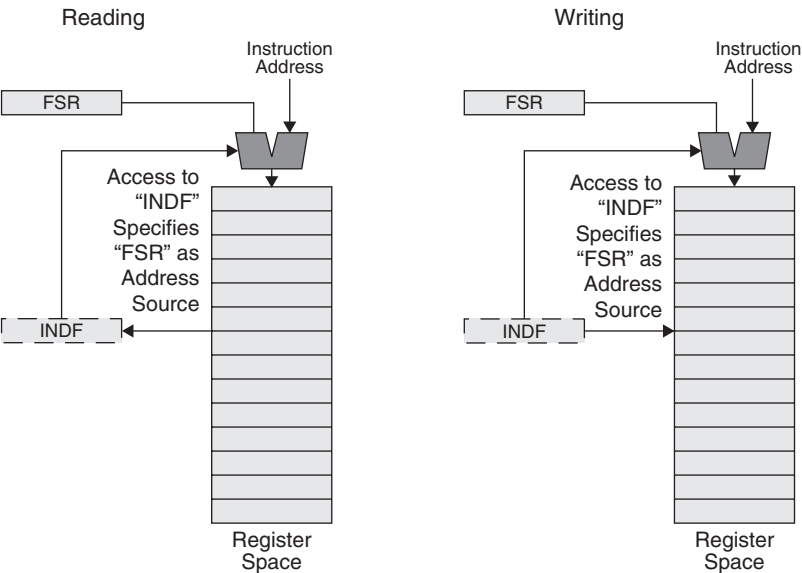


Figure 6.13 To access indexed data pointed to by the FSR register, the INDF register is used to select the FSR register to provide the address instead of the least significant 7 bits of the instruction as in direct addressing.

The start of the array variable is the label `Array`, whereas the byte (or *element*) within it is the `Index`.

When specifying the array variable and element in the PIC microcontroller, the offset to the start of the array variable has to be added to the element number to get the register address within the PIC microcontroller. Thus, to carry out the array increment operation shown above in the PIC microcontroller, the following steps would have to be taken:

```
w = Index;
w = w + Array;           // The Element Address is the Index
                        // into the
// array variable added to the start of the
// array variable
FSR = w;                 // Load the Index register with the
                        // Element Address
// Address
INDF = INDF + 1;        // Increment the Element Address
```

This example is fairly simple. Accessing array variables that have elements that are larger than 1 byte or cases where the destination is not the same as the source (and a constant isn't added to them) make the operations of the PIC microcontroller somewhat more complex.

Single-byte, single-dimensional arrays can be implemented quite easily, as can multidimensional arrays. Multidimensional arrays are treated like single-dimensional arrays, but the index is calculated arithmetically from each parameter (i.e., the index for element 3, 5 in an 8×8 array would be $2 * 8 + 5$).

The Program Counter and Stack

Understanding how the program counter (PC) works and taking advantage of its design to provide conditional execution in your application code are two of the more difficult things you will have to learn with the PIC microcontroller. Looking across the different families of PIC microcontroller devices, implementing `gotos`, `calls`, and `table writes` (writing to the program counter registers directly) will seem inconsistent and difficult to understand. Actually, these operations work according to a similar philosophy in the different architectures, and once you understand it, they really won't seem all that scary. In this section I explain how the program counter works in mid-range devices, and later in this chapter I will discuss the minor differences in the program counters used in the other PIC microcontroller families.

The mid-range's program counter can be represented by the block diagram in Fig. 6.14. When you see this diagram for the first time, it probably will seem very complex. As I work through this section, I will explain how the different parts of the program counter work and how they interrelate. When I discuss the low-end and PIC18 program counters, I will present similar block diagrams for you to work through.

In all PIC microcontroller devices, instructions take one word or address. This is part of the Reduced Instruction Set Computing (RISC) philosophy that is used for the design. This may mean that there is not sufficient space in a `goto` or `call` instruction for the

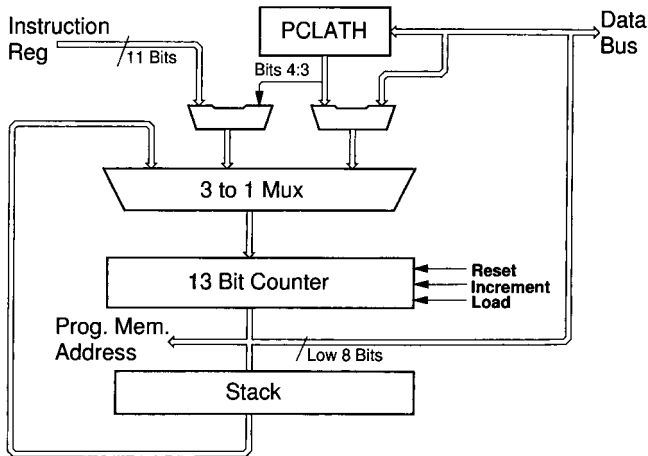


Figure 6.14 Mid-range PIC microcontroller program counter and stack subsystem block diagram.

entire address of the new location of the program counter. A certain number of the address's least significant bits are put in the instruction. These bits in the instruction are directly related to the page size of the PIC microcontroller.

Tables are an important feature of the PIC microcontroller that allow for conditional jumping or data access. Many of the applications presented in this book use tables for user interfaces or conditional execution. Tables are code artifacts in which the program counter is written to force a jump to a specific location in program memory. The least significant 8 bits of the program counter can be accessed by application software via the PCL register. Writing to these bits will change the program counter to a new value. When the 8 least significant bits are written to the PCL, the remaining, more significant, bits are taken from the PCLATH register and concatenated to the 8 bits written to PCL. The value in the PCLATH register is written into the program counter any time PCL is changed. This is also true for `goto` and `call` instructions, but it works somewhat differently in these cases.

To demonstrate how this works, you could consider the example of wanting to jump to address 0x01234 within a mid-range PIC microcontroller's program memory using a direct write to the program counter. First, the value 0x012 is written into the PCLATH register. Next, the value 0x034 is written into the PCL register. When the write to the PCL register is made, the upper bits of the program counter are loaded from the PCLATH register. This operation could be modeled as

```
PCLATH = 0x012;    // Set the PCLATH Value
PCL = 0x034;        // Change the Program Counter
                    // Program Counter = (PCLATH << 8) +
                    // PCL
                    //           = (0x012 << 8) + 0x034
                    //           = 0x01200 + 0x034
                    //           = 0x01234
```

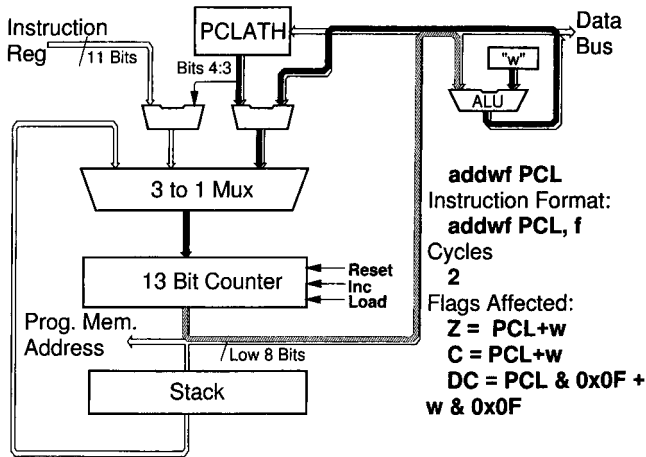


Figure 6.15 **addwf PCL, f instruction operation.**

Another way of approaching how the write to the PIC microcontroller's program counter is to look at the block diagram of the PIC microcontroller program counter hardware to see how the data flows from the processor into the program counter. In Fig. 6.15, the **addwf PCL, f** instruction, which adds the current value in PCL to the contents of the **w** register and puts the result back into the program counter, is shown. In the diagram, you can see that the PCLATH bits are combined with the data coming out of the ALU after the addition operation and then are passed back to the 13-bit counter (the actual PIC microcontroller program counter) through the 3-to-1 mux (multiplexor).

When the **addwf PCL, f** instruction is executed, 8 bits of data are added to the program counter. This means that only 256 unique addresses can be accessed (they can be anywhere in the PIC microcontroller's program memory because the PCLATH register will provide the upper address bits). While a table size of 255 seems to be the maximum, there are some tricks that you can do that will increase the size significantly.

In each PIC microcontroller, a *page* is the number to the power of 2 instructions that can be conveniently jumped within using the available bits in the instruction. The page size for the low-end PIC microcontroller is determined by the 9-bit address that is embedded in the 12-bit instruction. These 9 bits can address 512 (0x0200) instructions, which is the low-end PIC microcontroller's page size. In mid-range devices, 11 bits are used for the address within an instruction, which gives the devices a 2,048 (0x0800) instruction page size. Any address within a page can be accessed directly by a **goto** or **call** instruction.

The addresses specified by **gotos** and **calls** are zero-based within the page and are not relative to the location of the **goto** or **call** instruction. This is an important point and one that can be confusing because in the assembly-language instructions; **goto** and **call** instructions can jump to instructions that are relative to the **goto** and **call** instructions without regard to the start of the page.

If addresses outside the page have to be accessed, then the new page has to be selected. In mid-range devices, the selected page is provided to the program counter by the PCLATH register. In this case, only the bits that are not specified by the `goto` or `call` instruction are added to the address that is loaded into the PIC microcontroller's program counter. The PCLATH bits that are in conflict with the instruction's address are ignored, and the instruction's address bits are used instead. For mid-range PIC microcontrollerS, this means that PCLATH bits 0 through 2 are ignored when a `goto` or `call` instruction is encountered.

Going back to the preceding example, if PCLATH were loaded with 0x012 and the instruction `goto 0x0567` were encountered, the PIC microcontroller's program counter would be loaded with 0x0567 for the 11 least significant bits, and the least significant 3 bits of PCLATH (0b0010) would be ignored:

```
PCLATH = 0x012;      // Set the Page Value
goto 0x0567           // PC = ((PCLATH & 0x018) << 8) +
                      // Address
                      // = ((0x012 & 0x018) << 8) + 0x0567
                      // = (0x010 << 8) + 0x0567
                      // = 0x01000 + 0x0567
                      // = 0x01567
```

For this example, when the `goto` instruction is executed, the PIC microcontroller's program counter will be loaded with 0x01567.

The preceding example's 0x01234 is correct because the PCL is updated directly. If a `goto 0x034` instruction were in place, then the address would jump to 0x01034 because the most significant 3 bits of the address to `goto` are equal to 0.

Thus a `goto` or a `call` typically gets its address from the instruction and the PCLATH register, as shown in Fig. 6.16. In this diagram you should see that the PCLATH register is accessed to make up the complete address but that only 2 bits (4 and 3) are used when the new address is calculated.

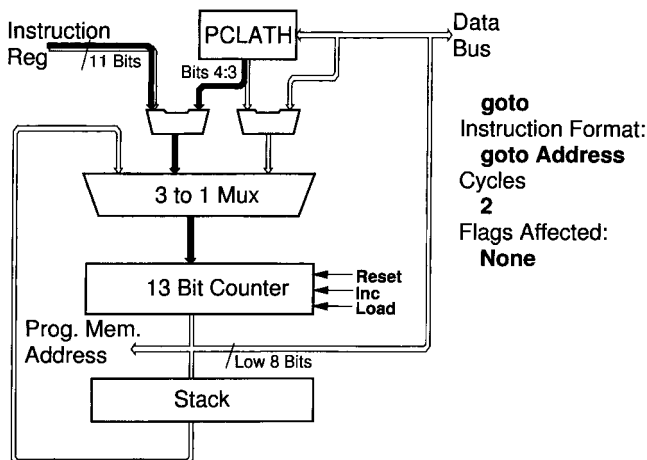


Figure 6.16 PIC MCU `goto` instruction operation.

Subroutine calls work very similarly to `gotos` or writes to the PIC microcontroller's program counter except that before the program counter is updated, it is pushed into the stack. The value pushed onto the stack is not the address of the `call` instruction but the address of the instruction *after* the `call`—which is the return address for the subroutine. In virtually all processors (the PIC microcontroller included), as soon as the instruction is fetched from program memory, the program counter is incremented. When a `call` instruction is executed, it is this incremented value that is saved on the stack, not the original value.

The PIC microcontroller's stack is a bit unusual in that it is devoted to the program counter, cannot be accessed by software, and is quite limited except in the PIC18. In most other processors, the stack is part of variable memory and can be accessed by the application code. By placing the stack in variable memory, almost infinitely large stacks can be implemented, allowing such programming constructs as recursive subroutines and data pushing and popping onto and off of the stack.

These limitations of the PIC microcontroller stack mean that nested subroutine calls and nested interrupt request handlers have to be limited in an application. In addition, data will have to be stored using the FSR index register into a simulated stack. This is not really a significant problem for your application code, and as I work through the application code in this book, I will show you how to implement your own data stack for saving and passing data between subroutines.

Reset

There are six different situations that cause the PIC microcontroller's reset (hardware reinitialized and processor stopped) to become active, followed by execution restarting at the reset vector address and execution of the application again. The operation of the PIC microcontroller is almost exactly the same in the different situations, although applications may use the different reset options or check different indicators.

The six reset options are

- 1** Power-on reset (POR)
- 2** Master clear (`_MCLR`) active during operation
- 3** Brown-out detect reset (BOR)
- 4** Watchdog timer reset (WDT)
- 5** `_MCLR` reset during sleep
- 6** WDT reset during sleep

`_MCLR` is the PIC microcontroller's negatively active master clear or reset pin. *Negatively active* means that when the pin is pulled to ground, it makes the reset circuit active, stops the internal PIC microcontroller oscillator, reinitializes the PIC microcontroller hardware, and holds the PIC microcontroller in an inactive state until the `_MCLR` line goes high again. The typical PIC microcontroller reset circuit is shown in Fig. 6.17.

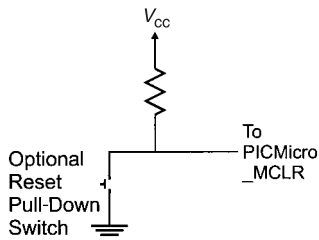


Figure 6.17 Simple external PIC reset circuit.

Many of the PIC microcontroller part numbers released in the past few years have optional internal reset circuitry, which eliminates the need for the circuit shown in Fig. 6.17. When this feature is enabled, the requirement for external reset circuitry is eliminated, but the MCLR pin typically is available only for use as an input pin. The reason for this limitation is the need for the MCLR pin to be used as the programming reset pin, and it may have high voltages (explained later in the book) applied to it during programming.

When power is applied to the PIC microcontroller and reset becomes disabled, the PIC microcontroller will begin its power-up sequence before starting to execute the application code. The most important aspect of the power-up sequence is the startup of the PIC microcontroller's clock and internal reset release, as shown in Fig. 6.18. After 1024 cycles (and an optional PWRTE internal 72-ms delay), the application code begins to execute at the *reset vector*. The reset vector will be discussed in more detail below.

The brown-out detect reset (BOR) is a function that is built into some PIC microcontrollers in which the reset circuit is activated when the input power drops below 4.0 V, or 1.8 V for low-voltage operations. This feature typically is used with battery-powered applications in which V_{CC} is not regulated.

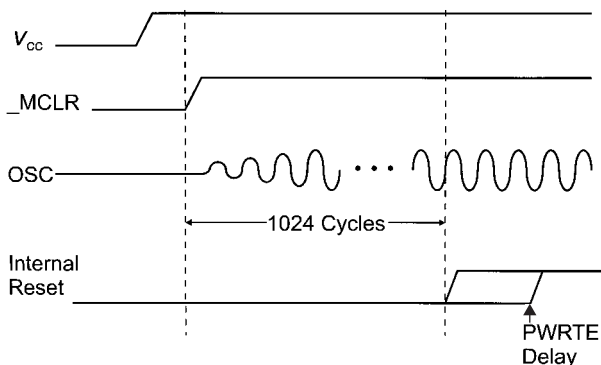


Figure 6.18 PIC microcontroller reset waveforms.

Sleep is a low-power state in which the PIC microcontroller executes the `sleep` instruction (explained in the next chapter) and stops running applications and stays dormant until it is reset or an external event causes it to restart. Sleep can be turned off, and the PIC microcontroller will resume execution by a `_MCLR` reset, a watchdog timer reset, TMRO interrupt, or external interrupt request. The last hardware feature that can cause a reset is the watchdog timer (WDT). This timer must be reset within a specified interval or the PIC microcontroller will be reset automatically. The purpose of the watchdog timer is to reset the PIC microcontroller when it has been upset by an external event and is unable to execute any further.

When the PIC microcontroller resets, 2 bits in the STATUS register and 2 other bits in the optional PCON register will change state. The PCON register is available on later-designed PIC microcontrollers and makes it much easier to determine the cause of a reset. The 2 bits affected by the reset in the STATUS register are `_TO` and `_PD`. `_TO` is active (low) when the watchdog timer has caused a reset. `_PD` is active (low) when the reset takes place after sleep. The `_PCON_BOR` register is active when a brown-out reset has occurred. And the PCON `_POR` bit is active when the reset follows the PIC microcontroller being powered up. Table 6.5 shows how these bits are set for the six different reset situations.

On *any* reset, file registers have the same values as they had before reset, and the hardware registers are given their power-up settings. This means that the I/O pins are returned to input, and peripheral functions are disabled. To restore operation after reset, you may have to save the hardware register content values before the expected reset operation so that they can be restored later.

The w register and file register contents on power-up are undetermined and can be any value. When you work with MPLAB and other simulators, these values generally are 0, which will lead to problems if they are not *initialized*. If an `_MCLR` or WDT reset occurs, the file register contents are the same as before the reset. This allows you to determine the reset type by placing a known value into these registers and checking them immediately following reset. These issues will be addressed in more detail later in this book.

TABLE 6.5 RESET EVENT TO STATUS BIT VALUES

EVENT	<code>_TO</code>	<code>_PD</code>	<code>_POR</code>	<code>_BOR</code>
Power on	1	1	0	1
Brown out	1	1	1	0
<code>_MCLR</code> reset	Unchanged	Unchanged	1	1
Watchdog timer reset	0	1	1	1
<code>_MCLR</code> during sleep	1	0	1	1
Watchdog timer reset during sleep	0	0	1	1

The *reset vector* is the program memory address the application starts executing after reset. For mid-range and PIC18 devices, this address is 0 (0x0000). For low-end PIC microcontrollers, the reset vector is the highest address of the program memory (i.e., for a 512 instruction device, this is address 511 decimal or 0x01FF). Most people leave the low-end PIC microcontroller reset vector address unprogrammed (0x0FFF, which is `xorlw 0x0FF`) and let the program counter roll over to 0 and start executing the application from there as if the reset vector were address 0 (like the other PIC microcontrollers). Ignoring the reset vector will make low-end devices behave just like the other devices and avoid issues with working with the reset vector when porting applications between PIC microcontroller families.

Interrupts

One of the things I've discovered as I get more experienced developing software applications for microcontrollers is when to use interrupts appropriately. I, like most other people, was reluctant to use interrupts in my applications when I first started out—they seemed like they were complex and difficult to work with. Over time and with experimentation with them, I found that they are actually quite easy to work with and can make many applications much simpler, so much so that I started using them in every situation. The strategy of always looking toward using them had its own pitfalls, and as applications became more and more complex, it can be very difficult to time the applications properly to ensure that each interrupt gets serviced correctly and in a timely basis. Today, I feel that I have developed good strategies for deciding when it is appropriate to apply interrupts to an application and when to use time in-line I/O operations, which I will share with you later in this book. Before I can discuss strategies for attacking problems in applications, I will discuss how interrupts are handled in the PIC microcontroller and how service routines (or handlers) are created.

Owing to the operation of the PIC microcontroller, some interrupt-requesting events are coming into the processor all the time. These requests are a result of TMR0 overflowing during normal operation, PORTB input pins changing state, and so on. In fact, many of the peripheral hardware events don't have a completion bit or flag; instead, they rely on the *interrupt request flag* to indicate that the operation has completed or the input event has occurred. Elsewhere in this book I describe these bits as the F bits because their labels always end in F.

To have these interrupt-event requests passed to the PIC microcontroller processor, the *interrupt request enable* bit (which I call the E bit) specific for the interrupt-event request has to be set along with the GIE bit of the INTCON register. For the three basic interrupts in the PIC microcontroller (TMR0 overflow, RBO/INT pin state change, and PORTB pin change), the E and F flags in the INTCON register must be set. Other interrupt-event E and F flags can be located in the PIR and PIE registers or in peripheral control registers depending on the peripheral requesting the interrupt and the PIC microcontroller part number.

When the interrupt request comes in, the F bit is set halfway through the current instruction cycle. If the GIE bit is set, then on the next instruction, instead of executing the next instruction, the address of the next instruction (or destination address) is saved in the program counter stack, and execution jumps to address 0x0004 (for mid-range PIC microcontrollers), which is the interrupt vector. At this time, the GIE bit is reset, preventing any other interrupts from being acknowledged.

The code starting at address 0x0004 is known as the *interrupt handler* or *interrupt service routine*, and its purpose is to respond to the incoming event, reset the interrupt-requesting hardware and prepare it for requesting another interrupt event, and reset the interrupt-controller hardware. For many interrupt events, all that is required to reset the requesting hardware and the interrupt controller is simply to reset the F bit requesting the interrupt. During the interrupt handler, GIE is reset, which prevents other interrupt events from interrupting the interrupt handler, which could cause problems with the PIC microcontroller having to handle a nested interrupt.

Execution continues from here until the `retfie` (return from interrupt) instruction, which sets the GIE bit again to allow additional interrupts to execute and returns the PIC microcontroller’s program counter to the address after the interrupt was acknowledged. This entire process is shown in Fig. 6.19.

In this figure you can see the different aspects of the interrupt handler’s execution. There are a few things to notice in this diagram. The first is the two instruction cycles required for the jump to the interrupt handler and the two cycles required for the `retfie` instruction to execute. As discussed in earlier sections, when a jump takes place in the

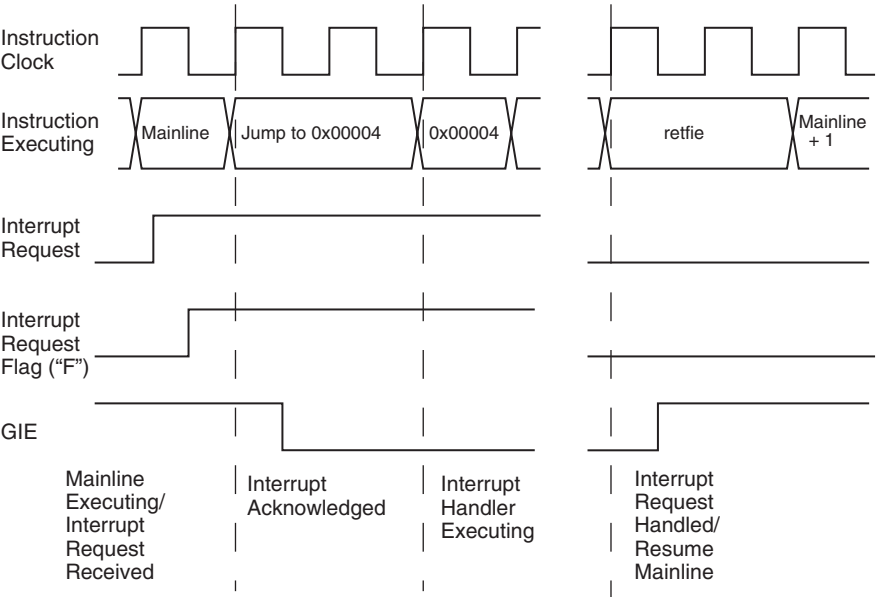


Figure 6.19 Mid-range PIC microcontroller response to an interrupt request.

PIC microcontroller, two cycles are required to flush the prefetch buffer and to load in the next instruction before it can be executed.

Along with looking at the two-instruction cycle operation of the execution changes, note that the jump to the interrupt vector cannot take place until the current instruction has finished executing. This is important because it means that the timing for the interrupt handler is not 100 percent predictable. The operation of the interrupt handler will lengthen by one instruction cycle if a `call`, `jump`, or PCL update is taking place when the interrupt request comes in. In these cases, the jump to the interrupt handler will have to wait for the two-cycle instruction to complete before the jump can take place, and this results in a maximum four-instruction-cycle interrupt latency instead of the best-case situation of three-instruction-cycle latency.

I am mentioning this because where you are most likely to see a difference in the response to an interrupt request is in the MPLAB simulator, where the jump to the interrupt handler may happen one cycle later than you expect. Not expecting this can cause you to look through the code, trying to find the reason for this anomaly. There are very few cases where the one-instruction-cycle delay will be a problem, but when you are first working with the PIC microcontroller, this can cause you some confusion.

If you forget to reset the F flag, or if another interrupt event requests the interrupt handler before the current request has completed, you will find that execution will not seem to return to the mainline. Instead, immediately following the `retfie` instruction, the first instruction of the interrupt handler (at address 0x0004) will be executed. You can “starve” the mainline code of cycles if interrupt requests come in continuously before the handler returns to the mainline code or if the interrupt request flags are not reset.

Starving the mainline of instruction cycles owing to interrupt operation is something that is very hard to find when you are debugging your application. In fact, I would consider it to be one of the hardest problems to find for someone new to the PIC microcontroller because the simulator probably will not show what happens with the volume of requests (especially if they come from peripherals).

Architecture Differences

The four different PIC microcontroller architectures have a number of similarities, and many of their differences are more a result of the instruction word sizes than of features added or deleted in the processor itself. The mid-range PIC microcontroller architecture is the most popular at the time of this writing because of the number of packages and pin configurations, as well as the variety of peripherals available. The low-end architecture has the basic features of the mid-range architecture, although it does not have the peripheral options and interrupt capabilities. The high-end architecture is represented by the PIC18, with more memory, a more sophisticated processor, and peripherals. The PIC17 architecture is an older architecture that did not gain wide acceptance, and there are no plans by Microchip to develop new part numbers for it; the PIC17 is not recommended as a platform for new applications. As I will point out elsewhere in

this book, I have focused on the mid-range PIC microcontroller processor architecture because it has the most commonality with the other architectures.

The primary differences you must understand before working with different PIC microcontroller architectures are

- 1 Program counter circuitry
- 2 Register organization

So far in this chapter I have focused exclusively on the mid-range PIC microcontroller architecture and how it operates. For the remainder of this chapter I will discuss the differences in the architectures of the low-end and PIC18 microcontrollers. The PIC17 architecture is also discussed briefly.

LOW-END DEVICES

The low-end PIC microcontroller devices have a very similar architecture to that of the mid-range devices, although it is missing some of the features of mid-range devices. The most obvious omission is lack of the `addlw` and `sublw` instructions, but there are some other, subtler differences as well that you will have to deal with.

One of these differences is a change in the reset vector compared with mid-range PIC microcontrollers. In the mid-range devices, reset is always 0, but in low-end devices, this address is always the last address in program memory. Table 6.6 lists the reset vector addresses for different low-end devices’ program memory sizes.

I recommend ignoring the reset vector address and instead treating the reset vector address as address 0, which will be the next instruction executed after the instruction at the real reset vector and the processor’s program counter rolls over and continues from 0. If the instruction is left unprogrammed, then it will be executed as the instruction `xorlw 0xFF`, which essentially negates the initial contents of the `w` register—which are unknown because the value in the `w` register is undefined at power-up, as are all the other file registers. By ignoring the last instruction, you are allowing applications to be written very similarly to mid-range applications and not have any differences in regard to reset. It is important to remember that this last address must be left unprogrammed with no instructions placed in it.

TABLE 6.6 LOW-END PIC MICROCONTROLLER PROGRAM MEMORY SIZE TO RESET VECTOR ADDRESS	
PROGRAM MEMORY SIZE	RESET VECTOR
512 instructions	0x1FF
1024 instructions	0x3FF
2048 instructions	0x7FF

In the following two sections I will describe the differences in the program counter hardware and the register addressing hardware between low-end and mid-range devices. These are the major differences between the two architectures (along with the availability of interrupts in the mid-range). Later in this book I will discuss strategies for writing applications in such a way that moving code and full applications between the two architectures is relatively simple.

Register access I consider the register organization of the low-end PIC microcontrollers to be the largest differentiator between them and the mid-range devices. The use of a 32-bit bank with no bank select bits considerably reduces the possible number of file registers and the usability of (relatively) large tables in low-end PIC microcontrollers. While I am disappointed by how few file registers are available and the difficulty in accessing what is available, I do think that low-end PIC microcontroller's are usable and should be considered when specifying which PIC microcontroller to use in an application. The low-end register space is shown in Fig. 6.20.

The low-end PIC microcontroller's TRIS and OPTION registers can be written to only using the `tris` and `option` instructions. These instructions are explained in detail in Chapter 7, but note that I write them in lower case—this is done to differentiate them from the TRIS and OPTION registers, which are denoted by writing the register labels in upper case.

Low-end instructions only provide 5 bits for a register address in a direct-addressing instruction and take the form

INSTRTdRRRRR

Bank 0	Bank 1	Bank 2	Bank 3	
Addr - Reg	Addr - Reg	Addr - Reg	Addr - Reg	
00 - INDF 01 - TMR0 02 - PCL 03 - STATUS 04 - FSR 05 - PORTA* 06 - PORTB 07 - PORTC 08-0F Shared File Regs	20 - INDF 21 - TMR0 22 - PCL 23 - STATUS 24 - FSR 25 - PORTA* 26 - PORTB 27 - PORTC 28-2F Shared File Regs	40 - INDF 41 - TMR0 42 - PCL 43 - STATUS 44 - FSR 45 - PORTA* 46 - PORTB 47 - PORTC 28-2F Shared File Regs	60 - INDF 61 - TMR0 62 - PCL 63 - STATUS 64 - FSR 65 - PORTA* 66 - PORTB 67 - PORTC 68-8F Shared File Regs	Shared Registers
10-1F Bank 0 File Regs	30-3F Bank 1 File Regs	50-4F Bank 2 File Regs	70-7F Bank 3 File Regs	Bank Unique Registers

* - "OSCCAL" may take place of "PORTA" in PICMicros with Internal Oscillators

OPTION - Accessed via "option" Instruction

TRIS# - Accessed via "TRIS PORT#" Instruction

Figure 6.20 The low-end PIC microcontroller architecture has I/O registers at the same position within each bank as well as a common file register area.

where INSTRT is the bit pattern for the instruction, d is the destination (1 stores the result back in the register, and 0 stores the result in the w register), and RRRRR is the register address. In these direct-addressing instructions, only the registers in the first bank can be accessed. Accessing registers in other banks requires use of the FSR register.

As can be seen in Fig. 6.20, the first 16 addresses of each bank are common. The 16 bank-unique file registers are located in all the last 16 addresses of the bank. This limitation of only being able to address data 16 bytes at a time prevents the construction of arrays or other data structures that are longer than 16 bytes. Of course, you could work out an algorithm for changing the FSR's high-order bits (bits 5 and 6) to simulate an array of greater than 16 bytes, but rather than doing this, I would recommend that you go to one of the other PIC microcontroller architectures for the application instead.

There can be up to four banks in low-end devices. If 16 file register bytes are available in the last half of each bank and 8 or 9 file registers are available in the first half (depending on whether or not port C is available), the maximum number of unique file registers in the low-end PIC microcontroller is 72 or 73.

One quirk that I should point out is that the low-end PIC microcontroller's FSR register can never equal 0. Instead of ignoring unused high-order FSR bits, Microchip's designers instead elected to set them. Even if all four bank registers are used for a total of 128 FSR accessible registers, the FSR register cannot be equal to 0; the FSR register bit 7 will be set. Table 6.7 lists which bits will be set in the low-end's FSR depending on how many bank registers the PIC microcontroller has.

It can be hard to remember that the low-end's FSR register can never be 0. Chances are that you'll only remember it after you've tested the contents of FSR with an instruction sequence such as

```
movlw 0
xorwf FSR, w
```

and discovered that the result never returns 0. If you check the contents of the FSR in applications such as stacks, arrays, and circular buffers, but if you try to implement the set-bit boundaries as a way of avoiding having to check or reset bits in the FSR, then you may discover that the code is not very portable either to mid-range PIC microcontroller devices or to other low-end parts that may have fewer or more file registers, which affects the number of banks and which bits of the FSR are set.

TABLE 6.7 LOW-END PIC MICROCONTROLLER MINIMUM FSR VALUE TO NUMBER OF BANKS		
NUMBER OF BANKS	SET FSR BITS	MINIMUM FSR VALUE
1	7, 6, 5	0xE0
2	7, 6	0xC0
4	7	0x80

Program counter The low-end PIC microcontroller's program counter is quite a bit different from that of the mid-range PIC microcontroller. If you look at the standard register set for the low-end device, you'll see that there is no PCLATH register, and the page-select bits are part of the STATUS register (where the bank-select bits are in the mid-range PIC microcontroller). In addition, owing to limitations in the low-end architecture, there are some problems with being able to place and work with tables and subroutines that you should be aware of.

The differences between the low-end PIC microcontroller's program counter and that of the mid-range device are partially based on the 512 instruction page size of the low-end PIC microcontroller (the mid-range has a 2,048 instruction page size). In low-end devices, execution stays within these 512 instructions unless an interpage jump on call is executed or execution simply passes from a lower page to an upper page.

The low-end PIC microcontroller's program counter block diagram is shown in Fig. 6.21. The PA0 and PA1 bits of the STATUS register (bits 5 and 6) perform the same function as the PCLATH register of the mid-range PIC microcontrollers. Bit PA0 is used to provide bit 9 of the destination address to jump to during a `goto` or `call` instruction or when PCL is written to. Bit PA1 is address bit 10. In some low-end PIC microcontrollers, you will see bit 7 of the STATUS register being referred to as PA2. This bit is not used for addressing in any of the low-end PIC microcontrollers available at the time of this writing.

In mid-range devices, to perform a jump based on changing PCL, the following code is used:

```
PCLATH = HIGH new_address;
PCL = LOW new_address;
```

In low-end PIC microcontrollers, this operation is quite a bit more complex because while the PA0 through PA2 bits are updated, none of the other bits in the STATUS register

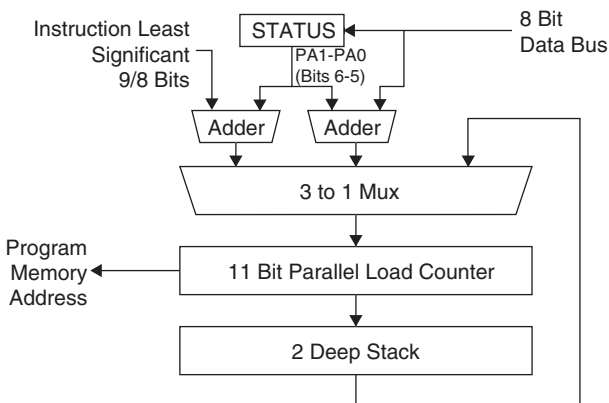


Figure 6.21 Low-end PIC microcontroller program counter block diagram.

should be changed. The equivalent low-end PIC microcontroller operation to the PCLATH/PCL updating of the program counter is

```
STATUS = (STATUS & 0x01F) + ((HIGH new_address & 0x0FE) << 4);  
PCL = LOW new_address;
```

In low-end PIC microcontroller code, the contents of the STATUS register are ANDed with 0x01F to reset the PA0 through PA2 bits. In the preceding formula that changes the STATUS register, note that I also delete the least significant bit of the upper byte of `new_address`. Bit 8 of the destination address can never be specified within the STATUS register or PCL and is always 0 as a computed address (just like it is always 0 for a `call` instruction). Once the correct bits for PA0 and PA1 have been calculated, they are added to the other bits of the STATUS register.

Note that for `call` instructions, bit 8 of the new address is always 0 because the instruction word only provides an 8-bit address, and PA0 becomes the ninth bit of the new address. This is the same as mid-range PIC microcontrollers ignoring the PCLATH bits, which are encompassed by the address within the page address in the instruction. This is not a problem for the `goto` instruction because 9 bits, which encompass a full low-end page of 512 instruction addresses, can be specified within the `goto` instruction itself. For the `call` instruction, which has only 8 address bits, the last 256 instructions of a low-end page cannot be accessed. For subroutines that are located in the second half of the 512 address page, the label will have to be located in the first 256 addresses with a `goto` to the code in the second 256 address half of the page.

Table jumps (direct writes to the PCL register) also have the same restriction as the `call` instruction addresses; they all must be in the first 256 instructions of an instruction page. I suppose that larger than 256-entry tables could be created, but they would require a bit of software to calculate the jump across page boundaries to make the table appear contiguous.

Previously in this chapter I discussed the idea that the unused RP bits of the mid-range programming could be used as temporary flags, but I didn't recommend it. Using any of the PA bits for flags in the STATUS register *never* should be done. Incorrect updates of these bits that are not returned to the correct value before the next table operation, `goto`, or `call` will result in the application jump being invalid. This will be almost impossible for you to debug, so avoid any potential problems and don't modify these bits except when you are about to change your address location.

OPTION and TRIS registers When I presented the low-end register map, you might have noticed that some of the registers discussed in previous sections of this chapter were not present; these are the `OPTION_REG` (also known as *option*) and the I/O control registers `TRISA`, `TRISB`, and so on. These registers are not addressed in the register map and cannot be accessed using the traditional register read and write instructions. Instead, specialized instructions, `option` and `tris`, must be used to access these registers directly.

There is a bit of confusion concerning these instructions, and over the past few years, there has been a change to help minimize the confusion regarding them. The `option` instruction writes the contents of the `w` register into the `OPTION` register—to avoid confusion, the `OPTION` register is identified as `OPTION_REG` in datasheets and in the register definition files. This isn't an issue for the low-end devices because the `OPTION` register cannot be accessed directly, but it does cause a problem in mid-range devices, which have both the `OPTION` register, which can be accessed like any other register, and the `option` instruction, which works identically to the low-end device's instruction. To minimize the confusion, I recommend that you always refer to the `OPTION` register as `OPTION_REG`, and I have used this convention throughout this book.

The `tris` instruction is used to copy the contents of the `w` register into the appropriate `TRIS#` register, where `#` is A, B, or C depending on the I/O port register being accessed. The difference between the instruction and the actual register is why the `TRIS` registers have not been changed like the `OPTION_REG`.

Both the `OPTION_REG` and `TRIS#` registers in the low-end PIC microcontroller architecture cannot be read back—they can only be written to. This means that some of the dynamic changes of the port I/O control bits cannot be accomplished using the bit-change instructions like they could be in the mid-range and PIC18 architectures.

PIC17Cxx ARCHITECTURE

The PIC17 architecture was developed originally before the mid-range architecture as one that could be used for advanced applications. The architecture met with limited success and has not been proliferated like the other three architectures. There are much more significant architecture differences between the PIC17 and the mid-range parts than differences between the mid-range and low-end and PIC18 architectures. The unique features of the PIC17Cxx compared with the other PIC microcontroller's include

- 1 The ability to access external, parallel memory
- 2 Up to seven I/O ports
- 3 A built-in 8×8 multiplier
- 4 Up to 902 file registers in up to 16 banks
- 5 Up to 64 kB of address space
- 6 The ability to read and write program memory
- 7 Multiple interrupt vectors

Along with these enhanced features, block diagrams of the PIC17, such as Fig. 6.22, further make you feel like the PIC17Cxx is unique and not that “portable” between the other PIC microcontroller architectures.

The important differences in the PIC17 architecture are

- 1 The `STATUS` and `OPTION_REG` register functions are spread across different registers.
- 2 The program counter works slightly differently from the other architectures.
- 3 The registers are accessed differently, and accesses can bypass the `WREG`.

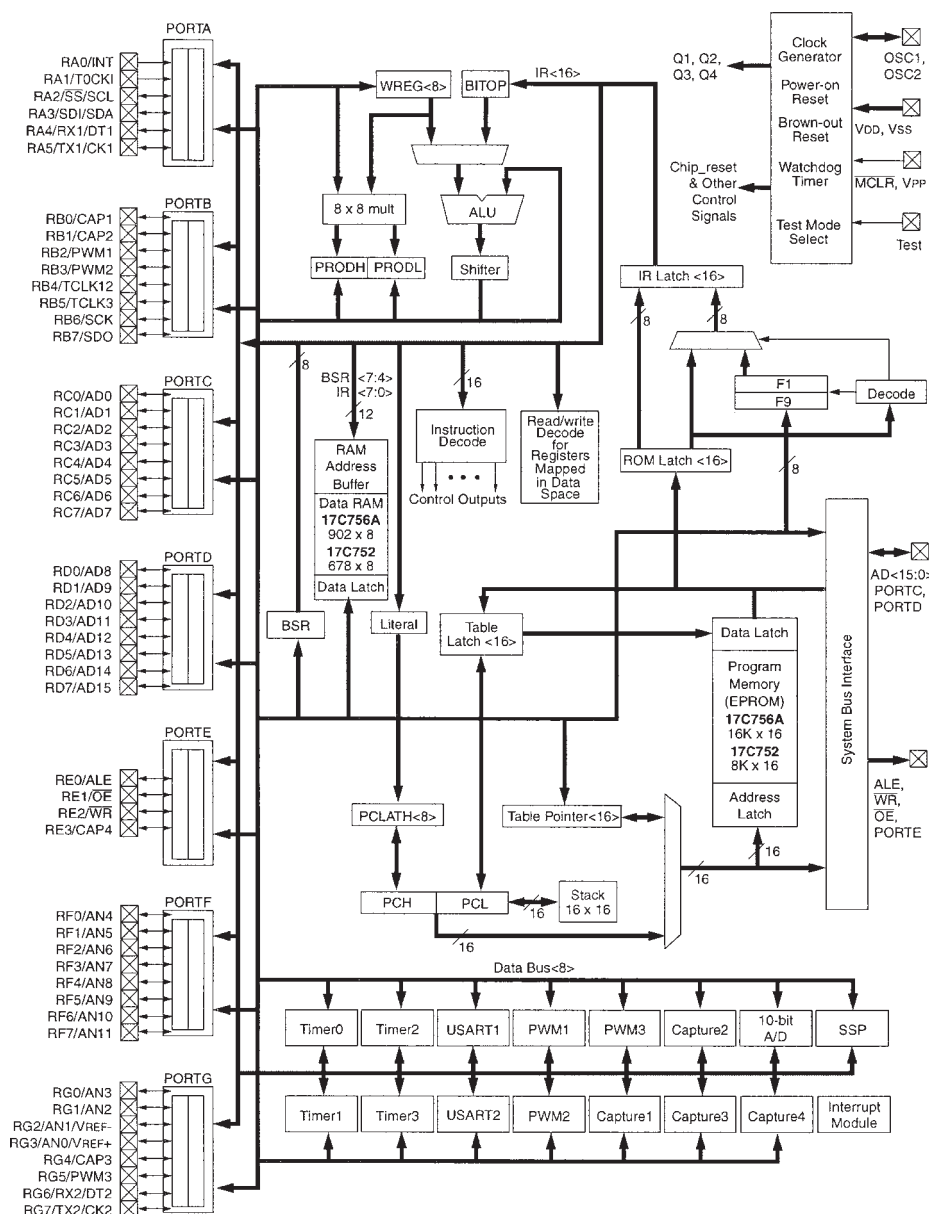


Figure 6.22 PIC17 processor and register architecture.

Interrupts in the PIC17 are similar in operation to those in the mid-range PIC microcontroller, with an E bit enabling the interrupt request flag bit (the F bit) to request that the processor execute the appropriate interrupt handler. The PIC17 does not have a GIE bit that enables interrupts but does have the GLINTD bit, which must be reset for interrupt requests to be passed to the processor—I like to think of it as the `_GIE` (negative GIE) bit.

TABLE 6.8 PIC17 INTERRUPT VECTOR ADDRESS AND PRIORITIES FOR DIFFERENT INTERRUPT REQUEST SOURCES

PRIORITY	VECTOR ADDRESS	SOURCE
High	0x08	RA0/INT pin interrupt request
	0x10	TMR0 overflow interrupt request
	0x18	TOCKI pin interrupt request
Low	0x20	Peripheral device interrupt request

Depending on which interrupt is requested and acknowledged, execution will jump to a different interrupt vector address. If multiple interrupts are requested at the same time, the highest priority one will be serviced first. The interrupts, their priorities, and their vectors are listed in Table 6.8.

The PIC17Cxx's register space is designed around a single 8-bit register address built into the instruction set. Like the low-end and mid-range PIC microcontrollers, the PIC17Cxx uses multiple register banks to allow the user to access more registers than just this base number. Unlike the low-end and mid-range PIC microcontrollers, there are two bank areas to access registers, and each one has its own set of address bits. I normally think of the PIC17Cxx's registers as being organized like Fig. 6.23.

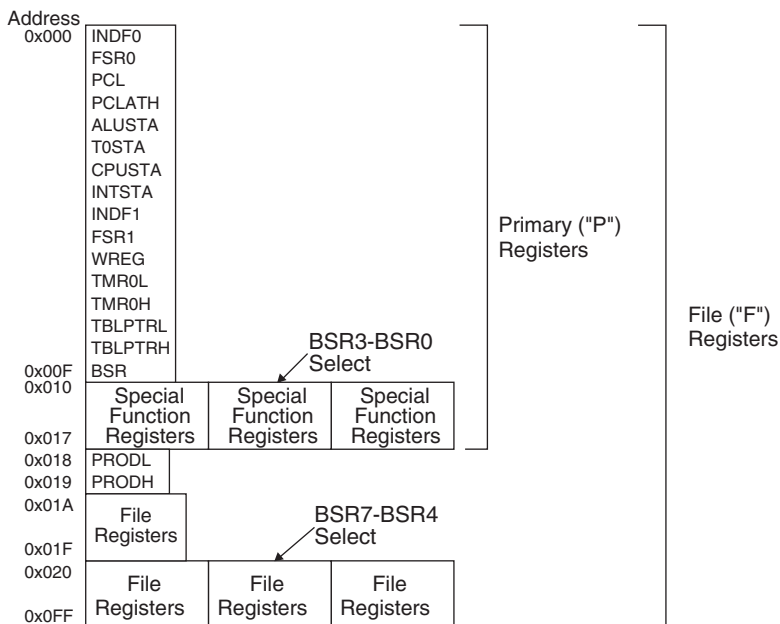


Figure 6.23 PIC17 register organization showing multiple register blocks.

The first 32 register addresses (0x000 to 0x0IF) are known as the *primary register set*. These registers are the primary processor and PIC microcontroller hardware features. The hardware interface registers are the special function registers (SFR) located at the register banks in 0x010 to 0x0iF with up to 16 banks. The P register special function register banks are selected by the least significant 4 bits of the bank select register (BSR), as I've shown in Fig. 6.23.

The 5 address bits of the primary register set and the 8 of the full register set allow data to be moved quickly and easily within the register space without having to go through the WREG. For example, moving the contents of 0x042 and 0x043 to the TMROL and TMROH registers could be accomplished by using the `movfp` instructions, which pass data from the *f* (full) register set to the primary:

```
movfp 0x042, TMROL
movfp 0x043, TMROH
```

These two instructions perform the same operations as the mid-range operations

```
temp = w;
w = contents of 0x042;
TMROL = w;
0 w = contents of 0x043;
TMROH = w;
w = temp;
```

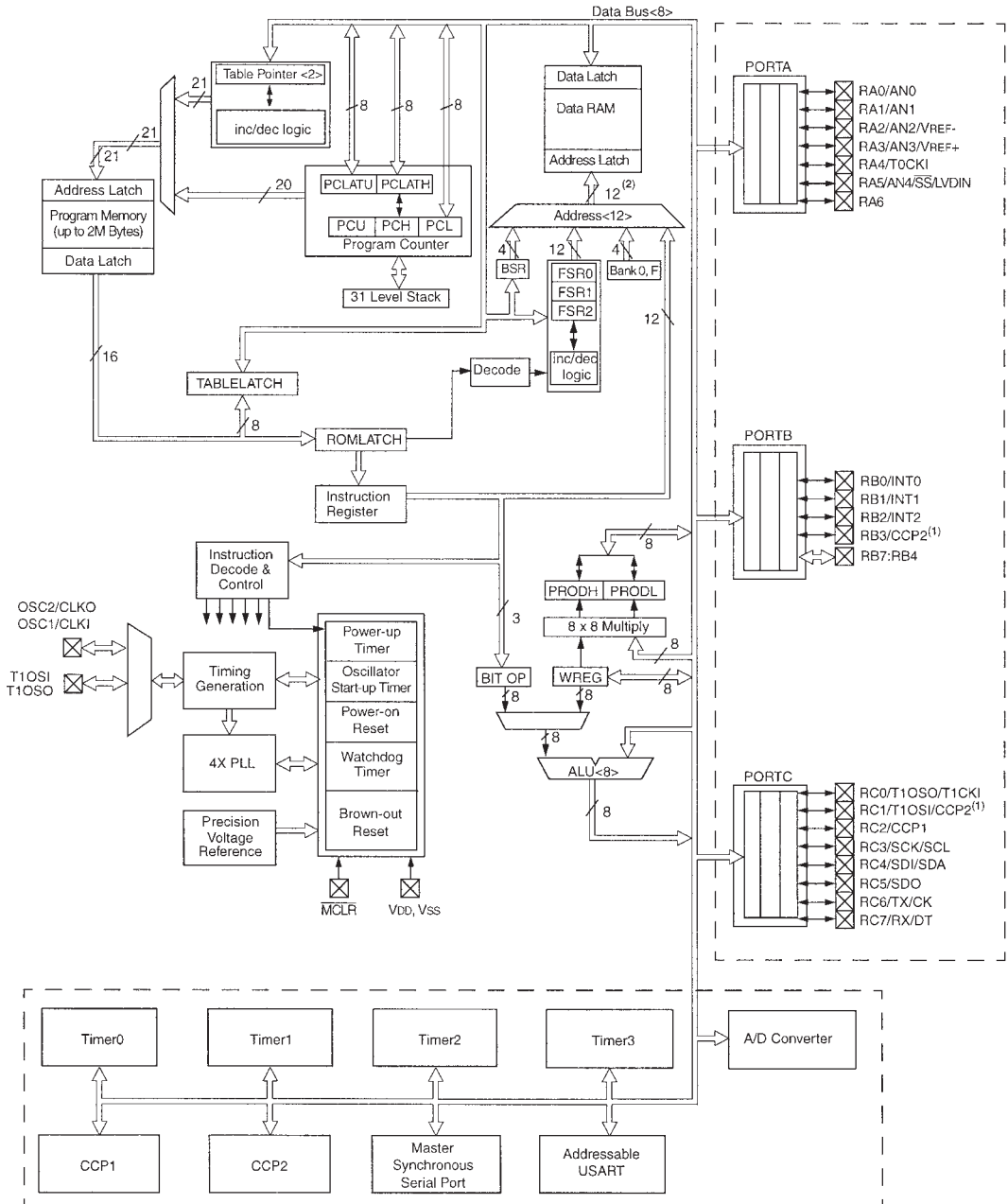
without requiring the `temp` variable to store the current copy of `w`.

The PIC17Cxx's processor can access 64 kB of 16-bit words of program memory, either internally or externally to the chip. Each instruction word is given a single address, so to address the 64 kB of words (or 128 kB), 16 bits are required. From the application developer's perspective, these 16 bits can be accessed via the PCL and PCLATH registers in exactly the same way as in low-end and mid-range PIC microcontrollers.

While PIC17-based microcontrollers continue to be available for sale, they are really only available to existing applications—this line has not seen the constant improvements and upgrades of the other three architectures. Whereas the other architectures all have Flash-based parts with upgraded peripherals and are being built on smaller chip geometries, no new PIC17 devices have been introduced since around 1998, and the parts that are available are EPROM-based and do not have many of the peripherals available in the other PIC microcontroller architectures. The main feature for selecting the PIC17 architecture, the ability to access external memory devices, is available in the PIC18 architecture, which makes the PIC17 redundant. For this reason, I have not concentrated on the PIC17 architecture in this book and suggest that if you are looking for PIC microcontrollers capable of accessing large amounts of external memory, look at the PIC18 architecture.

PIC18 ARCHITECTURE

It is unfortunate, but the first diagram that you see when you open up the PIC18Cxx's datasheet is Fig. 6.24. This block diagram, while very accurate, is very imposing. Like



- 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 MOVFF 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 6.24 PIC18 processor architecture.

the process I used with the mid-range PIC microcontrollers, I want to work through the various processor features of the PIC18 to let you see how the different pieces fit together, without being overwhelmed. In the following sections I will help round out your understanding of the PIC18’s registers and program counter operation based on what I have already presented for the mid-range PIC microcontrollers.

It will be surprising to you, but the PIC18 is probably the easiest PIC microcontroller for which to develop assembly-language code. This is due to its large linear register space that can be accessed simply and multiple index registers that are able to operate like a data stack with pushes and pops. Further simplifying the software development process are new instructions, including new subtract instructions that work in a more conventional manner than those of the other PIC microcontroller architectures. These additions reduce the amount of thinking (and remembering) involved in developing application code for the architecture, and as time goes on with Microchip working to broaden the line, I can see the fourth edition of this book focusing on the PIC18 architecture and presenting the other two as devices to consider for specific applications.

In Fig. 6.25 I have tried to show that the registers are all contained in a 4,096-byte contiguous register space. What is important to realize (and may not be very clear in Fig. 6.25) is that the WREG register provides an input to the ALU as well as a possible destination to all the arithmetic and bitwise instructions. When registers are accessed directly, an 8-bit address is specified in the instruction. To access every byte within the register space, a 4-bit BSR register has been provided with the ability to select each 256-register bank. As I will show in the next section, direct register access has some shortcuts you can take advantage of to avoid using the BSR in your applications.

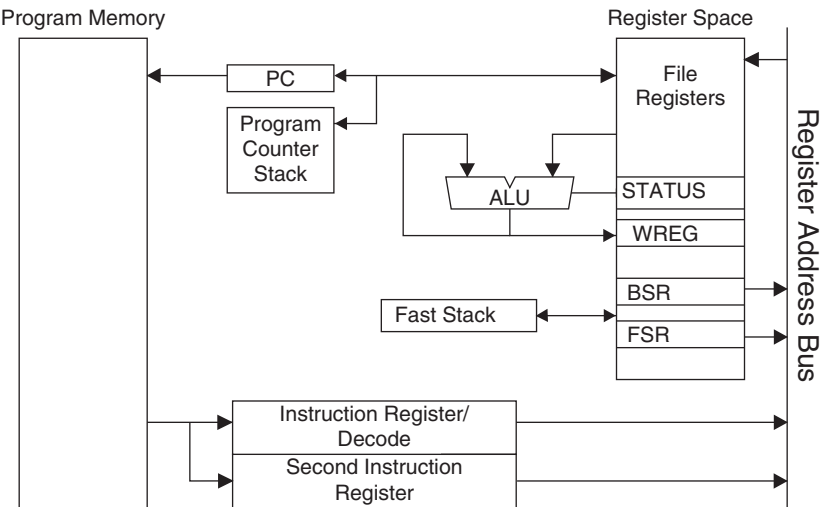


Figure 6.25 The basic PIC18 architecture use for referencing instruction operation.

The PIC18 processor has a number of FSR index registers with FSR before and after increment, after decrement, and the ability to access data relative to the FSR. This will make compiler development much simpler for the PIC18 than for any of the other PIC microcontrollers.

As I mentioned earlier, instruction, formatting, and execution are similar in the PIC18, with the major difference being the direct register addressing options. In the PIC18, there are a number of word instructions that allow `goto` and `call` instructions throughout the entire 2-MB maximum program memory space, as well as the ability to move register contents directly within the memory space.

The PIC18's ALU has been enhanced compared with the other PIC microcontrollers by the inclusion of add with carry and *true* subtract with borrow instructions. The true subtract with borrow instruction works as you would expect instead of the typical reversed PIC microcontroller instruction, that is,

```
subwfb  Reg, Dest          ; Dest = Reg - WREG - B
```

The PIC18 offers the

```
subwfb  Reg, Dest          ; Dest = WREG - Reg - B
```

which works in a more traditional manner than the standard PIC microcontroller subtraction instructions. This new subtraction instruction will make the transition to PIC18 microcontrollers easier for people familiar with other computer processors.

The PIC18 has an 8×8 multiplier that runs in a single-cycle instruction. This feature multiplies the contents of the WREG register by a constant or the contents of a register and places the 16-bit result in the PRODH and PRODL registers. The multiplier allows the implementation of some basic DSP algorithms in your applications.

Remembering a trick from high school, you can use the 8-bit multiply capability to multiply two 16-bit numbers together. When you were learning basic algebra and multiplying two, two value expressions together, you were taught to multiply the two *first* values together, followed by the outside, inside, and last. The acronym you were given was *FOIL*, and it could be described as

$$(A + B) \times (C + D) = AC + AD + BC + BD$$

By breaking a 16-bit number into 2 bytes and recognizing that the high byte is multiplied by 0x0100, *A* and *B* can be written as

$$A = (AH \times 0x0100) + AL \quad \text{and} \quad B = (BH \times 0x0100) + BL$$

For $A \times B$, the numbers can be broken up into two parts and then FOILED:

$$\begin{aligned} A \times B &= (AH \times 0x0100 + AL) \times (BH \times 0x0100 + BL) \\ &= AH \times 0x0100 \times BH \times 0x0100 + AH \times 0x0100 \times BL + AL \times BH \times 0x0100 \\ &\quad + AL \times BL \end{aligned}$$

Knowing that multiplying by 0x0100 is the same as shifting up by 1 byte (or by 8 bits), the two 16-bit variables, *A* and *B*, can be multiplied together into the 32-bit product using the code

```
Product = MUL(AL, BL);
TProduct = MUL(AL, BH) << 8;
Product = Product + TProduct;
TProduct = MUL(AH, BL) << 8;
Product = Product + TProduct;
TProduct = (MUL(AH, BH) << 8) << 8;
Product = Product + TProduct;
```

This algorithm can be converted very easily to assembly language.

Register access and bank addressing The PIC18 register architecture is probably the nicest of the four PIC microcontroller architecture families. While there is still banking, the variable placement rules I discuss elsewhere in this book still apply, with the ability to access key variables directly, as well as the I/O hardware registers [called the *special function registers* (SFRs) in the PIC18]. In the applications that I have done for the PIC18, I have found that I have had to think the least about variable placement and hardware register accessing.

The PIC18 can access up to 4,096 eight-bit registers that are available in a contiguous memory space. Twelve address bits are used to access each address within the *register map* space shown in Fig. 6.26. While there are still register banks, the file registers from

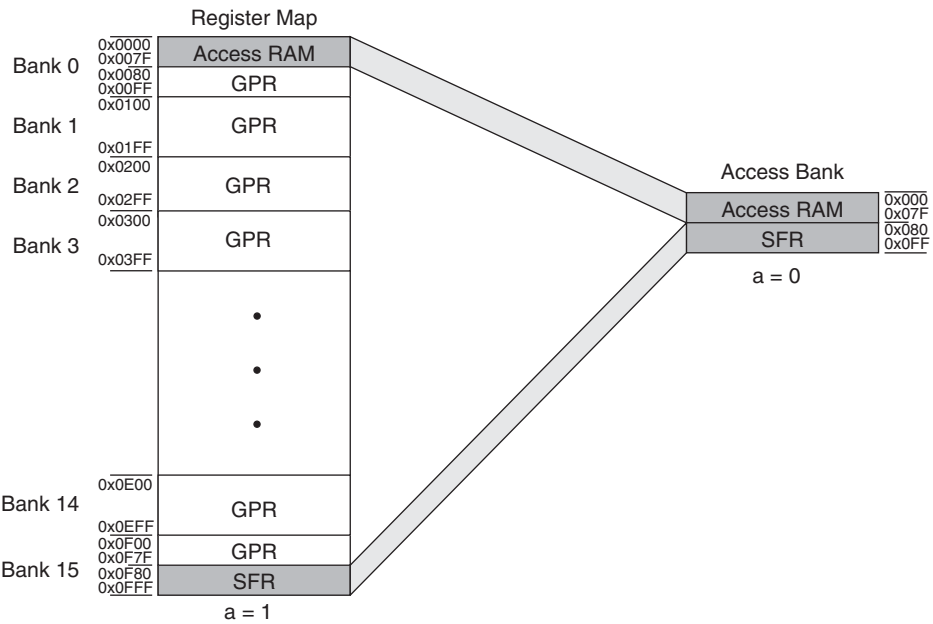


Figure 6.26 The PIC18 register organization allows for sixteen 256-byte register banks or a single bank combining 128 file registers and 238 SFRs.

one bank to the next can be accessed simply by incrementing one of the three FSR registers instead of the redirection of the FSR register into the next bank that is required in the other PIC microcontrollers. The FSR registers either can be loaded with a full 12-bit address using the `lfsr` instruction or can be accessed directly by the application.

To access a register in a specific bank directly, the PIC18's bank select register (BSR) must be set to the bank in which the register is located. The BSR contains the upper 4 bits of the register's address, with the lower 8 bits explicitly specified within the instruction. The direct address is calculated using the formula

$$\text{Address} = (\text{BSR} \ll 8) + \text{direct address}$$

To simplify directly accessing variables, the first 128 addresses are combined with the second 128 addresses as shown in Fig. 6.26 to make up the *access bank*. This bank allows direct addressing of the special function registers (SFR) in the PIC microcontroller, as well as a collection of variables, without having to worry about the BSR register. Practically speaking, the access bank means that for the first time in the PIC microcontroller, you will be able to access most, if not all, of the registers required in an application without having to specify a bank or use a special instruction (such as `tris` and `option`). This greatly simplifies the task of developing PIC18 applications and avoids some of the more difficult aspects of learning how to program the PIC microcontroller in assembly language—how to access data in different banks.

The hardware I/O registers (SFR) are located in the last 128 addresses of the register space. This may seem limiting, but remember that this is more dedicated hardware register space than is available in any of the other PIC microcontrollers.

The index register operation of the PIC18 is very well organized and will make it much easier for compiler writers to create PIC18 compilers than for other PIC microcontrollers. Along with the three 12-bit-long FSR registers, when data is accessed, it can result in the FSR being incremented before or after the data access, decremented after, or access to the address of the FSR contents added to the contents of the `w` register. A specific access option is selected by accessing different INDF register address. Table 6.9 lists the different INDF registers and their options concerning their respective FSR registers.

TABLE 6.9 PIC18 FSR CHANGE ACCESS REGISTERS

INDF REGISTER	OPERATION
INDF#	Access the register pointed to by FSR#
POSTINC#	Access the register pointed to by FSR# and then increment FSR#
POSTDEC#	Access the register pointed to by FSR# and then decrement FSR#
PREINC#	Increment FSR# and then access the register pointed to by FSR#
PLUSW#	Access the register pointed to by the contents of the WREG added to FSR#

When I discuss compilers later in this book, I discuss the operations carried out based on the traditional PIC microcontroller architectures. The capabilities of the FSR register in the PIC18 allow the FSR registers to simulate stack operations. For example, to simulate a push of the contents of the WREG using FSR0 as a stack pointer, the operation

```
POSTDEC0 = WREG;
```

could be used. Going the other way, a pop WREG could be implemented as

```
WREG = PREINC0;
```

In the first case, the stack is decremented after a data value is placed on it. When the data value is to be popped off, the stack pointer (FSR0) is incremented, and the data value it is pointing to is returned. I specified this order of operations to allow access to pushed stack items. Each time a value is pushed, the FSR register is decremented. To go back and access other items, I can use the PLUSW0 register to read a stack element. For example, to read the element placed three pushes earlier, I would use the code

```
WREG = 3;
```

```
WREG = PLUSW0;
```

This example, while showing how the FSR access with offset works, does not take into account the abilities of the PIC18 instruction set. Using the preceding example as a basis, you probably would assume that writing into the FSR stack at a specific offset is not simple. This is so because there is no way to add a constant from the WREG and have a value somewhere that can be accessed and written by the FSR register. The PIC18's `movff` instruction allows data transfers using an FSR index register and the WREG offset without accessing WREG in any way.

Program counter The PIC18's program counter and its stack are similar in operation to those of the other PIC microcontrollers, but they have the ability to be modified under application software control. This new capability greatly enhances the PIC18's ability to run multitasking operating systems or monitor programs compared with the other PIC microcontrollers. This is an exciting feature and one that I will take advantage of later in this book.

The PIC18Cxx program counter and stack are similar to the hardware used in the other PIC microcontroller architectures except for three important differences. The first difference is the additional bits required for a program counter accessing 20 address bits for the maximum 1 million possible instructions of program memory. The second difference is the availability of the *fast stack*, which allows interrupt context register saves and restores to take place without requiring any special code. The last difference is the ability to read and write from the stack. These differences add a lot of capabilities to the PIC18 that allow applications that are not possible in the other PIC microcontroller architectures.

In the PIC18, when handling addresses outside the current program counter, not only is a PCLATH register (or PA bits as in the low-end devices) update required, but also a high-order register update for addresses above the first 64 instruction words. This register is known as PCLATU. PCLATU works identically to the PCLATH register, and its contents are loaded into the PIC18Cxx PIC microcontrollers program counter when PCL is updated.

Each instruction in the PIC18 starts on an even address. This means that the first instruction starts at address 0, the second at address 2, the third at address 4, and so on. Setting the program counter to an odd address will result in the MPLAB simulator halting and the PIC18 working unpredictably. Changing the convention used in the previous PIC microcontrollers to one where each byte is addressed means that some rules about addressing will have to be relearned for the PIC18.

The fast stack is an interesting feature that will simplify your subroutine calls (in applications that don't have interrupts enabled), as well as working with interrupt handlers. To use the fast stack in the `call` and `return` instructions, a 1 parameter is put at the end of the instructions. To prevent the fast stack from being used, a 0 parameter is put at the end of the `call` and `return` instructions.

The fast stack is a 3-byte memory location where the `w`, `STATUS`, and `BSR` registers are stored automatically when an interrupt request is acknowledged and execution jumps to the interrupt vector. If interrupts are not used in an application, then these registers can be saved or restored with a `call` and `return` such as

```
call sub, 1                      ; Call "sub" after saving "w",
"STATUS"                        ; and "BSR"
:

Sub:                             ; Execute "Sub", Ignore "w",
"STATUS"                        ; and "BSR"
:

return 1                        ; Restore "w", "STATUS" and "BSR"
before
; Return to Caller
```

The reason why the fast option is not recommended in applications in which interrupts are enabled is because the interrupt overwrites the saved data when it executes. For this reason, the fast option cannot be used with nested subroutines or interrupts.

If the fast option is not used with interrupts, then the three context registers can be saved, restored by using the code

```
Int:
movwf _w                        ; Save Context Registers
movff STATUS, _status
movff BSR, _bsr

:                                ; Interrupt Handler Code

movff _bsr, BSR                 ; Restore Context Registers
movf _w, w
movff _status, STATUS
retfie
```

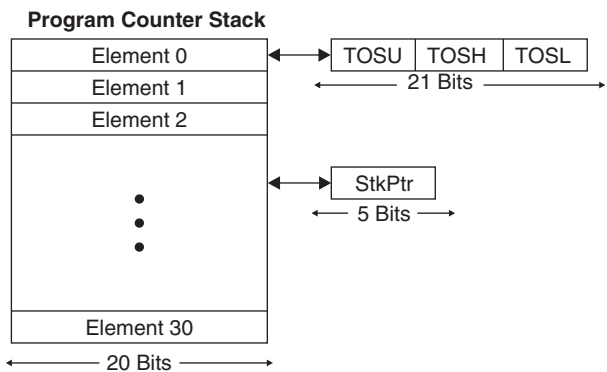


Figure 6.27 The data on the PIC18 stack is accessible from the program using the TOS# registers.

Note that in the interrupt handler, the *w* register is restored before the STATUS register so that the status flags are not changed by the *movf* instruction after they have been restored. This code should be kept in your hip pocket until it is required.

The last difference is also the most significant. The ability to access the stack is quite profound, and a deeper understanding of the PIC18’s stack is required than for the other PIC microcontroller processor architectures. The stack itself, at 31 entries, is deeper than the other PIC microcontroller stacks, and the hardware monitoring the stack is available as the STKPTR register. A block diagram of the stack is shown in Fig. 6.27, whereas the STKPTR register bit definitions can be found in Table 6.10. The STKUNF and STKFUL bits will be set if their respective conditions are met. If the STVREN bit of the configuration fuses is set, then when the STKUNF and STKFUL conditions are true, the PIC microcontroller will be reset.

I’m of a mixed mind as to the appropriateness of resetting the PIC microcontroller after an invalid stack operation. While a reset definitely will indicate that an error has occurred (just like a watchdog timer timeout), there would be a problem with decoding what has happened. Ideally, there would be an interrupt that could have its own handler to report and deal with the issue.

The value at the top of the stack can be read (or written) using the top or stack (TOSU, TOSH, and TOSL) registers. These registers are pseudoregisters like INDF.

TABLE 6.10 PIC18 STKPTR REGISTER BIT DEFINITIONS	
BIT	DESCRIPTION
7	STKFUL—stack full flag, which is set when the stack is full or overflowed
6	STKUNF—stack underflow flag, which is set when more stack elements have been popped than pushed
5	Unused
4-0	SP4:SP0—stack pointer

These registers access the top of the stack directly. When an address is pushed onto the stack, the SP bits of the STKPTR register are incremented, and then the TOSU, TOSH, and TOSL registers are updated. Address pops happen in the reverse order; data is taken out of the TOSU, TOSH, and TOSL registers, and then the stack pointer bits are decremented. The PIC18 has push and pop instructions that increment and decrement the stack pointer and SP bits. These instructions should be used for changing the stack pointer, and the SP bits never should be written to directly to avoid any possible damage to the stack and being unable to return to the caller.

Using the TOS registers, the stack can be recorded or changed. For example, if you want to implement a computed return statement, the following computational algorithm could be used:

```

TOSU = ((TableStart & 0xFF0000) >> 8) + ((offset & 0xFF0000) >> 8);
if (((TableStart & 0xFF00) >> 4) + ((offset & 0xFF00) >>
    4)) > 0xFF)
    TOSU = TOSU + 1;
TOSH = ((TableStart & 0xFF00) >> 4) + ((offset & 0xFF00) >> 4);
if (((TableStart & 0xFF) + (offset & 0xFF)) > 0xFF)
    if ((TOSH + 1) > 0xFF) {
        TOSH = 0;
        TOSU = TOSU + 1;
    } else
        TOSH = TOSH + 1;

```

Return

The block diagram for the PIC18's program counter and stack is similar to that of the other PIC microcontrollers but incorporates the differences that have been discussed and is shown in Fig. 6.28. You can see that the 21-bit program counter can be updated from

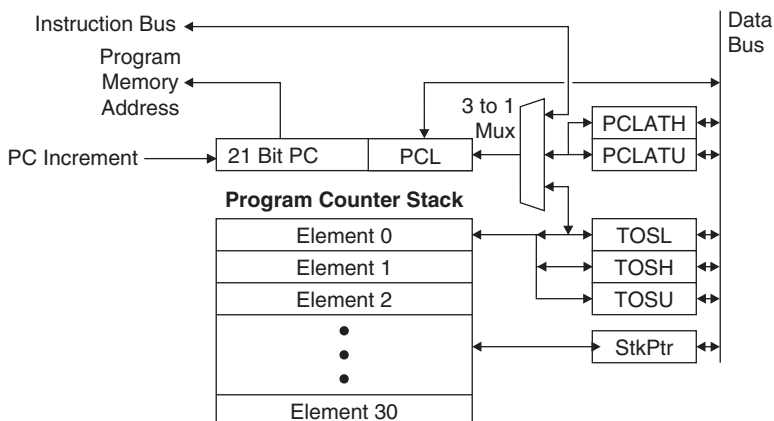


Figure 6.28 The program counter is integrated into the stack, and both can have their data accessed by the program.

either the stack, the PCL registers of the processor, or a new address from the instruction or incremented after normal instruction execution. The output value is used for accessing program memory during execution and/or the configuration fuses during programming.

Interrupts The PIC18 has some unique interrupt capabilities. Rather than having multiple interrupt vectors, each dedicated to a type of interrupt such as the PIC17, the PIC18 has the ability to specify high-priority interrupts, which are given a different address from low-priority interrupts. As well as providing a fast path for the high-priority interrupts, this feature allows splitting up of interrupts to avoid having to check F bits to determine which interrupt is active.

When you look at a PIC18 device datasheet, you will see that the high-priority interrupts are at address 8, and the low-priority interrupts are at address 0x18. This may seem different from the mid-range devices, which have a single interrupt vector at address 4; the addresses are exactly the same because the PIC18 does not handle addresses in the same way as the low-end and mid-range devices. Each PIC18 instruction takes up 2 bytes, and the addresses are based on the number of bytes, not on instructions, as is the case in the low-end and mid-range architectures. This difference means that the high-priority interrupt vector is at the fifth instruction after the reset vector, exactly the same as in the mid-range devices.

I have pointed out this difference because the two-address increment for every PIC18 instruction affects every aspect of the architecture and any comparisons or application porting between architectures. This is something that you must be cognizant of when you are working with the chips to ensure that you do not make the mistake of jumping to an incorrect address because you were following the addressing convention of another specific architecture instead of the one you are actually using. I find that errors of this type are the number one mistake I make in my PIC18 programming because I am so familiar with the low-end and mid-range PIC microcontroller architectures and addressing operations.

USING THE PIC MCU INSTRUCTION SET

In this book, I have put a lot of emphasis on understanding the PIC[®] microcontroller's processor architecture and visualizing how application code and instructions move data through the PIC microcontroller. I do this because the PIC microcontroller's instruction set is somewhat unusual. Most people first learn assembly language programming on a conventional Von Neumann processor, like the Motorola 6800, and when presented with the PIC microcontroller they feel like they are starting all over again. By developing a good understanding of the PIC microcontroller device, you will be able to code it quite easily. Along with being able to develop software for the PIC microcontroller, you will also be able to look for opportunities to optimize your application and simplify it using the PIC microcontroller's architectural features.

To characterize a processor's instruction set, I find that it is best to break the instructions into functional groups. The instruction sets used by the three different PIC microcontroller architectures discussed in this book can be broken up into four such groups. The first group contains the data movement instructions, which are used to move data in and out of the processor. As I indicated earlier in the book, data movement within the PIC microcontroller always takes place through WREG, although register arithmetic instructions have the option of storing the result into WREG or back into the source register. Data processing instructions include adding, subtracting from registers, along with incrementing, decrementing, and doing bitwise operations. The arithmetic instruction group can be broken up into two subgroups, the register arithmetic (where only the contents of registers are used) and the immediate arithmetic (where an explicitly stated constant value is used for the operation). Execution change instructions make up the next functional group. These are the `gotos`, `calls`, and `returns` as well as conditional instruction skips. The PIC microcontroller instruction sets differ from other traditional processor instruction sets in that a "jump on condition" requires two instructions instead of a single explicit one. To carry out conditional jumps or other conditional operations, a "skip next instruction" is executed before the actual operation. Along with traditional `goto` and `call` instructions, there is the opportunity to write to the PIC microcontroller's

program counter directly. This gives you the opportunity to create conditional jumps based on arithmetic values or implement data tables, which return constants for different values. This is a very powerful function in the PIC microcontroller and one that I will spend a number of pages discussing and showing examples of in this book. Finally, there are processor control instructions that are used to control the operation of the PIC microcontroller's processor. These instructions are quite typical for most microprocessors (not only microcontrollers) and there shouldn't be any surprises in this set.

As in the previous chapter, I will initially focus on the mid-range PIC microcontrollers and the MPLAB IDE simulator to demonstrate the various instructions' operation. This will be done by creating small applications designed to be run on the simulator, to help you become more familiar with MPLAB IDE and help you visualize the operation of the instruction. When working through the instructions, you will see that all PIC microcontroller instructions execute in one instruction cycle unless the program counter value is changed; later in the book I will discuss how time-critical code is written. After reviewing the operation of the mid-range PIC microcontroller's instructions, I will review the differences between the instruction sets used by the other PIC microcontroller families.

Setting Up the MPLAB IDE Simulator with a Test Template

In this chapter, I will provide you with some small programs that you can simulate in MPLAB IDE to demonstrate the operation of the various instructions. If you are working through the book in order, you will have already downloaded and installed the latest version of MPLAB IDE from Microchip's web site. If you haven't, then you should go through the steps to do this in Chap. 3. Once you have MPLAB IDE installed on your computer, you can start adding the basic programs presented in this chapter and stepping through them to see the execution of the PIC microcontroller instructions.

To create a project that is appropriate to the material in this chapter you will have to create a generic or template project that will be used for all the sample programs. After starting up MPLAB IDE, click on the Project pull-down menu and select Project Wizard. This will bring up the project start tool. For the programs presented in this chapter, you should select a PIC16F877A for the PIC microcontroller as it uses all the features available to the mid-range PIC MCU processor to help you learn about the various instructions. Next, make sure that you have the Microchip MPASM toolsuite selected, because we are going to work in assembly language to learn about the instructions. Next specify that the project name is Instruction Template and create the C:\PICMCU\Assmblr\InsTemplt folder using the New Folder feature of the wizard—the folder name is somewhat compressed to allow longer project names because MPLAB IDE has a folder and project file name maximum of 64 characters. Once this is done, don't select "Add any existing files to your project" (just click Next) and then click Finished to create the project.

Now we're going to spend a few moments customizing the project so it can be used for the small programs that I will present you with for the remainder of this chapter. To add a new source file to the project, click on the File pull-down menu followed by "Add new file to the project" and select the C:\PICDwnld\Templates directory and choose InsTemplate.asm from the list.

The InsTemplate.asm file should contain the following program statements:

```
LIST R=DEC
    INCLUDE "p16f877a.inc"

CBLOCK 0x20
ENDC

    _CONFIG _CP_OFF & _DEBUG_OFF & _WRT_OFF & _CPD_OFF & _LVP_OFF &
    _BODEN_OFF & _PWRTE_ON & _WDT_OFF & _XT_OSC

    org      0

; Insert Code Here

    goto    $
end
```

These statements are the bare-bones statements required for you to add a few assembly language statements to test out their operation. For the mid-range instructions, there is no need to change any statement except for the comment line; **Insert Code Here**. The semicolon (;) indicates that the characters to the right are human comments and should be ignored by the assembler. With the file in the project, click on the File pull-down menu and then Save As and store the InsTemplate.asm file in the C:\PICMCU\Assmblr\InsTemplt directory. Next, you will have to right-click on the file underneath Source Files in the project window and click on Remove. Finally, right-click on Source Files followed by Add File and select InsTemplate.asm from C:\PICMCU\Assmblr\InsTemplt. This will prevent you from losing the original template file source and having to download it again from the McGraw-Hill web site or key it in manually.

Next, I would like you to add a couple of windows to provide you with the ability to monitor the operation of the simulated PIC microcontroller. First, click on View followed by File Registers and move and resize the window that comes up as shown in Fig. 7.1. This window shows the special function registers (SFRs) and the file registers in the system. Unless you have a very high pixel count display, you are going to have to obscure some of the higher address registers. Finally, click on the View pull-down menu, followed by Watch, and then move it to the bottom-left corner as shown in Fig. 7.1.

Watch is a user-configurable window in which different registers can be selected and their contents displayed or changed. To add registers to the window, select the SFR that you want to put in and then click on the Add SFR button—for the sample programs in this chapter, you will need WREG, STATUS, OPTION_REG, PCL, and PCLATH.

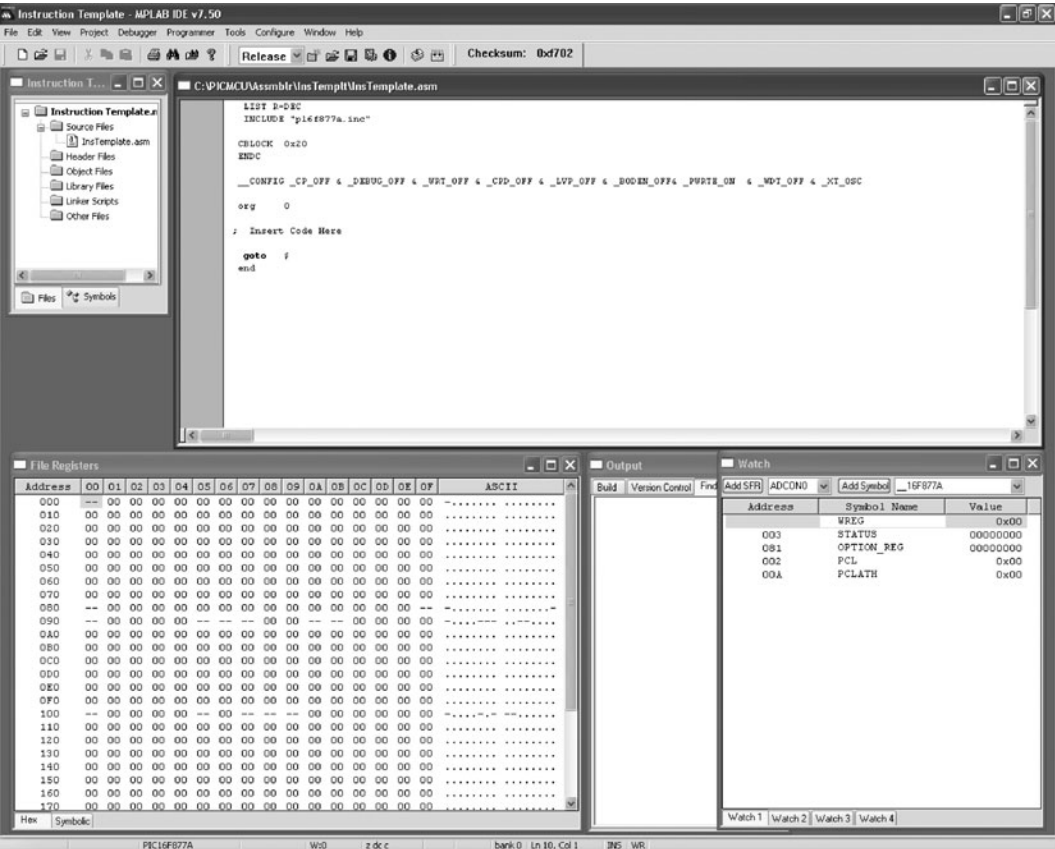


Figure 7.1 Start up MPLAB IDE and arrange the Instruction Template file window, Output, Ins Template.asm, File Registers, and Watch windows on the desktop as shown here.

After adding these registers, right-click on STATUS and click on Properties from the list that comes up. The Watch Properties window that opens (Fig. 7.2) allows you to select the register from the list in the Watch window and specify it as ASCII, binary, decimal, or hex by first selecting the register followed by the Format and clicking on Apply. To complete the Watch window setup, change STATUS and OPTION_REG to binary. Once this is all done, click on the Project pull-down menu and then Save Project to make sure the changes are saved.

With MPLAB IDE set up as shown in this section and following the instructions laid out here and below, you can now take a predefined project and source code template, add a few lines of code into the template, build (assemble) the code, and then simulate the operation of the code to see exactly how the instructions execute. The template file and project were created for this chapter, but I would suggest that you keep it where you can access it easily to allow you to test out some basic algorithms or step through your own project logic. Little “what if” tools like this can make learning a new chip and

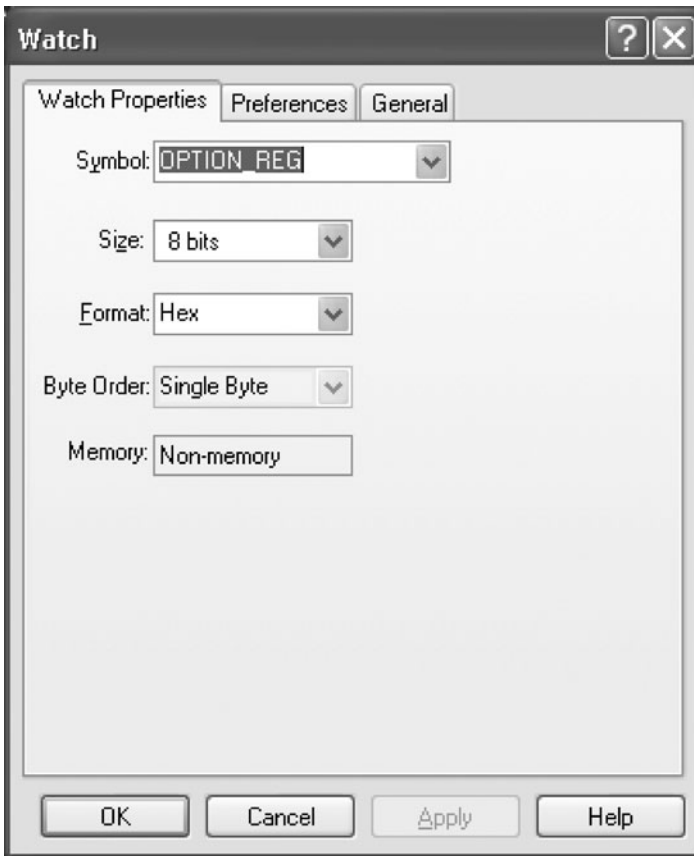


Figure 7.2 The Watch Properties window allows you to change the displayed data format of the specified Watch register.

assembly language easier as well as providing you with the capability of trying out new ideas separate from an already established package.

PIC MCU Instruction Types

It is a mistake to assume that the instructions of the three different PIC microcontroller architectures discussed in this book work separately from each other. One of the major differences between the three architectures consists of the number of registers each can access, but the methodologies used to access data in the registers are identical. Similarly, the ALU of the different PIC microcontrollers also work identically as do the execution change instructions and the processor control instructions. Despite the differences in the size of the instruction word, the number of registers that can be accessed, and the

maximum program memory size available to each of the three PIC microcontroller architectures, the instructions execute very similarly. You will find it quite easy to write code for any of the architectures and port the instructions between them.

Before I review the processor instruction sets, I want to introduce you to the basics of each instruction type so that you are looking at the instructions from the perspective of what they are doing. The three PIC microcontroller families have different instructions and they operate differently in a variety of situations, but they all accomplish the same tasks in substantially the same way. If you look at what's being done instead of how it's being accomplished, the actual function will become much clearer, and you will be able to leverage the knowledge you have for a specific architecture and apply it to another.

Looking ahead, I will introduce you to three of the four instruction types that I discuss throughout the book. The missing instruction type is processor control, and the reason for omitting this type is not because it is unique for each of the three PIC microcontroller processor architectures, but because they are largely the same. The processor control instructions really only provide a minimum of services, the watch-dog clear instruction being the single basic and common instruction that you will have to content yourself with. The other instruction types do execute differently depending on the PIC microcontroller architecture that you choose to work with for a given application.

DATA MOVEMENT INSTRUCTIONS

Data movement within the PIC microcontroller architectures is limited to moving data from the program memory to the register space and within the register space. Typically, data moves through the arithmetic/logic unit (ALU) to either WREG rather than directly between registers. Except through special hardware, data cannot be stored outside of the register space. The limited data movement types through the PIC microcontroller processor make application software development easier and faster to learn.

The first type of data movement instruction is the literal or immediate addressing in which the least significant 8 bits of an instruction are the actual data to be used in the instruction (Fig. 7.3). The data is a constant value specified when the application is developed and used as a parameter for an arithmetic or Boolean arithmetic operation. It is important to not confuse this instruction with a `goto` or `call`, even though these instructions have constant values that are loaded into registers—in these cases, the constant values are loaded into program counters, not general purpose registers, like the literal addressing instructions. Literal data can only be found in the program memory space, not in the register space.

Data that is stored in the register space and accessed using an address built into the instruction is known as “direct” addressing and is shown in Fig. 7.4. The PIC microcontroller



Figure 7.3 Literal or immediate data is stored in instructions.

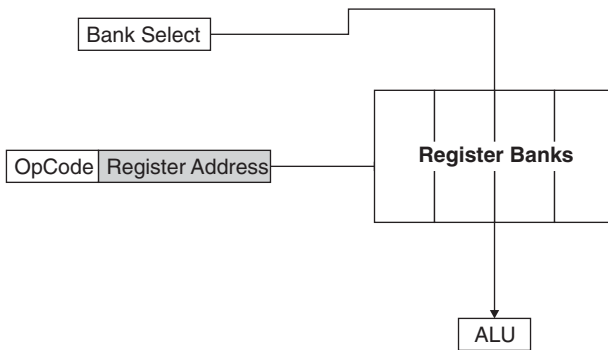


Figure 7.4 Direct addressing specifies the register in the specified bank.

architectures have a number of register banking options, each of which has to be appropriately selected. The address within the bank is specified by the address in the instruction, similar to how the data was specified in the literal instructions.

In all PIC microcontroller architectures the size of the bank, limited by the number of bits available in the instruction, is usually determined to be less than adequate so multiple register spaces (known as banks) are collected together to create the complete register space shown in Fig. 7.4. The selection of the bank that is currently being accessed is made by bits located in a register whose address is common in each of the banks.

When you are accessing a register in a bank other than zero, you should XOR any set bit 8 or bit 9 address bits indicating the bank location of the register. For example, if you were to load WREG with the contents of TRISB, which has an address of 0x86 (which is greater than 0x7F, the maximum address in a bank), you should use the instruction and XOR value:

```
movf    TRISB ^ 0x80
```

Without the 0x80, you will be specifying an address greater than the maximum size of the bank and MPASM will issue a message. When you change banks, every bank access should have an XOR with a constant like this to indicate the offset within the bank. Another, more obvious approach would be to AND the address with 0x7F like:

```
movf    TRISB & 0x7F
```

but this method is generally rejected in favor of the XOR because if a register is not in the currently selected bank then the result of the operation will be greater than 0x7F and the message will be produced telling you there is a register being accessed that isn't in the bank you have selected. This little trick could save you a lot of time finding an error caused by an incorrect attempt to access a register that is actually located in another bank.

The final method of accessing data in the PIC microcontrollers is indexed addressing in which a register is used to specify the register address (Fig. 7.5). Indexed addressing

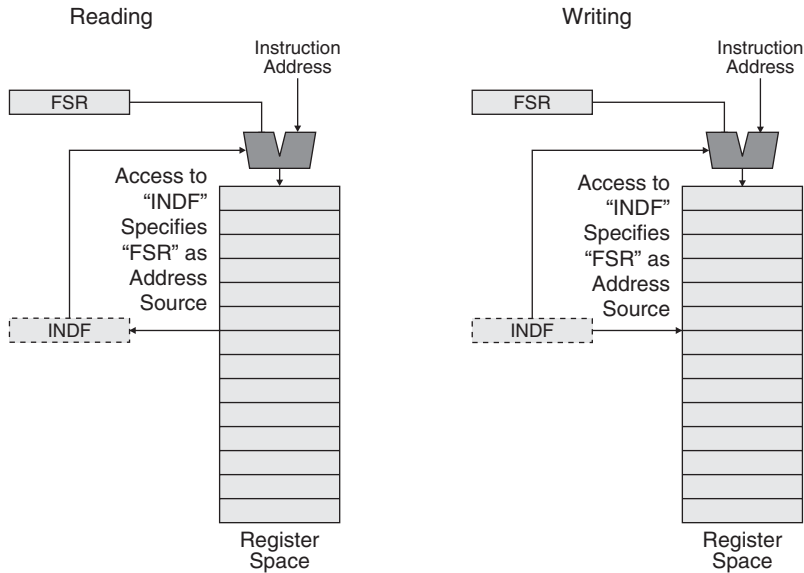


Figure 7.5 Indexed addressing uses a register to specify the address of the data to be accessed.

differs from the previous addressing type because to a certain extent the bank can be specified by the value within the index register and not rely totally on extra bits in another register to select the register bank to be accessed. It is important to note that array variables cannot pass over register banks, except in the case of the PIC18 architecture, because of the use of common variables at specific locations within the banks—writing to these variables will change the operation of the PIC microcontroller.

If you are familiar with other small processors, you will probably be wondering where are the data stack (push and pop) instructions as well as the ability to read and write to program memory. These addressing modes are largely not available in the PIC microcontroller architectures except by providing them through special instruction sequences, which are discussed later in the book.

DATA PROCESSING INSTRUCTIONS

The arithmetic and Boolean arithmetic instructions executed in the PIC microcontroller typically follow the path shown in Fig. 7.6. As will be discussed later in the chapter, data is typically taken from a register and passed through the ALU along with the contents of WREG, processed, and then stored back into the register or in WREG. This path is quite unusual for computer processors, but it actually allows for surprisingly sophisticated and efficient application software.

The operations typically available within the PIC microcontroller's ALU are:

- Addition
- Subtraction

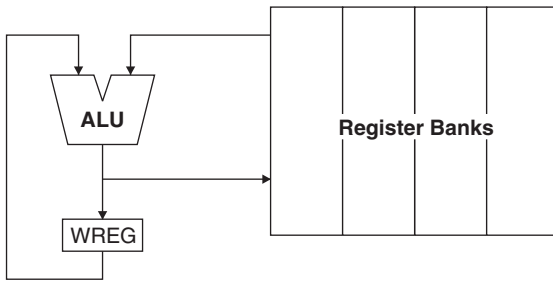


Figure 7.6 Data processing in the PIC microcontroller is a result of processing the contents of WREG with the contents of a register.

- Incrementing
- Decrementing
- Clearing
- Bit setting
- Bit resetting or clearing
- Bitwise AND
- Bitwise OR
- Bitwise XOR
- Bitwise complement

These operations are the basis of the mathematical and bitwise operations performed by the PIC microcontroller and while this list probably seems modest, it will provide you with the functions required for you to be able to develop complete applications.

I believe it is critical to be able to visualize the flow of data in a processor in order to be able to effectively program it. Even though the flow is somewhat unusual, you should be able to see the data flow in your mind when you start developing your own application code for the PIC microcontroller architectures.

EXECUTION CHANGE INSTRUCTIONS

Execution change instructions are any instruction that can cause the flow of the program execution to deviate from being sequential, executing one instruction after the previous in program memory. When this capability is discussed, most people think of the `goto` or `call` instructions, which cause execution to be relocated to a new address, or the conditional branch instructions, which cause execution to be relocated when specific conditions are met. These instructions and capabilities are provided in the PIC microcontroller along with additional capabilities that are unique to the architecture and will provide you with ways of approaching programming execution control problems that you probably have never seen before.

`goto` and `call` instructions are supported by the PIC microcontroller architectures, but there are some aspects to them that you have to be aware of. Unlike many other processors, the instruction size remains constant and is used to describe the

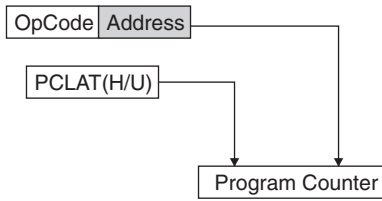


Figure 7.7 The PIC microcontroller `goto` and `call` instructions use the address specified within the instruction as well as the high order bits stored in the PC latch registers.

architecture it is running on (the low-end PIC microcontroller architecture is often referred to as the 12-bit architecture, the mid-range as the 14-bit architecture, and the PIC18 as the 16-bit architecture). The requirement for the instruction size to stay constant means the instruction cannot hold the full number of bits needed to specify a location anywhere in the address space. The number of bits that can be stored in the `goto` or `call` instruction allows a change of execution anywhere in the page size, which is defined as the maximum number of addresses that can be accessed by the address bits available in the instruction.

The solution to this dilemma was to provide some additional address bits in a register, which are stored in the program counter when the new address is loaded into it. As shown in Fig. 7.7, the additional address bits are loaded into the program counter at the same time as the new address. Typically, these bits remain unchanged—`gotos` normally take place within the currently executing page, which does not require a change in these bits. `gotos` or `calls` to another page require the PC latch register(s) to be loaded with the high order bit addresses before the `goto` or `call` instruction is executed. This may sound cumbersome, but it actually isn't and you may be surprised that you can do most of your PIC microcontroller application development without ever changing the PC latch registers, except in one case outlined below.

Traditional conditional execution instructions are not available in the PIC microcontroller. Instead, changes in execution are typically implemented using `skip` instructions, which increment the program counter past the next instruction based on the state of a bit in the system. This is a tremendously powerful feature because unlike other processors that can only execute conditionally based on the results of arithmetic or Boolean operations, the PIC microcontroller can test any bit in the file register space, allowing for tests of flags or other values that would require many instructions to set up and test in other processors. The operation of the `skip` instruction is shown in Fig. 7.8. Later in the book, I will describe the status bit values you will have to know to be able to provide the same basic capabilities as are in other processors.

Along with the ability to test and execute conditionally on the state of a variable, you can also mathematically calculate a new address to start executing at. As is shown in Fig. 7.9, an 8-bit value can be written to the low 8 bits of the program counter (known as PCL) at the same time the PC latch register bits are also loaded into the program counter, providing a completely new address for execution. This feature is most commonly

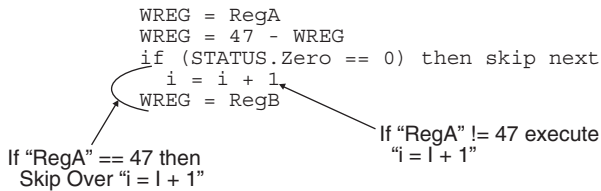


Figure 7.8 The skip instructions provide conditional execution by causing execution to ignore the following instruction.

used for providing the ability to read and return table values for strings or other types of data.

The Mid-Range Instruction Set

The mid-range PIC microcontroller has historically been the most popular of the architectures to learn because of the large number of part numbers with a variety of I/O and memory options. The mid-range architecture was the first in which advanced peripherals were added, it was the first available with EEPROM and Flash program memory, and it was the first to have MPLAB ICD debuggers built in. They are also the devices that have the most written about them. If you were to review books and websites available for you to learn more about the PIC MCU, you would discover that the mid-range devices have vastly more material than the others. As this book is written, the mid-range PIC microcontrollers also have the most/best third-party development tools developed for them, which continues to make it a popular subject for learning as well.

As I work through the instructions, I have included a diagram with the data flow for each instruction as well as the number of cycles required for the instruction to execute and the STATUS register flags that are affected. This format will be optionally used in

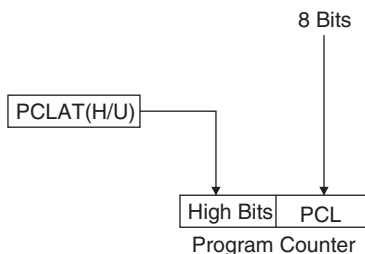


Figure 7.9 New addresses can be calculated and then loaded into the program counter via the PC latch and PCL registers.

the explanation of the instruction sets for the other PIC microcontroller processor architectures, as the instruction operation may be identical to what has been explained previously.

DATA MOVEMENT INSTRUCTIONS

If you are familiar with the Intel 8086 (the base processor used in the IBM PC), you will probably appreciate the ability of some instructions to execute without having to store temporary results in the accumulator registers. This feature can significantly simplify applications as well as avoid the need for temporarily saving the contents of the accumulators. Unfortunately, the PIC microcontroller does not have this capability and data has to pass through WREG before it can be put in a destination register. The typical data flow for information in the PIC microcontroller is:

```
w = Source;           // Load "w" with the Source Data
Destination = w;      // Store the contents of "w"
```

To load *w*, two primary instructions are used. The `movlw` instruction (see Fig. 7.10) loads *w* with an 8-bit constant value. The format of the instruction is:

```
movlw Constant        ; Load "w" with "Constant"
```

This is the basic method of loading *w* with a constant value. None of the STATUS flags are changed by the `movlw` instruction.

Using the `InsTemplate.asm` project that you set up at the start of the chapter, you can demonstrate the operation of the `movlw` instruction using the simple program (which replaces the statement `; Insert Code Here`):

```
movlw 0x12             ; Load "W" Reg with Constant
```

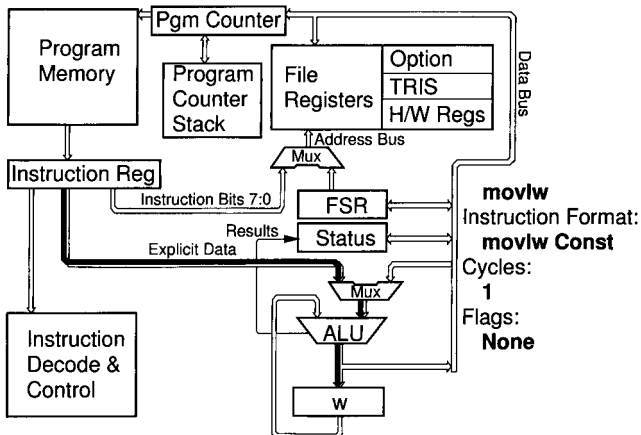


Figure 7.10 The `movlw` instruction: load literal value into *w*.

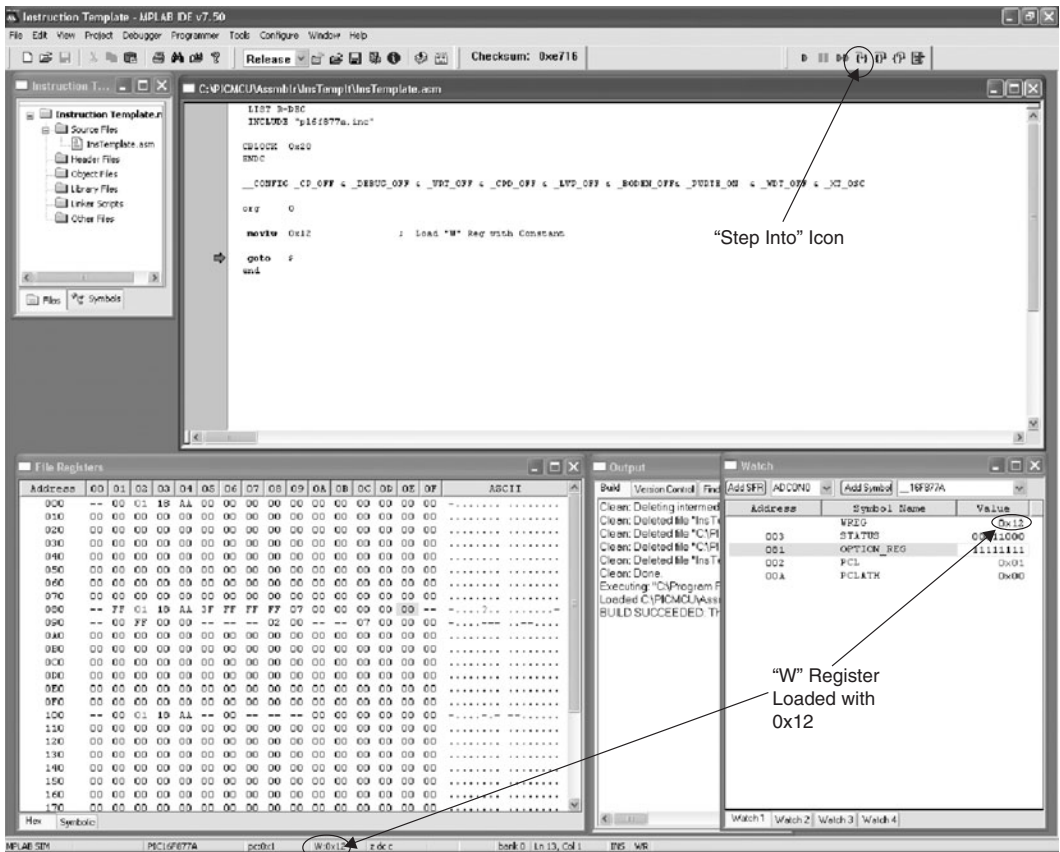


Figure 7.11 The operation of the `movlw` instruction can be easily demonstrated using the MPLAB IDE simulator.

As has been discussed in this section, the `movlw` instruction simply loads an 8-bit constant value into WREG. After adding the line of code, build it by pressing Ctrl-F10 or clicking on Project and then Build All. Make sure that the MPLAB IDE Simulator is selected by clicking on Debugger, then Select Tool, and finally MPLAB Simulator. Single-step through the `movlw` instruction by clicking once on the Step Into icon on the top left toolbar of MPLAB IDE (as shown in Fig. 7.11). When you are done the arrow will be pointing to the `goto $` instruction.

Looking at Fig. 7.11, you will see that the WREG value in the Watch window changes to 0x12, but you should also notice that WREG contents are displayed on the toolbar at the bottom of the desktop.

To load `w` with the contents of a register, the `movf` instruction (see Fig. 7.12) is used. The format of the `movf` instruction is:

```
movf  Variable, d ; Move the contents of "Variable" through the
                  ; ALU and set the "Zero" flag based on its
                  ; value. Store "Variable" according to "d"
                  ; which can be "w" or "f"
```

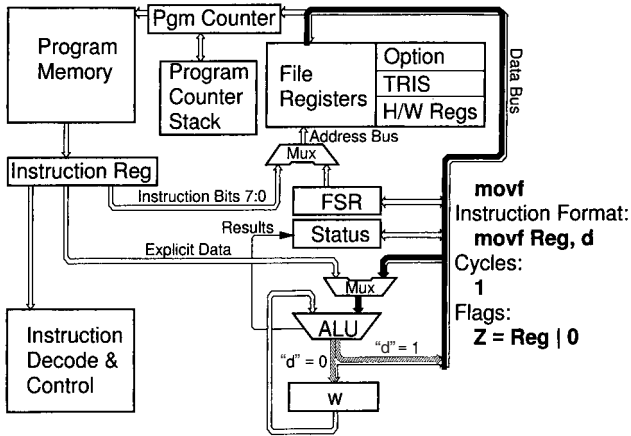


Figure 7.12 The `movf` instruction: load WREG with the contents of a register operation.

where `d` is the destination of the contents of the variable and can be either 0 or 1. When you use the MPASM assembler, the values `w` or `f` can be used in place of 0 or 1, respectively. The `w` and `f` constants are declared in the device include file. If `d` is reset (`w` or 0), then the contents of `Variable` will be stored in WREG. If `d` is set (`f` or 1), then the value of `Variable` will pass through the PIC microcontroller's ALU, change the zero flag according to its value, and then be written back into `Variable` without changing the contents of WREG.

I think of the `movf` instruction as being used to set the zero flag according to the contents of the register and optionally load `w` with the contents of the register. This may seem like a backward way of describing the instruction, but it is actually quite accurate in terms of how the instruction executes.

To test the contents of a register, the ALU ORs the value read from the register with `0x000`, which sets the zero flag if the result of this operation is zero. To demonstrate this operation, put in the following code into the `InsTemplate.asm` project:

```
movlw 0x47          ; Load a Register with non-Zero Constant
movwf 0x20

movlw 0x00          ; Load Another Register with Zero
movwf 0x21

movf 0x20, w        ; Read in Non-Zero Constant
movf 0x21, f        ; Test Zero, Note Zero Flag getting Set
```

This program first loads a non-zero value into the file register at address `0x20` and then the simulated processor will test the contents of the file register at `0x21` to see if they are zero. When you first started up MPLAB IDE and loaded in the `InsTemplate.asm` project, you probably noticed that most of the registers had zero in them. These values cannot be

considered to be an accurate representation of the initial values in a PIC microcontroller. As I will repeat throughout this book, you must *always* initialize all your variables and file registers to ensure you know exactly what the contents of the registers are.

When you single-step through the `movf 0x20, w` instruction, the value in register 0x20 (0x47) will be loaded into WREG. When you execute `movf 0x21, f` even though WREG does not change, the zero flag of the STATUS register (bit 2) will become set; you can see this in the STATUS register displayed in the Watch window as well as on the bottom toolbar of the MPLAB IDE desktop. To the right of the WREG indicator, there are three flags, **z**, **dc**, and **c**, which represent the three arithmetic result flags of the STATUS register. After executing the `movf 0x21, f` instruction, you will see that the zero flag will become a capital **Z**, indicating that the bit is set and showing how the `movf` instruction can be used to test the contents of a register to see if it is zero.

Another way of setting a register value is to use the clear instructions: `clrw` and `clrf` (see Fig. 7.13). `clrw` clears WREG and sets the zero flag, while `clrf` clears the specified register and also sets the zero flag.

The `clrw` instruction does not have any parameters that are invoked by simply entering:

```
clrw
```

The `clrf` instruction only has one parameter and that is the register that is specified:

```
clrf Register
```

Storing the contents of w into a register is accomplished using the `movwf` instruction (Fig. 7.14). This instruction simply loads the specified register with the contents of WREG. No STATUS flags are affected by the operation. The format of `movwf` is:

```
movwf Register
```

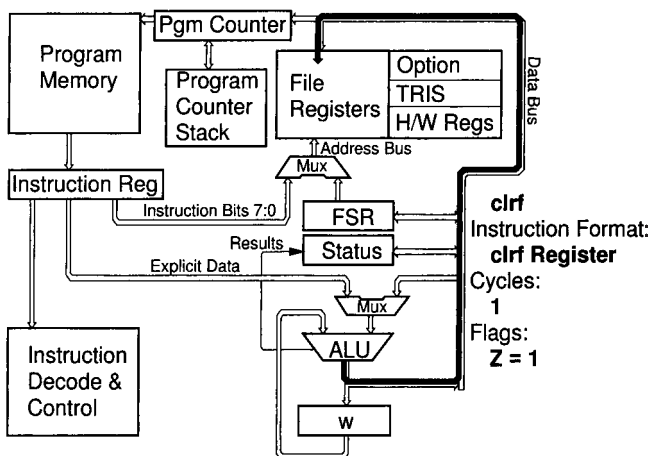


Figure 7.13 The `clrf` register clear instruction operation.

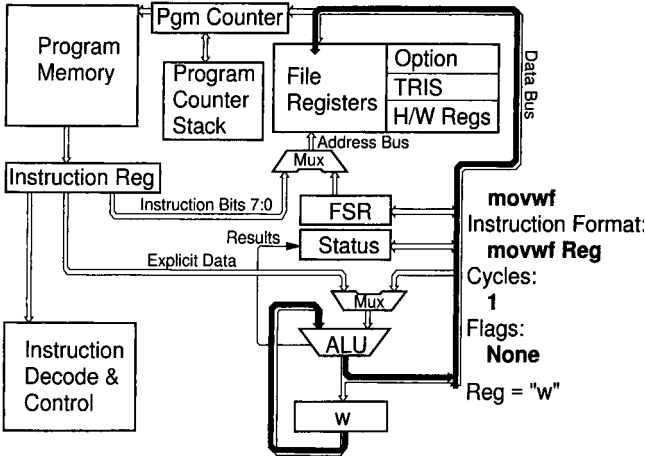


Figure 7.14 The `movwf` instruction which copies the contents of `w` into a register operation.

where `Register` is the destination register for the contents of `WREG`. The last test program (demonstrating the operation of the `movf` instruction) uses the `movwf` instruction to store values into registers for testing later.

Along with the `movwf` instruction, the `option` and `tris` instructions will store the contents of `w` into specific registers. None of the `STATUS` flags are changed (or required to be changed, as will be discussed below) for these instructions.

The `option` instruction (which doesn't have any parameters and whose operation is shown in Fig. 7.15) is specified as:

`option`

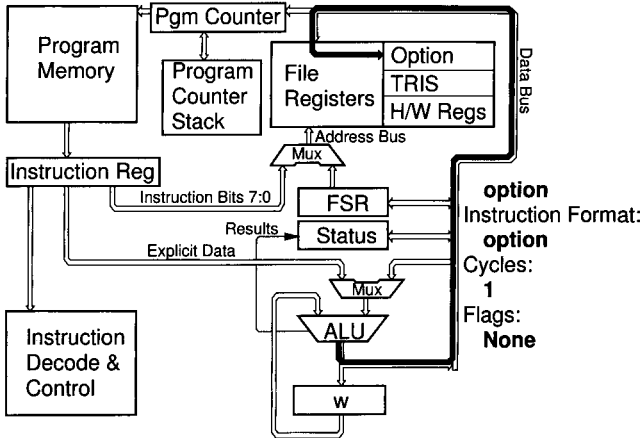


Figure 7.15 The `option` instruction operation.

This instruction copies the contents of `w` into the `OPTION_REG` register (at address 0x081), bypassing the need (in the mid-range PIC microcontrollers) to set the `RP0` bit of the `STATUS` register to set the contents of `OPTION`. As I indicated elsewhere, remember that in the Microchip PIC microcontroller manuals, you will most often see this register referred to as `OPTION_REG`. The reason for this is due to the same label being given to both the `OPTION` register and `option` instruction. In this book, I will use the `OPTION_REG` convention as much as possible to specify the register instead of the instruction, but I will capitalize the register (as I have done with all registers) and put the instruction in lowercase.

`tris` (shown in Fig. 7.16) is used for loading an I/O port driver enable (TRIS) register with the contents of `w` in the same manner as the `option` instruction saves the contents of `WREG` into `OPTION_REG`. The `TRISA`, `TRISB`, and `TRISC` registers can be accessed using these instructions, which have the format:

```
tris    PORTx
```

Microchip does not recommend the use of the `option` and `tris` instructions in the mid-range PIC microcontrollers. These instructions were originally created for the low-end PIC microcontrollers, which do not have the `OPTION` and `TRIS` registers mapped into specific banks (as does the mid-range). Microchip, while continuing their use in the current mid-range PIC microcontrollers, may not continue them in future devices.

As I have indicated elsewhere in the book, I personally don't recommend their use because they do not access all the `PORT` registers in all PIC microcontrollers. If you look at the bit pattern for the instruction:

```
00 0000 0110 0fff
```

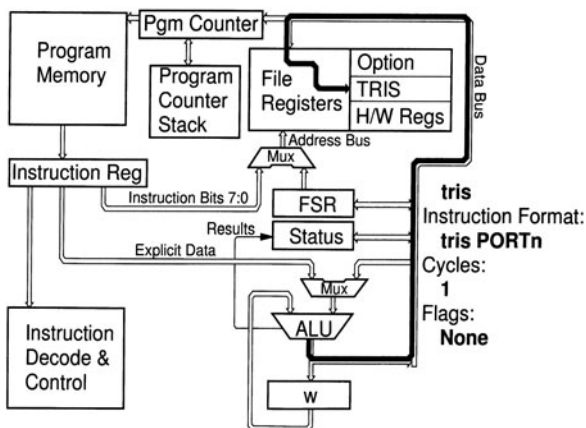


Figure 7.16 The `tris` copy from `w` into the specified `TRIS` register.

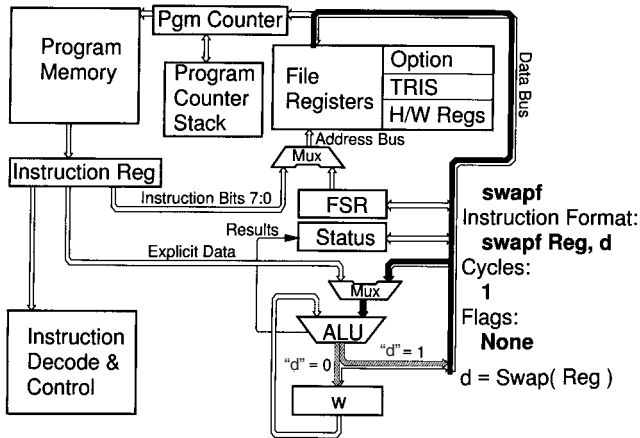


Figure 7.17 The `swapf` nybble exchange instruction operation.

where `fff` is the PORT register written to by the instruction, you will see that the TRIS registers for PORTA (address 5), PORTB (address 6), and PORTC (address 7) are the only ones that can be written to. With PIC microcontrollers that have a PORTD (address 0x008) and PORTE (address 0x009), the `tris` instruction cannot be used because these TRIS registers cannot be accessed by the `tris` instruction. Despite this, there still are times when you may want to use the `tris` and `option` instructions, especially when debugging an application on a mid-range PIC microcontroller that was originally written to be programmed into a low-end PIC microcontroller.

One of the most interesting instructions in the PIC microcontroller is the `swapf` instruction (see Fig. 7.17). This instruction exchanges (or swaps) the contents of the high and low nybbles of the source register and stores the value in `w` or back in the source register depending on the value of `d` in its invocation:

```
swapf Register, d
```

The most obvious use for `swapf` is to use it for displaying a byte as two ASCII nybbles. The code:

```
swapf Register, w    ; Move the Most Significant four bits into the
                      ; Least Significant four bits of "w"
call NybbleDisplay   ; Output the Least Significant four bits of
                      ; "w" as an ASCII Character
movf Register, w     ; Move the byte into "w" without modification
call NybbleDisplay   ; Display the Least Significant four bits of
                      ; "w" as an ASCII Character
```

loads the least significant 4 bits of `w` with the digit to display before calling `NybbleDisplay`, which converts these 4 bits into an ASCII hex code representation.

The example code above will first output the most significant 4 bits of the contents of `Register` followed by the least significant 4 bits.

`swapf` does not modify any of the STATUS flags, which makes it useful in loading `w` without changing any of the STATUS flags. The code snippet:

```
swapf Register, f
swapf Register, w
```

exchanges the high and low nybbles of `Register` and stores the result back into `Register` before exchanging them again and loading the contents into `w`. This double exchange returns the contents of `Register` to the original value for loading into WREG without modifying any of the STATUS register bits. The ability to load `w` with a register value without affecting the contents of the STATUS bits (specifically the zero flag, which is modified by the `movf` instruction) is something that is taken advantage of in PIC microcontroller interrupt handlers. In the PIC18, data movement instructions have been included that do not modify any of the STATUS bits, so this application of `swapf` is not necessary in these PIC microcontroller processors.

I recommend that you demonstrate the operation of the `swapf` instruction using the `InsTemplate.asm` program:

```
movlw    0xF5          ; Load a register with a non-zero value
movwf    0x20

movlw    0x1F          ; Set the Arithmetic STATUS bits
movwf    STATUS

swapf    0x20, f       ; Swap the value in 0x20
swapf    0x20, w       ; and load in "w" reg without affecting STATUS
```

After single-stepping through the program, you will see that the three arithmetic status bits (`z`, `dc`, and `c`) do not change although the value `0xF5` does change in address `0x20`.

The last two instructions used for data movement are the `bcf` and `bsf` instructions that reset or set a specific bit in a register, respectively. The operation of the `bcf` instruction is shown in Fig. 7.18 and is specified in assembly language source code as:

```
bcf      Register, Bit
```

In the `bcf` instruction, the selected `Bit` of `Register` is reset. The operation of this instruction could be characterized as:

$$\text{Register} = \text{Register} \& (0\text{x}0\text{FF} \wedge (1 \ll \text{Bit}))$$

In this operation, the contents of `Register` are ANDed with a value that has all the bits set except for the one that you want to be reset. In the equation above, the `<<` operation shifts the value 1 over `Bit` times to the left. When 1 has been shifted over `Bit` times to the left, it is XORed with `0x0FF`, resulting in the specified bit reset. When this is ANDed with the contents of the register, that bit will be reset.

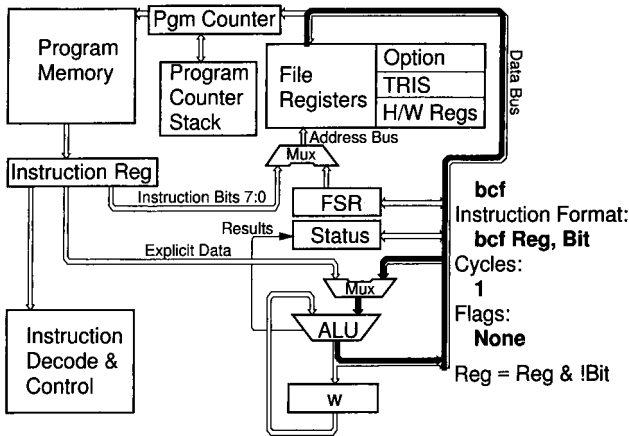


Figure 7.18 The `bcf` single bit clear instruction operation.

The opposite instruction, `bsf`, which sets the register bit specified by the instruction as:

```
bsf    Register, Bit
```

can be characterized by an equation:

$$\text{Register} = \text{Register} \mid (1 \ll \text{Bit})$$

In `bsf`, the value 1 is shifted to the left `Bit` number of times and ORed with the contents of `Register`.

To demonstrate the operation of the `bcf` and `bsf` instructions, I recommend that you single-step through the `InsTemplate.asm` program:

```
movlw  0xDF          ; Save Value to Change
movwf  0x20

bcf     0x20, 1       ; Clear Bit 1
bsf     0x20, 5       ; Set Bit 5
                          ; New Value is 0xFD
```

Both `bcf` and `bsf` are useful instructions when you just want to change the state of a single bit. They are paired with the `btfsf` and `btfsb` instructions that test the state of a register bit and skip the next instruction accordingly. The `bcf` and `bsf` instructions belong in this section because they move new values into registers. The `btfsf` and `btfsb` instructions are in the “Execution Change Instructions” section because they are the primary method that is used to conditionally change how a PIC microcontroller application is executing.

Bank addressing Bank addressing is a difficult concept to understand, even if you are an experienced assembly language programmer. The need for banked register

addressing goes back to how the PIC microcontroller's instructions are implemented; maintaining a single word for all instructions means that jump to addresses and register addresses restrict the amount of memory that can be accessed by the processor. Complicating the issue is the design of the PIC microcontroller and its ability to implement useful programs with sufficient variable memory without having to change the bank register (or the execution address page). Unfortunately, there are registers, such as the I/O port TRIS registers, which are located in another bank. This is the only situation where you need to understand how bank addressing works.

A difficult concept for new developers to understand is that execution really takes place in a single bank and you cannot directly access any other registers except for a few WREGs that are shadowed across multiple banks. One of these registers is the STATUS register, which contains the RP0 and RP1 bits that select which bank is active. Normally when execution starts, the PIC microcontroller defaults to bank 0. To change the active bank, you have to change the RP0 or RP1 to select the desired bank as shown in Fig. 7.19.

To show how this works, start up MPLAB IDE with the InsTemplate.asm project and add the simple program:

```
movlw    0x5A                ; Value to write into registers
movwf    0x20                ; Store at address 0x20

bsf      STATUS, RP0         ; Change execution to Bank 1
movwf    0x20                ; Write to offset 0x20 in Bank 1
```

Single-step through the program and in the File Registers window, 0x5A will be written at address 0x20, which is what you would expect from the program. Continue single-stepping and you will see that the second `movwf` instruction writes 0x5A into address 0xA0 and not 0x20 (as you might expect).

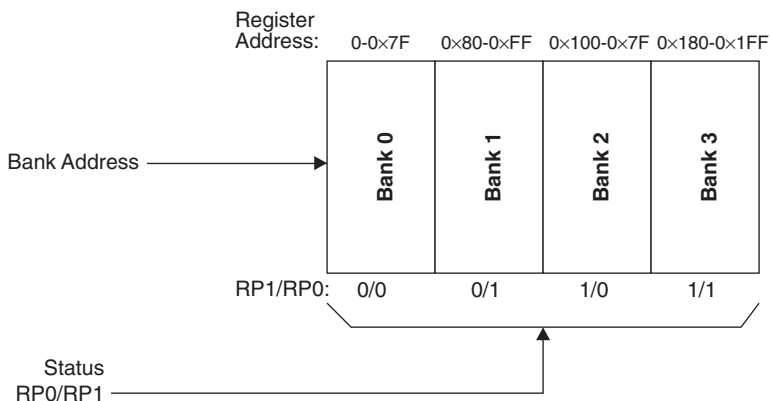


Figure 7.19 Changing the active bank is accomplished by changing the value of the RP0 and RP1 bits in the STATUS register.

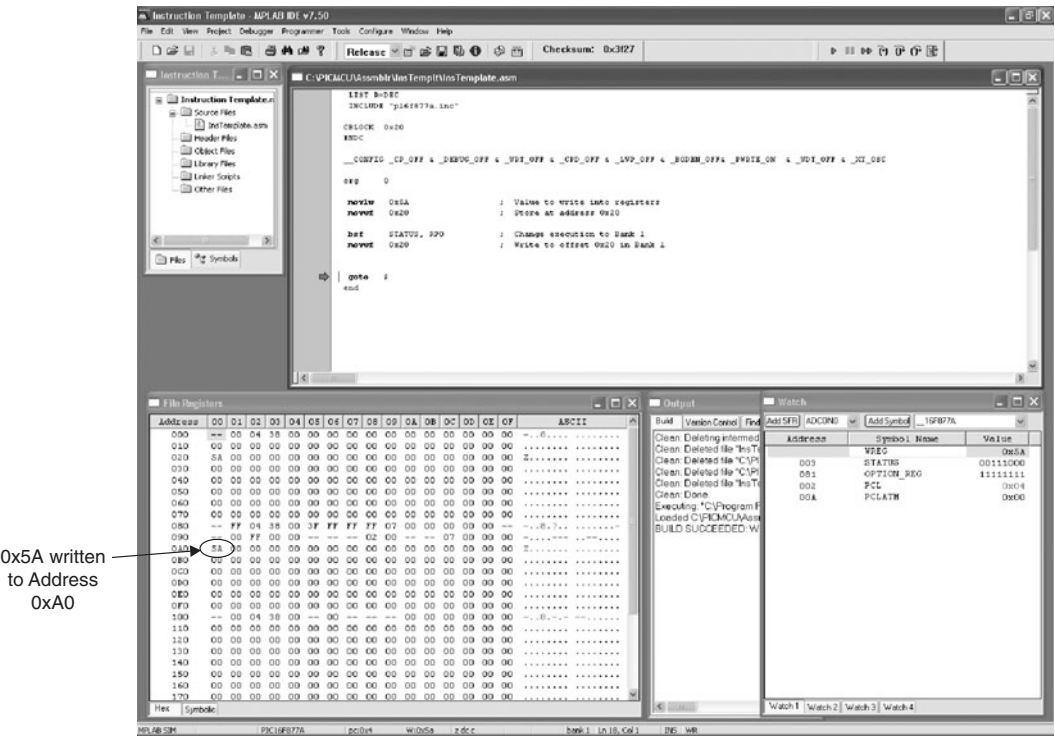


Figure 7.20 After the bank register select bit RP0 is set, a `movwf 0x20` instruction writes a value at register address 0xA0.

The reason for this is the `bsf STATUS, RP0` instruction before the second `movwf`, which changes the currently executing bank from 0 to 1. Now the address written to by the second `movwf 0x20` is writing to the address 0x20 from the start of the bank. Since the start of the bank is 0x80, adding 0x20 to it results in 0xA0, where the byte was written to (Fig. 7.20). To return back to bank 0, all you have to do is reset RP0 using `bcf STATUS, RP0`.

To help you understand how bank switching works, change the value in WREG, change the `movwf` instructions to store data at different addresses within the banks, and select different banks to see the data being written throughout the processor's register space. I do have one word of caution for you: if you write a value to the register at offset 0x2 in the bank, the program will no longer execute correctly. The arrow showing you where the instruction pointer is will disappear and will not return. The register at 0x2 is PCL, which is the least significant 8 bits of the program counter. Writing to this register will change the program counter to some address rather than having it increment normally.

That is all there is to bank switching. What I've just shown you probably seems too easy, especially if you tried to understand bank switching before. Changing banks is very easy, but understanding why you would like to start executing out of a specific bank is

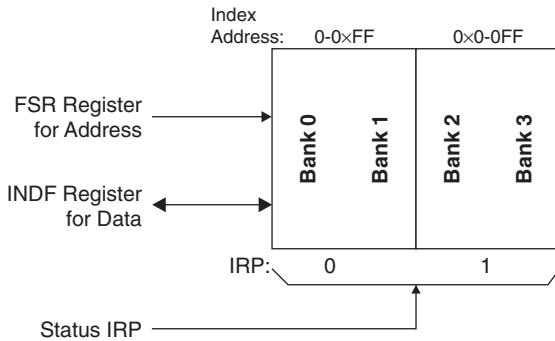


Figure 7.21 To indirectly access a register, load the FSR register with the register's address and read or write to INDF.

a more difficult topic and one that confuses many people when they are working on their first application.

Indexed addressing Indexed addressing is a very straightforward operation in the PIC microcontroller although it is a bit unorthodox. In most processors, to implement an indexed addressing access, the index register is specified as part of the instruction (often enclosed in a pair of parentheses). The mid-range PIC microcontroller works essentially the same, but has a “phantom” register (INDF) which is accessed to cause an index read or an index write. If you are familiar with indexed addressing in other processors' assembly language, you might want to mentally substitute the [indexregister] instruction parameter for the INDF register.

The index register is known as FSR and is 8 bits, which gives it the ability to access up to 256 registers in bank 0 and bank 1 or bank 2 and bank 3. To select between the bank pairs, the IRP bit of the status register is used; when IRP is reset, banks 0 and 1 are selected and when IRP is set, banks 2 and 3 are selected. Figure 7.21 shows a block diagram of how the FSR and INDF registers along with the IRP bit interact.

You can test out the operation of indexed addressing in the mid-range PIC microcontroller using the `InsTemplate.asm` program:

```
movlw 0xAA                ; Load FSR with register in Bank 1
movwf FSR

movlw 0xA5                ; Store a value in the Bank 1 register
movwf INDF
```

After loading in these instructions, build the program and add the FSR register to the Watch window (Fig. 7.22) then single-step through to the `goto $` instruction. At this point, you will see that the register at address 0xAA has been loaded with the value 0xA5 and the FSR register will have the value 0xAA. I chose an FSR value of 0xAA because it is in bank 1 (as can be seen in Fig. 7.20), showing you that you can access bank 1 without changing the RP0 bit.

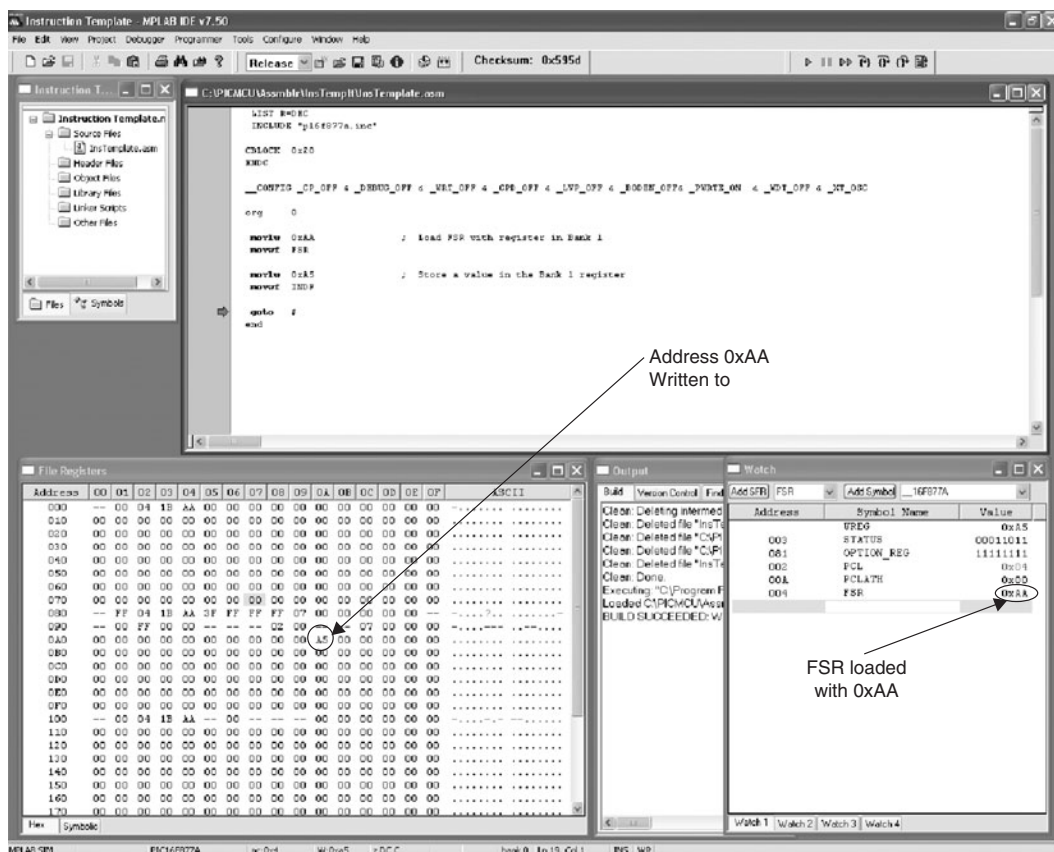


Figure 7.22 Indexed addressing InsTemplate.asm test program operating in the MPLAB IDE simulator.

DATA PROCESSING INSTRUCTIONS

Relatively speaking, the PIC microcontroller does not have a very wide range of instructions that algorithmically or logically change data values. The 7 unique operations (implemented over 15 instructions) available in the PIC microcontroller may not seem to be that comprehensive, but they provide all the basic arithmetic operations needed to implement virtually any application. As I work through the different operations, I will show some simple optimizations and tricks that will help you with your applications as well as explain exactly how the instructions work. Throughout the book, I will be presenting you with algorithms and snippets that will allow you to implement very sophisticated applications, despite the limited number of data processing instructions.

The arithmetic operation that probably comes to mind first is addition. In the PIC microcontroller, addition is carried out in a very straightforward manner, with the contents of the register specified by the `addwf` instruction added to the contents of `w` and the result stored in either the specified register or `w`. The operation of `addwf` is shown in Fig. 7.23.

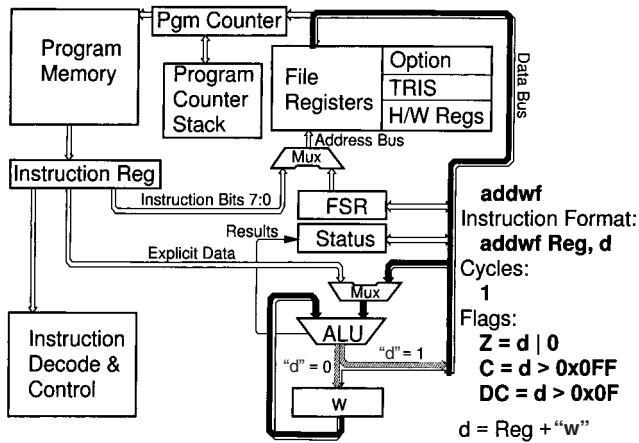


Figure 7.23 The **addwf** register addition instruction operation.

The format used for the **addwf** instruction is:

```
addwf Register, d
```

where **Register** contains the value to be added to the contents of **w**. The **STATUS** register operation bits (carry, digit carry, and zero) are reset or set according to the result of the addition operation as described next.

addlw (Fig. 7.24) is used to add an immediate (**Constant**) value to the contents of **WREG** with the result being stored back into **WREG**. The source code format for the instruction is:

```
addlw Constant
```

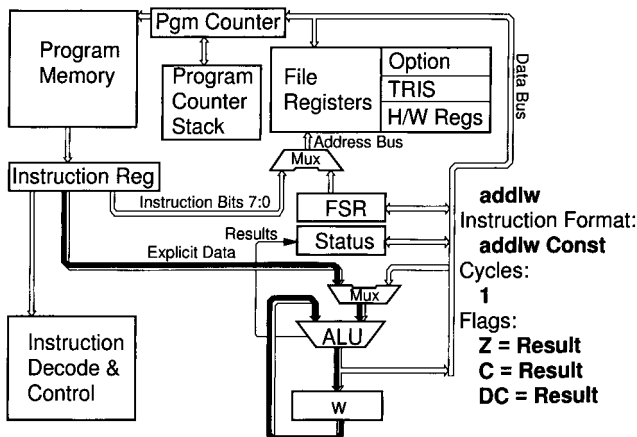


Figure 7.24 The **addlw** immediate addition instruction operation.

As I have indicated in the previous instruction, all the STATUS register operation bits are affected by the addition and subtraction instructions. The zero flag is set if the result ANDed with 0x0FF is equal to zero. The carry flag is set if the result is greater than 0x0FF (255).

The digit carry flag is set when the sum of the least significant 4 bits is greater than 0xF (15). For example, if you had the code:

```
movlw 10                      ; Add 0x0A to 0x0A
movwf Reg
addwf Reg, w                  ; Put the Result in "w"
```

at the end of execution, WREG would contain 20 (or 0x14), Reg have 10 (0xA), the zero and carry flags would be reset (equal to zero), and the digit carry flag would be set because the sum of the least significant 4 bits was greater than 15 (0xF).

I would suggest that you try out the three instructions above, along with three additional instructions, which should set the carry flag and none of the other STATUS register flags:

```
movlw 10                      ; Add 0x0A to 0x0A
movwf 0x20
addwf 0x20, w                  ; Put the Result in "w"/Set DC Flag

movlw 160                     ; Add 0xA0 to 0xA0
movwf 0x20
addwf 0x20, w                  ; Put the result in "w"/Set C Flag
```

After the first three instructions, just the DC flag will be set, and after the next three instructions, just the C flag will be set. The operation of these instructions is very easy to understand for addition—one of the measures of somebody who truly understands how to program the PIC microcontroller is how well they understand the operation of these flags after subtraction instructions.

Subtraction in the PIC will take you some time before you understand the process well enough to be able to implement the function in your own application code. This is not to imply that the instruction works differently, just that it doesn't work as you would intuitively expect, which makes its operation problematic for many people. The `subwf` instruction (see Fig. 7.25) invocation is:

```
subwf Register, d
```

in which the contents of `Register` have the contents of `w` subtracted from it and the result placed either in `w` or `Register` based on the destination parameter. This operation probably doesn't make a lot of sense; the best way to explain subtraction in the PIC microcontroller is to note that it is not subtraction at all. Instead, the `subwf` instruction adds a negative value to the contents of the parameter register.

Instead of `subwf` operating as:

$$\text{Destination} = \text{Source} - w$$

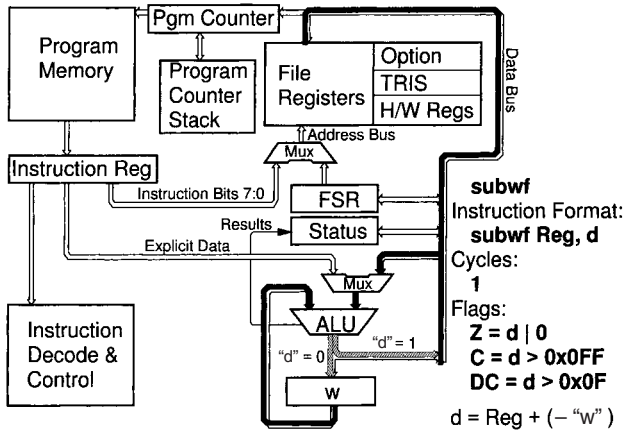


Figure 7.25 The **subwf** instruction subtracts the contents of the **w** register from the contents of the parameter register.

which is what you would probably expect, subtraction in the PIC microcontroller is actually:

$$\text{Destination} = \text{Source} + (-w)$$

The negative w term of the equation above is found by substituting in the 2's complement negation formula:

$$\text{Negative} = (\text{Positive} \wedge 0x0FF) + 1$$

for $-w$, which makes the true operation of the instruction:

$$\text{Destination} = \text{Source} + (w \wedge 0x0FF) + 1$$

I find that when I am using the instruction it helps to remember this formula, because I can easily understand what **subwf** is doing and predict how it will behave. Remembering this formula also helps me to understand how the carry flags work. Looking at the instruction above, the carry and digit carry flags probably run counterintuitively to what you expect (and may have experienced with other processors).

For example, if you were to subtract 2 from 1 in the PIC:

```
Source      = 1
w           = 2
Instruction = subwf Source, w
```

the formula:

$$\text{Destination} = \text{Source} + (w \wedge 0x0FF) + 1$$

is used, which yields (for the `subwf Source, w` instruction):

$$\begin{aligned}w &= \text{Source} + (w \wedge 0xFF) + 1 \\&= 1 + (2 \wedge 0xFF) + 1 \\&= 1 + 0xFD + 1 \\&= 0xFF \\&= -1\end{aligned}$$

This result matches what you would expect. Note that in this case the carry flag is reset, which is *not* what we expect in a typical processor. If a negative result was produced in a traditional processor, we would expect the carry (or an explicit borrow) flag to be set.

Working through the same instruction (`subwf Source, w`) and the registers were loaded with:

```
Source = 2
w       = 1
```

The `subwf` formula values become:

$$\begin{aligned}w &= \text{Source} + (w \wedge 0xFF) + 1 \\&= 2 + (1 \wedge 0xFF) + 1 \\&= 2 + 0xFE + 1 \\&= 0x101\end{aligned}$$

The value 0x1 (0x101 AND 0xFF) is the value actually stored in `w`. But in this case (subtracting a lower value from a higher value), the carry flag (and, possibly, the digit carry flag) are set. If you work through an example where the contents of `w` were equal to `Source`, you will find that the result is 0x100 and the carry flag will also be set in this situation.

The carry flag can be used as a “negative active borrow” flag in your PIC microcontroller applications. When the result is less than zero, the carry flag is reset, indicating that if there were higher order bits, one would have to be taken away (or “borrowed”) from them. When the result of a subtraction in the PIC microcontroller is zero or a value greater than zero, the carry flag is reset, indicating that the result is not negative and there is no need to take away a value from any higher order bits.

For the next program to run in MPLAB IDE, go through the subtraction examples above and before the `subwf` instruction is executed for each case, try to predict the operation of the carry flag, followed by the digit carry flag.

```
movlw 1                ; First test case, 1 - 2
movwf 0x20
movlw 2
subwf 0x20, w          ; Predict the value of "C" after instruction

movlw 2                ; Second test case, 2 - 1
movwf 0x20
movlw 1
subwf 0x20, w          ; Predict the value of "C" after instruction
```


`sublw` instruction executes, try to predict the result; I think you will be surprised at how difficult it is to do.

```
movlw 2          ; First test case, 1 - 2
sublw 1          ; Predict the value of "C" after instruction

movlw 1          ; Second test case, 2 - 1
sublw 2          ; Predict the value of "C" after instruction
```

When I wrote the first edition of this book, I commented that I avoid this instruction except for the case:

```
sublw 0          ; Negate Value in "w"
                ; w = 0 - w
```

This statement is probably a bit harsh because after spending some time, you will understand how the `sublw` instruction works and you will see where it can be used in your application.

I must point out that the `sublw` does not work as you would expect it; subtracting a constant value from the contents of WREG. Going back to the equations above, you will have seen that the `sublw` doesn't subtract a constant from the contents of WREG; instead it subtracts the contents of WREG from the constant. If you are looking for the capability of subtracting a value from the contents of WREG, I would suggest that you add the negative as shown by the instruction:

```
addlw -47
```

If you wanted to carry out this same operation using the subtraction instructions, the required code would be:

```
movwf Temp
movlw 47
subwf Temp, w
```

which takes up three times the number of instructions and requires one extra variable as well.

Registers can have their contents incremented (one added to it) or decremented using the `incf` and `decf` instructions. (Figure 7.27 shows how data flows through the process when the `decf` instruction executes.) To add 1 to a register, it is incremented, and taking 1 away from a register is decrementing. To invoke the instructions, the format

```
incf Register, d
```

is used for incrementing a register and

```
decf Register, d
```

is used for decrementing the register.

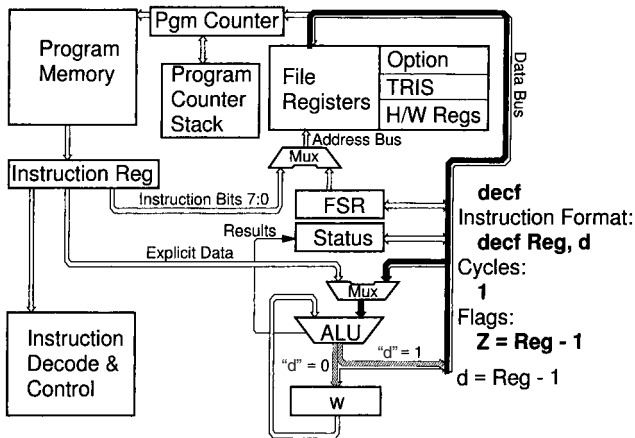


Figure 7.27 The **decf** instruction decrements the contents of a register by 1.

To decrement the value, instead of subtracting 1 from the register, 0xFF (–1) is added to it. This saves the cost of a subtractor built into the circuit and uses the zero flag circuitry of the adder to implement the function.

The result of the increment and decrement can be stored back in the original register (d set to f or 1) or into WREG (d set to w or 0). Upon completion of the instruction, the zero flag of the STATUS register is either set (the result is equal to zero) or reset (the result was not equal to zero). The carry and digit carry flags are not affected by the operation of these two instructions.

comf inverts the contents of a register; its operation is shown in Fig. 7.28. This operation is the same as XORing the contents of a register with 0xFF to invert or complement each bit. To invoke **comf**, the instruction has the format:

comf Register, d

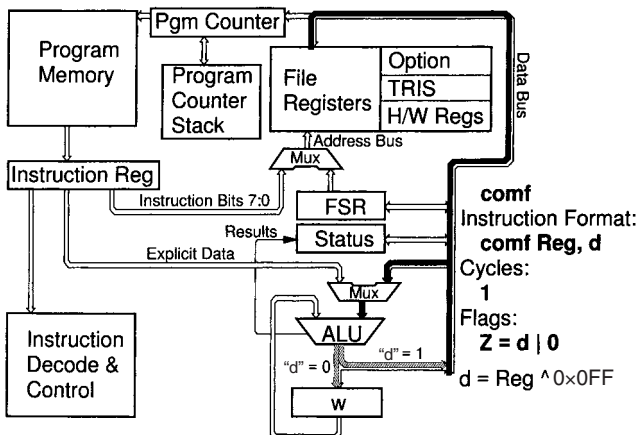


Figure 7.28 The **comf** register complement instruction.

It is important to remember that complementing a register is not the same as negating it. To two's complement operation negates the contents of a register, the register has to be complemented and then incremented. To negate a `Register` in the PIC microcontroller, the following two instructions should be used:

```
comf    Register, f
incf    Register, f
```

To demonstrate the operation of the increment and decrement instructions as well as showing how the contents of a register can be complemented, put the following application code into the `InsTemplate.asm` MPLAB IDE project. Along with this, click on the line below the last line of the Watch window and in the Address column enter 20. After the value comes up, right-click on the value in 0x20 and click on Properties. In the window that comes up, specify that the contents of 0x20 are to be displayed in signed decimal format (select Decimal and click on the Signed box that comes up).

```
movlw   47                ; Start with a known value
movwf   0x20

incf     0x20, f           ; Increment the contents of 0x20
incf     0x20, f
incf     0x20, f

decf     0x20, w           ; Decrement contents of 0x20 and put in "w"
decf     0x20, w
decf     0x20, w           ; Check the value in 0x20

comf     0x20, f           ; Two's complement negate the value in 0x20
incf     0x20, f
```

At the end of the instructions, the contents of 0x20 should have -50. It was loaded with 47, incremented three times (to get to 50) and then decremented with the result being stored in WREG, which means that the contents of 0x20 never change. Finally, the value in 0x20 (50) is complemented and incremented, producing the result -50. As with all programs presented in this chapter, I highly encourage you to change values and experiment with the instructions to help you understand exactly how they should work.

I should point out that you can repeat the negation operation of the contents of file register 0x20 by double-clicking on the "PC:A" text on the bottom MPLAB IDE toolbar and entering 0x8 to move to the instructions before the `comf` instruction and then single-step through. When you return to the infinite loop statement (the `goto $`), you will discover that the contents of 0x20 have been turned back to the value before the first negation (50).

Along with arithmetic operations, the PIC microcontroller can also perform the bitwise logical functions AND, OR, and XOR. These instructions are available for combining the contents of a register along with the contents of w or combining the contents of w with a constant value. When 2 bits are ANDed together, the result will be 1 if both

inputs are 1. If either input is not 1, then the result will be 0. The PIC microcontroller has two AND instructions, `andwf` and `andlw`, that perform these functions.

To invoke `andwf`, use the instruction format

```
andwf Register, d
```

where `d` can specify that the result of ANDing the contents of `w` with `Register` is placed into `w` or back into `Register`.

`andlw` is invoked using the instruction format

```
andlw Constant
```

In both AND instructions, the zero flag is set when the result of the AND operation is equal to zero. Neither the carry nor the digit carry flags are affected.

You may find it a bit confusing to discover that Microchip refers to what I call ORing bits together as “inclusive ORing.” In this operation, when either bit of an OR input is set (1), then the result will be set (1). There are two instructions that execute this function, `iorwf` (Fig. 7.29) and `iorlw`.

`iorwf` is invoked using the instruction format

```
iorwf Register, d
```

ORing a constant with the contents of `w` is accomplished by the `iorlw` instruction, that has the format:

```
iorlw Constant
```

In both PIC microcontroller inclusive OR instructions, the zero flag is set if the result of the operation is zero.

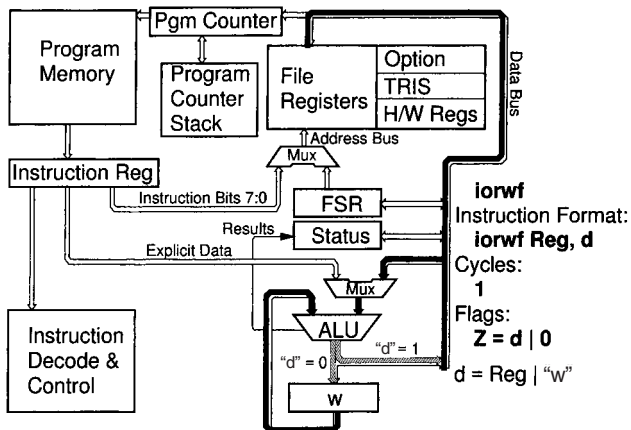


Figure 7.29 The inclusive OR instruction `iorwf` operation.

The last bitwise logical operation is the exclusive OR or XOR. In this operation a bit is set if a single input bit is set and the other is reset. If both inputs are at the same state (set or reset), the result will be reset. Like ANDing and ORing, XORing in the PIC microcontroller can be done with either the contents of a register being XORed with the contents of *w* or the contents of *w* are XORed with a constant.

The `xorwf` instruction XORs the contents of a register with *w* and stores the result according to the value of the destination *d* in the format:

```
xorwf Register, d
```

To XOR the contents of *w* with a constant and place the result back into *w*, the `xorlw` instruction (Fig. 7.30) is used:

```
xorlw Constant
```

Like ANDing and ORing, XORing will set the zero register if the result of the operation is equal to 0x00.

The rotate left (`rlf`, Fig. 7.31) and rotate right (`rrf`) instructions are useful for a number of reasons. Their basic function is to move a register 1 bit to the left (upward) or to the right (downward), with the least significant value being loaded from the carry flag, and the most significant value put into the carry flag.

The `rlf` instruction will rotate the contents of a register to the left (up) and has the format:

```
rlf Register, d
```

When this instruction executes, the contents of the STATUS carry flag are stored in the least significant bit of the destination (either *w* or *Register*) while the contents of *Register* are shifted up by 1 bit. To shift a register up by 1 bit, the contents of bit 0 are stored in bit 1, the contents of bit 1 are stored in bit 2, and so on. When the register

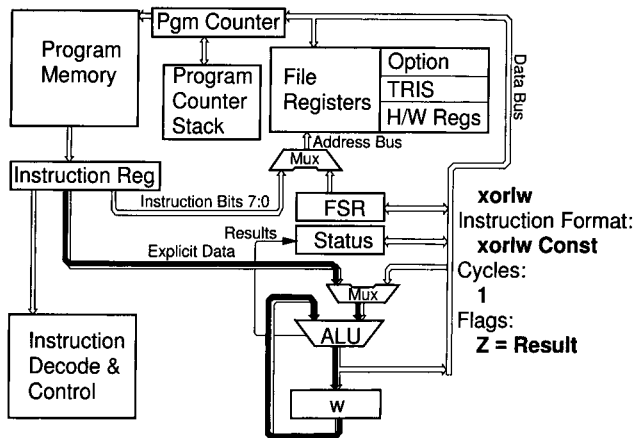


Figure 7.30 The `xorlw` instruction XORs the contents of WREG with a constant.

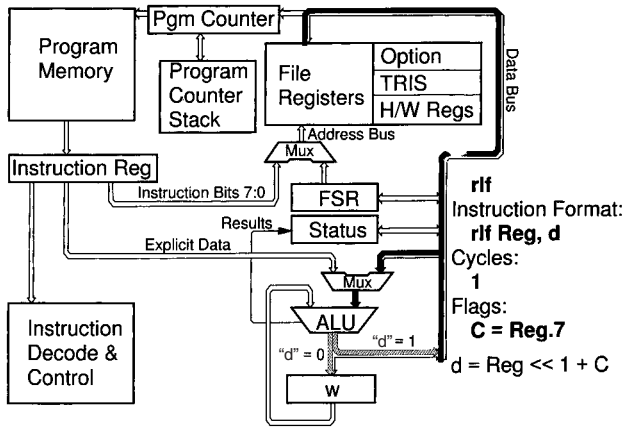


Figure 7.31 The `rlf` instruction rotates the contents of the register to the left one time.

is shifted up by 1, bit 0 is left open and is given the contents of the carry flag. Bit 7 of Register is stored in the STATUS carry flag to complete the rotation.

To rotate a register to the right by 1 bit, the `rrf` instruction is used:

```
rrf    Register, d
```

Instead of moving the registers up, in a right rotate the registers are moved down; the contents of bit 7 are stored in bit 6, the contents of bit 6 are stored in bit 5, and so on. The contents of the carry flag before the invocation of the instruction are stored in bit 7 and upon completion of the `rrf` instruction, the original contents of bit 0 of Register are stored in the carry flag.

A little trick that can be used to rotate a register and not lose a bit is to execute the snippet:

```
rrf    Register, w
rrf    Register, f
```

In the first instruction, the carry flag will be loaded with the contents of bit 0 and the shifted value will be placed into w, where it can be ignored. The second instruction then repeats the rotate instruction with carry loaded with its least significant bit, which is placed in the most significant bit of the destination. This trick also works for the `rlf` instruction.

The operation of the rotate instructions can be illustrated by the `InsTemplate.asm` program:

```
movlw   1 << 4           ; Load 0x20 register with just bit 4 set
movwf   0x20

rlf     0x20, w           ; Rotate 0x20 until the set bit rolls over
rlf     0x20, f           ; to bit 0
```

```
rlf      0x20, w
rlf      0x20, f

rlf      0x20, w
rlf      0x20, f

rlf      0x20, w
rlf      0x20, f
```

The rotate instructions can be used for doing multiplication and division on a value with powers of 2. This can also be done on 16-bit values. The following example shows how to multiply a 16-bit number by 4:

```
bcf      STATUS, C           ; Clear the Carry flag before Rotating
rlf      Reg, f              ; Shift the Value up (Multiply by 2)
rlf      Reg + 1, f
bcf      STATUS, C           ; Now, Repeat to Multiply by 2 again
rlf      Reg, f
rlf      Reg, f
```

EXECUTION CHANGE OPERATORS

Before attempting to use a `goto` or `call` instruction, it is imperative that you understand how the mid-range PIC microcontroller program counter works. If you haven't done so, go back and read Chap. 6, with an emphasis on "The Program Counter and Stack". The `goto` and `call` instructions of the mid-range PIC microcontroller processors are different from many other small processors' `goto` and `call` instructions. The reason for the difference is the requirement that all mid-range PIC microcontroller instructions have to be the same length, and this does not leave sufficient space to access all the memory available to the architecture. Many books and tutorials introducing new developers to the PIC microcontroller architecture don't go into detail explaining how the PCLATH register bits work with respect to the `goto` and `call` instructions. I believe this is a mistake because there will be cases (such as tables, which are discussed next) where you will have to understand the operation of the program counter to be able to properly implement items in your application.

Both `goto` and `call` explicitly specify an absolute address within a mid-range PIC microcontroller page, which is 2048 instruction addresses. This limitation comes from the 14-bit size of the instruction word; the instruction has to have sufficient bits for both the instruction coding and for the new address. In the mid-range chips, 3 bits are devoted to the instruction coding, leaving 11 bits for the instruction address. This limitation is not a problem for chips like the PIC16F84A, which is often cited as a device people start working with first because it has 1024 instruction addresses, less than the page size of the architecture. The issue arises when the device (and the application code) have more than 2048 instructions—then there has to be a way of specifying `gotos` and `calls` into other pages.

If the destination address is outside the page, the PCLATH register is used to set the correct execution page. This register has 5 bits in it, 2 of which are appended to the

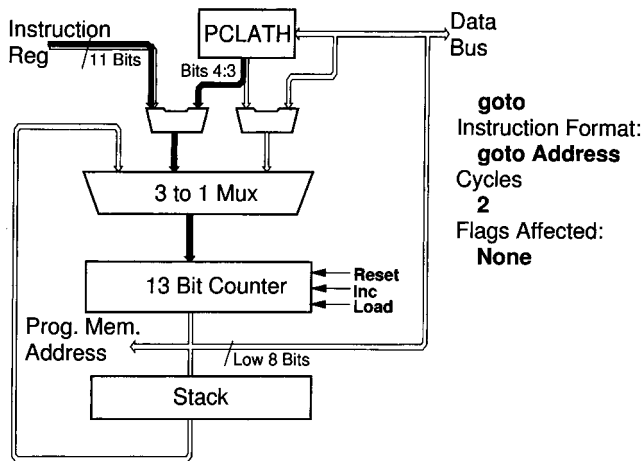


Figure 7.32 When the `goto` instruction executes in the mid-range PIC microcontroller, the 11 address bits in the instruction are appended to 2 of the PCLATH register bits to select the new actual address within the page.

11 address bits specified by the instruction. The operation of the `goto` instruction is shown in Fig. 7.32 with the 11 bits having the 2 PCLATH bits (4 and 3) added to the address to create a full 13-bit address able to access the entire mid-range PIC microcontroller address range. This operation takes place for all mid-range chips, not just the ones with the full 8k of program memory allowed by the 13-bit address space.

For example, jumping between pages in the mid-range PIC microcontroller can be accomplished by the code:

```
movlw  HIGH Label           ; Interpage "goto"
movwf  PCLATH
goto   (Label & 0x7FF)      ; Address within the page
```

In this snippet of code, the PCLATH register is updated with the new page before the `goto` instruction is executed. This forces the program counter to be loaded with the correct and full `Label` address when the `goto` instruction is executed.

Execution for all the instructions that change the program counter will take two instruction cycles instead of the customary one of the data movement, data processing, or processor control instructions. This is caused by the PIC microcontroller's instruction register being already loaded with the instruction at the next address and then having to be loaded with a new address and prefetch the instruction at that address before it can be executed. The actual timing for the operation is two cycles, as is shown in Fig. 7.33. When the `goto` (or any other program counter changing) instruction is executed, the prefetched instruction (`Goto + 1` in Fig. 7.33) is no longer valid and it must be changed with the `Destination` instruction. Before the `Destination` instruction can be executed, it must be loaded into the PIC microcontroller's prefetch register so that it can be executed

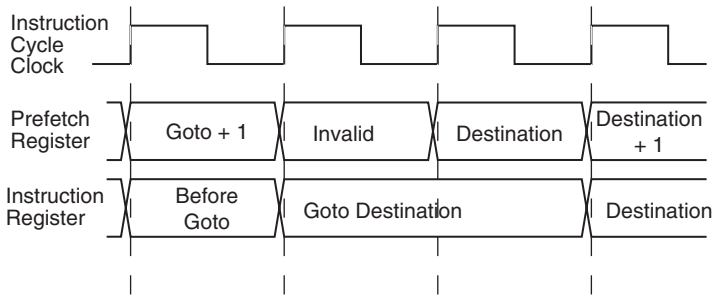


Figure 7.33 When the program counter is changed, the prefetch register must be loaded with the new address followed by the instruction at this address before execution can start at the new address. This process takes two instruction cycles.

in the next instruction cycle. This is the second instruction cycle after the `goto` instruction rather than the first, which would be the case for any of the other instructions that don't change the PIC microcontroller's program counter.

The operation of the `goto` instruction can be demonstrated in the following `InsTemplate.asm` program:

```

goto    FirstLabel
movf    STATUS, w           ; Execution should not take place here

FirstLabel:                ; This should be 2nd instruction to execute
    movlw HIGH SecondLabel ; Do an Interpage goto
    movwf PCLATH
    goto  SecondLabel & 0x7FF

    org 0x1B76              ; Page 3 Address
SecondLabel:                ; Execution at "goto $" that follows

```

The `org` directive specifies the address where the following instructions are placed. The `$` directive used in `goto $` returns the current address, so the instruction literally means “jump to this address” and forms an infinite loop that execution cannot escape from. These two directives will probably be used in every assembly language program you write.

The `call` instruction (shown in Fig. 7.34) works almost exactly the same way as `goto`, except the pointer to the next instruction is stored on the program counter stack.

There are three types of return statements in the mid-range PIC and PIC18 microcontrollers. Each of these instructions pops the current value from the top of the hardware stack and stores it in the program counter. These addresses are the next instructions that were saved when the subroutine was called or an interrupt handler was completed. The simple `return` instruction (Fig. 7.35) returns the stack pointer to the address pointed after the instruction calling the subroutine and no registers or control bits are changed.

Another of the return instructions is `retlw`, which loads `w` with a constant value before returning from a subroutine (Fig. 7.36). This instruction is the only return

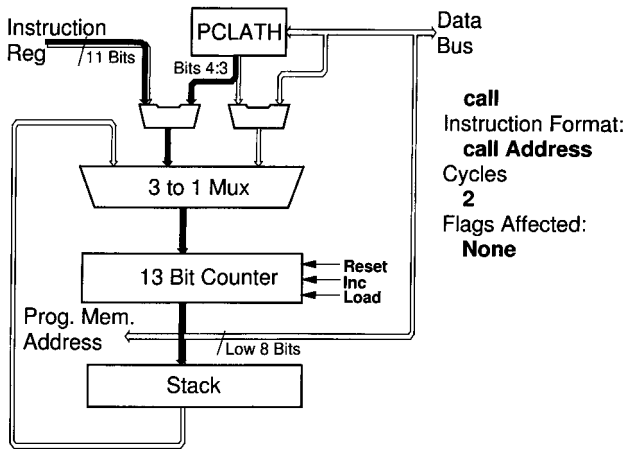


Figure 7.34 Before the program counter is loaded with the 13-bit address from the instruction and PCLATH register in the **call** instruction, it is stored on the program counter stack.

instruction available in the low-end devices. This instruction is useful for implementing tables that return constant values. The operation of the **retlw** instruction:

```
retlw Constant
```

could be simulated in the mid-range PIC microcontroller by the use of the two instructions:

```
movlw Constant  
return
```

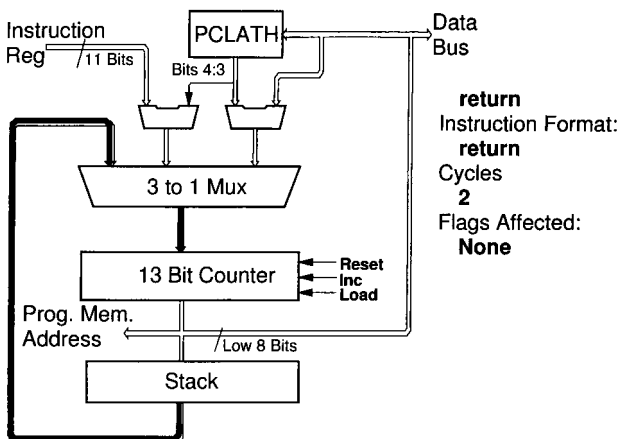


Figure 7.35 The **return** instruction will be pulled from the stack and stored into the program counter.

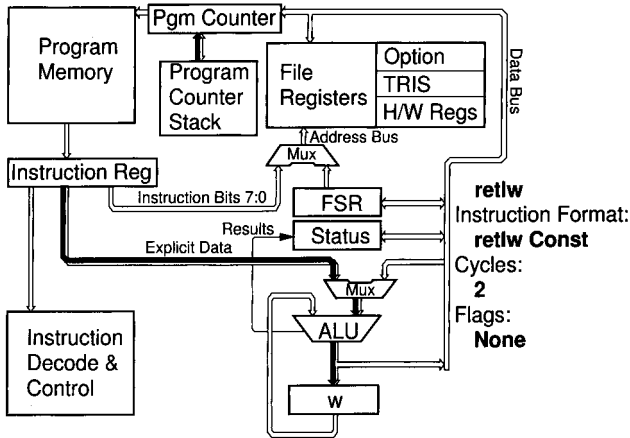


Figure 7.36 The `retlw` instruction loads literal value into `W` and returns from subroutine instruction operation.

The instruction is useful for returning table information (which is explained in the next section) or returning condition information in `w` that can be tested by the calling program. The instruction loads `w` with an immediate value before executing a return from interrupt.

The `retfie` instruction is similar to the `return` instruction except that it is used to return from an interrupt; in fact, the only difference between `retfie` and `return` is that the `GIE` bit in the interrupt control register (`INTCON`) is set during the `retfie` instruction. This allows interrupt requests to be acknowledged immediately following the execution of the instruction and the interrupt handler.

As I have indicated elsewhere in this book, the PIC microcontroller architecture doesn't use "jump on condition" instructions. Instead, there are a number of instructions that allow skipping the next instruction in line based on specific conditions. The basic instructions that carry out this function are the "skip on bit condition" instructions, `btfsf` and `btfsb`. These two instructions use the same format as the bit set and reset instructions (`bcsf` and `bsf`) although they test the condition of a bit rather than specify its state.

The `btfsf` instruction skips over the next instruction if the bit condition is reset (0). The format of the instruction is:

```
btfsf Register, Bit
```

`btfsb` skips over the next instruction if the bit condition is set (1). The format for the instruction is:

```
btfsb Register, Bit
```

If the bit condition is not true and the skip is not executed, then the `btfsf` and `btfsb` instructions execute in one instruction cycle. If the condition is true, then the instruction executes in two cycles, essentially treating the following instruction like a `nop`.

These two instructions are used as the basic method of conditionally changing the execution of an application. For example, to jump to an address if the zero flag is set, a `goto` is added to the `btfsf` instruction:

```

btfsc STATUS, Z ; Test Zero, Skip over Next if Zero Flag is Reset
goto Label      ; Zero Flag Set — Jump to "Label"

```

Notice that in the snippet above I test for the negative condition and skip over the next instruction if it is true. If you learned programming in an environment that didn't have structured programming instructions, you should be familiar for this. To explain what I mean, consider the structured `if/else` code in C:

```

if (a == b) {

// Instructions to execute if "a" equals "b"

} else {

// Instructions to execute if "a" does NOT equal "b"

}

```

To execute this in an unstructured language, you would code it as:

```

if (a != b) then goto aNotEqualb;

// Instructions that execute if "a" equals "b"

goto ConditionEnd

aNotEqualb:

// Instructions that execute if "a" does NOT equal "b"

ConditionEnd:

```

To implement the nonstructured case, you have to invert the test. To implement the structured `if/else` C code in PIC assembler, you would write it as:

```

movf    a, w          ; "if (a != b) then goto ANotEqualb
subwf   b, w
btfss   STATUS, Z
goto    ANotEqualb

// Instructions that execute if "a" equals "b"

goto ConditionEnd

aNotEqualb:

// Instructions that execute if "a" does NOT equal "b"

ConditionEnd:

```

The actual conditional execution code can be written in such a way to simulate structured code, and in fact, when you first start writing PIC microcontroller assembly language code you can block it out using pseudo C code, like the example above, and then convert it to assembly code using the two steps I show here. Doing the correct bit condition test is something to be cognizant of when you first start working with the PIC microcontroller.

Conditional jumps based on a comparison of two values follow a very definite format that you can use when you first start programming. From a high level, the code implements:

```
if (A <i>condition</i> B) then goto Label
```

The basic instruction format is:

```
movf    FirstValue, w
subwf   SecondValue, w
btfs#   STATUS, flag
goto    Label
```

where FirstValue, SecondValue, #, and flag are defined in Table 7.1. If either of the conditional values are constants, then movf or subwf are replaced with movlw or sublw, respectively. In Table 7.1, I included the complement tests for when you are creating your own structured code.

TABLE 7.1 PARAMETER VALUES TO IMPLEMENT goto ON CONDITION					
CONDITIONAL STATEMENT	FIRSTVALUE	SECONDVALUE	btfs#	STATUS FLAG	COMPLEMENT STATEMENT
if (a == b) then goto Label	a	B	btfs#	Z	if (a != b) then goto Label
if (a != b) then goto Label	a	b	btfs#	Z	if (a == b) then goto Label
if (a > b) then goto Label	a	b	btfs#	C	if (a <= b) then goto Label
if (a >= b) then goto Label	b	a	btfs#	C	if (a < b) then goto Label
if (a < b) then goto Label	b	a	btfs#	C	if (a >= b) then goto Label
if (a <= b) then goto Label	a	b	btfs#	C	if (a > b) then goto Label

TABLE 7.2 COMBINING `btfsc` AND `btfss` WITH `goto` TO CREATE JUMP ON STATUS FLAG CONDITIONS

CONDITIONAL JUMP	<code>btfsc/btfss</code> AND <code>goto</code> INSTRUCTIONS	COMPLEMENT JUMP
<code>jz Label</code>	<code>btfsc STATUS, Z</code> <code>goto Label</code>	<code>jnz Label</code>
<code>jnz Label</code>	<code>btfss STATUS, Z</code> <code>goto Label</code>	<code>jz Label</code>
<code>jc Label</code>	<code>btfsc STATUS, C</code> <code>goto Label</code>	<code>jnc Label</code>
<code>jnc Label</code>	<code>btfss STATUS, C</code> <code>goto Label</code>	<code>jc Label</code>

If you have experience programming in assembly language with other chips, you might wonder why there aren't any "jump on STATUS flag condition" instructions. The `btfsc` and `btfss` instructions with the `goto` instruction can provide you with this capability, as listed in Table 7.2.

Built into the MPLAB assembler, there are a number of "pseudo-instructions" like these that provide similar functions. Personally, I would recommend that you avoid using these instructions because they mask the true usefulness of the `btfsc` and `btfss` instructions and what is actually happening in the program.

The discussion on simulating the conditional jump instructions of other processors and implementing simple two-parameter conditional jumps can distract people from the true power of having an architecture that can branch on the condition of any bit in the register address space. Having these instructions avoids the need to use additional instructions to test the value of an arbitrary bit, as would be required in other processors. Flag bits can be very easily changed in the PIC microcontroller using the `bcf` and `bsf` instructions and tested using the `btfsc` and `btfss` instructions. To show you how these instructions can be used, consider the instructions needed to set a bit in another processor. Using PIC instructions the code would look like:

```
movf   Register, w
iorlw  1 << Bit
movwf  Register
```

while in the PIC microcontroller, the following single instruction is required:

```
bsf    Register, Bit
```

Similarly, if a bit is to be tested in a register and execution jumps to a label if it is set, a traditional processor would require the code:

```
movf   Register, w
andlw  1 << Bit
jnz    Label
```

In the PIC microcontroller, the code would be:

```
btfsc Register, Bit
goto Label
```

In both cases, the PIC microcontroller is able to change a bit and test its condition in fewer instructions and without affecting the contents of the accumulator or the STATUS register. This very powerful feature is something to keep in mind when you are developing your applications. By spending a few minutes planning on how the registers are going to operate, you can improve the efficiency of your application by some remarkable margins.

Bit skip instructions are useful in a variety of cases, from checking interrupt active flag bits to seeing if a number is negative (checking the most significant bit). Throughout the book, I present a number of ways in which the bit skip commands can be used to simplify the software development. The bit commands are quite unique to the PIC compared to other microcontrollers and I believe they give the application developer a significant advantage in developing simple applications that quickly and easily allow you to efficiently manipulate bits in a variety of ways.

To finish off the discussion of bit skip and `goto` instructions, I should point out that there are a number of ways to conditionally jump to an address in another address page. Probably the best way of doing this is to put the PCLATH change before any calculations that are required before the conditional `goto` as well as restoring the PCLATH register after the conditional `goto`. The restore is required in case the conditional `goto` doesn't execute and sometime later you are jumping within the current page; if the PCLATH register is not restored to its original value, then you will be jumping to an invalid address in another page. The PCLATH register restore, shown in the example conditional interpage `goto` below, avoids the possibility of forgetting to restore PCLATH:

```
movlw HIGH LongJump ; PCLATH Set up for goto
movwf PCLATH
movf a, w ; Conditional goto calculations
subwf b, w
btfss STATUS, Z
goto (LongJump & 0x7FF) | ($ & 0x1800)
movlw HIGH $ ; PCLATH restore
movwf PCLATH
```

The `goto` statement might seem a bit more complex than is necessary, but the $(\$ \& 0x1800)$ ORed with the page offset of the destination will add the appropriate high order bits to avoid an assembly message stating that the address is outside of the current page. This code is like the XORing of the register address bits to prevent any messages being produced by MPASM that indicate that an invalid access is taking place.

Along with the bit test (`btfsc` and `btfss`) instructions, there are two other instructions that skip on a given instruction. They increment or decrement a register and skip

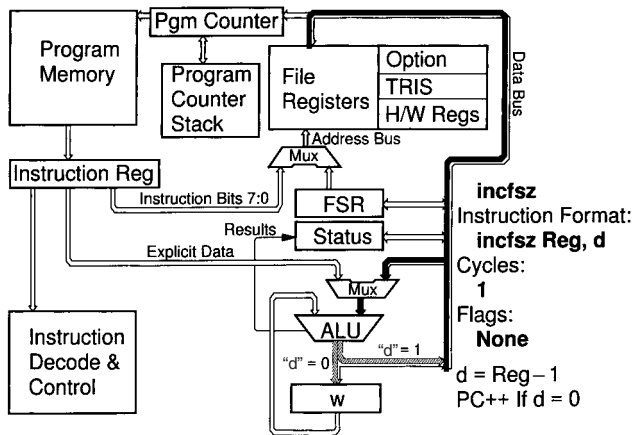


Figure 7.37 The **incfsz** instruction increments the contents of the specified register and skips the next one if the result is equal to zero.

the next instruction if the result is equal to zero. Figure 7.37 shows the operation of the **incfsz** instruction while the instruction format is:

incfsz Register, d

In the **incfsz** instruction, the contents of **Register** are incremented and stored according to the value of **d**. If **d** is **w** or **0**, then **WREG** is stored with the result of the operation. If **d** is **f** or **1**, then the **Register** is updated with the result of the operation. No **STATUS** flags are modified by the operation of **incfsz** or **decfsz** (which is a difference between them and the **incf** and **decf** instructions).

decfsz is similar in operation to **incfsz** except the **Register** is decremented and the skip takes place if the result is equal to zero. Its format is:

decfsz Register, d

These two instructions work exactly the same as the **incf** and **decf** instructions in terms of data processing: 1 is added or subtracted from **Register**. The result is then stored either in **w** or back in the source register. The important difference is these instructions is that if the result is equal to zero following the increment/decrement, the next instruction is skipped.

If the result is not equal to zero (and the next instruction is not skipped), **incfsz** and **decfsz** execute in one instruction cycle. **incfsz** and **decfsz** execute in two instruction cycles if the result is equal to zero. If the result is zero, then the next instruction is skipped over and treated like a **nop**. Often these instructions are used in critically timed loops, so understanding the exact timing of the two instructions can be critical.

This means that **decfsz** and **incfsz** can be used for loop control operations. Actually, I should say that **decfsz** is normally used for loop control. The code example

below can be added to `InsTemplate.asm` to show how a loop can be repeated 37 times with very little software overhead:

```
    movlw    37                ; Load the Count Register
    movwf    0x20
Loop:                ; Repeat for each iteration of the loop

; Instructions in the loop normally go here

    decfsz   0x20, f          ; Decrement the Count Register
    goto     Loop            ; If not == zero, loop again
                                ; Continue on with the Program
```

This code can be used anywhere a loop is required and, as you can see, the code required to implement it is only four instructions long and only requires two or three instruction cycles each loop.

A couple of notes on the `incfsz` and `decfsz` instructions: if you are using them on processor registers, care should be taken to ensure that the hardware registers are capable of reaching zero. In the low-end PIC microcontrollers, the FSR can never be equal to zero, which makes the `incfsz` and `decfsz` instructions useless.

As well, these instructions do not affect any status flags (zero would probably be expected). This means that you may want to put a `bsf STATUS, Z` after the instruction following the `incfsz/decfsz` instruction to indicate that the loop has finished because the variable being decremented has reached zero:

```
    decfsz   Count           ; Decrement the Count Value
    goto     Loop            ; Jump back to Loop if Count != 0

    bsf      STATUS, Z        ; Set Zero Flag to Indicate Loop End
```

Tables So far I have touched upon explicitly changing the contents of the PIC microcontroller's program counter to provide explicit jumps within an application. As I work through some of the more advanced programming techniques that can be used with the PIC microcontroller, the need to arithmetically calculate an address to jump to will become more obvious. The table programming construct in the PIC microcontroller is one that requires arithmetically calculated `goto` addresses that I am sure you will use a lot of in your application programming.

When implementing a PIC microcontroller application that can communicate with humans, the ability to send text messages will be required. Tables of text messages can be implemented in the PIC microcontroller quite simply with the advantage that they will execute quickly and with a consistent number of cycles, no matter where the data in the table to be retrieved is.

The most traditional method of implementing a table is to provide a subroutine that adds a constant to a known point in the application and stores this value in the PIC microcontroller's program counter. At the new address, a `retlw` instruction is used to store the table value in WREG and return to the caller's code.

The most basic way of doing this is to use the `addwf PCL, f` instruction to update the PIC microcontroller's program counter with the table immediately following it as the table's destination addresses. The most basic version of this subroutine could be:

```

Table:                                ; Return Table Value for Contents of "w"
    addwf PCL, f                      ; Add the Table Index to the Program Counter
    retlw "T"                        ; ASCII "Table" to be returned
    retlw "a"
    retlw "b"
    retlw "l"
    retlw "e"
    retlw 0

```

The `addwf PCL, f` instruction (which is shown in Fig. 7.38) adds the contents of `w` (which is the table value to return that has been passed to the `Table` subroutine) to the program counter via `PCL`. When the `addwf PCL, f` instruction executes, the program counter is already incremented to the next instruction, so to return the `T` in the table, a value of zero has to be passed in `w`. To return `a`, a value of 1 is passed in `w` and so on.

The zero value at the end of `Table` is used to indicate that the table value has ended. Normally when I am using a text table like this one, I want to have some way of determining when I am at the end of the table; a NUL character (ASCII 0x00) is my usual choice. I like ending a table with 0x00 because when it is ORed with 0x00 or ANDed with 0xFF, the zero flag will be set without changing the value of the contents of `WREG`.

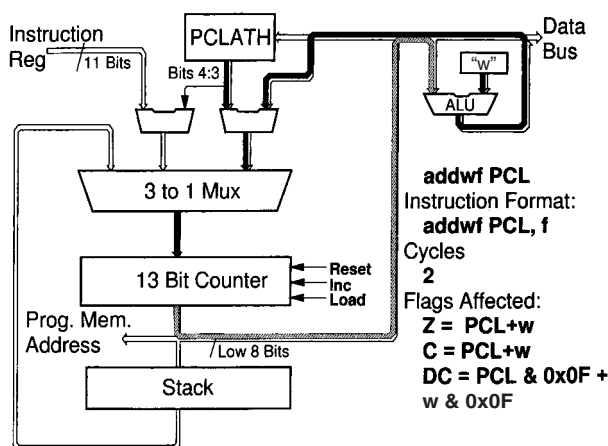


Figure 7.38 The `addwf PCL, f` instruction adds the contents of `WREG` to the lower 8 bits of the program counter (`PCL` register) to compute a new execution address.

The `Table` subroutine above can be enhanced by using the `dt` assembler directive (command), which will combine the table's `retlw` instructions. Using `dt`, the subroutine becomes:

```
Table2:                ; Return Table Value for Contents of "w"
    addwf PCL, f        ; Add the Table Index to the Program Counter
    dt    "Table", 0
```

If you were to compare the instructions produced by the `Table` and `Table2` subroutines, you would find that they are identical, including the number of instructions used by each subroutine. Elsewhere in the book, I describe what directives are and list the entire set available to you for the MPLAB PIC microcontroller assembler that will be used later for experiments and applications.

This simple table is useful for many applications, but only under one condition: the table itself has to be located in the first 256 instructions of the PIC microcontroller's program memory. If the table is not located in the first 256 instructions or straddles the first 256 instruction boundary, then execution will jump to an invalid address because the `PCLATH` register is not correctly set up for the table.

To rectify this, the generic table code could be used:

```
Table3:                ; Return Table Value for Contents of "w"
                        ; Anywhere in PIC microcontroller Memory
    movwf Temp          ; Save the Table Index
    movlw HIGH TableEntries ; Get the Current 256 Instruction Block
    movwf PCLATH        ; Store it so the Next Jump is Correct
    movwf Temp, w       ; Compute the Offset within the 256
    addlw LOW TableEntries ; Instruction Block
    btfsc STATUS, C
    incf PCLATH, f      ; If in next, increment PCLATH
    movwf PCL           ; Write the correct address to the
TableEntries:          ; Program Counter
    dt    "Table", 0
```

In this example, I update the `PCLATH` register according to the starting location of the instructions at `TableEntries`. When I calculate the address of the actual table element to access, I increment `PCLATH` if the destination is outside the initial 256 instruction address block.

The only problem with `Table3` is that it requires a register to store the offset while `PCLATH` is updated. Conditional assembly directives (which are discussed in Chap. 10) could be used to select the correct `bsf` and `bcf` instructions for the 256 address block the table starts in. By using these statements, the table code is one instruction longer than `Table3` but does not require a file register:

```
Table4:                ; Set PCLATH Before Calculating Offset
    if ((TableEntries & 0x100) != 0)
        bsf    PCLATH, 0
    else
```

```

    bcf    PCLATH, 0
endif
if ((TableEntries & 0x200) != 0)
    bsf    PCLATH, 1
else
    bcf    PCLATH, 1
endif
if ((TableEntries & 0x400) != 0)
    bsf    PCLATH, 2
else
    bcf    PCLATH, 2
endif
if ((TableEntries & 0x800) != 0)
    bsf    PCLATH, 3
else
    bcf    PCLATH, 3
endif
if ((TableEntries & 0x1000) != 0)
    bsf    PCLATH, 4
else
    bcf    PCLATH, 4
endif
    addlw  LOW TableEntries    ; Calculate Offset
    btfsc  STATUS, C          ; If carry, goto the next Offset
    incf   PCLATH, f
    movwf  PCL
TableEntries:
    dt     "Table", 0

```

Throughout this chapter and the book, I have given you examples of typical processor code and how the PIC microcontroller's instruction set and features can be used to improve the operation of the code. In this case, I have given you a very general set of code, and I recommend that you always use it for your tables instead of the single instruction program counter update. Tables do not have to be only used for ASCII information, as I have shown in the examples above. They are the basis for computed address and are useful for conditional execution and execution state machines, as I will show later in the book.

Interrupts Creating interrupt service routines (also known as interrupt handlers) for the mid-range PIC microcontroller differs from the approach taken for other processors, as a clear understanding of the application is required to ensure that when execution is returned to the previously running code the hardware is in exactly the same state it was left in. In traditional processors, the basic interrupt acknowledge operation and return from interrupt instructions save the context registers correctly, avoiding the need for the programmer to explicitly save them. The interrupt acknowledge and return from interrupt are very basic operations in the mid-range PIC microcontroller, which fits in with the RISC philosophy used to architect the device. Unfortunately, the process is complicated

TABLE 7.3 CONTEXT REGISTERS AND THE BITS THAT COULD BE AFFECTED DURING AN INTERRUPT SERVICE ROUTINE	
REGISTER	AFFECTED BITS
WREG	All
STATUS	IRP, RP1, RP0, Z, DC and C
PCLATH	All
FSR	Optional—only if the interrupt service routine sets and uses FSR

by the use of bank registers and execution pages as well as instructions that inconveniently change the context registers of the chip. There are some conventions that should be used to ensure that your interrupt handlers return execution with all context registers in the exactly same state as they were when the interrupt was acknowledged.

The context registers in the mid-range PIC microcontroller are listed in Table 7.3 with bits that you should be concerned about. The list shouldn't be surprising, considering the discussion of bank addressing and interpage jumps that have been presented in this chapter. When starting an interrupt service routine, these four registers should be saved. Complicating the process is assuming that the register bank bits specify a bank other than the bank used to store the context register contents and if the PCLATH bits are not set to page 0 (where the interrupt service routine is located).

The recommended entry code for the mid-range PIC microcontroller interrupt service routine (at address 4, as noted in the previous chapter) is:

```
org 4
Int:
    movwf _wreg           ; Save the contents of WREG
    movf  STATUS, w       ; Save the contents of STATUS
    movwf _status
    movf  PCLATH, w       ; Save the contents of PCLATH
    movwf _pclath
    movf  FSR, w          ; Optional – save the contents of FSR
    movwf _fsr
    movlw 0x1F             ; Reset IRP, RP1 & RP0 (Assuming that the
    andwf STATUS, f        ; registers accessed are in Bank 0)
    clrf  PCLATH           ; Reset PCLATH to execute out of first page
; Test and Reset the "f" bit requesting the interrupt
```

Saving the contents of the FSR register is marked as optional because its contents only have to be saved if the interrupt service routine modifies the FSR in any way.

The "Test and Reset . . ." code will be discussed elsewhere in the book, but I recommend that the interrupt source (if there are multiple possible sources) is checked and

the `f` bit is reset as soon as possible in case another interrupt request comes in while the interrupt handler is executing and becomes ignored.

Before implementing this code, there are two rules you should be aware of. The first is not to execute any `goto` or `call` instructions until after all the instructions have been executed—if `PCLATH` has not been reset, you could have a jump to an unexpected address. For example, the code:

```
org 4
Int:
    goto    IntHandler        ; Skip over Mainline Code

; Mainline Code

IntHandler:
    movwf   _w                ; Save the contents of WREG
    movf    STATUS, w         ; Save the contents of STATUS
    movwf   _status
    movf    PCLATH, w         ; Save the contents of PCLATH
    :
```

is a very dangerous practice if there is any chance `PCLATH` is not zero (or set to the correct page for the interrupt handler body).

The second issue is where to put the four context saving registers. If you were to look at the datasheet of all mid-range PIC microcontroller part numbers, you would see that there is always a portion of the file registers that is common to or shadowed across all of the register banks. These file registers can be used to temporarily store data that is being passed between banks or, more importantly for this discussion, used to store the context information. For example, in the PIC16F877A, the file registers at offsets 0x70 to 0x7F in each bank access the same file registers and the four context saving registers, `_wreg`, `_status`, `_pclath`, and `_fsr`, are placed in here. You could place the `_wreg` context register in the shadowed file register with the other context registers located in a specific register bank, but this will require extra instructions to set the bank (and restore it) in the interrupt service routine).

Once the interrupt has been serviced (and the requesting flag bit has been reset), to return from the interrupt use the instructions:

```
IntEnd:
    movf    _fsr, w           ; Restore the basic context registers
    movwf   FSR
    movf    _pclath, w
    movwf   PCLATH
    movf    _status, w
    movwf   STATUS
    swapf   _wreg, f          ; Restore WREG without affecting Z STATUS bit
    swapf   _wreg, w
    retfie
```

The two `swapf` instructions are used to load WREG without possibly changing the Z bit of the STATUS register.

When you look at different people's applications, you may see interrupt handler context save and restore code that is as simple as:

```
org 4
Int:
    movwf    _wreg            ; Save the contents of WREG
    movf     STATUS, w       ; Save the contents of STATUS
    movwf    _status

    :

IntEnd:
    movf     _status, w
    movwf    STATUS
    swapf    _wreg, f         ; Restore WREG without affecting Z STATUS
    swapf    _wreg, w         ; bit
    retfie
```

This code should only be used in PIC microcontroller part numbers where the program memory is one page (2038 instructions) or less and where its structure is based on the model that all bank 1 register accesses take place before interrupt acknowledges are enabled. These are not unreasonable assumptions for many of the smaller devices (like the PIC16F84A), which have less than 2k instructions in program memory and do not require accessing bank 1 registers except for the initial hardware setup.

When you are comfortable with the PIC microcontroller architecture and instructions, you might want to implement interrupt handlers that do not have to save the context register bits at all. For example, if the interrupt handler was used to set a flag when TMR0 has overflowed six times, the code could be:

```
org 4
Int:
    bcf      INTCON, T0IF     ; Reset the Timer 0 Flag
    bsf      Register, Flag   ; Set the Flag
    decfsz   TMR0Count, f     ; Decrement the Counter
    bcf      Register Flag    ; Reset the Flag if Counter not zero
    retfie                   ; Return to Mainline
```

This code assumes that `Register` is a file register in the shadowed address space and not a hardware register, to allow it to be updated twice in the interrupt handler without affecting operation of the application. Remembering that `bcf`, `bsf`, `btfsf`, `btfsfss`, `incfsz Register, f`, `decfsz Register, f`, and `swapf Register, f` all execute without affecting any of the context registers, and their bits will give you the capability of coming up with interrupt service routines that do not require saving the context registers.

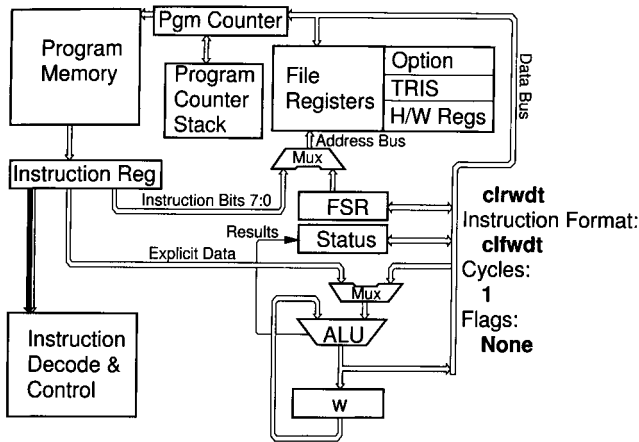


Figure 7.39 The `clrwtd` instruction operation.

PROCESSOR CONTROL INSTRUCTIONS

There are only three instructions that are used to explicitly control the operation of the PIC microcontroller's processor. The first, `clrwdt` (see Fig. 7.39) is used to reset the watchdog counter. The second, `sleep`, is used to hold the PIC in the current state until some condition changes and allows the PIC to continue execution. The last processor control instruction is `nop` (no operation), which simply delays one instruction cycle of time.

`clrwdt` clears the watchdog timer (and the TMR0/WDT prescaler if it is used with the watchdog timer), resetting the interval in which a timeout can occur. The purpose of the watchdog timer is to reset the PIC microcontroller if execution is running improperly (i.e., caused by an external EMI upset or there is a problem with the application code which causes execution to run amok). To ensure a watchdog timer timeout (and reset) is not executed at an inappropriate time, a `clrwdt` instruction is inserted in the code to reset the timer before the watchdog timer timeout if the application is running properly. Ideally, the application code should only have one `clrwdt` instruction written into it and this should only be executed through one path (i.e., every time an input event has processed and the queue for the next input event is about to be checked).

There are two purposes of the `sleep` instruction, which executes as shown in Fig. 7.40. The first is to shut down the PIC microcontroller once it has finished processing the program. This prevents the PIC microcontroller from continuing to run and potentially affecting any other hardware in the application while it is executing. Using the PIC microcontroller in this manner presumes that the PIC microcontroller is only required for a certain aspect of the application (i.e., initialization of the hardware) and will not be required after this aspect.

The second purpose of the `sleep` instruction is to provide a method of allowing the PIC microcontroller to wait for a certain event to happen. A sleeping PIC microcontroller can be flagged of this event in one of three ways. The first is a reset on the `_MCLR` pin

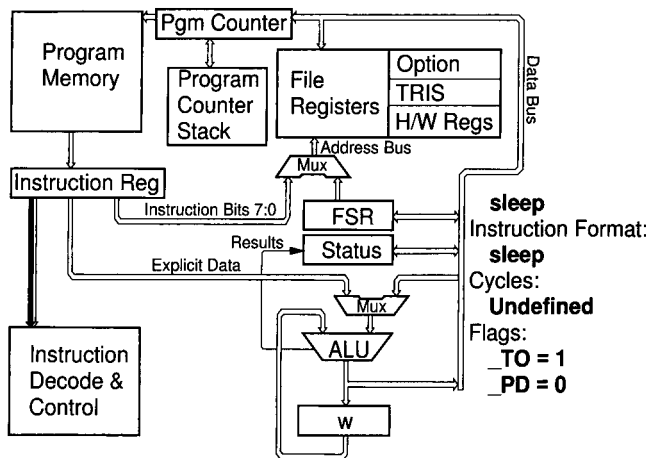
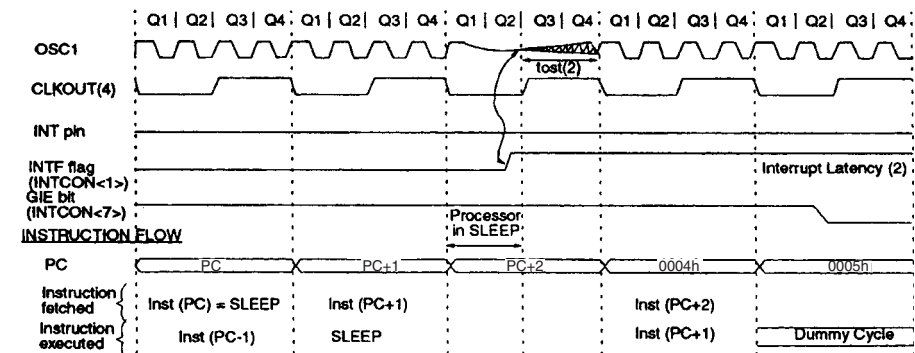


Figure 7.40 The sleep instruction operation.

(which will cause the PIC to begin executing again at address 0), the second is if the watchdog timer wakes up the PIC, and the third method is to cause wakeup by some external event (i.e., interrupt).

Using `sleep` for either of these reasons will allow you to eliminate the need for wait loops and could simplify your software. The Parallax BASIC Stamp and the PICBASIC compiler use `sleep` for its `nap` instruction to simplify the operation of the code and to minimize the current requirements of the PIC microcontroller while it is stopped.

Figure 7.41 shows how a sleeping PIC microcontroller can be awakened by an interrupt. During `sleep`, the built-in oscillator is turned off. When the PIC microcontroller is woken up, it restarts in a similar manner to the initial power-up of the microcontroller. This wake up takes a relatively long time (1024 clock cycles) to wait



Note 1: XT or LP oscillator mode assumed.

2: $t_{\text{lost}} = 1024 t_{\text{osc}}$ (drawing not to scale). This delay will not be there for RC osc mode.

3: When GIE = 0 processor jumps to interrupt routine after wake-up. If GIE = 1, execution will continue in line.

4: CLKOUT is not available in these osc modes, but shown here for timing reference.

Figure 7.41 The instruction cycles executed when a mid-range PIC microcontroller is interrupted when `sleep` is active.

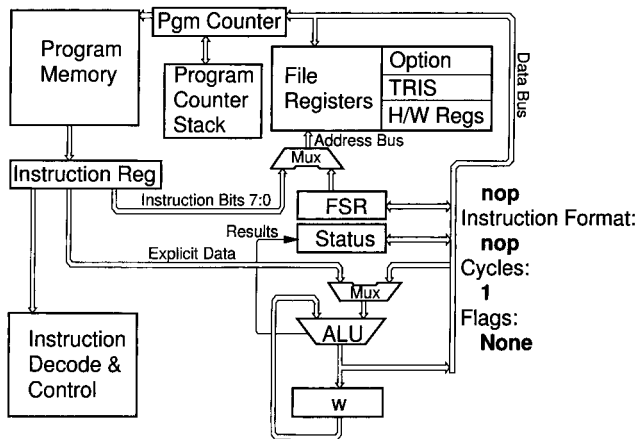


Figure 7.42 The `nop` instruction does not affect any of the internal functions of the PIC microcontroller.

for the built-in oscillator to stabilize before it resets the PIC microcontroller and resumes executing the application code.

`nop` means “no operation” and is usually pronounced “no-op.” When this instruction (see Fig. 7.42) is executed, the processor will just skip through it, with nothing (registers or STATUS register bits) changed. If you study a number of different processors, you will find that they all have a `nop` instruction. `nops` are traditionally used for two purposes. The first is to provide time synchronizing code for an application; if you look at various timed routines in this book, you will see that I use these instructions to provide a one instruction cycle delay.

A simple way to get a two instruction delay is to use a “goto the next instruction” instead of two `nops`. The format for this `goto` instruction is:

```
goto    $ + 1
```

The second traditional use of `nops` is to provide space for patch code. Patching code in processors is usually done by replacing instruction locations that have all their bits set with instructions that were placed in line to see how the operation of the application is affected. When PROM, EPROM, EEPROM, and Flash memory technology is ready to be programmed (i.e., cleared) all the cells are set (equal to 1). The programming burns zeros into memory to make the various instructions. To support this, `nop` instructions consisting of all bits set are often provided within processor instruction sets.

The `nop` cannot be used in the mid-range PIC microcontroller for providing patch memory code space because the `nop` instruction is all zeros. Despite this, there is a method of providing space in the code for patches in the PIC microcontroller. To do this, there must be unprogrammed code left in the application that can be changed by a programmer. To do this, the reverse of making instructions from `nops` is used.

For example, you may put the following code in your mid-range PIC microcontroller application to provide patch code space:

```
goto    $ + 6                ; Skip Over five patch addresses
dw      0x03FFF              ; Instruction Word with all bits set
dw      0x03FFF
dw      0x03FFF
dw      0x03FFF
dw      0x03FFF
```

To enter some patch code, all the 1s in the `goto` statement must be programmed to 0s, changing the instruction into a `nop`. The `dw 0x03FFF` assembler directives (commands) are used because they keep all the bits set at the instruction word where the `dw` directive is located. The code snippet above will allow you to add up to five instructions without having to reassemble your code.

To add patch code, convert the `goto $ + 6` instruction to a `nop` and then write over the `dw` statements with the instructions needed for the patch. For example, adding code to invert the contents of `w` by using `xorlw 0x0FF` could be accomplished by changing the six instructions above to:

```
nop                ; FORMERLY: goto    $ + 6
xorlw 0x0FF        ; FORMERLY: dw      0x03FFF
goto    $ + 4      ; FORMERLY: dw      0x03FFF
dw      0x03FFF
dw      0x03FFF
dw      0x03FFF
```

Instructions in PIC microcontroller program memory can be programmed out (erased or turned into `nop` instructions) very easily, because the `nop` instruction consists of all the bits set to zero.

I should point out that the need for patching memory is just about nonexistent due to the wide availability of Flash-based PIC microcontroller part numbers, which can be completely reprogrammed in the same amount of time as it would take to change the instructions as outlined above. Where you may want to provide patch memory space is in EPROM-based parts, which do not have Flash memory equipped comparable devices.

Low-End PIC Microcontroller Instruction Set

The low-end PIC microcontroller's architecture and instruction set can be considered a subset of the mid-range chip. All the low-end instructions are available in the mid-range (with the difference being that the low-end does not have `addlw`, `sublw`, `retfie`, and `return`) and many of the software methods that were discussed earlier in the chapter can be used on the low-end. The memory spaces (both register and instruction) are smaller

than the mid-range devices with fewer bits available for accessing registers. Rather than go through each instruction as I did above, in the following sections I just want to discuss the differences and issues that you will have to deal with working with low-end PIC microcontrollers. A complete table of instructions can be found in App. B.

Just so there is no confusion, the low-end PIC microcontroller instruction set is 12 bits wide. The mid-range architecture has 14-bit instructions and the PIC18 instructions are 16 bits wide. I'm making this distinction because there is no part number differentiator between the low-end and the mid-range PIC microcontroller architectures (for example, a PIC16C54 is a low-end device and a PIC16C554 is a mid-range device). I find the best way of determining what processor architecture a device is built with is the width of instructions in bits.

To experiment with the low-end architecture, you can use the LEInsTemplate.asm file:

```
LIST R=DEC
INCLUDE "p16f505.inc"

CBLOCK 8
ENDC

__CONFIG _MCLRE_OFF & _CP_OFF & _WDT_OFF & _IntRC_OSC_RB4EN

org      0

; Insert Code Here

goto    $
end
```

A project should be set up for this template in the same way you set up the template for InsTemplate.asm. When you use the Project Wizard, remember to select the device to be used as the PIC16F505, which is a 14-pin, low-end architecture device with 1024 instructions and 72 file registers.

DATA MOVEMENT INSTRUCTIONS

The low-end data movement instructions are identical to the mid-range PIC microcontrollers except in their ability to directly access data in banks other than the bank 0. The low-end devices do have a set of banks, which are shown in Fig. 7.43, but, unlike the mid-range chips, the low-end devices do not have the RP# bits that allow the changing of banks. Instead the upper banks of the low-end PIC microcontrollers can only be accessed using the index register (FSR with the access register INDF).

This requirement restricts the number of file registers that can be used in an application. When I work with the low-end architecture, I remember that memory is available using the formula:

$$\text{Memory} = (\text{PORTC\_present} \neq \text{true}) + 8 \text{ GP Registers} + (\# \text{ of Banks} \times 16)$$

Bank 0	Bank 1	Bank 2	Bank 3	
FSR Bits 6:5 = 0:0	FSR Bits 6:5 = 0:1	FSR Bits 6:5 = 1:0	FSR Bits 6:5 = 1:1	
Addr - Reg	Addr - Reg	Addr - Reg	Addr - Reg	
00 - INDF 01 - TMR0 02 - PCL 03 - STATUS 04 - FSR 05 - PORTA* 06 - PORTB 07 - PORTC 08-0F Shared File Regs	20 - INDF 21 - TMR0 22 - PCL 23 - STATUS 24 - FSR 25 - PORTA* 26 - PORTB 27 - PORTC 28-2F Shared File Regs	40 - INDF 41 - TMR0 42 - PCL 43 - STATUS 44 - FSR 45 - PORTA* 46 - PORTB 47 - PORTC 28-2F Shared File Regs	60 - INDF 61 - TMR0 62 - PCL 63 - STATUS 64 - FSR 65 - PORTA* 66 - PORTB 67 - PORTC 68-8F Shared File Regs	Shared Registers
10-1F Bank 0 File Regs	30-3F Bank 1 File Regs	50-4F Bank 2 File Regs	70-7F Bank 3 File Regs	Bank Unique Registers

Figure 7.43 Access to register banks 1 through 3 of the low-end PIC microcontroller is accomplished using the FSR register.

So, if an 18-pin device (no PORTC) has two banks, then the amount of memory available is:

Memory = (PORTC_present != true) + 8 + (2 Banks x 16)
 = 1 + 8 + 32
 = 41

For bank 0 direct file register addressing, you have 8 GP registers and the 16 bank specific registers along with an additional file register depending on whether PORTC is present for a total of 24 or 25. This amount will seem very small, but for most very low-end applications it is quite adequate.

OPTION and TRIS Instructions The low-end PIC microcontrollers do not have directly accessible OPTION_REG or TRIS# registers and instead, you must use the `option` and `tris` instructions, respectively, to write values to them. The most important issue that you should be aware of is that you cannot read back the contents of these registers in your application code. This is rarely an issue for OPTION_REG, but it can be a problem with the TRIS registers in case you are changing the operation (direction) of I/O pins later.

The best solution to this issue is to keep file register copies of the various TRIS registers and after they are updated with the new state, their value is written to the appropriate TRIS register. For example, to implement changing 1 bit to output from input, the mid-range PIC microcontroller code would be:

```
bsf    STATUS, RP0      ; Execute in Bank 1
bsf    TRISB ^ 0x80, 3  ; Convert bit 4 to input
bcf    STATUS, RP0      ; Return to Bank 0
```

In the low-end architecture, it would be:

```
bsf    _trisb, 3      ; Convert bit 4 to input
movf   _trisb, w      ; Update the I/O pin
tris   PORTB
```

Both instruction sequences take the same amount of instructions and cycles.

Indexed addressing Indexed addressing in the low-end PIC microcontroller architecture works in exactly the same way as in the mid-range architecture, but there are two things that you should always be cognizant of. The first is to remember that the maximum size of any array variable is 16 bytes, and the second is that the FSR can never equal zero. Both of these characteristics are something that you will have to keep in the back of your mind when working with the architecture.

The 16-byte maximum array size is important to remember because if you go beyond this limit, you are going to overwrite the hardware registers in the next bank. This is another programming problem that may turn up and get you some time after you have written and released the application. It will be very difficult to find and resolve.

To avoid this issue, I suggest that you always monitor bit 4 of the FSR because when this bit changes state, you know you have gone over the 16 address limit for the array. For example, if you have a 16-byte LIFO buffer (stack), you may choose to ignore any data pushes after the 16 bytes are filled up:

```
DataPush:                ; Push the byte in "StackData" onto stack
    incf   StackTop, w    ; Pointer to the next position in the stack
    movwf  FSR
    btfss  FSR, 4         ; Bank 1 used for stack
    retlw  -1             ; Return Indicating unable to add to stack
    movwf  StackTop       ; Save new stack top
    movf   StackData, w   ; Get data to push
    movwf  INDF
    retlw  0              ; Return with no error
```

Similarly, a data pop could be accomplished by:

```
DataPop:                 ; Pop byte from stack and put in "StackData"
    movf   StackTop       ; Get Stack Top Register
    movwf  FSR
    btfss  FSR, 4         ; If Bit 4 Clear, then Stack is Empty
    retlw  -1
    movf   INDF, w        ; Get the Stack Data
    movwf  StackData
    decf   StackTop, f     ; Point to previous value in the stack
    retlw  0              ; Return with no error
```

Recognizing that bit 4 changes when the 16-byte array is full is a trick that can be used in both the low-end and the other PIC microcontroller architectures to indicate the

I realize that these operations add quite a few instructions (and require a file register), but they will simulate the `addlw` and `sublw` instructions and can be placed in a macro for your use.

EXECUTION CHANGE INSTRUCTIONS

Like the other aspects of the low-end PIC microcontroller architecture compared to the mid-range architecture, the smaller address size affects the way that execution change operates. The changes that you will have to keep in mind when you are creating low-end applications are the smaller page size, the difference in operation between the `goto` and `call` instructions, and the lack of a return instruction. None of these differences will prevent you from being able to port your applications between the two architectures, they are just a few things to watch for when you do it.

For the `call` and `goto` instructions as well as those in which the contents of PCL are modified, the PA0 to PA2 bits of the STATUS register are used as the high order address bits of the actual jump address within a specified bank. These bits, while similar in operation to the PCLATH bits, require a slightly different approach because they are part of the STATUS register.

In the mid-range devices, the two instructions:

```
movlw  HIGH NewAddress
movwf  PCLATH
```

are all that are required to set up the high order address bits for the `goto` or `call` instructions because the bits are organized to allow a direct transfer of the address to the PCLATH bits. This is not possible in the PA0 to PA2 bits in the STATUS register; instead, I recommend that you use the conditional assembly statements I introduced earlier in the chapter for updating the PA0 to PA1 bits. To do this, remember that PA0 is bit 9 of the address while PA2 is bit 11:

```
if ((NewAddress & 0x200) != 0) ; Required if Program Memory >= 1024
    bsf    STATUS, PA0
else
    bcf    STATUS, PA0
endif
if ((NewAddress & 0x400) != 0) ; Required if Program Memory >= 2048
    bsf    STATUS, PA1
else
    bcf    STATUS, PA1
endif
if ((NewAddress & 0x800) != 0) ; Required if Program Memory >= 4096
    bsf    STATUS, PA2
else
    bcf    STATUS, PA2
endif
```

The interpage `goto` setup code above can be demonstrated on the MPLAB IDE LEInsTemplate.asm project using the program:

```
if ((NewAddress & 0x200) != 0) ; Only 1024 instructions available
    bsf    STATUS, PA0
else
    bcf    STATUS, PA0
endif
goto      (NewAddress & 0x1FF) | ($ & 0x200)

ReturnAddress:
    goto   $

org        0x234
NewAddress:
if ((ReturnAddress & 0x200) != 0)
    bsf    STATUS, PA0
else
    bcf    STATUS, PA0
endif
goto      (ReturnAddress & 0x1FF) | ($ & 0x200)
```

In this program, the modifications to the address in the `goto` statements are the same ones as would be done in a mid-range device.

The number of bits available for addressing a page in the low-end PIC microcontroller's `goto` instruction is 9, which makes the size of the page 512 instructions. There is a problem with the `call` instruction because it has only 8 bits available for the offset of the start of the subroutine within the page and the 9th bit is always assumed to be reset. This means that a `call` instruction (as well as a write to PCL, which will be discussed in the next section) will only be able to access the first 256 instructions of a low-end device page. If you attempt to call an address in the second 256 instructions of the page, the MPASM assembler will return an error.

The solution to the problem should be obvious: never put a subroutine in the second 256 addresses of a low-end page. Unfortunately, this is not always possible, so the solution that normally used is to put a label with a `goto` in the first 256 instructions of the page to the actual subroutine code. This allows the body of the subroutine to be in the second half of the page while not having any errors flagged.

When calling a subroutine in another page, always remember to restore the PA# bits upon return. To do this, check the page of the current address (using the `$` directive) as shown below:

```
if ((CallAddress & 0x200) != 0) ; Required if Program Memory >= 1024
    bsf    STATUS, PA0
else
    bcf    STATUS, PA0
endif
```

```

if ((CallAddress & 0x400) != 0) ; Required if Program Memory >= 2048
    bsf    STATUS, PA1
else
    bcf    STATUS, PA1
endif
if ((CallAddress & 0x800) != 0) ; Required if Program Memory >= 4096
    bsf    STATUS, PA2
else
    bcf    STATUS, PA2
endif
call    (CallAddress & 0x1FF) | ($ & 0x200)
if (($ & 0x200) != 0)          ; Required if Program Memory >= 1024
    bsf    STATUS, PA0
else
    bcf    STATUS, PA0
endif
if (($ & 0x400) != 0)          ; Required if Program Memory >= 2048
    bsf    STATUS, PA1
else
    bcf    STATUS, PA1
endif
if (($ & 0x800) != 0)          ; Required if Program Memory >= 4096
    bsf    STATUS, PA2
else
    bcf    STATUS, PA2
endif

```

The lack of a `retfie` instruction should not be surprising because there are no interrupts in the low-end PIC microcontrollers. Where you should be surprised is the lack of the `return` instruction. This is further confused by different versions of the Microchip assembler accepting the `return` instruction for the low-end devices and substituting a `retlw 0` instruction for it. This has caused problems for a number of people and is the reason why I tend to not return subroutine parameters in WREG because I may slip up in my low-end programming or port code from a mid-range application into a low-end PIC microcontroller and find that it doesn't work properly. The latest versions of MPLAB and the MPASM assembler will return a warning if a `return` instruction has been inserted into low-end PIC microcontroller application code and will tell you that a `retlw 0` instruction has been inserted in its place.

Tables Tables suffer from the lack of an address bit 8, the same way the low-end PIC microcontroller `call` instruction does. Rather than come up with a method to create an arbitrary sized table that can be located anywhere in the low-end device's memory, I am going to recommend that the tables be placed at the start of a page and that you are restricted to 255 maximum table entries (assuming that the `addwf PCL, f` instruction takes up the extra available instruction).

The low-end PIC microcontroller table would take the form:

Table:

```
addwf PCL, w          ; WREG has offset into table
dt      "Table", 0
```

where WREG was loaded before the `call` instruction with the offset within the table.

PROCESSOR CONTROL INSTRUCTIONS

There are no differences in the processor control instructions available to the low-end PIC microcontroller part numbers and their noninterrupt related behavior available to the mid-range.

PIC18 Instruction Set

Like the low-end PIC microcontroller architecture, much of the software written for the mid-range chips can be used on the PIC18 architecture. Differences include the change in bank register and program memory accessing. All the hardware I/O or special function registers (SFRs) in the PIC18 can be accessed directly, without the need for changing bank select bits. The multiple, enhanced index registers ease the effort in implementing arrays, data stacks, pointers, and other multibyte data structures. The PIC18 has additional instructions over the mid-range with many new capabilities that will make your application development simpler and much easier. Despite the differences in the architectures, you will find that most applications written for the mid-range PIC microcontroller processor architecture will execute without modification in the PIC18.

There has been a big improvement in the efficiency of the instruction set used in the PIC18 architecture. As a superset of the mid-range architecture, there are operations that can be carried out much more efficiently and much easier in the PIC18 than in the mid-range devices. While I have noted that the standard PIC microcontroller mid-range is at least 30 percent more efficient than other 8-bit processor architectures in executing operations at the same instruction cycle speed, the improved instruction set of the PIC18 easily doubles that value (or more). This makes the PIC18 more than twice as efficient in terms of application instruction size and execution cycles over other 8-bit processors available in the marketplace. Coupled with the very fast (four clock cycle per) instruction clock, the PIC18 is capable of remarkably fast and powerful applications without having excessive heat dissipation or current requirements.

For the remainder of this chapter, I will introduce you to the PIC18 and show what can be done to improve the efficiency of application execution over the earlier PIC microcontroller processor architectures. To allow you to learn as much as you can, I suggest that you start up MPLAB IDE, click on Project and then Project Wizard to create the `P18InsTemplt.asm` project for the PIC18F2510 with the execution template below. This will allow you to quickly try out the different instructions and code snippets that are presented by entering them into the `P18InsTemplt.asm` file:

```

LIST R=DEC
INCLUDE "p18f2510.inc"

CBLOCK 0x0
ENDC

CONFIG OSC=INTIO67, FCMEN=OFF, IESO=OFF, PWRT=ON, BOREN=OFF, WDT=OFF
CONFIG WDTPS=1, MCLRE=OFF, LPT1OSC=OFF, PBAEN=OFF, CCP2MX=PORTC
CONFIG STVREN=OFF, LVP=OFF, XINST=OFF, DEBUG=OFF
CONFIG CP0=OFF, CP1=OFF, CP2=OFF, CP3=OFF, CPB=OFF
CONFIG WRT0=OFF, WRT1=OFF, WRT2=OFF, WRT3=OFF, WRTB=OFF, WRTC=OFF
CONFIG EBTR0=OFF, EBTR1=OFF, EBTR2=OFF, EBTR3=OFF, EBTRB=OFF

org 0

; Insert Code Here

goto $
end

```

DATA MOVEMENT INSTRUCTIONS

Before starting to work on a new PIC microcontroller architecture, I have found the first thing is to decide where variables are to be placed and how data is to be organized. In the mid-range PIC microcontroller devices, placing single and double byte variables in bank 0 cuts down on the number of execution bank changes that have to be made within the application. Array variable data is normally placed in bank 1 (or bank 2 or bank 3 is selected using the IRP bit), which can be addressed directly by the FSR register. Data for the low-end PIC microcontroller architecture is located similarly with bank 0 being used for single and double byte values and arrays placed in bank 1 or higher. In the PIC18 architecture, I follow a similar philosophy for single and double byte variable placement. These variables are always located in the first 128 addresses of bank 0, allowing them to be addressed via the access RAM. PIC18 array variables are placed in the upper banks of memory where they can be accessed by the index (FSR) registers, which do not require special bank access. In each architecture I have chosen a method that allows me to access basic 1- and 2-byte variables directly without changing the bank register and accessing arrays using the basic capabilities of the FSR register.

Data can be loaded and stored in the WREG register using the PIC18 `movf`, `movlw`, and `movwf` instructions. The important difference between these instructions and the other architecture's analogs is the addition of the access bit in the instruction bit patterns, which allows you to choose between using the bank register for accessing data or reading and writing the basic information directly. When a 1 is specified as the last parameter in the instruction, like:

```
movf i, w, 1
```

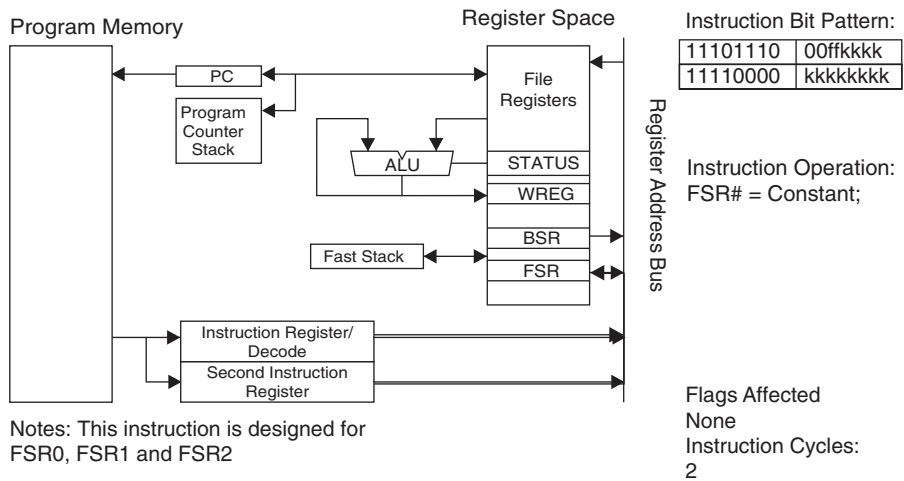


Figure 7.44 The PIC18 `lfsr` instruction loads an FSR register with a constant value.

the BSR register is used to select the bank the variable “i” is located within. If a 0 or nothing is specified as the instruction’s last parameter:

```
movf i, w
```

the value for i is taken out of the access bank or PIC18 addresses 0x000 to 0x7F for file registers or 0xF80 to 0xFFFF for the special function registers (SFRs). If the access bit isn’t present, then the access bank is selected by default (as if the access bit was zero).

The new instructions added to the PIC18Cxx architecture are `lfsr` (Fig. 7.44), `movlb` (Fig. 7.45), and `movff` (Fig. 7.46). These instructions are used to specify

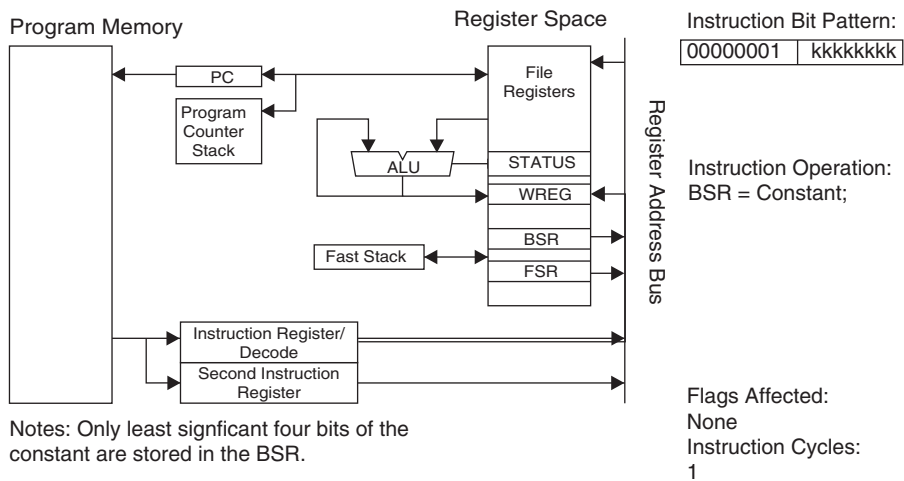


Figure 7.45 The PIC18 `movlb` instruction loads the BSR register with a literal value.

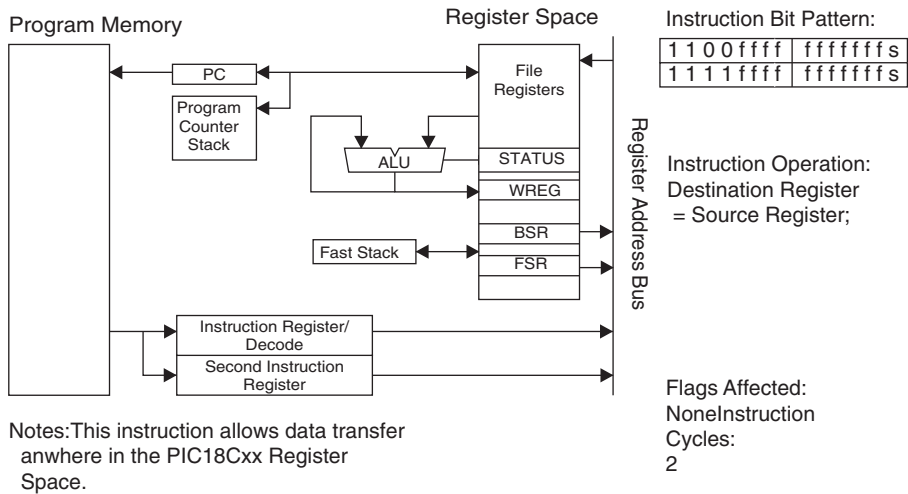


Figure 7.46 The PIC18 `movff` instruction allows direct transfer of data between registers without temporarily saving the value in WREG.

addresses anywhere in the PIC18's register space. The first two instructions are used to load constant full register addresses into the FSR index register and the bank select register (BSR), respectively. `lfsr` loads a 12-bit constant into the specified FSR register as the start of a table. `movlb` loads a 4-bit constant into the BSR register that specifies which bank variables are to be taken from.

The `movff` instruction is particularly useful because it does not change the STATUS register bits and does not affect the contents of WREG. I found that this instruction allows very simple string movement and data copies. For example, to copy 5 bytes of data from one string to another, the code is simply:

```
lfsr    FSR0, SourceString ; Point to the Start of the Strings
lfsr    FSR1, DestString
movlw   5                  ; Load WREG with 5
Loop
movff   POSTINC0, POSTINC1 ; Copy String and Increment the FSR
decfsz  WREG, f, 0         ; Registers
bra     Loop
```

In the mid-range architecture, the same function would be accomplished by the following code:

```
clrf    Count              ; Reset the Offset Within the String
Loop
movlw   SourceString       ; Get the Current Source Element
addwf   Count, w
movwf   FSR
movf    INDF, w
movwf   Temp
movlw   DestString         ; Save the Source in the Destination
```


To carry out a program memory read, the following instruction sequence could be used:

```
movlw    UPPER ReadAddr    ; Load the Top 5 Address Bits
movwf    TBLPTRU, 0
movlw    HIGH ReadAddr     ; Load the "Middle" 8 Address Bits
movwf    TBLPTRH, 0
movlw    LOW ReadAddr      ; Load the Bottom 8 Address Bits
movwf    TBLPTRL, 0

tblrd    *                  ; Read the Program Memory

movf     TABLATL, w, 0      ; Process the Low Byte of the Program
:                                     ; Memory Instruction

movf     TABLATH, w, 0      ; Process the High Byte of the Program
:                                     ; Memory Instruction
```

In `tblrd` (and `tblwt`, which follows), a `TBLPTR` increment or decrement specification can be optionally put in. In Figure 7.47, I have included the four options and how the bit pattern is changed.

Table write (`tblwt`, Fig. 7.48) instructions are available to write to the Flash program memory of PIC18 microcontroller architectures. Later in the book, I will show how this instruction is used to write to a PIC microcontroller’s internal Flash registers or to external bus devices. The PIC18Cxx `tblwt` instruction must be terminated by a reset or interrupt.

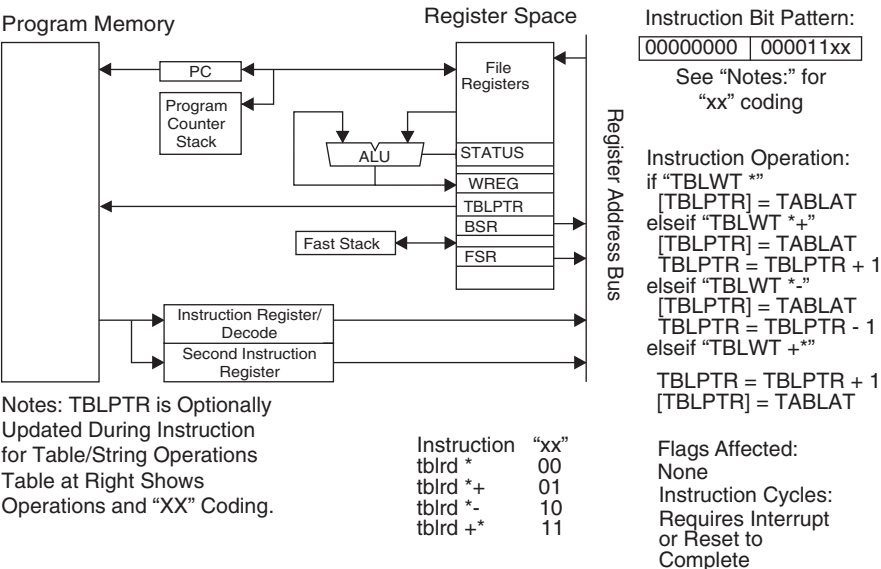


Figure 7.48 The `tblwt` instruction allows you to change program memory.

Indexed addressing The low-end PIC microcontroller architecture's FSR register or pointer can access every address in the register space, but there is a maximum of 128 addresses built into the architecture, broken into blocks of 16 bytes apiece. The mid-range chips have an address space of up to 512 addresses but require an extra addressing bit to access the up to 96 bytes in each block. These restrictions make it difficult to create an application that can access a large amount of data easily in a single, continuous block of memory. The PIC18 architecture does not have these restrictions with its single, large address space and SFRs at the high end range, resulting in a large number of file registers that can be accessed directly by the multiple index pointers. Not only can each index pointer access any address in the 4,096 register address space, but there are increment and decrement options as well as an offset option that will allow you to work with stacks or arrays much more efficiently than if you were working in one of the other architectures. The PIC18 provides much more advanced capabilities than the low-end PIC microcontroller architecture while still retaining its philosophy: being able to access every address in the register space.

Each of the three index pointers available in the PIC18 microcontroller architecture consists of two registers. The low register (marked with an L at the end of its name) provides access to the lower 8 bits of the address while the high register (marked with an H) provides access to the upper 4 bits. Together, the registers point to an address in the register space and can be initialized either by traditional `movlw`, `movf`, or `movwf` instructions or the `lfsr` instruction can be used to set the 12 address bits in a single instruction. Despite being larger than the mid-range index register and IRP bit, you will probably find that the PIC18 FSR registers can be initialized in fewer instructions.

As with the low-end and mid-range architectures, if you are going to access the byte addressed by the index pointer, you could read from or write to the INDF register, which would return the contents of the register pointed to by the FSR, but you also have four additional INDF registers that you can take advantage of to simplify the task of using the index pointer. The four additional PIC18 INDF registers for each index pointer include `POSTINC`, which increments the pointer after the access, `POSTDEC`, which increments the pointer after the access, `PREINC`, which increments the pointer before the access takes place, and `PLUSW`, which adds the contents of `WREG` to the index pointer for the address. Each of these indexed addressing registers provides you with additional capabilities that will come in very handy when you are implementing different functions using the index pointers.

To show what I mean, consider the case of a simple, single byte array. If you were going to use the index register to point to an array, you will probably find that once you point to the start of the array you will never have to modify the contents of the index register because of the ability to access a byte offset specified within the `WREG` using the `PLUSW` register.

For example, consider the case of the array variable `ArrayVar`, which has ten elements, and the FSR register, which was pointing to the first element in the array. If you wanted to load the variable `i` with the contents of third element in the array variable in the mid-range PIC microcontroller, you would use the code:


```
movff Temp, PLUSW0      ; Store value in "Temp"
movlw 1                  ; Get offset to "k"
movf PLUSW0
addwf Temp, f, 0         ; Add "k" to "j" in "Temp"
movlw 3                  ; Get offset to "i"
movff PLUSW0, Temp       ; Store result
```

While this code may look very complex, it is actually simple and, once you are comfortable with it, very easy to implement. This capability is also critical for efficient implementation of compilers that implement local variables as shown here.

DATA PROCESSING INSTRUCTIONS

The PIC18 has some added flexibility and conventional capabilities compared to the other PIC microcontroller processors. As you look through the PIC18 instruction set, you will see that the additions and modifications to its instruction set make it more similar to that of other processors while retaining the PIC18’s ability to create very efficient code.

The most significant addition to the PIC18’s data processing instructions is the `subfwb` (Fig. 7.49) instruction. This instruction carries out a subtract operation with borrow in the order most people are familiar with if they have worked with other processors. Instead of the typical PIC microcontroller subtraction instruction:

Result = (Source Value) – WREG [- !C]

the `subfwb` instruction executes as:

Result = WREG – (Source Value) – !C

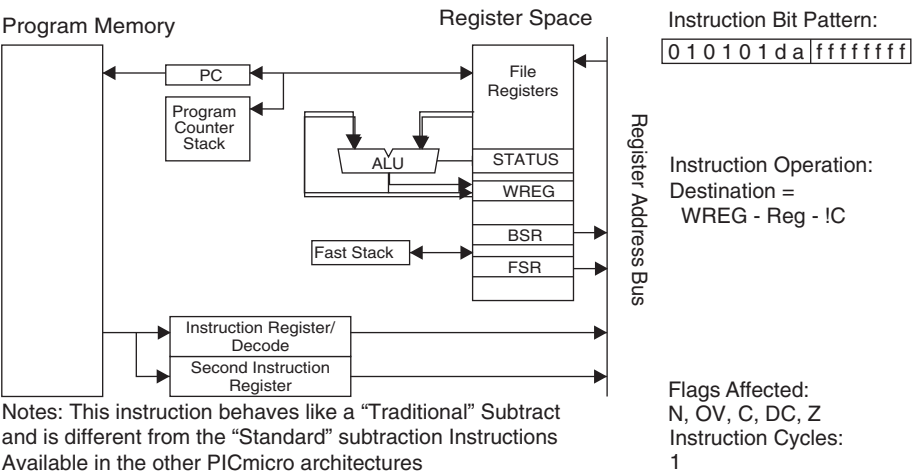


Figure 7.49 The `subfwb` instruction provides the expected subtract operation instead of the addition of the negated value of WREG used by the other subtract instructions.

This instruction frees you from the need of thinking backwards when subtraction instructions are used in an application. To use the `subfwb` instruction, WREG is loaded with the value to be subtracted from (the subtrahend) and the value to take away (the subtrahend) is specified in the instruction. This means that if you have the statement:

$$A = B - C$$

the values of the expression can be loaded in the same left to right order as the PIC microcontroller instructions and use the sequence:

```
bcf      STATUS, C, 0
movf     B, w, 0
subfwb   C, w, 0
movwf    A, 0
```

This is the same order as would be used in most other processors. Note that I reset the carry flag before the instruction sequence to avoid any possibilities of the carry being reset unexpectedly and taking away an extra 1, which will be very hard to find in application code.

A PIC18 16-bit subtraction operation could be:

```
bcf      STATUS, C
movf     B, w, 0
subfwb   C, w, 0
movwf    A, 0
movf     B + 1, w, 0
subfwb   C + 1, w, 0
movwf    A + 1, 0
```

Or if you want to save on the instruction used to clear the carry flag at the start of the sequence:

```
movf     C, w, 0
subwf    B, w, 0
movwf    A, 0
movf     B + 1, w, 0
subfwb   C + 1, w, 0
movwf    A + 1, 0
```

Another difference between the PIC18 and the other PIC microcontroller processors is the inclusion of the `negf` (Fig. 7.50) instruction, which can negate any register in the PIC18's register space.

The single instruction cycle multiply instructions multiply the contents of WREG against the contents of another register (`mulfw`) or a constant (`mul1w`) and store the 16-bit product in the PRODH:PRODL register combination (Fig. 7.51). These instructions are very well behaved and will work for 2's complement numbers and can provide you with some basic digital signal processing (DSP) capabilities in the PIC18.

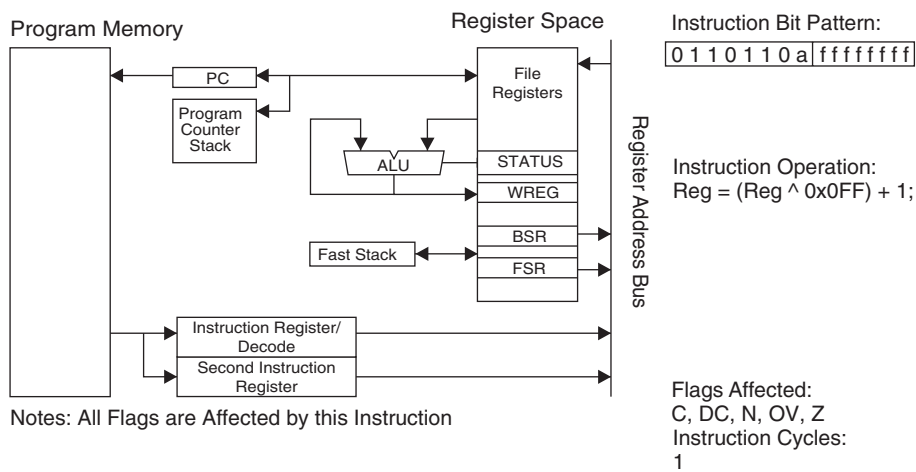


Figure 7.50 The `negf` instruction will two's complement ny register in the PIC18 register space.

EXECUTION CHANGE INSTRUCTIONS

The PIC18's execution change instructions, upon first glance, should be very familiar to you if you are familiar with the other PIC microcontroller families. The PIC18 has the `btfs`, `btfs`, `goto`, and `call` of the low-end and mid-range PIC microcontrollers along with the compare and skip on equals (`cpfseq`), greater than (`cpfsgt`), and less than (`cpfslt`). The PIC18 also has the enhanced increment and skip on result not equal to zero (`infsnz` and `dcfsnz`). Along with these similarities, the PIC18 has four new features that you should be aware of (and remember their availability) when you are developing applications for it.

The first feature that you should be aware of is the `goto` and `call` instructions, which can directly any address in the program memory space. As shown in Fig. 7.52, these instructions are built from two 16-bit words and contain the entire 20 word address bits to allow you to jump anywhere in program memory.

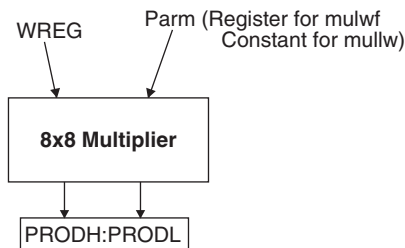


Figure 7.51 The two single instruction cycle 8 by 8 multiplication instructions store their result in PRODH:PRODL.

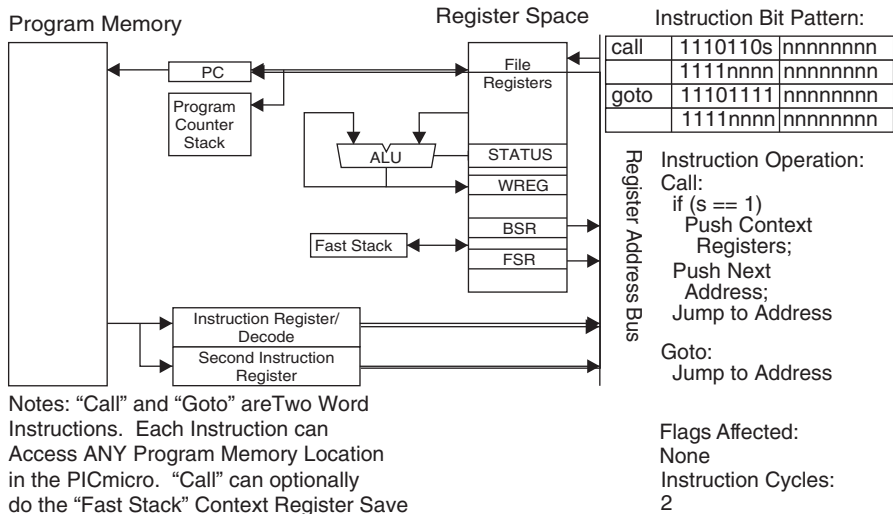


Figure 7.52 The PIC18 `call` and `goto` instructions provide the capability of accessing any address in program memory without the need of updating the PCLATH or PCLATU registers.

The `call` instruction (as well as the corresponding `return` instruction) has the capability, when a 1 is specified as the `s` bit at the end of the instruction, of saving the context registers WREG, BSR, and STATUS in shadow registers of the fast stack, which are retrieved by the `return` instruction by specifying a 1 as well. The issue that you should be aware of for the context save is that you can only save one set of values on the fast stack and the context values of the mainline are always saved when an interrupt is acknowledged. This limits the usability of the context register save to applications that only have a single deep call and no interrupts.

For your first PIC18 applications, I would recommend that you use the instruction set's single word instructions only. The only time you should be using the `goto` or `call` instructions is if you have to access a memory location outside the range of the relative branches. This range is -512 to +511 instruction addresses for the `bra` (branch always) and `rcall` (relative call) instructions and -64 to +63 instruction addresses for the conditional branch instructions that I will discuss below. The `rcall` instruction information is shown in Fig. 7.53.

Along with using the single word execution change instructions, I also recommend that you be careful when using the `$` directive and branching relative to it. When the assembler is calculating addresses, it works on a byte basis, not a word basis as it does for other PIC microcontrollers. This means that you must multiply the number of instructions by 2 to get the correct address. Consider the simple delay loop:

```
movlw 47                ; Loop 47x3 instruction cycles
decfsz WREG, f, 0
bra $ - 1
```

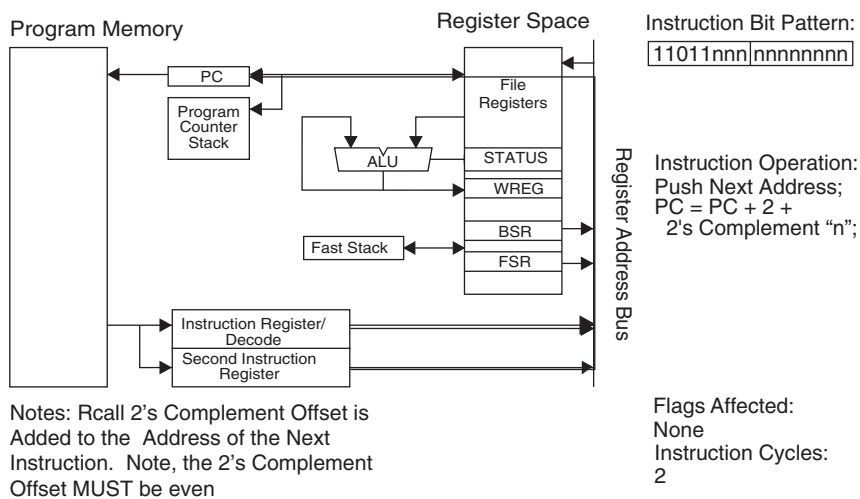


Figure 7.53 The `rcall` instruction allows accessing subroutines that start -64 to +63 instructions from the current program counter location.

If you were to enter the code into the PIC18InsTemplt.asm project and build it, you would get a warning indicating that the instruction cannot start at an odd address. To fix the problem, you have to multiply the offset by 2, producing the code:

```
movlw 47 ; Loop 47x3 instruction cycles
decfsz WREG, f, 0
bra $ - (1 * 2)
```

which will build cleanly and you can simulate to see that it actually takes 141 (47 times 3) instructions. If you want to avoid this difference between the PIC18 and the other devices, I would recommend that you always use labels and never use relative addressing.

Above, I indicated that there was a one word `goto` instruction called `bra` (branch always). This instruction type (shown in Fig. 7.54) changes the program counter according to the 2's complement offset provided in the instruction according to the formula:

$$PC_{new} = PC_{current} + 2 + Offset$$

where `PCcurrent` is the current address of the executing branch instruction. The 2 added to `PCcurrent` results in the address after the current one. `Offset` is the 2's complement value, which is added or subtracted (if the `Offset` is negative) from the sum of `PCcurrent` and 2.

The MPASM assembler computes the correct offset for you when the destination of a branch instruction is a label. MPASM computes the 2's complement offset using the formula:

$$Offset = Destination - (Current Address)$$

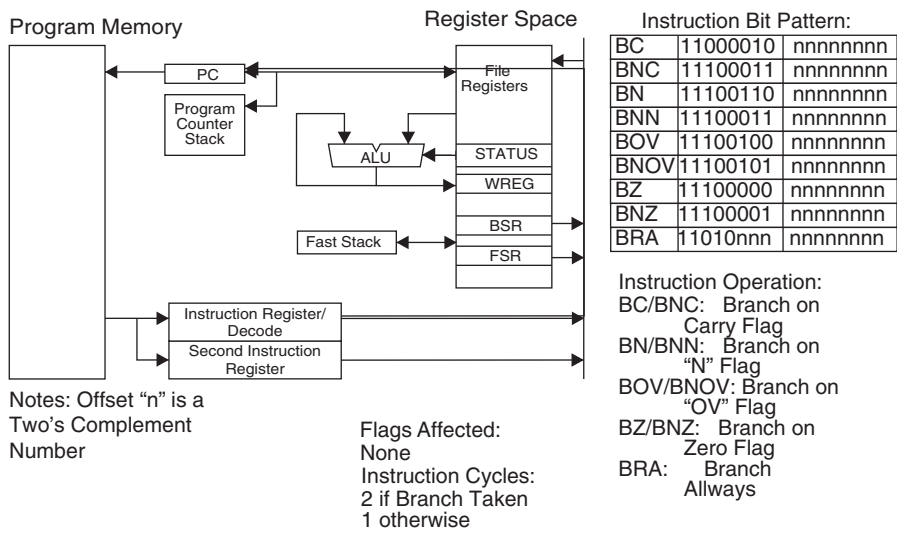


Figure 7.54 The branch instruction can access addresses –512 to +511 instructions from the current program counter location.

If the destination is outside the range of the instruction it is flagged as an error by the MPASM assembler.

Along with the nonconditional branch, there are 8 conditional branch instructions available in the PIC18 and they are shown in Fig. 7.54. They are branch on zero flag set (bz), branch on zero flag reset (bnz), branch on carry flag set (bc), branch on carry flag reset (bnc), branch on negative flag set (bn), branch on negative flag reset (bnn), branch on overflow flag set (bov), and branch on overflow flag reset (bnov). These instructions are equivalent to the branch on condition instructions found in other processors.

These instructions behave similarly to the bra instruction except that they have 8 bits for the offset address (to the bra instruction's 11). This gives the instructions the ability to change the program counter by –64 to +63 instructions.

The last new feature of the PIC18 architecture that is different from the other architectures is the fast stack, in which WREG, STATUS, and BSR registers are saved non-conditionally upon the interrupt acknowledge and vector jump and conditionally during a subroutine call instruction. These registers can be optionally restored after a return or retfie instruction.

Tables PIC18 tables are executed as:

```
TableRead:
    movwf    TableOff, 0
    bcf      STATUS, C, 0          ; First Calculate if past first 256
    rlcf     TableOff, w, 0        ; addresses and by how much
```

```
    addlw    Table & 0xFF
    movf     STATUS, w, 0
    andlw    1
    btfsc    TableOff, 7, 0
    addlw    1
    addlw    (Table >> 8) & 0xFF ; Add Offset to start of table to
    movwf    PCLATH, 0           ; PCLATH
    movf     STATUS, w, 0       ; If in Next Page, increment PCLATU
    andlw    1
    addlw    UPPER Table
    movwf    PCLATU, 0
    rlcfc    TableOff, w, 0      ; Calculate offset within 256 address
    addlw    LOW Table
    movwf    PCL, 0
Table:
    dt      ...
```

If the purpose of the computed `goto` is to return a byte value (using `retlw`), then I would suggest taking advantage of the 16-bit instruction word, store 2 bytes in an instruction word, and use the table read instructions to read back two values. This is somewhat more efficient in terms of coding and requires approximately the same number of instructions and instruction cycles.

A computed byte table read (which allows compressed data) consists of the following subroutine.

```
TableRead:
    movwf    TableOff
    movlw    LOW Table           ; Calculate address
    addwf    TableOff, w, 0
    movwf    TBLPTRL, 0
    movlw    (Table >> 8) & 0xFF
    btfsc    STATUS, C, 0
    addlw    1
    movwf    TBLPTRH, 0
    movlw    UPPER Table
    btfsc    STATUS, C, 0
    addlw    1
    movwf    TBLPTRU, 0
    TBLRD    *                   ; Read byte at address
    movf     TABLAT, w, 0        ; Return the byte
    return
Table:
    db      ...
```

Interrupts When I show a basic interrupt handler for the mid-range PIC microcontrollers, along with the `w` and `STATUS` registers, I also include saving the contents of the `FSR` and the `PCLATH` registers. This is not required in the PIC18 because of the

multiple FSR registers available and the ability to jump anywhere within the application without using the PCLATH or PCLATU registers. If an FSR register is required within an interrupt handler, chances are it can be reserved for this use within the application when resources are allocated.

When a hardware interrupt request is acknowledged, the current WREG, STATUS, and BSR are saved in the fast stack. The PCLATH (and PCLATU) registers should not have to be saved in the interrupt handler unless a traditional table read (i.e., using a computed goto) is implemented instead of a table read using the built-in instructions (and shown in the previous section). The goto and branch instructions update the program counter without accessing the PCLATH and PCLATU registers. These conditions will allow a PIC18 interrupt handler with context saving to be as simple as:

```
org      8
Int

; ##### - Execute Interrupt Handler Code

retfie   1
```

so long as nested interrupts are not allowed and subroutine calls do not use the fast stack.

PROCESSOR CONTROL INSTRUCTIONS

The PIC18Cxx has the same processor instructions as the other PIC microcontrollers, but there is one instruction enhancement that I would like to bring to your attention. When designing the PIC18Cxx, the Microchip designers did something I've wanted for years: they created a nop instruction (Fig. 7.55) that has two bit patterns, all bits set and all

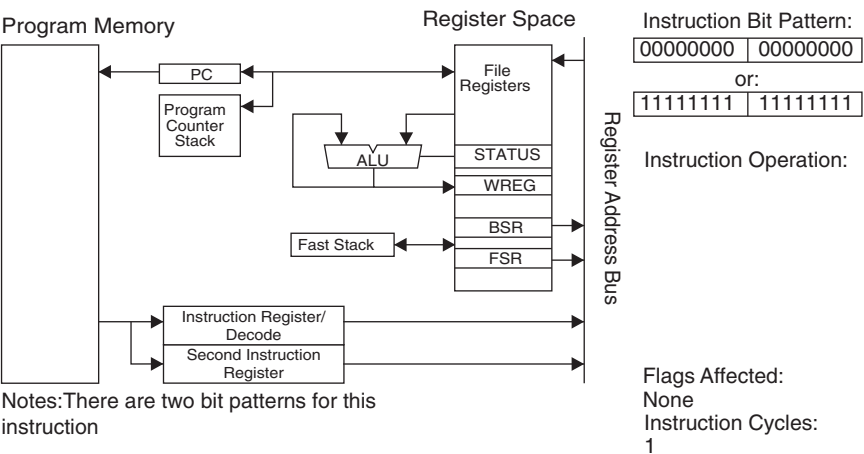


Figure 7.55 The nop instruction is coded as either all bits set or all bits reset.

bits reset. The profoundness of this instruction and what can be done with it will probably not be immediately obvious to you.

In the PIC18, just the patch space instructions that are to be modified are changed and no space is required for jumping around instructions. For the same example in the PIC18, the patch space would be:

```
dw      0x0FFFF          ; nop
dw      0x0FFFF          ; nop
dw      0x0FFFF          ; nop
dw      0x0FFFF          ; nop
dw      0x0FFFF          ; nop
dw      0x0FFFF          ; nop
```

To add three instructions to the patch space, just the required changes for the three instructions are made:

```
movf     B, w, 0          ; Formerly "dw 0x0FFFF"
addwf    C, w, 0          ; Formerly "dw 0x0FFFF"
movwf    A, 0             ; Formerly "dw 0x0FFFF"
dw      0x0FFFF          ; nop
dw      0x0FFFF          ; nop
dw      0x0FFFF          ; nop
```

Note that to add three instructions in this case, only three instructions of the patch space are modified and there is no need for a `goto` instruction to jump around the unprogrammed addresses as you would for the low-end or mid-range PIC microcontroller architectures.

ASSEMBLY-LANGUAGE SOFTWARE TECHNIQUES

The PIC[®] microcontroller is an interesting device for which to write application software. If you have experience with other processors, you probably will consider the PIC microcontroller to be quite a bit different and perhaps even “low end” if you are experienced with RISC processors. Despite this first impression, very sophisticated application software can be written for the PIC microcontroller, and if you follow the tricks and suggestions presented in this chapter, your software will be surprisingly efficient as well.

Much of the information I will give you in this book will leave you scratching your head and asking, “How could somebody come up with that?” The answer often lies in necessity—the application developer had to implement some features in fewer instructions, in fewer cycles, or using less variable memory (*file registers* in the PIC microcontroller). For most of these programming tips, the person who came up with them not only had the need to do them but also understood the PIC microcontroller architecture and instruction set well enough to look for better ways to implement the functions than the most obvious.

At the risk of sounding Zen, I want to say that the PIC microcontroller is best programmed when you are in the right “head space.” As you become more familiar with the architecture, you will begin to see how to exploit the architecture and instruction-set features to best implement your applications. The PIC microcontroller has been designed to pass and manipulate bits and bytes very quickly between locations in the chip. Being able to plan your applications with an understanding of the data paths in mind will allow you to write applications that can require as little as one-third the clock cycles and instructions that would be required in other microcontrollers. This level of optimization is not a function of learning the instruction set and some rules. Instead, it is a result of thoroughly understanding how the PIC microcontroller works and being able to visualize the best path for data within the processor and have a feel for the data flowing through the chip.

Sample Template

When I am about to create my own mid-range PIC microcontroller applications, I always start with the following template:

```
title "FileName—One Line Description"
#define _version "x.xx"
;
; Update History:
;
; Application Description/Comments
;
; Author
;
; Hardware Notes:
;
LIST R=DEC                      ; Device Specification
INCLUDE "p16cxx.inc"           ; Include Files/Registers

; Variable Register Declarations

; Macros

__CONFIG __CP_OFF & __XT_OSC & __PWRTE_ON & __WDT_OFF & __BODEN_OFF

org 0
Mainline:

    goto Mainline_Code

org 4                ; Interrupt Handler at Address 4
Int:

MainLine_Code:

; Subroutines

end
```

This template “structures” my applications and makes sure that I don’t forget anything that I consider critical. The file `template.asm` can be found in the **Templates** folder. Before starting any application, this file should be copied from the subdirectory into the MPLAB IDE project, and the specifics for the application should be added to it. When you are working with low-end or PIC18 chips, you can use this template as a basis and modify it accordingly—looking over it, the only change I would make to it for other PIC microcontroller processor architectures is to delete or change the interrupt handler code because the vector at address 0x004

is specific to the mid-range chip. I first created this template around 1998, and it has remained very constant over the years; I first started creating assembly-language templates for IBM PC assembly-language programming, and this practice has served me well with the PIC microcontroller as well as other devices.

The `title` and `_version` at the top of the file show what the application does so that I can scan the files very quickly instead of going by what the file name indicates. The `title` line will show up on the top of each new listing file page. The `_version` define statement then will show the code revision level and can be inserted in any text displayed by the application.

There may be a `Debug` define directive after the `_version` define directive if the `Debug` label is going to be tested for in the application. This directive is used with conditionally assembling code to take out long delays and hardware register operations that are not available within the MPLAB IDE simulator. Before building the application for burning into a PIC microcontroller, the `Debug` define is changed so that the “proper” code will be used with the application. Later in this book I will discuss the `Debug` defines in more detail and how they can help you with debugging your application code.

Next, I put in a description of what the application does, along with its update history (with specific changes). One thing that I do that seems to be a bit unusual is that I list the hardware operating parameters of the application. I started doing this so that I could create stimulus files easily for applications. This seems to have grown into a much more comprehensive list that provides a cross-reference between an application’s hardware circuit and the PIC microcontroller software.

Before declaring anything myself, I load in the required `include` files and specify that the default number type is to be decimal. As I will comment on elsewhere, I only use the Microchip PIC microcontroller `Include Files` because these have all the documented registers and bits of the data sheets, avoiding the need for me to create my own. There are two points here that you should recognize are implemented to avoid unnecessary work and possible errors. The first is specifying that numbers default to a decimal radix to avoid having to continually convert decimal numbers to the normal hexadecimal default. The second is to only use Microchip-developed (or in the case of high level languages, the compiler provided) chip register and constant `include` files to avoid the possibility that I will mistype registers or constants that will leave me with mistakes that are very hard to find later. It is interesting, but when I have worked with teachers, they tend to have their students specify registers and constants in the program and only work in hexadecimal; unfortunately, this causes a lot of problems that are very difficult for the students (and the teachers helping them) to find because they are in areas that are thought to be immune to errors). Another problem is that students, to avoid a few keystrokes, will give registers different labels, which adds the task of cross-referencing datasheet register names to the ones that the students have picked. I highly recommend that you save yourself some mental effort and go with the register definitions that are predefined by Microchip or the compiler vendor.

With the device declarations completed, I then do my variable, `defines`, and macro declarations. When doing this, remember to always specify prerequisites before they are used. The MPASM assembler will not be able to resolve any macro or define labels that

are defined after their first use. This is not true for labels that are referenced before their use in the application instruction code that follows the declarations and operating parameters.

Finally, I declare the device operating parameters that are to be programmed into the CONFIGURATION register or CONFIGURATION fuses (which will be explained in more detail later in this chapter) followed by the application code. I put subroutines at the end of the application code simply because the reset and interrupt handler vectors are at the “beginning” of the data space. Putting the subroutines after the mainline and interrupt handler seems to be the most appropriate in this situation.

This template is used for single-source file applications, which make up the vast majority of my PIC microcontroller applications. If multisource file applications are created, then the `__CONFIG` line is left out of anything other than the first (or header) file, which is explained elsewhere. `Publics` and `externals` are added in its place with the code following as it does in the single-source file template. Variables should be declared in the header file and then passed to the linked in files as `publics`.

This template can be modified and used with the other PIC microcontroller device architectures.

Labels, Addresses, and Flags

If you have skipped ahead in this book and taken a look at some of the code examples in the various chapters, you probably will be a bit concerned because there are a number of different ways program addresses are used that probably won't be familiar to you. The PIC microcontroller's architecture and instruction set require a more careful watch of absolute addresses than you are probably used to when programming other devices. In this section I want to discuss the different memory “spaces” in the PIC microcontroller, what is stored in them, and how they are accessed.

The most familiar memory space to you is probably the instruction program memory. As I said earlier in this book, the program memory is the width of the instruction word. The maximum “depth” of the memory is based on the word size and can have the instruction word size minus one for addressing for the low-end and mid-range PIC microcontrollers. The PIC18 is a bit different because it expands on the concepts used by the other architectures, allowing you to have a much larger program space.

To figure out the maximum depth of program memory in the low-end and mid-range PIC microcontrollers, the formula

$$\text{Maximum program memory} = 2^{**} (\text{word size} - 1)$$

is used. It is important to note that while the low-end, mid-range, and PIC18 program counters are the size of the instruction word (12, 14, and 16 bits, respectively), the upper half of the addressable space is not available to the application. This upper half to the program memory is used for storing the configuration fuses, IDLOC nibbles, and device identification bytes, as well as for providing test addresses used during PIC

microcontroller manufacturing. The PIC18 architecture is the exception to the rule because its configuration fuses can be accessed from the application using the `table read` function.

When an application is executing, it can only jump with a set number of instructions, which is known as the *page*. The page concept is discussed in more detail elsewhere in the book. The size of the page in the various PIC microcontroller architecture families is based on the maximum number of bits that could be put into an instruction for the address. In the low-end PIC microcontroller, a maximum of 9 bits are available in the `goto` instruction that is used to address a page of 512 instructions. In the mid-range PIC microcontroller, the number of bits specified in `goto` is 11, for a page size of 2,048 instructions. The PIC18 can either branch relative to the current program counter value or jump anywhere within the application without regard to page size.

The point of discussing this is to note that in these three families, addresses are always absolute values within the current page. For example, if there was the code

```
org    0
goto  Mainline

:                               ; Code to Skip Over

org    0x0123
Mainline:
```

the address value loaded into the `goto` instruction is an absolute address (given the label `Mainline`), which is 0x0123. In other processors with which you may be familiar, an offset would be added to the program counter, making the address in the `goto` instruction 0x0122 because the program counter had been incremented to the next instruction.

This can be further confused by an instruction sequence such as

```
btfsc Button, Down           ; Wait for Button to be Pressed
goto $ - 1
```

The `$` character returns a constant integer value that is the address of the instruction where it is located. The `goto $ - 1` instruction loads the address that is the address of the `goto $ - 1` instruction minus 1.

Further confusing the issue is how the PIC18 operates. The PIC18 microcontroller processor behaves more like a traditional processor and has absolute address jumps and relative address branches and does not have a page per se. The `goto` and `call` instructions in the PIC18 can change application execution anywhere within the PIC microcontroller's 1-MB program memory address range. Along with this, the PIC18 has the ability to "branch" with an 8- or 11-bit two's complement number. The branch instructions do not use an absolute address and instead add the two's complement to the current address plus two. In the PIC18, the instruction sequence

```
btfsc Button, Down, 1        ; Wait for Button to be Pressed
bra $ - (2 * 1)
```

would perform the same operation as the preceding example, but I replaced the `goto $ - 1` instruction with a “branch always.”

In the PIC18 example, if an odd address is specified, the MPLAB simulator will halt without a message. If the code is burned into the PIC18 along with a jump to an odd address, execution may branch to an unexpected address. As I noted earlier, each byte is addressed in the PIC18 and not the word. Further complicating the use of relative jumps is the instructions that take up more than one instruction word. These complexities lead me to recommend that you do not use relative jumps with the `$` character with the PIC18 and instead use a define such as

```
#define CurIns(Offset) $+(2*Offset)
```

which would be inserted into the instruction sequence like

```
btfsc Button, Down, 1      ; Wait for Button to be Pressed
bra CurIns(-1)
```

and provide the same value as the original PIC18 example but eliminate the need for you to create the formula for calculating the actual address. `Offset` in `CurIns` can be a negative or positive value.

You probably will be comfortable with how the destination values for either the `goto` and `bra` instructions are calculated depending on your previous experience. If this is your first assembly-language experience, the absolute addresses of the low-end and mid-range probably will make a lot of sense to you. If you have worked with other processors before, the PIC18 will seem more familiar to you.

Regardless of which method is the most comfortable for you, I recommend writing your applications in such a way that absolute addresses should not be a concern. This means that labels should be used in your source code at all times, and the `org` directive statement is used only for the reset vector and interrupt vectors. For all other addresses, the assembler should be used to calculate the absolute addresses for you. By allowing the assembler to generate addresses, you will simplify your application coding and make it much more “portable” to multiple locations within the same source file or others.

For example, the mid-range code

```
org    0x012
btfsc Button, Down      ; Address 0x012
goto 0x012              ; Address 0x013
```

will do everything that the preceding example code will do, but it is specific to one address in the PIC microcontroller. By using the `goto $ - 1` instruction, the code can be “cut and pasted” anywhere within the application or used in other applications.

Letting the assembler generate the addresses for you is accomplished in one of two ways. The first is the `Label`, which is placed in the first column of the source code and should not be one of the reserved instructions or directives used by the assembler. In the MPLAB assembler, a *label* is defined as any unknown string of characters. When

one of these strings is encountered, it is loaded into a table along with the current program counter value for when it is referenced elsewhere in the application.

Labels in MPLAB's assembler can have a colon (:) optionally put on the end of the string. To avoid any potential confusion regarding whether or not the label is to be replaced with its address or is a define or macro, I recommend putting the colon after it.

Using the example above, a `Loop` label can be added to make the code a bit more portable:

`Loop:`

```
btfsc Button, Down      ; Address = "Loop"
goto Loop               ; Address = "Loop" + 1
```

The disadvantage of this method is that there can only be one `Loop` (and `Skip`) put into an application. Program memory labels are really best suited for cases where they can be more global in scope. For example, an application really should only have one main `Loop`, and that is where this label should be used.

Personally, I always like to use the labels `Loop`, `Skip`, and `End` in my applications. To allow their use, I will usually preface them with something like the acronym of the current subroutine's name. For example, if the code was in the subroutine `GetButton`, I would change it to

`GB_Loop:`

```
btfsc Button, Down      ; Address = "Loop"
goto GB_Loop:           ; Address = "Loop" + 1
```

Instead of using labels in program memory for simple loops, I prefer using the `$` directive, which returns the current address of the program counter as an integer constant and can be manipulated to point to the correct address. Going back to the original code for the button poll snippet, the `$` directive eliminates the need for a label altogether:

```
btfsc Button, Down      ; Wait for Button to be Pressed
goto $ - 1
```

You do not have to expend the effort trying to come up with a unique label (which can start becoming hard in a complex application), and as you get more comfortable with the directive, you will see its what is happening faster than if a label were used.

The problem with using the `$` directive is that it can be difficult to count out the offset to the current instruction (either positive or negative). To avoid making mistakes in counting, the `$` should be done only in short sections of code, such as the one above, because the destination offset to `$` can be found. Also, beware of using the `$` directive in large sections of code that has instructions added or deleted between the destination and the `goto` instruction. The best way to avoid this is to use the `$` only in situations such as the one above, where code will not be added between the `goto` and the destination.

If you have worked with assemblers for other processors (Von Neumann), chances are that you have had to request memory where variables were going to be placed. This

operation was a result of variable memory being in the same memory space as program memory. This is not a requirement of the PIC microcontroller in which the register space (where variables are located) is separate from the program memory space.

To allocate a variable in the PIC microcontroller, you have to specify the references to a label to a file register address. In the first edition I specified that this was done by finding the first file register in the processor and then starting a list of equates from there. As discussed elsewhere, an `equate` is a directive that assigns a label a specific constant value. Every time the label is encountered, the constant that is associated with it is used. Program memory labels can be thought of as equates that have been given the current value of the program counter.

For the PIC16F84, variable `equate` declarations for an application could look like

```
i EQU 0x00C
j EQU 0x00D    ; Note, "j" is Sixteen Bits in Size
k EQU 0x00F
:
```

The problems with this method are that adding and deleting variables are a problem—especially if there are not very many free file registers available. To eliminate this problem, Microchip has come up with the `CBLOCK` directive that has a single parameter that indicates the start of a label equate. Each label is given an ascending address, and if any labels need more than 1 byte, a colon (:) and the number of bytes are added after the label. When the definitions are finished, the `ENDC` directive is specified.

Using the `CBLOCK` and `ENDC` directives, the variable declarations above could be implemented as

```
CBLOCK 0x00C                ; Define the PIC16F84 File Register Start
i, j:2, k
ENDC
```

This is obviously much simpler than the previous method (i.e., it requires less thinking), and it does not require you to change multiple values or specify a placeholder if one address is deleted from the list.

What I don't like about `CBLOCK` is that specific addresses cannot be specified within it. For most variables, this is not a problem, but as I will indicate elsewhere in this book, I tend to put variable arrays on power of 2 byte boundaries to take advantage of the PIC microcontroller's bit set/reset instructions to keep the index within the correct range. To make sure that I don't have a problem, I will specify an `equate` for the variable array specifically and ensure that it does not conflict with the variables defined in the `CBLOCK`.

The last type of data to define in the PIC microcontroller is the bit. If you look through the Microchip MPLAB assembler documentation, you will discover that there are no bit data types built in. This is not a significant problem if you are willing to use the `#define` directive to create a `define` label that includes the register and bit together.

For example, you could define the STATUS register's zero flag as

```
#DEFINE zeroflag STATUS, Z
```

A **define** is like an **equate** except where the **equate** associates a constant to the label, a **define** associates a string to the label. For the **zeroflag** define, if it were used in the code

```
movf TMR0, f                ; Wait for TMR0 to Overflow
btfss zeroflag
goto $ - 2
```

in the **btfss** instruction, the string **STATUS, Z** would replace **zeroflag**, as is shown below when the application was assembled:

```
movf TMR0, f                ; Wait for TMR0 to Overflow
btfss STATUS, Z
goto $ - 2
```

Defining bits like this is a very effective method of putting labels to bits. Using this method, you no longer have to remember the register and bit number of a flag. This can be particularly useful when you have a number of bits defined in a register (or multiple registers). Instead of remembering the register and bit numbers for a specific flag, all you have to remember is the **define** label. Using the bit **define** with the bit instructions of the PIC microcontroller allows you to work with single-bit variables in your application.

Subroutines with Parameter Passing

For subroutines to work effectively, there must be the ability to pass data (known as *parameters*) from the caller to the subroutine. There are three ways to pass parameters in the PIC microcontroller, each with their own advantages and potential problems. The first is to use global variables unique to each function, the second is to create a set of variables that are shared between the functions, and the third is to implement a data stack. Most high-performance computer systems have a stack for storing parameters (as well as return addresses), but this feature is not built into the low-end and mid-range PIC microcontrollers. In this section I want to introduce you to each of the three methods and show how they can be implemented in the PIC microcontroller.

In most modern structured high level languages, parameters are passed to subroutines as if they were parameters to a mathematical function. One value (or parameter) is returned. An example subroutine (or function) that has data passed to it would look like

```
A = subroutine(parm1, parm2);
```

in C source code.

The subroutine's input parameters (`parm1` and `parm2`) and output parameter (which is stored in `A` in the preceding above) can be shared and are common to the caller and subroutine by the following methods:

- 1** Global variables
- 2** Unique shared variables
- 3** Data stack

Passing parameters using global variables really isn't passing anything to a subroutine and back. Instead, the variables, which can be accessed anywhere in the code, are used by both the main line and the subroutine to call a subroutine that uses global variables. Just a `call` statement is used:

```
call subroutine
```

The advantage of this method is that it requires a minimal amount of amount of code and executes in the least number of cycles. The problem with this method is that it does not allow implementation of nested subroutines, and if you do want to have nested subroutines, you would have to copy one subroutine's parameters into separate variables before using the global variables for the nested subroutine call. This method cannot be used for recursive subroutines, nor can it be used for interrupt handlers that may call subroutines (or use the common global variables) that are already active. Despite these drawbacks, this method of parameter passing is an excellent way for new-application developers to pass subroutine and function parameters in assembly language because it is so simple.

The second method is to use unique parameter variables for each subroutine. Before the call, the unique variables are loaded with the input parameters, and after the call, the returned parameter is taken from one of the variables. In this case, the statement

```
A = subroutine(parm1, parm2);
```

can be implemented in assembler as

```
movf    parm1, w                ; Save Parameters
movwf   subroutineparm1        ; passed to Subroutine
movf    parm2, w
movwf   subroutineparm2
call    subroutine
movf    subroutinereturn, w     ; Get Returned
movwf   A                      ; Parameter
```

This method allows nested subroutines to be implemented and even optimizes the amount of variable space if the nested subroutine paths are plotted and the variables are chosen in such a way that the variables passed in the different paths are not reused. As with the global variable method, this method does not allow for calls from the interrupt handler, nor does it allow for recursive code. Despite these drawbacks, this method is

often the preferred method of implementation because it is fast and very memory-efficient.

The method normally used by most processors and high level languages is to save parameters on a stack and then access the parameters from the stack. As indicated earlier, the low-end and mid-range PIC microcontroller cannot access stack data directly, but the FSR (INDEX) register offsets can be calculated easily. Before any subroutine calls can take place, the FSR has to be offset with the start of a buffer:

```
movlw  bufferstart - 1
movwf  FSR
```

When the parameters are “pushed” onto the simulated stack, the operation is

```
incf   FSR, f
movwf  INDF
```

The increment of FSR is done first so that if an interrupt request is acknowledged during this operation, any “pushes” in the interrupt handler will not affect the data in the mainline.

“Popping” data from the stack uses the format

```
movf   INDF, w
decf   FSR, f
```

With the simulated stack, the example call to subroutine could use the code

```
movf   parm1, w      ; Save Parameters
incf   FSR, f
movwf  INDF
movf   parm2, w
incf   FSR, f
movwf  INDF
incf   FSR, f        ; Make Space for Return
call   subroutine
movf   INDF, w        ; Get Returned Value
decf   FSR, f
movwf  A
decf   FSR, f        ; Reset the STACK
decf   FSR, f
```

This method is very good because it does not require global variables of any type and allows for subroutines that are called from both the execution main line and the interrupt handler or recursively. In addition, data on the stack can be changed (this operation has created “local” variables).

The disadvantage of this method is the complexity required for accessing data within the subroutine and adding additional variables with low-end and mid-range PIC microcontrollers. When accessing the variables and changing FSR, you will have to disable interrupts. For the preceding example, to read `parm1`, the following code would have to used:

```
movlw    0 - 3
bcf      INTCON, GIE
addwf    FSR, f
movf     INDF, w
movwf    SUBRTN_TEMP          ; Read "param1"
movlw    3
bcf      INTCON, GIE
addwf    FSR, f
movf     SUBRTN + TEMP, w
```

The SUBRTN_TEMP variable is used to save the value read from the stack while the FSR is updated. For most changes in the FSR, simple increment and decrement instructions could be used instead and actually take fewer instructions and not require the temporary variable. The preceding code could be rewritten as

```
bcf      INTCON, GIE
decf     FSR, f
decf     FSR, f
decf     FSR, f
movf     INDF, w
incf     FSR, f
incf     FSR, f
incf     FSR, f
bsf      INTCON, GIE
```

While this code seems reasonably easy to work with, it does become a lot more complex as you add 16-bit variables and arrays.

The PIC18 can be used effectively for passing parameters on a data stack created using the FSR register and the POSTDEC# (where # is the FSR register from 0 to 2), PREINC#, and PLUSW# INDF registers. These registers will maintain the stack for you automatically and allow you to access data placed on the stack directly. To call a subroutine with two parameters and one returned, the following instructions could be used (assuming that FSR0 is already set up to point to the data stack):

```
movff    param1, POSTDEC0      ; Save Parameters
movff    param2, POSTDEC0
decf     FSR, f                ; Make Space for Return
call     subroutine
movff    PREINC0, A            ; Get Returned Value
incf     FSR, f                ; Reset the STACK
incf     FSR, f
```

This is just over half the number of instructions required for the call in low-end and mid-range devices. An even better improvement can be demonstrated reading param1, which is 3 bytes down from the top of the stack:

```
movlw    3                    ; Read "param1"
movf     PLUSW0, w
```


In these instructions, the byte that was pushed down onto the stack with 2 bytes on top of it is accessed by adding three to the stack pointer and storing the value in the `w` register (destroying the offset to the byte put there earlier). This capability makes the PIC18 a very powerful architecture to work with and allows you and compiler writers to develop code that is similar to what is used on high-end processors.

There is one method of passing parameters to and from that I haven't discussed because I do not believe that it is an appropriate method for the PIC microcontroller, and that is using the processor's registers to store parameters. For the PIC microcontroller, there is only the `w` register, which is 8 bits wide, that can be guaranteed for the task. To frustrate using this method, the low-end devices' lack of a `return` instruction prevents passing data back using the `w` register except in the case of using the `retlw` instruction and a jump to a table offset. The `zero`, `carry`, and `digit carry` STATUS register flags also could be used for this purpose, and they are quite effective for being used as pass/fail return flags.

Subtraction, Comparing and Negation

This section was originally titled "Working with the Architecture's Quirks" because there are some unusual features about the architecture that make copying assembly-language applications directly from another microcontroller to the PIC microcontroller difficult. However, as I started listing what I wanted to do in this and the following sections, I realized that there were many advantages to the PIC microcontroller's architecture and that many of the "quirks" actually allow very efficient code to be written for different situations. In this and the following sections I will discuss how the PIC microcontroller architecture can be used to produce some code that is best described as "funky."

In addition, the basic operation sequence of adding two numbers together is

- 1** Load the accumulator with the first additional RAM.
- 2** Add the second additional RAM to the contents of the accumulator.
- 3** Store the contents of the accumulator into the destination.

In PIC microcontroller assembly language code, this is

```
movf    Parm1, w
addwf   Parm2, w
movwf   Destination
```

If it's required, the `movf` and `addwf` instructions can be changed to `movlw` or `addlw`, respectively, if either parameter is a constant.

Subtraction in the PIC microcontroller follows a similar set of instructions, but because of the way the subtraction operation works, the subtracted value must be loaded first into the accumulator. For example, for the high level language statement

```
Destination = Parm1 - Parm2
```

the sequence of operations is

- 1 Load the w register with the second parameter (which is the value to be taken away from the first).
- 2 Subtract the contents of the w register from the first parameter and store the result in the w register.
- 3 Store the contents of the w register in the destination.

In PIC microcontroller assembly code, this is

```
movf    Parm2, w
subwf   Parm1, w
movwf   Destination
```

As with the addition operation, the `movf` and `subwf` instructions can be replaced with `movlw` or `sublw`, respectively, if either `Parm1` or `Parm2` is a constant.

The PIC microcontroller's instructions contrasts with those of the 8051 and other microcontroller architectures, in which the `subtract` instruction takes away the parameter value from the contents of the accumulator. As I have indicated elsewhere, the PIC microcontroller `subtract` instruction actually works as

```
PIC microcontroller subtract = parameter - w
                             = parameter + (w ^ 0xFF) + 1
```

This operation affects the zero, carry, and digit carry STATUS register flags. In most applications, it is how the carry flag is affected that is of the most importance. This flag will be set if the result is equal to or greater than zero. This is in contrast to how the carry and borrow flags work in most processors. I have described the `carry` flag after a subtract operation as a "positive flag." If the carry flag is set after a subtract operation, then a `borrow` of the next significant byte is not required. It also means that the result is negative if the carry flag is reset.

This can be seen in more detail by evaluating the `subtract` instruction sequence for

Result A – B

which is

```
movlw B      ; Assume A and B are Constants
sublw A
movwf Result
```

By starting with A equals to 1, different values of B can be used with this sequence to show how the carry flag is set after `subtract` instructions. Table 8.1 shows the result, carry, and zero flags after the snippet above.

I did not include the digit carry (DC) flag in the table because it will be the same as carry for this example. In subtraction of more complex numbers (i.e., two-digit hex),

TABLE 8.1 SUBTRACTION CARRY AND ZERO FLAG RESULTS

A	B	RESULT	CARRY	ZERO
1	0	1	1	0
1	1	0	1	1
1	2	0x0FF(-1)	0	0

the DC flag becomes difficult to work with, and specific examples for its use (such as the ASCII-to-nybble conversion routines) have to be designed.

When you are first learning how to program in assembly language, you may want to convert high level language statements into assembly language using formulas or basic guides. When you look at subtraction for comparing, the code seems very complex. In actuality, using the PIC microcontroller `subtract` instruction isn't that complex, and the instruction sequence

```
movf  Parm1, w/movlw Parm1
subwf Parm2, w/sublw Parm2
btfsc status, C
goto label
```

can be used each time the statement

```
if (A Cond B) then go to label
```

is required, where `Cond` is one of the values specified in Table 8.2.

By selecting a `STATUS` flag (carry on zero) to test, the execution of the `goto` instruction can be specified, providing you with a simple way of implementing the conditional jumps using the code listed in Table 8.3.

TABLE 8.2 if CONDITION DEFINITIONS

CONDITION	OPERATION
==	Jump if equal
!=	Jump if not equal
>	Jump if FIRST is greater than the second
>=	Jump if FIRST is greater than or equal to the second
<	Jump if FIRST is less than the second
<=	Jump if FIRST is less than or equal to the second

TABLE 8.3 CODE EXAMPLES TO BRANCH ON A SPECIFIC CONDITION

JUMP if	CONDITION TO CHECK	CODE
A == B	A - B = 0	movf A, w/movlw A subwf B, w/sublw B btfsc STATUS, Z goto Label ; Jump if Z = 1
A != B	A - B != 0	movf A, w/movlw A subwf B, w/sublw B btfss STATUS, Z goto Label ; Jump if Z = 0
A > B	B - A < 0	movf B, w/movlw A subwf B, w/sublw B btfss STATUS, C goto Label ; Jump if C = 0
A >= B	A - B >= 0	movf B, w/movlw B subwf A, w/sublw B btfsc STATUS, C goto Label ; Jump if C = 1
A < B	A - B < 0	movf B, w/movlw B subwf A, w/sublw A btfss STATUS, C goto Label ; Jump if C = 0
A <= B	B - A > 0	movf A, w/movlw A subwf B, w/movlw B btfsc STATUS, C goto Label ; Jump if C = 1

This is a useful table to remember when you are working on PIC applications, even if you aren't simply converting high level language source code by hand into PIC microcontroller assembly.

Negation of the contents of a file register is accomplished by performing the two's complement operation. By definition, this is done by inverting the contents of a register and then incrementing:

```
comf reg, f
incf reg, f
```

If the contents to be negated are in the `w` register, there are a couple of tricks that can be used to carry this out. For mid-range devices, the `sublw 0` instruction can be used:

```
sublw 0          ; w = 0 - w
; = 0 + (w ^ 0xFF) + 1
; = (w ^ 0xFF) + 1
; = -w
```

However, in low-end PIC microcontroller devices, there is a little trick you can use, and that is to add and subtract the `w` register contents with a register as shown below:

```
addwf Reg, w    ; w = w + Reg
subwf Reg, w    ; w = Reg - (w + Reg)
; = -w
```

`Reg` should be chosen from the file registers and not any of the hardware registers that may change between execution of the instructions.

Bit AND and OR

One of the most frustrating things to do is to respond based on the status of two bits. In the past, I found that I had to come up with some pretty “funky” code, only to feel like it was not good enough. To try and find different ways of carrying out these tasks, I spent some time experimenting with two skip-on-bit-condition instructions. The two skip parameters are used in such a way that the first one jumps to an instruction if a case is true, and the second jumps over the instruction if the second case is not true.

To show how the double-skip-on-bit-condition instructions could be used, consider the example of setting a bit if two other bits are true (the result is the AND of two arbitrary bits). You could use the code

```
bcf Result      ; Assume A and C = 0
btfss A
goto Skip       ; A = 0, don't set Result

btfsc B         ; B = 0, don't set Result
bsf Result      ; A = B = 1, set result
Skip:
```

This code is quite complex and somewhat difficult to understand. A further problem with it is that it can return after a different number of cycles depending on the state of `A`. If `A` is reset, the code will return after four instruction cycles. If it is set, six instruction cycles will pass before execution gets to `Skip`.

By combining the two tests, the following code could be used to provide the same function:

```
bsf Result          ; Assume A = B = 1
btfsc A             ; A == 0, Result = 0
  btfss B           ; B == 1, Result = 1
  bcf Result        ; A == 0 or B == 0, Result = 0
```

This code is smaller, always executes in the same number of cycles, and is easier to work through and see what is happening.

An OR function could be implemented similarly:

```
bcf Result          ; Assume A = B = 0
btfss A             ; A == 1, Result = 1
  btfsc B           ; A == B == 0, Result = 0
  bsf Result        ; A == 1 or B == 1, Result = 1
```

This trick of using two conditions to either skip to or skip over an instruction is useful in many cases. As I will show later in this chapter, this capability is used to implement constant-loop timing for 16-bit delay loops.

16-Bit Operations

As you start creating your own PIC microcontroller applications, you'll discover that 8 bits for data is often insufficient for the task at hand. Instead, larger base values have to be used for saving and operating on data. In the appendices I present a number of snippets for accessing 16-bit data values, but in this section I want to introduce the concepts of declaring and accessing 16-bit (and greater) variables and constants.

Declaring 16-bit variables in MPASM using the CBLOCK directive is quite simple. To declare a variable that is larger than 8 bits using CBLOCK, a colon (:) follows the variable name, and the number of bytes is specified afterward. For example, 8-, 16-, and 32-bit variables are declared in the PIC16F84 as

```
CBLOCK 0x00C
i
j:2
k:4
ENDC
```

To access data, the address with the offset to the byte address can be used as shown in the following example:

```
movf j + 1, w
```

When working with constant values, instead of coming up with arithmetic operations to capture the byte data at specific locations, you can use the LOW, HIGH, and UPPER operators (how they work is presented in Table 8.4).

TABLE 8.4 OPERATION OF LOW, HIGH, AND UPPER MPASM OPERATORS		
OPERATOR	DESCRIPTION	MATHEMATICAL OPERATION
LOW	Return low byte (bits 7–0) of value	Value & 0xFF
HIGH	Return bits 15–8 of value	(Value >> 8) & 0xFF
UPPER	Return bits 21–16 of value	(Value >> 16) & 0x3F

One confusing aspect of MPLAB for me is the default of “high/low” data storage in the MPLAB simulator and MPASM. The “low/high” format works better for using application code and makes more sense to me (this is known as *Intel format*, and the reason why it makes sense to me is because of all the years I’ve spent working with Intel processors). In addition, you will note that all 16-bit registers in the PIC microcontroller are defined in “low” (byte/address) followed by “high” (byte/address) data format, so using this format in my coding keeps me consistent with the hardware registers built into the chip processor architecture.

The preceding paragraph may be confusing for you, but let me explain exactly what I mean. If 16-bit data is saved in the “high/low” (what I think of as *Motorola format*, which is where I first saw it), when 16-bit information is displayed in memory, it looks correct. For example, if 0x1234 was stored in “high/low” format staring at address 0x10, the file register display would show

0010 1234

which appears natural.

If the data is stored in “low/high” (Intel) format, 0x1234 at 0x10 would appear as

0010 3412

which is somewhat confusing.

I recommend storing data in “low/high” format for two reasons; the first is that it makes logical sense saving the “low” value byte at the “low” address. The second reason is that in your career, you probably will work with more Intel-architected devices than Motorola devices, and you might as well get into the habit of mentally reversing the bytes now. The act of mentally reversing the two bytes becomes second nature very quickly, and I dare say that you will become very familiar and comfortable with it after working through just a few applications.

When multibyte data is displayed in MPLAB “watch windows,” the default is in the “high/low” format. Make sure that when you add a multibyte variable to the window, you click on the “low/high” selection. Working with multibyte variables is not as simple as working with single-byte variables because the entire variable must be taken into account.

For example, when incrementing a byte, the only considerations are the value of the result and the zero flag. This can be implemented quite easily for a 16-bit variable:

```
incf LOW variable, f
btfsc STATUS, Z
    incf HIGH variable, f
```

Addition with two 16-bit variables becomes much more complex because along with the result, the zero, carry, and digit carry flags must be involved as well. This code correctly computes the 16-bit result and correctly sets the zero and digit carry flags. Unfortunately, it requires five more instructions than a simple case and does not set carry correctly. To set carry correctly, a temporary variable and 20 instructions are required:

```
clrf Temporary
movf HIGH A, w
addwf HIGH B, w
movwf HIGH C, w
btfsc STATUS, C
bsf Temporary, 0
movf LOW A, w
addwf LOW B, w
movwf LOW C
btfsc STATUS, C
    goto $ + 6
incf HIGH C, f
btfsc STATUS, Z
bsf Temporary, 0
movf LOW A, w
addwf LOW B, w
xorwf LOW C, w
bcf STATUS, C
btfsc Temporary, 0
    bcf STATUS, C
```

This level of fidelity is not often required. Instead, you should pick the multibyte operation that provides you with the result that you need. In the appendices I present routines that provide the correct 16-bit result, but the STATUS flags will not be correct for the result. For correct flags, more code will be required.

MulDiv, Constant Multiplication and Division

When you get into advanced mathematics (especially if you continue your academic career into electrical engineering), you will learn to appreciate the power of arithmetic series. With a modest amount of computing power, quite impressive results can be

produced in terms of calculating data values. A good example of this is using arithmetic series to calculate a sine, cosine, or logarithmic function value for a given parameter. Arithmetic series can be used in analog electronics to prove that summing a number of simple sine waves can result in squarewave, sawtooth, or other arbitrary repeating waveforms.

An arithmetic series has the form

$$\text{Result} = + P1X1 + P2X2 + P3X3 + P4X4 + \dots$$

where the “prefix” value (P#) is calculated to provide the function value. X# is the “parameter” value that is modified for each value in the series. The parameter change can be a number of different operations, including squaring, square rooting, multiplying by the power of a negative number, and so on. For the multiplication and division operations shown here, I will be shifting the parameter by 1 bit for each series element.

The theory and mathematics of calculating the “prefix” and “parameter” for arithmetic series can be quite complex—but it can be used in cases such as producing the prefix values for simple multiplication or division operations, as I am going to show in this section. To demonstrate the operations, I have created the `multiply` and `divide` macros that can be found in the `muldiv.inc` file in the `Macros\MulDiv` folder.

The two macros provide the multiplication and division functions using MPLAB assembler capabilities that I haven’t explained yet (although I do in Chap. 10). To avoid confusion, I will explain how the macros work from the perspective of a high level language before presenting the actual code. To further help explain how the macros work, I will present them from the perspective of implementing the function in straight PIC microcontroller assembler.

Multiplication (and division) can be represented by a number of different methods. When you were taught basic arithmetic, multiplication was repeated addition. If you had to program it, you would use the high level code

```
Product = 0;
for (i = 0; i < Multiplier; i++ )
    Product = Product + Multiplicand;
```

This method works very well but will take a differing amount of time based on the multiplier (i.e., eight times something takes four times longer than two times something). This is not a problem for single-digit multiplication, but when multiplication gets more complex, the operations become significantly longer, which can have a negative impact on operation of the application code.

Ideally, a multiplication method (or *algorithm*) that does not have such extreme ranges should be used. As you would expect, this is where the arithmetic series is involved.

As you know, every number consists of constants multiplied by exponents of the number’s base. For example, 123 decimal is actually

$$123 = 1 * \text{Hundreds} + 2 * \text{tens} + 3 * \text{ones}$$

This also works for binary numbers and is used to convert constants between numbering systems. 123 decimal is 1111011 binary (0x07B). This can be represented like 123 decimal above as

$$123 = 1 * \text{sixty-four} + 1 * \text{thirty-two} + 1 * \text{sixteen} + \\ 1 * \text{eight} + 0 * \text{four} + 1 * \text{two} + 1 * \text{one}$$

In this binary sequence, I also have included any digits that are zero (which is 4 in the case of 123) because they will be used when multiplying two numbers together. This binary sequence can be used as the “prefix” of a multiplication arithmetic series if each value is used to add the multiplicand that has been shifted up by the number of the bit. The shifted-up multiplicand can be thought of as the “parameter” of the series. This series can be written out as

$$A * B = ((A \& (1 \ll 0)) \neq 0) * (B \ll 0) + \\ ((A \& (1 \ll 1)) \neq 0) * (B \ll 1) + \\ ((A \& (1 \ll 2)) \neq 0) * (B \ll 2) + \dots + \\ ((A \& (1 \ll 7)) \neq 0) * (B \ll 7)$$

This series can be converted to high level code very easily:

```
int Multiply(int A, int B)           // Multiply two eight bit
values                               values
// together and return the result
{

    int Product = 0;
    int i;

    for (i = 0; i < 8; i++) {         // Repeat for each bit
        if ((A & 1) != 0)             // Bit of Multiplier is Set,
            Product = Product + B;    // Add
                                      // Multiplicand to Product
        A = A >> 1;                  // Shift down the Multiplier
        B = B << 1;                  // Shift up the Multiplicand
    }

    return Product;                  // Finished, Return the
                                      // Result
} // End Multiply
```

This function will loop only eight times, and each time will shift up the multiplicand, which is the $B \ll 2$ term in this series, and shift down the multiplier, which is the equivalent of the $(A \& (1 \ll 0)) \neq 0$ term in this series. This term is 100 percent mathematically correct; if the prefix result is not equal to zero, then the shifted term will be added to the Product.

For example, if you were multiplying together 13 (0b01101) and 10 (0b01010), the terms would be

```

A * B = ((A & (1 << 0)) != 0)*(B << 0) +
        ((A & (1 << 1)) != 0)*(B << 1) +
        ((A & (1 << 2)) != 0)*(B << 2) +
        ((A & (1 << 3)) != 0)*(B << 3)
= ((13 & (1 << 0)) != 0)*(10 << 0) +
  ((13 & (1 << 1)) != 0)*(10 << 1) +
  ((13 & (1 << 2)) != 0)*(10 << 2) +
  ((13 & (1 << 3)) != 0)*(10 << 3)
= ((13 & 1) != 0)* 10 +
  ((13 & 2) != 0)* 20 +
  ((13 & 4) != 0)* 40 +
  ((13 & 8) != 0)* 80
= (1 != 0)* 10 +
  (0 != 0)* 20 +
  (4 != 0)* 40 +
  (8 != 0)* 80
= 1 * 10 +
  0 * 10 +
  1 * 40 +
  1 * 80
= 10 +
  40 +
  80
= 130

```

For humans, this probably seems like a very slow way of implementing multiplication, but for the PIC microcontroller, it is actually very fast and consistent. Doing an 8-bit by 8-bit multiply, the following PIC microcontroller code is used:

```

clrf  Product
clrf  Product + 1
clrf  TempMultiplicand
movlw 8
movwf Count
Loop:
  btfss Multiplier, 0          ; If Bit 0 Set, then Add
  goto Skip                   ; "Multiplicand" to the Product
  movf TempMultiplicand, w     ; Add the High Eight Bits First
  addwf Product + 1, f
  movf Multiplicand, w         ; Add Low Eight Bits Next
  addwf Product, f
  btfsc STATUS, C
  incf Product + 1, f
Skip:
  bcf  STATUS, C

```

```
rlf    Multiplicand, f           ; Shift the Multiplicand Up
rlf    TempMultiplicand, f
rrf    Multiplier, f            ; Shift the Multiplier Down for
decfsz Count, f                 ; Bit Check
goto   Loop
```

When this code is exited, `Product` will contain a 16-bit result. Note that I added a `TempMultiplicand` variable for the high 8 bits of the shifted multiplicand. Compared with the repeated addition case, i.e.,

```
clrf   Product
clrf   Product + 1
Loop:
movf   Multiplicand, w           ; Add Multiplicand to the Product
addwf  Product, f
btfsc  STATUS, C
incf   Product + 1
decfsz Multiplier, f            ; Repeat Multiplier times
goto   Loop
```

while this code takes up less than half the number of cycles than what I consider to be the “better” example and does not require an extra file register, it will require anywhere from 8 to 1,773 instruction cycles to execute (a variability of 22,063 percent). The “better” case executes in anywhere from 84 to 124 instruction cycles, which has a variability of no more than 47 percent and runs in fewer cycles for all cases except when the multiplier is less than 17.

This algorithm can be used to multiply a variable value by a constant and to allow this operation to be used easily in your PIC microcontroller assembly code. I have created the `multiply` macro shown below:

```
multiply macro Register, Value   ; Multiply 8 bit value by a
    variable i = 0, TValue       ; constant
    TValue = Value               ; Save the Constant
                                ; Multiplier

    movf Register, w
    movwf Temporary              ; Use “Temporary” as Shifted
                                ; Value

    clrf Temporary + 1
    clrf Product
    clrf Product + 1
while (i < 8)
if ((TValue & 1) != 0)           ; If LSB Set, Add the Value
    movf Temporary + 1, w
    addwf Product + 1, f
    movf Temporary, w
    addwf Product, f
    btfsc STATUS, C
    incf Product + 1, f
```

```

endif
    bcf STATUS, C                ; Shift Up Temporary
                                ; multiplicand

    rlf Temporary, f
    rlf Temporary + 1, f
TValue = TValue >> 1            ; Shift down to check the
                                ; Next Bit
i = i + 1
endw
endm

```

to multiply the contents of an 8-bit file register by a constant. The `multiply` macro is invoked as

```
multiply Register, Constant
```

where `Register` is the file register containing the multiplicand to be multiplied with the multiplier in `Constant`. The result will be stored in the 16-bit variable `Product`. The macro itself will insert the code needed to perform the operation, but without the looping functions. This method of shifting data is how I perform all multiplication in the PIC microcontroller. If you look at Appendix G, you will see that this is the method used to multiply two 16-bit numbers together.

Division is always much more difficult to perform than multiplication. Since multiplication is repeated addition, division could be thought of as repeated subtraction. The basic code version for division is

```

Remainder = Dividend;
for (Quotient = 0; (Dividend - Divisor) > 0; Quotient++)
    Dividend = Dividend - Divisor;
Remainder = Dividend;
Quotient = Quotient - 1;

```

This code also includes returning a remainder from the operation.

To simplify the division operation, I can use an arithmetic series like I did for multiplication, but this one will work differently than multiplication. I would call the series produced for multiplication a “closed” series because the series is defined for a set range of numbers. In division, this is possible for some numbers, but not for all; numbers do not always divide “evenly” (i.e., have a remainder equal to zero) into others. In these cases, a decision has to be made about what to do about them.

To come up with a division arithmetic series, I would want to use the property of division that a number can be divided by a second one by multiplying by the *reciprocal* of the second number. This can be shown as

$$A/B = A * (1/B)$$

This probably will be unexpected because fractions (which are what reciprocals really are) are not possible in the PIC microcontroller; to get around this problem, you just have to look back in the history of mathematics to see how this problem has been encountered

and solved before. Four hundred years ago, when scientists were taking the results from plotting the path of the planets about the sun and trying to come up with a general mathematical theory about the motion of the planets, they had to work with trigonometric tables. The problem with using these tables was that the decimal point had not yet been invented. Instead of having values based as fractions of 1, these tables returned the numerical fraction over 6,000. For existence, if you look up the sine of 45 degrees, you would get the value 0.707107. In the 6,000-based table, the sine of 45 degrees would be 4,243. This same principal can be applied to the finding of fractions in the PIC microcontroller. Instead with coming up with a result that is less than 1, the division method that I am going to present here calculates the fraction as a result less than 65,536 (0x010000).

By doing this, 1/3 is not processed as 0.33333 but as 21845 (0x05555 or 0b00101010101010101), 1/5 is 13,107 (0x03333 or 0b00011001100110011), 1/7 is 9,362 (0x02492 or 0b00010010010010010), and so on. It is important to note that these fractions as binary strings are repeating or “open” series, which complicates the division operations somewhat. Powers of 2 will result in a closed series, but for the most part, the fractional values will not be closed. To ensure that the result is as correct as possible, the fractional bits should be taken as far as possible, which is why I divided by 65,536 (2^{16}) and not 256 (2^8). Now that I have the fraction, I can develop the arithmetic series.

For 8-bit division, this series is

$$A / B = (((65,536 / B) \& (1 \ll 15)) \neq 0) * (A \gg 0) + \\ (((65,536 / B) \& (1 \ll 14)) \neq 0) * (A \gg 1) + \\ (((65,536 / B) \& (1 \ll 13)) \neq 0) * (A \gg 2) + \dots + \\ (((65,536 / B) \& (1 \ll 0)) \neq 0) * (A \gg 15)$$

The reason why I shift A down for each element in the series is because each test of the shifting down bit in the fraction requires that the dividend be shifted down as well. This operation could be written in a high level language as

```
int Divide(int A, int B)           // Carry out Eight Bit Division
{

    int Quotient = 0;
    int Divisor;
    int TempDividend;
    int i;

    Divisor = 65536 / B;           // Get the Fractional Divisor
    TempDividend = Dividend << 8; // Get the Dividend to be
                                   // used.
    for (i = 0; i < 8; i++ ) {    // Repeat for sixteen cycles
        if ((Divisor & 0x08000) != 0) // Have to add Quotient
            // Fraction
            Quotient = Quotient + TempDividend;
        TempDividend = TempDividend >> 1;
        Divisor = (Divisor & 0x07FFF) << 1;
    }
}
```

```

}

if ((Divisor & 0x080) != 0)      // Calculate Rounded Divisor
    Divisor = Divisor + 0x0100;
Divisor = Divisor >> 8;

return Divisor;

} // End Divide

```

As you work through this function, there should be two things that are unexpected in the code. The first is that I use 16-bit values for an 8-bit result. This was done because I wanted to get a “rounded” (to the nearest one) result. If the result has a fraction of 0.5 or greater (in which bit 7 of the result is set), then I increment the returned `Divisor`.

The second is that I shift up the dividend and divisor by 8 bits. This is done so that as I shift down the dividend, I do not lose the fractional bits of the result and cannot produce an accurate “rounding” of the result.

When I implemented this function, I had not written it out as straight PIC microcontroller assembler for use in an application. The multiplication operation, because it is “closed,” can be carried out within straight code. The division operation laid out above does not have this advantage because the result is most likely “open.”

This open result means that the PIC microcontroller’s internal functions cannot be used for calculating the fraction of the divisor. To calculate the divisor fraction, I have written the following macro with the divisor calculated by the “macro calculator” (explained later in this book):

```

divide macro Register, Value      ; Divide 8 bit value by a constant
    variable i = 0, TValue
    TValue = 0x010000 / Value     ; Get the Constant Divider
    movf Register, w
    movwf Temporary + 1          ; Use “Temporary” as the Shifted
                                ; Value

    clrf Temporary
    clrf Quotient
    clrf Quotient + 1
while (i < 8)
    bcf STATUS, C                ; Shift Down the Temporary
    rrf Temporary + 1, f
    rrf Temporary, f
    if ((TValue & 0x08000) != 0) ; If LSB Set, Add the Value
        movf Temporary + 1, w
        addwf Quotient + 1, f
        movf Temporary, w
        addwf Quotient, f
        btfsc STATUS, C
        incf Quotient + 1, f
    endif
endmacro

```

```
TValue = TValue << 1      ; Shift up to check the Next Bit
i = i + 1
endw
    movf Quotient + 1, w    ; Provide Result Rounding
    btfsc Quotient, 7
    incf Quotient + 1, w
    movwf Quotient
endm
```

The divisor fraction is calculated at assembly time and used at assembly time to create the series of instructions for dividing a variable value by a constant. The macro will produce code that ranges from 39 to 81 instructions and takes the same number of instruction cycles to execute. This is only marginally larger than the analogous multiplication code.

There are two concerns with this code. The first is that if 1 is selected as the divisor, the code will return a quotient of 0. This is due to the fact that 0x010000 divided by 1 is 0x010000 and will not cause any of the loops to add the current value. A divisor Value of 1 could be checked in the macro and an error returned if this is a potential problem. The second problem is a bit more insidious and is reflective of how division algorithms work. The quotient returned is rounded to the nearest 1. In many applications requiring a division operation, this would not be acceptable—instead, the quotient and remainder would have to be returned.

This macro was written to round the value so that indicator operations (such as RPM in a tachometer) could be implemented quickly and efficiently. The value returned from the `divide` macro should not be passed onto any other arithmetic functions to prevent the error in the result from being passed down the line. If the quotient were required for subsequent operations, I would suggest that you use either the 16-bit division routine presented in Appendix G. If this macro is to be used, then the entire 16-bit quotient calculated by this macro (the lower 8 bits being the fractional value less than 1) is passed along with the final result divided by 256 (by “lopping off” the least significant byte).

Delays

Delays are often critical aspects of an application. In PIC microcontroller applications, it is not unusual to have microsecond, millisecond, or even full-second delay routines built in. In the first edition of this book I didn’t do a very good job of explaining how to create useful delays and how they are used in applications. In this section I want to clear up the errors I made and help you to understand how adding delays to an application can make your life simpler, as well as help you to understand how critically timed application code works.

The basic unit of timing in an application is the *instruction cycle*. The instruction clock rate is one-quarter the external clock frequency (as was explained earlier in this book). The reciprocal is the *instruction cycle period*. The instruction cycle period is found using the formula

Instruction cycle = 4/clock frequency

Thus, for a clock frequency of 3.58 MHz, the instruction cycle is found as

$$\begin{aligned}\text{Instruction cycle} &= 4/\text{clock frequency} \\ &= 4/3.58 \text{ MHz} \\ &= 1.12 \text{ ms}\end{aligned}$$

Actual time delays should be converted into instruction cycle delays as quickly as possible. The formula I use for doing this is

$$\text{Instruction Delay} = \text{Time Delay} * \text{clock frequency}/4$$

For example, if you had a PIC microcontroller running at 10 MHz and wanted a 5-ms delay, the preceding formula would be used:

$$\begin{aligned}\text{Instruction Delay} &= \text{Time Delay} * \text{clock frequency}/4 \\ &= 5 \text{ ms} * 10 \text{ MHz}/4 \\ &= 50 * (10 ** 3)/4 \\ &= 1.25 * (10 ** 4) \\ &= 12,500\end{aligned}$$

Thus, for a delay of 5 ms in a PIC microcontroller running at 10 MHz, 12,500 instruction cycles would have to execute.

For a one-instruction delay, a `nop` instruction is used. For two cycles, the `goto $ + 1` instruction is used. Four cycles can be implemented by calling a subroutine that simply returns. The two instructions take four instruction cycles to execute:

```
:
call Dlay4          ; Delay 4 instruction cycles
:
```

```
Dlay4:
return
```

This won't seem that special until you realize what can be done with it. By putting on another "layer" to the subroutine that calls `Dlay 4`, you can double the delay very simply. For example, to delay 16 instruction cycles, you could use the code

```
:
call Dlay16         ; Delay 16 instruction cycles
:
```

```
Dlay16:
call Dlay8
Dlay8:
call Dlay4
Dlay4:
return
```

In this code, when `Dlay16` is called, the `call` instruction requires two instruction cycles to reach `Dlay16`, the call to `Dlay8` requires an additional two instruction cycles, and it calls `Dlay4` for a total of 6 instruction cycles. When `Dlay4` returns, 8 instruction cycles have executed. When the code returns to the call at `Dlay8`, it then returns to the call at `Dlay16`, which continues executing to `Dlay8`, and the process continues. As you work through the four instructions above, you will find that a total of 16 instruction cycles are executed by the `Dlay16` subroutine.

For longer delays, I recommend a loop for two reasons. The first is because of the limited program memory stack built into PIC microcontrollers; the low-end PIC microcontrollers have a two-entry deep stack, which means that the 16-cycle delay uses more stack entries than are available in low-end devices. Even if an 8-cycle delay were implemented as shown above, there would be no stack space available for subroutine calls. In mid-range PIC microcontrollers, the three subroutine calls of the 16-instruction-cycle delay code are probably the practical maximum (with interrupts in the application). The second reason is that this method is generally suboptimal for delays that are not powers of 2. For example, to get a 31-instruction-cycle delay, the following calls and instructions are required:

```
call Dlay16
call Dlay 8
call Dlay 4
goto $ + 1
nop
```

Using loop code, the same 31-cycle delay could be implemented in one less instruction and does not change the `w` and `STATUS` registers:

```
movwf _w
movlw 9
movwf Dlay
decfsz Dlay, f
    goto $ - 1
swapf _w, f
swapf _w, w
```

The loop code's only real disadvantage is that it requires two variables (one for saving the `w` register and one for counting down the value).

Single-variable loops, such as the preceding example, can work in three-cycle increments (like above) or in four-cycle increments by using the `w` register as the temporary register:

```
movwf _w
movlw 7
addlw 0 - 1
btfss STATUS, Z
    goto $ - 2
```

```
swapf _w, f
swapf _w, w
```

This code does not change the w register but does change the STATUS register. Using these two methods, single-variable loops can delay up to 768 or 1,024 instruction cycles.

For longer delays, TMR0 using the instruction clock and the prescaler could be used, which will be discussed in Chap. 9. For many applications, a TMR0-based (or other timer-based) delay is much more efficient than using the software delays presented here.

For long delays, I recommend using 2 bytes for the delay and decrementing them. When I wrote the first edition, I suggested the rather simple code

```
movlw HIGH Value
movlw Dlay + 1
movlw LOW Value
movlw Dlay
Loop:
  Decfsz Dlay, f
  goto Loop
  decfsz Dlay + 1, f
  goto Loop
```

While this is very simple, it is also very cumbersome to be able to derive a formula for Value that will give a reasonably precise delay. By simply clearing the 16-bit Dlay variable, you get a delay of approximately 200 instruction cycles (or 0.2 second for a PIC microcontroller running at 4 MHz).

A much better way of creating a moderately long delay is the five-instruction-cycle loop using a 16-bit value for the delay:

```
movlw LOW Value
movwf Dlay
movlw HIGH Value
movwf Dlay + 1
Loop
  decf Dlay, f
  btfsc STATUS, Z
  decfsz Dlay + 1, f
  goto Loop
```

In this code, each time through the loop will take five instruction cycles. To calculate the Value to be loaded into the Dlay variables, I use the simple formula

$$\text{Value} = ((\text{delay} * \text{frequency}/4)/5) + 256$$

The + 256 in the formula increments the high byte so that the value stored into the low byte will execute along with the number of times the high byte has to be decremented with the low byte set to zero (which causes the code to loop 256 times).

For the example of a 5-ms delay in a 10-MHz PIC microcontroller, `Value` is calculated to be

$$\begin{aligned}\text{Value} &= ((\text{delay} * \text{frequency} / 4) / 5) + 256 \\ &= ((5 \text{ msec} * 10 \text{ MHz} / 4) / 5) + 256 \\ &= (12,500 / 5) + 256 \\ &= 2,500 + 256 \\ &= 2,756 \\ &= 0x0AC4\end{aligned}$$

For this method, a maximum value of 65,505 (0xFFFF) can be calculated for `Value`, but I prefer stopping at 50,256 (0xC450), which gives a 250,000-instruction-cycle delay, which can be built on by an outside loop. For example, in the 10-MHz PIC microcontroller, to get a 3-second delay, I would first calculate the delay of 250,000 instruction cycles, which is

$$\begin{aligned}\text{Delay} &= \text{Instruction Cycles} * 4 / \text{frequency} \\ &= 250,000 * 4 / 10 \text{ MHz} \\ &= 0.1 \text{ Seconds}\end{aligned}$$

To get a 3-second delay, this 250,000-instruction-cycle delay would have to execute 30 times. Loading a register with the value of 30 and using the PIC microcontroller's `decfsz` instruction would accomplish this:

```
movlw 30          ; Load the Outside Loop
movwf Outside
OuterLoop:
movlw LOW 0x0C450 ; Inside 250,000 instruction Delay
movwf Dlay
movlw HIGH 0x0C450
movwf Dlay + 1
Loop:
decf Dlay, f
btfsc STATUS, Z
decfsz Dlay + 1, f
goto Loop
decfsz Outside      ; Repeat 30x for a 3 Second Delay
goto OuterLoop
```

You might notice that each time `OuterLoop` is executed, four extra cycles to load the `Dlay` variable are required and three extra cycles to decrement `Outside` and jump back to `OuterLoop` are required for a total of seven each time through the loop. The actual number of cycles required for this 3-second delay is

$$\begin{aligned}\text{Total Cycles} &= (29 * 250,000) + (29 * 7) + 2 \\ &= 7,250,000 + 203 + 2 \\ &= 7,250,205\end{aligned}$$

The final two instructions added to the total cycles are the two instructions required to load `Outside` with 31.

For a “true” 3-second delay, a total of 7.5 million instruction cycles is required, but in the preceding code, an extra 205 instruction cycles is added. If you are thinking of changing the `Loop` delay value to better match the desired number of instructions, consider what the error of these 205 instruction cycles is over 3 seconds:

$$\begin{aligned}\text{Total Error} &= 205 / 7,500,000 * 100 \text{ percent} \\ &= 0.002733 \text{ percent} \\ &= 27.33 \text{ ppm}\end{aligned}$$

With this low error level, I would recommend that any extra cycles in large delays such as this be ignored. Even in the case where the timer is used, the error is quite slight and approaches the tolerance of the PIC microcontroller’s clock.

Patch Space

Patch code is instruction space left in the program memory of a computer system to allow changes to be added to an application without having to rebuild it. Using patch code, changes would be made locally to a processor’s program memory to allow the developer to try different things before going through the chore of changing the source, rebuilding it, and then loading it into the system to try again. When I was first taught assembly-language programming, I was always told to leave patch space in my applications. This space would be used to add code to help fix applications by providing space in which updated code that corrected the problem could be added. Using patch space was something I never did very well. I always found that I never accurately documented the changes I had made, and thus the entire debug process was slower than if I simply tried to figure out what was wrong and changed the source code directly.

The reasons for providing and using patch space are

- 1 Application rebuild and reprogramming is a long and tedious process.
- 2 Application memory is usually RAM and can be changed easily in a debugger.

Both these arguments are largely false for the PIC microcontroller. Using tools such as MPLAB IDE, changes to an application source can be done in literally seconds. Programming using an ICD application can be accomplished within a minute or so. When I first learned assembly-language programming (for the Motorola 6800), a source-code change was made and reassembled on a main frame, downloaded to a minicomputer, downloaded onto a cassette tape, and then downloaded into the 6800 computer system. This was very complex and presented a lot of opportunities for problems compared with using MPLAB IDE with a Microchip programmer for developing software and programming PIC microcontroller devices.

Many PIC microcontroller applications can use Flash-based parts for application debug or an emulator that can allow changes in source code to be transferred to the application's PIC microcontroller almost immediately. Furthermore, one of the characteristics of Flash memory is that blocks of it can be erased at any time, resulting in large areas of memory that have to be reprogrammed—this means that while space can be left for updating the application, it can only be used once. These capabilities basically invalidate the arguments for providing patch space in an application.

However, there are some cases where only an EPROM PIC microcontroller part can be used, and there is no emulator available for it. The memory that is going to be used for patch space must be left erased (all bits set if they were to be read back) so that a new set of instructions can be programmed in. In the case of an EPROM PIC, some program memory can be devoted to providing patch code in the application.

The typical way to do this is to define a block of code inline to the application with a jump around it. The format for this is

```
goto $ + size + 1
variable i = 0
while (i < size)
    dw 0x3FFF
i = i + 1
endw
```

In Chap. 10, I will explain the format of the `variable`, `while`, and `wend` directives, but the preceding code defines a block of program memory `size + 1` instructions long that can be loaded with any instructions. The `dw 0x3FFF` instructions leave each memory untouched and ready to accept patch code. To put instructions into the patch-code space, the first `goto` is overwritten with `0x0000`, which is the NOP instruction. Following this, the instructions can be programmed into the PIC microcontroller's program memory.

To show what I mean, let's go through an example. An application has the statement

```
Result = 47 - B</CP>
```

which the developer has coded

```
movlw 47
subwf B, w
movf Result
```

When this was entered into the source code, the developer was not sure that it was correct. In case it wasn't, the developer provided three instructions of patch-code space to rewrite the code in case the original code was wrong. The actual source code used for the application was

```
movlw 47
subwf B, w
```

```

movwf Result
goto $ + 4           ; Jump Over Patch Code
dw 0x3FFF
dw 0x3FFF
dw 0x3FFF

```

In the course of the application's debug, the developer discovered that the original subtraction method was incorrect and that the use of 47 and B had to be reversed. To try out the code, the original subtraction operation is overwritten with zeros (to make the three instructions `nops`), along with the `goto` instruction. Next, the three correct subtraction instructions are programmed in. These seven instructions in program memory become

current instructions	previous instructions
<code>nop</code>	<code>; movlw 47</code>
<code>nop</code>	<code>; subwf B, w</code>
<code>nop</code>	<code>; movwf Restart</code>
<code>nop</code>	<code>; goto \$ + 4</code>
<code>movf B, w</code>	<code>; dw 0x3FFF</code>
<code>sublw 47</code>	<code>; dw 0x3FFF</code>
<code>movf Restart</code>	<code>; dw 0x3FFF</code>

If you look at the PIC18's instruction set, you will see that `nop` instructions can either be 0xFFFF or 0x0000. This allows patch space to be put inline as all 0xFFFF, eliminating the need for jumps around the saved patch area.

Of course, to be able to provide patch-code areas, you will need a programmer that can write to specific addresses with specific values. The programmers discussed in this book do not have this capability (they are designed to write the entire program memory). Some programmers can be "fooled" into providing this function by changing the source code so that the old code and jump-over are zeroed out (and become `nops`) and then are programmed into the PIC microcontroller. Before making the change, the programmer will issue an error message at the start indicating that the PIC microcontroller isn't blank and that you will carry on with the programming regardless.

Structures, Pointers, and Arrays

If you have been trained in a classical programming environment, you are probably very comfortable with working with structures and pointers. Complicating matters, you may want to use these programming constructs with arrays. Developing applications with these features is possible for the PIC microcontroller using MPLAB's assembler, although they are used rarely because of the lack of large contiguous file register space in most of the PIC microcontroller device families and the difficulty in working with the PIC microcontroller processor. Despite these difficulties, structures, pointers, and arrays can be created in PIC microcontroller assembly language, although using them will be somewhat more complex than what you are used to with other processors.

The three programming constructs presented here all will require use of the FSR register and the bank select features of the PIC microcontroller. In actuality, this will be the most challenging and sophisticated use of the FSR that you will see in this book, and you may want to try out the code snippets provided here to see exactly how they work.

The MPLAB assembler does not have an explicit “structure” directive that can be used to define a structure within the application. In other assemblers and compilers, the structure directive is used to set up a single data type consisting of a number of other data types.

In C, a structure is defined like

```
struct VarStruct {           // Define a Structure
    int varA;                // 16 Bit Variable
    char varB;               // 8 Bit Variable
    char * NextVarStruct;    // Pointer to an 8 Bit Variable
};
```

When a variable is declared, the structure is used as the data type. In C, a variable defined from the preceding structure would be declared as

```
struct VarStruct VarValue;
```

To access the structure elements in `VarValue`, the element name is added to the variable name with a period or dot (`.`):

```
VarValue.varA = VarValue.varB * 4;
```

In this statement, the structure element `varA` is loaded with the contents of the structure element `varB` after it has been multiplied by 4. This capability allows quick and easy structure variable element access and modification.

This is not a high level language-specific capability; many assemblers allow structures to be defined and used with structure variables. The MPLAB assembler does not, but you can still specify a structure and use it in MPLAB using the `CBLOCK` directive. The `VarStruct` structure shown above can be defined in MPLAB as

```
; VarStruct – Define an MPLAB Structure
CBLOCK 0                ; Start with Offset equal to Zero
varA:2                  ; 16 Bit Variable
varB                    ; 8 Bit Variable
NextVarStruct:2        ; Pointer to an 8 Bit Variable
SizeOfVarStruct        ; Set to Number of Bytes of “VarStruct”
ENDC
```

The last entry in the `CBLOCK` statement above is assigned the number of bytes in the structure and will not be used except for keeping track of the number of bytes required by the structure. The `SizeOf` built-in C function returns the size of a structure, which is why I chose the identifier used above for this structure function.

To declare a structure variable in MPLAB, the `CBLOCK` directive is used in the declaration

```

    CBLOCK 0x??                ; Define Variables at File Register
Start
:
VarValue:SizeOfVarStruct      ; Define the Structure Variable
:
    ENDC

```

In these `CBLOCK` statements, `VarValue` is defined along with the other variables the size of the structure (as returned by `SizeOfVarStruct`). For this declaration to work properly, the structure must be defined before the variable to ensure that `SizeOfVarStruct` is valid and does not change between the assembly passes.

To address the structure elements in the structure variable, the structure element offset (defined by the first `CBLOCK`) is added to the address of the structure variable. To show how this is done, the PIC microcontroller assembly code for the example C statement

```
VarValue.varA = VarValue.varB * 4;
```

could use the code

```

clrf  VarValue + varA + 1      ; VarValue.varA = VarValue.varB
movf  VarValue + varB, w
movwf VarValue + varA
bcf   STATUS, C                ; VarValue.varA = VarValue.varB*2
rlf   VarValue + varA, f
rlf   VarValue + varA + 1, f
rlf   VarValue + varA, f ; VarValue.varA = VarValue.varB*4
rlf   VarValue + varA + 1, f

```

Pointers are variable types that can point to other variables anywhere within the application variable memory space. In the PIC microcontrollers, there are no devices that have more than 4,096 file registers, so the data size I normally use for PIC microcontroller pointers is a 16-bit variable.

Depending on the PIC microcontroller architecture, pointers can be somewhat confusing and difficult to work with, although after a few second's thought you probably can come up with a method for accessing registers in the different devices. For low-end devices, pointers are quite simple to implement with a single 8-bit register that can be stored in the FSR register. The general case for mid-range PIC microcontrollers is 9 bits to fully access registers in the four banks, which requires a 16-bit variable. For the PIC18, I would recommend that the 4-bit bank address of the selected register be included with the 8-bit register address as part of a 16-bit variable.

Having a pointer address an 8-bit register is quite easy to implement, but pointers become much more complex if they are pointing to a structure. The last structure element

in the `VarStruct` examples above is designed as a pointer to the next structure variable in a “list” of `VarStructs`. To implement a list of `VarStructs`, three structure variables can be defined along with a pointer to `VarStructs` that will have structure variable `strung` along in a list. In C, the variables would be declared using the code

```
struct VarStruct StructureA;  
struct VarStruct StructureB;  
struct VarStruct StructureC;  
struct * VarStruct StructurePtr;
```

Note that `StructureA`, `StructureB`, and `StructureC` are each used to define the 5 bytes of `VarStruct` above. `StructurePtr` simply points to a structure and initially has no value behind it.

In C, returning a pointer address is accomplished by using the splat (*) character, and the ampersand (&) is used to return the address of a value.

To set up the list of `StructureA` pointing to `StructureB` pointing to `StructureC` and `StructurePtr` pointing to the start of the list (`StructureA`), the following C code is used:

```
StructureA.NextVarStruct = &StructureB;  
StructureB.NextVarStruct = &StructureC;  
StructurePtr = &StructureA;
```

To access a structure pointer in the list, the `StructurePtr` pointer is used with the `->` operator, which indicates that it is referring to the pointed to value. To set `StructureB.varB` to 0x012, the following code would be used:

```
StructurePtr -> NextVarStruct -> varB = 0x012;
```

As you would imagine, this is quite complex to do in the PIC microcontroller assembler and following the operation of the pointer. To give you an idea of how this would be done, the following code shows the assembler code for the preceding C statement using the `SetPointer` macro:

```
SetPointer macro Address  
    if ((Address & 0x0100) == 0)  
        bcf    STATUS, IRP  
    else  
        bsf    STATUS, IRP  
    endif  
    movlw    Address 0x0FF  
    movwf    FSR  
endm
```

This macro was written for the mid-range PIC microcontroller, and to simplify operation of the pointer load, I made my pointer structure 9 bits (as discussed earlier) in size, with the value being the register address used in the Microchip device `.inc` file. The

macro's `Address` parameter will be loaded into the FSR register and IRP bit of the STATUS register:

```

    SetPointer StructurePtr    ; Point to the Start of the List
; (StructureA)
    movlw    NextVarStruct      ; Point to the next List Element
; (StructureB)
    addwf    FSR, f
    movf     INDF, w            ; Get the Low Byte of the Next List
; Element
    movwf    PointerTemp
    incf     FSR, f
    movf     INDF, w            ; Get the High Byte of the Next List
; Element
    movwf    PointerTemp + 1
    bcf      STATUS, IRP      ; Load the Next List Element into FSR
    btfsc    PointerTemp + 1, 0
        bsf   STATUS, IRP      ; AND the IRP
    movf     PointerTemp, w
    movwf    FSR
    movlw    varB              ; Point to the Element within the
                                ; Structure

    addwf    FSR, f
    movlw    0x012              ; Finally, do the Assignment
    movwf    INDF

```

Despite the simplification of the macro, the PIC microcontroller code to implement the pointer statement is still quite complex and, more important, very confusing. Trying to follow the preceding code will be a bit difficult—which is characteristic of assembly-language pointer programming.

Looking at the preceding code, one consideration about creating pointers and structures in the PIC microcontroller is not readily apparent, and that is the importance of never having a structure go over a bank boundary. This is probably obvious, but I just point it out as something to watch for if your application grows, and the latest structure you added to the application code seems to cause the PIC microcontroller to lock up or fail in strange ways. When a structure goes over a bank boundary, accesses to the structure will modify the PIC microcontroller's context registers (such as the STATUS, PCL, and OPTION registers) at the start of the bank, causing all kinds of chaos.

If you are new to programming, you will find working with pointers confusing. I'm not new to programming, and I still find pointers confusing. I realize that there are times when programming constructs such as the linked list example earlier is required for an application, but they should be avoided unless absolutely necessary in the PIC microcontroller. With a bit of thought about the application design, they should not be required at all.

The last programming construct I want to introduce you to in this section is arrays. Elsewhere in the book I have discussed PIC microcontroller array programming, but I

want to discuss some advanced aspects of it, including how arrays can be used with structures. Arrays with pointers are simply too complex to implement in the PIC microcontroller assembler to even think about it.

Multidimensional arrays have to be defined according to the amount of space that is required. If you are going to implement a 5 by 5 array of bytes, the CBLOCK directive can be used as follows:

```
CBLOCK 0x??  
:  
Array:3*5                ; Define a 5 by 5 array  
:  
ENDC
```

While the declaration of this array is simple, working with it is not. For example, accessing the byte at array element 2, 3 would require multiplying the first dimension specification by 5 before the second dimension specification is added to it. For the C code

```
Array[2][3] = 0x012;      ; Load Element 2, 3 with 0x012
```

the PIC microcontroller assembler code would be

```
movf   Parm1, w           ; Multiply the first dimension specified  
movwf  FSR                ; by 5  
bcf    STATUS, C          ; Multiply the first dimension by 4  
rlf    FSR, f             ; first  
bcf    STATUS, C          ;  
rlf    FSR, f             ;  
addwf  FSR, f             ; Add first dimension to 4x first  
                        ; dimension to get 5x  
movf   Parm2, w           ; Add the second dimension to 5x first  
addwf  FSR, f             ;  
movlw  0x012              ; Do assignment at array element 2,3  
movwf  INDF
```

Note that the multiply by 5 uses the function of multiplying by a power of 2 and then adding the value again to get the odd multiplier. Without using this trick, this array access code would be much more complex and probably require the use of a temporary variable.

To simplify this operation, I would like suggest two improvements. The first is to change the way the array is declared to 5 by 4 from 5 by 3. Four is a power of 2 and very easy to multiply by. By doing this, 5 bytes are added to the array, and hopefully, this is not a significant amount of memory in the application (it could be for something like a low-end PIC microcontroller, where only 16 unique file registers are available in each bank).

Second, I would reverse the order in which data is stored in the array. Instead of the first parameter being multiplied by 5, I want the second parameter to be multiplied by 4. Making these changes, the code becomes

```

bcf    STATUS, C      ; Multiply the second dimension
rlf    Parm2, w       ; specified by four
movwf  FSR
bcf    STATUS, C
rlf    FSR, f
movf   Parm1, w       ; Add first dimension to second*4
addwf  FSR, f
movlw  0x012          ; Do assignment at array element 2,3
movwf  INDF

```

Making these two changes results in an over 20 percent decrease in application code size. It is debatable whether or not it is easier to read and understand than the first example (it is for me).

Arrays can be used with structures to allow quite complex data-tracking operations. For example, a 5 by 3 array of `VarStruct` could be created. This would be done in C using the code

```
struct VarStruct VarStructArray[3][5];
```

In the MPLAB assembler, this operation is also quite simple:

```

CBLOCK 0x0??
:
VarStructArray:3*5*SizeOfVarStruct ; Define a 5 by 5 VarStruct
array
:
ENDC

```

To access elements in the two-dimensional array, the same code as above is used, but once the result is calculated, it will have to be multiplied by the size of `VarStruct` (which is 5). For example, the C code statement

```
VarStructArray[2][3].varB = 'A';
```

would be implemented in PIC microcontroller assembler as

```

movf   Parm1, w       ; Multiply the first dimension specified
movwf  FSR            ; by 5
bcf    STATUS, C      ; Multiply the first dimension by 4
rlf    FSR, f         ; first
bcf    STATUS, C
rlf    FSR, f
addwf  FSR, f         ; Add first dimension to 4x first
; dimension to get 5x
movf   Parm2, w       ; Add the second dimension to 5x first
addwf  FSR, w         ; Save first * 5 + second and multiply
movwf  FSR, w         ; it by the "SizeOfVarStruct" which is
bcf    STATUS, C      ; 5

```

```
rlf    FSR, f
bcf    STATUS, C
rlf    FSR, f
addwf  FSR, f
movlw  "A"           ; Do assignment at structure element
movwf  INDF           ; in array element 2,3
```

Even when working with single-byte multidimensional arrays, make sure that you are always aware of what the ultimate array size is. The preceding example produces the same result as if a three-dimensional array of 3 by 5 by 5 was created. If the array were increased in size to 5 by 5 by 5, 125 file registers would be required, which is outside the capabilities of all the PIC microcontrollers except for the PIC18, unless you are playing around with bank registers (and the base address was desired). Implementing single arrays across multiple register banks is something that I highly recommend you avoid, and if there is no other way of implementing the application, then you should be looking at another microcontroller.

Sorting Data

One area in which the PIC microcontroller is deficient is in sorting data. The reason for this comment is the relatively small file register “banks” for storing array data, as well as the single FSR index address register in low-end and mid-range PIC microcontrollers. These two features make sorting data quite difficult and inefficient.

Having said this, there are always cases where code for sorting data is required. I have come up with my own bubble sort code for the PIC microcontroller. The **bubble sort** algorithm consists of repeatedly running through an array of values, comparing one value to the next, and if the next value is lower than the current, they are swapped. The algorithm ends when it can be determined that every element has been checked in every position in the array (which makes the algorithm quite slow and is known as an “order N squared,” which means that how long it takes to execute is proportional to the number of elements squared). The example C pseudocode for the subroutine follows:

```
BubbleSort(Int reg[4])
{
    int i, j;

    for (i = 0; i < 5; i++)    // Test 1x for each array value
        for (j = 0; j < 4; j++) // Compare entire array
            if (reg[j] > reg[j + 1]) {
                Temp = reg[j];    // Swap if next is lower than current
                reg[j] = reg[j + 1];
                reg[j + 1] = Temp;
            } // endif
    } // End BubbleSort
/
```

TABLE 8.5 REQUIRED VARIABLES FOR THE SORTING ROUTINE

VARIABLE	FUNCTION
reg1	Start of array of values to be sorted
Rega	Array of sorted values
Next	Location to put the next sorted value
Llow	Value of the last lowest number
Addr	Location of the last lowest number
Lend	Location of the list end

The input is a single-dimensional array of values, and the output is a single-dimensional array of the values sorted in ascending order. The variables used by the code are listed in Table 8.5.

The code is as follows and is designed for sorting four values. The size of the array to be sorted can be increased easily by changing the `lend` value before the start of the routine:

```

Sort:
; Now, In the Sorting Routine
movlw    rega        ; Setup Where you are Storing the Result
movwf    next
movlw    reg4        ; For Shrinking List, Get the Last Addr
movwf    lend        ; Watch for the Ending Value

Loop:
; Loop Around Here Until List is Empty
movlw    reg1        ; Load FSR for Searching for the lowest
movwf    FSR
movwf    addr        ; At Start, Assume the First is Lowest

movf     INDF, w      ; Get the Current and Use As the Lowest
movwf    llow        ; Save it as the Current Lowest

Loop2:
; Loop Here Until FSR = lend
movf     FSR, w       ; Are we at the end?
subwf    lend, w
btfsc    STATUS, Z    ; If Zero Flag is Set, We're At the End
goto     Save         ; Save the Currently Lowest Value

incf     FSR, f       ; Now, Look at the Next Value

movf     llow, w
subwf    INDF, w      ; Do we Have Something that's Lower?
btfsc    STATUS, C    ; If Carry Set, then current is Lowest
goto     Loop2

```

```
movf    INDF, w      ; Current is the Lowest - Save It
movwf   llow
movf    FSR, w       ; And, Save the Address It's at
movwf   addr
goto    Loop2        ; Loop Around and Look at the Next

; The List has been Checked and "low" and "addr" have Lowest
; Current
; Value and its Address, Respectively.
Save:                ; Now, Save the Currently Lowest Value
movf     next, w     ; Store it in the FSR
movwf    FSR

movf     llow, w      ; Get the Lowest
movwf    INDF         ; Store it in the Sorted List

movf     next, w      ; Are we at the End of the List
sublw    regd
btfsc    STATUS, Z    ; If NOT Zero, then Loop Around
goto     PEnd         ; Else, They Match, End the Program
incf     next, f      ; Increment Pointer to the Next Value

; The Lowest Current Value has been Put in the "Sorted" List,
; Now, Shorten the List at the Value we took out
movf     addr, w      ; Get Address the Value was Taken Out Of
movwf    FSR          ; Put in the FSR for Later

Loop3:                ; Now, Loop Around Storing the New List
movf     FSR, w       ; Are we at the End of the List?
subwf    lend, w
btfsc    STATUS, Z    ; Is the Zero Flag Set?
goto     Skip         ; Yes, List Has been Copied

incf     FSR, f       ; Get Next Value and Store in Current
movf     INDF, w
decf     FSR, f
movwf    INDF
incf     FSR          ; Increment the Index and Loop Around
goto     Loop3

Skip:
decf     lend, f      ; Decrement the Ending Address
goto     Loop

; Sort is All Finished
PEnd:                ; Program End
return
```


There's a bit of a story to this routine. On the PICLIST, somebody asked for a routine that sorted four numbers in one list and put them in another. After working for about 3 hours, I came up with the preceding solution.

It was quite an eye opener when I saw what other people came up with; while I improved the baseline code by about three times (three times shorter execution time and about a third of the original code size), the best solutions were almost a hundred times better!

The solutions presented were designed to do exactly what was required: Sort four numbers from one list and put them in another. Developing a macro for comparing two values and putting the lowest first, the solution was

```
least regc, regd          ; Move the Lowest Value to Front of
least regb, regc          ; beginning of List
least regb, rega
least regc, regd          ; Move 2nd Lowest Value to 2nd from
least regb, regc          ; the Front
least regc, regd          ; Put the two highest in order
```

The macro used to accomplish this was

```
least macro reg1, reg2
    movf reg2, w
    subwf reg1, w
    btfsc STATUS, C          ; If no Carry, Swap the Values
    goto $+6                ; Else, Skip over the Rest
    movf reg1, w             ; Now, Swap the Values
    xorwf reg2, w
    xorwf reg2
    xorwf reg2, w
    movwf reg1
endm
```

The lesson I learned in all this is to understand what the customer wants. While the routine I created is very clever and took a bit of work, it is too general, and a more specific solution was perfect for the customer's requirements.

Because of the hardware restrictions in the PIC microcontroller, I limit the sorting routines to basically what you see here. In other microcontrollers with processors that can access large amounts of memory indirectly and have built-in data stacks, I would recommend using the `QuickSort` routine. This routine divides an array into two halves, one above and one below the "mean". This process is repeated (i.e., the routine is called recursively) until the data is sorted.

The pseudocode for `QuickSort` is

```
QuickSort( int Bottom, int Top ) // Sort the Array from
{                                // Bottom to Top

    int i, j;
    int mean;
```

```
int temp;

if ( Top == ( Bottom + 1 )){ // Do A Sort on Last
                           // Two Elements

    if ( Array[ Bottom ] > Array[ Top ] ) {
        temp = Array[ Bottom ]; // Swap the Two
                                // Elements
        Array[ Bottom ] = Array[ Top ];
        Array[ Top ] = temp;
    }

} else { // Sort > 2 Elements in to 2
        // Halves

    for ( i = Bottom; i < ( Top + 1 ); i++ ) // Get Array
                                             // Total

        mean += Array[ i ];

    mean = mean / ( Top + 1 - Bottom ) // Get Array Mean

    i = Bottom; j = Top;
    while ( i != j ) { // Split the Data into
                      // two halves

        while ( ( Array[ i ] < mean ) && ( i != j ) )
            i++; // Find Array Element Above Mean

        while ( ( Array[ j ] < mean ) && ( i != j ) )
            j++; // Find Array Element Below Mean

        if ( i != j ) { // Swap the Two Value Positions
            temp = Array[ i ]; // - Lower Half is Less than Mean
            Array[ i ] = Array[ j ];
            Array[ j ] = temp; // - Upper Half is >= mean
        }
    } // Finished Splitting the Data

    QuickSort( j, Top ); // Sort the Top Half of the
                        // Data
    if ( i > Bottom ) // Sort the Bottom Half of
                    // Data

        QuickSort( Bottom, i );

} // Finished Sorting
} // End QuickSort
```

The advantage of `QuickSort` is that it is an $O(N \log N)$ sort, which means that the time required for the sort is proportional to the product of the number of elements times the base 2 logarithm of the number of elements. For many small arrays of data to sort, `BubbleSort` is more efficient than `QuickSort`, but as the array size grows, `QuickSort` becomes the preferred sorting method very quickly.

I do not recommend trying to implement `QuickSort` in the PIC microcontroller for a number of reasons. First, `QuickSort` is a recursive algorithm. You will find that the PIC microcontroller's program counter stack will be used up very quickly if `QuickSort` is used. I have implemented `QuickSort` in BASIC without requiring recursive calls, but it does require a lot of memory for storing intermediate array starts, ends, and means. `QuickSort` may be an option for the PIC18 devices, but it is definitely inappropriate for the low-end and mid-range PIC microcontrollers.

Interrupts

I've written a lot about interrupts and interrupt handlers in this book. While they are not terribly hard to create, there are some rules and conventions that you should follow when writing the software handlers:

- 1** Use the standard header information provided in the earlier section of this chapter.
- 2** Keep them as short as possible with interrupt controller hardware reset taking place as early in the interrupt handler as possible.
- 3** Avoid nested interrupts.
- 4** Do not call subroutines from an interrupt handler, or if this is not possible, avoid reentrant subroutines.

When you first start writing interrupt handlers on your own, by following these four rules, you should minimize the problems with the interrupt handler code that will make debugging your application easier.

CONTEXT SAVING

When working with interrupts, the standard interrupt handler saves the `w`, `STATUS`, `PCLATH`, and any other "context" registers that are used by both an interrupt handler and a mainline application. The `FSR` register often fits into this definition and should be saved along with the other three registers as well.

The "standard" assembly-language context saving and restoring code is

```
Int
movwf _w           ; Save "w" Contents
movf STATUS, w
bcf STATUS, RP0    ; Save "STATUS" in Bank 0
movwf _status
movf PCLATH, w
```

```
movwf _pclath
clrf PCLATH           ; Make Sure Execution out of Page 0

:

movf _pclath, w
movwf PCLATH
movf _status, w
movwf STATUS
swapf _w, f
swapf _w, w
retfie
```

The first time you see this code, it will seem somewhat strange and difficult to understand what is happening. The first `movwf w` seems straightforward enough as saving the contents of the `w` register into a temporary register. There is one thing you should beware of, and that is that the bank bits in the `STATUS` register (`RP0` and `RP1`) can be any value. To ensure that the contents of the `w` register can be saved safely, the variable `_w` should be placed either at the same address in each page or in a “common” register that is “shadowed” across all pages. Personally, I prefer the latter method because it means that only one declaration is registered for the variable, whereas the other method requires one declaration per page. Next, the `STATUS` register is placed in the `w` register, and the `STATUS` register’s `RP` bits are loaded with the bank used to store the context registers. In the example code, I used bank 0, but any bank can be used. The important point to remember is that the `RP` bits are set after the `STATUS` register’s contents are saved in the `w` register. Finally, the `PCLATH` register is saved and reset to zero (the interrupt handler starts executing in page 0 of the PIC microcontroller). These three lines can be avoided if the PIC microcontroller you are using has less than one page (2048 instructions for the mid-range device) of PIC microcontroller, and the interrupt handler doesn’t change `PCLATH`. Other context registers (such as the `FSR` register, as mentioned earlier) can be saved using the process of loading them in the `w` register and saving them in a variable.

With the mainline context register values stored, the interrupt handler now can load the registers with any values required as it responds to the interrupt request. Before completing, the requesting `IF` flag will have to be reset.

The process of restoring the registers is the reverse of saving them, except that no instructions are used to set the registers to specific values. The only surprising aspect of the context registration restore is the two `swapf` instructions before the `retfie` instruction. These instructions address the issue of how to load the `w` register without changing the `STATUS` register’s zero flag. If you look at the instruction definitions, you’ll see that the `swapf` instruction does not change any `STATUS` register flags.

Using the `movf` instruction will modify the zero flag but by first swapping the two nibbles in place and then swapping them again as the value is loaded into the `w` register. This operation will load the `w` register with the correct mainline value without changing the `STATUS` register’s zero flag.

This method is quite clever and efficient and avoids having to do something like

```
movf _status, w
movwf STATUS           ; Restore the Status Register
movlw 0
btfss STATUS, Z
movlw 1
movwf _status          ; Save Value According to Zero
movf _w, w
movf _status, f        ; Set Zero According to the Original
retfie                 ; Value
```

This code, which was the method I came up with originally for returning from interrupts, uses the `_status` variable and makes it zero or nonzero based on the zero flag's original state. Once the `w` register is restored, the new `_status` value is compared with zero. This method requires twice the number of instructions as the method using the two `swap f` instructions to restore the `w` register from `_w` without changing the zero flag.

NO CONTEXT SAVING INTERRUPT HANDLERS

Sometimes in the PIC microcontroller the interrupt handler just resets an interrupt active (IF) flag along with a program flag to indicate that the interrupt happened. Another simple interrupt implementation could be a timer that only requires an interrupt handler to increment or decrement a counter. In these cases, there is no reason to save the context information, which reduces the number of active variables and allows the interrupt handler to execute in as few as seven instruction cycles.

Looking through the mid-range PIC microcontroller instruction set, there are eight instructions that are appropriate in this type of interrupt handler because they don't change the `w` or `STATUS` registers (Table 8.6).

TABLE 8.6 THE EIGHT MID-RANGE PIC MICROCONTROLLER INSTRUCTIONS THAT DO NOT AFFECT W OR STATUS REGISTER BITS
bcf
bsf
btfsc
btfss
decfsz
incfsz
nop
swapf

I realize that `goto` and `call` do not change either the `w` register or the `STATUS` register, but they do use `PCLATH`, which may be incorrect depending on whether or not the application is executing outside the first page.

Using these instructions, an interrupt handler that sets a flag when `TMR0` overflows could be

```
int:

    bcf  INTCON, T0IF
    bsf  T0Flag

    retfie
```

and an interrupt handler that increments a counter on `TMR0`'s overflow could be

```
int:

    bcf  INTCON, T0IF
    incfsz TMR0Count, f
    nop

    retfie
```

The two interrupt handlers can be combined to set a flag when `TMR0` has overflowed a set number of times:

```
int:

    bcf  INTCON, T0IF
    bsf  T0Flag
    decfsz TMR0Count, f
    bcf  T0Flag

    retfie
```

In the last example, 9 to 10 instruction cycles are required for the interrupt handler.

These examples assume that register bank 0 will always be the active bank. In my basic program format, during hardware initialization, I execute out of the different banks, but when the application is running, I work at staying within bank 0. If you cannot guarantee that execution will always take place in register bank 0, then you probably will not be able to execute the no context save interrupt handlers shown here.

One style of writing programs is to execute the application functions from entirely within interrupt handlers. After initializing hardware, the mainline just executes an endless loop (`goto $`). When an interrupt occurs, the handler not only handles the interrupt request but also provides all the responses to the interrupt. In this case, saving the context registers is not required because the mainline code is only executed intermittently and does not perform any logical functions.

I would like to discourage this style of application coding because it has the very definite possibility that interrupts will be missed if an interrupt request is received while another is being processed. This is especially true if multiple interrupt sources are used with an application.

Reentrant Subroutines

In many applications, subroutines that are used both by interrupt handlers and mainline code are required. In these cases, the subroutines may be interrupted and called again from the interrupt handler. Subroutines that can support being called multiple times, from the mainline and interrupt handlers, are known as *reentrant*. For most processors, sharing a subroutine between the mainline and interrupt handler code can be carried out quite safely and efficiently. However, in the PIC microcontroller, there can be problems with doing this, and I would recommend that you duplicate the subroutine and make the two copies specified to the interrupt handler and the mainline.

The reason for this recommendation is the PIC microcontroller's lack of a data stack that can be used to store "temporary" or "local" variables. Some high level programming languages may provide this capability (and you can provide it as well in assembly-language programming), but doing this will require more complex subroutines and will take FSR away from mainline use in the code. Often a PIC microcontroller subroutine that supports interrupt handler and mainline access will be larger than two copies of the routine.

If nested interrupts are allowed, care should be taken to avoid calling subroutines that may have been called and were executing by the previous handler when the nested interrupt took place. Not properly handling the subroutines can result in data variables that are overwritten or previous interrupts' data being lost. To avoid these potential problems, simply do not call subroutines from your interrupt handler code, and avoid allowing nested interrupts from executing.

Simulating Logic

One of Microchips early application notes was on how to simulate logic functions using a PIC microcontroller. This was an interesting example of how logic functions could be processed by the PIC microcontroller in very slow-speed application. Included in the application note were details on how the PIC microcontroller could be made to simulate logic functions.

For example, to simulate the 8-bit address, look at the circuit shown in Fig. 8.1, which could be implemented using the hardware shown in Fig. 8.2.

This circuit could be simulated within a PIC microcontroller using the model shown in Fig. 8.3, where the comparison to the `set addr`, which was stored previously in the PIC microcontroller memory, could be implemented in software as

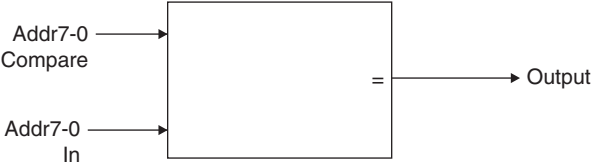


Figure 8.1 Eight-bit address comparator.

```
Loop
movf  PORTB, w      ; Get Current
xorwf SetAddr, w
btfsc STATUS, Z     ; Zero if Match
goto matchHI
nop
bcf   RAO           ; No Match
goto Loop
matchHI
bsf   RAO
goto Loop
```

This sequence of instructions takes eight instruction cycles under a match or mismatch, and herein lies the problem with using the PIC microcontroller for simulating logic; this sequence is relatively very slow. For the preceding above, if the PIC microcontroller were running at 20 mHz, the worst-case switching time you could expect is 1.6 μ s. The best case is 1.2 μ s (if the change is just before the `movf` instruction).

Comparatively, most CMOS and TTL logic would execute this in much less than 100 ns—one full order of magnitude speed increase (with two or three orders of magnitude increase being more likely). This improvement becomes even more profound when more complex operations are required, and the PIC microcontroller code becomes lengthier.

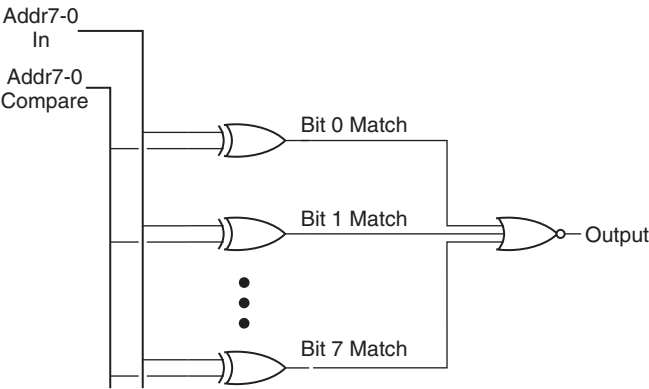


Figure 8.2 Equivalent circuit to Fig. 8.1.

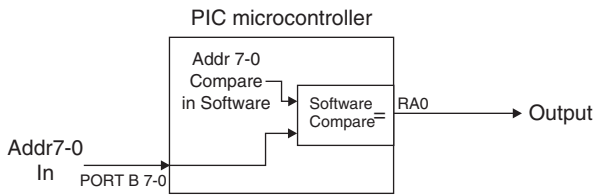


Figure 8.3 Eight-bit address comparator using a PIC microcontroller.

Even a simple case such as devoting a PIC microcontroller to ANDing two inputs together is very slow. For example, if you had two inputs and wanted to implement an AND output, you could use the code

```

Loop
  btfsc Bit1
  btfss Bit2
  goto reset      ; If Bit1 or Bit2 is Low, Turn Off Output
  nop
  bsf  Output
  goto Loop
reset
  bcf  Output
  goto Loop
  
```

This requires seven instruction cycles.

You might want to simplify the code by assuming the output is high before the comparison:

```

Loop
  bsf  Output
  btfsc Bit1
  btfss Bit2
  bcf  Output ; If Either Input is Low, Output is Low
  goto Loop
  
```

which reduces the number of cycles in the loop by 1 but will either have a constant 1 output or an alternating 1 and 0 (three cycles each) if either one of the inputs is low. This is shown in Fig. 8.4.

Even at six cycles a bit, this is really not a contender for competing against any kind of logic in terms of speed, and the PIC microcontroller always will be more expensive in terms of cost. In the final analysis, I never recommend using a PIC microcontroller for a logic replacement. Instead, TTL/CMOS chips should be used.

This is not to say that the PIC microcontroller cannot be used for implementing simple logic functions in software that will avoid the need for external gates. Logic functions can be implemented in software and let you avoid adding logic externally to the PIC microcontroller. Providing the logic functions within the PIC microcontroller will

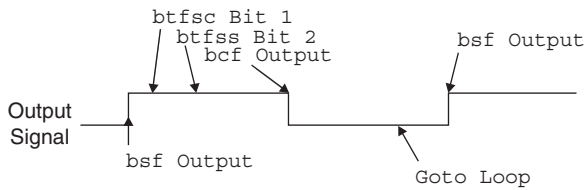


Figure 8.4 Two input AND gate software operation analysis.

reduce the number of input-output (I/O) pins required for the PIC microcontroller and will reduce the cost of the application.

I realize that there will be cases where you will consider the PIC microcontroller because one gate is required and power, space, and cost requirements do not allow an additional chip to be put into the application. In such cases, I recommend that you look at different logic analogs that are available using simple discrete circuits—even a few resistors and diodes will provide meaningful logic functions for CMOS circuitry.

Event-Driven Programming

The PIC microcontroller is particularly well suited to event-driven programming applications owing to how its interrupt handler code works. Many of the experiments and applications presented in this book use event-driven programming to respond to numerous inputs without affecting the processing of other inputs or tasks within the application.

The typical high level format for event-driven programming is

```
main()                                // Event Driven Updated Initial
                                     // Application
{

int i = 0;

TimerDelay = 1sec;

interrupts = TimerHandler | ButtonHandler;

if (Button == Up)    // Load in the Initial Button State
    LED = Off;
else
    LED = On;

while(1 == 1);

} // end main
```

```

interrupt TimerHandler()// Display "i" and Increment

    TimerInterrupt = Reset;

    output( i & 0x00F);

    i = i + 1;

} // End TimerHandler

interrupt ButtonUp( )
{

    ButtonInterrupt = Reset;

    LED = Off;

} // end ButtonUp

interrupt ButtonDown( )
{

    button interrupt = reset

    LED = on;

} // end ButtonDown

```

This style of programming can be copied easily into the PIC microcontroller by testing the F (interrupt request flag) bits and responding to the first one that is set. The interrupt handler, for the preceding application, would look like this:

```

Org          4
Int
    movwf _w
    movf  STATUS, w
    movwf _status

    btfsc INTCON, T0IF
    goto INTTMRO
    btfsc INTCON, INTF
    goto BUTTONINT

INTERRUPT REQUEST          ; Clear all other Interrupt Requests
    clrf  INTCON
    goto INTEND

TMROINT                    ; Respond to the TMRO Interrupt

```

```
    bcf  INTCON, TOIF

    movf  i, w           ; Output (i and 0x00F)
    xorlw 0x00F
    movwf OUTP

    incf  i, f

    goto  INTEND

BUTTON INT               ; Respond to the Button Pressed

    bcf  INTCON, INTF

    btfss BUTTON, UP
    goto BUTTONDOWN

BUTTONUP                 ; Button Released, LED Off

    bsf   STATUS, RPO    ; Change Button Interrupt Request Direction
    bcf   OPTION_REG & 0x080, INTEDG
    bcf   STATUS, RPO

    bsf   LED            ; LED Off

    goto  INTEND

BUTTON DOWN              ; Button Pressed, LED On

    bsf   STATUS, RPO    ; Change Button Interrupt Request Direction
    bcf   OPTION_REG & 0x080, INTEDG
    bcf   STATUS, RPO

    bcf   LED            ; LED On

    goto  INTEND

INTEND                   ; Interrupt Handler Finished – Return to
                        ; Mainline (the “while (1 == 1)” Statement)

    movf  _status, w
    movwf STATUS
    swapf _w, f
    swapf _w, w
    retfie
```

This code should be a straightforward conversion of the high level code, except for the button. Before jumping to button up or button down, the polarity has to be determined. When the polarity is determined, then the interrupt edge bit is set to interrupt

when the button input changes state. This state determination really is used to help differentiate which part of the event handler should execute.

Note that the event to be responded to can be given a priority, with the first check being the interrupt source responded to before any of the others. In the example code above, TMR0 is responded to first, even if a button interrupt has been received. The order (and priority) of event checks and responses can be selected in such a way as to ensure that no events are missed and the most important ones are responded to first.

State Machine Programming

There are two different methods of implementing a state machine in PIC microcontroller assembly language. The first method is to use a table such as

```
movf  State, w          ; Jump to the Appropriate State Table Entry
addwf PCL, f
goto  State0            ; Routine for State == 0
goto  State1            ; Routine for State == 1
goto  State2            ; Routine for State == 2
:
```

This method is quite efficient for implementing a state machine because with implementing any table, execution takes a constant amount of time regardless of the value of `State`. The only requirement for this method is to have the state values as linear numbers starting with zero.

The second method is to repeatedly test the `State` variable for specific values and then executing the appropriate code. The “typical” PIC microcontroller code for this is

```
movf  State, w          ; Load the “State” Variable
xorlw 0                 ; Test for State == 0
btfss STATUS, Z
    goto NotState0
    :                   ; Routine for State == 0
    goto StateEnd
NotState0:
    xorlw 0 ^ 1         ; Test for State == 1
    btfss STATUS, Z
    goto NotState1
    :                   ; Routine for State == 1
    goto StateEnd
NotState1:
    xorlw 1 ^ 2         ; Test for State == 2
    btfss STATUS, Z
    goto NotState2
    :                   ; Routine for State == 2
    goto StateEnd
```

```
NotState2:
:
StateEnd:                ; Finished with State Machine Execution
```

In this code, `State` never has to be reloaded because the previous value XORed with the contents of `State` is XORed again in the next instruction when it is tested for a different value. The advantages of this method are that nonsequential values of `State` can be implemented. The `State` variable also can be done away with if this method is used, and the `w` register can be used for the new state value. The obvious drawbacks to this method over the previous one are that many more statements are required and the time to execute statements is different for each state's `Routine`.

I use this code for implementing character data processing to state machines when specific values are required. This method is superior to the previous method when a potentially large number of inputs could be received and only a few will be processed.

Either method can be useful for implementing complex applications as state machines in low-end PIC microcontrollers, where single states may have multiple sources and you do not want to use the limited stack in a subroutine.

Porting Code Between PIC Microcontroller Device Architectures

Historically, I design my PIC microcontroller assembly-language code in terms of the mid-range architecture; if the application is destined for another PIC microcontroller family, I then change the methods used for the code by the rules I present in the next two sections. When it comes right down to it, the changes are quite subtle, and application code can be created that can be ported between PIC microcontroller architectures very easily. This was illustrated in Chap. 7 when I talked about the different types of instruction elements and showed how they had analogies in each of the three architectures. Designing your application on the mid-range PIC microcontroller devices first is not a bad approach because these devices have many of the same capabilities as low-end and PIC18 architectures, and the availability of MPLAB ICD-enabled parts will allow you to cost-effectively develop and debug your applications before porting them to the final device.

PORTING MID-RANGE APPLICATIONS TO THE LOW END

All the low-end PIC microcontrollers' instructions are available in the mid-range device and work exactly the same way. The lack of some instructions requires some work-arounds, but I find that by writing mid-range code with the low-end device in mind, the code can be ported remarkably easily with few changes. There are seven primary differences to be aware of between the mid-range and low-end PIC microcontroller instruction sets. Some of these differences are a result of the 12-bit instruction word of the low-end devices, and others are due to the low-end device's architecture.

These seven differences are

- 1 Smaller instruction page size
- 2 Four fewer instructions
- 3 Subroutine `call` instruction differences
- 4 Reduced program counter stack
- 5 Subroutine `return` instruction differences
- 6 No interrupt capability
- 7 No register “bank select bits”

The low-end PIC microcontroller’s page size is 512 instructions and requires 9 address bits. This will seem like a lot smaller than the mid-range device, which has a page size of 2,048 instructions (11 address bits), and will seem to be difficult to create applications for. With this restriction, along with others discussed in this section, I tend to look at the low-end devices as being better suited for applications that require no more than 512 instructions with limited text/data output.

Five hundred and twelve instructions may not seem like a lot. To put this in perspective, I always thought that the 512 bytes used for booting in the PC was frightfully restrictive. When the PC boots, a single 512-byte sector is loaded into memory from disk, and this code is used to load and start up the operating system in the PC. Some years ago I wrote my own code for an operating systemless boot from a diskette in a PC and found that I only required 107 bytes. Knowing that each 8086 instruction averages 3 bytes in size, you can see that a lot can be done in the 512 bytes of a low-end PIC microcontroller page.

The mid-range instructions that are not available in the low-end devices are listed in Table 8.7.

The lack of `addlw` and `sublw` can make some applications awkward where these instructions are made on immediate values stored in the `w` register. To counter the lack of the `addlw` and `sublw` instructions, I first try to arrange the statements to avoid the need for these instructions. For example, instead of

```
movf  A, w
addlw 3
andlw 0x3F
movf  PORTB
```

TABLE 8.7 THE MID-RANGE PIC MICROCONTROLLER INSTRUCTIONS NOT AVAILABLE IN LOW-END DEVICES

<code>addlw</code>
<code>sublw</code>
<code>return</code>
<code>retfie</code>

I would write the code function as

```
movlw 3
addlw A, w
andlw 0x3F
movwf PORTB
```

which can execute on both mid-range and low-end PIC microcontroller processors.

Things can get a bit tricky when you have specific requirements for the code. When I wrote the preceding example for the second edition of this book, I wrote it as

```
movf A, w
andlw 0x3F
addlw 3
movf PORTB
```

which is a subtle change to the preceding and provides a different function—the sum of the least significant 6 bits of `A` have 3 added to them, and the sum placed in `PORTB`. In this situation, it seems like there is a need for setting bit 6 of `PORTB` if the least significant 6 bits of `A` are `0x3D` or greater. To implement this as efficiently in the low-end PIC microcontroller architecture is not possible. After some deliberation, the best conversion that I could come up with for the four mid-range instructions is the low-end six-instruction snippet that requires a temporary value:

```
movf A, w
andlw 0x3F
movwf temp
movlw 3
addwf temp, w
movf PORTB
```

The `temp` register contains the intermediate value of the calculation to avoid storing the temporary value in the destination (`PORTB`), which could affect the operation of the application. The intermediate value is saved from the source and retrieved for processing and saved in the destination. As a rule of thumb, you must never use a hardware register as a temporary register because it can cause you a lot of problems in your application (and ones that are particularly difficult to find and fix).

This example illustrates an important point in porting applications (or even maintaining them): You have to understand exactly the requirements of the code. In these examples, the original author may have been simply sloppy (since the result is the same in the least significant 6 bits regardless of the method used), or there was some definite need to set bit 6 if the original value in the least significant bits of `A` were `0x3D` or greater at the same time as the sum of these bits and 3 were stored in `PORTB`. Making an invalid assumption on the purpose of the code that could result in an error in the operation of the application will be extremely difficult to find and fix.

Subroutine calls can be quite awkward in low-end devices because the label must be located in the first 256 bytes of the PIC microcontroller's instruction page. This is due

to how the `goto` and `call` instructions are defined, a result of how bits are allocated in low-end PIC microcontrollers.

If you look at the instruction bit patterns for the two instructions, i.e.,

`goto`—0b0 101k kkkk kkkk

`call`—0b0 1001 kkkk kkkk

you'll see that `goto` has 9 bits available to the jump to the address, whereas `call` has only 8. The 9 address bits in the `goto` instruction mean that the `goto` address can be anywhere in the 512 instructions of a page. The 8 bits of the `call` instruction mean that only 256 addresses in the instruction page can be accessed. This restriction can be a problem in applications if it is not planned for.

To counter this problem, the address that your subroutine calls can be a `goto`, in the first 256 instructions of a page, pointing to the actual subroutine code. This looks like

```
call  sub1

:

sub1:          ; "sub1" label/goto redirecting execution
goto subroutinel ; to the second 256 instructions in page
sub2:
goto subroutine2
```

Table operations have a similar problem with the two instructions `addw PCL, f` and `movwf PCL` having a similar restriction to `call` and also can be located only in the first 256 instructions of a page. This is due to the inability of accessing bit 8 of the address, and a zero is used in its place. PA0 in the STATUS register affects address bit 9, and PCL affects address bits 0 through 7. This restriction means that no tables can be longer than 256 instructions in low-end PIC microcontrollers.

There are two other restrictions you also should be aware of for low-end devices. The first is the availability of only two positions on the program counter stack. This means that there can only be one nesting level in low-end applications. Subroutines that have been called by other subroutines cannot call subroutines themselves. This is not a particularly onerous restriction because the number of nesting levels in all PIC microcontrollers is quite limited. When you are beginning to develop applications for the PIC microcontroller, I don't recommend using many nested subroutines because of the possibility of interrupts causing the program counter to "overflow" the stack and return addresses being lost in some cases.

The second restriction is the availability of only a `retlw` (return from subroutine after loading the w register with a constant) instruction for returning from subroutines. There is no `return` statement in low-end PIC microcontrollers.

In early versions of MPASM, when a `return` instruction was encountered in a low-end application, it would convert the instruction into

```
retlw 0          ; Return with "0" in w.
```

without notification of the conversion. With current versions of MPASM, you will get the message

```
Warning[227] <program filename>.ASM <line number> : Substituting RETLW  
0 for RETURN pseudo-op
```

In an earlier version of MPLAB IDE, this message was not produced, and the assembler automatically inserted the `retlw 0` instruction. This caused me several hours of confusion one night trying to figure out why code that ran fine in a mid-range PIC microcontroller wouldn't work in a low-end device. To avoid this problem, you may wish to always return values in a subroutine in a common variable rather than in the `w` register or use the STATUS flags to return pass/fail or other binary information.

No interrupts in low-end PIC microcontrollers changes the way inputs and hardware events are monitored. Instead of waiting for an event, your code will have to poll for the event. If a signal waited for is very short, you may miss it. In Appendix G, I have a small snippet of code that will set up TMR0 of PIC microcontrollers to overflow when a signal is received. This will allow you to use the TMR0 hardware in the PIC microcontroller to indicate if a pulse has been received or a line has changed state without having to continually poll it.

The last difference, the lack of bank select bits, will be the biggest issue you have in porting mid-range applications to low-end devices. The most obvious problem will be the need to convert all the bank switch writes to the OPTION_REG and the TRIS registers to the single-instruction equivalents, but there is a larger problem you should be aware of.

In low-end devices, a register bank uses only 5 bits for addressing (in high-end devices, 7 address bits for each address are used). This only gives a maximum of 32 registers per bank. There can be up to four banks, each with shared INDF, TMR0, PCL, STATUS, FSR, PORTA, PORTB, PORTC, and file registers. The top 16 registers are usually left as being unique to each bank. This layout leaves a maximum of 25 bytes for an array, with 16 being the maximum in any bank other than bank 0.

The low-end register organization looks like Fig. 8.5, and in low-end PIC microcontroller devices, even though a total of 7 address bits are used when all four banks are taken into account, only a maximum of 73 unique addresses are implemented, with 48 of them accessible only by the FSR. In most mid-range devices, the registers in each bank are unique to that bank, although there are "shadowed" registers to allow data movement between banks. In low-end devices, the first 16 registers of each bank are shared (or "shadowed") between the maximum four banks.

Another issue to be aware of with low-end devices and their restricted register banking operation is that the FSR index register can never be equal to 0. Because of this, make sure that you never write any application code such as

```
movlw 10  
movwf FSR  
Loop  
decfsz FSR, f  
goto Loop
```

Bank 0	Bank 1	Bank 2	Bank 3	
Addr - Reg	Addr - Reg	Addr - Reg	Addr - Reg	
00 - INDF 01 - TMR0 02 - PCL 03 - STATUS 04 - FSR 05 - PORTA* 06 - PORTB 07 - PORTC 08-0F Shared File Regs	20 - INDF 21 - TMR0 22 - PCL 23 - STATUS 24 - FSR 25 - PORTA* 26 - PORTB 27 - PORTC 28-2F Shared File Regs	40 - INDF 41 - TMR0 42 - PCL 43 - STATUS 44 - FSR 45 - PORTA* 46 - PORTB 47 - PORTC 28-2F Shared File Regs	60 - INDF 61 - TMR0 62 - PCL 63 - STATUS 64 - FSR 65 - PORTA* 66 - PORTB 67 - PORTC 68-8F Shared File Regs	Shared Registers
10-1F Bank 0 File Regs	30-3F Bank 1 File Regs	50-4F Bank 2 File Regs	70-7F Bank 3 File Regs	Bank Unique Registers

* - "OSCCAL" may take place of "PORTA" in PICMicros with Internal Oscillators

OPTION - Accessed via "option" Instruction

TRIS# - Accessed via "TRIS PORT#" Instruction

Figure 8.5 The low-end PIC microcontroller register space showing limited contiguous file register groups.

This loop will never end because FSR can never equal 0.

The FSR can be a useful temporary register in mid-range devices because you don't have to define it, and if indexed addressing is not used in the application, it really becomes a free file register for your use. Since the FSR can never be 0 in low-end devices, I recommend that it is never used except in its primary function as an array index register.

PORTING TO THE PIC18

There really aren't that many differences between the three major PIC microcontroller architectures. In fact, using the rules outlined in the preceding section will help you to write code that can be passed back and forth between any of the different PIC microcontroller devices quite simply and with very little device-specific modification.

The significant differences in the PIC18 devices from the lower-end architectures are

- 1 Different PCLATH and goto/call instruction operation
- 2 Sixteen-bit instruction words with the ability to address them directly
- 3 Compare and branch instructions
- 4 Different register bank organization
- 5 An 8-bit hardware multiplier
- 6 Additional arithmetic and bitwise operation instructions

Other than these differences, application code written for low-end and mid-range PIC microcontrollers should execute without any problems in higher-end devices. There is

a caveat here: In early PIC18 device datasheets there is the comment that they are “source code compatible” with mid-range devices. While there is some truth to this comment because many of the arithmetic and bit instructions work identically, data addressing with the execution change instructions, as well as operation of the program counter and index registers, is decidedly different, and you cannot rebuild mid-range PIC microcontroller source code for a PIC18 without considerable modification.

When you develop your first applications for PIC18 devices, I recommend that you do not access the PCLATH and PCLATU registers for your first applications. This should not be a significant hardship for you because the page size is quite large, and the PIC18 can jump to any address in program memory. The only time that you should be modifying the PCLATH and PCLATU registers is when you are executing computed (table) jumps.

The addition of the “compare a register to the w register and skip the next instruction on a condition” is a unique capability to the PIC18. If these instructions are going to be used, then I recommend that their functions be implemented in low-end and mid-range PIC microcontrollers using macros. For example, the `cpfseq` instruction, which skips the next instruction if the contents of the register are equal to the w register, could be implemented in low-end and mid-range PIC microcontrollers as

```
cpfseq macro Register
    subwf Register, w           ; Subtract "w" from Register
    btfss STATUS, Z
endm
```

“Compare and skip if less than” (`cpfslt`) also can be implemented easily with the macro code

```
cpfslt macro Register
    subwf Register, w           ; Subtract "w" from Register
    btfsc STATUS, C
endm
```

“Compare and skip if greater” is a bit more complex and requires the use of a “double” bit skip on condition because the carry flag will be set if the contents of `Register` are equal to or greater than the contents of “w”:

```
cpfsgt macro Register
    subwf Register, w           ; Subtract "w" from Register
    btfss STATUS, Z             ; Don't Skip if Zero Set
    btfss STATUS, C             ; Skip if Zero Reset and Carry Set
endm
```

Note that these instructions change the contents of the w register and the STATUS register flags. For most applications, this should not be an issue, but if previous flag values are used in the application, then when porting to low-end and mid-range PIC microcontrollers, you might want to save the STATUS register before executing the macros.

The PIC18's "branch on condition" is somewhat unusual because it uses a relative offset rather than a page address, as do all other PIC microcontroller execution change instructions. This should not be an issue when creating macros for porting the function to low-end and mid-range PIC microcontrollers unless a page boundary is going to be crossed. For this reason, I recommend making sure that the appropriate PAX bits in low-end devices and the appropriate PCLATH bits in mid-range devices are changed before the instruction.

The PIC18 BC ("branch if carry") instruction can be simulated in low-end PIC microcontrollers using the macro

```
bc macro Address
    local EndAddress
    if ((Address & 0x0400) != 0)          ; Set up Page Bits for Jump
        bsf STATUS, PA1
    else
        bcf STATUS, PA1
    endif
    if ((Address & 0x0200) != 0)
        bsf STATUS, PA0
    else
        bcf STATUS, PA0
    endif
    btfsc STATUS, C
    goto Address & 0x01FF                ; If Carry Set, Jump
    if ((EndAddress & 0x0400) != 0)      ; Restore the Page Bits
        bsf STATUS, PA1
    else
        bcf STATUS, PA1
    endif
    if ((EndAddress & 0x0200) != 0)
        bsf STATUS, PA0
    else
        bcf STATUS, PA0
    endif
EndAddress                             ; End Address for the macro
endm
```

The use of the EndAddress is to ensure that there is no opportunity for the situation where the page boundary occurs within the macro to cause a problem. The mid-range device's bc macro is similar except that the PCLATH bits are changed and then reset. It is surprising that these macros, which simulate instructions in the PIC18, actually have a greater address range capability in low-end and mid-range devices. The simulated macros can jump anywhere in program memory, whereas the actual PIC18 instructions can only jump -128 to +127 addresses from the next instruction.

Along with the PIC18 bc instruction, the bnc, bnz, bra, and bz instructions all can be implemented in low-end and mid-range PIC microcontrollers. The bov, bnov, bn, and bnn instructions cannot be simulated using a macro because the OV and N flags are not implemented in low-end and mid-range PIC microcontrollers.

Porting variable access code, despite the different file register bank organization, is not an issue in really for the PIC18. For analogous translation of file register access application code, the BSR register can be updated with new addresses, but when I move applications between architectures, I simply keep my variables starting at address 0x020 and access them directly rather than keeping track of things with BSR.

The PIC18 hardware I/O register organization is quite a bit different from how the hardware I/O registers are implemented in low-end and mid-range devices. When porting applications, changes will have to be made to the source code based on the destination device. This is actually the most significant amount of code development that you will have to do for porting applications.

The 8-bit hardware multiplier is an obvious omission when going from PIC18 devices to low-end and mid-range PIC microcontrollers. This hardware multiplier capability will have to be simulated in software using the code presented elsewhere in this book.

There are a number of enhanced instructions available to PIC18 microcontrollers that can cause some problems when porting devices. If you are expecting to port applications between low-end and mid-range PIC microcontrollers, I would suggest that working with the low-end PIC microcontroller's arithmetic instructions (`addwf` and `subwf`) be the limit of the instructions used. The other instructions, while they will really enhance your application code, will be very difficult to simulate in lower-end PIC microcontroller architectures.

Limiting the use of some of the most useful features in the PIC18 really begs the question of whether or not porting the application to a lower-end device is appropriate. The PIC18 has many new instructions and capabilities that are not available in low-end and mid-range devices, and over time, I expect to see a great number of new PIC18 part numbers that can replace mid-range PIC microcontroller functions. When this becomes the case, the reason for considering porting applications becomes much less important. For this reason, I consider this section to be more of a guide for porting snippets of code between PIC microcontroller architectures rather than full applications. Porting full applications will require a fair amount of work and will take quite a bit of time for you to make sure that all the necessary changes between the architectures are done correctly and that the application will work with all the differences between the source and destination PIC microcontrollers properly compensated for.

Optimizing PIC Microcontroller Applications

As you start developing you own PIC microcontroller applications, you are going to discover that the device you choose does not have enough program memory, file registers, or I/O pins for your needs. One solution may be to try another device or part number that has more of these resources or built-in hardware. The problem with this solution is that it usually costs more, and some of the basic problems with the application are not addressed. Instead of jumping to a new chip, there are some strategies that you can work

through to allow you to use the device that you have already chosen. This optimization of the application is not terribly hard to do, and I find it fun to see how much more function I can cram into a PIC microcontroller. In this section I want to discuss some strategies that you can use.

One of the most frustrating things that you can experience in an application is running out of file registers. None of the PIC microcontroller part numbers have a lot to begin with, and you probably will find that you will run out of them when you plan complex applications in small (low-end) devices. There are a few things that you can look at to try to alleviate the need for file registers.

The first and most obvious action to take is to look for file registers that are used as flags. Flags should be implemented as individual bits. This can reduce the requirements from eight registers down to one. One of the reasons why people use a file register for a flag is to avoid having to remember the bit number for a specific flag. This concern can be eliminated by using the `#DEFINE` directive to specify individual bits.

For example, 8 bits of a “flags” register could be declared like

```
#DEFINE RUNFLAG  FLAGS, 0
#DEFINE STOPFLAG  FLAGS, 1
#DEFINE REQUESTFLAG  FLAGS, 2
```

and to set or reset a flag, the `bcf` or `bsf` instruction is used like

```
bsf  RUNFLAG
```

The `#DEFINE` actually puts two parameters to the label so that when `RUNFLAG` and the other `#DEFINE` labels are encountered, the instruction is loaded with the register and the bit number instead of having to put both parameters in the instruction.

This optimization is interesting because not only does it reduce register requirements, but it also reduces the number of instructions required to test the state of a flag to one from the multiple instructions used in other processors and makes the code easier to read. For example, to jump if `STOPFLAG` is set, the instructions

```
movf  FLAGS, w  ; Clear everything but the stop bit
andlw 1 << 1
btfss STATUS, Z      ; Jump to label if the bit is set
goto LABEL
```

could be used and typically are what would be used in other processors, but in the PIC microcontroller, using the define for a bit in a register, the same function can be implemented in

```
btfss STOPFLAG
goto Label
```

which is much easier to read and understand than the previous code and requires fewer instructions.

Some people will try to use hardware registers for temporary storage of data, and I would like to discourage this as much as possible. Writing random values to a register

can result in unexpected and unwanted hardware operations. The only exception to this rule is using the FSR as a temporary register.

For reducing code space, there are a number of things to look at that will not require large changes to the application. First off, check to see that all arrays are in bank 1 rather than in bank 0. The FSR register can access data in either bank, and code space will be saved if the single-byte variables are stored in the bank out of which the application mostly executes.

If you have to switch banks in the application and access registers, use the multiple “shared” file register addresses available in the PIC microcontroller instead of the *w* register. This will allow the sharing of multiple bytes between the banks without having to toggle the RP0 or RP1 bits. Spending some time thinking about how variables are placed in the application will result in huge code savings. Ideally, you should try for never changing the bank register when accessing file registers.

Inefficient array and stack accesses can be particularly wasteful in terms of instructions. When you are implementing arrays, look for built-in features of the PIC microcontroller architecture that will help you to simplify the accesses. One trick I like to use is to put array elements starting at a specific offset instead of relying on CBLOCK to allocate the address for me. For example, if I were to put a 16-byte array at 0x040, to access an element, all I would have to do is load the index to the element and set bit 6 to create a correct address. This avoids the complications of adding the starting offset to the index, which may not seem like a major inconvenience, but it can add to the number of instructions required.

This trick becomes very useful when working with circular buffers. In the preceding example, this 16-byte array is located in file register addresses 0x040 to 0x04F. To increment to the next address within the buffer, only the two instructions

```
incf ArrayIndex, f
bcf  ArrayIndex, 4
```

are required. These two instructions will increment the address and keep it within the correct range (which has bit 4 of its address always reset).

If an arbitrary starting point for the circular buffer is used, then the required code becomes

```
incf  ArrayIndex, w
xorlw ArrayEnd +1
btfsc STATUS, Z
movlw ArrayStart ^ (ArrayEnd +1)
xorlw ArrayEnd +1
movwf ArrayIndex
```

These six instructions test for the array pointer to be past the end of the array after incrementing it and reset to the start of the array, if required. This code also takes up six instruction cycles and modifies the *w* register, which may require saving and loading the value in it, which the two-instruction solution does not.

Jumping between pages can eat up a lot of instructions. To avoid this, try to keep functional blocks of code together on one page as much as possible. You may find that if

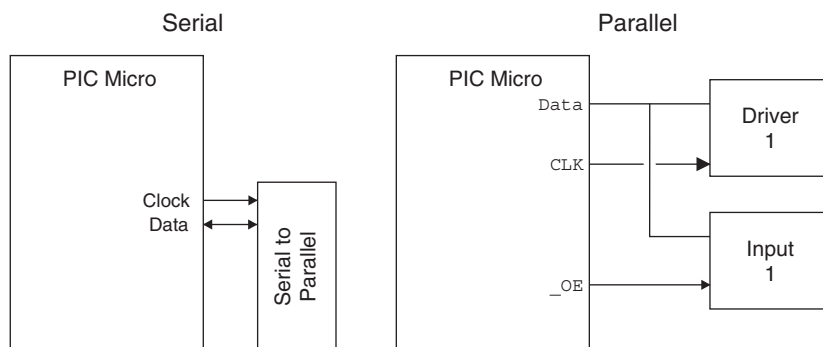


Figure 8.6 A synchronous serial bus with a serial-to-parallel interface chip or a simulated parallel bus can be used to increase the number of I/O pins available to the PIC microcontroller.

you copy subroutines that cross over page boundaries into one consistent page, the actual program memory requirements will decrease.

For optimizing file register and program memory usage, the best suggestion I can give you is to try to come up with as many possible solutions as you can. This also includes looking at other people's code for examples. Chances are that you will be able to reduce the requirements above and beyond the immediate needs of the application.

Not having enough I/O pins can be a particularly troubling problem, but additional pins can be added to the circuit either via a synchronous serial bus and using serial-to-parallel interface chips or by simulating a parallel bus, as shown in Fig. 8.6. The obvious drawbacks to using the synchronous serial bus is the time required for shifting bits and possible "bit ripple" as the bits get shifted in or out. This bit ripple can be avoided by placing a latch between the shift register and the I/O pins. The latch output is updated after the data has been shifted from the PIC microcontroller.

To shift data out, the following code could be used:

```
bcf  DataBit
btfsc SourceRegister, 0
bsf  DataBit
bsf  ClockBit
bcf  ClockBit
```

These five instructions for shifting out a bit can be reduced to four instructions by keeping a file register loaded with a shifted value and placing the data bit in bit 0 or bit 7 of the I/O port. The code for a data pin at RBO, with a clock, also in port B (and rising edge active), to shift out a bit first has the port B value saved in shifted format and the clock bit low:

```
rrf  PORTB, w
andwf 0x0FF ^ (1 << (clock -1))
movwf PORTBSave
```

When the least significant bit of the data is to be shifted out, the four instructions

```
rrf SourceRegister, w
rlf PORTBsave, w
movwf PORTB
bsf ClockBit
```

could be used. These four instructions can be put into a loop to shift out a byte:

```
movlw 8
movwf Count
Loop:
  rrf SourceRegister, f
  rlf PORTBsave, w
  movwf PORTB
  bsf ClockBit
  decfsz Count, f
  goto Loop
```

This code requires 16 instruction cycles less than if the original five instructions were used in Loop:

```
movlw 8
movwf Count
Loop:
  bcf DataBit
  btfsc SourceRegister, 0
  bsf DataBit
  bsf ClockBit
  bcf ClockBit
  rrf SourceRegister, f
  decfsz Count, f
  goto Loop
```

Creating a parallel bus can be challenging to wire and may require a more substantial change to your application than the synchronous serial interface. Despite these drawbacks, data can be passed quickly and does not have the “ripple” of the serial method. Note that both these methods require additional chips that can drive the parts cost up to the point where a PIC microcontroller with more pins is more economical.

The last issue for optimization is often the most difficult to overcome, and this is meeting minimum timing specifications. Often the only solution to this problem is to use a PIC microcontroller (or another microcontroller) with built-in hardware that provides the basic function of running the PIC microcontroller at a faster speed. This is not to say that the problem is insolvable; with a bit of work, you can work through your code to find solutions to the problem. Remember to look at as many different solutions as possible, and remember that there can be some advantages to rearranging the hardware. As was shown in the data-shifting example, a substantial improvement in instruction cycles resulted in a nine-cycle loop for a 22 percent speed improvement of the application

function. Always remember that you should be treating all issues and problems the same way in which you would if you had a failure in some hardware or software; approach the problem using the techniques discussed in Chap. 14.

A Baker's Dozen Rules to Follow That Will Help to Avoid Application Software Problems

In PIC microcontroller programming, I find that there are a number of rules to always apply that will prevent the opportunity for basic problems later. Here are 13 rules that I always follow when I develop a PIC microcontroller application in assembly language that help to keep the application from having problems that will be difficult to fix later:

- 1** Always initialize your variables. I have been caught more than a few times reading the contents of a variable before I have written to it. In the PIC microcontroller, file registers are not initialized to any specific value—they can be any value from 0x000 to 0x0FF. If you don't know what to initialize them to, use 0. This matches the initial values used by the MPLAB simulator to at least guarantee that they will work the same way in the application as they do in the simulator.
- 2** Indent conditionally executing code after a `skip` instruction. This is naturally taught for high level languages, but it does have its place in PIC microcontroller assembly language to make conditionally executing instructions stand out visually.
- 3** Let the compiler/assembler do the calculations for you. This will help to make your code more portable to other applications and save you the hassle of working with a calculator trying to figure out what the values should be (and potentially making a mistake).
- 4** Use Microchip's register definition files without modification in all your applications. I can't tell you how many times I've helped people with broken applications where the only problem was that they copied in a register or bit address incorrectly. Microchip has spent a lot of time developing the include files that are shipped with MPLAB and making sure that they are correct. There's no reason for you to duplicate this effort. In addition, don't change register/bit labels or use different ones. Even if you are more familiar with different terms, don't use them. By changing the labels to what you are comfortable with, you are making the code more difficult for others to follow and increasing the opportunity for errors to be introduced into your application's source code.
- 5** Keep your code as simple as possible. When the PIC microcontroller application is working, everybody will be impressed with how it works, not the complexity or cleverness of your code.
- 6** Develop your application in terms of functional blocks and interfaces. Instead of creating one massive application, develop it as a series of "steps," each of which is simulated or tested on hardware before proceeding and integrating them together.

- 7** Establish a plan to test and confirm that your code is correct. Test your code each step of the way, and do not move on to the next step until you are 100 percent satisfied with the performance of the code up to that point.
- 8** Use CBLOCK or other utilities to allocate variables instead of defining them manually. Using a variable define utility will avoid problems later when variables have to be added or deleted, at which time you will have to manually work through all the potentially affected addresses. This also goes for goto and call instruction destinations. Let the assembler generate the absolute addresses unless a specific address is absolutely required.
- 9** Avoid changing the register bank unless it is absolutely necessary. Ideally, an application should be designed so that all the bank 1 registers and hardware are initialized after reset and then executes in bank 0 for the rest of the application. Going along with this, design your application so that single-byte/word variables are in bank 0 and array variables are in bank 1 (which can be accessed by the FSR register without changing the RP0 bit in the STATUS register). This will help to avoid having to keep straight what bank the application is currently running in.
- 10** Don't allow code to go over page boundaries except when calling subroutines. If your mid-range PIC microcontroller code is larger than 2,048 instructions, place your subroutines in the upper page. Code that is allowed to "drift" over page boundaries can have problems with the correct PCLATH register contents.
- 11** Use the `_ _CONFIG` statement always in your source code, and use a programmer that programs the configuration information automatically instead of one that requires manual intervention. When developing your application, keep the watchdog timer (WDT) disabled, and only enable it in the `_ _CONFIG` statement when all other functions have been debugged.
- 12** Simulate as much of your application as is possible. The time used to develop a stimulus file and single stepping through it will be saved several times over in the time needed to debug the application if simulation has not been carried out.
- 13** Try to keep the amount of nested subroutine calling in your application to a minimum. All the PIC microcontroller families have limited stacks and are not designed for recursive functions in applications. Always make sure that your maximum calling "depth" is less than 2 for low-end devices, 8 for mid-range devices, and 31 for the PIC18 PIC microcontrollers. Note that interrupt handlers use the same stack, and the maximum depth of their execution must be summed with the maximum depth of the mainline to avoid any potential problems in the application's execution.

BASIC OPERATING FEATURES

To get a good idea of what a remarkable device the PIC[®] microcontroller is, take a look at Fig. 9.1. This tiny 8-pin chip actually contains 11 of the basic hardware features that are built in to all PIC microcontrollers—along with this, the processor portion runs at 1 million instructions per second (MIPS), about four times faster than the Apple II, and the chip contains memory for approximately the same number of instructions. The common features in the different PIC MCU part numbers (and their different families) allow you to select the part number that is best suited for your application while still retaining commonalities that will allow you to switch between parts with a minimum of problems. In this chapter I will introduce you to the basic hardware features that are common to all the PIC microcontrollers, along with some guidelines for using them, whereas in later chapters I will discuss other features that are optionally available in different PIC MCU part numbers.

When bringing up a new PIC microcontroller application or debugging one that is not working, I have a mantra that I have found works virtually 100 percent of the time: “Power, clocks, reset.” The first thing you should always check is the power going into the PIC microcontroller. There could be a chance that a battery has died, a wire connection has broken, or you’ve forgotten to plug the ac/dc adapter into the wall. If you have an external oscillator, you should probe both sides of the crystal or the resonator to see if it is running. Finally, checking over the reset circuitry will ensure that the PIC microcontroller *can* execute. When I first came up with this list of items to check, they were all external to the chip, but with enhancements in the operating options of the PIC microcontroller families, the clocks and resets now can be totally internal to the device, and you will have to ensure that the configuration fuses are specified properly and the PIC microcontroller is programmed properly. This mantra has been institutionalized at Logitech; the engineers here know that there is no point in asking for help unless power, clocks, and reset have been checked and appear to be working correctly.

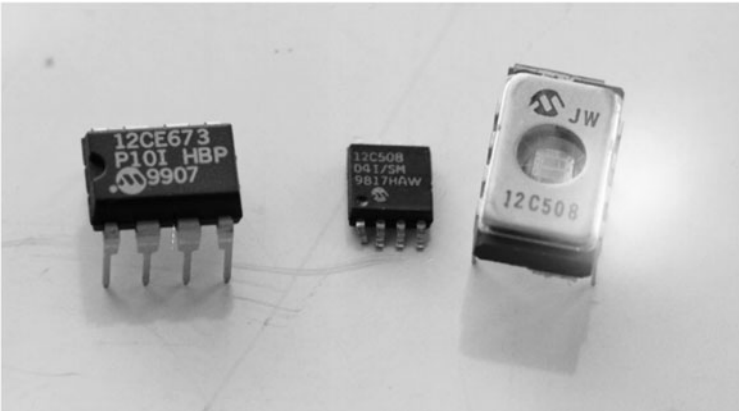


Figure 9.1 A small sample of the variety of different packages available for the different PIC microcontroller part numbers.

Power Input and Decoupling

The different PIC microcontroller part numbers will work with an extremely wide range of power input voltages and are quite tolerant of noise or sags in the supplied power. Power supplies can range from batteries to poorly rectified and filtered ac sources with minimal loss of operation or performance. The ability of the PIC microcontroller to work in less than ideal situations makes the microcontroller families ideal for learning about digital electronics, microcontroller application development, and applications where the quality of power cannot be guaranteed.

Connecting a PIC microcontroller is very simple and only requires a 0.01- to 0.1- μF decoupling capacitor connected to all the V_{dd} pins and wired to ground, as shown in Fig. 9.2. This capacitor filters the voltage both within and without the PIC microcontroller. During transition of the circuits from one state to another (and from low current requirements to high), the internal (and external) current draws inside the chip will change, which may produce fluctuations in the internal voltage levels of the chip, causing it to lock up, reset, or behave unpredictably in other ways. The capacitor filters the power fluctuations and provides a stable power supply for the chip.

The most typically used values for the decoupling capacitors is 0.01 to 0.1 μF . Personally, I prefer using a 0.1- μF tantalum capacitor because it has a low equivalent series resistance (ESR), which minimizes any RC delays in responding to changes to the voltage level. Ceramic disk or polyester caps also can be used for this purpose. Electrolytic capacitors should not be used because of their slow response to quickly fluctuating voltage levels. The lower the capacitor value, the higher frequency it responds to; in some cases you may want to use 0.01- and 0.1- μF capacitors in parallel if your environment is particularly noisy.

Tantalum capacitors are the best choice for decoupling active chips such as the PIC microcontroller, but there are a few things to watch for. The first is the polarity of the

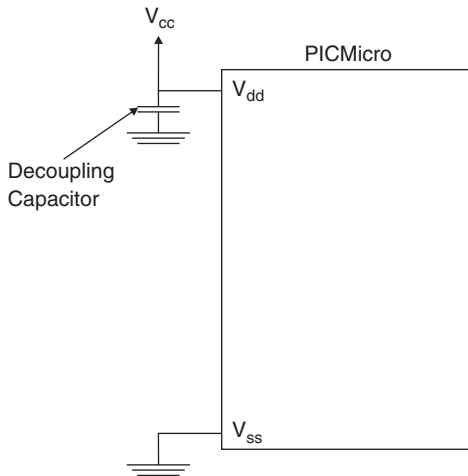


Figure 9.2 The PIC microcontroller requires a 0.01 to 0.1 μF capacitor wired between each V_{dd} pin and ground.

capacitor. Tantalum capacitors inserted backwards, like electrolytic capacitors, can catch fire or explode. This is not an issue with ceramic disk or polyester caps. The second thing is to make sure that you derate the specified voltage of the part to 40 percent or less for your application. By following these simple rules, you should never have any problems working with tantalum capacitors.

Derating a capacitor means that instead of using it at its rated voltage, you should choose a tantalum capacitor that is rated at two and a half times or more of the voltage in the application in which you plan to use them. For my applications, which call for tantalum decoupling capacitors at 5 V, I use parts that are rated at 16 V (which is a derated value less than one-third the rated value).

The primary reason for this seemingly large derating is to protect the capacitors at application power-up. Voltage spikes (which are caused by power-supply startup and current transients within chips) can cause them to develop pinhole breakthroughs in the dielectric, which can cause the plates to touch. When the plates touch, heat is generated by the short circuit, which boils away more dielectric. This process snowballs until the part explodes or catches fire. By derating the cap values, the dielectric layer is thicker, minimizing the opportunity for pinholes to develop.

Most of the recently released PIC microcontroller part numbers are designed for power anywhere from 2.0 to 6.0 V. Some older chips are only able to run with from 4.0 to 6.0 V, whereas other older part numbers have been “qualified” to run with from 2.0 to 6.0 V and are identified as having this capability as being low-voltage devices. Some recent PIC microcontroller part numbers can only run with 2.0 to 3.6 V. This wide variation means that you will have to ensure that you have reviewed the datasheet and understand the operating voltage level for a specific PIC microcontroller part number.

Older low-voltage PIC microcontroller parts are identified by the addition of the letter L before the C or F in its part number. This convention has not been followed for

all parts, so you will have to review the datasheet of the PIC microcontroller that you want to work with to make sure that it will tolerate the voltage extremes.

Many PIC microcontrollers have built-in brown-out reset (BOR) and brown-out detect (BOD) circuitry that is designed to become active when the input power goes below a present point. BOR circuitry will hold a PIC microcontroller reset when the input power falls below a specific level until it rises above it, and BOD circuitry will alert the application that the power has gone below a preset value. For some older chips, this value is set at 4.5 V, whereas in most recent devices the voltage is programmable and will allow you to work with low voltages. The older devices will work with lower voltage levels but cannot have their BOR/BOD circuitry enabled.

If you are working with a PIC microcontroller in a low-voltage application that does not have the programmable BOR/BOD capability, you will have to develop your own brown-out detect circuits like the one shown in Fig. 9.3. In this circuit, if V_{DD} goes below the brown-out voltage determined by the Zener diode, then _MCLR will be pulled low, and the PIC microcontroller will become reset.

A very common problem new application developers face with PIC microcontroller power is accidentally plugging the chip into the circuit backwards after programming. This might sound like it is an unusual problem and not warranted to be mentioned, but it is quite common if you're working on a failing application on which you just can't seem to figure out what has gone wrong and it's 3 a.m. As you get more and more frustrated and tired, the PIC doesn't seem to work at all any more, and you notice that the pin 1 indicators seemed to have jumped to the other end of the chip. The best solution to this issue is to leave the PIC in circuit and program it using ICSP or ICD. I must confess that I have done this on occasion and have been amazed to discover that after the PIC microcontroller is allowed to cool off (it can get very hot when it's plugged in backwards), the chip continues to run perfectly and can be reprogrammed after this event. It seems that unless the chip is allowed to get so hot that the plastic top pops off (which is possible), the PIC microcontroller will work fine after this type of abuse.

The second point about this is that your power supply should be capable of crowbarbing (shutting down) the output if the current draw increases dramatically (such as in the case

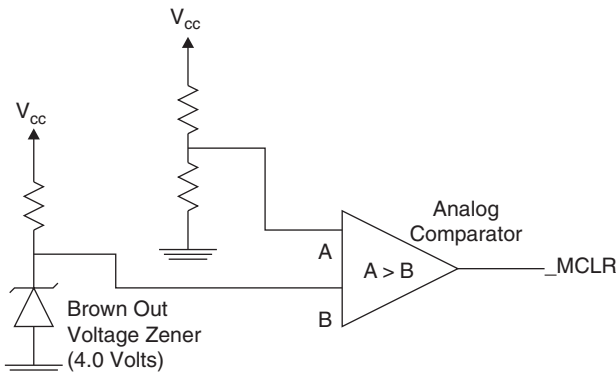


Figure 9.3 You can design your own brown-out reset circuit using a comparator.

of a reversed PIC microcontroller). This is one of the reasons why I like the 78(L)xx series of voltage regulators; they may be more expensive than some other parts, but they won't burn out when they experience overcurrent conditions or cause the parts they are driving to burn out.

Many PIC microcontroller part numbers are identified as having nanoWatt technology. This is a reference to Microchip's advanced device manufacturing processes and the ability to consume as little as 50 nA (50×10^{-9} A) during "sleep." This feature is important for battery-power applications, which can be put to sleep with no functions working other than the watchdog timer and wake on interrupt to ensure maximum battery life. Logitech's Harmony remotes take advantage of this feature by keeping the PIC microcontrollers built into them asleep except for the three cases of sending IR data, updating the LCD display, and being connected to a PC via USB.

HIGH-VOLTAGE DEVICES

As I go through the practical aspects of the PIC microcontroller, one aspect will seem annoying to do for every circuit, and that is creating a power-supply circuit with a voltage regulator with sufficient current rating to drive the circuit. While this is not terribly difficult to do, it can take up valuable real estate and drive up the cost of your application. There are a number of PIC microcontrollers with built-in voltage regulators that allow the PIC microcontroller to be driven without any external regulators in applications where the power input is significantly higher than the nominal 2.0 to 6.0 V normally applied to a PIC microcontroller. These parts are given the letter value HV for high voltage and have a similar pinout as other, more standard PIC microcontroller part numbers, and the output of the voltage regulators is not made available outside the chips. To support the high-voltage PIC microcontrollers, there are a few tricks that you should be aware of, as well as some enhanced features that affect the operation of the part.

To connect a PIC16HV540 (the first high-voltage device to become available) to a battery, the circuit can be as simple as the one shown in Fig. 9.4. In this figure, I have

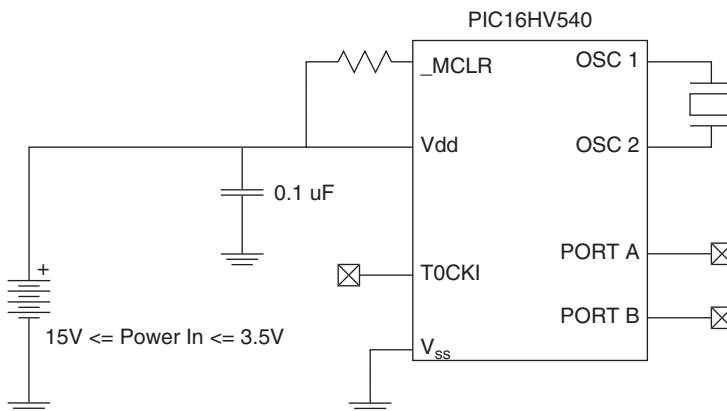


Figure 9.4 The power wiring of the PIC16HV540 is very similar to that of a PIC microcontroller with a regulated power supply.

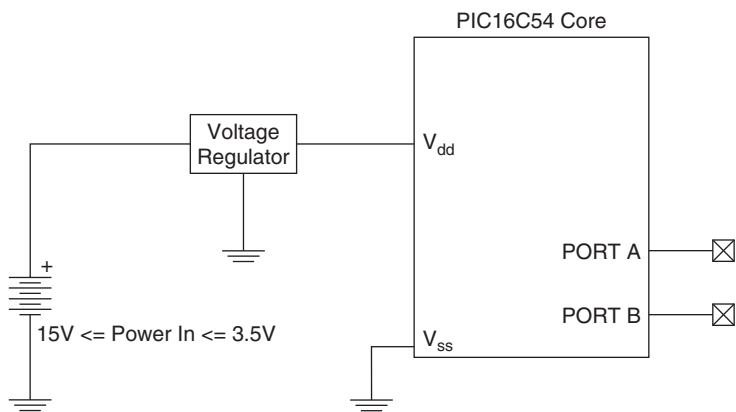


Figure 9.5 Assumed PIC16HV540's internal power system block diagram.

shown the PIC16HV540 as being connected directly to a battery with only a 0.1- μ F decoupling capacitor. I don't show a switch because sleep can be used to turn off the device and put it into a low-current (no more than 14 μ A required) mode. Wakeup from sleep can be accomplished either by watchdog timer timeout, `_MCLR`, reset or a PORTB pin change.

As I've drawn the PIC16HV540, you may think that the device is similar to a PIC16C54 with a voltage regulator in front of it, as is shown in Fig. 9.5. This is not quite true because the I/O port pins may use the device's regulated voltage of the input voltage. The PORTA pins can provide up to the regulated voltage, whereas PORTB provides swings from ground to the input voltage. The actual device's block diagram looks like Fig. 9.6, and PORTA can be used to power +5/+3 V TTL CMOS devices, whereas PORTB is well

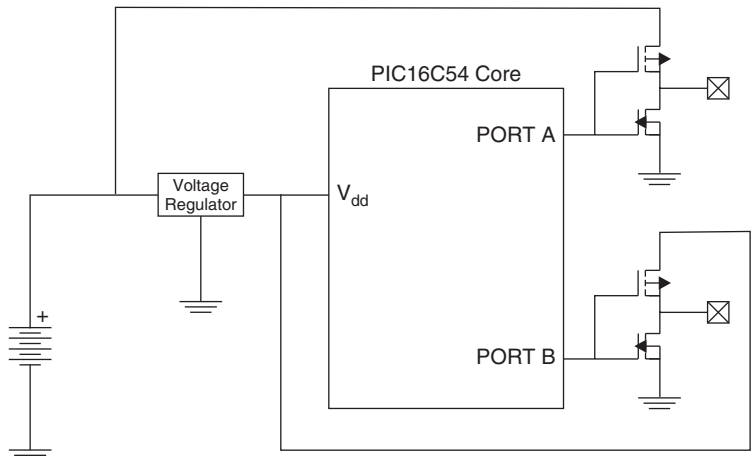


Figure 9.6 Actual PIC16HV540 internal power system block diagram.

TABLE 9.1 PIC16HV540 OPTION2 REGISTER DEFINITION

BIT	DESCRIPTION
7-6	Unused
5	WPC—When set, device will wake up on RB0–RB3 changing.
4	SWE—Software watchdog timer. If the WDT is not enabled in the configuration fuses, setting this bit will enable it in software.
3	RL—Regulated voltage select bit (set for 5 V; reset for 3 V).
2	SL—Sleep voltage level setting (if set, use RL voltage; when reset, use 3 V).
1	BL—Brown-out voltage select (when set –3.1 V for 5-V operation; when reset, –2.2 V for 3-V operation)
0	BE—Brown-out checking enabled when set.

suited for high-voltage I/O. PORTB's threshold voltage is similar to the PIC16C54's (i.e., anything greater than 2.5 V is a 1) and can be used for buttons and LED I/O.

In the PIC16HV540, the voltage regulator can work as either a 5- or 3-V regulator by setting or resetting, respectively, the RL bit of the OPTION2 register, which is in the OPTION/TRIS address space of the low-end PIC microcontroller processor. This register is an auxiliary configuration fuses register that can be modified within an application. The bits of the OPTION2 register are defined as shown in Table 9.1.

The PIC16HV540 is a low-end device, which means that there is a limited amount of space for hardware registers. To write to the OPTION2 register, which is not part of the standard memory map, the `tris` instruction is used as

```
tris    7
```

or

```
tris    OPTION2
```

When the PIC16HV540 is powered on, the RL bit is set, which selects 5-V output from the voltage regulator. A voltage of less than 5 V can be input, but it must be greater than 3.1 V to avoid brown-out reset (which is enabled on power-up) from holding the device reset. If you are powering the PIC16HV540 from a source that is less than +5 V, then the first thing you should do is set the PIC microcontroller for 3-V operation.

Configuration Fuses

The PIC microcontroller is unique in that it has a set of bits that specify the operating specifications of the chip when it is powered up. These bits are collected in a memory address that is outside the execution addresses and are known as the *configuration bits*,

configuration register, or *configuration fuses*. Among the options that can be selected are the clocking circuitry, reset operation, the watchdog timer, and memory protection. Without setting these bits correctly, your PIC microcontroller will not operate properly (chances are that it won't work at all), and understanding their operation is critical to being able to create your own applications.

I *always* recommend using only programmers that can pick up and program the configuration register value embedded in the hex file. In MPASM and PIC microcontroller compilers, there is a directive (usually being some form of or containing the word *config*) in which the bit values are specified. In MPASM, the directive is `__CONFIG`, which is followed by a constant value that will be loaded into the configuration fuses. There are some programmers that allow the user to specify the configuration bit values, but I do not recommend using these programmers because you can forget to specify a bit or specify it incorrectly, which could result in the PIC not working in the application or even having to be discarded because the settings have made it impossible to program again (this is the case with an EPROM device with the code protect feature enabled). By specifying the configuration value in the source code, building it into the hex file, and then programming it automatically into the PIC microcontroller, you have eliminated one potential way in which the chip won't work in the application, and I have to say that one of the most annoying bugs to find and fix is a PIC microcontroller that doesn't have correctly programmed configuration fuses. A typical set of configuration fuse values (taken from the PIC16F877A) is listed in Table 9.2.

These files can be found in the P[part number].inc. file that Microchip includes with MPLAB, where the PIC microcontroller's part number is in the brackets. The configuration fuse values are meant to be ANDed together to get the actual bit pattern to be loaded into the configuration register. For example, for the watchdog timer to be disabled, bit 2 of the configuration register has to be reset. If you look through the label values in Table 9.2, you'll see that this is the only parameter that has bit 2 reset.

Because the different options controlled by the configuration register depend on bits being reset, I recommend that you select one option for each hardware feature in the list of configuration fuses for the PIC microcontroller with which you are working. This is done by matching up each parameter with the same character string inside both. For example, `_CPD_OFF` and `_CPD_ON` control the same option, as does `_LP_OSC`, `_HS_OSC`, `_XT_OSC`, and `_RC_OSC`. The different hardware features that can be controlled by the configuration bits in the PIC16F877A are listed in Table 9.3.

Next, select the options that you require for your application. To understand how the different features work, load the PIC microcontroller's datasheet, and go to the section entitled, "Special Features of the CPU" (it is the same for every part number). This section will explain to you the function of each bit setting.

When you are first starting out, I recommend that you start with the following standard feature settings:

- `_BODEN_OFF`. This is brown-out detect.
- `_CP_OFF`. This is program memory copy protection off.
- `_PWRTE_ON`. This is a 70-ms startup timer that helps to ensure proper clock operation.

TABLE 9.2 PIC18F877 CONFIGURATION FUSE LABELS AND VALUES

LABEL	VALUE	OPERATION
<code>_BODEN_ON</code>	0x03FFF	Brown-out reset detect enabled
<code>_BODEN_OFF</code>	0x03FBF	Brown-out reset detect disabled
<code>_CP_ALL</code>	0x01FFF	All program memory protected
<code>_CP_OFF</code>	0x03FFF	No program memory write protection
<code>_WRT_256</code>	0x03DFF	Upper 256 bytes of program memory write protected
<code>_WRT_1FOURTH</code>	0x03BFF	First quarter of program memory write protected
<code>_WRT_HALF</code>	0x039FF	First half of program memory write protected
<code>_PWRTE_OFF</code>	0x03FFF	72-ms time up delay off
<code>_PWRTE_ON</code>	0x03FF7	72-ms time up delay on
<code>_WDT_ON</code>	0x03FFF	Watchdog timer on
<code>_WDT_OFF</code>	0x03FFB	Watchdog timer off
<code>_LP_OSC</code>	0x03FFC	LP oscillator type
<code>_XT_OSC</code>	0x03FFD	XT oscillator type
<code>_HS_OSC</code>	0x03FFE	HS oscillator type
<code>_RC_OSC</code>	0x03FFF	RC oscillator
<code>_DEBUG_ON</code>	0x037FF	Internal emulator enabled
<code>_DEBUG_OFF</code>	0x03FFF	Internal emulator disabled
<code>_CPD_ON</code>	0x03EFF	Data EEPROM write protected
<code>_CPD_OFF</code>	0x03FFF	Data EEPROM not write protected
<code>_LVP_ON</code>	0x03FFF	Low-voltage program enabled
<code>_LVP_OFF</code>	0x03F7F	Low-voltage program disabled

- `_LVP_OFF`. If available, this feature should be disabled because it uses some I/O pins for controlling the low-voltage programming.
- `_DEBUG_OFF`. If you are not using an MPLAB ICD 2 to debug the application.

Reset and clocking options will be specific to the device you are using, but when you are starting out, I recommend that you select internal options (if they are available).

After listing the different options, the `_CONFIG` statement for the application can be created like the following MPASM statment:

```
_CONFIG _BODEN_ON & _CP_OFF & _WRT_ENABLE_ON & _PWRTE_ON & _WDT_OFF
& _XT_OSC & _DEBUG_OFF & _CPD_ON & _LVP_OFF
```

**TABLE 9.3 PIC18F877A CONFIGURATION CONTROLLABLE
HARDWARE FEATURES GROUPED BY __CONFIG PARAMETER NAMES**

LABEL	OPTION
__BODEN_##	Brown-out detect
__CP_##	Code protect
__WRT_##	Program memory application write enable
__PWRTE_##	72-ms power-up wait timer
##_OSC	Oscillator type
__DEBUG_##	Built-in emulator
__CPD_##	Data EEPROM write enable
__LVP_##	Low-voltage programming enable

When this statement is evaluated, the constant values are replaced with the constants values located in the .inc file and evaluated together into a constant to be loaded into the configuration bits. For the labels selected earlier, the __CONFIG statement could be reduced to

```
__CONFIG 0x03F71
```

which is the value that will be loaded into the configuration fuse register.

SYSTEM CLOCK/OSCILLATORS

There are four different methods of clocking the PIC microcontroller. The different options are designed to fill different application requirements for cost, speed, and accuracy. There are few applications that require extreme accuracy, so cheaper clock designs are more usual.

The clock/oscillator options are

- 1 Internal oscillators
- 2 External resistors and capacitors
- 3 Crystals and ceramic resonators
- 4 External oscillators

Internal oscillators A feature that is becoming more and more prevalent in low-cost PIC microcontrollers is the built-in RC oscillator, as shown in Fig. 9.7, which can be used in place of external oscillator circuitry. The OSCAL register is loaded with a calibration value determined by Microchip during the manufacturing process. This type of oscillator will have an accuracy of 1.0 percent or better while running at 4 MHz. The internal oscillator simplifies application wiring, and if it is used, the pins that are normally used by it can be used as I/O pins, which improves the efficiency of small-pin-count devices.

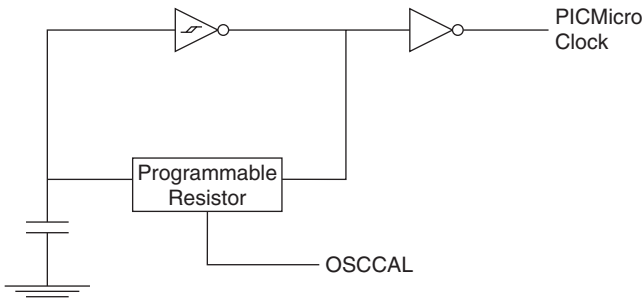


Figure 9.7 The internal oscillator consists of a built-in RC oscillator with a calibration register.

The first parts that were built had this circuit, but in the last few years, a number of different parts have become available with two built-in oscillators running at 8 MHz and 31 kHz, as shown in Fig. 9.8. Operation of the oscillators, along with the value passed to the first multiplexor from the postscaler, can provide a system clock of 31, 125, 250, and 500 kHz and so on up to 8 MHz, giving you a considerable range of operating speeds. Unlike the 4-MHz oscillator of the first PIC microcontrollers built with internal oscillators, this circuit allows for the execution speed of the chip to be changed during operation to minimize power drain, except when you need to perform complex calculations very quickly.

External resistors and capacitors Before introduction of the internal oscillator, if you wanted a very-low-cost oscillator circuit, you would have to go with the external RC oscillator option, which consisted of a resistor-capacitor network that provides the clocking for the PIC microcontroller, as shown in Fig. 9.9. The resistor-capacitor charging/discharging voltage is buffered through a Schmidt Trigger noninverting buffer, which is used to enable or disable an N-channel MOSFET transistor pull-down switch. This circuit is quite simple but only accurate to about 10 percent, and it really should not be considered for applications when the internal oscillator is available with no external components and better accuracy.

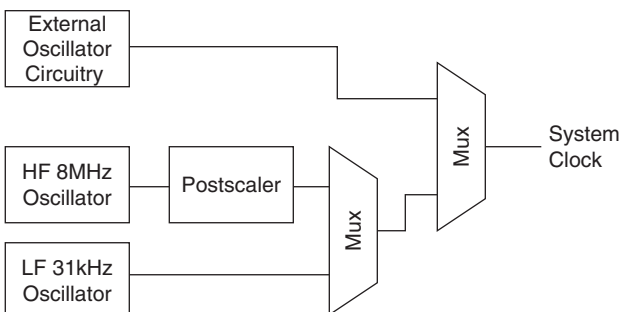


Figure 9.8 Newer PIC microcontrollers have an internal oscillator circuit that can select operating speeds from 31 kHz to 8 MHz.

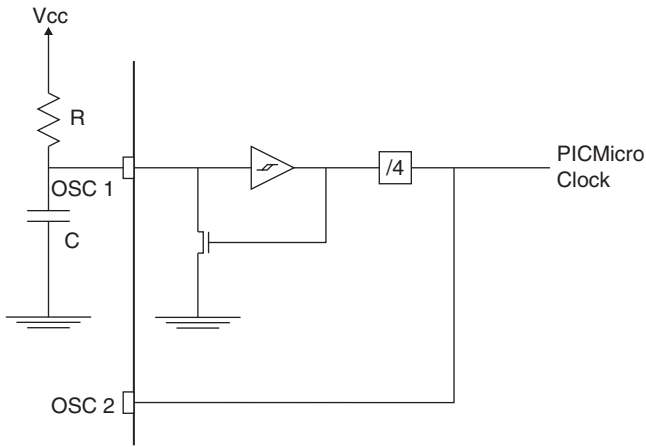


Figure 9.9 The external RC oscillator circuit, which can be built very inexpensively but is very inaccurate.

Originally, the advantage of the RC oscillator was its very low cost and its ability to drive an instruction clock whose output can be used by other devices in the circuit. There are many disadvantages to this method compared with the internal oscillator, and they include

- Lower operating speed (normally the RC oscillator can run at a maximum of 1 MHz).
- Poor accuracy (an error of 30 percent to the target speed is not unusual with poor-tolerance capacitors).
- Difficulty in selecting components to be used to produce a specific operating frequency (there is no single table defining the frequency; instead you must use a graph in the datasheet to select the resistor and capacitor values for different frequencies).
- Generally uses more power than the internal oscillator for the same speed owing to the additional current passing through the resistor.

Some devices have a single external resistor option for an inexpensive oscillator, but this has many of the same disadvantages as the full RC oscillator discussed earlier.

Before introduction of internal oscillators to PIC microcontrollers, the external RC oscillator was the only option to a very-low-cost system clock. Today, I would discourage you from considering this circuit and instead suggest that you select a PIC microcontroller part number that has an internal oscillator.

External crystals, resonators, and oscillators Crystal and ceramic resonators use a similar connection scheme for operation. The crystal or ceramic resonator is wired into the circuit as shown in Fig. 9.10. Crystals and ceramic resonators delay the propagation of a signal a set amount of time. This set amount depends on how the crystal is cut. In addition, for best results, a parallel-circuit crystal should be used.

The two capacitors shown in Fig. 9.10 are connected to one side of the crystal or ceramic resonator and the other to ground. The values of the two capacitors are specified

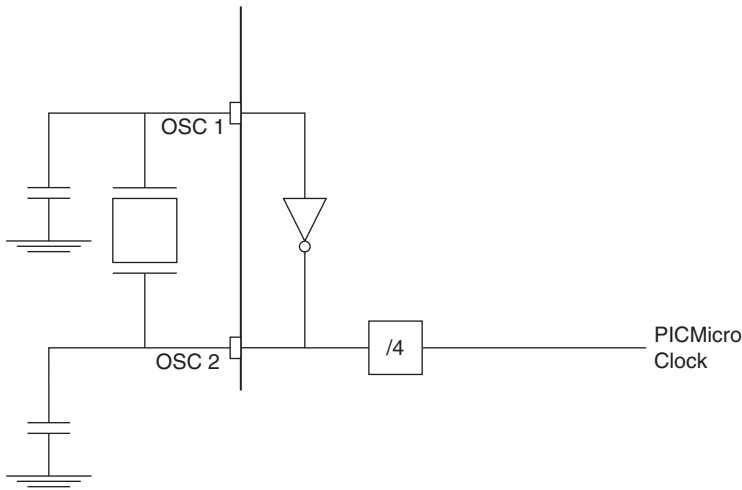


Figure 9.10 The external crystal oscillator circuit is used for both parallel-cut crystals and ceramic resonators.

in the PIC microcontroller device’s datasheet and usually match one another. You may find for some speeds that different values are specified for the two capacitors.

Every PIC microcontroller has three external crystal or ceramic resonator speed ranges requiring different operating parameters within the chip. Microchip refers to these speed ranges as *oscillator types*, and they are set in the configuration register to enable the different circuitry needed to ensure proper operation. Table 9.4 lists the oscillator types and their operating frequency ranges.

Using the crystal or ceramic resonator, the OSC2 pin can be used to drive one CMOS input, as shown in Fig. 9.11. Note that the input capacitance of the CMOS buffer could affect the operation of the PIC microcontrollers oscillator (i.e., causing it to not run at all). Make sure that you understand what is the capacitance of the input circuit that you are using and whether or not it is significant enough to affect the operation of the oscillator. You may find that you have to use an external oscillator and wire its output to the PIC microcontroller and other devices.

A ceramic resonator behaves similarly to a crystal resonator, except that it is not built with a quartz device, like the crystal. Ceramic resonators are much more robust (i.e., can withstand more severe physical shocks) than crystals, and in large quantities,

TABLE 9.4 PIC MICROCONTROLLER OSCILLATOR FREQUENCY RANGES

OSCILLATOR TYPE	OPERATING FREQUENCY RANGE
LP	0–200 kHz
XT	200 kHz–4 MHz
HS	4–20 MHz (or the device maximum)

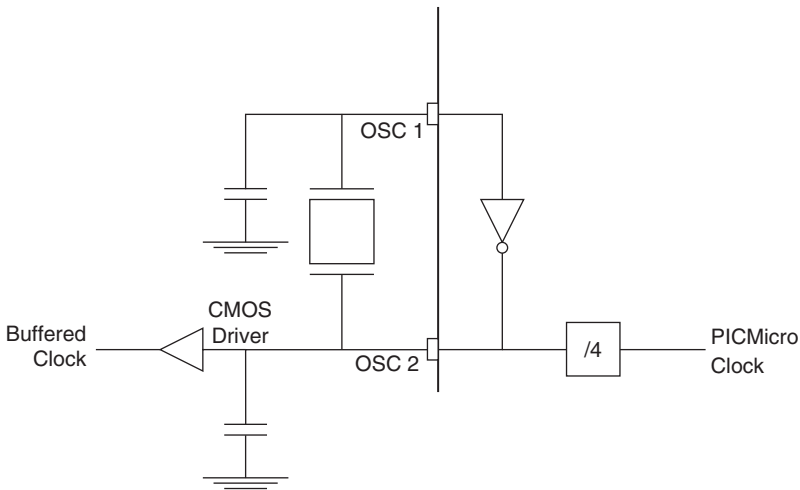


Figure 9.11 The OSC2 pin can be used to drive a single CMOS input, which allows the PIC clock to be distributed throughout the application circuit.

they are much cheaper. Many ceramic resonators are available as three-pin devices, which have the external capacitors built into them, meaning that along with just wiring them to the PIC microcontroller's OSC1 and OSC2 pins, just a ground connection is required. The downside of ceramic resonators is that their accuracy is not as good as that of crystal resonators (usually accurate to 0.5 percent versus a crystal accuracy of 0.02 to 0.1 percent), and some devices with built-in capacitors may not be suitable for use with the PIC microcontroller.

When designing and laying your application circuit, always remember to keep the parts of the oscillator as close as possible to grounds. When the specified parts are used for oscillators and are kept close to the PIC microcontroller, you'll be impressed with the stability and robustness of PIC microcontrollers' oscillator and instruction clocking.

Finally, you can use an external oscillator driving the OSC1 pin directly, as shown in Fig. 9.12. The OSC2 pin can be used to redrive the clock (although inverted) to other devices.

If you look through *PICList* and other forum archives, you periodically will see postings from people who drive their PIC microcontrollers at much beyond their rated speed. This is possible for most devices because of the design qualification Microchip does to ensure that PIC microcontrollers run at the rated speeds. It is not recommended that you count on your PIC microcontrollers to run at faster than rated speeds because this capability is not guaranteed, and you will find devices that will not run at the speeds required by your application.

PIC18 oscillator circuitry As discussed earlier, the typical PIC microcontroller has four oscillator modes that are selected from the configuration fuses when the PIC microcontroller starts up. The PIC18 has one more oscillator option available, along

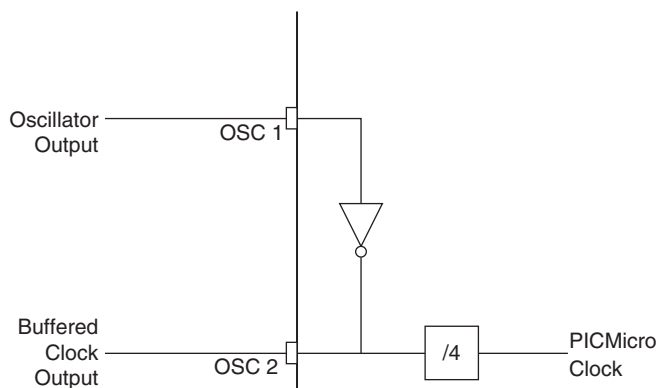


Figure 9.12 An external oscillator can drive the OSC1 pin of the PIC microcontroller with OSC2 as an available driver for other circuits in the application.

with the ability to change an unused OSC2 pin into an additional input/output (I/O) pin. A PLL clock that is four times the multiplier circuit is available, and it allows the PIC18 microcontroller to run with a one instruction cycle per clock cycle. Rounding out the features, the PIC18 can be driven by a TMR1 oscillator for reduced power or faster application execution. This may seem like a dizzying array of options, but they can be listed as

- 1 RC oscillator
- 2 LP oscillator
- 3 HS oscillator
- 4 XT oscillator
- 5 External oscillator
- 6 TMR1 clock

LP and XT modes execute exactly the same way on the PIC18 as on the other PIC microcontrollers. RC can work exactly the same with a one-quarter-speed clock driven out of the OSC2 pin or can have the OSC2 pin changed to the RA6 I/O pin; this is known as *RCIO mode*. The external oscillator option will take in an external clock signal and output a one-quarter-speed clock on OSC2 unless the OSC2 pin is to be used as RA6 (like the RC oscillator mode and known as *ECIO mode*). The external oscillator will work for all data speeds from dc to 48 MHz that the PIC18 can run at. The RC, RCIO, XT, LP, ES, ESIO, HS, and HSPLL oscillator modes are selected at programming time within the PIC18's configuration fuses.

One of the most interesting features of the PIC18 is HS mode, which can run like a typical PIC microcontroller, from 4 to 40 MHz and in some part numbers 48 MHz. An optional phased-locked-loop (PLL) clock multiplier can be added to the external clock (with the external clock limited to 10 MHz). This feature (known as *HSPLL mode*) allows a 10-MHz PIC18 microcontroller to run as if it were being driven with a 40-MHz clock.

The advantages of this feature include reduced EMI emissions from the PIC microcontroller and reduced oscillator power consumption.

The last oscillator mode consists of software enabling the TMR1 oscillator as the PIC18 microcontroller's processor clock source. This feature allows application execution with I/O polling and interrupts to continue, except at a vastly reduced rates (and power consumption). This mode is enabled and disabled by setting and resetting, respectively, the SCS bit of the PIC18's OSCON register. For this mode to work, the OSCON bit of the configuration fuses must be reset. When the TMR1 oscillator is enabled (by setting the SCS bit), execution moves over immediately to the TMR1 clock, and the standard oscillator is shut down. This transition is very fast, with only eight TMR1 clock cycles lost before execution resumes with TMR1 as the clock source. When transitioning from TMR1 to the standard oscillator, the oscillator is restarted with a 1024-cycle delay for the clock to stabilize before resuming execution. The oscillator circuit in the PIC18Cxx appears in block diagram form in Fig. 9.13.

RESET

Reset in the PIC microcontroller is very simple and easy to implement, as shown in Fig. 9.14, and with many recently released PIC microcontroller part numbers, the task of implementing reset has been simplified even more with the addition of internal reset capabilities and optional built-in brown-out reset (BOR) or brown-out detect (BOD) functions in some of the new parts. These functions, along with a 72-ms power-up delay function (the PWRTE bit of the configuration register), make it easy to come up with an application with good reset characteristics in a variety of situations with little or no external hardware.

The simple pull-up on MCLR# (the master clear pin) is really all that is required for most applications where an external reset is required. This may be a departure for you,

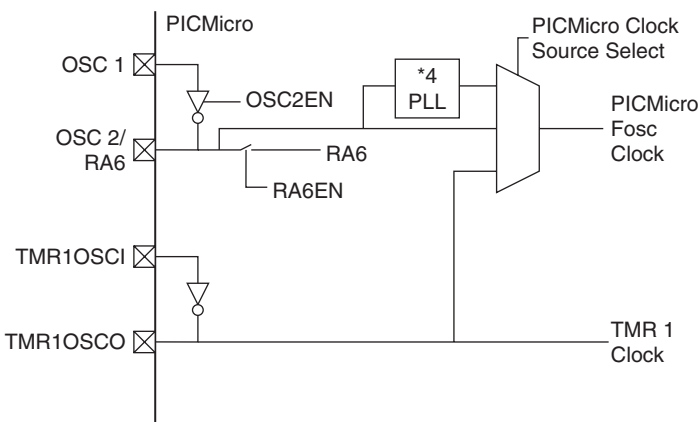


Figure 9.13 The PIC18 has the base oscillator capabilities of the low-end and mid-range PIC microcontrollers but also has the ability to run from a second oscillator as well as multiplying the incoming clock frequency by four using a PLL.

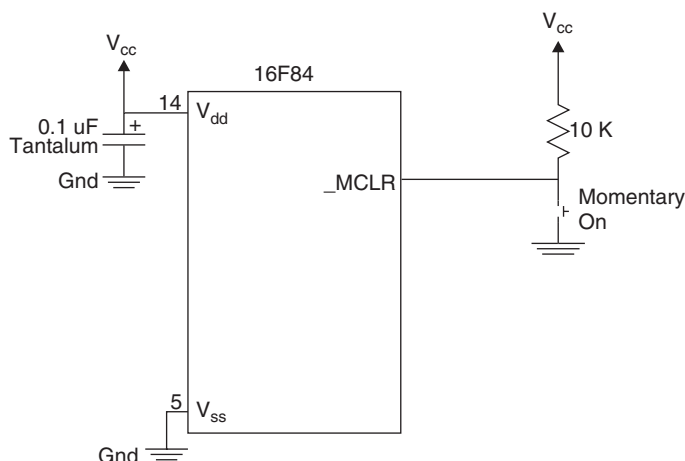


Figure 9.14 The external reset circuit can be as simple as a pull-up on MCLR#. A momentary on switch tying MCLR# to ground provides a reset control.

if you have experience with other chips that required a delay on the reset circuit to ensure that power is up and stabilized before starting the device. The internal PIC microcontroller circuitry will not poll the MCLR# pin until power is stabilized and it is ready to start executing. To help make sure that the power-up sequence is reliable, the PWRTE function, which is discussed below, will allow you to delay the power-up sequence even further, primarily for the reason of ensuring that the oscillator is running correctly before the application code starts to execute.

The internal reset frees up the MCLR# pin from use as the reset control source and allows it to be used as an input pin. When this feature is enabled, the device reset is active as long as more than 4.0 V is available at V_{dd} for regular parts or 2.0 V is available in extended voltage parts. The freed MCLR# pin can only be used as an input (no output drivers are built in) and does not have the same clamping diodes as the other PIC microcontroller I/O pins.

The brown-out detect (BOD) function, if it is on the PIC microcontroller that you are working with and is enabled, will cause the internal reset circuitry to become active when the V_{dd} voltage becomes less than 4.0 V in 5-V applications. Some PIC microcontroller part numbers also have a programmable voltage for the BOD function, allowing you to work with other voltages. This can be useful in battery-powered applications, where a drop in battery power can cause intermittent application execution. The base BOD function is enabled through the configuration register and cannot be accessed within the application code except for checking the PCON _BOD bit.

POWER-UP TIMER

The PWRTE function, which is enabled in the configuration register and cannot be accessed by the application code, will cause a delay in the start of a PIC microcontroller

application for 72 ms. This feature is designed to allow the PIC microcontroller internal clock to stabilize before the application starts executing. The PWRTE bit of the configuration register always should be active, unless the application has a clock that is external to the PIC microcontroller that is stable when reset becomes disabled and the PIC microcontroller starts executing.

WATCHDOG TIMER

Environments with noisy power that could encounter large electrical fields or experience large electrostatic discharges (ESDs) can cause the circuits within the PIC microcontroller to become “upset” and stop executing properly. The reason for the upset is that internal bits in the PIC microcontroller become set to invalid values, resulting in the program counter changing to invalid addresses or the instruction decoder processing an instruction improperly. Often when this happens the PIC microcontroller “locks up” and will stop executing all together. To help counter this problem, Microchip has designed a watchdog timer (WDT) in all PIC microcontrollers that will reset the PIC microcontroller if normal application execution is lost and the PIC microcontroller starts executing incorrectly or locks up.

The watchdog timer (Fig. 9.15) consists of an 18-ms RC oscillator clock delay that will reset the PIC microcontroller if it is allowed to time out. Normally in an application, it is reset before timing out by executing a `clrwdt` instruction. The overflow (O/F) output of the counter is optionally passed to a prescaler before going to the PIC microcontrollers reset circuit. The prescaler will be discussed in more detail later in this chapter, but its basic purpose is to count overflow events, either from the PIC microcontroller’s TMR0 or the watchdog timer. In the case of the watchdog timer, the overflow is passed to the PIC microcontroller’s reset circuit when the specified number of watchdog timer events has occurred and the prescaler has overflowed. The prescaler allows watchdog timer reset delays of from 18 ms to 2.3 s.

When the watchdog timer causes a reset in the PIC microcontroller, the `_TO` bit of the STATUS register is reset. In the initial code of your application, if watchdog timer resets are used, then the `_TO` bit should be checked because the current file register settings probably will be set from the previous application execution.

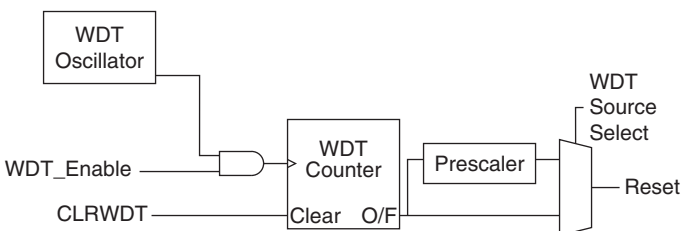


Figure 9.15 The watchdog timer (WDT) reset signal can be passed through a prescaler to delay the onset of a timeout reset.

It is recommended that the watchdog timer be reset by a single `clrwdt` instruction in the application after half the reset period has passed. The nominal error in the RC oscillator used for the watchdog timer function is 20 percent, which means that watchdog timer timeouts can take place anywhere from 14 to 22 ms (when no prescaler is used). To be on the safe side, executing `clrwdt` every 9 ms in this situation will avoid any potentially invalid watchdog timer resets.

The watchdog timer is enabled from within the configuration word and cannot be disabled within the application. This means that you have to be very careful to avoid enabling the watchdog timer unless you have provided support for it in the application code. *Providing support* for the watchdog timer means that the `clrwdt` instruction is executed repeatedly to prevent the watchdog timer from resetting the PIC microcontroller unexpectedly. In many PIC microcontrollers, the watchdog timer enable bit of the configuration word is positive active and enabled when the bit is set (i.e., unprogrammed). This can cause some problems for new application developers who forget to disable the watchdog timer explicitly.

MEMORY PROTECTION

All PIC microcontrollers have the ability to protect the contents of memory either from being read back by programmers or from being altered inadvertently by an errant application when the application code has the capability of changing the contents of the program memory. Preventing data from being read back is a security procedure to thwart pirates from downloading and copying application code. In some PIC microcontrollers, the Flash program memory can be written to by the application; in some cases this is highly desirable, whereas in others it cannot be allowed. The programmer read-back code and self-write disable protection functions of the PIC microcontroller are controlled by configuration register bits and cannot be read back or altered by the application code.

Code protection must be implemented only when the application code burned into the PIC microcontroller has been proven to be correct. With code protection enabled, you will not be able to examine a part during execution using ICD, and in the case of windowed EPROM parts, there is often an aluminum layer over the code-protection bits protecting them from ultraviolet (UV) light and rendering them impossible to reprogram. Only when you are *absolutely* sure that the code is correct should you enable the code-protection bits.

EXTERNAL MEMORY

Parallel memory devices can be connected to selected part numbers of the PIC17 and PIC18 microcontroller families to increase the program memory space available to the processor. In the PIC17, the interface provided is up to 64 kB of 16-data-bit “words” via a multiplexed address/data bus, and in the PIC18, up to 2 MB of data (either 8 or 16 bits wide) can be accessed. The multiplexed bus may seem somewhat difficult to use, but it actually isn’t. Memory devices can be added quite easily and quickly, and in the case of external Flash memory added to the PIC18 devices, the MPLAB ICD 2 can be used to program them with application code. Since the second edition of this book, the capabilities of the PIC18 devices have increased to the point where adding Flash memory to the application is a very viable alternative.

An unprogrammed PIC17's configuration fuses set the PIC microcontroller into microprocessor mode, which cannot access any internal program memory. This allows output devices to be placed into applications, with external program memory providing the application code. This feature presents a way to debug an application before it is burned into the PIC microcontroller.

For both the PIC17 and PIC18 microcontrollers, external memory can be read from or written to using the **TABLRD** and **TABLWT** instructions. In both devices where there is both internal and external program memory, the internal program memory can be read using the **TABLRD** instruction in the microcontroller modes. When accessing data, these table instructions use the table pointer registers (**TBLPTRU** for bits 16 and above, **TBLPTRH** for bits 15 through 8, and **TBLPTRL** for the low 8 bits) to address the operation. During 16-bit table reads and writes, the table latch register (**TABLATH** for the high byte and **TBLATL** for the low byte) is used to buffer the 16 bits during the transfer. The PIC17 can only access data 16 bits at a time, whereas the PIC18 can access data either 8 or 16 bits at a time.

Connecting external devices to a PIC17Cxx microcontroller is relatively simple, with two 74LS373 latches used for buffering the address before the I/O operation takes place (Fig. 9.16). The address bits can be decoded to provide access for multiple devices. A 74LS138 can be used to decode three lines into eight negative active outputs. When performing a read, the AD bus and ALE and **_OE** lines look like Fig. 9.17.

These waveforms are actually quite traditional and match up with those of many microprocessors (such as the Intel 8086), but they should be reconciled with the waveforms for the **TABLWT** and **TABLRD** instructions presented earlier. Note that only one of the two data transfers will be “visible” on the PIC microcontroller's external bus. You also should note that two clock cycles are used for the data transfer. This means that the data access speed of the external device has to be less than twice the period of the PIC microcontroller's clock. For example, if the PIC microcontroller were running at 10 MHz,

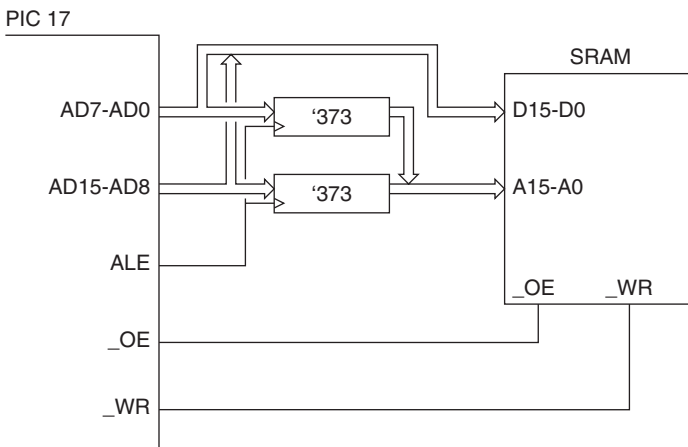


Figure 9.16 The PIC17 external memory interface is quite simple, with two 373 latches used to buffer the lower address bits.

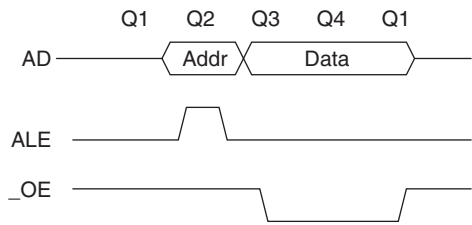


Figure 9.17 The PIC17 memory read is quite straightforward.

the clock period would be 100 ns, and any external devices connected to the bus would have to have an access time of 200 ns or less.

The PIC18 works very similarly to the PIC17, but there are a few differences that you should be aware of. The first is that a JEDEC standard parallel-interface NOR Flash memory connected to the PIC microcontroller can be programmed using the MPLAB ICD 2. As part of this operation, the availability of external memory is selected during the MPLAB ICD 2 enable, and then during the programming operation, the device ID of the Flash chip is read and the appropriate programming algorithm is used. There are a few wrinkles to this operation, but these center around the PIC18 used and the features (such as BootRAM and internal Flash) that affect how the Flash is actually accessed. The ability to program Flash that is connected to the PIC microcontroller is a huge advantage when developing applications because it eliminates the need for a separate Flash programmer, as well as the need to convert the hex file to the data format required by the Flash chip and the programmer used with it.

Second, the PIC18 can support either an 8- or 16-bit external memory device. The 16-bit read/write timing is similar to that of the PIC17, but the 8-bit timing is quite a bit different, as shown in Fig. 9.18. The 16-bit access is controlled by the BA0 pin of the PIC microcontroller, which is initially low to read the low byte of the 16 bits and then high to

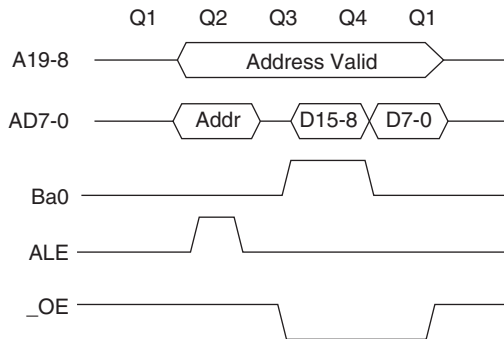


Figure 9.18 PIC18 16-bit memory read operation with BA0 switching to allow both bytes to be read in a single instruction cycle.

initiate read of the high byte. The sequential read of the 2 bytes is done in the same instruction cycle, effectively halving the time available for the data to be output on the bus. This will be an issue when you are running the PIC microcontroller at high speeds; for example, at 48 MHz, the time available for the Flash to deliver the data will be less than 20 ns. To alleviate the potential for this problem, many devices allow for wait states to be inserted into the read process, allowing slower memory devices to be used.

PIC MICROCONTROLLER CONFIGURATION REGISTERS

When I discuss programming PIC microcontrollers, I recommend using only programmers that can pick up the value specified in the `CONFIG` statement and programming it into the PIC microcontroller automatically without any intervention from the user. This is important because it eliminates one step that is often missed or done incorrectly by the user. The `__CONFIG` MPASM directive (note that there are two underscores before `CONFIG`) is used to specify the configuration fuse options for the PIC microcontroller. Along with the `__CONFIG` directive, a number of constants are ANDed together to provide the configuration fuse values.

A typical set of configuration fuse values (taken from the PIC16F877) is listed in Table 9.5 and can be found in the `P[part number].inc` file that Microchip includes with MPLAB (where [part number] is the PIC microcontroller's part number). When working with other tools, there should be a similar include file with the same information.

These values are meant to be ANDed together to get the actual bit pattern to be loaded into the configuration register. For example, for the watchdog timer to be disabled, bit 2 of the configuration word has to be reset. If you look through the label values in the preceding table, you'll see that this is the only parameter that has bit 2 reset. If the watchdog timer to be enabled, then the full 14 bits of the configuration word are set, which doesn't affect any other value ANDed with it.

Because the different options controlled by the configuration register depend on bits being reset, I recommend that you select one of every option in the list in Table 9.5. This is done by matching up each parameter with the same character string inside both. For example, `_CPD_OFF` and `_CPD_ON` control to the same option, as does `_LP_OSC`, `_HS_OSC`, `_XT_OSC`, and `_RC_OSC`. Table 9.6 lists the different option for the PIC16F877.

After listing the different options, the `__CONFIG` statement for the application can be created in the application as follows:

```
__CONFIG _BODEN_ON & _CP_OFF & _WRTE_ENABLE_ON & _PWRTE_ON & _WDT_OFF  
& _XT_OSC & _DEBUG_OFF & _CPD_ON & _LVP_OFF
```

When this statement is evaluated, the constant values are replaced with the preceding constants and evaluated together into a constant to be loaded into the configuration fuses. For the labels selected above and replaced with the corresponding constants, the `__CONFIG` statement could be reduced to

```
__CONFIG 0x03F71
```

TABLE 9.5 PIC16F877 CONFIGURATION FUSE CONSTANTS TAKEN FROM P16F877.INC

LABEL	VALUE	OPERATION
<code>_BODEN_ON</code>	0x03FFF	Brown-out reset detect enabled
<code>_BODEN_OFF</code>	0x03FBF	Brown-out reset detect disabled
<code>_CP_ALL</code>	0x00FCF	All program memory protected
<code>_CP_HALF</code>	0x01FDF	Upper half of program memory protected
<code>_CP_UPPER_256</code>	0x02FEF	Upper 256 bytes of program memory protected
<code>_CP_OFF</code>	0x03FFF	No code protection
<code>_WRT_ENABLE_OFF</code>	0x03DFF	Program memory can be written to from application
<code>_PWRTE_OFF</code>	0x03FFF	72-ms time-up delay off
<code>_PWRTE_ON</code>	0x03FF7	72-ms time-up delay on
<code>_WDT_ON</code>	0x03FFF	Watchdog timer on
<code>_WDT_OFF</code>	0x03FFB	Watchdog timer off
<code>_LP_OSC</code>	0x03FFC	LP oscillator type
<code>_XT_OSC</code>	0x03FFD	XT oscillator type
<code>_HS_OSC</code>	0x03FFE	HS oscillator type
<code>_RC_OSC</code>	0x03FFF	RC oscillator
<code>_DEBUG_ON</code>	0x037FF	Internal emulator enabled
<code>_DEBUG_OFF</code>	0x03FFF	Internal emulator disabled
<code>_CPD_ON</code>	0x03EFF	Data EEPROM write protected
<code>_CPD_OFF</code>	0x03FFF	Data EEPROM not write protected
<code>_LVP_ON</code>	0x03FFF	Low-voltage program enabled
<code>_LVP_OFF</code>	0x03F7F	Low-voltage program disabled

One feature of the `__CONFIG` directive is that it is located differently for the different PIC microcontroller families. For low-end devices, the configuration fuses are at 0xFFF. For mid-range devices, they are at 0x2007. The PIC17's configuration fuses are in the address range 0xFE00 to 0xFE07. Finally, the PIC18's configuration fuses are at the address ranges 0x300000 to 0x30000F.

For low-end and mid-range parts, I like to think of the `__CONFIG` directive being the equivalent to an `ORG` statement with a `dw` statement. Using the preceding example, this would make the `__CONFIG` statement equivalent to

```
org 0x02007
dw 0x03F71
```

TABLE 9.6 PIC18F877 CONFIGURATION FUSE CATEGORIES	
LABEL	OPTION
BODEN	Brown-out detect
CP	Code protect
WRT_ENABLE	Program memory application write enable
PWRTE	72-ms power-up wait timer
OSC	Oscillator type
DEBUG	Built in ICD interface hardware
CPD	Data EEPROM write enable
LVP	Low-voltage programming enable

Along with the configuration fuses, four nybbles are available for serial numbers, IDs, or application version information and are loaded using the `__IDLOCS` (again, two underscores) directive. Each nybble of the 16-bit `__IDLOCS` directive parameter is placed in one of the IDLOCS program memory locations. For example, in a mid-range PIC microcontroller, the statement

```
__IDLOCS 0x01234
```

will load the program memory with the contents listed in Table 9.7.

This data storage may seem like a waste (because every instruction location has 14 bits and only 4 are used), but this is how `__IDLOCS` works. Taking a clue from my description of how `__CONFIG` works, you could code the data into the four `__IDLOCS` locations as

```
org      0x02000
dw      IDLOCS_VALUE1
```

TABLE 9.7 MID-RANGE PIC MICROCONTROLLER IDLOCS DATA PLACEMENT	
ADDRESS	DATA
0x02000	0x00001
0x02001	0x00002
0x02002	0x00003
0x02003	0x00004

```

dw          IDLOCS_VALUE2
dw          IDLOCS_VALUE3
dw          IDLOCS_VALUE4

```

The `__IDLOCS` and `__CONFIG` data are stored outside the application-accessible program memory in the low-end, mid-range, and PIC18 microcontrollers. The values put into the respective program memory areas can be accessed directly by the PIC17.

PIC18 configuration fuses The PIC18 devices have a large number of configuration fuses, and they conceivably can be stored in sixteen 16-bit program memory addresses with a total number of possible options of 256. As with low-end and mid-range PIC microcontroller's configuration registers, I recommend that you list every category of configuration fuse for the PIC18 part number you are working with and establish the category of each configuration feature and make sure that a selection is made for each one. The large number of options will make this task tedious but should save you from having problems later when you are trying to bring up the application.

When the PIC18 microcontrollers first came out and were added to MPASM, each of the different program memory word bytes had to have its address and the high/low byte specified, along with the fuses that were to be modified and ANDed together. The `__CONFIG` statements looked something like:

```

__config 0x300000L _HS_OSC & _WDT_OFF
__config 0x300000H _PWRTE_ON
__config 0x300002L _CP_OFF & _CPD_OFF

```

and it is a lot of work to go through the datasheet and .inc file to make sure that everything is listed and correct.

The latest versions of MPASM improve this work and make it somewhat easier to get right by having you specify the feature and state while ignoring the register address and the need to AND them together. For example, the five options above could be entered into the single `__CONFIG` statement:

```

__config osc=hs, wdt=off, pwrt=on, cp=off, cpd=off

```

The need for listing all the different categories of configuration fuse has not gone away, but the task has been simplified somewhat.

You can put in multiple `__CONFIG` directives, and I recommend that you do so to keep track of the different features with respect to the register. This means that instead of loading up all the configuration values on one line, I recommend that you use the following format to ensure that no values are missed and that you can cross-reference to the datasheet very easily:

```

__config osc=hs, wdt=off      ; 0x300000L
__config pwrt=on              ; 0x300000H
__config cp=off, cpd=off      ; 0x300002L

```

OPTION Register

The OPTION register is a cornerstone register to operation of the PIC microcontroller. This register controls operation and delay of the prescaler, selects the clock source, and specifies operation of the interrupt source pin. It is a very useful register and one that you always should keep in the back of your mind and make sure that it is set up properly for your application.

Table 9.8 shows the low-end chip’s OPTION register definition, and Table 9.9 shows the mid-range PIC microcontroller’s OPTION register definition. The functions provided by the OPTION register are provided in multiple registers in the PIC18. PIC17 microcontrollers do not have an OPTION register because many of the functions continued by option are either not present (such as the prescaler and PortB weak pull-ups) or are provided in other registers.

TABLE 9.8 LOW-END PIC MICROCONTROLLER OPTION REGISTER DEFINITION	
BIT	LABEL/FUNCTION
7	_GPWU—enables wakeup on pin change (not available on all devices)
6	_GPPU—enable I/O PortB weak pull-ups (not available on all devices)
5	TOCS—TMR0 clock source select
	1—Tock1 pin
	0—instruction clock
4	TOSE—TMR0 increment source edge select
	1—high to low on Tock1 pin
	0—low to high on Tock1 pin
3	PSA—prescaler assignment bit
	1—prescaler assigned to watchdog timer
	0—prescaler assigned to TMR0
2–0	PS2—PS0—prescaler rate select
	000—1:1
	001—1:2
	010—1:4
	011—1:8
	100—1:16
	101—1:32
	110—1:64
	111—1:128

TABLE 9.9 MID-RANGE PIC MICROCONTROLLER OPTION REGISTER DEFINITION

BIT	LABEL/FUNCTION
7	_RBPV—enables PortB weak pull-ups 1—pull-ups disabled 0—pull-ups enabled
6	INTEDG—interrupts request on 1—low to high on RBO/INT 0—high to low on RBO/INT
5	TOCS—TMR0 clock source select 1—Tock1 pin 0—instruction clock
4	TOSE—TMR0 update edge select 1—increment on high to low 0—increment on low to high
3	PSA—prescaler assignment bit 1—prescaler assigned to watchdog timer 0—prescaler assigned to TMR0
2–0	PS2–PS0—prescaler rate select 000—1:1 001—1:2 010—1:4 011—1:8 100—1:16 101—1:32 110—1:64 111—1:128

Updating the OPTION register in a low-end device is accomplished by the `option` instruction, which moves the contents of the `w` register into the `OPTION_REG` (which is the MPLAB label for the OPTION register). The OPTION register of mid-range devices can be written to using the `option` instruction, as in low-end devices, or by setting the RPO bit or the STATUS register and accessing it at address one. The address of the `OPTION_REG` is 0x081 or 0x001 in bank 1 of the address space.

Using the `option` instruction is not recommended in mid-range devices and may not be supported in them in the future. Instead, writing to the `OPTION` register directly from bank 1 always should be used; except for a few special cases, which I will explain below, the `option` instruction should never be used.

PARALLEL INPUT/OUTPUT

The most basic way of getting data in and out of the PIC microcontroller is via the parallel I/O bits that are located in the ports. In many PIC microcontroller's, these pins have peripherals behind them to provide advanced I/O capabilities. Despite this capability, in virtually every PIC microcontroller application that you create, the straight I/O port functions will be required.

The PIC microcontroller's typical I/O pin is capable of being either an input pin or an output pin. When in output mode, the pins are able to source or sink roughly 20 mA of current.

The block diagram of a PIC microcontroller I/O pin is shown in Fig. 9.19. Each register port is made up of a number of these circuits, one for each I/O bit. Depending on your previous experience, this I/O pin can look as if it is very complex, needlessly complex, or pretty basic. When you compare this controller with other microcontrollers, you'll find the PIC microcontroller's I/O pins are very typical of some devices and more complex than others are. Regardless of your own feelings about the I/O pins, there are a few aspects of the pins that you should be aware of.

I/O pins are associated with the bit number of the port to which they belong. The maximum size for an I/O port is 8 bits (or pins) for 1 byte, and you will find that for many devices there are ports with fewer than 8 bits. The convention used by Microchip is to label the pins according to their bit number and the with which port they're associated.

The convention is `R%#`, where `%` is the port letter (port A, port B, etc.) and `#` is the bit number of the port. Using this convention, `RB3` is port B, pin 3, and I will use it

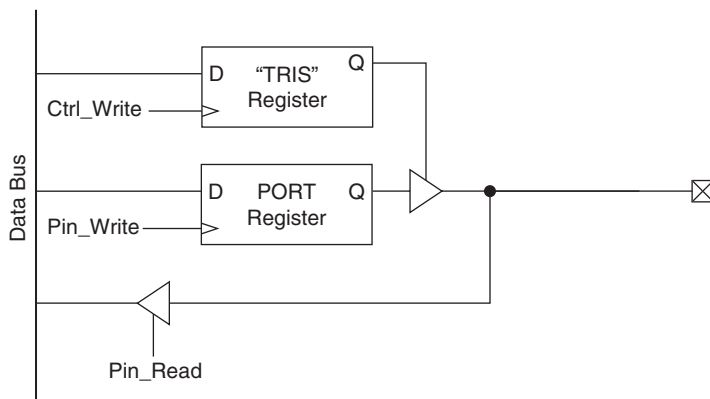


Figure 9.19 The PIC microcontroller's parallel input/output pin consists of a data bit, a direction control bit, and a feedback bit, returning the state of the output pin.

throughout this book to label individual pins. In some places you will see the convention `PORT%#`, which uses the same values for `%` and `#` as `R%#`.

The TRIS (tristate buffer enable) register is used to control the output capabilities of the I/O pin. When the register is loaded with a 1 (which is the power-up default), the pin is input-only (or in *input mode*) with the tristate buffer disabled and not driving the pin. When a 0 is loaded into a pin's TRIS bit, the tristate buffer is enabled (*output mode*), and the value that is in the data-out register is driven onto the pin.

The TRIS register can be confusing in regard to where it is located in low-end and mid-range PIC microcontroller register maps. In low-end devices, the TRIS register can be written to using only the `tris` instruction. In mid-range and higher PIC microcontrollers, the TRIS register is often in a different bank from the port data register, which requires you to change the bank or use the `tris` instruction. These two methods are somewhat awkward to use, and both have their own quirks.

The `tris` instruction has the format

```
tris  PORT #
```

where `#` is A, B, or C. When this instruction executes, the contents of the `w` register is loaded into the TRIS register for the specified port. If the TRIS value is going to be updated in the application, then I recommend saving the value loaded into the TRIS register into a file register and updating that (and saving it again) before executing. The `tris` instruction code to do this would be

```
movlw  0x0FF ^ (1 << Bit)    ; Find the Bit to Reset
andwf  TRIS#SaveReg, w       ; Update the Saved TRIS Value
movf   TRIS#SaveReg, w       ; Use as the TRIS Value
tris   PORT#                 ; Write to the TRIS Register
```

The mid-range parts can execute the `tris` instruction, but it is not recommended by Microchip because support for it is not guaranteed for all future products. Personally, I would not recommend it for use with mid-range devices simply because not all the possible registers can be controlled with this instruction. In mid-range PIC microcontrollers, there can be five I/O ports (PORTA through PORTE), the `tris` instruction can only access ports A, B, and C, which means that access to ports D and E is not possible.

The recommended way of accessing mid-range PIC microcontroller TRIS registers is to change the RPO bit of the STATUS register and read or write the register directly using the code

```
bsf    STATUS, RPO
movlw  NewTRISA
movwf  TRISA ^ 0x80
bcf    STATUS, RPO
```

Note in this code that I XORed the TRISA register address with 0x80 to avoid any messages telling me to check the page that is being accessed. TRISA is a bank 1 register and

has bit 7 set—its address value is 0x85. XORing 0x85 with 0x80 will return a value of 0x05, which is a valid address within a mid-range PIC microcontroller bank. Bank addressing is explained in greater detail in Chap. 6.

There is an important mnemonic to remember when you are setting or resetting a TRIS bit to put the register into input or output mode. Note that putting a port bit into output mode is accomplished by loading it with a 0, and putting it into input mode is accomplished by loading it with a 1. I always remember which is which by 0 = *output* and 1 = *input*. The digit approximates the first letter of the corresponding word.

Going back to Fig. 9.19, I want to point out that pin input is read from the I/O pin and not output of the data-out register. This is important to remember because there are occasions when this arrangement will cause problems. This issue is somewhat unique to low-end and mid-range PIC microcontrollers; other microcontrollers (including the PIC18) either have a separate input address or have a method of selecting which to read (the data-out register or the pin) based on whether or not the pin is in input or output mode. In Fig. 9.20 I show the organization of the PIC18 I/O pin with the LAT (or LATCH) register feedback showing the value stored in the PORT bit.

The problem lies in the fact that many instructions read and write the register contents in ways that you do not expect. Probably the biggest culprits are the *bcf/bsf* instructions, which perform a register read, bit OR or AND, and then write the new value back to the register. There is the opportunity that a PORT bit will be in input mode with a specific value set into it. If the pin is at a different value, then this value will be read and written back by the PIC microcontroller. The best way to avoid this problem is to always write the desired output values to the PORT register before changing the TRIS register.

I wish I kept track of problems that people report on the PICList; I would bet that the number one or two problem encountered by people is with RA4 or PORTA.4 pin (the block diagram for which is shown in Fig. 9.21). This pin is an open-drain driver

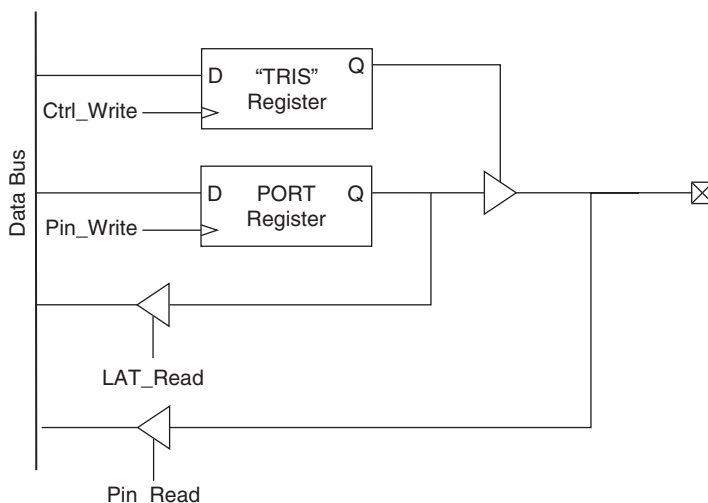


Figure 9.20 The PIC18 has a feedback register (LAT) that allows you to read back the contents of the PORT output latch.

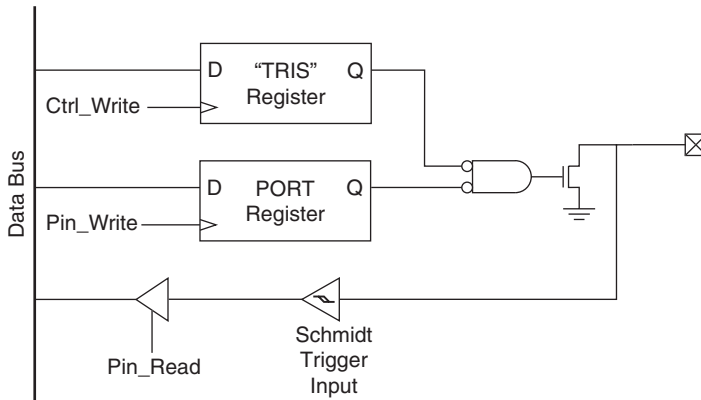


Figure 9.21 The RA4 pin is an open-drain output instead of the typical totem-pole driver.

that can be used with other open-drain or open-collector drivers to create a dotted-AND bus. When most people first use RA4 (I'm guilty of this as well), they forget or don't realize that the pin does not have the ability to drive a high voltage, and they can't understand why this pin seems to be broken. This pin cannot drive a positive voltage out unless it is pulled up (I normally use a 1 to 10 k Ω resistor depending on the input capacitance of what is being driven).

Another feature of some pins (including RA4) is that they have *Schmitt trigger inputs*, which have a different threshold voltage for signals going low to high than those for going high to low, as shown in Fig. 9.22. The purpose of the Schmitt trigger is to provide hysteresis for an input signal and to try to eliminate some incorrect bouncing errors. This input causes RA4 to behave differently than other I/O pins in some circumstances.

A feature that you should be aware of in mid-range parts is the availability of a controllable pull-up on the PORTB pins. This pull-up is controlled by the RPB_U bit of the OPTION register and is enabled when this bit is reset and the bit itself is set for output. The port B pin block diagram is shown in Fig. 9.23. The weak pull-up is approximately

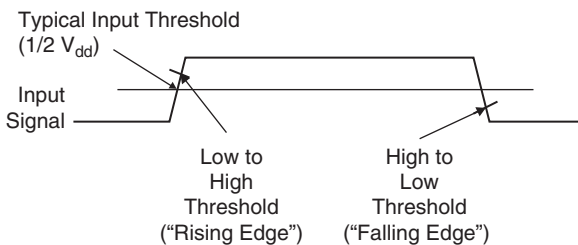


Figure 9.22 Schmitt trigger inputs have threshold voltages unique to the direction of the edge and closer to the destination voltage than typical inputs, which switch at one-half V_{dd} in both directions.

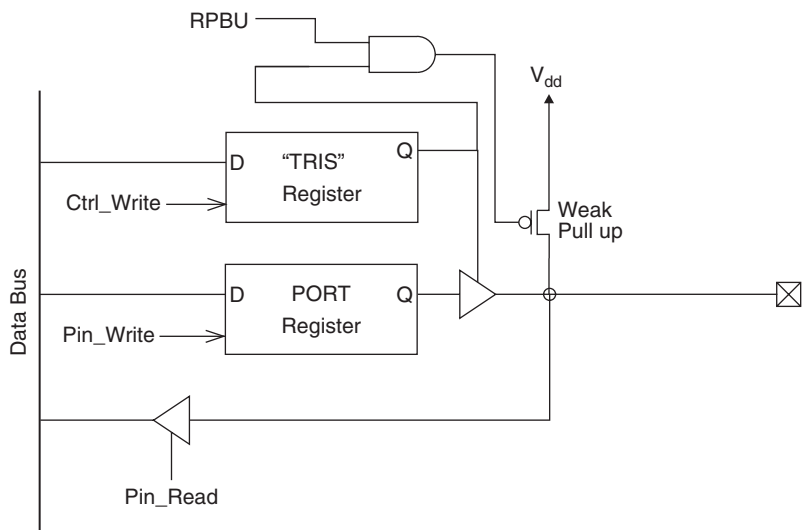


Figure 9.23 The weak pull-up built into PORTB pins eliminates the need for pull-up resistors.

50 k Ω and can simplify button inputs, eliminating the need for an external pull-up resistor.

Changing pin inputs can initiate interrupt requests to the processor. The function that is normally used is the RBO/INT pin, which can request an interrupt (if the `INTE` bit is set in the `INTCON` register). An interrupt request can be made on rising or falling edges as selected by the `INTEDG` bit of the `OPTION` register. If `INTEDG` is set, interrupts can be requested on the rising edge of signal into RBO/INT. `INTEDG` reset will cause an interrupt on a falling edge. I like to think of the RBO/INT interrupt request hardware as the block diagram shown in Fig. 9.24. While mid-range devices generally only have one interrupt request pin, PIC18 microcontrollers usually have several.

Once the interrupt is acknowledged by the processor, the `INTF` bit of the `INTCON` register has to be reset to enable another interruption RBO/INT pin transition. I should note that the interrupt request goes through a Schmitt trigger buffer, whereas the standard pin feedback does not. This will affect the loading of the pin somewhat.

The other type of interrupt that can be requested from the I/O pins is the port B *change on interrupt*. If the `RBIE` bit of the `INTCON` register is set, then any changes

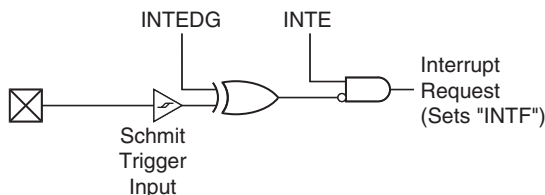


Figure 9.24 The RBO/INT pin can be used to generate interrupt requests.

to the RB4–RB7 pins while they are in input mode will request an interrupt on port change and set the RBIF flag in the INTCON register. To clear this interrupt request, PORTB has to be first read to set the current value that is followed by resetting the RBIF flag. The port change on interrupt is only available on RB4–RB7 when they are in input mode. Changing the state of any of these pins while they are in output mode will not cause a port change interrupt request.

This interrupt is a bit tricky to use, but by remembering that PORTB should never be polled, you can avoid any situations where an unknown interrupt request resets before it can be acknowledged. To avoid this problem, when I use the port change interrupt feature, I do not make any other PORTB pins inputs, and I never use an instruction that could read the I/O port (such as `movf`, `andwf`, `addwf`, `bcf`, `bsf`, etc.).

PIC12C50X AND PIC16C505 PIN ACCESSING

I really like PIC12C5xx parts and the PIC16C505, but the first time you use them, you probably will find that you cannot access all the I/O pins by default. This is due to the various built-in features of the chips, and you need to make sure that the internal features are all specified correctly before you attempt to use them in an application. Changing the internal hardware to allow these pins to be used to access external devices is a relatively simple operation. I've included the process here because there are pins in other devices that are also devoted initially to specific functions, and you will have to make sure that you understand how to figure out which default functions need to be changed before the pins they affect can be used.

In these parts, the built-in oscillator is selected by the `_IntRC_OSC` parameter of the `__CONFIG` statement. When the PIC microcontroller is programmed, a value for the calibration register (OSCCAL) has to be inserted. By convention, a

```
movlw OSCCAL_VALUE
```

is put in at the reset address and then at address 0 (when the program counter overflows and becomes 0); this value is saved into the OSCCAL register using the `movwf OSCCAL` instruction.

When choosing the programmer you are going to use for this part, make sure that it can handle the calibration value of the PIC microcontrollers with the built-in oscillator feature and that it can program the calibration value into the application. For EPROM-based parts, the calibration value should be read out using the programmer before erasure and then specified by the user separately from the application's .hex file during application programming. Programmers for Flash-based parts should read this calibration value before erasing the device and programming it back in as part of the programming operation.

Note that calibration values cannot be predicted. When I was doing the second edition of this book, I took a number of windowed PIC12C508s that I had bought together. They were all marked with the same date/lot code, and I found that their calibration values were

0xD0

0x90

0x30

0x80

0xA0

0x90

Developing your own calibration value could be done by trying values out against a calibrated PIC microcontroller and seeing which value has the minimum difference in timing with the calibrated device. However, it is much easier to record the factory calibration value when you first buy the part before you program it.

The internal reset is enabled by the `_MCLR_OFF` parameter of `__CONFIG`. This parameter disables the external (the E in `_MCLRE_OFF`) MCLR# pin and ties the PIC microcontroller's internal reset to Vdd. The MCLR# pin now becomes an input pin for the application.

Once the `_IntRC_OSC` and `_MCLRE_OFF` parameters are put into the `__CONFIG` statement and the `OSCCAL` value in the w register is saved, you will find that the pin that provides the clock input (GP2 in the PIC12C5xx) cannot be used for I/O. This is due to the reset value of the T0CS bit of the OPTION register being set, which causes the pin to be selected for TMR0 input. This overrides the I/O functions of the pin. Simply resetting the bit in the OPTION register will allow the pin to be used for normal I/O.

When I create a PIC12C5xx or PIC16C505 application, the initial code that I use is

```
__CONFIG _MCLRE_OFF & _IntRC_OSC    ; Add Application Specific
; "CP" and "WDT" parameters
org 0
movf  OSCCAL
movlw 0x0FF ^ (1 << T0CS)
option
; All I/O pins are NOW Available and Internal 4 MHz Clock is Running
; - Start Application
```

There are a few points to remember. First, note that the pin that can be used for MCLR# is only available for input. Second, this pin is not clamped with diodes inside the PIC microcontroller and probably is used for the Vpp pin, which means that high-voltage invalid inputs could reset the PIC microcontroller and put it into programming mode. Lastly, remember that any and all writes to the OPTION register must keep the T0CS bit at reset or the bit that can be used for TMR0 input may stop being I/O-capable.

TMR0

TMR0 is the basic 8-bit timer available to all PIC microcontrollers. While low-end PIC microcontrollers do not support interrupt requests from TMR0 (the other PIC microcontroller architectures do), TMR0 still can provide many useful functions for applications.

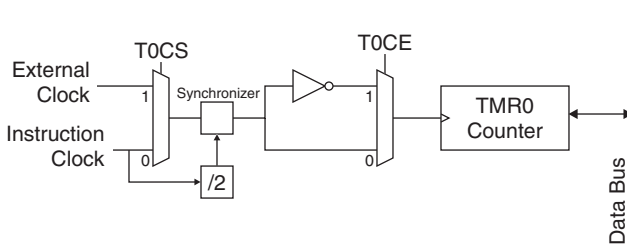


Figure 9.25 TMR0 is a basic 8-bit timer available to all PIC microcontrollers.

TMR0 has a few peculiarities with regard to its operation that you should be aware of, but for the most part it is quite straightforward to use, and I will present some example applications that use it later in this book.

TMR0 is an 8-bit incrementing counter that can be preset (loaded) by application code with a special value. The counter can be clocked by either an external source or the instruction clock. Each TMR0 input is matched to two instruction clocks for synchronization. This feature limits the maximum speed of the timer to one-half the instruction clock speed.

The TMR0 block diagram is shown in Fig. 9.25. The TOCS and TOCE bits are used to select the clock source and the clock edge that increments TMR0 (rising or falling edge). These bits are located in the OPTION register that was discussed earlier in this chapter. The synchronizer is a glitch-elimination feature of the PIC microcontroller. This circuit updates TMR0 only when two instruction cycles have passed without a change to the input. For most applications, this circuit means that TMR0 is updated after two instruction cycles have passed.

TMR0 can be driven by external devices through the T0CKI pin. The T0CKI pin is dedicated to this function in low-end devices (although in the PIC12C5xx and PIC16C505 microcontrollers the pin can be used for digital I/O). In the other PIC microcontroller architectures, the pin also can be used to provide digital I/O. When a clock is driven into the TMR0 input, the input is buffered by an internal Schmidt trigger to help minimize noise-related problems with the input.

Input to TMR0 can be made with or without the prescaler, which provides a divide by feature to the TMR0 input. As will be discussed later in this chapter, the prescaler can count anywhere from 1 to 128 cycles before passing along an increment signal to the PIC microcontroller.

For low-end and mid-range PIC microcontrollers, TMR0 is located at register address 0x01. The contents of TMR0 can be read from and written to directly. One thing to always remember when TMR0 is being updated is that the synchronizer (and prescaler, if connected) will be reset.

As I will show in experiments, if TMR0 is polled for being equal to 0 using the code

```
movf    TMR0, f
btfss   STATUS, Z
goto    $ - 2
```

and a 4:1 or greater prescaler is used with TMR0, it will loop forever. Even if the prescaler is not used, you will find that the delay will be twice as long as you would expect because the `movf TMR0, f` instruction will reload TMR0 and reset the synchronizer. This mistake is very common, and I know of many people who have not quite believed the source until they simulated it and it actually was happening.

When polling TMR0, the code

```
movf  TMR0, w      ; Load "w" with the contents of TMR0
btfss STATUS, Z    ; Skip if the Result is Equal to Zero
goto  $ - 2
```

should be used. It may seem unusual to poll TMR0 for reaching 0 because the timer is incremented with each synchronized clock tick instead of checking it against a specific value, but this is the conventional way of checking the result. In mid-range and higher PIC microcontrollers, a TMR0 interrupt is requested when TMR0 rolls over or overflows from 0xFF to 0x00, so this method fits in well with the interrupt request.

Note that `bcf` and `bsf` instructions, as well as any others that put data into TMR0, will reset the synchronizer (and prescaler, if selected) and can affect the operation of your application in unexpected ways.

In mid-range and higher PIC microcontrollers, when TMR0 transitions from 0xFF to 0x000, an interrupt can be requested. This is accomplished by loading TMR0 to the specified delay value and then setting the TOIE bit and resetting to 0x00; TOIF will be set, and if GIE is set, the interrupt request will be acknowledged.

Anytime TMR0 transitions to 0x00 from 0xFF, the TOIF flag will be set. This flag must be reset explicitly before enabling TMR0 interrupts to prevent the chance for a spurious interrupt when the TMR0 interrupt is enabled and TMR0 has already overflowed. The TOIF (TMR0 interrupt request flag) must be reset in the interrupt handler. It is not reset automatically when the interrupt request is acknowledged.

TMR0 DELAYS

One of the basic operations of a timer is to provide a set time delay. In many of my experiments and applications, I often use TMR0 to request an interrupt after a specific time delay. It isn't difficult to use a timer to calculate a delay, although there are a few things you should be aware of before you attempt it.

The first thing that you should recognize is that the interrupt is requested when TMR0 overflows or equals 256. 255 or 0xFF is the largest value that can be saved in the bit timer register. 0xFF does not cause an interrupt (although an interrupt will be requested when the timer increments or overflows to 256 or 0x100). Therefore, the delay is calculated for the time when the timer equals 256 and not 255.

The initial timer value must be the number of clock increments (or *ticks*) required to get to 256 within a specific period of time because the timer can only count up. To calculate the initial value (with no prescaler, which will be described later), the following formula is used:

$$\text{TMR0 initial} = 256 - \text{delay cycles}$$

The delay cycles value is found by taking the delay time, dividing by the frequency, and multiplying by 4:

$$\text{Delay cycles} = (\text{delay time} * \text{frequency})/4$$

If the prescaler is specified for TMR0, and even if it is left at 1:1, then each clock cycle is divided by 2 before it is passed to TMR0. This changes the formula for delay cycles to

$$\begin{aligned}\text{Delay cycles} &= [(\text{delay time} * \text{frequency})/4]/2 \\ &= (\text{delay time} * \text{frequency})/8\end{aligned}$$

These formulas can be used to calculate a 160- μ s delay in a 3.58-MHz PIC microcontroller application. Assuming that the prescaler is not assigned to TMR0, the first formula for delay cycles can be used, and the number of delay cycles timed by is calculated as

$$\begin{aligned}\text{Delay cycles} &= (\text{delay time} * \text{frequency})/4 \\ &= (160 \mu\text{s} * 3.58 \text{ MHz})/4 \\ &= [160(10^{-6}) \text{ s} * 3.58(10^6) \text{ 1/s}]/4 = (160 * 3.58)/4 \\ &= 143.2 \\ &= 143\end{aligned}$$

The result is rounded to the nearest whole number so that it can be stored in TMR0.

This value is used to calculate the initial timer value using the first formula

$$\begin{aligned}\text{Initial} &= 256 - \text{delay cycles} \\ &= 256 - 143 \\ &= 113\end{aligned}$$

Therefore, to wait 160 μ s before a TMR0 overflow interrupt in a PIC microcontroller running at 3.58 MHz, an initial value of 113 must be loaded into TMR0.

For longer delays than 256 instruction cycles, the prescaler can be used to divide the number of cycles input into the timer. The prescaler, as will be discussed in the next section, divides the incoming data by powers of 2 from 2 to 256.

When calculating the delay, the delay cycles are halved continually until the value is less than 256. For example, if a delay of 5 μ s were required using the original formula in a PIC microcontroller running at 4 MHz, the delay cycles would be calculated as

$$\begin{aligned}\text{Delay cycles} &= (\text{delay time} * \text{frequency})/4 \\ &= (5 \text{ ms} * 4 \text{ MHz})/4 \\ &= [5(10^{-3}) \text{ s} * 4(10^6) \text{ 1/s}]/4 \\ &= 5(10^3) \\ &= 5,000\end{aligned}$$

Since the calculated delay cycle is greater than 256, it is continually divided by 2 to get an appropriate prescaler divisor. Dividing by 2, the delay count would be 2,500, which is still greater than 256. Dividing by 2 again, delay count would be 1,250, which is also greater than 256. Dividing the original value by 8 yields a delay count of 625, which is still greater than 256, as does dividing by 16 (which results in 312.5). Finally, dividing the original by 32 yields a delay count of 156.25. Rounding it off, the value can be put into the initial value formula:

$$\begin{aligned}\text{Initial} &= 256 - 156 \\ &= 100\end{aligned}$$

Now, the rounding that I did (taking off 0.25) will result in a difference of four cycles between the actual delay of 5 ms and the delay calculated within TMR0. If this is a critical loss, then I suggest that you do one of two things. The first is to add four delay cycles to your code, which can be four nops, a dummy call, or even delaying loading the timer by four instruction cycles.

THE TMR0/WATCHDOG TIMER PRESCALER

The prescaler is a power of 2 counter that can be selected for use with either the watchdog timer or TMR0. Its purpose is to divide the incoming clock signals by a software-selectable power of 2 to allow the 8-bit TMR0 to time longer events or increase the watchdog delay from 18 ms to 2.3 s.

The prescaler’s operation is controlled by 4 bits contained within the OPTION register. PSA selects whether the watchdog timer uses the prescaler (when PSA is set), or TMR uses the prescaler (when PSA is reset). Note that the prescaler has to be assigned to either the watchdog timer or TMR0; fortunately, both functions are able to execute with no prescaler or with the prescaler’s delay count set to 1, which results in no delay.

The prescaler itself is a power or 2 counter, with the trigger point selected by the PSO, PS1, and PS2 bits of the OPTION register. I call prescaler operation a power of 2 because the number of cycles delay is a function of the PSA bit value. Table 9.10 shows

TABLE 9.10 PRESCALER VALUES TO DELAYS		
PS2:0	PRESCALER (WDT) DELAY	TMR0 DELAY
000	1 cycle (1:1)	2 cycles (1:2)
001	2 cycles (1:2)	4 cycles (1:4)
010	4 cycles (1:4)	8 cycles (1:8)
011	8 cycles (1:8)	16 cycles (1:16)
100	16 cycles (1:16)	32 cycles (1:32)
101	32 cycles (1:32)	64 cycles (1:64)
110	64 cycles (1:64)	128 cycles (1:128)
111	128 cycles (1:128)	256 cycles (1:256)

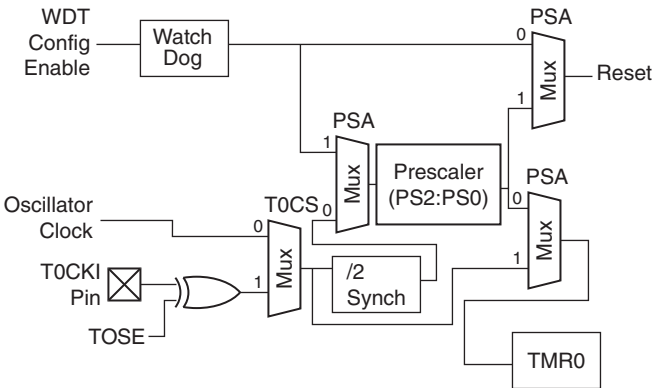


Figure 9.26 The TMR0/WDT prescaler can be devoted to either device.

the prescaler cycle delay for varying values of the PS# bits. In this table I noted that the prescaler delay is double when it is applied to TMR0.

For PS2–PS0 equal to 011, one of every eight clock cycles input into the prescaler will be passed to the device the prescaler is driving.

The watchdog timer and TMR0 block diagrams can be combined with the prescaler to show how the functions work. This is shown in Fig. 9.26 and shows the synchronizer (which divides the TMR0 input by 2) in place as part of the path to the prescaler.

Interrupt Operation

In mid-range PIC microcontrollers, the INTCON register is the focal point for interrupts. This register (defined in Table 9.11) is used to globally enable interrupts, as well as to control the response to different interrupt inputs. The register consists of four different bit types and is located at address 0x0B in all active register banks. The bit usage is similar for all mid-range devices.

The GIE bit must be set for interrupt requests to be passed to the processor. This bit can globally mask or allow (unmask) interrupt requests going to the processor. If a critical section of code is being entered, by resetting this bit, the interrupt request will be ignored until GIE is set, allowing the code to execute without being interrupted.

The INTCON bit names the grounds in E that are the interrupt enable flags. When these bits are set, any incoming interrupt requests will set the corresponding interrupt request active flags, which have a bit name that ends in F. The request active flag must be reset in hardware and is not reset automatically by the operation of the interrupt acceptance (which I also call *acknowledgment* elsewhere in this book). In addition, the requesting hardware may have to be reset before the F (request active flag) can be reset.

Using the block diagram in Fig. 9.24 as a model, if the E bit for a particular interrupt source is set, then interrupt requests, which have set their appropriate F bits, will request an interrupt of the PIC microcontroller processor if the GIE bit is set.

TABLE 9.11 MID-RANGE PIC MICROCONTROLLER INTCON REGISTER DEFINITION	
BIT	DESCRIPTION
7	GIE—global interrupt enable
6	Device-specific
5	T0IE—TMR0 overflow interrupt enable
4	INTE—RBO/INT pin interrupt enable
3	RBIE—PORTB input change interrupt enable
2	TOIF—TMR0 overflow interrupt request active
1	INTF—RBO/INT pin interrupt request active
0	RBIF—PORTB input change interrupt request active

When the processor receives the interrupt request, it completes the current instruction before jumping to the interrupt vector. Instruction execution in the PIC microcontroller can be one or two cycles long, and when added to the two-instruction delay for calling the interrupt handler, the total delay (which is known as *interrupt latency*) is three or four instruction cycles. In mid-range devices, the interrupt vector’s address is 0x0004 for all interrupt sources. What happens during execution is shown in Fig. 9.27.

As was noted earlier, when the processor jumps to the interrupt vector, the GIE bit is reset. The return address (the address of the instruction *after* the interrupted one) is saved on the program counter stack. With GIE reset, no subsequent interrupts can interrupt the interrupt handler’s execution.

Because of the lack of a data stack in the PIC microcontroller’s processor, I do not recommend that the GIE bit be set within the interrupt handler. Instead, it should be set by the `retfie` instruction after completing servicing of the current interrupt. If the timing delay of restoring the context registers, executing `retfie`, jumping back to the interrupt

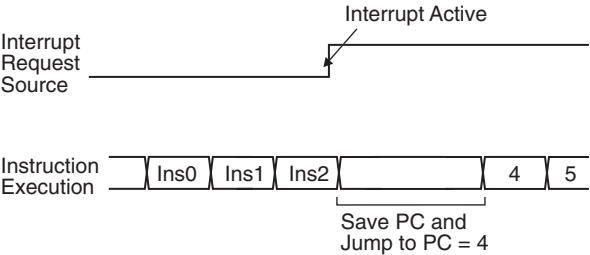


Figure 9.27 The PIC microcontroller responds to an active interrupt signal by first storing the address of the next instruction to execute on the program counter stack.

handler, and then storing the context registers again is too long, then I recommend that you poll the interrupt request flags before executing the `retfie` and jump to the appropriate service routine rather than returning to the mainline execution.

I should point out that this section is hardware-centric. Back in Chap. 7 I discussed the assembly-language code required for saving and restoring the context registers for the various PIC microcontroller families.

Interrupts will execute and return properly from any instruction or instruction combination. At one time there were reports that mid-range interrupts would have problems if they were acknowledged after a PCL update, but this is not true; there are no PIC microcontroller hardware deficiencies on which interrupts can execute. The only issue will be the application software.

In mid-range parts, there are three interrupt sources that are handled from within the INTCON register. The TOIF and TOIE bits allow an interrupt request when TMR0 overflows (equals 255 or 0x0100). The second responds to an input on the RBO (usually marked RBO/INT by Microchip), when the input on the pin goes high or low (depending on the STATUS register state).

The last interrupt source, PORTB change interrupt, requires some comments because there is some confusion about the operation of this interrupt. The port B change interrupt will request an interrupt if any of the RB7–RB4 pins (which are in input mode) change state. The pins will not request an interrupt if they are in output mode. To reset the interrupt, port B must be read to set the input state before resetting the RBIF flag. If this is not done, the RBIF flag will remain set and request an interrupt, no matter how you try to reset it.

Along with these three interrupt sources, in many PIC microcontrollers there are a number of built-in peripherals that can request interrupts to allow these additional sources. Bit 6 can be used as another E bit (with the F bit in another register) or an enable bit for the PIE and PIR registers. There may be one or two sets of PIE and PIR registers, according to the features built into the PIC microcontroller. Unfortunately, there doesn't seem to be any convention for how the bits are set. But by looking at the PIC microcontroller device datasheet, you will get a bit specification with the interrupt bits.

The Right PIC Microcontroller to Learn On

Before leaving this chapter, I wanted to make a few comments about which PIC microcontroller is the right one for somebody to learn on. If you have read other books for the PIC beginner, you will see that they discuss starting out with the PIC16C54, PIC16F84(A), PIC16F684, etc.; the thing that all of them have in common is that they all focus on one part number for you to work with initially. Similarly, new developers are encouraged to work with one development environment and one set of software development tools, as well as only a single operating voltage (normally 5 V) and methods of wiring the chosen PIC microcontroller into an application. I do not believe that this is

the appropriate method for everybody learning the PIC microcontroller, and as you work through the examples in this book and start on your own applications, you should think about whether or not these approaches are correct for you.

The problems with only working with one device center around the lack of sophisticated built-in I/O peripheral functions. With the typical PIC microcontroller part number that is recommended for learning, the advanced peripherals (analog-to-digital converters, UARTs, I2C busses, etc.) are simply not present. This does not mean that the functions that these features provide are not implemented, just that they are implemented using the basic I/O pins and operating them in a manner for which they aren't designed (as the saying goes, "If all you have is a hammer, everything looks like a nail"). Along with this, you will find that the peripheral functions are multiplexed along with peripheral functions on pins that also can function as digital I/O pins (it isn't unusual to see PIC microcontroller I/O pins being usable for four different functions). Thinking of the case of analog-to-digital converter inputs, some pins are selected as analog inputs even though the ADC function is not enabled; for these parts, either the analog input is turned off or the ADC is turned on and used to measure voltages at the different pins. By learning the PIC through the use of multiple devices, you gain greater insight into how the different peripherals work and how they interact with other functions and capabilities of the basic I/O pins.

If you are new to microcontrollers, electronics, and programming, then a device with multiple functions on each I/O pin will be overwhelming, and this would be the case when starting with a basic device that does not have any advanced functions on the I/O pins. The important thing to remember in this case is that you should not be using the digital I/O pins for anything other than digital I/O. If analog or other standard peripheral functions are required for the application, then you should move to a PIC microcontroller that has this function built in and learn how to work with it.

While I am strongly recommending that you should only work with the MPLAB IDE, I believe that you should experiment with different languages and programmers to find out which tools are best for you. Don't limit yourself to just assembly language; spend some time with the free versions of the high level languages to see which one you feel comfortable with. Modern PIC microcontroller compilers produce remarkably efficient code that, as a new developer, you will be hard-pressed to improve on.

The question of voltage levels is one that is gaining more importance. Even with the second edition of this book, 5-V logic was standard for many products, as well as for educational kits. Today, 3.3 V is becoming the logic voltage of choice, with 5 V being somewhat of an anachronism. Along with 3.3 V becoming the new standard, for microcontrollers, 2 V (using a single lithium battery or two discharged alkaline batteries) is not that unusual. The newer PIC microcontrollers support 3.3 V, and many of them work through a range of 2.0 to 6.0 V. You should not lock yourself into 5.0 V but instead be prepared to handle voltages through a wide range and assume that you will be powered by batteries as well as an external power supply.

Another aspect of the power-supply question is the voltage used for programming and the process used for burning the application onto the PIC microcontroller. The MPLAB

ICD 2 can work with a variety of different voltages, allowing you to leave the PIC microcontroller in circuit during programming. Something that you should be aware of is that the efficiency and life of the programming operation increase with voltage. This means that if you have an application that is powered by a couple of AA cells, then you might want to use an external programmer, such as the PICStart Plus, that programs the chips at 5 V.

The real question that you should be asking is whether or not you are learning about the PIC microcontroller or learning about microcontrollers with the PIC MCU being the first device you work with. There is a subtle and important difference in the two requirements. If you are familiar with microcontrollers, programming, and electronics, then you will want to get exposure to as many different PIC microcontroller part numbers as possible so that you can understand how the different devices and their peripherals work, as well as getting exposure to different operating situations. If you are new to microcontrollers, their programming, and the electronics needed to run them, then you will want to stick with one device (I would suggest the PIC16F690 or PIC18F1320) running at 5 V to allow easy interfacing with other chips, as well as the widest possible variety of (free) development tools that you can try out and learn what is best for you. Regardless of your skill and knowledge level, you should be working toward understanding all the features that are available to you so that you can develop the best possible applications.

SIMPLIFYING YOUR FIRST APPLICATIONS

Regardless of the PIC microcontroller that you are using, the voltage level of the circuit, and the tools that you are using to develop the code, there are a few rules that you should follow when you are creating your first applications to make it easier on yourself and help to guarantee success (or, at the very least, some kind of powering up so that you can see what is going on):

- 1** Use a PIC microcontroller that has Flash program memory. Looking down the Microchip linecard, this suggestion may seem unnecessary, but there are a number of EPROM-based devices still available and, more important, a number of books and Internet references that have projects and experiments designed for these parts.
- 2** Unless there is a hard reason for using an external crystal or ceramic resonator, you should only select parts that have an internal clock. Reasons for having an external oscillator include applications that provide real-time clock functions or USB applications.
- 3** The internal reset (MCLR) and programming LVP, Data, and Clock pins should not be enabled or used for any other purpose to allow the use of MPLAB ICD 2 for programming in circuit or debugging.
- 4** Choose a PIC microcontroller that runs on from 2.0 to 6.0 V, and when programming in circuit, provide application power to the PIC microcontroller and do not depend on the programming power of the MPLAB ICD 2.
- 5** Make sure that all the I/O pins are high current and capable of sinking or sourcing at least 20 mA. Recently, there have been a number of PIC microcontroller part numbers

released that have a number of low-current (maximum 6 mA) I/O pins built into them. The low-current pins are much more easily damaged than the high-current versions.

- 6 Select a PIC microcontroller part number that has a free compiler available for it so that you can work through the PIC MCU features without drowning in the minutiae of the assembly-language operations. Regardless of whether or not you have programmed microcontrollers before, there is still a lot to learn before you can start developing your own applications.

MACRO DEVELOPMENT

Since writing the first edition, I have discovered that some of the biggest questions people have about developing PIC® microcontroller software applications are about how macros work and how they are implemented. This was surprising to me because I have always felt that macros are a very simple tool to understand and use. They are a feature of most assemblers (including MPASM), and they can be used to make application development much easier.

A *macro* can be defined as a function that is “invoked” from within an application’s code. The invocation statements are replaced by the code within the macro. Macros can be thought of as being similar to subroutines, but they have one big difference: Instead of calling a central routine, the routine’s code is embedded into the source, as shown in Fig. 10.1.

PIC Microcontroller Assembly-Language Macros

When macros are *invoked* (which is the term used to indicated their use, like calls for subroutines), parameters or arguments are used that replace labels within the macro. These parameters are not similar to the arguments of a function call—the macro parameters are used to replace the macro arguments, and their contents are not copied from a variable into another one, as is done in a function call. As will be shown later in this chapter, the invocation parameters do not have to be variables; instead, they are strings, which allows for some interesting capabilities.

For example, instead of repeatedly writing the code

```
incf    A, f  
btfsc   STATUS, Z  
incf    A + 1, f
```

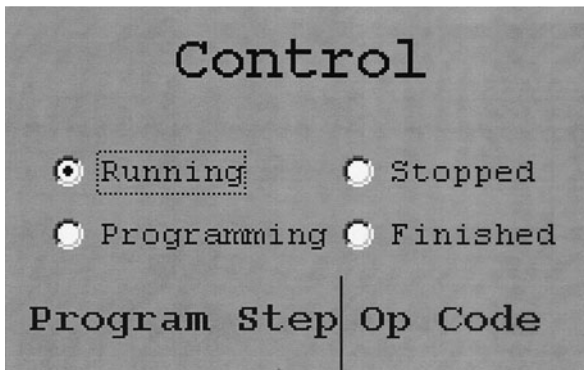


Figure 10.1 Macros differ from subroutine calls by inserting the code to be executed at the current location in the application code rather than jumping to a new address, executing the required instructions, and then returning.

every time a 16-bit increment is required, the following macro could be used:

```
incl6    macro Variable
incf     Variable, f
btfsc    STATUS, Z
incf     Variable + 1, f
endm
```

Four strings have been added to the three 16-bit incrementing codes. After the identifying label `inc16`, `macro` indicates that the following code is to be placed in line each time the identifying string is encountered until the `endm` directive is encountered. The `Variable` is a parameter that can be used to customize the macro code that is inserted into the application—in this macro the parameter `Variable` will be replaced by whatever string you specified for it. If the macro was invoked with the statement

inc16 B

the instructions

```
incf    B, f
btfsc   STATUS, Z
incf    B + 1, f
```

would be inserted into the application.

The advantages of using macros are that the overhead of subroutines can be avoided. The immediate overhead that comes to mind is the `call` and `return` instructions, but parameters that are passed to the routine do not have to be saved. This leads to some interesting situations where macros can create much more efficient application code than calling a subroutine. For example, to add two 16-bit numbers together, if a subroutine were used, the

parameters have to be saved in temporary variables that are processed by the subroutine, and then the result is returned in a temporary variable and then passed to the final destination.

In mid-range PIC microcontrollers, the code to do this could be

```

;   C = A + B

movf    A, w           ;   Save "A" in a Temporary Register
movwf   Temp1
movf    A + 1, w
movwf   Temp1 + 1
movf    B, w           ;   Save "B" in a Temporary Register
movwf   Temp2
movf    B + 1, w
movwf   Temp2 + 1

call    Add16Bits      ;   Call the 16 Bit Addition Subroutine

movf    Temp1, w       ;   Put the Result in "C"
movwf   C
movf    Temp1 + 1, w
movwf   C + 1

:

Add16Bits      ;   16 Bit Addition Subroutine
movf    Temp2 + 1, w   ;   Add the High Byte First
addwf   Temp1 + 1, f
movf    Temp2, w       ;   Add the Low Byte
addwf   Temp1, f
btfsc   STATUS, C
    incf Temp1 + 1, f   ;   If Low Byte Carry, Increment High Byte
return   ;   Return to Caller

```

In this example, the actual number of instructions required for supporting the routine is almost twice what the subroutine uses!

In comparison, a macro for the same function, i.e.,

[illegible]

requires just one more instruction for the general case (and one less if the destination is the same as one of the source variables) and requires none of the instructions used to pass data between the temporary registers and the parameters. In this example, an application written that uses this macro instead of the general subroutine always will be more efficient.

An added advantage of a macro over a subroutine is that if the macro is made available and not used, then it will not take up any program memory. If you write a subroutine, it is available within the program, whether it is called or not.

One point that may not be terribly obvious is that macros execute during “build time” and not at run time. This means that macros execute when the source code is assembled and not when it is executing in the PIC microcontroller. If you place variables as macro parameters, the variables’ addresses will be used and *not* the contents of the variables themselves. This point is important and trips many people up when they first start working with macros.

The Difference Between Defines and Macros

I consider *defines* to be simple macros in their own right because of the way they operate. This perspective is slightly unusual, but I believe that it is the most accurate way to visualize how defines work and helps you to find good uses in a variety of different situations for which they may not be considered in the first place. Defines are often just as flexible, if not more so, as macros; along with providing a great deal of flexibility, they also can add significantly to the ease of application programming and application readability.

Defines in most languages are declared using the format

```
#Define Label String
```

The `Label` is a standard alphanumeric text label. What makes defines special is that when the label is encountered, it is replaced by a string. Unlike a macro text, the define string replaces the label that it encounters and not the entire line (or more, depending on the macro). This is a powerful concept and one that can make programming a lot easier. When used in PIC microcontroller assembly language, defines can be used to simplify the operation setting, resetting hardware bits and flags in your application.

For example, if bit 3 of the `Flag` variable is used to indicate the ready state of the serial port, it could be declared as

```
#Define Serflag Flag, 3
```

to reset the bit in the serial initializing routine instead of using the statement

```
bsf     Flag,3
```

which is not very helpful or easy to understand without a reference to how the pins are organized. In comparison, the same instruction with the `Serflag` define, i.e.,

```
bsf      Serflag
```

is much easier to understand without the aid of documentation or even descriptive comments. The single define eliminates remembering (or looking up) where the bit is located and which bit it is. When reading the code, using the `define` directive enhances the readability of the purpose of the instruction over the actual source information.

Defines work because they directly copy in the string and let the assembler evaluate it. This is different from `equates`, which evaluate a string and store it as a constant referenced to the label. This can cause some subtle differences that can be a problem if you don't know what you are doing.

For example, if you had the code

```
variable A = 37                ; "Variable" is a run time variable

Test1 equal A * 5              ; Test1 = 185

#define Test2 A * 5            ; Test2 is Evaluated when it is used

:

A = A + 5                      ; A = 42

:

movlw    Test1                  ; "Test1" is replaced with 185

:

movlw    Test2                  ; "Test2" is replaced with A * 5
                                ;   = 42 * 5
                                ;   = 210
```

in this case, even though `Test1` and `Test2` are declared at the same point in the code identically, they are evaluated differently and will be different values in different locations of the application. This is a very useful capability, but one that can be a problem elsewhere.

In the sample code above, I declared the `variable` to be `A`. `Variable` is an MPASM directive to create a temporary storage value during the application's assembly and will be discussed in the later sections of this chapter. When it is used in `Test1`, the value of `A` when the assembler processor encounters the statement is used, multiplied by 5 and assigned to `Test1`. After `Test1` and `Test2` are defined in the code, `A` is modified, which results in a different value being calculated for `Test2` when it is used later in the application.

A useful function that defines can provide is that they can provide constant strings throughout an application. I use this ability to keep track of my "version number" of an

application. In many of the example application programs presented in this book, you'll see that the second or third line of an application is

```
#define _version "1.00"
```

Because it is at the top of the source, I can see the version number as soon as the source code is brought up in an editor. This define can be used throughout the code to provide the version information without me having to update when I come up with a new version. Often I will output the version information to indicate what source code is burned into the PIC microcontroller. This `#define` statement above can be put into a `dt` statement and read out of a table using conventional table read code:

```
dt    "Version: ", _version, 0
```

In this code, each byte of the `_version` define is added to the string of characters used with the `dt` statement as if the code was entered as

```
dt    "Version: 1.00", 0
```

Define labels do not have to have strings associated with them. This may seem unusual (and seem to defeat the purpose of defines), but this allows them to be used as assembly-time execution control flags that I will show later in this chapter. This function allows fast customization of an application modification for debug.

Defines can be used for many different purposes. While macros can be used only for replacing full lines of text, defines can simplify instruction programming and provide common information or text strings. Neither macros nor defines can replace each other, but their functions are complementary and allow you to create richer, more easily programmed and understood applications.

The Assembler Calculator

While not really part of the macro processor, the ability of the MPASM assembler (and most compilers) to do constant calculations during assembly or compilation can make applications easier to develop and avoid your having to spend time with a handheld calculator working out the constant values needed in your application. The assembler calculator provides capabilities for both your macros and regular programming that can be taken advantage of in a variety of situations.

The assembler calculator works on algebraic equations, similar to how they're used in high level languages. It is important to remember that the calculator works as part of the last pass of the assembler—to allow the insertion of data generated during the build cycle, such as the address of an application variable. This can be confusing because variables available for use by the assembler calculator are declared within the source in a manner similar to that of variables used in the application.

So far in this book you have seen the assembler calculator in operation calculating constant-value arguments for instructions such as

```
movlw    (1 << GIE) | (1 << T0IE)
```

This instruction loads w register with a byte, destined for the INTCON register, which has the GIE (7) and T0IF (5) bits set. In this case, the assembler calculator is used to change bit numbers to actual values to be loaded into a byte. The trick in this statement is knowing that shifting one by the bit number converts the bit number into a constant value that will set the bit when loaded into a register.

This is useful and avoids having to figure out what value is used for specific bits being set. In the preceding example, if this trick had not been used, I would have to remember (or generate on a calculator) that bit 7 being set is the same as adding 128 and that bit 5 being set is the same as adding 32. The result of these two values is 160 decimal or 0x0A0. Using the assembler calculator, I didn't have to worry about any of this.

To reset specific bits, the same trick can be used, but the bits have to be reset, which is done by a bitwise inversion of the bits and then ANDing the result with the current value. XORing the set bit value with 0xFF accomplishes the bitwise inversion. For example, to clear bits 4, 2, and 1 in the w register, the following instruction could be used:

```
andlw    0xFF ^ ((1 << 4) | (1 << 2) | (1 << 1))
```

If you were to do this manually, you would have to follow these steps:

- 1 Calculate the values for bits 4, 2, and 1 being set:

```
(1 << 4) = 16
(1 << 2) = 4
(1 << 1) = 2
```

which translates to

```
(1 << 4) | (1 << 2) | (1 << 1) = 16 | 4 | 2
                                = 22
                                = 0x016
```

- 2 Calculate the inverse (XOR with 0xFF):

```
0xFF ^ 0x016 = 0xE9
```

- 3 Put the value into the andlw instruction:

```
andlw 0xE9
```

If you go through the manual process, you can see that there are more than seven opportunities for you to calculate constant values incorrectly or copy down the wrong value. Avoiding these manual calculations with their inherent opportunities for error is

TABLE 10.1 OPERATORS AVAILABLE TO THE ASSEMBLER CALCULATOR		
OPERATOR	DESCRIPTION	COMMENTS
+	Addition	
—	Subtraction/negation	If no first parameter, then negation
*	Multiplication	
/	Division	
%	Modulus	Return remainder from divide operation
<<	Shift left	Shift the first parameter to the left by second parameter number of bits
>>	Shift right	Shift the first parameter to the right by second parameter number of bits
&	Bitwise AND	AND together the parameter's bits
	Bitwise OR	OR together the parameter's bits
^	Bitwise XOR	XOR together the parameter's bits
~	Complement	XOR the parameter with 0xFF to get the complemented or "inverted" value
LOW	Low 8 bits of a constant value	AND the parameter with 0xFF
HIGH	High 8 bits of a 16-bit constant value	AND the parameter with 0xFF00 and then shift right 8 bits
\$	Current address	Current address of the instruction

what I meant when I said the assembly calculator is easier and less prone to mistakes. Table 10.1 lists the calculator's arithmetic operators. All the operators have two parameters, except for when “—” negates a value or the complement (“~”) operator, which only have one parameter.

In the clear bits example, I could have used the equation format

```
andlw      ~((1 << 4) | (1 << 2) | (1 << 1))
```

instead of adding the 0x0FF ^ characters in the preceding instruction.

For 16-bit values, you can use the “low” and “high” assembler directives. For example, if you wanted to jump to a specific address in another page, you could use the code

```
movlw      HIGH Label          ; "Label" is the
movwf      PCLATH              ; Destination
movlw      LOW Label
movwf      PCL
```


which is the same as

```
movlw    ((label && 0xFF00) >> 8)
movwf    PCLATH
movlw    LABEL && 0xFF
movwf    PCL
```

In this example, the function of the first four instructions (which use **HIGH** and **LOW**) is much clearer to somebody reading the code than the second four instructions, which require the reader to evaluate what the arithmetic operations are doing.

As has been discussed earlier in this book, the **\$** operator returns the current program counter, which is a 16-bit value that can be manipulated using the assembler calculator's operators as if it were a constant.

Along with the arithmetic operations, parentheses (the “(” and “)” characters) can be used in the expressions to make sure that the operation is executed in the correct order. In the preceding examples I have used parentheses to make sure that the correct order of operations takes place for these instructions.

Variables that are only used in assembly can be declared using the format

```
variable label [ = constant ][ , ... ]
```

These variables are 32 bits in size and can be set to any value using the operators listed above and employing the “=” operator to make an assignment statement such as

```
LABEL1 = LABEL1 * 2
```

It is important to remember that the label is not an application variable (i.e., it cannot be modified by the PIC microcontroller as it is running), and when it is assigned a new value, it must be in the first column of the assembly-language source. When it is being read in another statement, it can appear in any column (except for the first) in the line.

Taking a cue from C, assembler variable assignment statements can be simplified if the destination is one of the source parameters. These operations can be confusing to use and read unless you are familiar with C. Table 10.2 lists the combined assignment statements.

The assembler calculator also can do comparisons between two parameters using the operators listed in Table 10.3. If the comparison is true, a 1 is returned; otherwise, a 0 is returned. The comparison operators are required for the “conditional” assembly operations presented in the next section.

These comparisons can be compounded with **|** and **&&**, which are the logical **OR** and logical **AND** operators, respectively. **|** returns 1 if either of its two parameters are not equal to 0. **&&** returns a 1 if both parameters are equal to 0. This operation brings up an important point: In the assembler calculator, a “true” condition is *any* nonzero value. The variable **A**, after executing

```
A = 7 && 5
```

**TABLE 10.2 COMBINED ASSIGNMENT OPERATORS
AVAILABLE TO THE ASSEMBLER CALCULATOR**

OPERATOR	EQUIVALENT OPERATION
<code>+=</code>	<code>Parm1 = Parm1 + Parm2</code>
<code>-=</code>	<code>Parm1 = Parm1 - Parm2</code>
<code>*=</code>	<code>Parm1 = Parm1 * Parm2</code>
<code>/=</code>	<code>Parm1 = Parm1 / Parm2</code>
<code>%=</code>	<code>Parm1 = Parm1 % Parm2</code>
<code><<=</code>	<code>Parm1 = Parm1 << Parm2</code>
<code>>>=</code>	<code>Parm1 = Parm1 >> Parm2</code>
<code>&=</code>	<code>Parm1 = Parm1 & Parm2</code>
<code>!=</code>	<code>Parm1 = Parm1 ! Parm2</code>
<code>^=</code>	<code>Parm1 = Parm1 ^ Parm2</code>

**TABLE 10.3 COMPARISON OPERATORS AVAILABLE
TO THE ASSEMBLER CALCULATOR**

OPERATOR	OPERATION
<code>==</code>	Return 1 if two parameters are equal
<code>!=</code>	Return 1 if two parameters are different
<code>></code>	Return 1 if the first parameter is greater than the second parameter
<code>>=</code>	Return 1 if the first parameter is greater than or equal to the second parameter
<code><</code>	Return 1 if the first parameter is less than the second parameter
<code><=</code>	Return 1 if the first parameter is less than or equal to the second parameter
<code> </code>	Return 1 if either of the two parameters is not zero
<code>&&</code>	Return 1 only if both of the two parameters are not zero
<code>!</code>	Toggle the logical value of a single parameter

will be loaded with 1 because 7 and 5 are not 0, and both are assumed to be “true.” This operation of logical values is not unique to the MPASM assembler calculator; most languages use this convention for “true” and “false.”

The last operator is `!`, which toggles the logical state of a value, for example,

```
A = !4      ; 4 != 0 and is "true"
            ; A = not true
            ;   = false
            ;   = 0

C = !0      ; 0 == 0 and is "false"
            ; C = not false
            ;   = true
            ;   = 1
```

The comparison and logical operators may seem unnecessary for arithmetic calculations, but there are cases where they can be useful.

Multiline C Macros

If you are a new C language programmer, you might be wondering how to implement macros similar to the ones shown in this chapter. Macros are not only for assembly language—they can be used effectively for high level languages as well. If you are learning to program in C, you may have noticed that there isn’t a “macro” directive as there is in assembly language, but you can use `defines` as macros, and using C’s ability to concatenate the text on the next line to the current line, you can produce your own multiline macros.

The `#define` directive in C is similar to the `define` directive in assembly language; when it is encountered, it replaces the label with the string following the define and replaces any parameters with the arguments of the define. For example, if you had a circular buffer 20 entries in size and you wanted to increment the indexes to the buffer, you could use the code

```
buffindex++;           // Increment the buffer index
if (19 < buffindex)    // If the index is 20 or greater, reset
    buffindex = 0;
```

If the code were used often and for different index variables, you might want to consider turning it into a single line of code and using it as the basis for a define. If you are familiar with C, you would know that you could reduce the three lines above to the single line

```
if (19 < ++buffindex) buffindex = 0; // keep ++index in buffer range
```

Despite being difficult to read, this single line will increment `buffindex` and ensure that it is within range for the 20-element circular buffer. The line then could be turned

into the define (which can be used as a macro within C):

```
#define buffinc(indexvar) if (19 < ++indexvar) indexvar = 0;
```

and each time it is invoked, the code `buffinc(buffindex)` would be replaced with the single-line buffer increment and tested to ensure that it is with the index range for the 20-element circular buffer.

The problem is that the operation of the define is not easily read, and if there were an error in the code, you would not be able to see it easily. What is needed is a way to format the define so that the code can be seen easily.

Fortunately, C has the backslash character (\) directive, which appends (or concatenate) the next line of text onto the end of the current line. By using this character directive, you can create a multiline define (or macro) that is much easier to read and understand, minimizing the opportunity for errors to come into your application. For example, using the backslash character, the `buffinc` define could be rewritten as

```
#define buffinc(indexvar)      \  
    indexvar++;               \  
    if (19 < indexvar)        \  
        indexvar = 0;         // keep ++index within buffer range
```

When the backslash is used, the four lines are concatenated together, providing a similar form to the C compiler as the previous define. However, it's in a format that is much easier for you to read or understand.

Note that when using the backslash character, comments to line end (using the `//` format) cannot be used. Instead, you must either restrict your comments to the last line of the macro or use the `/*—*/` comment form.

Reading over this section, you might be tempted to ignore using multiline defines as macros as I've shown here and just create functions for the code. This is possible, but it is a much less efficient way of implementing short pieces of code, such as the example shown here. Implementing the code here as a function will require saving the parameter in a temporary area and then saving it in a destination variable when the function has completed. Depending on the PIC MCU you are working with and the size and complexity of the function you wish to implement, the overhead of saving the data, calling the function, returning from the function, and restoring the data can take more instructions and instruction cycles than what is required for the macro.

Conditional Assembly/Compilation

If you have taken a look at some of the more complex macros presented in this book, you probably will be surprised to see that there are “structured language” statements (`if`, `else`, `endif`, `while`, and `endw`, as listed in Table 10.4). At first glance, these statements are providing high level language capabilities to the PIC microcontroller assembly code. Unfortunately, they're not; these statements provide you with the

TABLE 10.4 CONDITIONAL ASSEMBLY DIRECTIVES AVAILABLE IN MPASM

CONDITIONAL ASSEMBLY DIRECTIVE	FUNCTION
<code>if</code>	Return “true” if both parameters are the same
<code>ifndef</code>	Return “true” if parameters are not the same
<code>ifnlt</code>	Return “true” if the first parameter is greater than the second
<code>else</code>	Return “true” if the first parameter is greater than or equal to the second
<code>endif</code>	Return “true” if the first parameter is less than the second
<code>while</code>	Return “true” if the first parameter is less than or equal to the second
<code>endw</code>	Return “true” if both parameters are “true”

capability of conditionally including statements in your application. Conditional assembly statements are not part of the macro processor; they can be used outside macros and can be used anywhere in application code. Unlike the operation of `if` and `while` that you are used to, allowing condition execution of the application, they are actually executed when the source code is being assembled and can be used to conditionally change constant values or to add or delete sections of code.

Conditional assembly statements are actually “directives,” and they are processed along with other directives (such as `EQU`, `dt`, and so on). For example, if an application were to be run on two different PIC microcontrollers, each with different built-in hardware, the `ifdef` (“execute if define label is found”) conditional assembly could be used in the following manner:

```
#define USARTPres

:

ifdef USARTPres
:           ; Put in USART Handler Code
else
:           ; Put in Non-USART Serial Handler
endif
```

In this case, the `#define` statement creates a label in the application code that can be tested. Later in the code, when the `ifdef` statement is encountered, if the `USARTPres` label is present, the first block of code is put into the assembler source code, and the

second is ignored. If the `USARTPres` label is not defined, then the first block of code is ignored, and the second block is assembled as source code.

For all `if` conditional assembly directives (including `ifdef` and `ifndef`), `endif` is required, and `else` is an optional conditional assembly directive that will include the code if the original condition was not true. You may see a null statement after an `if`, to have the `else` execute like

```
ifdef USARTPres                ; If USART Present, don't add any code
else
    :                          ; Put in Non-USART Code
endif
```

It can be somewhat difficult to understand what is happening. Instead of using `else` to provide conditional execution in the case of the label not being present, the absence of the define should be checked for using the `ifndef` directive.

There are a number of tricks that you can use with `ifdef` and `ifndef` conditional assembly statements that can make your code development easier and more flexible. The first is conditionally deleting code. As you work through an application, often you will want to remove some code to test out different aspects of it. Elsewhere in the book I talk about the idea of “commenting out the code” simply by putting the comment directive (the “`;`” or semicolon character) before the statement such as

```
; addlw 0x0FF                ; #### - Instruction not needed, but kept
```

For single instructions, this is easy to do and easy to keep track of. For many instructions, it can be difficult to keep track of everything that has to be removed (but kept). An easy way of doing this is to put an `ifdef` and `endif` statement before and after the code, as in

```
ifdef Used                    ; #### - Ignore following
    :                        ; Block of Code NOT to be part of application
endif                         ; #### - Ignore above
```

It takes literally just a few seconds to remove the code and can be disabled just as quickly (by defining `Used` or deleting the `ifdef` and `endif` statements).

The second trick is to allow the logic of an application to be used in multiple PIC microcontrollers that may have different built-in hardware features. In the preceding example, by using the `ifdef` directive, I can have the code that takes advantage of the built-in serial port hardware of a specific PIC microcontroller or insert bit banging code in its place if the selected PIC MCU does not have a built-in serial port. I should point out that when an MPLAB IDE application is built, a define label is created indicating the PIC MCU part number. For example if you were assembling an application for the the PIC16F84, the `__16F84` label is available for testing by the `ifdef` and `ifndef` directives.

I have taken advantage of this feature in some applications where the code can run in different devices. For example, the following code will allow an application to run

on either PIC16F84 or PIC16F877 MCUs or generate an error indicating that a supported PIC microcontroller has not been selected for the application:

```

ifdef __16F84A
    include "p16f84a.inc"
else
ifdef __16F877
    include "p16f877.inc"
else
    error "Code is Not Designed for Specified Processor"
endif
endif

```

In this example, note that I have put in “nested” `if` statements. Up to eight levels of nesting are possible in MPASM, although this can be very confusing to read. I would recommend that no more than two levels be used, as in the preceding example.

Along with conditional assembly based on the presence or absence of labels, constant and variable condition testing also can be done with conditional assembly statements. For example, tests against addresses could be performed for interpage jumping in mid-range PIC MCUs:

```

if ((( $ & 0x01800) ^ (Label & 0x01800)) != 0)
    movlw HIGH Label          ; Different Pages - Update PCLATH
    movwf PCLATH
endif
goto Label & 0x07FF          ; Jump to Label

```

In this example, if the destination is in a different page from the current location (which is returned by the `$` directive in MPLAB), then `PCLATH` is updated before the `goto` statement.

The preceding example is suboptimal for four reasons. The first is that whether or not `PCLATH` has to be updated is variably based on the address of the `goto` statement. A more accurate way of doing this would be

```

if ((( ( $ + 2) & 0x01800) ^ (Label & 0x01800)) != 0)
    movlw HIGH Label          ; Different Pages - Update PCLATH
    movwf PCLATH
endif
goto Label & 0x07FF          ; Jump to Label

```

In this case, the possible address of the `goto` is checked rather than the current address. There is the possibility that the current address will be in a different page than the `goto`, and `PCLATH` may or may not be updated correctly.

The second problem is that this code takes up a different amount of space depending on which path is taken. Doing this can result in a address “phase” error that indicates that during the different passes in the assembler, required addresses change in a way that makes correct assembly impossible. These different addresses are caused when

the conditional code executes for a second time, and addresses come out differently. Phase errors are very hard to find, and chances are that if you have one in one location, there will be a number of them.

The best way to avoid phase errors is to always make sure that the same number of instructions are used no matter what path is taken in the conditional assembly. For the preceding code, I can add two `nops` as the code is inserted if `else` (assembled if the condition is not true) is active to make sure that no addresses in the application will change:

```
if (((($ + 2) & 0x01800) ^ (Label & 0x01800)) != 0)
    movlw  HIGH Label          ; Different Pages - Update PCLATH
    movwf  PCLATH
else
    nop                        ; Add Two instructions to prevent
    nop                        ; "Phase" Errors
endif
goto  Label & 0x07FF          ; Jump to Label
```

The third problem with this code is that a message may be produced indicating that the jump is to a different page. To avoid this, the `goto` address should have the current page bits added to it. This changes the code to

```
if (((($ + 2) & 0x01800) ^ (Label & 0x01800)) != 0)
    movlw  HIGH Label          ; Different Pages - Update PCLATH
    movwf  PCLATH
else
    nop                        ; Add Two instructions to prevent
    nop                        ; "Phase" Errors
endif
goto  (Label & 0x07FF) | ($ & 0x01800) ; Jump to Label
```

The next problem with this code is that it changes the `w` register. This means that the preceding code cannot be used if the contents of the `w` register are going to be passed to the destination `Label`. Instead of explicitly loading `PCLATH` with the destination, the bits can be changed individually using the code

```
if (((($ + 2) & 0x01000) ^ (Label & 0x01000)) != 0)
if ((($ + 2) & 0x01000) == 0)
    bsf  PCLATH, 5              ; Label in Pages 2 or 3
else
    bcf  PCLATH, 5              ; Label in Pages 0 or 1
endif
else
    nop                        ; No Difference in High Pages
endif
if (((($ + 2) & 0x00800) ^ (Label & 0x00800)) != 0)
if ((($ + 2) & 0x00800) == 0)
    bsf  PCLATH, 4              ; Label in Pages 1 or 3
```



```

else
    bcf PCLATH, 4                ; Label in Pages 0 or 2
endif
else
    nop                          ; No Difference in Low Pages
endif
goto (Label & 0x07FF) | ($ & 0x01800) ; Jump to Label

```

Looking at this mess of conditional assembly statements, it is starting to look a lot like a macro, and this is the reason why I have included conditional assembly statements in this chapter. Conditional assembly statements, while simplifying your applications in some ways, will result in fairly complex applications in others. The preceding code has really become the `lgoto` macro:

```

lgoto Macro Label
    if (((($ + 2) & 0x01000) ^ (Label & 0x01000)) != 0)
    if (((($ + 2) & 0x01000) == 0)
        bsf PCLATH, 5                ; Label in Pages 2 or 3
    else
        bcf PCLATH, 5                ; Label in Pages 0 or 1
    endif
    else
        nop                          ; No Difference in High Pages
    endif
    if (((($ + 2) & 0x00800) ^ (Label & 0x00800)) != 0)
    if (((($ + 2) & 0x00800) == 0)
        bsf PCLATH, 4                ; Label in Pages 1 or 3
    else
        bcf PCLATH, 4                ; Label in Pages 0 or 2
    endif
    else
        nop                          ; No Difference in Low Pages
    endif
    goto (Label & 0x07FF) | ($ & 0x01800); Jump to Label
endm

```

which can be placed anywhere in your mid-range PIC MCU application, with three instructions replacing the macro each time it is encountered. Similar macros for low-end PIC MCU architectures can be created to allow jumping and calling subroutines (this is not an issue in the PIC18 processor architecture because there are instructions that allow movement to any address that the processor can execute).

Along with program constants, you also can declare integer variables that can be updated during assembly of the application. The `variable` directive, appropriately enough, is used to declare the variables with optional initial values, as shown below:

```
variable i, j=7
```

The variables that can be used as constants in the application or as parts of labels. As I pointed out in the preceding section, variables can be used to avoid having to calculate your own constant values.

I often use variables as counters for use with the `while` conditional assembly statement. For example, if I wanted to loop six times, I could use the code

```
variable i=0          ; Declare the Counter
while (i < 6)
:                    ; Put in Statements to be repeated six times
i = i + 1             ; Increment the Counter
endw
```

Note that when the variable `i` is updated, the statement starts at the first column of the line. The MPLAB assembler requires this.

In the preceding code, the statements within the `while` and `endw` statements are inserted into the assembly-language source file each time the condition for `while` is true. Looking at how `while` has executed in the list file can be a bit confusing. For the code

```
goto    $ + 7          ; Put in Patch Space
variable i = 0
while (i < 6)
dw      0x03FFF        ; Add Dummy Ins
i = i + 1
endw
```

which puts in a jump over six instructions of “patch” space, the listing file looks like

0000 2807	00036 goto	\$ + 7	; Put in Patch Space
0000	00037 variable	i = 0	
	00038 while	(i < 6)	
0001 3FFF	00039 dw	0x03FFF	; Add Dummy Ins
00000001	00040 i =	i + 1	
0002 3FFF	00039 dw	0x03FFF	; Add Dummy Ins
00000002	00040 i =	i + 1	
0003 3FFF	00039 dw	0x03FFF	; Add Dummy Ins
00000003	00040 i =	i + 1	
0004 3FFF	00039 dw	0x03FFF	; Add Dummy Ins
00000004	00040 i =	i + 1	
0005 3FFF	00039 dw	0x03FFF	; Add Dummy Ins
00000005	00040 i =	i + 1	
0006 3FFF	00039 dw	0x03FFF	; Add Dummy Ins
00000006	00040 i =	i + 1	
	00041 endw		

This listing looks more like six `dw 0x03FFF` instructions and `i = i + 1` statements rather than the two of them being repeated six times.

The conditional assembly instructions `if` and `while` use the same condition test format as the `if` and `while` statements of the C language. The condition tests can take place only

TABLE 10.5 COMPARISON OPERATORS AND RETURN VALUES

CONDITION	FUNCTION
==	Return “true” if both parameters are the same
!=	Return “true” if parameters are not the same
>	Return “true” if the first parameter is greater than the second
>=	Return “true” if the first parameter is greater than or equal to the second
<	Return “true” if the first parameter is less than the second
<=	Return “true” if the first parameter is less than or equal to the second
&	Return “true” if both parameters are “true”
	“True” if either one of the parameters is “true”

on constant values of up to 32 bits in size. Like C’s `if` and `while`, the MPASM assemblers conditional assembly statements use the two parameter conditions listed in Table 10.5.

When the statements are “true,” a nonzero value is returned. If the statements are “false,” then 0 is returned.

Using Defines and Conditional Assembly for Application Debug

In this book I talk a lot about the need for simulating applications before they can be burned into actual PIC microcontroller hardware. For many applications, this isn’t an issue, but for applications that have very long delays built into them, this can be a very significant issue because the time required for external hardware initializations actually can take many minutes in MPLAB because the simulation has to go through long delay loops. Another situation could be for hardware that is nonessential to the task at hand but that requires a relatively complex interaction with simulated hardware or uses built-in PIC microcontroller interfaces that are not simulated in MPLAB. An example of the latter situation is a PIC microcontroller application that uses the ADC for testing battery voltage levels, but if the hardware registers are accessed in the simulator, then chances are the operation complete flag and interrupt will not work properly, and even if they did, the value returned by the ADC would be invalid.

Dealing with this problem is relatively simple with use of the conditional assembly directives built into the MPASM assembler. These instructions will allow execution to skip over problem code areas simply by specifying a flag using the `#DEFINE` directive.

The state of the `#DEFINE` flags is generally defined by their presence or absence. The most common one that you will see in my code is the `Debug` label that is defined for simulation by using the statement

```
#DEFINE Debug
```

You usually will see this statement on the second or third line of my applications, although when I am ready to burn an application into a PIC microcontroller, I change the statement to

```
#DEFINE nDebug
```

to disable any changes to the source code and to make sure that the object file is created with the correct code.

Use of the `Debug` label to replace blocks of code that would be a problem in the simulator (such as code that will take a long time to execute and is not germane to the testing at hand) with nops (to ensure that the overall addresses aren't changed) is as follows:

```
ifndef Debug
    call UnreasonablyLongDelay    ; Wait for External Hardware
else                               ; "Debug" #DEFINED
    nop                           ; Put in Instruction Anywise
endif
```

In this code, if `Debug` has not been set in a `#DEFINE` statement, the instructions that are to be skipped over are built into the application. If `Debug` has been defined, then the `nop` is put into the application in the place of the `call` instruction. If the number of instructions is not matched with an equal number of nops, you could run into the situation where the code assembles in different sizes for the different execution paths, which results in the *phase error*, which is very hard to debug.

Along with being used for taking out lengthy delays or hardware inconsistencies in the MPLAB simulator, `Debug` conditional assembly code can be used to put the application into a specific state. This can save quite a bit of time during simulation and also can be used to initialize the state of the application before simulation begins. For example, when I was creating the EMU-II emulator application code, I used the `Debug` label to enter in the following commands:

```
ifdef Debug                ; Clear simulated application program Flash
    movlw 'P'              ; Start a program memory Program/Clear Flash
    call BufferAdd
    movlw 0x00D
    call BufferAdd
    movlw 0x003            ; Stop the transfer after the data is cleared
    call BufferAdd
endif
```

This code initiates the command to load a new application into the Flash (and in preparation for the code, the Flash is cleared), followed by a `Ctrl-C` to stop the expected transfer. The reason for requiring this code is that any writes to the simulated Flash would not be reset when I rebuilt the application. These commands “clear” the simulated Flash so that each time the application code is reset, I would be working with cleared Flash instead of something that I would have to simulate the operation of downloading code into it. This simulation isn’t possible because of my use of the USART in the application, which is not simulated within MPLAB.

When the EMU-II application was being debugged, I placed a breakpoint at a location where I wanted to start debugging, knowing that the program memory was cleared and I was ready to start looking at how the application executed.

Multiple `Debug` statements could be put into the application code, but I would not recommend that you do this. Multiple statements get confusing very quickly, as well as often becoming incompatible. Instead of using multiple `Debug` labels, I would recommend that you just use one, and when you have fixed the current problem you are working on, then you can change the `ifdef Debug` and `ifndef Debug` statements to target the next problem.

Debugging Macros

In the preceding pages I’ve given you a lot of information on how to create complex macros; now I want to spend a few pages discussing how to debug these monsters. As the macro processor executes, it is probably going to put in some code that will be surprising to you. I should point out that normally a C compiler will not include a listing showing the inserted statements, the same way an assembly-language program will. It is not unusual to discover that the application does not work as you expect because the macro processor has inserted some code that does not work as you would expect, and now you are left with the task of determining where the problem lies. In this section I will pass along a few tricks that I have discovered that make the debugging of macros easier and allow you to use them with greater confidence.

The first trick is something that a surprising number of people do not think of: before using a macro in their applications: Spend some time testing your macro in all possible ways to ensure that it does work as you expect. Earlier I showed the rather complex macro that I have developed for producing code that will allow a mid-range PIC MCU to jump to addresses outside the current execution page:

```
lgoto Macro Label
if (((($ + 2) & 0x01000) ^ (Label & 0x01000)) != 0)
if (((($ + 2) & 0x01000) == 0)
    bsf PCLATH, 5                ; Label in Pages 2 or 3
else
    bcf PCLATH, 5                ; Label in Pages 0 or 1
endif
```

```

    else
        nop                                ; No Difference in High
Pages
    endif
    if (((($ + 2) & 0x00800) ^ (Label & 0x00800)) != 0)
    if (((($ + 2) & 0x00800) == 0)
        bsf  PCLATH, 4                    ; Label in Pages 1 or 3
    else
        bcf  PCLATH, 4                    ; Label in Pages 0 or 2
    endif
    else
        nop                                ; No Difference in Low Pages
    endif
    goto    (Label & 0x07FF) | ($ & 0x01800)        ; Jump to Label
endm
```

Before using this macro, I would create a simple application that would test it in a variety of different situations to ensure that it worked as I expect. When designing this application, I would spend some time creating a table listing the different test cases with expected outcomes such as the ones in Table 10.6.

This table only lists a few of the possible starting points and destinations that are possible. Note that the second case should produce an error because the starting address is one address away from the destination, and the macro takes up three instructions. In this case, the macro will attempt to write instructions over the instructions already placed at address 0x800.

The more time you can spend debugging your macros, the better. In Table 10.6, I have only listed a few of the different test cases (ideally, cases should be created in which each of the PCLATH bits is set and reset), and no registers or bits that aren’t part of the jumps are not affected. The macro is designed to work without changing the w register or the W, C, or DC bits—these values should be checked before and after each invocation of the `lgoto` macro to make sure that this is the case. Spending time up front debugging

TABLE 10.6 TABLE DEVELOPED TO TEST THE OPERATION OF THE <code>lgoto</code> MACRO						
TEST CASE	STARTING VALUES			FINAL VALUES		
	DESTINATION ADDRESS	STARTING PC	PCLATH	ADDRESS	PC	PCLATH
1	0x0800	0x000	0b00	0x0800	0x000	0b01
2	0x0800	0x7FF	0b00	Error	Error	Error
3	0x0800	0x7FD	0b00	0x0800	0x000	0b01
4	0x0010	0x200	0b01	0x0010	0x010	0b00
5	0x10AA	0x010	0b00	0x10AA	0x0AA	0b10

the macros will avoid the need to go back later when your code isn't behaving properly and you are trying to understand why some cases work properly and others don't.

The key to debugging macros is being able to observe what changes the macro code made and whether or not they are appropriate for use in the application. This may seem to be identical to the testing done on the macro, but there is a subtle and very important difference: In macro testing, you are creating test cases and comparing the changes made to specific registers to verify the operation of the macro. When you are observing the changes made by a macro in an application, you have to understand what the purpose of the macro was and whether or not the changes that were generated were appropriate for that point in the application.

If you were debugging the `lgoto` macro, you might want to check to see what the execution address is after the macro's code has executed and if any other registers changed inadvertently. Ideally, you should not be setting a breakpoint at the expected destination (because there is a chance that execution would end up there at some point) but instead single-stepping through the macro's statements to make sure that there is no jump to an unexpected address. It can take some ingenuity to test the operation of the macro by seeing how it works in the application.

An important key to debugging macros is being able to read what has been added to the application. When macros are expanded into the source code, the new instructions can be difficult to see and work through. For example, when macros execute, the conditional statements (`if`, `ifdef`, `ifndef`, and `while`) may or may not be displayed depending on the operation. I think of the regular instructions of a macro statement to be `print` or `echo` statements, and they are copied into the source as is (except in the case where one of the parameter strings is present in the instruction or on the line or the assembler calculator is used to produce a specific value). To illustrate these points, I have created conditional assembly statements for a macro:

```
variable i = 0
if (i == 0)
    addlw    0 - i
else
    sublw    i
endif
```

When the macro is "expanded" (or executed), the following information will be displayed in the listing file:

```
00000000      M      variable i = 0
              M      if (i == 0)
Addr  3E00      M      addlw    0 - i
              M      else
              M      sublw    i
              M      endif
```

As this example illustrates, the setting of the assembly variable (`i`) results in a constant value being placed somewhere between the start of the line and the middle. The instructions that

are actually added to the listing file are identified by the address of the instruction and its bit pattern broken up as I have shown for the `addlw 0 - 1` instruction. In this example, only `addlw 0 - 1` is inserted into the source code; the directives (`variable`, `if`, `else`, and `endif`), as well as the `sublw i` instruction, are all ignored.

While directives are somewhat unusual to follow because the code is repeated within them, and there is no start and end reference information that can be easily seen. For the example

```
variable i = 0
while (i < 2)
    movlw    i
    i = i + 1
endw
```

the following listing file information is generated:

00000000	M	variable i = 0
	M	while (i < 2)
Addr 3000	M	movlw i
00000001	M	i = i + 1
Ad+1 3001	M	movlw i
00000002	M	i = i + 1
	M	endw

In this case, the `while` and `endw` directives are not repeated as you would expect. Again, to really understand what is happening, you have to go back and look at the instructions entered into the application code. These instructions are displayed to the left of the column of M's along with the variable updates.

Probably the best way to debug a macro is to single-step in the simulator through the macro. If you are doing this in the source file, you will find execution jumps to where the macro is defined that can be quite disconcerting and confusing because the parameters are displayed, not what the parameters actually are. These problems can be avoided by simulating from the listing file instead of from the source file.

Breakpoints cannot be placed in MPLAB at a macro invocation for a source file. When I want to set a breakpoint at a macro invocation, I will put a `nop` instruction before it to give the MPLAB simulator somewhere on which to hang the breakpoint. This also works with a listing file, but in this case you do not have to add the breakpoint because breakpoints can be put at instruction statements within the macro-generated code.

Debugging macros is really the same as debugging a standard application, except that there is no simulator or debugger. This is an important point to remember when you are creating the macro. You may want to “flag” where the code is executing using messages (using the `messg` directive), which is the same as `printf` debugging in C. For example, if you have the code

```
if (A == B)
messg "A == B"
;    put in code for A == B
```



```

else
messg "A != B"
;    put in code for A != B
endif

```

the `messg` directives will flag the path execution takes through the macro's conditional assembly. This (and the preceding) tricks can be used with conditional code external to macros to help you follow their execution and find any problems within them.

Structured Programming Macros

To finish off this chapter, I wanted to leave you with a few macros that should give you some ideas on how powerful macros are when you need some processing to be done when the application is being created, as well as give you a tool for making your PIC microcontroller assembly-language code easier. The `structre.inc` file contains nine macros that you can use for your own applications to add structured programming conditional execution.

The macros are in the format

```
"_" Test Const/Var
```

where *Test* can be `do`, `until`, `while`, `if`, `else`, and `end`, and *Const/Var* specifies whether or not the second parameter is a constant or a variable. The different macros are listed in Table 10.7.

When conditional macros are invoked, the parameters passed to the macro include the two conditions for testing as well as the condition to test for. The standard C test conditions are used, as listed in Table 10.8.

Only one condition can be accessed within a macro at any time. There is no ability to AND or OR conditions together. Using these macros in your application is quite straightforward, and if you are new to PIC microcontroller programming, you might want to take a look at using these macros for your conditional programming. Executing some code conditionally if two variables are equal would use the following macro invocations:

```

_ifv VariableA, ==, VariableB
:                               ; Code to Execute if VariableA == VariableB
_end

```

Using these macros can be expanded to include an `else` condition for code that executes if the condition is not true:

```

_ifv VariableA, ==, VariableB
:                               ; Code to Execute if VariableA == VariableB
_else
:                               ; Code to Execute if VariableA != VariableB
_end

```

TABLE 10.7 STRUCTURED PROGRAMMING MACRO INVOCATIONS	
INVOCATION	DESCRIPTION
<code>_ifv Parm1, Condition, Parm2</code>	Execute the following code if the two variables (Parm1 and Parm2) operated on with the Condition are “true.” <code>_else</code> following is optional, and the code following it will execute if the condition is “false.” Must have <code>_end</code> following.
<code>_ifc Parm1, Condition, Parm2</code>	Execute the following code if the variable (Parm1) and constant (Parm2) operated on with the condition is “true.” <code>_else</code> following is optional, and the code following it will execute if the condition is “false.” Must have <code>_end</code> following.
<code>_else</code>	Execute the following code if the result of the previous <code>_if</code> macro was “false.” Execution before the <code>_else</code> jumps to the next <code>_end</code> macro.
<code>_end</code>	End the <code>_if</code> or <code>_while</code> macro invocation. If previous operation was “_while”, then jump back to “_while” macro.
<code>_whilev Parm1, Condition, Parm2</code>	Execute the following code while the two variables (Parm1 and Parm2) operated on with the Condition are “true.” Must have <code>_end</code> following, which will cause execution to jump back to the <code>_while</code> macro.
<code>_whilec Parm1, Condition, Parm2</code>	Execute the following code while the variable (Parm1) and constant (Parm2) operated on with the Condition are “true.” Must have <code>_end</code> following, which will cause execution to jump back to the <code>_while</code> macro.
<code>_do</code>	Start of <code>do/until</code> loop.
<code>_untilv Parm1, Condition, Parm2</code>	Jump to previous <code>_do</code> if the variables (Parm1 and Parm2) operated on with Condition are “false.”
<code>_untilc Parm1, Condition, Parm2</code>	Jump to previous <code>_do</code> if the variable (Parm1) and the constant (Parm2) operated on with Condition are “false.”

TABLE 10.8 COMPARISON OPERATORS USED IN STRUCTURED PROGRAMMING MACROS	
CONDITION	DESCRIPTION
<code>==</code>	Equals
<code>!=</code>	Not equals
<code>></code>	Greater than
<code>>=</code>	Greater than or equals
<code><</code>	Less than
<code><=</code>	Less than or equals

For these macros, I decided to use as close a label to actual programming structures as possible, which is why I used the standard names with the underscore character before them.

There are five aspects and features of the macro processor that have influenced how these structured conditional execution macros were created. The first is that you cannot distinguish between constants and variable addresses in a macro. When the macro is invoked, a variable address is passed to the macro instead of the variable label. This was probably done to simplify the effort in writing the macro processor. For this application, it means that either what the value types are must be specified explicitly or there must be a way of declaring variables so that they can be differentiated from constants.

I looked at a number of different ways to tell the two types of values apart and found that I could not do it without changing how variables were declared, which would make the application code more complex. Because I could not tell the two different types of data apart, I decided to always make the first parameter a variable and the second one a variable (v) or constant (c) depending on the character at the end of the macro.

Interestingly enough, this conversion of data does not extend to nonalphanumeric characters. In the macros, the condition test is specified, and this is passed into the macro. I take advantage of this to avoid having to create multiple macros each with a different condition test. Inside the macro, I use this `test` condition parameter to determine what it actually is (there is no way of comparing nonnumeric values in the conditional assembly functions of the macro processor). For example, in the `_ifv` macro, to find out what is the condition, I test the specified condition against different cases:

```

if (1 test 1)                ; Check for "=="
    movf    a, w
    subwf   b, w
    btfss   STATUS, Z        ; Zero Flag Set if True
else
    if (1 test 0)            ; Check for "!=" / ">" / ">="
        if (0 test 1)        ; Check for "!="
            movf    a, w
            subwf   b, w
            btfsc   STATUS, Z    ; Zero Flag Reset if True
        else                  ; Else ">" / ">="
            if (1 test 1)      ; Check for ">="
                movf    b, w
                subwf   a, w
                btfss   STATUS, C    ; Carry Set, ">="
            else
                movf    a, w
                subwf   b, w
                btfsc   STATUS, C    ; Carry Reset, ">"
            endif
        endif
    endif
else
    if (0 test 1)            ; Check for "<" / "<="
        if (1 test 1 )

```

```

        movf    a, w
        subwf   b, w
        btfss   STATUS, C
    else
        movf    b, w
        subwf   a, w
        btfsc   STATUS, C
    endif
    else
        error Unknown "if" Condition
    endif
endif
endif
endif

```

To help determine if there is an error in the code, note that if no `Condition` test is “true,” an “error” is forced, indicating that the input condition is unknown. This can be another technique for debugging macros: If conditional code ends up somewhere where it shouldn’t be, an error message will alert you to the situation and help you to debug the application.

The macros themselves use conditional code to produce simple code for the actual functions. For example, in the code

```

_ifv Parm1, >, Parm2
:                ; Code to Execute if Parm1 > Parm2
_else
:                ; Code to Execute if Parm1 <= Parm2
_end

```

the best-case assembler would be

```

; _ifv Parm1, >, Parm2
    movf    b, w
    subwf   a, w
    btfss   STATUS, C
    goto    _ifelse1    ; Not True, Jump to "else" code
:                ; Code to Execute if Parm1 > Parm2
; _else
    goto    _ifend1    ; Finished with "true" code, Jump to "_end"
_ifelse1:
:                ; Code to Execute if Parm1 <= Parm2
; _end
_ifend1:

```

If you try out the macros in `structre.inc`, you will find that this is the exact code that is created for this example. To do this, I had to create three stacks to keep track of where I was. The first stack records what is the nesting level of the structured conditional statements. For these macros, I only allow four nesting levels deep. The next stack

records what was the previous operation. This is important for `_else`, `_end`, and `_until` to make sure that they are responding correctly. The last stack records the “label number” for the previous operation. These stacks are combined with a label number to keep track of what is the correct label to use and jump to.

The label number is appended to the end of the label using the `#v(Number)` feature of the MPASM macro assembler. When a label is encountered with this string at the end, the `Number` is evaluated and concatenated to the end of the string. For the label

```
_test#v(123)
```

the actual label recognized by MPASM would be

```
_test123
```

I use this feature to keep track of which label should be jumped to. After every statement, I increment this counter for the next statement to use for its labels. Expanding the preceding example to

```
_ifv Parm1, >, Parm2
_whilec ParmA == ParmB
:                               ; Code to Execute if Parm1 > Parm2
:                               ; while ParmA == Constant ParmB
_end                           ; End the “_while”
_else
:                               ; Code to Execute if Parm1 <= Parm2
_end
```

the structured programming macros will push the appropriate label number onto the stack and retrieve it when necessary. Thus, for the expanded example, the actual PIC micro-controller code would be

```
; _ifv Parm1, >, Parm2
movf  b, w
subwf a, w
btfss STATUS, C
goto  _ifelse1           ; Not True, Jump to “else” code
; _whilec ParmA == ParmB
_ifwhile2:
movf  ParmA, w
subwf ParmB, w
btfss STATUS, Z
goto  _ifend2
:                               ; Code to Execute if Parm1 > Parm2
:                               ; while ParmA == Constant ParmB
;_end                           ; End the “_while”
goto  _ifwhile2
_ifend2:
```

```
; _else
    goto _ifend1          ; Finished with "true" code, Jump to "_end"
_ifelse1:
    :                    ; Code to Execute if Parm1 <= Parm2
; _end
_ifend1:
```

I realize that this code is somewhat hard to follow, but if you work through it, you will discover that the `_while` loop is separate from the `_if` code by virtue of the label number, which is 2 for the `_while` and 1 for the `_if`. These values are kept track of by the three stacks I mentioned earlier.

The stacks used for storing the label number and the other values are not stacks per se but actual variables that are shifted over by four and then loaded with the value. This limits the total label numbers to 16 different labels, but for most small PIC microcontroller applications, this should be sufficient. If you feel that more are needed, then you could modify the macros to use a separate stack for `_if`, `_while`, and `_do`, as well as come up with a way of having multiple stack values for each one. With a bit of work, you could have up to 64 label numbers for each type of structured programming macro by expanding the type of macro saved to four different types.

The macros described in this section could be considered to come under the heading of "out there." These macros involved quite a bit of work to get them to the point where they are now. After reading this chapter, you do have the knowledge to produce macros like this, but I want to caution you to think through what you are trying to accomplish. Macros, almost by definition, do not produce functions that are easy to debug or even understand.

BUILDING AND LINKING

One of the most important enhancements to the PIC[®] application code development process made by Microchip over the past few years is the inclusion of a linker and library manager for MPLAB IDE. These tools allow code to be developed much more efficiently by multiple individuals than the traditional method of assembling or compiling a single large file. To fully take advantage of linked applications, there are a number of new concepts that you will have to be comfortable with. These concepts are not very difficult to understand, and you should be able to apply them almost immediately. The ability to link portions of code together will provide you with the ability to develop applications more effectively for the PIC microcontroller.

I have indicated that it is not very difficult to learn how to create applications that are linked, but I would recommend that when you start learning to program the PIC microcontroller in assembly language, you do it by the traditional single-file approach. The reason I make this recommendation is that you are learning a number of new concepts, even if you are an experienced programmer, and adding additional concepts will make the process more complicated. The additional complications are not limited to development of the application code but also to using MPLAB IDE for application simulation and debugging. In these cases, you will have to have multiple source file windows active and will have to be very familiar with your code to be able to follow its execution path between different modules. When you become proficient at creating and debugging simple applications for the PIC microcontroller, you should come back to this chapter to learn more about developing linked applications because you will be able to understand them more effectively as they relate to the PIC and PIC microcontroller applications.

Creating Linked Applications

Creating a linked C application (using the OIC18 or PICC compilers) is surprisingly easy in MPLAB IDE. It is accomplished simply by right clicking on “Source Files” in the file application box of the MPLAB desktop and then selecting “Add Files . . .” and

then selecting the files that you would like to add. The source files could be but don't have to be in the same folder as each other or, for that matter, the project hex files that make your management of source files quite a bit easier. You also can customize where the generated code is placed into the PIC microcontroller. Unfortunately, these capabilities are not so easy to implement in assembly and other languages.

When I am talking about *linking*, I am describing the combining of “object files” (which have the file extension `.o`) and library routines (with the file extension `.lib`) together to create a single application. These object files are produced by compiling multiple source code files. Together, these source code files make up the entire application and may have been written by different people; they even could have been written years before or could be provided by a third party. The primary reason for providing this function is to make things more convenient for application developers (meaning that it will take only minutes to compile and build the application with the latest changes), and it protects the application code (once some code is determined to be working correctly, it is compiled, and the linked files are used from then on).

When I was writing the first two editions of this book, it was quite unusual to have more than one person developing PIC microcontroller application code. The devices had a very modest amount of program memory, and the processor architecture did not lend itself well to compiled languages with linked object files. Over the years since the last edition of this book, the PIC18 devices have become very popular because they have up to one million instruction application capabilities and an architecture for which it is easier to write traditional compilers. This has lead to products requiring teams of developers and also has required source code control to ensure that the correct versions of different functions are used in the final application. This trend toward multiple developers is sure to continue and is being taken advantage of in low-end and mid-range devices, where compilers capable of producing object files that can be linked in to applications are becoming more available.

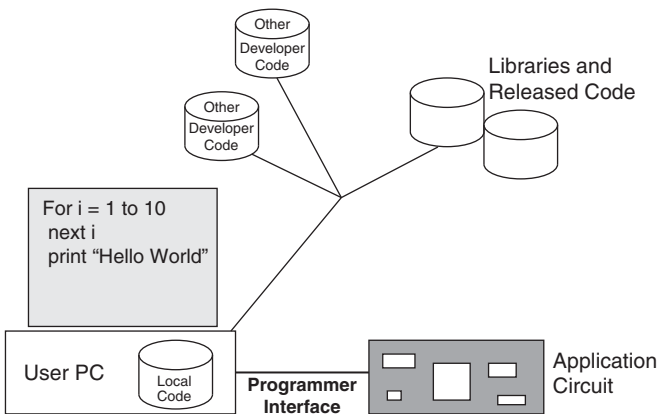


Figure 11.1 Modern application code development takes place over several systems, linking in object files to produce the final code image that is programmed into the system.

In Fig. 11.1 I have drawn a block diagram showing how modern application development typically takes place. A developer, on his or her local machine, is modifying a source file on one aspect of the application. When he or she is ready to test it out, he or she will build the application on the local machine not compiling the code, but he or she may take source code from other users and compile it into object files to be linked to his or her code as well as “released” object files and libraries that are available on central servers. Once the “build” is complete, the developer can download the resulting application into target hardware to test it out. This method of development allows multiple developers to work simultaneously and test out their code on local application hardware using the latest and best released code available. This process is very convenient for developers and ensures that once code is deemed to be “good,” it, along with its object files, can be located on a central server where it cannot be modified inadvertently, resulting in problems for the developers who will have to try to figure out what went wrong and fix it.

To demonstrate how simple it is to link together source files in an application, consider a simple HT Soft PICC (Lite) application that has one subroutine that can be located in a separate source file. The mainline code consists of

```
#include <pic.h>
/* Compile Test - Try to Create a simple application which calculates
prime numbers

myke predko

07.06.17

*/

int PrimeCheck(int Value);           // Prime Checking Function

main()
{
int i, j;

    for (i = 1; i <= 100; i++)        // Test Numbers from 1 to 100
        j = PrimeCheck(i);

    while (1);

} // naim

and the PrimeCheck subroutine file is

// Primecheck - Moved from One File to a New One

int PrimeCheck(int Value)
{
```

```

int i, j;

j = 1;                                // Assume value is prime
for (i = 2; (i < ((Value / 2 ) + 1) && (0 != j)); i++)
    if (0 == (Value % i))              // If Evenly Divisible by i,
        j = 0;                        // Value is not Prime

return j;                             // Return 1 if "Value" is Prime
} // end PrimeCheck

```

These files can be found in the “Linking Test” folder.

Now, create a project in MPLAB IDE with a PIC16F84 processor (which is supported by PICC Lite), and add `Compile Test.c` and `PrimeCheck.c` as the two source files. When the MPLAB IDE desktop comes up, you will see that under the file view window, both files are listed under “Source Files.” When this is done, you can display both files on the MPLAB IDE desktop and then select “Build All,” which will compile both files and link them together. The only issue that I found in doing this was that I had to have the `PrimeCheck` prototype in “Compile Test.”

After building the files and closing MPLAB IDE, you might want to take a look at the folder into which you copied the two source code files and put the resulting hex file. On my PC, this folder now contains 23 different objects when I would have thought that there should only be 6 (with the 6 being the two source files and their `.o` object files, the `.hex` file, and a `.cod` or `.coff` file that would be used by the MPLAB IDE simulator or MPLAB ICD for debugging the application). The additional 17 files are used by the compiler and the linker to produce the application. The files that you should be concerned with are

- 1** The source files (`.c`)
- 2** The object files (`.o`)
- 3** The compiler listing files (`.lst`), which provide you with basic information regarding the results of the compilation
- 4** The hex file (`.hex`)
- 5** The debugger file (`.cod` or `.coff`), which is used to allow source-code-level simulation and debugging
- 6** The linker file (`.lkr`), which may have been brought into the build folder to modify how the code is implemented
- 7** The application map file (`.map`), which outlines how the application uses data and program memory

All the other files can be ignored.

The linker file will be explained in more detail in the next section, but I wanted to point out its existence and its function, which is to specify where code and variables can be placed. Each location for code or memory is labeled, and you can specify different functions to be placed in specific locations in the PIC microcontroller.

The application map file is a very complex file that lists how the application was organized and will be stored in memory. To truly understand how the application is laid out, you will need to cross-reference this file to the listing files because the listing files have

the sizes needed by the different object files, and you can see how they were located in the final `.hex` file using the `.map` file. For the most part, you won't care how the code is placed in the PIC microcontroller, but as you develop more complex applications and use different tools such as "bootloaders," you will be taking on a greater interest in how the application code is stored.

When you are implementing C applications, you are going to have to identify global variables that are used in the different source files properly. This is actually quite simple; I would recommend that all global variables be declared in the same file as the application "main" is located. In the other files, these declarations can be copied, but after the file type, put in the directive `extern` like

```
int extern GlobalVariable;
```

When the `extern` is used, the variable cannot be initialized as part of the declaration. This can be done only in the primary variable declaration.

This overview is actually all you have to do to implement linked PICC and PIC18 applications. Creating linked applications for code written in other languages is quite a bit of work, with the need to define code segments and data segments for variables and ensure that the code is either relocatable (which means that it can be placed anywhere in memory) or that you specify explicitly where it is going to be located. This work is not terribly difficult, but it is very easy to make a mistake in.

As you start working with more complex applications, I would suggest that you minimize the amount of assembler code that you create and work exclusively in PICC or PIC18, with any necessary assembly language being embedded in C source files.

.1kr FILES

The linker script (PIC MCU part number `.1kr`) describes how the different memories (program memory, file registers, and data EEPROM) are set up in a specific PIC microcontroller. Memory areas (known as `CODEPAGE` or `DATABANK`) within the architecture can be `PROTECTED`, preventing the linker from placing code or variables at these locations. As you become more familiar with linked-code development on the PIC microcontroller and you are looking to develop more and more complex applications, you will want to customize the linker files so that you can specify explicitly how applications are loaded into the target PIC microcontroller.

A sample `.1kr` file for the PIC16F84 is

```
// Sample linker command file for 16F84
// $Id: 16f84.1kr,v 1.4.16.1 2005/11/30 15:15:29 curtiss Exp $
```

```
LIBPATH .
```

CODEPAGE	NAME=vectors	START=0x0	END=0x4	PROTECTED
CODEPAGE	NAME=page	START=0x5	END=0x3FF	
CODEPAGE	NAME=.idlocs	START=0x2000	END=0x2003	PROTECTED
CODEPAGE	NAME=.config	START=0x2007	END=0x2007	PROTECTED
CODEPAGE	NAME=eedata	START=0x2100	END=0x213F	PROTECTED

```

DATABANK  NAME=sfr0    START=0x0    END=0xB    PROTECTED
DATABANK  NAME=sfr1    START=0x80   END=0x8B   PROTECTED

DATABANK  NAME=gprs    START=0xC    END=0x4F

SECTION   NAME=STARTUP ROM=vectors    // Reset and interrupt vectors
SECTION   NAME=PROG    ROM=page       // ROM code space
SECTION   NAME=IDLOCS  ROM=.idlocs   // ID locations
SECTION   NAME=DEEPROM ROM=eedata    // Data EEPROM

```

This file notes that the reset vectors instruction area (the four locations before the interrupt vector) are PROTECTED and cannot have user-developed code stored in them. Next, the program memory that the application can reside in is specified (0x5 to 0x3FF). Following this, the IDLOCS configuration register addresses are protected as is the EEPROM data area. For variable storage, the DATABANK statements outline the locations of the special function registers (which cannot be used for variables), along with the file registers (gprs) that are used for variables.

The PIC16F84 cannot show an important aspect of the .lkr file, and that is how code pages and register banks are handled by the linker. In other devices, such as the PIC16C63, which has two register banks and two code pages, there are additional statements to indicate their presence:

```

// Sample linker command file for 16C63
// $Id: 16c63.lkr,v 1.3.16.1 2005/11/30 15:15:28 curtiss Exp $

```

```
LIBPATH .
```

```

CODEPAGE  NAME=vectors  START=0x0    END=0x4    PROTECTED
CODEPAGE  NAME=page0    START=0x5    END=0x7FFF
CODEPAGE  NAME=page1    START=0x800   END=0xFFFF
CODEPAGE  NAME=.idlocs  START=0x2000 END=0x2003  PROTECTED
CODEPAGE  NAME=.config  START=0x2007 END=0x2007  PROTECTED

DATABANK  NAME=sfr0    START=0x0    END=0x1F   PROTECTED
DATABANK  NAME=sfr1    START=0x80    END=0x9F   PROTECTED

DATABANK  NAME=gpr0    START=0x20   END=0x7F
DATABANK  NAME=gpr1    START=0xA0    END=0xFF

SECTION   NAME=STARTUP ROM=vectors    // Reset and interrupt vectors
SECTION   NAME=PROG1   ROM=page0      // ROM code space - page0
SECTION   NAME=PROG2   ROM=page1      // ROM code space - page1
SECTION   NAME=IDLOCS  ROM=.idlocs   // ID locations

```

It is important to note that in low-end and mid-range devices, the linker does not allow functions to go over page boundaries, so the available areas are specified separately (as shown in the preceding example).

The PIC18 architecture is a different case because the program memory space is flat and can be accessed anywhere using the `goto` and `call` instructions. This results in a simpler `.lkr` file like the one for the PIC18F87J50 (which I chose because it has a 128 kB Flash program memory space):

```
// $Id: 18f87j50i.lkr,v 1.1.2.1 2006/09/27 23:12:31 curtiss Exp $
// File: 18f87j50i.lkr
// Sample ICD2 linker script for the PIC18F87J50 processor

// Not intended for use with MPLAB C18.  For C18 projects,
// use the linker scripts provided with that product.

LIBPATH .

CODEPAGE    NAME=vectors    START=0x0      END=0x29      PROTECTED
CODEPAGE    NAME=page       START=0x2A     END=0x1FFF7
CODEPAGE    NAME=config     START=0x1FFF8  END=0x1FFFD   PROTECTED
CODEPAGE    NAME=devId      START=0x3FFFFE END=0x3FFFFF  PROTECTED

ACCESSBANK  NAME=accessram  START=0x0      END=0x5F
DATABANK    NAME=gpr0       START=0x60     END=0xFF
DATABANK    NAME=gpr1       START=0x100    END=0x1FF
DATABANK    NAME=gpr2       START=0x200    END=0x2FF
DATABANK    NAME=gpr3       START=0x300    END=0x3FF
DATABANK    NAME=gpr4       START=0x400    END=0x4FF
DATABANK    NAME=gpr5       START=0x500    END=0x5FF
DATABANK    NAME=gpr6       START=0x600    END=0x6FF
DATABANK    NAME=gpr7       START=0x700    END=0x7FF
DATABANK    NAME=gpr8       START=0x800    END=0x8FF
DATABANK    NAME=gpr9       START=0x900    END=0x9FF
DATABANK    NAME=gpr10      START=0xA00    END=0xAFF
DATABANK    NAME=gpr11      START=0xB00    END=0xBFF
DATABANK    NAME=gpr12      START=0xC00    END=0xCFF
DATABANK    NAME=gpr13      START=0xD00    END=0xDFF
DATABANK    NAME=gpr14      START=0xE00    END=0xEF3
DATABANK    NAME=dbgspr     START=0xEF4    END=0xEFF     PROTECTED
DATABANK    NAME=gpr15      START=0xF00    END=0xF3F
DATABANK    NAME=sfr15      START=0xF40    END=0xF5F     PROTECTED
ACCESSBANK  NAME=accesssfr  START=0xF60    END=0xFFF     PROTECTED
```

Note that in the PIC18F87J50 `.lkr` file there is a single code page specification throughout the entire memory map, but there are multiple file register databanks. The flat architecture of the PIC18 allows for seamless placement of code, but not of data. Also in the PIC18 `.lkr` file, you will see that the access bank is incorporated as well.

The biggest problem I have with working with the `.lkr` files is discovering where they are. For the standard MPLAB IDE linker files [which can be used for PICC (Lite) applications], they are in

C:\Program Files\Microchip\MPASM Suite\LKR

Microchip PIC18 linker files are found in

C:\MCC18\lkr

MPLIB LIBRARIAN

When you work with a compiled language, you generally are given a set of libraries that are linked into the application. These libraries contain a number of object (.o) files that can be included in an application when needed. In the appendices I have listed the library functions that are available in the standard C programming language, as well as PIC18's extensions. Each of these functions is included in the library, but they are included in the application only when they are required. Libraries are extremely useful programming constructs that will make your application development quite a bit easier.

To create your own libraries, create a project with the files that you want to include in the library and then click on "Project\Build Options\Project" followed by selecting the "MPASM/C17/C18 Suite" tab and then clicking on the "Build library target (invoke MPLIB)" radio button. The next time you build your project, a library will be produced. The MPLIB librarian also can be invoked from the MS DOS prompt command line—this interface is useful when you have an existing library and you want to change or delete object files in that library. The only thing that MPLIB librarian doesn't do that I would have liked is to automatically create a header (.h) file that would provide the prototypes of all the functions located inside the library.

There are several reasons for using the MPLIB librarian for your PIC18 applications. They include

- 1** Helping to control the source code used in development builds. With certain object files only available within a standard library, you can guarantee that the application will not be built with invalid code sources.
- 2** The speed of the build will be better than in the case where you are bringing in multiple object files. This is especially true if the object files are found in various locations throughout your local area network.
- 3** The final size of the application code could be smaller. Libraries add a level of intelligence to the application build by including only the object files that are being called by the various functions within the application. Without a library, unused functions would be linked into the application along with the ones that are used.
- 4** If you have produced a product that can be programmed by third parties. The creation of a custom library would allow the users to access only the functions that you want them to access, and any intellectual property associated with the functions would be hidden from view.

After creating the custom libraries, you can add them for linking in your application by right clicking on the "Library Files" heading in MPLAB IDE's file view window. As with other files, the custom library could be in a folder outside the project files.

BOOTLOADERS

One of the basic functions of the first PC operating system, MS-DOS, was to load and start executing applications from some source (typically a diskette, but in the first PCs from audiotape). This function probably seems obvious and quite simple, but there are quite a few steps involved in loading the code into executable memory at an address that can be referenced, making sure that it is linked into the operating system APIs, allocating variable memory, and starting code execution. The complexity of these steps depended on what kind of application was being run on the PC and the system requirements of the code. The loading and execution of an application are a function that has been available to many different microcontroller architectures for years but is something that has become available only recently to PIC[®] microcontrollers.

The ability to load an application from some source, save it in memory, and then execute the code requires microcontroller program memory that can be changed by an executing program. PIC17 microcontrollers with external memory had this capability, but the systems designed around them were quite complex—what was needed was a PIC microcontroller “bootloader” application that had the ability to update its own internal Flash program memory to become available so that this function could be implemented in applications, eliminating the need for an ICD connector or pulling out the PIC microcontroller to update the application code. To summarize the important feature of a bootloader; it is a program that will allow a new application to be installed without requiring any special hardware.

Bootloaders may seem to be an unnecessary requirement because of the availability of MPLAB ICD, which performs many of the same capabilities as this type of tool, but they are still useful because there are situations where MPLAB ICD is not practical to use or is unavailable. If you had a problem with a robot and were at a competition, it probably would be much easier for you to connect a USB or serial cable between the robot and a laptop to download a new application (in the case where the competition calls for you to come up with a custom application for your robot). Other cases where a bootloader is preferable over MPLAB ICD is if the application is being used by someone who isn't

familiar with MPLAB ICD, or it is used in a hostile environment where there is danger of damage to the product or MPLAB ICD, and a simple, rugged host computer is the best choice for updating the application. Personally, a couple of years ago I would have dismissed the usefulness of bootloaders in applications, but I have seen a number of cases where they are extremely useful and more effective than other methods of updating an application.

Bootloader Requirements

Bootloaders have a basic set of requirements that are needed to perform their functions. When reviewing these requirements, remember that they are high level and not meant to direct you in a specific direction; a good example of this is PIC18C801 chips that have a bootloader function built into the chip that allows an external parallel Flash chip to be programmed via USB with no other code running. I'm noting this because in many of the smaller pin-out devices, you may see some advantages to implementing a bootloader, but the traditional resources that are available in chips such as the PIC16F877A (for which I demonstrate a bootloader later in this chapter) are not available, which may seem to eliminate a bootloader from being considered for the application. When working with the PIC microcontroller, many times when you are considering implementing a bootloader, you will discover that there are common requirements between the bootloader and providing the interface hooks for using MPLAB ICD.

The six basic requirements of a bootloader are

- 1** The application can communicate with the host system.
- 2** The host system has a mechanism for sending data to the PIC.
- 3** The PIC microcontroller must be able to write to its own memory.
- 4** The application code can execute without modification (except for reduced size for bootloader code).
- 5** If an application is loaded, the system should boot it automatically.
- 6** On startup, the system can be commanded to go into boot mode to allow the downloading of a new application.

Some of these requirements probably will seem obvious—such as the need for the application (both hardware and software) to be able to communicate with the host system. Despite being obvious, this capability is not something that will come automatically; along with making sure that input-output (I/O) pins are available for the bootloader, you must make sure that the schematic and the printed circuit board (PCB) have provisions for connectors and interface chips such as RS-232 translators and the software that supports this interface. For Logitech Harmony remote controls, bootloader functionality is implemented three ways: through MPLAB ICD, through the remote's USB connection, and using the NRZ UART interface built into the PIC18. Each of these interfaces is used in different aspects of development, manufacturing, and remote setup.

Just as the system needs to be able to communicate with the host, the host must be able to communicate with the system. Again, this is a system requirement, meaning that there must be hardware as well as software to take advantage of the bootloader functionality built into the system. With the trend toward simpler systems using PCs with more complex, proprietary, and licensed interfaces, this is becoming more and more of a problem. As will be shown later in this chapter, I will tend to stick with a simple RS-232-based (UART) bootloader using a basic terminal emulator program on my PC, but this is a slow method of passing new application code to the PIC microcontroller, and you might want to implement a USB-based bootloader, although the code is significantly more complex and will require a very good understanding of USB Windows device model programming as well as PC application programming. When you are first starting out with a bootloader, use a simple interface (such as the RS-232 bootloader shown later in this chapter) and develop the right technology for your application from there.

While MPLAB ICD capabilities are becoming more prevalent throughout the PIC microcontroller line card, there still are not that many part numbers that can write to their own program memory. This limits the devices that you can use in your application and makes it more difficult to select the part number that is best for your application. Unfortunately, there isn't a search parameter on Microchip's Web site that allows you to get a fast list of PIC microcontrollers that would support bootloaders, so you will have to go through the list of part numbers that have the basic I/O features that you are looking for and then go through each datasheet doing a search on the EEPGM bit of EECON1—this bit is used to select between data EEPROM and Flash program memory for reading and burning in new values. There also should be an EEDATH register that is used for transferring the high byte's data to and from program memory Flash.

When you create your bootloader code, it must be done with an eye toward not impeding execution of the actual application code. In the bootloader below I have moved as much of the code as possible to the "top" of the PIC microcontroller's memory and made sure that it does not overwrite the interrupt vector (address 0x0004). You also should keep MPLAB ICD in the back of your mind—you might have an application in which the bootloader is going to be implemented in the field, but you may want to use MPLAB ICD for debugging both the initial application code and the bootloader; to avoid any problems with MPLAB ICD, you should keep the last 256 addresses free.

In some cases it will be impossible to avoid placing the bootloader over interrupts. A good example of this is a bootloader that uses the USB port built into the PIC microcontroller. In this case you will have to do two things. The first is to ensure that there will be no application code addressed in this area, and the second is to write only applications that do not use interrupts. This does limit the applications somewhat, and you will have to decide whether or not the advantages make this course of action appropriate.

Keeping assembly application code from being placed in this area is easy to do using the `org` directives. For compiled code, this will require modifying the device `.1kr` file; below I have listed the modified `18f4450i.1kr` file that could be used for building an application that will be loaded by a bootloader.

```
// $Id: 18f4450i.lkr,v 1.1.2.2 2006/03/03 23:33:13 curtiss Exp $
// File: 18f4450i.lkr
// Sample ICD2 linker script for the PIC18F4450 processor

// Not intended for use with MPLAB C18. For C18 projects,
// use the linker scripts provided with that product.
```

LIBPATH .

CODEPAGE	NAME=vectors	START=0x0	END=0x7	PROTECTED
CODEPAGE	NAME=bootloader1	START=0x8	END=0x1FF	PROTECTED
CODEPAGE	NAME=page	START=0x200	END=0x2FFF	
CODEPAGE	NAME=bootloader2	START=0x3000	END=0x3DBF	PROTECTED
CODEPAGE	NAME=debug	START=0x3DC0	END=0x3FFF	PROTECTED
CODEPAGE	NAME=idlocs	START=0x200000	END=0x200007	PROTECTED
CODEPAGE	NAME=config	START=0x300000	END=0x30000D	PROTECTED
CODEPAGE	NAME=devid	START=0x3FFFFE	END=0x3FFFFF	PROTECTED
ACCESSBANK	NAME=accessram	START=0x0	END=0x5F	
DATABANK	NAME=gpr0	START=0x60	END=0xFF	
DATABANK	NAME=gpr1	START=0x100	END=0x1EF	
DATABANK	NAME=blpr	START=0x1D0	END=0x1F3	PROTECTED
DATABANK	NAME=dbgspr	START=0x1F4	END=0x1FF	PROTECTED
DATABANK	NAME=usb4	START=0x400	END=0x4FF	PROTECTED
ACCESSBANK	NAME=accesssfr	START=0xF60	END=0xFF	PROTECTED

The modifications to the `.lkr` file include changing the `vectors` code page to end at 0x7 instead of 0x29 and adding the `bootloader1` and `bootloader2` code pages, which are off-limits to the application code, along with `blpr`, which is used for the registers involved with the bootloader. The code produced by the linker should avoid these areas, and if there are insufficient resources (program memory or registers) to support the application and the bootloader, then you will receive an error during linking.

The final two requirements are really use cases: If there is no application loaded in, then the system should go into a state where it is ready to accept an application, and if there is one already, it can poll an external control that indicates whether or not there is a new application to be loaded. For a basic system, this control is simply a bit that is either high or low to indicate that the bootloader should go into application load mode.

Mid-Range Bootloaders

As I write this, there are just a few mid-range PIC microcontroller part numbers that allow a bootloader to be implemented, but these devices are quite comprehensive in terms of I/O features and can be used in a number of applications. I should point out that there are no devices that can use the USB port for passing data in—you are limited to using

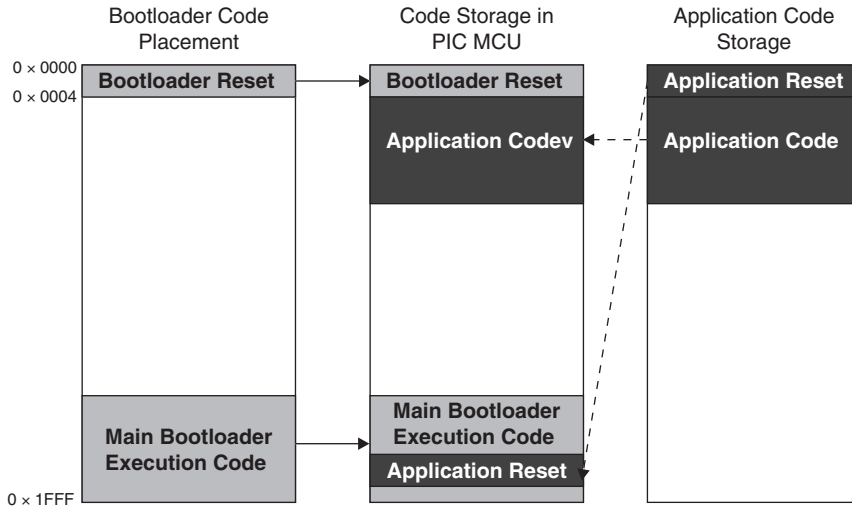


Figure 12.1 A mid-range bootloader must be designed to have minimal impact on the application code loaded into the system.

the serial port or a synchronous protocol of some kind. Despite this limitation, adding a bootloader as part of your application is something that you always should keep in the back of your mind, and even if you don't implement it, you can keep this functionality in your hip pocket for inclusion at a later point in time.

The model that I use for mid-range bootloaders is shown in Fig. 12.1, and in it I place the majority of the bootloader code at the “top” of program memory but also use the bottom four instructions. These four instructions mean that the bootloader will have to save the four instructions provided by the loaded application and be able to execute them from elsewhere in the application. The code with these instructions is at `StartVector`:

```
StartVector:                ; Otherwise, Execute the existing program
    bcf     STATUS, RP0     ; Make sure Bank Addresses == 0
    bcf     STATUS, RP1
    bcf     STATUS, IRP
    clrf    PCLATH          ; Jump to the starting address

ProgramStart:               ; Put the first 4 instructions here.
    dw      0x3FFF          ; Address 0
    dw      0x3FFF          ; Address 1
    dw      0x3FFF          ; Address 2
    dw      0x3FFF          ; Address 3

    goto    0x0004 | ($ & 0x1800)
```

When the application is being downloaded, the first four instructions are loaded into the four instruction locations starting at `ProgramStart`. When the application is executed, such as at boot up when the bootloader determines that there is an application loaded

and there is no request to load in a new one, execution jumps to StartVector, and the page and bank bits are reset. Then the four original instructions are allowed to execute. Note that there is a final `goto 0x0004 | ($ & 0x1800)` instruction that executes if in the first four instructions execution does not branch to some other location.

As indicated earlier, as this is being written, there are no USB-equipped mid-range PIC microcontroller part numbers that can support a bootloader through the USB. If you cannot use the UART port, then there are a number of other interfaces that can be used, including

- 1** I2C, with the PIC microcontroller as a slave device
- 2** SPI, again with the PIC microcontroller as a slave device
- 3** The parallel slave port (PSP)
- 4** A proprietary “bit banging” synchronous protocol

Of these four options, the one that probably would be most reasonable to use with a PC is to the PSP connected to the parallel (printer) port. When the printer port is configured for a basic dot matrix printer (such as the EPSON FX80), the data is strobed out continuously with just polling for a “busy” pin, which could be used to synchronize the transfer of the application file to the PIC microcontroller (waiting for the PIC to program the data into its program memory).

The need for synchronizing the write of program memory with the new data coming in is a very important point when implementing a bootloader in a PIC microcontroller. You probably will find that the program memory write will be slower than the data rate that you would like to use, and even more important, when program memory is being written, the PIC microcontroller’s processor stops executing, meaning that any incoming data cannot be processed, leading to the possibility of overwrites of the incoming data buffer. You always should make sure that you have implemented some kind of handshaking protocol between your bootloader-equipped system and the host application downloader.

SAMPLE BOOTLOADER CODE

Implementing a generic bootloader in the PIC microcontroller was surprisingly difficult, although I am very pleased with the final functionality of the result. A big part of the issue was designing the bootloader to work within the confines of the PIC16F877A microcontroller and with its Flash program memory burning algorithms; I was expecting that I would be able to write the code in 256 instructions, but it turned out that I needed about twice that to implement the function. The resulting application does work well and provides a method of downloading an application over a single 9,600-bps serial link using a basic Terminal emulator.

The development hardware for the bootloader that I used was a PIC16F877A running at 4 MHz with the serial port interface wired to an MAX232 and a button placed at RC1. The circuit was the same base circuit I used to develop the BASIC87 application (but using a PIC16F877A instead of a PIC16F877). The difference between the two devices made quite a bit of difference in the code because the PIC16F877A requires four instructions to be loaded before programming can commence.

To interface to this bootloader, I used my old standby Hyperterminal by Hilgreave (www.hilgreave.com) and wired the application to my PC via a serial port. Communication takes place at 9,600 bps and does not use any hardware handshaking, but the XON/OFF protocol is enabled. To send the updated application .hex file, the user simply clicks on “File Transfer” followed by “Send Text File.” This interface is very simple and very reliable, and when you look at the various RS-232-connected applications used in this book, you will see that I tend to use this interface almost exclusively.

The XON/XOFF protocol is used for handshaking and stopping the PC from sending code to the bootloader application if it is currently burning program memory. While at 9,600 bps the timing is such that the data cannot come in faster than the bootloader program could burn the program memory, as noted earlier, the processor is stopped when programming is taking place, so I felt that it was best to indicate to the PC to stop sending data until the programming operation had completed.

Before starting to develop the application, I created the pseudocode shown below and converted it into assembler (bl87a.asm, which can be found in the “bl87a” folder).

```
main(){

    if ((0 == RC1) ||                                // Button Pressed
        (0xFF == TestFlag))                          // No Program
    {                                                  // Load Hex File
        UARTInit();
        ProgramIns(TestFlag, 0x3FFF);                // Mark as Can't Use
        do {
            printf(XOFF);
            printf("Ready\n");
            ProgramGood = 1;
            printf(XON);                             // Enable datasend

            // Loop Until End of Program received
            do {
                StoreOffset = 0;
                do {
                    while (!CharReady);
                    DataChar = UARTRead();
                    if ((LF != DataChar) &&
                        (44 > StoreOffset))
                        DataBuffer[StoreOffset++] = DataChar;
                } until (CR == DataChar)

                printf(XOFF); // No Rx while data processed

                // Make Data into Hex
                for (i = j = 1; j < StoreOffset; i++, j += 2)
                    DataBuffer[i] = (MakeHex(DataBuffer[j]) * 16) +
                        MakeHex(DataBuffer[j + 1]));
                StoreOffset = i;                       // Compress data
            }
        }
    }
}
```

```
// Need At Least 9 Compressed Characters in Line
if (9 > StoreOffset)
    ProgramGood = 0;
else if ((0 == ProgramGood) ||
        // Skip to end if bad program
        (04 == DataBuffer[5]) ||
        // Skip Over First Line
        (01 == DataBuffer[5]))
        // Skip Over Last Line
{
    for (Checksum = 0, i = 1; 0xFF != DataBuffer[i]; i +=1)
        Checksum += DataBuffer[i];
    InsNum = DataBuffer[1] / 2;
    Address = DataBuffer[2:3] / 2;
    // Store in an instruction buffer
    for (i = 0; i < 3; i++)
        InsBuffer[i] = 0x3FFF;
    // Skip if Config, IDLOCS or Data EEPROM
    if (0x2000 <= Address)
        if ((':' != DataBuffer[0]) || // Stop if Error
            (0 != Checksum) ||
            (BootLoaderCodeStart <= (Address + InsNum)))
            ProgramGood = 0;
    else
    { // Program in Line of Data
        for (i = (Address & 3), j = 0; (j < 4) &
            (i < InsNum); i++, j += 2)
            InsBuffer[i] = DataBuffer[5 + j:6 + j];
        for (k = 0; k < 4; k++)
            ProgramIns(Address & 0x1FFC, InsBuffer[k]);
        if (InsNum > (Address + 4))
        { // Do Upper Bytes
            for (; (i < InsNum); i++, j += 2)
                InsBuffer[i] = DataBuffer[5 + j:6 + j];
            for (k = 0; k < 4; k++)
                ProgramIns(Address & 0x1FFC, InsBuffer[k]);
        } // fi
    } // fi
    printf(XON);
} // fi
} until (01 == DataBuffer[4])

if (ProgramGood)
{
    printf("Pass\n");
    ProgramIns(TestFlag, 0); // Mark as Can Use
} else
{
```

```

        printf("Fail\n");
        ProgramIns(TestFlag, 0x3FFF);           // Mark as Cannot Use
    } // fi

    } until (ProgramGood)
} // fi

goto StartVector;                             // Start Executing

} // niam

```

The `goto StartVector;` statement is not a lapse away from structured programming principles. It is the jump to the application's four statements at address 0x0 that was discussed in the preceding section. If this application was to be ported to C, then this statement would have to be manually converted to the suggested assembly-language statements.

This bootloader application allows downloading of applications very efficiently, and I have not seen a failure in the download process. This tool could be ported to the PIC18 architecture quite easily, with the only changes being to support the different program memory model and placing the buffer variables in a different bank from what the application would be expected to use.

PIC18 Bootloaders

Implementing a PIC18 bootloader has many of the same issues as the mid-range chips, but there are some unique situations that you should be aware of. Obviously, the differences in the program memory and register architecture are things that you are going to have to understand, but there are also interfaces that are available on the PIC18 that aren't available in other devices, as well as the ability to implement a *real-time operating system* (RTOS, which is discussed in Chap. 13) that will affect your use model and what kind of capabilities you are going to build into your system.

While this is common in many other microcontrollers, the PIC18C601 and PIC18C801 are the only members of the PIC microcontroller family that are shipped without any memory. These chips are designed to be driven by external parallel Flash chips, and implementing a bootloader in them is accomplished in a rather unusual manner, relying on 1.5 kB of SRAM that is located in the PIC microcontroller itself that can be accessed either as file registers or as program memory. This SRAM, generally known as BootRAM, is used to store the bootloader program while the Flash is being updated.

The process of downloading a new application in the PIC18C601/PIC18C801 using a bootloader is as follows:

- 1** On startup, have a recognized bootloader application load indicator active.
- 2** Configure BootRAM as file registers.

- 3** Copy bootloader code from existing Flash contents into BootRAM.
- 4** Configure BootRAM as program memory.
- 5** Change execution to BootRAM.
- 6** Indicate to the host that the system is ready to receive the new application code.
- 7** Copy new application code into external Flash.

The PIC16C601 and PIC16C801 require the external Flash to be loaded initially with your bootloader code, but this can be accomplished by using MPLAB ICD or some other ICSP programmer. Built into these chips is code that will execute with the ICD program download, which will accomplish programming of the Flash chip without any additional effort on your part. Once the external Flash has been loaded using ICSP, you can use a bootloader that will allow you to use another port such as the serial port or USB to update the external Flash. There are newer PIC microcontrollers that have internal Flash and can access external parallel Flash (such as the PIC19F87J50). These chips can have their bootloaders stored in the internal Flash, and they can be used to update the external Flash.

REAL-TIME OPERATING SYSTEMS

One of the most useful application programming environments that can be used in a microcontroller is the *real-time operating system* (RTOS). When used properly, an RTOS can simplify application development, and by compartmentalizing the different execution threads of an application, it provides the opportunity to reduce errors significantly. This option has not been available to PIC microcontrollers until the initial availability of the PIC18 MCU family. This PIC[®] microcontroller family allows access to the processor stack, which then can be modified with different “task” data. This feature is not available in other PIC microcontrollers and will be taken advantage of in the PIC18 for the example RTOS that I will present in this chapter.

The best definition I can come up with for a RTOS is “. . . a program that controls the execution of multiple ‘threads’ of an application in a computing system by prioritizing their execution and allowing them to communicate with each other.” *Thread* is the term used to describe the individual *subtasks* (of just *tasks*) of an application and is analogous to an individual thread of conversation between two people in a room. There may be multiple threads, but each one is following its own path. An important point about threads is that if the operating system is executing more than one of them simultaneously, then it is known as a *multithreaded operating system* or a *multitasking operating system* (often abbreviated to *multitasker*).

There are two primary types of multitasking operating systems: The *event-driven* operating system changes only the executing thread when some kind of hardware event occurs, whereas the *time-sharing* operating system stops each thread after it has executed for some length of time and passes execution on to another thread. For the RTOSs that I have written, I like to implement a hybrid of the two: If the currently executing thread is not stopped by a hardware event, a timer event will stop it and start up the next thread in the queue. An event-driven operating system is used primarily in microcontroller interfacing applications, whereas a time-sharing operating system is used in mainframes where CPU time is shared between users.

You may have heard the term *processes* when multitasking operating systems are discussed. In some operating systems, the word *process* is interchangeable with *task*, but

the most common usage is in PC operating systems, in which a process is an application running in the operating system that is composed of multiple threads that perform the necessary operations of the application. When discussing PIC microcontroller RTOSs, in which there is only one application running, it is common to use the terms *process*, *task*, and *thread* interchangeably because there is normally only one application running at a time in the chip.

Similarly, the term *device driver* loses its meaning when applied to an RTOS designed for a chip such as the PIC microcontroller. In a PC, a device driver is a low-level code that provides an interface to hardware peripherals and the threads of the operating system and application processes. The purpose of a device driver is to provide a common set of application programming interfaces (APIs) to the application and operating system that is independent of the hardware. In a much smaller system, the need to provide a common API is essentially not required because the peripherals to be accessed are built into the chip, and there is little need or opportunity to use different peripheral hardware in the system. Unlike processes, tasks, and threads that become synonymous in a single-chip RTOS, the term *device driver* is not used at all when describing the code that provides an interface to the hardware in the system; instead, single threads are used to provide hardware access for the application.

I tend to lump both an operating system and an RTOS together because the central *kernel* (the part of the operating system that is central to controlling the operation of the executing tasks) is really the same for both cases. The difference comes into play with the processes that are loaded initially along with the kernel. In a PC's operating system, the console input-output (I/O), command interpreter, and file system processes usually are loaded with the kernel, and everything has been optimized to run in a PC environment (which means they respond to operator requests). In an RTOS, the actual application tasks are loaded in with the kernel, with priority given to tasks that are critical to operation of the application.

You might be a bit suspicious of an RTOS after what I've just written. After all, you probably have a PC running Windows/95 or Windows/NT, and you are probably familiar with problems working with different pieces of hardware or software applications. I would be surprised if somebody reading this book had never had a problem with Windows not coming up properly, crashing when you least expect it, or hanging up and not responding to input or displaying a "blue screen of death." Along with these problems, these operating systems require literally hundreds of megabytes on a hard drive to operate. With this background, you're probably wondering, How can a multitasking operating system be implemented in an 8-bit microcontroller with only a few kilobytes of program memory? Another question you might be asking is, What features does the PIC18 microcontroller have that would make me feel like I would want to invest the time and effort into developing a multitasking operating system for it?

To answer these questions, I would look at it from the negative and ask what the PIC18 *doesn't* have compared with the PC. The smaller "system environment" of the PIC18 is what is important and makes an RTOS very appropriate for this type of device. The PIC18 does not have a file system, arbitrary amounts of memory (including "virtual memory" models), or the sophisticated user interface (unless you want to provide one yourself in your application). In addition, in the microcontroller's case, you can

specify the interfaces and hardware rather than come up with standards that may have to be “bent” when new technology becomes available or may be interpreted incorrectly by other developers.

To make matters worse for the PC, there are literally millions of programmers working on the Windows operating system, drivers, and applications. With the more code that is written, the greater is the chance that incompatibilities between the system, drivers, and applications will occur, making it more likely that there will be problems with everything working properly together. In a small microcontroller, the team of people developing the application and the interface software is very small, and the chances for incompatible application software and hardware interface code not being detected and being passed along to the end user is much more remote.

To illustrate how a multitasking RTOS works, I am going to present a web surfer’s PC and show how the different features and software interconnect to provide a method of connecting to the Internet. Instead of having multiple computers, each one providing one function; the PC makes functions (known as *resources*) available to the different applications, as shown in Fig. 13.1.

In a real system, the different tasks (which are represented by different boxes in Fig. 13.1) would be given “priorities.” This reflects the importance of messaging and operation of the specific tasks relative to the others. This means that if high- and medium-priority tasks are waiting to execute, the high-priority task will run first. In most PCs, the Modem task would be given a specific priority because it is the resource that is the most constrained by the speed at which it can operate and the demands by the different applications for the available bandwidth.

In periods of inactivity (between the surfer’s keystrokes with no web pages being currently downloaded), low- or medium-priority tasks are executed because the high-priority tasks are not required. Low-priority tasks normally are interactions with the operator because if a task takes a long time to execute (in computer terms), a human working with it probably won’t notice any delays in the computer’s operation.

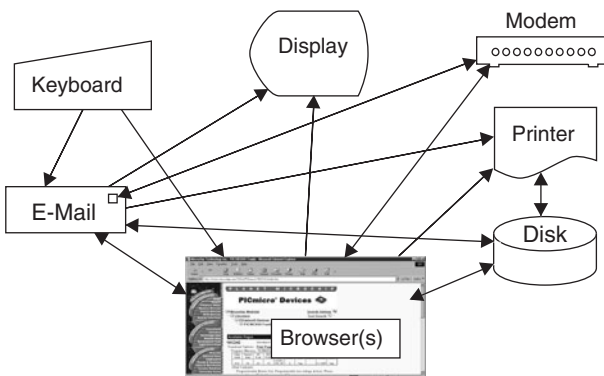


Figure 13.1 The major tasks and hardware interfaces that are needed to implement a web-surfing application.

All the tasks can be arranged in a diagram to show how they communicate via messages. In Fig. 13.1, the arrows represent the directions messages move in. An important concept about messaging in an RTOS is that each message is initiated by a task. Normally, each task is waiting (*blocking*) on a message, waiting to respond to it and execute the request that is part of the message. After the task has executed its function, it then may respond to the conditions by sending messages to other tasks.

For example, to send e-mail, the Modem task could execute the code

```
ModemSend()                                // "Modem" Data Packet Send Routine
{
    char * Packet;                          // "Packet" to be Sent

    while ( 1 == 1 ) {                     // Loop forever

        while (GetMsg() == 0);             // Wait for a Message to Send Out

        while (ModemBusy != 0);           // Wait for the Modem to be Available

        Packet = ReadMsg();                // Read the Sent Packet

        SendMsg(Packet);                  // Send the Packet over the Internet

        AckMsg();                          // Acknowledge the Original Message

    } // elihw

} // End ModemSend
```

This task will wait for a packet to be passed to it that will have to be sent to the Internet. If the modem is already being used (i.e., a previous packet is being sent or a packet is being received), it will wait for the modem to become free before sending the packet. Once the packet is in the queue to be sent by the modem, the ModemSend task "acknowledges" the original request to send the packet and waits for the next request to come in.

This is a grossly oversimplified example of what actually happens, but it should give you the idea that tasks send request messages to other tasks to request that a function be performed. The receiving tasks only process the request when they are able to and acknowledge that they have done so when the resources to do so are free.

I should point out one aspect of tasks and processes in RTOSs that may be not be readily apparent. Tasks should access only one hardware device. Thus the SendModem task in the preceding example only accesses the modem output functions. To access other hardware in the system, the RTOS application will send messages to the appropriate tasks that control the different hardware.

This is an important philosophical point about access and RTOSs because, obviously, applications could be written that accessed multiple hardware interfaces. This

allows code reuse or modification and allows multiple software developers to work on an application without having to understand how the other interfaces work. At first, it may be unnatural for you to think in terms of simple, single devices, but once you get the hang of it, you'll be amazed at how easy it is to work with an RTOS.

The last aspect of RTOSs that I want to present to you is the *semaphore*. The semaphore is a flag that is controlled by the operating system that can be used to restrict access to a resource (which can be a hardware device, data, or even another task) until the “owning” task has finished with it. Semaphores are used in situations such as controlling access to an operator console. In the `SendModem` task above, a semaphore could be used to indicate that the `Modem` hardware is available for a packet transmission or that the modem is involved in a data packet transfer.

Low-End and Mid-Range RTOSs

It is not possible to create a full-featured RTOS for low-end and mid-range PIC microcontroller architectures. The reason for making this sweeping statement is the two architectures' inability to save and restore program counter stack values—the PIC18 architecture has this ability and is able to have RTOSs developed for it. There are some tricks that you can use to create an application that mimics the operation of an RTOS, but they do not provide a truly satisfying RTOS and may make application development more complex rather than simpler, which is what is expected when development is carried out well in a multitasking operating system.

The most likely method for implementing an RTOS in low-end or mid-range devices is to use a ring execution model that could be described in pseudocode as

```
while (1 == 1) {
    for (i = 0; i < NumberOfThreads; i++)
        if (ThreadActive[i] || MessageWaiting[i])
            Thread[i];    // Execute the next waiting thread

    INTCON.GIE = 1;    // Enable Interrupts
    NOP;                // Service waiting interrupt request
    INTCON.GIE = 0;    // Disable Interrupts
} // elihw
```

In this code, each thread is polled to determine whether or not it can execute, and if it can, then control of the processor is passed to the thread until it finishes and then checks the next thread. There are a few issues with this model, not the least of which is the time required to poll each of the different threads, as well as the latency involved with going ahead and servicing the interrupt request. Another issue is that each thread is an autonomous subroutine with no initialization code (because it would be executed each time the thread executes). The best way to deal with initialization would be to create an initialization thread for each main thread, and on application startup, the initialization thread is set to execute, and at the end of the initialization thread, its ability to execute (`ThreadActive == True`) is terminated.

If there is a need to execute threads asynchronously in order to respond to interrupts, you may want to resort to building up multiple polling routines. For example, in the TAB Electronics Build Your Own Robot Kit, I was faced with the need to perform six tasks while implementing the motor PWM loop:

- 1** Time PWM duty cycle as required.
- 2** Send out IR data modulated at 38 kHz for collision detection.
- 3** Poll IR receivers and indicate when there is an object close by.
- 4** Poll IR receivers and indicate when an infrared remote-control command was being received.
- 5** Poll the BS2 synchronous serial clock and data lines and receive data from the BS2.
- 6** Update real-time clock.

Implementing these tasks simultaneously in an event-driven RTOS would be quite simple, but it was a significant challenge in the robot because these tasks were implemented in a PIC16C505, a low-end device with no interrupts. To do this, I spent many hours carefully architecting the PWM loop and ensuring that polling of the different interfaces was done at appropriate intervals and that no changes in the incoming data were lost. Once this was done, I spent 6 weeks (literally) on the simulator, working through every possible scenario, to ensure that there could be no situations where the application would not work as expected. The code was very successful, and no problems were reported in the field.

Unfortunately, this effort could have been minimized significantly by implementing this application in a device that could support an RTOS. The PIC16C505 was chosen for cost and not for functionality, and while it gave me a very good price for the manufacturing of the robots, it took significantly more effort than I would have expected for a similar application created for an RTOS. This is a reiteration of the one of the central tenets for developing application code for an RTOS—it is significantly easier to produce high-quality code for an RTOS than it is using standard programming techniques.

PIC18 RTOS Design

It is possible to implement an RTOS in the PIC18 architecture owing to the processor's ability to access the program counter stack. This feature allows the return address context information that is required to restore execution of the thread to be read or written by the RTOS task-switching code. Unlike other processors, the PIC18 does not place all context data onto the stack, which means that other means will have to be used to save the context information and then restore it into the appropriate registers. With a strategy that allows the saving of the return address information along with the context registers, you should be able to create an RTOS for the PIC18 architecture quite easily.

There are a number of ways to save the context registers to ensure that the thread switching is fast and that the appropriate registers are saved and then retrieved. Probably the simplest way of doing this is to access all “thread local variables” via an index register with an offset; by doing this, the context registers (including the saved program

counter stack information) are “pushed” onto the stack using the POSTDEC INDF register, and when it is time to change the thread, the index register pointing to the stack information is changed to the next thread. A slower and more cumbersome way would be to copy all the thread context registers (including the program counter) into a memory space that is unique to the thread. Saving and restoring the context registers is the most important aspect of the design of the RTOS; you must have a methodology that is both efficient and reliable—the task switch must be fast and accurate, or your applications will not execute properly, and this is a very difficult bug to find and fix.

I must point out that the PIC18 architecture does not implement a memory management unit (MMU) that will flag conditions where memory is accessed inappropriately or attempts to execute beyond the confines of the thread. In most modern 32- and 64-bit processors, the MMU will provide protection between processes to ensure that threads do not overwrite critical portions of memory. Another important feature of an MMU is to move physical memory in and out of specific address ranges of the executing thread (this is known as *virtual memory*), and this allows the file system to load information for a thread in convenient locations and not have to relocate memory to make space for new information. The MMU function as relating to program memory is really not required in the PIC18 because the program memory, for all intents and purposes, is hard-coded and cannot be changed during application execution—file and special function registers should have some kind of protection in the PIC18 to ensure that they are accessed appropriately.

When you are designing your own RTOS, you have to be careful of the “fast stack” built into `call` and `return` instructions, as well as executing in interrupt handlers. The registers saved on the fast stack are critical context registers, but you will find that coming up with a way to read these values from the fast stack will be more difficult and take more instructions than simply saving the register contents directly, as you would in an interrupt handler. It is easy to get lazy and rely on the fast stack for subroutines, but this is a habit that you should break in the case where a thread is suspended during a subroutine and the contents of the fast stack are overwritten by even interrupt requests.

Finally, you will have to have a plan for the access bank file registers. These 128 file registers are very useful to use as general-purpose registers, and it would be easy to use them in your everyday programming, but you will have to take care as to how you use them in an RTOS situation. If you were to use them in the threads of your application, you would have to save these registers as part of the thread context information. This is a relatively lot of data to save and restore as part of the task-switching operation, and it should really be avoided at all costs. Using a strategy such as only employing a stack pointer for local thread variables avoids the need for using the access bank file registers and will help you to keep the movement of your context-saving information to an absolute minimum.

In the following sections I will outline the design for a PIC18 RTOS and provide you with a simple application to demonstrate operation of the threads. It is important to remember that there is not only one way to implement an RTOS for the PIC18 (or other processors, for that matter), and I recommend that you spend some time researching the requirements of your application and work through a plan to switch between tasks and manage your file registers most efficiently.

RTOS01: SIMPLE PIC18 RTOS

When I originally set out to develop an RTOS for the PIC18, I had a number of ideas in mind. These were based on the RTOSs that I had written for other devices with the philosophy that what I came up with would have to be quite simple. The goals I gave for myself were

- 1 Required program memory for the application should be less than 256 instructions.
- 2 There should be fast task switch capabilities (less than 0.5 ms when running at 4 MHz).
- 3 Sixteen bytes or less of “overhead” file registers should be required.
- 4 Eight tasks could be accomplished with an initial `AllTask`.
- 5 Tasks would execute as if they were the only application in the PIC microcontroller.
- 6 Each task could have at least 32 bytes of variable space.
- 7 The RTOS would support at least one subroutine call.
- 8 Interrupts would be supported.
- 9 Application tasks would be assembled with the RTOS code.
- 10 There would be a single operating system entry point.

The result can be found in the `rtos` subdirectory of the downloaded PIC microcontroller folder. This is a reasonably full-function RTOS with a simple application that increments a series of light-emitting diodes (LEDs) on a PIC18C452’s `PORTB`. The application code was developed to test out the different functions of the PIC microcontroller. I will discuss the application in more detail in this section and present a more complex application at the end of this chapter.

To achieve a very small program memory requirement for the RTOS, I wanted to make sure that I had optimized the application as much as possible for the PIC18. I was able to write and debug the code over the space of about 12 hours, but the design of the RTOS was done over the space of a couple of weeks. This time spent in design and doodling resulted in fairly small code but what I feel is a reasonably elegant RTOS that is well suited to the PIC18 and offers good protection for the application developer.

Since the RTOS was the first “full” application I implemented on the PIC18, this was to be a learning experience for me. It probably sounds risky to do an entire operating system as a first application, but I actually used this occasion to make sure that I understood how the PIC18 worked and how the instructions execute. Making this more of an adventure was my use of an “interim” upgrade to MPLAB and “engineering sample” PIC18 when I first wrote the code. This was actually a right-sized application to learn on because it challenged me to learn the processor architecture completely but didn’t require an extremely large application.

In terms of measuring against requirements, the final RTOS weighs in at 748 instructions, requires only 8 bytes of overhead to run eight tasks, and executes a task switch in about 750 μ s with the PIC microcontroller running at 4 MHz. It may sound like I blew the RTOS size specification, but owing to the relatively large amount of program memory available to the PIC18 (I used a PIC18C452, which has 16,384 instructions) and the ability to multiply up the clock, these overages are really not that significant to the total memory available, and the speeding up brings the task switching well within the target time.

When you write RTOS applications, you will discover that the actual application code is quite small. The test application that is included with the RTOS requires only 26 instructions to create an application that starts up TMR0 to interrupt once every 16.384 ms and uses a counter to count down 64 TMR0 interrupts and increment the LEDs.

When I first learned about RTOSs, I was told that the maximum task-switching time should be 500 μ s. In this RTOS, I ended up taking 750 μ s at 4 MHz. I don't consider this to be a major concern because a 4-MHz clock is actually quite modest, and to meet the 500- μ s requirement, I could simply turn on the four times clock PLL built into the PIC18's oscillator circuit or add a faster clock. This may sound somewhat facetious, but I have found over the years with reasonably simple microcontroller RTOSs and properly designed and written applications that switching times of 2 or 3 ms can be tolerated without any problems by most applications.

When tasks are created for this RTOS, they execute as if they are the only task running in the PIC microcontroller. To facilitate this, I made the assumptions listed below about how the application tasks would work. This results in a somewhat simplified "view" of the PIC18 but one that is more than offset by the capabilities offered by the RTOS.

The basic assumption I made was that the application would only access the first 63 file registers of the "access bank." When I created the RTOS, I wanted to avoid having "global" variables as much as possible and instead to take advantage of the ability of the RTOS to send messages back and forth.

Sixty-three file registers is actually quite generous and will allow you to develop tasks for most applications with very few problems. The number of file registers is specified when the task is "started." For many applications, you will discover that quite a few tasks require only one or two file registers, which will allow you to run the RTOS quite easily in PIC18 devices with 512 file registers.

At the "top" of the memory space in the PIC18s with 1,536 file registers, you could put in some global variables, but I suggest that you place them starting at address 1,024 to make sure that none of the tasks can access them.

When the RTOS executes, you will notice that the file registers dedicated to the task are placed at the first beginning of the "access bank" and even the PIC microcontroller's file register memory itself. This was done to further simulate the perception that the task was the only code running in the PIC microcontroller.

The 23 special function registers that are specific to each task are located in the second 128 bytes of the "access bank" and are listed in Table 13.1.

The limitations are the lack of a third index register (FSR2) and the fact that restricting the stack depth to only three "pushes" could not be used by the application because it is changed during the RTOS's execution. I will discuss the stack in more detail below.

When a task is not executing, these 24 registers are read out of the PIC microcontroller's special-function register area and are copied into what I call the *task information block* (TIB). This data structure, which is provided for each task, consists of the 24 special-function registers listed above along with the variable file registers discussed earlier.

To make moving data back and forth between the TIB simpler in the RTOS, I copy the 24 bytes of tasks of the special-function registers into address 0x060 to 0x077 of

TABLE 13.1 PIC18 RTOS TASK CONTEXT REGISTERS	
REGISTER	COMMENTS
WREG	
STATUS	
BSR	
PCLATH	
PCLATU	
TBLPTRH	
TABLAT	
PRODL	
PRODH	
FSR0L/FSR0H	
FSR1L/FSR1H	
TOSL/TOSH/TOSU	Top three program counter “pushes” on the stack

the access bank registers. This allows me to set or restore the special-function registers without having to cross bank boundaries (which simplified development of the RTOS considerably). This is probably the largest area of inefficiency in the RTOS. If the TIB’s special-function register area would pass data to and from the saved TIB rather than the access bank copy, quite a few instructions and cycles could be saved in the RTOS’s execution and task switches.

The ability to access the program counter’s stack in the PIC18 is what sets this architecture apart from the others and makes the RTOS possible. For this application, I only save the top three elements of the stack and not the total 31. The primary reason for doing this is to save file registers. If all eight possible tasks saved all 31 program counter stack elements, then 744 bytes would be required. By cutting the saved value down to only the top three, I only require 72 file registers to save the task’s stack. Saving the top three program counter stack elements means that the task can be nested down two levels of subroutines when the RTOS is active. This means that the task’s mainline can call a subroutine and that subroutine can call another subroutine. Recursive subroutines cannot be implemented at all in the RTOS.

There is one case to watch out for, and that is when a new task is “started.” When the `TaskStart` macro is invoked, it “pushes” the starting address onto the stack. I did this because it allowed me to simply copy data from one TIB into a new one that I had set up for it. It also meant that I would not have to make space in any other registers when the RTOS is invoked.

TABLE 13.2 RTOS STATUS AND MESSAGE BYTE

BITS	FUNCTION
7–6	Task operating mode: 11—able to execute 10—task is waiting for a message to be sent 01—task has sent a message and is waiting for it to be acknowledged 00—task is waiting for an interrupt request
5–4	Task priority 11—highest : 00—lowest
3–0	Task operating indicator, specific to the “task operating mode bits” 7–6 Task operating indicator bits function 1x Bits are all zeros 01 Bits contain the task the message has been sent to 00 Bit ##

The first byte of the TIB is the *task byte*. This byte is used to indicate to the RTOS the status of a specific task. The task byte is defined in Table 13.2, and the interrupts supported for the task to wait for are listed in Table 13.3.

All the possible PIC18 interrupt request sources are accounted for except for the INT1 request. This request was left off because it is put in a register that doesn’t follow the conventions of the other bits, and I wanted to have a “special purpose” value that is used to flag the RTOS that an operating system request is required.

The interrupts are run in normal, not priority, mode. Priority mode *never* must be enabled with the RTOS because the code is designed for a single entry point for interrupts and code. Enabling interrupts will cause a second entry point that will result in task data being saved incorrectly when the interrupt request is acknowledged. When using interrupts with this RTOS, make sure that you only access the E bits to set them to enable the interrupt request.

When an interrupt request is acknowledged by the RTOS, the E flag is also reset by the RTOS, except in the case of TMR0. For all interrupts except for TMR0, the tasks will have to enable the interrupts and reset the hardware after the request has been acknowledged. This means setting up the hardware and setting the E flag to enable the interrupt.

TMR0 is enabled within the RTOS and will interrupt the application once every 16.384 ms (when running with a 4-MHz clock). This interrupt can be “waited” on, but

TABLE 13.3 INTERRUPTS SUPPORTED IN RTOS		
INTERRUPT NUMBER	INTERRUPT SOURCE	REGISTER
0	TMR1	PIR1
1	TMR2	PIR1
2	CCP1	PIR1
3	SSP	PIR1
4	TX	PIR1
5	RC	PIR1
6	AD	PIR1
7	PSP	PIR1
8	CCP2	PIR2
9	TMR3	PIR2
10	LVD	PIR2
11	BCL	PIR2
12	RB	INTCON
13	INT0	INTCON
14	TMR0	INTCON
15		Special purpose

TMR0IE never should be reset by a task. The purpose of the TMR0 interrupt is to stop an application from taking up too many processor cycles and “starving” the other tasks. If you have a task with a large amount of data processing or input-output (I/O) polling, I suggest that you place it at as low a priority as possible to allow other tasks the chance to execute.

During a task’s execution, the INTCON GIE bit can be reset for critically timed code or a high-priority operation. I do not recommend disabling interrupts for more than 20 instruction cycles because the longer interrupts are disabled, the better chance there is that an interrupt will be missed or overwritten. The RTOS code only responds to one interrupt at a time and responds to them in the order given earlier. Any delays in acknowledging an interrupt can result in problems later in the application.

The RTOS requests are listed in Table 13.4 along with the registers (and bits) that are changed. Each request is actually a macro, which was done to eliminate the need for keeping a reference for how to call the RTOS correctly.

When the RTOS is booted, it will start `AllTask`. This task has two purposes. The first is to start up the application code. To do this, it invokes the `TaskStart` macro to start up `Task1` at a priority of 1. `AllTask` has a priority of 0, and once `Task1` is

TABLE 13.4 RTOS TASK REQUEST DEFINITION

RTOS TASK REQUEST	INPUT PARAMETERS	OUTPUT PARAMETERS	INVOKING MACRO
Start task	Starting address Priority (0–3) file register requirements	Carry reset, OK, and new task # in WREG Carry set, NO available or more than 63 task variables	TaskStart Address, Priority, Size Address, Priority, and Size are all constants.
Execute next task	None	None	TaskNext
Wait for interrupt	Interrupt number	None	IntWait Interrupt Interrupt is a constant.
Send message	Task number, 2-byte message in FSR0L/FSR1L	None	MsgSend TaskNumber TaskNumber is the address of two consecutive file registers.
Wait for message	None	2-byte message in FSR0L/FSR1L TaskNumber in WREG	MsgWait
Read message	Task number	Carry reset, 2-byte message in FSR0L/FSR1L Carry set, no message to get	MsgRead TaskNumber TaskNumber is the address of a file register.
Acknowledge message	Task number	Carry reset, operation complete Carry set, no message to acknowledge	MsgAck TaskNumber TaskNumber is the address of a file register.

executing, it will just loop as the lowest-priority task in the RTOS, providing a lowest level of functionality for the RTOS. The code I used is

```

_AllTask:                                ; Always Return Here

TaskStart Task1, 1, 63                   ; Start Application Code

_AllTask_Loop:

TaskNext                                ; Jump to Next Active Task

bra    _AllTask_Loop

```

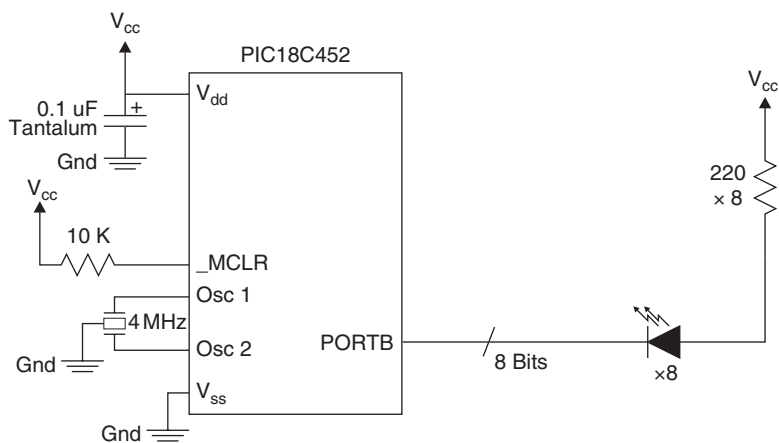


Figure 13.2 This simple circuit, using the parts listed in Table 13.5, will allow you to demonstrate the operation of a basic RTOS.

The `AllTask` looping may not seem like an important function, but it is because when the RTOS code is active, all interrupt requests are disabled. If all the other tasks in the PIC microcontroller are waiting on interrupt requests or messages, then `AllTask` will be the only one able to execute, and when the `branch always` instruction executes, interrupts are enabled. In other operating systems, this function is known as *system idle* and executes when all other tasks are waiting on some kind of event.

`Task1` is the first task of the application code. This is a “standard” label that is used to indicate the application code. Note that I have given it a priority of 1 and the full 63 file registers for variables. 1 is the lowest priority that your application should have to ensure that `AllTask` performs its function properly.

When I created the RTOS, I wanted to use it with a simple application. As I mentioned earlier, this application simply waits on the timer and updates LED values once per second. The circuit that I used is shown in Fig. 13.2 and is part of the `rtos01.asm` application that is found in the `code\rtos` subdirectory of your PC’s “PIC microcontroller” directory.

The bill of materials for the project is listed in Table 13.5.

TABLE 13.5 RTOS EXAMPLE APPLICATION BILL OF MATERIALS	
PART NUMBER	DESCRIPTION
PIC18C452	PIC18C452–10/JW
Capacitor	0.1-μF tantalum capacitor
R1	10 kΩ, 1/4 W
R2–R9	220 Ω, 1/4 W
Y1	4-MHz ceramic resonator with built-in capacitors
Misc	Prototyping board, +5-Volt power supply, wiring

The application code is

```

CBLOCK 0x000                                ; Put the Variables Starting at 0x000
LEDTask                                     ;
Counter                                   ; Number of Times Executing
LEDValue
ENDC

Task1:
    clrf    Counter, 0                        ; Clear the Variables
    movlw   0x0FF
    movwf   LEDValue

TaskStart LEDTaskStart, 1, 2                ; Put in the LED Executing Task
    movwf   LEDTask                          ; Save the Task Number of LEDTask

    movff   LEDValue, FSR0L                  ; Save the LED Task Value
    MsgSend LEDTask
    nop

Task1Loop:                                ; Loop Here Until 64x Past

    IntWait 14                              ; Wait for the Timer0 Interrupt

    incf    Counter, f, 0

    movlw   64                              ; Done 64x?
    xorwf   Counter, w, 0
    btfss   STATUS, Z, 0
    bra     Task1Loop:                      ; If Not, Loop Around Again
    movwf   Counter, 0

    decf    LEDValue, f, 0                  ; Increment the LEDs

    movff   LEDValue, FSR0L
    MsgSend LEDTask

    bra     Task1Loop

LEDTaskStart:                              ; Make PORTB Output and Wait for
; Messages

    CBLOCK 0x000                            ; Variables
    LEDTaskNumber
    ENDC

    clrf    TRISB, 0

```

```
LEDTaskLoop:                                ; Loop Here for Each Character

MsgWait
    movwf    LEDTaskNumber, 0                ; Save the Task Number

MsgRead LEDTaskNumber
    movff    FSR0L, PORTB                    ; Update the LED Value with the Message

MsgAck LEDTaskNumber                         ; Acknowledge the Message

    bra      LEDTaskLoop:
```

In this code, `Task1` first enables `LEDTaskStart`, which puts all the bits of `PORTB` into “output mode” and then waits for a message to update the LEDs. `Task1` saves the task number returned for `LEDTaskStart` and then sends a message to `LEDTaskStart` to turn off all the LEDs. Next, `Task1` waits for the 16.384-ms timer to overflow 64 times (which takes approximately 1 second) and then updates the LED value and sends a message to `LEDTaskStart`.

This is a very simple application, but there are a number of things to understand in it. The first is the `Task1` save of the `LEDTaskStart` task number. In this application, I have saved the value locally in `Task1`. For many other applications where a task provides a central resource to the complete application, you probably will want to make the task number available globally within the PIC microcontroller.

To do this, the variable should be placed at the end of the file registers to make sure that there isn’t a conflict with any task variables. Each task has its own unique “bank select register,” so access can be made by one task without affecting any of the others.

In the code, note that I only access the variables using the access bank (the , 0 at the end of variable/register access instructions). This is an important thing to remember to do in your PIC18 applications. Forgetting to do it can cause some problems with accessing data in an unexpected manner.

In the preceding code, note that I placed a `nop` instruction after the first `MsgSend` macro invocation in `Task1`. The reason for doing this is to provide a place to hang a breakpoint onto. You cannot place a breakpoint at a macro invocation, so I put in the `nop` to allow my code to break after the `MsgSend` to be able to go back and take a look at what happened in the application. This seems to be a trick that not many people know about and can save you frustration later when you have an application with a lot of macros with straight assembly-language instructions in between them and you want to understand what is happening.

Lastly, the `MsgRead` macro invocation in `LEDTaskStart` is unnecessary. This was done to test out the function, and `MsgWait` will return the same information.

This is the fourth RTOS that I have presented in my books, and this one has one important difference from the others. In the previous RTOSs that I have created (for the Motorola 68HC05 and Intel 8051), I put a lot of emphasis on placing the RTOS and task variables where I could find them easily and translate them to debug the application. For this PIC18 RTOS, I placed the emphasis on just making the task’s variable space

easy to look at. This made creating and debugging the RTOS somewhat more difficult because I had to leave the thread of the debugging to move the “program memory” window back and forth to see what was going on.

The “File Register Window” in MPLAB is shown in Fig. 13.3, and you can see how only the first 192 file registers can be shown with the source file. These 192 file registers are enough for the variables used within the tasks and the 24-byte special-function register block, as well as the first two tasks’ (`AllTask` and `Task1`) TIBs. Looking at the figure and finding specific values, other than the first three variables of `Task1`, at addresses 0x000 to 0x002 is not something that is easy to do.

This was a compromise that I made when I designed the application. For people who are developing applications, this display is not a problem (and in fact, it works very well to see what is happening in a specific task). When you are designing your own applications, you always should think about debugging and how data will be displayed to you and what you can do to make it easier to understand.

RTOS02: MULTITASKING APPLICATION EXAMPLE

The first example application in this book is a digital clock and thermometer that runs on a PIC18C452 with a Dallas Semiconductor DS1820 temperature sensor and a Hitachi 44780–based two-line liquid-crystal display (LCD). The circuit used for this application is shown in Fig. 13.4.

If you have read the *TAB Electronics Microcontroller Handbook*, this application probably will look pretty familiar to you. I used this application to demonstrate how different microcontrollers would implement the same application. For this book I am using this application to show how a “typical” microcontroller application can be implemented using an RTOS for controlling the application’s execution. The bill of materials is listed in Table 13.6

The application code interfaces with three hardware devices, the LCD, and the DS1820, and a button is used to set the time. The circuit itself is pretty easy to wire because I arranged the PORTB pins to be wired directly across to the LCD. The LCD’s pin 14, if it lines up with pin 40 of the PIC18C452, will allow mostly straight-through wiring to the LCD. The DS1820 has a 10-k Ω pullup on its line and uses one of the I/O pins as a simulated open-drain driver. The DS1820 will pull down the line when data is read back. By simulating the open drain and putting the I/O pin in “input mode” while the DS1820 is driving the line low, there is no opportunity for bus contention between the PIC microcontroller and the DS1820. The button circuit consists of a pulled-up line that is tied to ground when the button is pressed.

When I created this application, I used a 4-MHz ceramic resonator simply because it was already on the breadboard that I had wired previously for a PIC16F877 application, and the “core circuit” was identical to what I needed for the PIC18C452. The same pinout is a useful feature of the PIC18C452 and allows you to easily replace a mid-range PIC microcontroller with the PIC18x without changing the connections. If you are going to use this application as a clock, you might want to put in a 4-MHz crystal (with external capacitors) to get the most accurate timing possible.

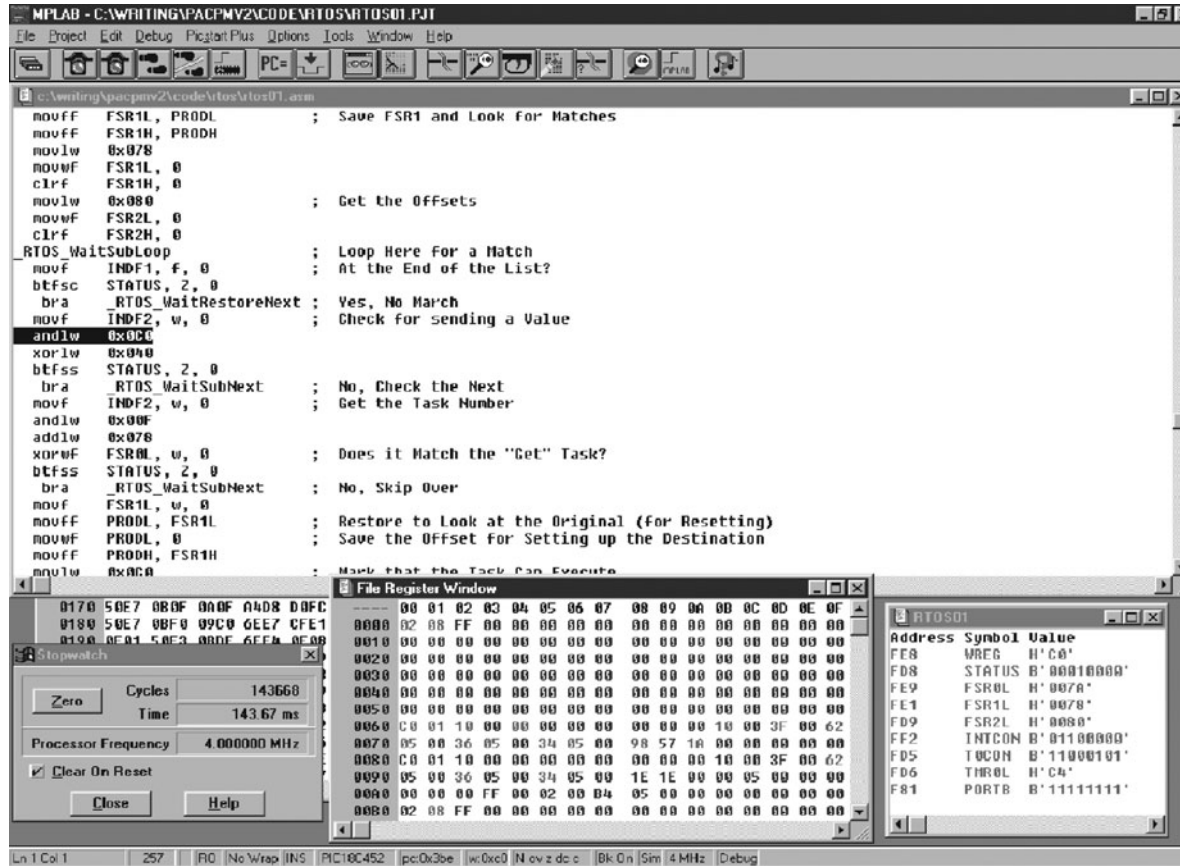


Figure 13.3 Screen shot of executing RTOS code in the MPLAB IDE simulator.

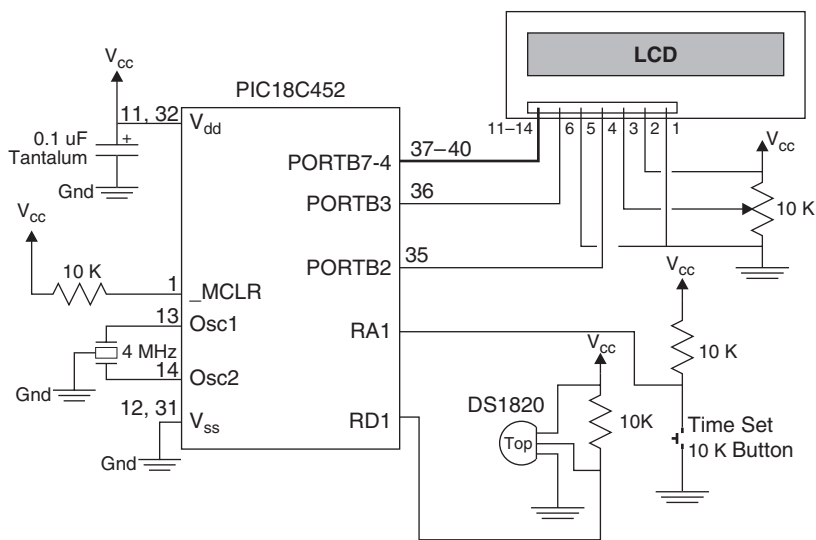


Figure 13.4 This simple circuit can be used to demonstrate the operation of a multitasking RTOS.

With the circuit designed, I then looked at how I would architect the application software. Because this code would be running under the RTOS, I wanted to make sure that I could develop it without using any more than the available resources.

I always find creating a simple block diagram of how I expect the tasks to execute to be invaluable when I am designing an RTOS application. For this application, I was able to create the task block diagram shown in Fig. 13.5 and allow only five tasks in total to be created.

TABLE 13.6 RTOS DIGITAL CLOCK/THERMOMETER APPLICATION BILL OF MATERIALS

PART	DESCRIPTION
PIC18C452	PIC18C452–10/JW
DS1820	DS1820 in TO-92 package
LCD	Hitachi 44780–based alphanumeric display
4-MHz	4-MHz ceramic resonator or 4-MHz crystal and capacitors
10-k Ω	3 $\times$ 10-k Ω , 1/4-W resistors
10-k Ω	10-k Ω single-turn potentiometer
0.1- μ F	0.1- μ F tantalum capacitor
Button	Momentary on button
Misc.	Prototyping card, wiring, +5-V power supply

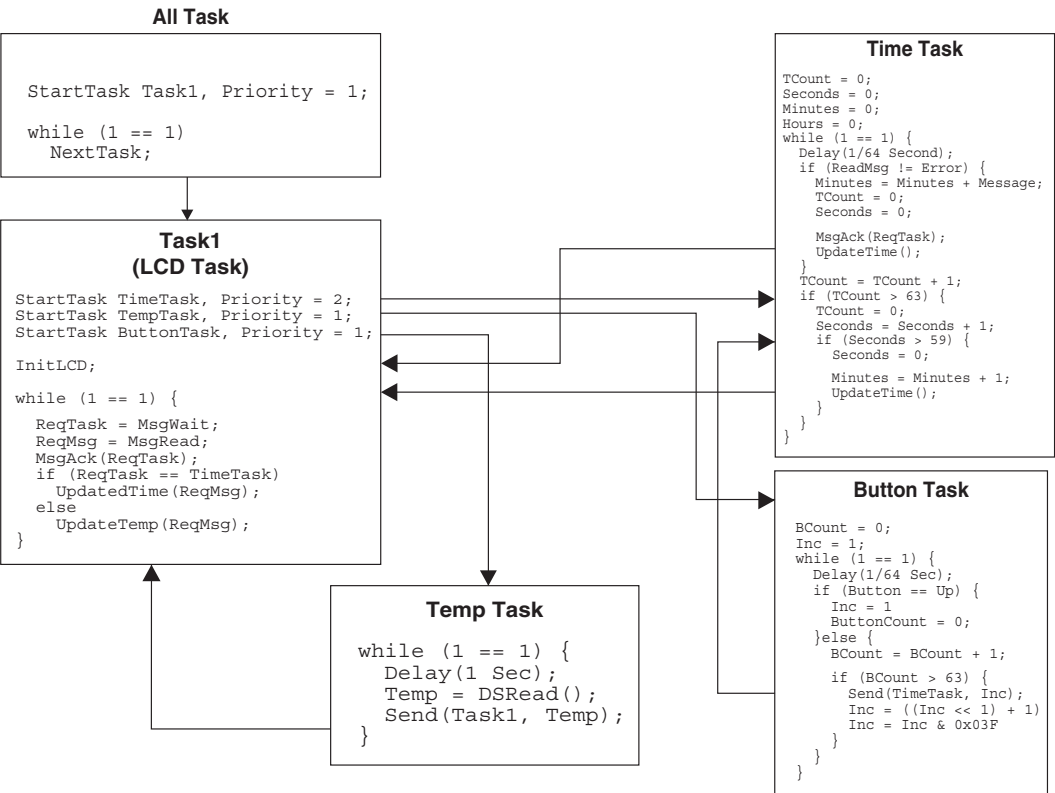


Figure 13.5 The task diagram for a simple RTOS-based digital clock and thermometer application.

The **AllTask** is built into the RTOS, and its function is to start the **Task1** task executing and provide a task that never blocks and is able to execute instructions that aren't masked for interrupts. **Task1** starts the other tasks and then initializes the LCD. Once the "temperature task" (**TempTask**) or the **TimeTask** has a message for it, it updates the LCD with the appropriate data.

Note that in **Task1**, when the LCD is set up, I use the **IntWait** request to wait on the 16.384-ms TMR0 interrupt interval to provide long delays for the LCD. For short delays, I use the **TaskNext** RTOS request, which has the operating system check for other tasks and return. As I indicated earlier, this executes in about 700 μ s, which is a more than long enough delay to allow LCD short-delay commands to execute.

The 16.384-ms TMR0 interrupt built into the RTOS is used by **TimeTask**, **ButtonTask**, and **TempTask** to initiate the requests. Each task spends most of its time doing nothing and just waiting for a specific delay. When you look through the application, you probably will be surprised to find that this 16.384-ms interval is used for the real-time clock. The advantages of using this interrupt interval is that regardless of the kinds of delays that occur owing to other tasks, each task will be interrupted every

16.384 ms on average. When you first single-step through the application, you'll find that `TimeTask` is interrupted after almost 19 ms, but the average will work out to 16.384 ms, allowing this interval to be used as a regular clock.

When you look at the `TimeTask` code, you will discover that I count the number of 16.384-ms intervals and add the delay until the total is greater than 1 second. Each time the TMR0 interrupt allows `TimeTask` to execute, the code executes as

```
TotalTime = TotalTime + 0x04000;    // Increment Second Fraction by
                                   // 16,384
if (TotalTime > 1000000) {          // One Second has Past?
    TotalTime = TotalTime - 1000000; // Yes, take 1,000,000 from the
                                   // total
    Seconds = Seconds + 1;
    if (Seconds > 59) {              // Increment the Minute if
                                   // Appropriate
        :
    }
}
```

The 0x04000 that is added to `TotalTime` each time TMR0 overflows is 16,384 decimal—which is the number of microseconds in hex. When the value is more than 1 million, 1 million is subtracted from the total, and the process repeats. Each time the process repeats, the `TotalTime` overflow after subtracting 1 million from it is added to the total. In Table 13.7, the `TotalTime` and number of TMR0 interrupts are shown for waiting for various seconds.

In the table, you can see that the number of TMR0 intervals used for each second changes, so the average time that the “second” is updated is 1 million μ s, or 1 second.

To ensure that the TMR0 interrupt is never missed by `TimeTask`, I have set it to the highest priority of any task in the PIC microcontroller. If an interrupt is missed by one of the other tasks (the temperature or button task), it is not critical. If there is an opportunity for the TMR0 interrupt to be missed by `TimeTask`, then the clock will not be accurate and will “loose” time the longer it executes.

When you read through the code, you'll see that `LCDTask` does the conversion of the time and temperature before they are displayed on the LCD. For this application, the sending task number is used to determine which message should be processed.

TABLE 13.7 DIGITAL CLOCK/THERMOMETER APPLICATION TIME UPDATE WITH TMR0 INTERVALS

TotalTime START	TotalTime END	TMR0 INTERVALS
0	1,015,808	62
15,808	1,015,232	61
15,232	1,014,656	61
14,656	1,014,080	61

One aspect of the RTOS messaging to `Task1` that I am pleased with is the ability of the RTOS `MsgWait` to return the task number of the transmitting tasks. I found this to be very useful in this application because it allowed me to respond immediately to incoming messages. If I hadn't put in this feature, I would have had to poll through each available task to find what was sending messages to it. Doing this polling could be a problem with `TimerTask` because I would like to respond to that task first so that I always will respond before the next timer "tick." To also make sure that `TimerTask` never misses an interrupt, I acknowledge the message before processing it and updating the time.

Before sending a message to the "LCD task" to notify it that it's time to update the display, the `ButtonTask` is polled to see if it is sending a message to `TimeTask`. If an increment command is sent to `TimeTask`, the `Seconds` information is deleted and the `Minutes` is updated with the latest `Increment` value. The `Increment` value is kept at one while the time set button is released and increments by powers of 2 (to a maximum of 63) for each second the button is pressed. Each second that the button is pressed, a message is sent to `TimeTask` indicating that it should increment the `Minutes` count by the `Increment` value. The "time increment" value is increased by the next power of 2 (to a maximum value of 63) or reset back to 1.

By pressing the button, the `Increment` value is increased by a power of 2. Thus, after 1 second, it will increase the minute counter variable by 3 minutes (and not 1); after 2 seconds, it will increase the minute counter variable by 7 minutes; and after 3 seconds, by 15 minutes. When an `Increment` variable value of 63 is reached, `Increment` won't be incremented any more, and every second, the hour will be updated. Using this algorithm, it's possible to run through a total of 24 hours in less than 30 seconds with quite a bit of control over the time you are trying to arrive at.

If you are a digital clock manufacturer, I hope that you remember where you first saw this. I would be happy to license this algorithm to you for a modest fee.

The temperature sensor used in this application, the Dallas Semiconductor DS1820, is a rather interesting little beast. It is available in a three-pin transistor (TO-92)-like package and only requires a 10-k Ω pull-up resistor for the driving signal. The data sent and received use a Manchester-like encoded signal in which the controlling device pulls down the line for a specific amount of time to write to the DS1820. A 0 is a low pulse of 15 to 60 μ s (with 30 μ s being typical), and a 1 is a low pulse greater than 1 and less than 15 μ s. The write operation is shown in Fig. 13.6.

To read data from the DS1820, the microcontroller pulls down the line for approximately 1 μ s and releases it, at which time the DS1820 holds the line low for a specific length of time to write a bit on the line. If the DS1820 holds the line low for less than 15 μ s, then a 1 is being sent. If the line is low for greater than 15 μ s, then a 0 is being transmitted. I found it best to poll the line 3 or 4 μ s after the PIC18 stops driving the line. Figure 13.7 shows a read of two 1s followed by two 0s to give you an idea of what data looks like on the single line.

Data is passed between the controlling device and the DS1820 8 bits at a time, with the least significant bit first. Before any command, a "reset" pulse of approximately 500 μ s is output onto the line. When the line goes high, the DS1820 responds by

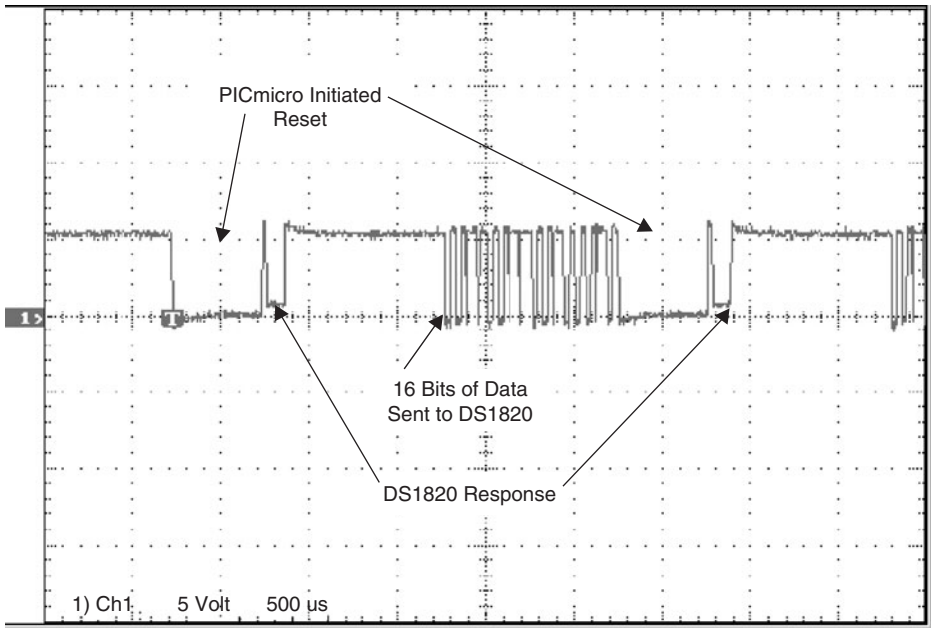


Figure 13.8 Resetting the DS1820 consists of holding down the data line for at least 500 ms.

pulling the line low for about 60 μ s after the line has gone high. After the “reset” pulse, the DS1820 should not be accessed again for 1 ms.

Figure 13.8 shows the reset pulse along with the DS1820 response for a temperature read operation. Note that the DS1820 response is at a slightly higher voltage level than the PIC microcontroller-initiated reset. This is an example of *bus contention*, which is caused by the PIC microcontroller still driving the line high and the DS1820 trying to pull it down low. In the code that follows and that on the CD-ROM, the PIC microcontroller’s I/O pin is not driving the line for the 1 ms of the reset operation to allow the DS1820 to drive the line without interference from the PIC18C452.

When I first coded this application, I created three subroutines for operating the DS1820 from the DS87C520 from this application. The first routine was `DSReset`, which pulled the line down for 480 μ s and then waited 1 ms before returning to its caller:

```
DSReset:                                ; Reset the DS1820
    bcf    DS1820                        ; Hold the DS1820 Low for 500  $\mu$ secs to reset
    movlw  125
    addlw  0x0FF                          ; Add -1 until Reset is Equal to Zero
    btfss  STATUS, Z                      ; 4 Cycles for Each Loop
    bra    $ - (2 * 2)
    bsf    DS1820
    bcf    DSTRIS
    movlw  0                              ; Wait 1 ms before sending a command
    addlw  0x0FF
```



```

btfss    STATUS, Z
bra      $ - (2 * 2)
bsf      DSTRIS
bsf      DS1820
return

```

The transmit routine, `DSSend`, was coded as a simple subroutine with no hooks used for the RTOS:

```
DSSend:                                ; Send the Byte in "WREG" to the DS1820
    movwf   FSR0L, 0
    clrf    FSR0H, 0
    movlw   8

DSSendLoop:
    bcf      INTCON, GIE                ; Make Sure Operation isn't interrupted
    bcf      DS1820                      ; Drop the DS1820's Control Line
    rrcf     FSR0L, f, 0                 ; Load Carry with Contents of the
    btfsc    STATUS, C, 0
        bsf   DS1820                     ; If "1" Sent, Restore After 4 Cycles
    bsf      FSR0H, 3, 0                 ; Loop for 24 Cycles
    decfsz   FSR0H, f, 0
        bra   $ - (2 * 1)
    bra      $ + (2 * 1)                 ; Put in a Full 30 Cycle Delay
    bsf      DS1820                      ; The Line is High
    bsf      INTCON, GIE                 ; Restore the Interrupts
    bsf      FSR0H, 3, 0                 ; Loop Another 24 Cycles for Execution Delay
    decfsz   FSR0H, f, 0
        bra   $ - (2 * 1)
    addlw    0x0FF                       ; Subtract 1 from the Count
    btfss    STATUS, Z, 0
        bra   DSSendLoop
    return                               ; Finished, Return to Caller
```

There is one thing to note in both `DSSend` and `DSRead` (the data receive routine from the DS1820), and that is that the first thing that I do is to mask interrupts. This was done to ensure that the `TMR0` interrupt does not interrupt the timed data transfers when the DS1820 line is low. If the `TMR0` interrupt occurs after the data bit is transferred, then there won't be any problems, and execution can pick up just where it left off.

Reading data from the DS1820 was accomplished by pulsing the line low and then seeing how long it would stay low:

```

DSRead:                ; Receive the Byte from the DS1820 and put
    movlw    8          ; in "WREG"
DSReadLoop:
    bcf      INTCON, GIE ; Make Sure Operation isn't interrupted
    bcf      DS1820      ; Drop the DS1820's Control Line

```

```
bsf      DSTRIS          ; Turn Port into a simulated Open Drain
                        ; Output
nop
bsf      STATUS, C, 0    ; What is Being Returned?
btfss    DS1820
    bcf   STATUS, C, 0    ; If Line is high, a "1"
rrcf     FSR0L, f, 0     ; Shift in the Data bit
bsf      INTCON, GIE     ; Can Interrupt from here
clrf     FSR0H, 0
decfsz   FSR0H, f, 0
    bra   $ - (2 * 1)
bsf      DS1820
bcf      DSTRIS
bsf      DS1820
addlw    0x0FF          ; Loop Around for another Bit
btfss    STATUS, Z, 0
    bra   DSReadLoop
movf     FSR0L, w, 0     ; Return the Byte Read in
return   ; Finished, Return to Caller
```

To read data from the DS1820, I carried out the following instruction process:

- 1** Reset the DS1820.
- 2** Send 0CCh followed by 044h to begin the temperature sense and conversion.
- 3** Wait 480 μ s for the temperature conversion to complete.
- 4** Send another reset to the DS1820.
- 5** Send 0CCh and 0BEh to read the temperature.
- 6** Wait 100 μ s before reading the first byte in the DS1820.
- 7** Read the first, or SP0, byte of the DS1820.
- 8** Wait another 100 μ s before reading the second, or SP1, byte of the DS1820.

The DS1820 has a unique serial number burned into it. This allows multiple temperature sensors (and other devices using the Dallas Semiconductor one-wire protocol) to be placed on the same pulled-up bus. To avoid first having to read the serial number out of the device and then referencing it each time a command is being sent to it, a 0CCh byte is sent first. This indicates to the DS1820 that it is to send the temperature without checking for a valid serial number being sent.

I found that for the DS1820 to work properly, I had to use the “typical” values for the data writes and not extremes in the specification. When I first started working with the DS1820, I used the minimums (15 μ s for a 0) because I wanted to keep the time the microcontroller was running with interrupts masked to a minimum. When I did this, I found that there were problems with the DS1820 not recognizing correct data. After changing the timings to the “typical” values in the datasheets, the DS1820 seemed to respond properly, but instead of a single, long pulse for each bit returned, I received a valid mix of short and longer pulses.

Using the RTOS and making sure that most operations are keyed off of the timer rather than instruction delays means that this application should be transferred easily to another microcontroller and RTOS combination. The only issue that you would find in porting this RTOS application to another microcontroller is with the DS1820 code and how it is timed in the application.

In *The TAB Electronics Microcontroller Handbook*, I implemented this application on a variety of microcontrollers, including a Motorola 68HC05 in an RTOS that I had written for it, and in *Programming and Customizing the 8051 Microcontroller*, I did the same thing with an 8051-derivative microcontroller. This application is quite a bit easier to implement with an RTOS because it avoids the need for controlling access to the LCD, which must be considered if the timer interrupt occurs while the LCD was being updated with the temperature.

This page intentionally left blank

DEBUGGING YOUR APPLICATIONS

Efficiently characterizing problems and then hypothesizing on the cause of the problem and investigating different methods of fixing the problem with implementation, testing, and documentation of the repair action are the process that I will present to you in this chapter. Fixing problems is generally referred to as *debugging*, but I personally dislike this term, and at work, I discourage its use. The term comes from the story of some technicians sent in to fix one of the massive computers of more than 50 years ago who discovered a fly in the works that was preventing the computer from working properly. The simple act of removing the fly to restore the computer to operation quickly became known as “debugging.” When hearing the term and the story, a person is left with the impression that debugging is a quick and painless operation. Unfortunately, this is rarely the case, and frequently when somebody goes for the quick fix of a problem, he or she either ends up spending a lot more time than if he or she had followed a structured process or introduces new problems (“bugs”) into the system. I prefer the term *failure analysis*. It’s interesting to see how exchanging a simple colloquial term for a “ten-dollar phrase” can change the perception of the process of fixing a problem in a computer system from being a “quick and dirty” exercise into a structured process that implies that the people working on the problem understand the characteristics of the failure, its root cause, and the most efficient method of eliminating it.

In the first edition of this book I rather facetiously suggested that to debug your application, you should follow a three-step process:

- 1** Simulate
- 2** Simulate
- 3** Simulate

To be fair, after making this declaration, I did present some information on how to develop stimulus files successfully and what to look for; but I think the information was pretty sketchy, and I think that I could have done better explaining how to find and fix application problems.

You will find that following the steps and procedures I layout for you in this chapter will allow you to fix problems faster than if you just look for the first problem and try to “nail it.” This will seem like a paradox because I will be pushing for you to “characterize” and understand exactly what is happening and what the symptoms are telling you. With this information and hypothesizing on what the defect is, you will find that you will understand exactly what is happening and why your fix should eliminate the problem.

Thus, instead of blindly going ahead and simulating the application in a three-step process, I will present to you the following failure analysis process to understand, find, correct, and release fixes to problems in your application:

- 1** Document the expected state.
- 2** Characterize the problem.
- 3** Hypothesize about what the cause of the problem is.
- 4** Propose corrective actions.
- 5** Test different fixes.
- 6** Release the corrective action.

This process is not limited to PIC[®] software or hardware. It works very well for all problems that you may encounter in your everyday life. A few years ago a coworker and I had similar problems with our 50-year-old houses—when rain hit the side of our houses, our basements would start to flood. My coworker has spent \$30,000 on the problem so far and still has problems, whereas I followed the process outlined in this chapter and spent less than \$1,500 right at the beginning to fix the problem and haven’t had a problem since.

Document the Expected State

Engineers and programmers have one thing in common: They’re both lousy at documenting their work. There are many reasons for this deficiency, including the schedule doesn’t allow for the time to be taken during the development process to document the project, the design isn’t finished and the documentation won’t be accurate for the final version so there’s no reason to bother, it’s somebody else’s responsibility, or the plain truth, it’s boring and generally unrewarding to do. This means that chances are that nobody has spelled out exactly what the system is supposed to do.

Not having a good description of what the system is supposed to do (*its expected state*) makes it virtually impossible to fix the problem. I would argue that you can’t claim to have a problem if you can’t point to a document that states how the application is supposed to work and be able to explain how the actual application deviates from its expected, documented operation. This step probably will seem redundant or even trivial; for example, you might have a situation where your application catches on fire—it will seem unnecessary to document that it should not burst into flames. However, documenting the expected operation and state of the application will provide you with the

means of plotting where the actual system execution deviates from the expected path when you start characterizing what is actually happening.

The better the description of the expected operation (or *state*) of the application, the better chance you will have of reaching to the best solution. If you have a liquid-crystal display (LCD) that isn't working correctly, you might be tempted to state that it is displaying an incorrect message. Documenting what the message should be could help to point you to a disconnected data bit wire that causes characters to be displayed as if that wire's bit value were always 0. Or it could point you to a software problem. The better the understanding you have of the expected state and having it written down to compare with your actual results, the better you will be able to characterize the problem and select the best possible fix.

Note that fully documenting the proper operation of the system is often not something that can be done immediately. You may have to go back and edit your expected state documentation because of what you have learned through the process of characterizing the problem or hypothesizing about what is causing the problem. However, I would not recommend that you start trying out fixes unless you fully understand what the expected operation of the application is, along with its expected state when the problem occurs.

For the example of the flooded basement, the expected state during a downpour is that any water near the cinder block basement walls will be drawn away by the weeping tiles and drainpipe at the base of the house. A more obvious expected state is that the basement shouldn't flood during a rainstorm, but this is less helpful and will not help you to get to the desired result—fixing of the problem. Understanding this expected state was achieved through discussions with home builders because I am not an expert in home basement construction.

Characterize the Problem

Being able to describe the problem accurately is critical to being able to fix it successfully. Even if your PIC microcontroller application just seems to be sitting there and doing nothing more than draining your batteries, there is information that you can find that will help you to figure out what the problem is. On the other end of the spectrum, you may have an application that fails after only a couple of hours of operation. In either case, recording what happened leading up to this state, as well as the system's operation at the time of failure, will give you information that you can compare against the expected state documentation you created earlier. The process of looking at a failing application and detailing all the relevant information about its operation and the environment that it is working in is known as *error characterization*.

One attitude that I feel is critical for failure analysis is being suspicious of anything that doesn't "feel right" and documenting what you are unsatisfied with. Suspicious activity could be a motor that doesn't respond as quickly to a button press at certain times or a light-emitting diode (LED) that appears to flicker. I have worked with a number of people over the years who have let problems escape out the door simply because they

didn't react to something that did not work exactly as expected, even though it was thought to be "good enough." If you have a suspicion that something isn't right, then you should go out and fix it before the problem escapes, and you end up with a reputation for shipping shoddy products.

When I have a PIC microcontroller application that isn't running properly, the first things that I check are

- 1** Power
- 2** Reset
- 3** Clocks

Fully 90 percent of the problems that I get called on to fix have issues in one of these three areas. These values can be checked quickly using a logic or oscilloscope probe. Once this is done, I carry out a scan around the PIC MCU to look at all the input and output voltage levels. It shouldn't be surprising that I record (document) what I observe on a sheet of paper for later comparison against the expected state. Digital logic levels can be checked with a logic probe but not voltage levels. When you first work with your applications, you may want to check the signal pins with both a logic probe and an oscilloscope or a voltmeter until you get a good understanding of what to expect when probing these pins and which tool works best for you.

When checking a PIC microcontroller's external crystal or ceramic resonator oscillator, make sure that you probe both pins with a logic probe or oscilloscope to make sure that the capacitance of the probe doesn't inadvertently set the application running. This will happen with the PIC MCU's `OscIn` pin but not with the `OscOut` pin because the extra capacitance of the probe will cause a current flow to the probe tip, which will change the voltage level of the input and cause the oscillator to start running. The best way to avoid this problem is to use a very low impedance FET oscilloscope probe or to check both pins in the case that the probe time affected the operation of the oscillator.

When you have done this, compare your notes with what the expected values are. This is an important point because I can't tell you how often I've thought I've caught a problem only to discover that I miscounted the pins and was looking at the wrong one, and the one that I thought had a problem was in the correct state.

Any actual discrepancies you find can be put in one of two "buckets": the ones that are caused by an external device driving the pin to the invalid state and the ones caused by the application code driving the pin incorrectly. To discover where the problem lies, an ohm meter check is used before and after isolating the input-output (I/O) pin from the circuit. This is known as *floating* the pin and can be checked by removing the PIC microcontroller and bending the pin out or breaking the connection from the pin to the circuit. Personally, I prefer breaking the connection to bending the pin; the metal used in chip pins tends to be brittle and easily broken if it is bent back and forth even once. If the circuit turns out to be okay, then you put the circuitry back together (i.e., restore lines by soldering them together), look back at the application code, and try to figure out why the pin could be in an invalid state.

When you are looking around the circuit, it is very important to not make any assumptions about what you think the problem is. I call the state of mind that you must be in

solution naïve. This is the ability to see and document exactly what is in front of you and turning off the filter that everybody has that allows them to think that they have the solution to the problem when they see something that matches their preconceived expectations. You might be thinking that more than 90 percent of the time when you see a discrepancy, you will be able to recognize it as either a symptom or a cause of the problem, but in my experience, the actual number is much closer to 25 percent. By not looking around naively, you probably will miss the shorted pins, incorrect wiring, or incorrectly typed instructions (i.e., `addwf register, w` instead of `addwf register, f`). Being solution naïve is a skill that I've seen in only one or two people who have truly mastered it over the course of my life. As humans, we tend to remember what we want to do and tend to see that first and miss what is actually in front of us.

In cases where the application starts up and then fails, the importance of keeping notes detailing what has happened is critical. For example, if you had an RS-232 interface that locked up, you would have to figure out if the problem was based on time, data values, or data volume (the number of incoming characters). Once this determination is made, then you can go ahead and look for the problem by simulating the data input and taking advantage of this parameter.

In trying to find a problem, don't be afraid of creating a small application to force the problem. If you can come up with an automated way of causing the error, then you have correctly characterized the situation and can translate this knowledge to a simulator to observe from the software's perspective exactly what is happening within the application code. The goal of characterizing the problem is to document exactly how the circuit is operating and what its inputs are so that you can reproduce these conditions (to make sure that the problem is fixed once you have made changes) and work at hypothesizing about what is actually happening.

Going back to the example of the leaking basement, both my coworker and I noticed that rainwater collected at the base of the house and did not drain off. For my part, I documented that when the basement started flooding, there was standing water on the outside wall and collecting in the basement window wells. We both called on builders and asked if this was part of the problem. The builders replied that the standing water was a symptom of the weeping tiles and drain pipe not doing the job. This description helped me to create a better document stating what the expected state actually was and describing the difference between the expected state (water falling on this wall would be carried away by the weeping tiles and the drain pipe and not stand in puddles against the wall and in the drain pipe) versus the actual state, which was the standing water.

Hypothesize and Test Your Hypothesis

Once you have the expected state documented and your problem characterized, you should be able to make a hypothesis on what causes the problem. This is the most important step because comparing the characterization data you collected against the expected information should point you toward the problem and the potential for a fix. Before going on and trying to fix the application using the hypothesis, you should try

to test it out either as a “thought experiment” or on actual hardware to see if you can reproduce the problem and have it behave exactly as on the failed board. An important part of making your hypothesis is deciding what the problem actually is.

When making your hypothesis, you should first list all the observations that seem to be pertinent to the problem at hand. One of the problems with the YAP-II programmer was that ASCII control characters were being accepted from the command prompt instead of being ignored. To build my hypothesis on what the problem was, I sifted through my notes on the problem (characterization of the problem) comparing them against the expected operation of the programmer and came up with the following observations:

- 1** When the Prompt is active, sometimes characters would be displayed on the hyperterminal screen as small boxes.
- 2** These boxes would cause an `<== Invalid` message when Enter was pressed.
- 3** I noticed that when the boxes were displayed, I tended to be pressing down on both the Shift and Ctrl keys at the same time.
- 4** When I pressed the Ctrl key with a character key, the small box would appear.

From these observations, I made the hypothesis that ASCII control characters could be produced using the Ctrl key and character keys. I further made the hypothesis that the YAP-II’s instruction-processing code was not filtering out these characters properly.

To test out the first hypothesis, I disconnected the YAP-II from the PC’s serial port and shorted together the receive and transmit lines (pins 2 and 3). I discovered that when I pressed Ctrl along with a character key, the box would be displayed on the hyperterminal window, as it did with the YAP-II. This led me to assume that ASCII control characters were being generated. I then made up a stimulus file for the application and tried passing ASCII control characters (hex values less than 0x020) to the code to see what happened. When I did this, I found that the filter I had put in place didn’t seem to be working correctly. At this point, I felt that I had identified the problem and was ready to start looking at different fixes for it.

This example probably seems trite and pretty obvious, but this is the basic point to following a structured approach to finding and eliminating problems. When you go through the process I outline here, you will find that fixing virtually all problems is very simple and “forehead slapping” obvious when you work through it. This is not to say that it is the case for all problems, but if you go through this process of characterizing and hypothesizing about the problem, you can now talk to an expert about the problem without feeling like you are potentially wasting his or her time. I know of very few people who will not spend a few minutes explaining what a problem is if the upfront work is done to characterize what is actually happening.

With my flooding basement, I accepted the hypothesis from the builder I talked to that the drainpipe at the base of the house was no longer working correctly. This is a common problem for houses over 50 years old. To test this hypothesis, I poured water against houses that were a similar age to mine and found that the water didn’t seem to drain and formed puddles against the wall of the house. I then performed the same experiment on a brand-new house and found that the water drained quickly and did not form any kind of standing water.

Propose Corrective Actions

At this point, you understand how the system is supposed to work, have characterized the problems, have hypothesized about what is the cause of the problem, and can start proposing some corrective actions. For most problems, if you have followed the process steps just described, you won't have any problem specifying an efficient corrective action that will not cause other problems in the application. You may have some problems, however, when there are multiple solutions available for you to choose from, there are no obvious solutions, or the solutions will cause other problems in your application. In this section I will suggest some points for finding the best corrective action for your problem.

I would argue that every problem has multiple solutions, and when I have helped out in high school classrooms, I suggest to the students that when they are faced with a problem, they should list three different possible corrective actions. This is often not easy to do, but when you do it, you will start thinking about what is the best approach for your problem. If you force yourself to come up with three solutions, you naturally will tend to create one that is obvious and will seem to you to be the best solution, the next solution will tend to be very simple, and the last one will be very complex and require a lot of effort. The interesting thing about this technique is that there will be cases where the best corrective action could be any one of the three proposed solutions, which means that you have a variety of choices and you are more likely to make the right one.

When proposing different corrective actions, you might want to create a table such as Table 14.1, in which I have listed three different possible solutions to the problem of my flooded basement, along with different criteria that will influence the decision regarding which is the best solution.

When you review this table, the different criteria for selecting the best solution should be pretty obvious. Note that by properly documenting the expected state of the system and being able to identify the symptoms of the problem (water pooling at the side of the house), you can simply and clearly state whether or not you expect the proposed solution to work. A table such as this also lets you note various other issues that aren't related to the problem but will influence your decision. In this case, I had to consider our two Siberian huskies that live out in our back yard—if you are familiar with huskies, you will know that they are rambunctious and destructive dogs, which means that any work being done in the backyard will require the dogs to be either tied up (if it is short-term work) or put in kennels (expensive and unpleasant for the dogs). Furthermore, I have two young daughters who play in the backyard, and the trench needed to fix the weeping tiles would be a danger to them. In this case, the optimal solution is number 2, having some eaves trough specialists come in and make sure that no water comes from the eaves (and this includes adding heaters for winter when ice forming in the eaves would cause water to spill over and pool at the base of the house), as well as making a gentle slope away from the house to redirect any water that might pool there.

My coworker decided on having a contractor dig up the side of her house and had the weeping tiles and drain pipe replaced. Her family did not have a dog in the backyard

TABLE 14.1 THREE PROPOSED SOLUTIONS TO PREVENT A FLOODED BASEMENT			
	SOLUTION 1: DIG UP SIDE OF HOUSE AND REPLACE WEEPING TILES AND DRAINPIPE	SOLUTION 2: PREVENT WATER FROM POOLING AGAINST SIDE OF HOUSE BY IMPROVING EAVES TROUGH AND BUILDING UP EARTH BESIDE HOUSE	SOLUTION 3: INSTALL SUMP PUMP IN BASEMENT
Approximate cost	At least \$10,000 plus kennel fees	\$1,000	\$1,000
Homeowner effort	Minimal	Moderate	Minimal
Elapsed time to implement repair	Long, 2+ months	Short, 1–2 days	Moderate (2–3 weeks)
Ability to test corrective action	Simple; should see water no longer pooling when hosing down side of the house	Simple; should see water no longer pooling when hosing down side of the house	Moderate; will continue to have water pooling but should not have flooding during testing
Possible damage from dogs	Minimal but trench danger to dogs/children	Moderate	Minimal
Likelihood of success	Good	Very good	Unknown

and her children did not play in the backyard, so the safety concerns that I had weren’t an issue for her.

This brings up an important point: What is obviously the best solution for you is not necessarily the best solution for other people. If you are working on a team or there are others involved in the issue, you will have to discuss the important issues relating to the problem and agree on their importance in the decision-making process.

Test Fixes

If you have followed the steps that I have outlined here, you will find that testing your fix is very straightforward—you merely have to look for the sequence of events that leads up to the invalid state, repeat them, and then evaluate the results against the expected state of the system. When you characterized the problem, you will have created a set of requirements to ensure that the sequence of events that resulted in the failure will be

recreated. You also should be able to see whether or not the corrective action affects any other aspects of the application. By following the process outlined here, you will have produced the test cases most likely to demonstrate whether or not the selected fix eliminates the problem.

I tend to avoid using the term *testing a fix* and instead prefer stating that this step is *verifying the system with the corrective action performs as expected*—the problem with this statement is that it makes a lousy section title. I also would recommend that you eschew thinking of testing fixes because it limits you to just looking at the individual problem at hand and not at whether or not the fix has resulted in other problems. Chances are that you have already created fixes for problems that have resulted in problems elsewhere in the application—this is particularly easy to do in microcontroller applications, where there are multiple activities taking place at a given time.

Remember that your test will be the steps taken to characterize the problem in the first place, as well as all the other tests that you have made to test the application. This process is often referred to as *regression testing*, and it is an invaluable tool for making sure that when you have said you have fixed the system, you can demonstrate that you actually have done so. What is important in this process is that as part of characterizing the issue, the sequence of events that was used to bring out the problem is added to the entire set of tests that you use to test the system.

When you use the term *verifying the system with the corrective action performs as expected*, this doesn't limit you to looking at just the original problem; it means that you will look at the overall operation of the system to make sure that no other issues have appeared. I'm sure that when you have tried fixing something, you have experienced other problems surfacing, which in itself is disheartening, but it gives you an opportunity to look at the situation and decide on two paths for resolving it. The first is to go back and look at your corrective action (as well as the other options that you considered) and decide if it really was the appropriate response to the original problem. The second is to look at the failure and decide whether or not it is a separate problem on its own. When you are new to programming and application development, it isn't unusual to have multiple problems that involve common hardware or functions, and when one problem is fixed, others are highlighted. By taking the perspective of the entire system and performing regression tests for each bug fix, you are continually challenging the overall quality of the system.

Going back to my house, I had a two-part test. The first part was taking a hose and verifying that water ran away from the back of the house and didn't pool there. The second part was waiting for rain and verifying that the fixed eaves and downspout directed the water away from the back of the house. The two-layer fix that I applied minimized the opportunity that water would be directed to the side of the house, and even if it were, the slope of the land would carry it away.

SIMULATE YOUR APPLICATION

Simulation of the application code should be done twice in the debugging of a problem with the application. The first time is when you characterize the problem. This process is repeated in order to confirm that the problem was fixed. The simulator will help you

to close the failure analysis circle and will provide you with a tool that will allow you to see the problem in operation as well as the fix to ensure that it addresses the original problem properly and doesn't cause any new ones on its own. As will be shown in this section, the simulator can be used to characterize the errors, help you to develop hypotheses as to the cause of the problem, along with different corrective actions, and finally help you to verify that the fix does indeed work correctly.

An aspect of simulating an application is deciding where to put breakpoints. By efficiently selecting where a breakpoint is placed, errors can be observed quickly and with a minimum of overhead. I find that correctly placing breakpoints is something of an art, with the goal of placing breakpoints in such a way that problems are identified quickly. Once the problem has been found, the same breakpoints can be used to confirm that the fix has been made before another PIC microcontroller is burned with the corrected code.

Problems that can affect the placement of breakpoints are interrupt handlers, subroutines, and macros. Subroutines and interrupts are a problem because they generally execute multiple times, and their execution can be part of the failure mechanism of the application. A good example of this is a "bit banging" UART serial receiver that polls an input line three times per bit period. In the example interrupt handler presented below, when a character is received, data is passed to the mainline without any problem, but it is always followed by the byte 0xFF, or if a byte immediately follows it, it is in error:

```
interrupt serin()                // 3x Sample USART Serial Input
{                                // Processor

    TMRO = 1/3 bit;              // Reset the Interrupt
    INTCON = 1 << TOIE;
    switch (Rx state) {
        case 0:                  // Nothing Active
            if (SerialInput == 0) // Check for Start Bit
                RXState = 1;      // Set up Byte Read
        case 1:                  // Confirm Start Bit
            RXState = 2;
            BitCount = 8;         // Read in 8 bits
            BitDelay = 3;         // Delay 3 Interrupts per bit
            break;
        case 2:                  // Check a Bit For Valid
            BitDelay = BitDelay - 1;
            if (BitDelay == 0) {   // Save the bit?
                RXData = (RXData << 1) + SerialInput;
                BitDelay = 3;
                BitCount = BitCount - 1;
                if (BitCount == 0) { // Eight Bits Read in?
                    DataAvailable = 1; // Yes
                    RXState = 0;      // Restart the polling
                }
            }
    }
}
```

```

    }
    break;
} // end switch
} // end serin

```

When you walk through the code manually, it seems to be good and should not have any problems. To understand the problem, you can create a stimulus file with incoming data and then try to look for the problem.

The first place to look is in the mainline code to see if you can repeat the problem. Chances are that you have a mainline poll of the “bit banging” RS-232 interface that looks like

```
while (1 == 1) {                                // Loop For Each Incoming Character

    while (DataAvailable == 0);                  // Wait for flag to indicate a new
byte o/p                                         byte
    DataByte = RXData;                           // Save Input Byte
    DataAvailable = 0;                            // Reset Flag

    :                                             // Process Data

} // end while forever
```

A breakpoint could be put at the

```
DataByte = RXData;
```

statement, and you could single-step through it to see the data in `RXData` to confirm the problem.

In a case like this, I would suggest that you put a breakpoint at the start of the `serin` interrupt handler and start single-stepping through the code to see what happens during the good data save. I placed the breakpoint at the “RXState = 1” statement which executes when the incoming data goes low, indicating a start bit. After moving to state 1 in the next TMR0 overflow, the interrupt handler will move on and receive the next 8 bits properly. This can be shown graphically as Fig. 14.1.

The three times sample “bit banging” receive works to place the sample as close to the middle of a bit as possible. This is done by waiting one “one-third bit interrupt period” after a low start bit is found. Unfortunately in this case, if bit 7 is a 0 (which is the case for ASCII characters), then the next three times bit sample will detect the still-low bit and start reading what is interpreted as a new data packet coming in.

There are a number of different ways to fix this depending on the applications requirements. Probably the simplest way is to add a “glitch detect” in state 1 of the `serin` interrupt handler:

```
case 1:
    if (SerialInput == 0)           // If Line is High, Assume a
        RXState = 0;               // "Glitch" and Reset State
```

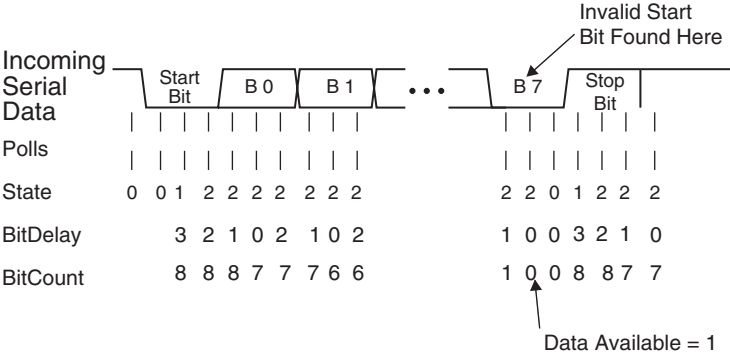


Figure 14.1 Data produced by looking at the simulated operation of the “bit banging” UART receiver.

```
else {  
    RXState = 2; // Else, Line is Low for two  
    BitCount = 8; // Polling Periods, Read in  
    BitDelay = 2; // the Character being Sent  
}  
break;
```

With this change in place, you then should simulate the application again to make sure that it works properly before you burn code into a PICmicro.

If this were real life, chances are that the preceding fix would mostly fix the problem. But there still will be characters that have a 0xFF character following them. The reason for this is that a sample right at the leading edge of the start bit would allow four samples in bit 7, each of which is low. Since the last two of these would be picked up by `serin`, the new byte read would start again and return 0xFF.

This is where your hypothesis will have to be used followed by changing the stimulus file so that the start bit fails on the first state 0 pull and the bit period is lengthened slightly (1 percent is all that is required). This should allow the two low reads at the end of the valid byte, but it will be difficult to time perfectly. To fix the code properly you should check the stop bit after bit 7; the preceding “fix” (to state 1) actually changes the application so that a problem caused by a different error mechanism would appear. This second error mechanism is also difficult to simulate.

Release Your Solution

There are no famous riddles when it comes to fixing problems, but if there were, one of the most popular ones would be, “When is a fix not a fix?” The answer is, “When nobody knows about it.”

It is critical for you to have a method of releasing your application updates so that your coworkers, teammates, teachers, and customers know about it and take advantage of the changes as soon as possible. Notification of a fix serves two purposes: The first

is to advise people who are dependent on your application that there are improvements and changes to it, and the second is to get others to try the changes to ensure that you have, in fact, fixed the problem. There are no benefits to your hard work if it is not available and communicated to others.

Notification in the twenty-first century should not be a big problem—with phone messages, faxes, e-mails, web forums, instant messaging, and so on, it is easy to get the message out to people that there are changes to the application that they can take advantage of. The problem is that despite all these communication tools, people still don't get the message. For many modern applications, the solution to this dilemma is to update systems automatically via the Internet (usually in the early hours of the day when Internet traffic is at its lightest, and it is very unlikely the application users are not affected by the change). Even with the different electronic and automatic methods of notifying people of changes to their systems as well as implementing them, you still should take the time to personally communicate with others to ensure that they know about the changes and that they know how they affect them.

Depending on the application and the circumstances, it may be difficult for you to ensure that everyone affected by the change is aware of it and can take advantage of it. Taking extra steps to ensure that the people who depend on your application are aware of the updates will avoid problems later with people who are trying to work with down-level pieces or customers who are dissatisfied with your product, even though the fixes are available to them.

Debug: An Application to Test Your Debug Skills

After working through some experiments and applications on your own, I'm sure that you will experience a few problems along the way with developing your application program incorrectly or building a circuit wrong. I'm also pretty sure that you will spend a lot of time backtracking, trying to figure out what the actual problem was. To end this chapter, I want to give you a bit of a test using the application circuit shown in Fig. 14.2

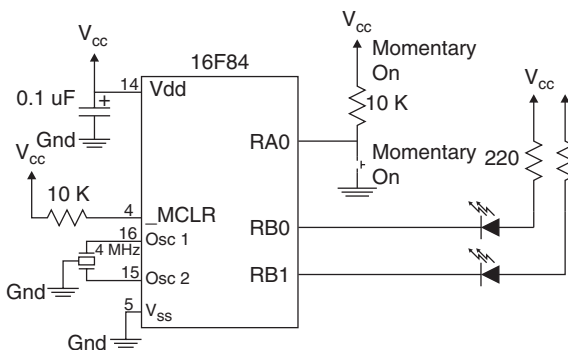


Figure 14.2 Debug circuit—turn on LED1 on power up LED2 when button is pressed.

TABLE 14.2 BILL OF MATERIALS FOR DEBUG CIRCUIT	
PART	DESCRIPTION
PIC16F84	PIC16F84-04/P
0.1-μF decoupling capacitor	0.1-μF Vdd/Vss decoupling capacitor
4-MHz ceramic resonator	4-MHz ceramic resonator with built-in capacitors
RA0 pull-up	10 kΩ, 1/4 W
RA0 switch	Momentary on
RB0, RB1 LED	Red LED
RB0, RB1 LED resistor	220 Ω, 1/4 W
Vcc	5-V power supply (4× “AA” NiMH rechargeable batteries or 3× “AA” alkaline batteries)

(with a bill of materials listed in Table 14.2) to see how efficient you are at finding problems and fixing them.

For this experiment, I would like you to build the circuit shown in Fig. 14.2 and build it using the breadboard layout shown in Fig. 14.3. Once the circuit is built, load your PIC16F84 with `debug.asm`:

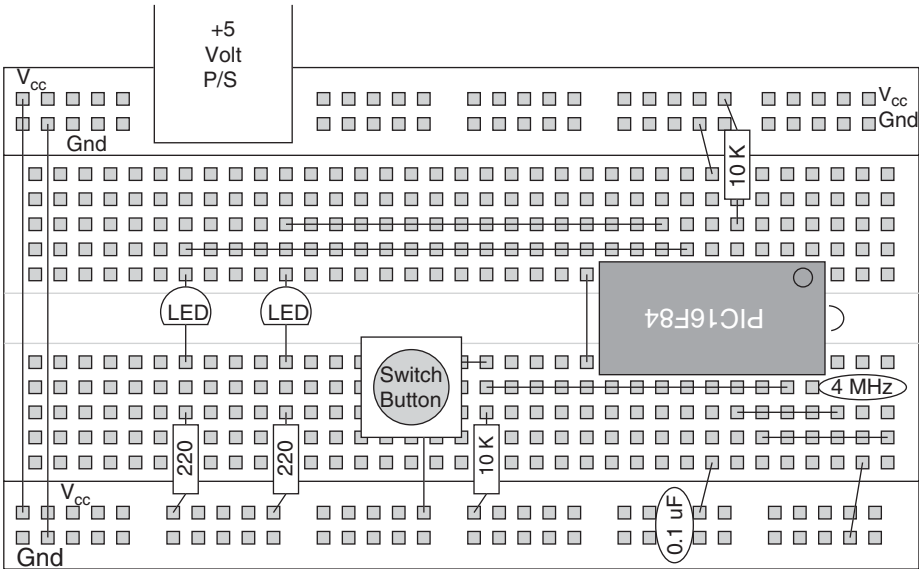


Figure 14.3 Breadboard circuit for debug circuit application.

```

title "debug - An application with a few problems"
;
; This is an application to demonstrate how insidious some
; problems
; can be. This application should be burned into a
; PIC microcontroller after
; assembly to see if the problems built into it can be
; found.
;
; The application is *supposed* to turn on a LED at RB1
; and wait for
; a button to be pressed. When it is, an LED at RB0
; should be turned on as well.
;
; Hardware Notes:
; PIC16F84 running at 4 MHz
; _MCLR is tied through a 4.7K Resistor to Vcc and PWRT
; is Enabled
; A 220 Ohm Resistor and LED is attached to
; PORTB.0/PORTB.1 and Vcc
; A 10K pull up is connected to RA0 with a Momentary on
; Switen
;
; Myke Predko
; 99.12.07
;
LIST R=DEC
ifdef __16F84
INCLUDE "p16f84.inc"
else
ifdef __16F877
INCLUDE "p16f877.inc"
endif

; Registers

ifdef __16F84
__CONFIG _CP_OFF & _WDT_ON & _XT_OSC & _PWRT_ON
else
__CONFIG _CP_OFF & _WDT_ON & _XT_OSC & _PWRT_ON &
_DEBUG_OFF & _LVP_OFF & _BODEN_OFF
endif

PAGE
; Mainline of debug

org 0

```

```

nop

movlw  0x001                ; LED at RB1 is On/RB0 is Off
movwf  PORTB

bsf     STATUS, RP0         ; Goto Bank 1 to set Port Direction
movlw  0x0FC                ; Set RB0/RB1 to Output
movwf  TRISB ^ 0x090
bcf     STATUS, RP0         ; Go back to Bank 0

Loop
btfsc  PORTA, 0             ; Wait for RA0 Button to be Pressed
goto   Loop

clrf   PORTA                ; Set RB0 = RB1 = 0 for Both LEDs
                        ; on

goto   Loop                ; Loop Forever

end
```

This application is designed to turn on a LED at RB1 and wait for the button at RA0 to be pressed (pulled down low). When the button is pressed, the LED at RB0 will be turned along with the button at RB1.

Without simulating this application, after you have built it and put in a programmed PIC16F84, I would like you to try to get the application to execute properly. If you have read through the proceeding chapters and worked through all the experiments, you should be able to get this application running without too many problems.

So that you don't cheat, I've asked that this book be laid out in such a way that you can't see my comments on finding the problems on the following pages. When you think you have found all the problems, turn the page to see what the problems are that I put in the code and how I think you should have approached finding them.

This page intentionally left blank

How did you do? You should have found two wiring errors, three definite errors with the source code, and one questionable instruction. The problems are very representative of what you will see both in your own applications and in designs you pick up from other people.

The biggest question you should be asking yourself is: How efficient was I at finding the problems? I would expect someone very experienced with PIC microcontroller to find them all within about 5 minutes. If you spent an hour or more trying to find all the problems, don't feel bad about it. Finding and eliminating problems is one of the biggest skills that you will have to learn for working not only with the PIC microcontroller but also with software that you write and applications that you build.

After building the circuit as shown in Fig. 14.2, when you apply power to the circuit, neither LED will turn on. The first thing that you should check is

- 1** Power
- 2** Reset
- 3** Clocking

If you built the circuit correctly, you will find that these three potential problems are nonissues. Power is going to the correct pins, reset is a 10-k Ω pull-up on the `_MCLR` pin, and the ceramic resonator is wired correctly. If you check the ceramic resonator with an oscilloscope or a logic probe, you will find that it is oscillating correctly at 4 MHz.

With these three problems out of the way, where do you go next? Personally, I would start with looking at the pins to which the LEDs are connected. With a logic probe, check to see if either one is in output mode [this will be indicated by the high or low LED (and audible tone) of the logic probe] after disconnecting the LEDs.

In both cases you will find that neither pin is in output mode. You have just characterized the first problem with the application because the expected I/O pins are not in output mode. What is your hypothesis?

For this problem, I would say that the `PORTB TRIS` register is not getting written to correctly. To confirm this problem, I would add the `P16F84.WAT` file and start simulating the application, as shown in Fig. 14.4.

In the figure, notice that I have single-stepped through the application to the instruction after the write to the `TRISB` register. At this point, the `TRISB` register is equal to `0b11111111`, which indicates that all the bits are in input mode and none are in output mode. This is unexpected because the value `0xFC` (`0b11111100`) has been written into the `w` register and should have been stored in `TRISB`.

To try and figure out what is happening, you should rerun the simulation and look at how the `movwf TRISB ^ 0x090` instruction executes. If the `TRISB` register is being written to, then the whole `trisb` line should turn red—but it doesn't. This indicates that the write to the register is not successful. In this case you probably will check to see that the `RP0` is in the right state (it is). Next, you should check the file register window to see if the write has gone awry. As you can see in Fig. 14.5, address `0x16` has the value `0x0FC`—which is not expected.

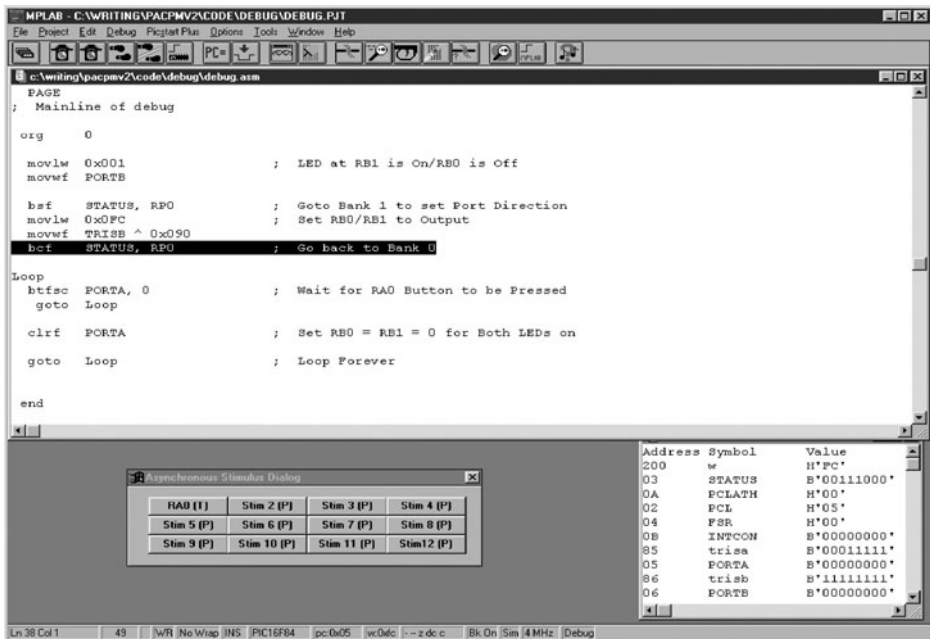


Figure 14.4 Debug circuit MPLAB IDE application desktop.

For some reason, the code is writing 0xFC to address 0x16 instead of address 0x0C of bank 1. Remembering that all the file registers of bank 0 are shadowed in bank 1, and knowing that RP0 was set correctly, you can deduce that the write actually was to address 0x96 and not 0x86, which is the address of TRISB.

Going back over the code, you'll see that the write is

```
movwf    TRISB ^ 0x090
```

and not

```
movwf    TRISB ^ 0x080
```

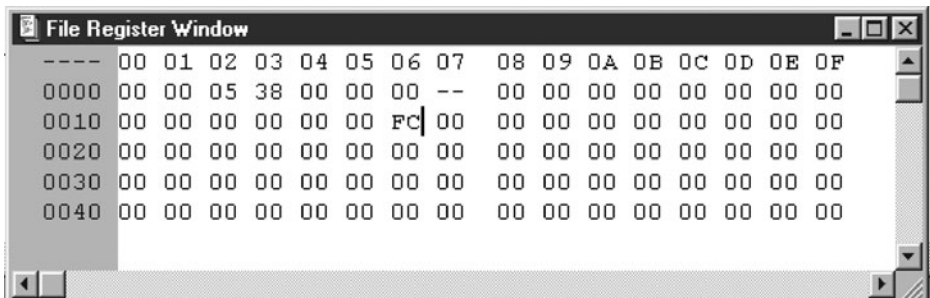


Figure 14.5 Register window contents for debug circuit application.

as is required. This typo is a very common mistake and one that you probably will make more than a few times as you work with the PIC microcontroller. Now you can hypothesize that the reason why the RB0 and RB1 bits are not going into output mode is because the address being written to is incorrect.

Now you can make the change to the source, simulate it confirm that the 0xFC value is going into TRISB, and then program the PIC16F84 with the updated code to see if this fixes the problem. If you put a logic probe on the PIC microcontroller in the circuit, you'll see that these pins are now going into output mode. However, if you connect up the LEDs, you'll find that they still don't light.

To find problems with LEDs, the first thing I always do is check to see if they *can* light. I often find that I install them backwards and don't check the application closely enough. The LEDs can be checked by disconnecting the PIC MCU connection and connecting the line to ground.

When the connection to ground is made, the LEDs should light. In this case, they won't. The obvious check (reversing the LEDs) will fix the problem. This type of error is very common with published graphics (and I hope and pray that I haven't made any in the circuit diagrams presented in this book).

Now, when you apply power, the LED on RB1 lights, but the LED on RB0 still does not light. If you disconnect the LED and check the output value on RB0, you'll see that it remains high, even after the button is pressed. After checking the switch and pull-up connections, you will have to go back to the source code and simulate some more to see if you can find the problem.

As you can see in Fig. 14.4, I have set up RA0 as an asynchronous input that toggles each time the button is pressed. As you simulate the code, you will see that bit 0 of PORTB never changes, even though you have changed RA0 from a high to a low and back again.

The problem here is another typo. Change the line

```
clrfr    PORTA
```

to

```
clrfr    PORTB
```

and see what happens. In the simulator, you will see RB0 now going low. When you program the PIC microcontroller with the new code, you will see that the application is now working as you expected . . .

Well, almost. There's still one strange problem, and that is that a few moments after the button is pressed, the LED turns off. It will turn on again after the button is pressed, but periodically it will turn back off.

At this point, most people would say that it is a reset problem. If you try to simulate the application, chances are that you won't see any problem with the code—even if you simulate for a long period of time, up to a second or so (which takes an unreasonable amount of time in MPLAB IDE to get to).

So what's going on? The problem is that the watchdog timer is enabled. In most PIC microcontrollers, leaving the WDT bit set in the configuration fuses enables the

watchdog timer. In the `debug.asm` code, I deliberately enabled it because I thought that leaving it out would be more noticeable than enabling it.

The most common problem with the watchdog timer and other configuration fuse-enabled features (such as the PIC16F87x's LVP and debug mode) is that they are enabled when the bit is set. Forgetting to put in parameter settings for all the configuration fuses can lead you to some problems where the device won't work or won't work 100 percent properly (as in this case). As I discuss elsewhere in the book, you should make sure that you know what all the configuration fuse options are for the PIC microcontroller part you are using and make sure that they are set properly.

Once the `_WDT_OFF` parameter is added to the `__CONFIG` statement of the `debug.asm` source file, the application will run properly.

There is still one thing that I am not happy with, and that how the application ends. The last instruction in the application is

```
goto    Loop
```

and if I had created this application from somebody else's code, I would be very suspicious of it. The reason for being suspicious is that in a "typical" application, after a button is pressed and acknowledged, I would think that the code would continue on or get stuck in a "hard loop" in which it does not jump to any other instructions. Jumping back to `Loop`, code that has already executed will be repeated over and over again. In this application, this instruction is not a problem, but I would feel more comfortable with

```
goto    $
```

which does not execute the polling for the button and does not clear `PORTB` repeatedly.

With the fixes I've outlined, the `debug.asm` code becomes `debugfix.com`. In `debugfix.com`, the lines with problems are commented out with the corrected lines inserted afterward:

```
title  "debugfix - An application with Fixed problems - FIXED"
;
; This is an application to demonstrate how insidious some
; problems
; can be. This application has the three source code
; problems
; fixed. The error lines are commented out with the
; correct line
; underneath them.
;
; The application will initially turn on a LED at RB1 and
; wait for
; a button to be pressed. When it is, an LED at RB0
; should be turned on as well.
```

```
;
; Hardware Notes:
; PIC16F84 running at 4 MHz
; _MCLR is tied through a 4.7K Resistor to Vcc and PWRT
; is Enabled
; A 220 Ohm Resistor and LED is attached to
; PORTB.0/PORTB.1 and Vcc
; A 10K pull up is connected to RA0 with a Momentary on
; Switch
;
; Myke Predko
; 99.12.07
;
LIST R=DEC
ifdef __16F84
    INCLUDE "p16f84.inc"
else
ifdef __16F877
    INCLUDE "p16f877.inc"
endif

; Registers

ifdef __16F84
; _CONFIG _CP_OFF & _WDT_ON & _XT_OSC & _PWRTE_ON
_CONFIG _CP_OFF & _WDT_OFF & _XT_OSC & _PWRTE_ON
else
; _CONFIG _CP_OFF & _WDT_ON & _XT_OSC & _PWRTE_ON &
_DEBUG_OFF &
; _LVP_OFF & _BODEN_OFF
_CONFIG _CP_OFF & _WDT_OFF & _XT_OSC & _PWRTE_ON &
_DEBUG_OFF & _LVP_OFF & _BODEN_OFF
endif

PAGE
; Mainline of debugfix

org      0

nop

movlw 0x001          ; LED at RB1 is On/RB0 is Off
movwf PORTB

bsf     STATUS, RP0   ; Goto Bank 1 to set Port Direction
movlw 0x0FC           ; Set RB0/RB1 to Output
; movwf TRISB ^ 0x090
movwf TRISB ^ 0x080
bcf     STATUS, RP0   ; Go back to Bank 0
```

```
Loop
    btfsc  PORTA, 0      ; Wait for RA0 Button to be Pressed
    goto  Loop

; clrf    PORTA          ; Set RB0 = RB1 = 0 for Both LEDs
on
    clrf    PORTB        ; Set RB0 = RB1 = 0 for Both LEDs
on

; goto    Loop           ; Loop Forever
goto    $                ; Loop Forever

end
```

The problems that I have outlined here are pretty common for applications you will find in magazines and books and over the Internet. They are also problems that you will make for yourself as well when you develop your own applications. Along with finding other people's problems, this process for finding and debugging problems can be applied to modifying circuits and getting them working even if you have used different parts or have changed the source code.

The purpose of this experiment has really been to show you that if you do encounter problems, you should be able to puzzle them out for yourself and fix them without getting angry at the source or giving up on the project completely.

Note that I could have thrown in a few more problems for you that I have discussed elsewhere in this chapter and this book. These problems include noninitialized variables and variables that are overlayed on hardware I/O registers. These problems are something else you will have to watch for when you are debugging an application that isn't working properly.

This page intentionally left blank

PIC MICROCONTROLLER APPLICATION DESIGN AND HARDWARE INTERFACING

As I was creating the second edition of this book, I received an e-mail with an excellent suggestion: Write a chapter on how a reader could develop his or her own applications. This is something I haven't done in any of my previous books, and it is something that I am sure would be useful for many people who are new to PIC[®] microcontroller programming and electronics. In this chapter I will introduce you to the aspects of developing PIC microcontroller applications, along with comments on what steps and roles I find to be useful for me when I'm designing my own PIC microcontroller applications and other engineering projects.

Before going through the steps that I use to develop PIC microcontroller applications, there are a few words of advice I want to give you. First of all, document everything. Get into the habit of carrying a notebook around with you so that if you get an idea on how to do something, you can record it. Human memory is a pretty fallible storage device, and I've spent many hours trying to remember that great idea I had the day before at lunch.

Second, start small; I get a lot of e-mails from people who want to create a substantial project such as the Lego MindStorms and ask me where they should start. My reply is to get a book and figure out how to program the PIC microcontroller and learn how to be effective with it before starting to plan that "killer" application. A large project for any beginner will start off strong and then become bogged down as the project seems to drag on and on. I know of some people who rise to the occasion of developing large applications and become experts through them, but these people are few and far between.

Next, don't settle on the first method you come up with. There's always more than one way to do something. Take, for example, a flashing LED; using the PIC microcontroller, the following methods could be used:

- 1** Delay by code
- 2** Delay by timer and polling the timer
- 3** TMR0 interrupt delay with interrupt handler

Which method is best in your application? Spending a few minutes thinking of options can make the application much easier later on.

It is unfortunate, but I usually discover better methods for carrying out something *after* I finish an application. If I figure out a better way, I'll usually keep it in my notebook for the next time rather than go back and recreate the application. I may go back and change the application, but that is only if the change offers a substantial improvement to what was already there (i.e., the circuit costs less to build or can use a PIC microcontroller with a smaller program memory). The basic rule here is "If it ain't broke, don't fix it."

Lastly, steal other people's ideas and methods. This does not mean stealing their code and circuit diagrams but instead understanding how their code, interfaces, and applications work and recreating them as they work best for you. Not only will this help you to avoid getting blocked, but the chances are that you also will learn from others and be able to use their ideas in ways that aren't readily apparent.

An excellent resource for this is Rudolf Van Graf's *Encyclopedia of Electronic Circuits* (more information about this set of books is available in Appendix A). This series of books provides a large selection of circuits from virtually every facet of electronics. Other resources include the various monthly electronics magazines (also listed in Appendix A) because these will have at least one PIC microcontroller application per month.

The Internet also has a plethora of circuits at various web sites. I just want to caution you not to believe everything that you read on the Internet because the information can be inaccurate. I've also found a number of web pages that will offer a few "free" circuits as an inducement for you to buy design information from the sponsor or use the sponsor as a consultant.

Requirements Definition

The most important thing that you can do before starting to design and code a PIC microcontroller application is to create a set of requirements for your application. These requirements will help you to understand what is required to do the project and allow you to check off whether or not you are meeting the original design requirements. When you first start developing PIC microcontroller applications, you should be very rigorous in defining the requirements of your application because this will allow you to keep track of what you are doing and not get overwhelmed with the task in front of you.

The first thing that I like to do is to create a simple, one-sentence objective statement specifying what the project is supposed to do. Experts in cognitive functioning will tell you that the human brain can only retain 25 words of information at one time, and your objective statements should never be longer than 25 words to avoid the need for your coworkers and customers to go back over the objective statement because they can't remember the full statement. Some examples taken from the experiments, projects, and tools in this book are

“I want to create a PC-controlled programmer that is easy to build and will work with virtually all available PCs.”

“I want to demonstrate how the operation of the PIC microcontroller’s input-output (I/O) pins can change state inadvertently when a high output is pulled low by peripherals.”

“This project will convert an NRZ serial data stream to LCD character data at 1200 and 9600 bps.”

Many of these application description statements can be found in the first paragraph of the application write-ups presented in this book. I always feel that you must be able to describe what you want to do—simply and concisely. This statement becomes a guide for you to follow when you create your other requirements definitions.

Next, the physical requirements for the application must be stated. These requirements specify the external requirements that will drive the circuit and software requirements that follow. A good list to have is

- 1** What is the desired size of the project?
- 2** How much power does the project consume?
- 3** What is the power source?
- 4** What type of circuit carrier/board is to be used?
- 5** Who will be using the project?
- 6** What are the user interfaces?
- 7** What are the safety concerns?
- 8** How much should the project cost?

For many of these questions, your answer will be “None” or “No restrictions.” Even though some answers may be vague, you are still narrowing down what you want to do.

Next, I decide on the hardware that is appropriate for the project. This list includes

- 1** What is the desired PIC microcontroller to be used?
- 2** What type of clocking will be used?
- 3** What kind of power supply?
- 4** Input devices?
- 5** Output devices?
- 6** Storage devices?
- 7** Part availability?

These questions probably seem very straightforward, except for the last one. As I write more books, I am discovering more and more that what I think is easy to find often isn’t. When I first designed the “el cheapo” programmer, I used a P-channel MOSFET that I could find easily from a variety of sources. Unfortunately, this wasn’t true for many other people. I changed the transistor to a 2N3906, a very common PNP transistor, and the complaints stopped and acceptance of the programmer increased.

Next, I prepare a list of requirements for software. These requirements include

- 1 Language application written in?
- 2 Simulator availability?
- 3 Programmer availability?
- 4 Interrupts to be used?
- 5 Built-in application interfaces to be used?
- 6 Data structures to be used?

I should point out that all the lists of requirements are interconnected. You may assume initially that a specific PIC microcontroller device is to be used only to discover that it is not appropriate for other requirements or that it may have capabilities you don't require.

You may note that I don't include an expected application size, which would help you to select the PIC microcontroller that will be used. When you are first starting out, it can be very difficult to predict what will be the final size of an application. I've been doing it for years, and I still can be off by 50 percent or more with what I guess at. Often I will specify a device with double the program memory that I think I will need.

As you develop your requirements list, you may feel that the different requirements are mutually exclusive; you can want to use a certain PIC microcontroller part number, but it doesn't support the features required in software. You may find that your suppliers cannot reliably provide you with the parts that are required. If this is the case, you may find that you have to go back and change your original defining statement. This is really not a serious problem. As you work through the requirements, if you are willing to work objectively with your requirements and the assumptions on which they're based, you will find that you can get a solid set of requirements that will lead you to a successful application.

DEVELOPING A QUALIFICATION PLAN

Part of the requirements that you define for your application should be an application *qualification plan*. This plan consists of a list of tests that the application must pass before you can consider it ready for use. In the list of qualification tests that I present in this section, some items may seem obvious, but part of the purpose of the qualification plan is to give you a checklist for when you build other units or instruct others on how to build the application.

A typical qualification plan would consist of

- 1 Test for the power supply to supply 4.75 to 5.25 V of regulated power at the required current load.
- 2 $\overline{\text{MCLR}}$ pulled to a high logic level.
- 3 The built-in oscillator runs when both OSC1 and OSC2 are probed.
- 4 User interface output functions are at the correct initial state on power-up.
- 5 User interface output functions respond correctly to user inputs.
- 6 Application outputs respond to user inputs as expected.

- 7 Application outputs respond to timer or application input events as expected.
- 8 Application test can detect all functional failures.
- 9 Reliability calculations have been performed to ensure that the application will not fail before its expected life is finished.
- 10 Environmental qualification.
- 11 FCC, ESD, CISPR, CE, BABT, and other emissions and regulatory testing.
- 12 User documentation is complete and correct.
- 13 Manufacturer documentation is complete and correct.

The first three points of this list (checking power, reset, and clocking) may seem to be obvious and unnecessary, but I would argue that they are needed to ensure that the application will work reliably. If the application isn't executing, then they should be the first things that are looked at.

The power range that I specified in the preceding list probably seems quite restricted, especially with the wide input voltage ranges available to the PIC microcontroller; but you should understand what the operating conditions are of other chips of the circuit and find out what their tolerances of input voltages are. Many active integrated circuits only work reliably within the ± 5 percent window that I have specified in the requirements list.

For many different kinds of inputs, a simple digital multimeter or logic probe will not be sufficient. Ideally, an oscilloscope or logic analyzer should be used to look at the actual I/O and confirm that it is correct. If you do not have access to these tools, then you should rigorously simulate the application to make sure that all timings are correct.

In most cases I would not consider it to be acceptable to “qualify” an application by seeing that it works when it is connected to another piece of hardware. You may get variances in the hardware that you are using that work in one application and do not work in another. For example, you may have a problem with a “bit banging” RS-232 interface depending on the tolerance of the clock that is used. If a three times sample algorithm, as I have discussed elsewhere in this book, is used, then the maximum a bit can be “out” is 30 percent at the end of the second to last bit. In 8-N-1 transmission, this is bit 7 of the application. In this case, the error after 9 bits can only be 30 percent or an error of 3.33 percent in the application clocking.

Depending on the oscillator, 3.33 percent accuracy is pretty easy to get—but what if the device you are communicating with uses the same “bit banging” serial algorithm? Depending on the hardware and the situation, one device can be out by as much as 6.67 percent, and the two devices will communicate with each other.

To avoid this type of problem, I would specify clocking accuracy that does not exceed one-third of the worst-case error. In the preceding example, this is 1.11 percent—again not very hard to get with most clocking schemes available for the PIC microcontroller, but something to be aware of.

This discussion on clocking brings up the point of *guardbanding* and what is the appropriate amount of spacing, slack, or slop you can have in your application's timing and still have it work reliably. In the preceding example, going with the rule of thumb, a 1.11 percent timing margin or a 100 percent error timing margin is allowed for a device's specified clock to be in error.

This may seem like a lot (especially when you phrase it as a margin of 1 percent out of specification), but in actual terms, it actually isn't. In the preceding example application, the margin is really only 1 percent of the total clock's accuracy. To properly qualify an application, you may have to understand the error distribution of the clock circuits to ensure that nothing is shipped that is this far out of tolerance. Another solution that would be part of the qualification plan is to ensure that a margin check is included in the application test.

Application test is something that is near and dear to my heart (mostly because I have spent almost all of my professional life ensuring that products work correctly). In the preceding list of qualification items, I noted that the test should be for all functional aspects of an application. If you've worked around electronic circuits for any length of time, you know that this is almost always impossible because there are so many different ways in which a circuit can fail.

The important point to make about test is that they ensure that the application will respond correctly to the expected inputs. There will be cases when unexpected inputs will be received (such as when the application connection is jostled or the circuit is zapped by static electricity), and these are hard to plan for. You should have a good idea of how the application works under normal operating conditions and be able to test these conditions.

It probably seems surprising that I have included a reliability calculation here—especially since you are probably planning that your first applications are going to turn on an LED based on a button's input. I will not go into reliability calculations in this book, but if you are going to develop an application for a commercial product, then you probably will want to do the calculations and even test out the application in an accelerated life test to make sure that your calculations are correct.

Reliability qualification is not something that many companies worry about, and with the ruggedness of modern components, it is not perceived as being a critical factor to a product's success. Personally, I believe that testing the reliability of a product is the most important indicator of the quality of the design and the quality of the components used. Admittedly, reliability testing can add significant costs to a product, but it can keep you from unhappy customers and lawsuits if your product fails prematurely.

Environmental qualification is simply ensuring that the product will run reliably in the location where it is expected to operate. Obviously, for most of the applications presented here, this is not a concern because the circuits will be run on a bench or in a home or office environment. Part of environmental testing is ensuring that the application does not radiate excessive noise. For products to be sold commercially, they will have to be FCC class B or BABT certified.

Many applications work in extremes of voltage, electrical "noise," vibration, temperature, or humidity. Along with this, the applications may be sealed into a package and end up being heated by the circuits used within them. If any environmental extremes are expected, then the product should be tested and qualified for these environments and the reliability calculations done to make sure that the product will operate for its required lifetime.

Many of the environments in which circuits are used will require testing to industry or governmental specifications. For example, if your application was going to be placed

in an aircraft, then you should make sure that all specifications are met for equipment operating in that environment. Many of these specifications (especially in the aerospace arena) will seem difficult and expensive to meet, but remember that in many cases these specifications are a result of other, earlier products failing—sometimes harming people.

Lastly, make sure that you have documented your application properly. I would say that this is an aspect of project development that is part of the qualification plan because documentation for users as well as manufacturers is critical to having your product integrated. User interfaces should be designed to make the product easy and intuitive to use (more on this later). Making sure that the extra hardware required by the user, as well as how to connect and power the application, is always required if anybody other than you is going to use the application or build it.

With the availability of the open document, open source software, I would recommend that you either use Microsoft's Word or the open-source equivalent. Word files are nice because their graphics can be inserted into the files and used as directions and documentations. Other options include Adobe .pdf or HTML. Of course, if worst comes to worst, you could document your product in a straight ASCII text file.

The documentation format and content should be discussed with the target audience to ensure that it is written appropriately and will be useable. Providing documentation electronically is usually preferable because it can be distributed easily and updated without having to send out multiple text copies.

All these requirements can be summarized in an ISO9000 registration. ISO9000 (the generic term) is a certification required by the European Union that is used to document development and manufacturing processes. Even if you are a one-person company, ISO9000 certification can be achieved for a modest cost and will add enforced rigor to your development process.

PIC Microcontroller Resource Allocation

Once you have a requirements plan, then you must decide how the resources in the PIC microcontroller are to be used by the application. Often you will find that the application interfaces require more resources than what the chosen PIC microcontroller can provide, or they conflict in such a way that the two resources cannot be used independently. In this case, you will have to go back and look at the application's requirements definition to see what can be changed so that you don't use more resources than what's available in the PIC microcontroller. Resource allocation is really a part of the requirements definition; no circuits have been designed and no code has been written, so there should not be any significant lost effort or time in the development process if you have to go back and change the requirements at this point.

Resources in the PIC microcontroller include

- 1** Program memory
- 2** File registers
- 3** I/O pins

- 4** Interruptions
- 5** TMR0
- 6** Prescaler
- 7** TMR1 and TMR2
- 8** USART
- 9** SPI
- 10** I2C
- 11** ADC
- 12** Data EEPROM

Looking at the list, you probably will be surprised at some of the things I consider resources. Program memory and file register usage were addressed in requirements definition, but they are resources that have to be allocated for specific uses. For example, to list the file register resources, you may want to list the arrays and variables with the purpose of trying to arrange them so that there is a minimum amount of space that cannot be used because of interaction of the variables and arrays.

Estimating actual program memory requirements is much more difficult, and probably the best allocation checking that can be done is to try to break out each block of code and estimate how large it will be. This isn't anywhere near 100 percent accurate, but it should alert you to any significant problems with not having enough space. As you work through applications, you will gain a better eye at estimating how many instructions some code is going to take up.

Timer allocation can be tricky, and you may find yourself with more timed functions than you have timers. In this case, you may have to run the timer with short common-denominator delays and processing according to the various counters required by the application. For example, to implement two routines that must be run at 50 and 60 ms, you could define TMR0 to reset once every 10 ms and process the two routines as

```
org          4
int:
    movwf    _w
    movf     STATUS, w
    movwf    _status

    bcf      INTCON, T0IF ; Reset TMR0 Interrupt Request

    movlw    10msecDelay
    movwf    TMR0          ; Reset 10 msec TMR0 delay

    decfsz   50msCounter, f ; Check for 50 msec Since
                           ; Last
    goto     60msCheck      ; Not Zero, Check 60 msec
                           ; Routine

    movlw    5
    movwf    50msCounter
```

```

:                                ; Execute 50 msec Routine
Code

60msCheck:                      ; Check 60 msec Since Last
    decfsz    60msCounter, f    ; to see if it should
                                ; execute

    goto      intend

    movlw     6
    movwf     60msCounter

:                                ; Execute 60 msec Routine
code

Int end:
    movf      _status, w
    movwf     STATUS, w
    swapf     _w, f
    swapf     _w, w

    retfie

```

The only restriction to this routine is that neither the 50- or 60-ms routine can run for more than 10 ms, and combined the total execution time cannot be more than 10 ms. Ideally, no more than 9 ms should be the longest delay to ensure that there will be no problems when the code executes.

Pins can seem like the devil in some applications; especially if built-in functions are used. My recommendations for allocating pins are to keep PORTA free for use as an analog input port or for simple digital I/Os and try to keep byte-wide functions on a PORT with no built-in I/O functions that are likely to be used.

Allocating resources is really a part of the application requirements definition process. As with the requirements definition, you may want to repeat the resource allocations planning and work through any conflicts or shortfalls. This may result in changing the application's requirements but in the long run will simplify the amount of work that has to be done to get the application working.

Effective User Interfacing

At Celestica, I used to have to work with an environmental stress screening (ESS) chamber that has one of the most difficult-to-use interfaces I have ever been exposed to. The microcontroller-driven LCD/push button interface is poor because the user is not prompted through the process of controlling or programming the ESS chamber. The interface does not give feedback with messages indicating the current operating mode and no prompting on how to jump to other operating modes. One of the biggest challenges is to actually figure out how to start the chamber operating (either in manual or automatic mode). Nobody approaches any of the chambers that have these controllers

without an inch-thick manual. This is a real mystery to me because the interface has a 16-bit ASCII LCD display and six buttons that could be used for feedback and control functions. Creating a user interface that is going to be liked by the application users generally isn't that hard to do or expensive (especially when compared with the costs of thick paper manuals, web sites or returned sales).

By following a few simple rules, you can come up with a user interface for your application that will make operation of the application intuitive and easy to use effectively by somebody with very little cost.

The first thing that you should do is to understand what kind of feedback is appropriate for the application. For example, an oxygen sensor in an automobile does not require a manned user interface, but an RS-232 interface may be appropriate for service-center maintenance. A burglar alarm circuit probably should have a light and a siren. A programmable microwave oven should have a keypad and LCD character display for setting the oven power level cooking time.

When you are deciding the best way to interface to the user, remember that the user probably won't have a manual with him or her. If an operating mode has to be selected or data entered, the user will have to be guided through the process. This could be done in a manual, but manuals get lost. Keeping the operating instructions in the circuit will keep the difficulty of using the application to a minimum.

In many order circuits, you will see that LEDs are used for this function. There are three problems with this method. The first is that extra money is required to have the panel(s) laid out and manufactured. This means that you have extra stock, and if the functions change, you will have to change the panel, requiring that the design and layout costs have to be paid repeatedly. The second concern is that the panel above is in English only. If other languages have to be supported, then additional panels have to be designed and procured. This can be a headache when the products are configured and shipped.

By far the biggest concern with using LEDs in a panel like that above is that the user will have trouble figuring out how to get to another mode or what to do next in the current mode. Unless a printed manual is included with the application, you will find users who have trouble figuring out how to work the application. Along with these problems, there are also the issues of labeling buttons and making sure that the labels are appropriate.

A much better method is to use a LCD or other alphanumeric display not only to guide the user but also to give feedback on the device's operation. In the various projects presented in this book, I have a couple of examples using LCDs for data display and user interfacing. With an alphanumeric display, the software should support a menuing system with prompts. When I wrote *PC Ph.D.*, I created a simple programmable ISA PIC microcontroller interface. This interface used a two-line LCD and two buttons to select whether or not a device was active and what address it works at (see Fig. 15.1). One button was used for selecting the parameter value, and the other was used for selecting the next menu.

Solutions such as using a LCD probably seem expensive, but they should be compared with the cost of writing and printing manuals and designing and ordering custom panels.

When the PIC microcontroller is connected to another device (or any two devices are wired together), there should be some kind of constant check (or "ping") to ensure that the link is active. If you look ahead, you will see that I go to considerable lengths to



Figure 15.1 LCD user interface showing address and register data information as part of the the user-interface console.

“ping” the programmer hardware and to confirm that whether or not the connection is active and displayed as such to the user. It may be a good idea to flag a broken link with a flashing indicator to bring it to the user’s attention.

Data transfer also must be done in such a way that the user can see that the link is operational and working. Again, looking at the programmer interfaces provided in this book, you can see that I put in gauges that show the progress of the programming operation.

If you have the ability to store execution parameters (i.e., dataports, file names, etc.), you might want to take advantage of this to allow the user to resume exactly where he or she left off. This feature makes the application easier to use and avoids reconfiguring between operations.

For input, obviously all inputs must be “debounced” either by hardware or within software. While the preceding discussion about menus may lead you to the conclusion that the best way to do an interface is with a keypad, you should take a look through this book to see the different interfaces that were created with just a few buttons (and the occasional potentiometer).

Project Management

Looking over this chapter so far, I realize that I have not done a lot to explain formal project management. Instead, I have just looked at the issues surrounding a hobbyist or new PIC microcontroller application designer encounter. While there are many excellent

books that discuss project management techniques and issues, I wanted to introduce you to some of the concepts and issues that you will face when developing a PIC microcontroller application and some of the tools that I have found to be helpful.

I think that I have done a reasonable job in this chapter of discussing how to develop a set of requirements for your applications. To summarize, you should have a plan for the application that includes

- 1** Tools to be used to develop the application
- 2** Interfaces
- 3** PIC microcontroller resource specification
- 4** Parts required
- 5** User interface
- 6** Application qualification plan

What is left is to estimate the cost of the work, the schedule of the work, and the creation and management of a team of people to do the work. If you are going to build the application for a company, then this information will be required before anybody will give you the money and people needed to do the project.

Costing a project is never simple. Along with planning on the costs of parts that will be used, you also have to take into account the costs of

- 1** Buying development tools and PCs and workstations to run them on
- 2** Printed circuit card (PCB) NRE costs
- 3** Shipping costs with customs and duty
- 4** Team members costs

Each of these items will require some research and an understanding of how the costs work for different sources. For example, if you were going to buy an emulator, would it be best to buy it retail or through Microchip? Which emulator is best for this project, and why? Can you lease the equipment for the length of time the project requires? The answers to these questions mean that the emulator costs for a project can be anywhere from a \$100 to \$10,000.

Also in these decisions are people costs. For example, are people available within your company to work on the project? Can you hire temporary employees or use a consultant? Again, the range of options can result in two orders of magnitude differences in “out of pocket” costs (just as in the emulator example above). When deciding on the people costs, you will have to first calculate the amount of time and effort required to develop the product and create a schedule of the important milestones. Calculating amount of time to develop a project is best done by first breaking the project down into every possible individual work item. When you have a list of work items, you can begin to list the time and effort required for each one. Doing the scheduling this way will help you to defend the final number to your management and enable you to create a schedule to monitor how the application’s development is progressing.

When all the work items are listed, you then should be able to produce a Gandt chart like the one in Fig. 15.2 or using Microsoft Project. This tool allows you to list all

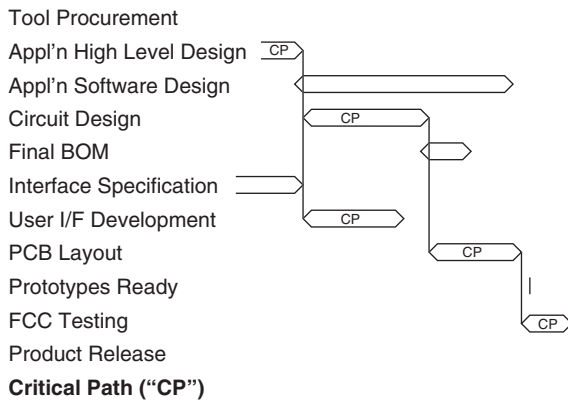


Figure 15.2 Simple Gantt project-tracking chart.

the work items for a project and draw in the *critical path*, as I have done in the figure. I have labeled “tool procurement,” “application circuit design,” “PCB layout,” and “FCC testing” as the critical-path (CP) items. These are the work items that must be completed on schedule or the project will not meet its product release date.

When you look at the figure, a few things probably will stick out at you, such as the scheduling of the application software development into the FCC testing time. When testing operations are being done, there is often the need to go back and fix some aspect of the application. The project may end up being behind schedule, and making up time will be critical. By extending non-critical-path work items in the schedule, contingency time and resources are available to prevent schedule slippage or movements to the right.

The *bill of material* (BOM) is the list of parts that are required for building a product. This list also contains packaging and extra materials (such as manuals and cables) and diskettes or CD-ROMs with which that the product is shipped. In the projects and experiments listed in this book, I have created simple bills of material for you to make sure that you have the parts on hand before starting the applications.

In any project, you should build in a contingency of money, people, and time for unforeseen problems. I like to have a minimum of 25 percent for each of these three parameters. You can pride yourself on creating accurate schedules and costs, but unforeseen events can cause problems with making your final date at cost. Some problems I have had to endure include an earthquake at a supplier’s factory, a strike at the contract manufacturer building the prototypes, and the government slapping a duty on one of the parts I wanted to use. Along with these different problems, I’ve had to overcome a lot of others. Always remember that contingency funds can be returned or used to move a release date up.

The hardest aspect of managing a project is to manage the people working on it. Ideally, you would like to have friends working with you, but this is rarely possible, and what you end up with is people you’ve never worked with before and aren’t sure you are going to like. I’ve been involved in many team projects, and here are some things I have found to help make them work. Having a team that doesn’t work well together will be the death of a project (and I have been there more times than I care to admit).

There should be one leader with a very clear vision of what the application should be and how it should work. Decisions regarding allowing deviations from the original product concept should be made only by the team leader. Deviations should be made to ensure that the original product vision is not lost. The team leader also should be the only person responsible for presenting status information to management with his or her interpretation of progress or problems.

The team should be made up of individuals with complementary and reinforcing strengths. It is not always possible to select the best people for the different work items, but the leader should be able to distribute the work to the team members in such a way that individuals' strengths are capitalized on.

It is the responsibility of each team member to keep the others apprised of their status and offer suggestions when other members are having problems. Some of the best solutions to hardware problems that I have seen have come from software developers, and vice versa. The best way to monitor progress is with a daily meeting. This meeting doesn't have to be very long, but recognizing everyone's contributions and ideas is important to make everyone feel like they are contributing toward the final goal.

Little things such as buying T-shirts for each member to wear while working on the project or having somebody bring in donuts and coffee to the daily meetings may sound corny, but they do help to foster a team attitude.

Care must be taken to avoid creating what is known as a *poisoned work environment*. The obvious example is calling another person a racist name or telling obscene jokes in front of people who could be offended. Not only will this hurt the team's overall spirit, but many such actions are considered harassment and illegal in many jurisdictions.

Other examples of subtle behaviors that will lead to problems are singling one person out to do stereotypical chores such as getting coffee or office supplies or taking notes. These activities must be shared among the group, and the leader should make sure that any tasks that haven't been assigned to a specific individual are shared by the group at large.

Awards are excellent tools to help motivate team members. They also can hurt morale if they are given on the basis of friendship or for other reasons that are not based on merit and contribution. Make sure that somebody isn't going to feel hard done by if someone else receives an award.

People will have personal problems and tragedies. The leader should be aware of any issues and make sure that other team members are aware that there is a problem (not necessarily what it is) and ensure that latitude is given. Often, this type of action will result in a stronger team because the message that is being given out is that the members of the team can rely on each other.

There are a lot of issues regarding being a part of a team, but they can be summarized in four points:

- 1** Be sensitive to others, and do not engage in or allow upsetting or offensive behaviors.
- 2** Keep everyone apprised to your status and any problems that you are having.
- 3** Work toward the team's goals and objectives.
- 4** The leader is the only person who can make decisions that will affect the outcome of the project.

Power Management

In the winter of 2006, I joined Logitech to become the manager responsible for electrical engineering/firmware development for the Harmony remote control division. Previously, I had worked on “big iron” servers and workstations in which the power consumption of a few chips was insignificant, especially when compared with the system processors that consumed 100 W of power or more. The Harmony remote controls operated at the other end of the spectrum; in order to get the best possible battery life, I had to calculate and measure every microamp of current when the unit was sending infrared (IR) or radiofrequency (RF) control codes and when it was in standby mode; this focus on detail resulted in products that lasted for a month or so on a single battery charge and was a far cry from my server experience, where things would come to a crashing halt if there was any interruption or sag in the power coming into the system.

In my tenure at Logitech’s Harmony remote controls, I really learned the value of understanding component current and power consumption and how to use this information to evaluate components or usage models to ensure that the product worked. In the following sections I would like to impart some of this knowledge to you and help you to think about how you can maximize the life of your designs.

DECOUPLING CAPACITORS

The three basic operating requirements for the PIC microcontroller are power, reset, and clocking. The PIC microcontroller is extremely tolerant of wide variances in power input and is quite tolerant of noise or sags in the supplied power, but this does not mean that you cannot avoid adding decoupling capacitors to the PIC microcontroller power input pins. These capacitors are used to filter the voltage both within and without the microcontroller, and during circuit transitions from one state to another, the internal voltages inside the chip will fluctuate, which could cause the chip to lock up, reset, or behave unpredictably.

Connecting a PIC microcontroller is very simple and only requires a 0.01- to 0.1- μF decoupling capacitor across the Vdd and Vss pins (see Fig. 15.3). The capacitor is shown as being polarized because I normally use tantalum capacitors for decoupling, although ceramic disk or polyester capacitors can be used for this purpose. Electrolytic capacitors should not be used for this purpose because they cannot react quickly enough to quickly fluctuating voltage levels.

Tantalum capacitors are the best type of capacitor for decoupling, but there are a few things to watch for. The first is the polarity of the capacitor; tantalum capacitors inserted backwards, like electrolytic capacitors, can catch fire or explode. This is not an issue with ceramic disk or polyester caps.

The second thing to watch for is to make sure that you derate the specified voltage of the part to 40 percent or less for your application. *Derating* capacitors means that instead of using them at their rated voltage, you choose capacitors that are rated at three times or more the voltage in the application in which you plan to use them. For my applications,

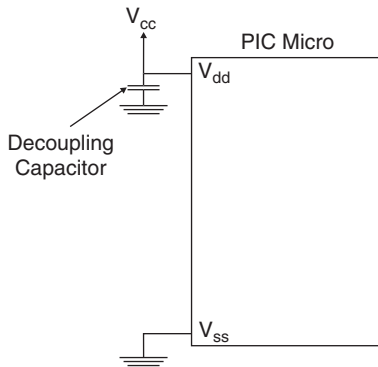


Figure 15.3 A 0.01- to 0.1- μF capacitor across the V_{dd} and V_{ss} (and very close to the V_{dd} pin) will provide noise and current transition immunity to the PIC microcontroller.

which use tantalum decoupling capacitors at 5 V, I use parts that are rated at 16 V (which is a derated value less than one-third of the rated value) to ensure proper and safe operation.

When tantalum capacitors are first powered up in an application, voltage spikes (which are caused by power supply startup and current transients within chips) can cause them to develop pinhole breakthroughs in the dielectric, which can cause the plates to touch. When the plates touch, heat is generated by the short circuit, which boils away more dielectric. This process snowballs until the part explodes or catches fire. By derating the capacitor values, the dielectric layer is thicker, minimizing the opportunity for pinholes to develop.

ESTIMATING APPLICATION POWER REQUIREMENTS

Accurate power estimating for your applications is important because the expected power consumption affects how much heat is produced by the circuit and what kind of power supply it requires. With a bit of practice and study of datasheets, you usually can estimate the power consumption of the PIC microcontroller and application circuitry very accurately, but for most applications only an order of magnitude estimation is needed to specify and select a power supply that won't overheat or be terribly inefficient, and you also should be able to specify batteries that will not run out before the specified period of time has passed.

For the PIC microcontroller itself, the *intrinsic current*, what I call the current consumed by the microcontroller with nothing connected to it, is available from Microchip in the datasheets. For the PIC16F87x, rated at 4 MHz, you will find a table in the datasheet that lists the I_{DD} (the *supply current* or *intrinsic current*) according to the

**TABLE 15.1 PIC MICROCONTROLLER
OSCILLATOR CURRENT CONSUMPTION
COMPARISON**

OSCILLATOR	IDD	FREQUENCY
LP	52.5 μ A	32 kHz
RC	5 mA	4 MHz
XT	5 mA	4 MHz
HS	13.5 mA	4 MHz

oscillator type. Table 15.1 shows that different clock drivers will produce different current draws, and this is due to how the circuitry is configured for the different clocking modes.

Next, select the current requirements for the devices that connect directly to the PIC microcontroller. Depending on their operation, the current requirements can change drastically. For example, an LED that is off consumes just about no current, whereas one that is on can consume from 5 to 10 mA. As you are starting out, use the worst-case values for your current consumption estimates.

Also note that different devices will respond differently depending on how they are used. A very good example of this is an LCD display. Many modern LCDs have built-in pull-ups to make interfacing easier for electronic devices that have open collector outputs (such as the 8051). Typically, the current requirements of these devices will be quoted with the minimum value rather than the maximum. To ensure that you have an accurate estimate, you will have to check the current drain with the LCD connected and operating with the PIC microcontroller.

Lastly, the power consumption of other devices connected to the circuit (but not the PIC microcontroller) will have to be determined through the device's datasheets. Again, the worst-case situation should be taken into account.

Once these three current values have been found, they can be summed together to get the total application power and then multiplied by the voltage applied to get the power consumed by the application. Once I have this value, I normally multiply it by a 25 to 50 percent derater to make sure that I have the absolute worst case.

In the applications in this book where I have specified the current, I have continually sought out the worst case and then derated the power to make it seem even worse. The reason for this is to ensure that you will not have any problems with your application power supply. Power really can make or break an application, and incorrectly specifying a supply can lead to problems with the application not powering up properly, failing intermittently, or not running as long on batteries as expected.

Marginal power supply problems can be an absolute bear to find as well. By going with a derated worst case for my application power requirements, I have eliminated one possible point in the application from going bad.

POWER-DOWN MODES

As I worked through the designs of Harmony remotes, one of the things that I learned is that the secret of long battery life is not keeping the active current low but working at keeping the standby current at an absolute minimum and then only running the circuitry at full power at as short a time as possible. Much of this work has been made easier by Microchip and the introduction of nanowatt devices (which dissipate as little as 100 nW of power when they are in standby mode). Having low standby current is not difficult, but it does require you to provide a great deal of attention to detail in making sure that you have checked over the operation of the components and how they are placed in very low current draw modes while still being able to work quickly.

Standby is the term I use to describe the situation where the PIC microcontroller is in “sleep” mode, and (if it is a nanowatt device) it is consuming 100 nW of power when it is not driving any other devices. There are a couple of subtleties here. The first is that it is easy to keep an LED powered while the PIC microcontroller is at sleep. I should have noted that the PIC microcontroller dissipates 100 nW of power (which is 30 nA when powered in a 3.3-V system) plus what else it is powering—an LED will require at least 5 mA to drive it, which means that in sleep mode a nanowatt device will be consuming roughly 5.00003 mA of current, not the 0.00003 mA of current that would be expected by the nanowatt device.

The solution to this problem is to stop driving any LEDs or other devices via the PIC microcontroller (which is often easier to say than do). Driving LEDs is an obvious area where current is being drawn, but when other components are involved, the current draw is much harder to detect, and it is more difficult to understand how to resolve the issue. To address the situation properly, you will have to have a clear understanding of how the circuitry is wired and how the different chips work and the expectations they have for going into a low-current standby mode.

For example, you might think that simply putting all the I/O pins into input mode would solve the problem of current flow between the PIC microcontroller and external devices. This is simply not the case; placing all the pins in input mode could result in lower than switching threshold voltages on the peripheral component, causing them to remain active. If there are pull-ups external to the PIC microcontroller, you may find that there are current flows from these pull-ups through the clamping diodes on the I/O pin circuitry. Finally, there may be definite power-down sequences that the external chips require, and simply putting the I/O pins in input mode will not be sufficient. To achieve a very low power standby state, you must spend some time reading and clearly planning how you expect the various chips to behave.

It should almost go without saying, but when you decide on the components for your application, you should make sure that you select parts that are designed for low-power operation. This clearly rules out any bipolar (i.e., 74xxx logic) components and analog chips that do not have specific low-power modes. When selecting parts, not only must you understand how they go into a low-power mode, but you also have to make sure that you understand how to bring these components out of low-power mode. Some chips will retain their state information, whereas others will require reloading and reinitialization. When choosing parts, you need to go beyond the first page of the datasheet

TABLE 15.2 SAMPLE HANDHELD GAME COMPONENT CURRENT DRAWS

COMPONENT	ACTION CURRENT DRAW	ACTIVE CURRENT DRAW	SOUND-OFF ACTIVE CURRENT DRAW	FULL STANDBY CURRENT	COMMENTS
PIC microcontroller	4 mA	4 mA	30 nA	30 nA	Nanowatt part
LCD	60 mA	60 mA	60 mA	60 μ A	
Sound chip	100 mA	30 mA	30 μ A	30 μ A	Action current draw is when a sound is being made.
dc/dc Converter	32.82 mA	18.82 mA	32.012 mA	38 μ A	Current draw is sum of others $\times$ 20 percent to reflect current loss + 20 μ A quiescent current.
Total current draws	196.82 mA	112.82 mA	92.012 mA	128.03 μ A	
Life expectancy	14.2 hours	24.8 hours	30.4 hours	21,870 hours	

and fully understand how the part works in terms of current consumption when it is active and in standby modes.

When you do truly understand how the components work, you will be amazed at the life and operation you can achieve. To show you what I mean, consider a small handheld game that is controlled by a PIC microcontroller, is powered by a single “AA” battery (which has a 2,800-mAh capacity rating), has an LCD, a sound chip, and a dc/dc converter (80 percent efficient) to convert the battery’s 1.5 V to 3.3 V for the other chips in the system. Table 15.2 lists the chips along with some sample current requirements with what would be the expected life expectancy on a single “AA” battery.

In this table you can see that the game will have an operating life expectancy of anywhere from 14.2 hours to almost 2½ years—clearly putting the components into standby mode as much as possible will be advantageous for the user. The issue now is to understand the customer’s usage model to see if you can provide customers with a product with the maximum life possible from the single “AA” battery. These calculations are best done on a spreadsheet, where you can modify your assumptions.

As I started working with my spreadsheet, I added the “Sound-Off Active Current Draw” column to Table 15.2 to see what could be gained by putting the PIC microcontroller and the source chip to sleep except when absolutely required. Table 15.3 lists some different usage models and the life expectancies that you get from the product. Note that to maximize the life of the product, I have assumed that the sound-off mode

TABLE 15.3 SAMPLE LIFE EXPECTANCIES FOR DIFFERENT USAGE MODELS FOR THE HANDHELD GAME						
OPERATING MODE	ACTION %	ACTIVE %	SOUND -OFF %	FULL STANDBY %	AVERAGE CURRENT DRAW	EXPECTED BATTERY LIFE
Hardcore gamer (6 hours/day)	1%	1%	23%	75%	24.80 mA	113 hours (4.7 days)
Moderate gamer (1.5 hours/day)	0.25%	0.25%	5%	94%	5.50 mA	509 hours (21 days)
Casual gamer (½ hour/day)	0.08%	0.08%	0.84%	99%	1.15 mA	2,435 hours (101 days)

will be a significant part of the gaming experience, with the game shutting down after 2 minutes of inactivity.

When I chose the values for this table, I used some components with which I am familiar (so the current draws are actually very representative of an actual system). Going through the analysis, to get a year’s life on the battery, the game would have to be active for about 15 minutes per day, which is less than I projected for the casual gamer, and before the product could go ahead, the usage model of 15 minutes would have to be vetted by company management and the decision made to modify the customer usage models, add more batteries (or batteries with more capacity or higher output voltage, eliminating the need for the dc/dc converter), or simply sell the product as is and accept that customers will be replacing batteries more frequently than the product.

The important thing to get from this analysis is that huge gains can be achieved through the aggressive use of power-down modes; for this simple handheld game, the life of the product went from a half day to over 100 days by understanding what is possible with putting the product in standby along with stating expected customer usage models.

Reset

Reset can be implemented in the PIC microcontroller simply with many new parts, eliminating the need for a separate circuit or having a built-in brown-out reset sensor. Even putting your own reset circuit into an application is simple, with only a couple of rules that must be followed.

Adding external reset circuit to the PIC microcontroller consists of a pull-up connected to the `_MCLR` pin of the PIC microcontroller. As shown in Fig. 15.4, a switch pulling `_MCLR` to ground potential can be implemented with a momentary-on switch. A resistor of 1 to 10 kΩ is probably most appropriate; the input is CMOS and does not draw any current through the resistor. The resistor is used primarily as a current-limiting device for the momentary-on switch.

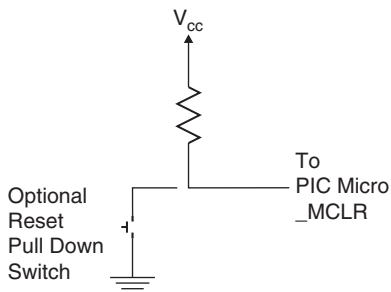


Figure 15.4 A simple pull-up is usually all that is required to hold the `_MCLR` or reset line high, allowing the PIC microcontroller to execute.

In the configuration registers of mid-range parts, there is a bit known as `PWRTE`. This bit will insert a 72-ms delay during PIC microcontroller power-up before the first instruction is fetched and executed. The purpose of this function is to allow the PIC microcontroller's clock to stabilize before the application starts. In low- and high-end PIC microcontrollers, this function is not always available. When the `_MCLR` pin is made active (pulled low), the oscillator stops until the pin is brought back high. In addition, the oscillator is also stopped during “sleep” to minimize power consumption. The `PWRTE` 72 ms is required in these cases as well to ensure that the clock is stable before the application's code starts executing. `PWRTE` does not have to be enabled if a stable clock is being input to the PIC microcontroller such as in the case where a canned oscillator is used as the PIC microcontroller's clock source instead of a crystal, ceramic resonator, or RC network. Personally, I always use the `PWRTE` function; the 72 ms really isn't detectable by a user, and it ensures that there always will be a clean power-up.

Some PIC microcontrollers that are run at low voltage or using alkaline batteries can be reset by the brown-out reset (BOR) circuitry; in some chips, the brown-out voltage is when V_{DD} is less than 4.0 V, and in others, this voltage is programmable. The programmable BOR voltage tends to be available in newer PIC microcontroller part numbers, which are able to run on from 2.0 up to 6.0 V. I rarely use the 4.0-V BOR parts and instead opt for the ones with a programmable voltage range. Once power drops below the BOR voltage, the circuit will become active and will hold the PIC microcontroller reset, even though it is receiving a valid voltage for the application.

INTERNAL RESET CAPABILITIES

If the PIC microcontroller you want to use does not have a built-in BOR function and you require it for your application, you can create one using a zener diode and a comparator, as shown in Fig. 15.5. In this circuit, voltage is reduced by a voltage divider to one-third its input. The 1-V source will be compared with the one-third V_{CC} input. If V_{CC} goes below 3 V, this circuit will put the PIC microcontroller into reset. The voltage

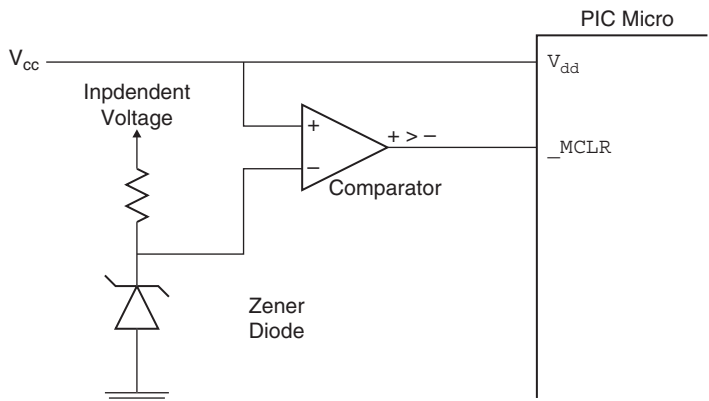


Figure 15.5 A rudimentary brown-out reset (BOR) circuit can be implemented with a zener diode and a comparator.

divider values can be changed for different ratios, and R can be quite high (100 k Ω +) to minimize current drain in a battery-driven application.

Many of the more modern PIC microcontroller part numbers have internal power-up reset circuitry, which frees up the MCLR pin for use as an input. A common use for this pin is RS-232 input using a resistor as a current limiter and providing “bit banging” software to read the incoming values. If you use the $\text{MCLR}/\text{I/O}$ pin in this fashion, make sure that you “clamp” the pin to ground, as shown in Fig. 15.6. If the incoming negative voltage is not clamped within the PIC microcontroller, the negative voltage could cause the PIC microcontroller to be forced into reset mode. The general-purpose pins are designed with clamping diodes built in and will not allow inputs to be driven outside V_{dd} or ground. Not clamping the input pins can cause some confusing problems when you first work with the PIC microcontroller in this type of application.

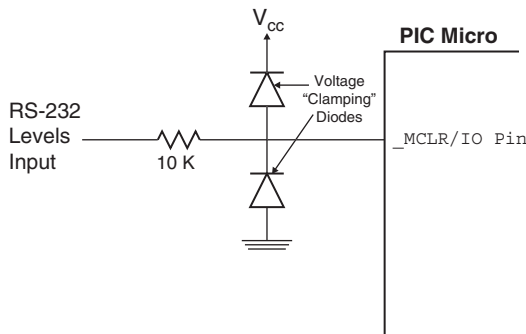


Figure 15.6 Two clamping diodes and a 10-k Ω resistor will allow the MCLR pin with an internal reset to be used as an RS-232 input.

Interfacing to External Devices

In previous chapters I have given you a lot of information about the peripheral hardware built into the PIC microcontroller that will help to make applications easier. Coupled with the information contained in the appendices, you would have thought I have it all covered . . .

In the following sections I want to go through some of the hints and tips I've learned over the years for interfacing PIC microcontrollers with other devices. With this information, I have also included source code for the different interface hardware throughout this book. Much of this information and code will be used later in this book when I go through the experiments and projects. Some of the interfaces will seem very complex or difficult to create, but I have tried to work through many of the difficulties and provide you with sample circuits that are simple and cheap to implement.

MEMORY-MAPPED IO

It should not be surprising that the PIC microcontroller can interface directly with TTL and CMOS digital logic devices. The PIC microcontroller's parallel I/O pins provide digital output levels that can be detected properly by both logic technologies and inputs that can detect logic-level output from these logic families.

If you check the PIC microcontroller datasheets, you will see that the output characteristics are

V_{ol} ("output low voltage") = 0.6 V (max)

V_{oh} ("output high voltage") = $V_{DD} - 0.7$ V (min)

This specification is given to allow for different V_{DD} power inputs. For a V_{DD} of 5 V, you can expect a high output of 4.3 V or greater (normally I see 4.7 V when a PIC microcontroller pin is not under load). If the power voltage input (V_{DD}) is reduced to 2 V, low output still would be 0.6 V, and high output becomes 1.3 V ($V_{DD} = 0.7$) or greater.

The PIC microcontroller pins are specified to drive (source) up to 20 mA and sink (pull the output to ground). These current capabilities easily allow the PIC microcontroller to drive LEDs. The total current sourced or sunk by the PIC microcontroller should not exceed 150 mA (which is six I/O pins sinking the maximum current).

The input *threshold voltage*, the point at which the input changes from an I to an O, and vice versa, also depends on the input power V_{DD} voltage level. The threshold is different for different devices. For a number of PIC microcontroller part numbers, this value is specified as being in the range

$$0.25 V_{DD} + 0.8V \geq V_{threshold} \geq 0.48 V_{DD}$$

As a rule of thumb, you should use the higher value. For higher V_{DD} s, this is approximately one-half V_{DD} . At lower V_{DD} voltages (2 V), the threshold becomes approximately two-thirds V_{DD} .

PARALLEL-BUS DEVICES

While the PIC microcontroller is very well suited for stand-alone applications, there will be many applications that have to connect to external devices. While there are built-in PIC microcontroller interfaces for non-return to zero (NRZ) asynchronous I/O and two-wire serial I/O, sometimes the best interface is a simulated parallel I/O bus. The parallel bus is useful for increasing the I/O capabilities of the PIC microcontroller using standard I/O chips. These devices can be accessed fairly easily using an 8-bit I/O port and a few extra control pins from the PIC microcontroller.

I realize that PIC18 chips have the ability to drive a parallel bus, but it has only been recently that they have been able to access 8-bit devices (normally, they are designed for 16-bit data transfers), but the accesses take place as either two 8-bit transfers or may require using additional byte-selection pins to select between high and low bytes. If you are starting out with the PIC microcontroller, I would recommend that you tend to shy away from using these devices for this purpose because designing the circuitry for the transfers is quite involved and also requires a significant level of software architecting to make sure that the external devices do not interfere with any internal memory in the chip. It is actually much simpler to using the digital I/O pins to create a bus to access parallel-bus devices.

When I create a parallel bus, I normally use PORTB for 8 data bits and other PORT pins for the `_RD` and `_WR` lines. To avoid the extra costs and complexity of decode circuitry, it is probably best to devote one I/O line to each device's chip select pin. Before writing from the PIC microcontroller to the device, TRISB is set to output mode, and the value to be written is output on PORTB. Next, the `_CS` and `_WR` lines are pulled low and remain active until the device's minimum access times are met. `_RD` is similar, with TRISB being put in input mode; the `_CS` and `_RD` pins are held active until the devices minimum read access time is met, at which point the data is strobed into the w register, and `_CS` and `_RD` are driven high.

This is shown in the example circuit Fig. 15.7, which requires two parallel output bytes and one parallel input byte. This could be implemented with a 40-pin PIC microcontroller

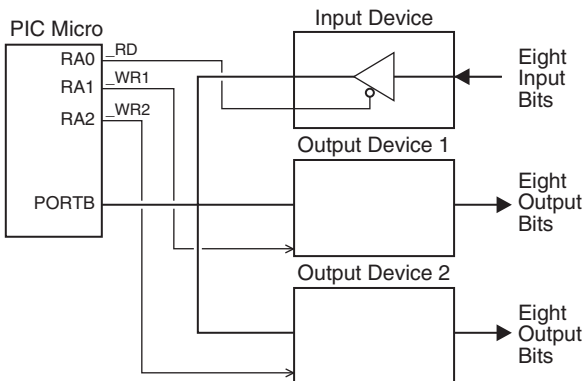


Figure 15.7 A processor bus like parallel port can be implemented using the I/O pins of the PIC microcontroller.

and use the I/O pins directly, but it is much more cost-effective to use an 18-pin PIC microcontroller (such as the PIC16F84) and a tristate output buffer and two 8-bit registers. The figure assumes that data is clocked in or out with negative active signal pulses.

With this circuit, RA0–RA2 would be set for output and initialized to 4 (high voltage) driven out. To read the 8 data bits from PORTB, the following code could be used:

```
bsf          STATUS, RPO          ; Put PORTB into Input Mode
movlw       0x0FF
movwf       TRISB ^ 0X080
bcf         STATUS, RPO
bcf         PORTA, 0              ; Drop the "_RD" line
call        Dlay                  ; Delay until Data Output
                                   ; Valid
movf        PORT B, w             ; Read Data from the Port
bsf         PORT A, 0             ; "_RD" = 1 (disable "_RD"
                                   ; Line)
```

Writing data out requires similar code:

```
bsf         STATUS, RPO
clrf        TRIS B ^ 0X080        ; PORTB Output
bcf         STATUS, RPO
bcf         PORTA, 1              ; Enable the "_WR1" Line
movwf       PORTB                 ; output the Data
call        Dlay                  ; Wait Data Receive Valid
bsf         PORTA                 ; "_WR1" = 1.
```

COMBINING INPUT AND OUTPUT

Often, when working on applications, you will find some situations where peripheral devices will use more than one pin for I/O. Another case would be when you are connecting two devices, one input and one output, and would like to combine them somehow so that you can reduce the number of PIC microcontroller pins required. Fewer PIC microcontroller pins means that you can use cheaper and less capable chips and avoid complex application wiring. In this section I will present you with two techniques for doing this and the rules governing their use. What I am presenting here may appear problematic at first and asking for problems with bus contention, but they really do work and can greatly simplify your application.

When interfacing the PIC microcontroller to a driver and receiver (such as a memory with a separate output and input), a resistor can be used to avoid bus contention at any of the pins, as shown in Fig. 15.8.

In this situation, when the PIC microcontroller's I/O pin is driving an output, it will be driving the "data in" pin register regardless of the output of the "data out" pin. If the PIC microcontroller and "data out" pins are driving different logic levels, the resistor will limit the current flowing between the PIC microcontroller and the memory "data out" pin. The value received on the "data in" pin will be the PIC microcontroller's output.

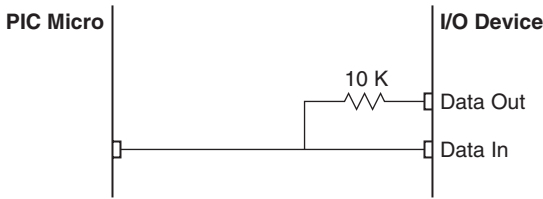


Figure 15.8 A single PIC microcontroller pin can interface with a pair of dedicated input and output pins.

When the PIC microcontroller is receiving data from the memory, its I/O pin will be put in input mode, and the “data out” pin will drive its value not only to the PIC microcontroller’s I/O pin but also to the “data in” pin. In this situation, the “data in” pin should not be latching any data in. To avoid this, in most cases where this circuit is combining input and output, the two input and output pins are on the same device, and the data mode is controlled by the PIC microcontroller to prevent invalid data from being input into the device. This is an important point because it defines how this trick should be used. The I/O pins to which the PIC microcontroller is connected to must be mutually exclusive and can never be transmitting data at the same time. A common use for this method of connection of “data in” and “data out” pins is SPI memories, which have separate data input and output pins.

The second trick is to have button input along with an external device receiver. As shown in Fig. 15.9, a button can be put on the same “net” as an input device and the PIC microcontroller pin that drives it. When the button is open or closed, the PIC microcontroller can drive data to the input device; the 100- and 10-k Ω resistors will limit

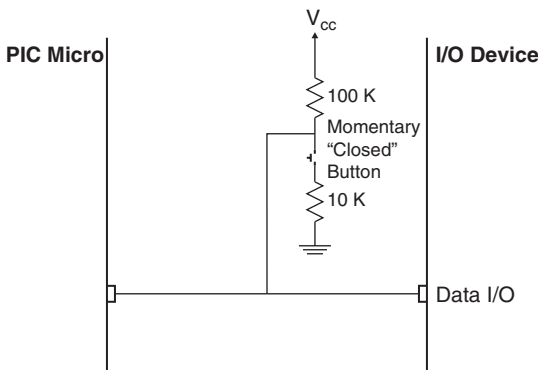


Figure 15.9 A digital interface line can be shared with a simple button input that has high-impedance pull-ups and pull-downs.

the current flow between V_{cc} and ground. If the PIC microcontroller is going to read the button high (switch open) or low (switch closed), it will be driven on the bus at low currents when the pin is in input mode. If the button switch is open, then the 100-k Ω resistor acts like a pull-up, and a 1 is returned. When the button switch is closed, then there will be approximately $\frac{1}{2}$ V across the 10-k Ω resistor, which will be read as a 0. The button with the two resistors pulling up and down are like a low-current driver, and the voltage produced by them is easily overpowered by active drivers. As with the first method, the external input device cannot receive data except when the PIC microcontroller is driving the circuit. A separate clock or enable should be used to ensure that input data is received when the PIC microcontroller is driving the line.

Two points about this method: The second approach can be extrapolated to work with a switch matrix keyboard. This can become very complex, but it will work. Second, a resistor-capacitor network for debouncing the button cannot be used with this circuit because a resistor-capacitor network will slow down the response of the PIC microcontroller driving the data input pin and cause problems with the correct value being accepted. When a button is shared with an input device; such as is shown in Fig. 15.9, software button debouncing will have to be done inside the PIC microcontroller.

SIMULATED OPEN-COLLECTOR/OPEN-DRAIN I/O

Open-collector/open-drain outputs are useful in several different interfacing situations. Along with providing a node in a dotted AND bus, they are also useful to interface with I2C and other networks. I find the single-open drain pin available in the different PIC microcontroller devices to be insufficient for many applications, which is why I find it useful to simulate an open-drain driver with a standard I/O pin.

An open-drain pin (shown in Fig. 15.10) consists of a N-channel MOSFET transistor with its source connected to the I/O pin. Because there is no P-channel high driver in the pin circuit, when a 1 is being output, the only transistor will be turned off, and

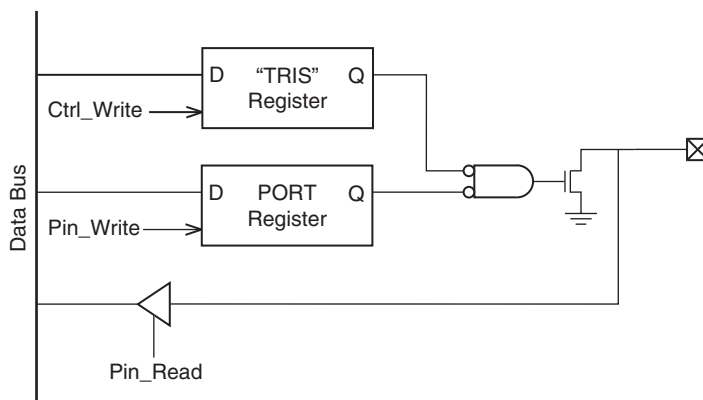


Figure 15.10 The open-drain I/O pin in the PIC microcontroller allows multiple drivers to share a single bus.

the pin is allowed to float. *Floating* generally is referred to as the case in which a pin is left unconnected; when used in this situation, the pin is at the voltage level of the rest of the net unless it is driving the net low.

When the “data out” bit is low (and TRIS is enabled for output), the pin is pulled low. Otherwise, it is not electrically connected to anything or “tristated.”

This action can be simulated by using the code listed below that enables the I/O pin output as low if the carry flag is reset. If the carry flag is set, then the pin is put into input mode.

```
bcf      PORT#, pin          ; Make Sure PORTB Pin Bit is "0"
bsf      STATUS, RPO
btfss    STATUS, C           ; If Carry Set, Disable Open
                                ; Collector
    goto  $ + 4              ; Carry Reset, Enable Open Collector
nop
bsf      TRIS ^ 0x080, pin
goto     $ + 3
bcf      TRIS ^ 0x080, pin
goto     $ + 1
bcf      STATUS, RPO
```

This code, which is designed for mid-range PIC microcontrollers, will either set the pin to input (tristate) mode or be pulled low on the sixth cycle after it has been invoked. I normally put this code into a macro, with the port and pin specified as parameters.

It will seem like I went to some lengths to ensure that the timing was the same for making the bit tristate (input mode) or pulled low to 0, as well as the state specified by the carry flag. Regardless of the execution path, this code will take eight instruction cycles, and the I/O pin value will be changed at five cycles. I did this because this function is often used with I2C or other network protocols, and using the carry flag allows bits to be shifted through the code easily.

In the sample open-drain simulation code above, I reset the specified pin before potentially changing its TRIS value. This is to prevent it from being the wrong value based on reads and writes of other I/O pins.

PIC MCU OPTIONAL HARDWARE FEATURES

As I've said throughout this book, most people think of the PIC[®] microcontroller as something like the PIC16F84, which has a processor, some program memory flash, and a number of digital input-outputs (I/Os) and is easy/cheap to program. The plethora of interfacing options available to the developer is rarely recognized because the large number of PIC microcontroller part numbers allows you to select a chip that best meets your application's requirements.

The phrase *best meets* requires a bit of consideration. When I am designing an application, I first look at it from three perspectives: overall system cost, electrical and software complexity, and responsiveness. It is important to come up with a solution that balances out all three of these dimensions. For example, you can get a PIC microcontroller that costs 50 cents or less, but it most likely will be very difficult to interface with other devices and will require a lot of software complexity and additional hardware devices, and it may take a long time to respond to user input. Similarly, a very sophisticated chip may provide the interface features that are required but have an unacceptably high cost. When I think of *best meets*, I want the various components of an application to work together effectively and the resulting application to be cost-effective and efficiently designed and work to the user's specifications.

This is why I encourage you to look at the features available within the different PIC microcontroller part numbers, and when you are learning the basics of the device, look at what is available as built-in features of other devices and think of ways that these features can simplify the task that you are trying to implement.

Mid-Range Built-in EEPROM/Flash Access

An increasingly popular feature in PIC microcontroller devices is the availability of built-in EEPROM memory that can be used to store configuration, calibration, or software data. In one of the example applications, I use it to store application source code. In mid-range devices, this feature can be accessed using the registers. In some low-end devices, EEPROM is accessed as if it were an I2C device attached to the PIC microcontroller. In the next section I will discuss how these devices work.

Along with accessing data, in some Flash program memory devices (such as the PIC16F87x and all the PIC18F chips) also can read and write Flash program memory from within the application. This feature was added originally to allow the MPLAB ICD debugger to update the Flash in circuit, but it is useful for storing larger amounts of data in a nonvolatile memory, as well as changing application codes, as I do in the EMU-II emulator and have discussed in Chap. 12.

For data EEPROM I/O, there are four registers that you should be aware of: EECON1, EECON2, EEADR, and EEDATA. These registers are used to control access to the EEPROM. As you would expect, EEADR and EEDATA are used to provide the address and data interface into the up to 256-byte data EEPROM memory. EECON and EECON2 are used to initiate the type of access as well as indicate that the operation has completed. EECON2 is a pseudoregister that cannot be read from but is written to with the data (0x55/0xAA to indicate that the write is valid).

EECON1 contains the bits listed in Table 16.1 for controlling access to the EEPROM. These bits may be in different bit positions in different devices, which is why I have not specified the bit values in the table.

TABLE 16.1 THE CRITICAL EECON1 BITS	
BIT	FUNCTION
EEPCD	Set to access program memory; reset to access data EEPROM only in PIC microcontrollers that have application/MPLAB ICD-updatable program memory Flash.
WRERR	Set if a write error is terminated early to indicate that data write may not have been successful.
WREN	When set, a write to EEPROM begins.
WR	Set to indicate an upcoming write operation; cleared when the write operation is complete.
RD	Set to indicate read operation; cleared by next instruction automatically.

Note: These bits will be in different locations in the EECON1 register of different devices.

Using these EEPROM registers, a read can be implemented using the code

```
movf / movlw  address/ADDR, w
bcf          STATUS, RPO
movwf       EEADR
bsf         STATUS, RPO
bsf         EECON1, ^ 0x08, RD
bcf         STATUS, RPO
movf        EEDATA, w           ; w = EEPROM [address/ADDR]
```

In this example code, it is assumed that these registers are in banks 0 and 1, which is true for the 16F84. For devices such as the 16F87x, however, where the ADC registers use these addresses, the EEPROM register addresses are actually in banks 2 and 3. The read operation above has to have a `bsf STATUS, RP1` at the start and `bcf STATUS, RP0` at the end, as shown below:

```
movf /movlw  address/ADDR, w
bsf          STATUS, RP1           ; Registers in Bank 3
bcf          STATUS, RP0
movwf       EEADR ^ 0x0100
bsf         STATUS, RPO
bsf         EECON 1 ^ 0x0180, RD
bcf         STATUS, RPO
movf        EEDATA ^ 0x0100, w    ; w = EEPROM [address/ADDR]
bcf         STATUS, RP1
```

Write operations are similar but have two important differences. The first is that the operation can take up to 10 ms to complete, which means that the WR bit of EECON1 has to be polled for completion or in the EPROM interrupt request hardware enabled. The second difference is that a “timed write” has to be implemented to carry out the operation.

Code to do an EPROM write could be

```
movlw /movf  constant/DATA, w
bcf          STATUS, RPO
movwf       EEDATA
movlw /movf  address/ADDR, w
movwf       EEADR
bsf         STATUS, RPO
bsf         EECON1 ^ 0x080, WREN
bcf         INTCON, GIE
movlw       0x055           ;          CRITICAL SECTION
movwf       EECON2 ^ 0x080   ;
movlw       0x0AA           ;
movwf       EECON2 ^ 0x080   ;
bsf         EECON1 ^ 0x080, WR ;
bsf         INTCON, GIE
```

```
btfscl    EECON1 ^ 0x080, WR    ; Poll for Operation Ended
goto      $ - 1                  ;
bcf       EECON1 ^ 0x080, WREN
bcf       STATUS, RPO
bsf       INTCON, GIE
```

For the devices with the EE access registers in banks 2 and 3, this code is modified in the same manner as the EEPROM read code was above.

Note that EEPROM cannot be accessed in any way until WR is reset; otherwise, there will be a WRERR.

The critically timed code (highlighted as “CRITICAL SECTION” in the example code) is used to indicate to the EEPROM access control hardware that the application is under control and that a write is desired. Any deviation in these instructions (including interrupts during the sequence) will cause the write request to be ignored by the EEPROM access control hardware.

Instead of polling, after the WR bit is set, the EEIE interrupt request bit can be set. Once the EEPROM write has completed, then the EEIF flag is set, and the hardware interrupt is requested.

For MPLAB ICD-enabled Flash devices, the program memory can be read or written to in a similar way to EEPROM data memory. The difference is the inclusion of the EEPGD bit in EECON1 that is not present in the devices with just EEPROM data memory. In the devices that do have programmable data and program memory, this bit always should be set (program memory) or reset (data memory) according to the memory access.

Along with inclusion of the EEPGD bit, there are also two additional registers used to address and access the greater than 8-bit data and number of address bits. These bits are known as (not too surprisingly) EEADRH and EEDATAH. Note that the maximum data value for EEDATAH is 3F because 14 bits per instruction is used for program memory.

To read to program memory, the following code is used for the 16F87x. Note the two nops to allow the operation to complete before the instruction is available for reading.

```
bsf       STATUS, RP1
movlw    /movwf  LOW address/ADDR, w
movwf    EEADR ^ 0x0100
movlw    /movwf  HIGH address/ADDR, w
movwf    EEADRH ^ 0x0100
bsf       STATUS, RPO
bsf       EECON1 ^ 0x0180, EEPGD
bsf       EECON1 ^ 0x0180, RD
nop                               ; NOPs used to wait for request to
nop                               ; complete before reading data
bcf       STATUS, RPO
movf     EEDATA, w
movwf    -----                ; Store Lo Byte of Program Memory
movwf    EEDATAH, w
movwf    -----                ; Store Hi Byte of Program Memory
bcf       STATUS, RP1
```

Writing to program memory is similar to writing to data but also has the two nops in which the operation takes place. No polling or interrupts are available for this operation; instead, the processor halts during this operation. Even though the processor has stopped for a program memory write, peripheral function (ADCs, serial I/O, etc.) are still active.

```

bsf          STATUS, RP1
movlw /movf   LOW address/ADDR, w
movwf        EEADR
movlw /movwf  HIGH address/ADDR, w
movwf        EEADRH
movlw /movwf  LOW Constant/DATA, w
movwf        EEDATA
movlw /movwf  HIGH Constant/DATA, w      ; Maximum 0x03F
movwf        EEDATAH
bsf          STATUS, RPO
bsf          EECON1 ^ 0x0180, EEPO
bsf          EECON1 ^ 0x0180, WREN
bcf          INTCON, GIE                  ; Critically
movlw        0x055                        ; timed
movwf        EECON2 ^ 0x0180              ; code.
movlw        0x0AA                        ;
movwf        EECON2 ^ 0x0180, OR ;
nop                                     ; Operation delay
nop                                     ; NOPs
bcf          EECON1 ^ 0x0180, WREN
bsf          INTCON, GIE

```

LOW-END BUILT-IN DATA EEPROM

The low-end architecture, with its 32-address-register page, does not have many registers or space that can be devoted to advanced peripheral I/O functions. For this reason, when PIC12C5xx parts were given built-in EEPROM (and designated the PIC12CE5xx), a fairly clever interface had to be developed. This interface consists of connecting the EEPROM within the PIC microcontroller as if it were an external I2C device. Reading and writing data is more complex than in mid-range devices (where there are registers for I/O operations). But the read/write operations are relatively simple to code.

I must point out that in the time since the second edition of this book came out, several new part numbers are available that are cheaper than the parts discussed here and use the same data EEPROM access as the mid-range and PIC18 microcontrollers described in the preceding section. While I have left this section in the third edition for completeness and to give you an example how an I2C “bit banging” interface to a single device would be implemented, I would recommend that you consider the other devices first before the PIC12CE5xx microcontrollers.

The EEPROM-included PIC12CE5xx parts use the most significant bits of the general purpose I/O (GPIO) register and its corresponding TRIS register. The PIC12CE5xx’s EEPROM interface can be described as I’ve shown in the block diagram in Fig. 16.1.

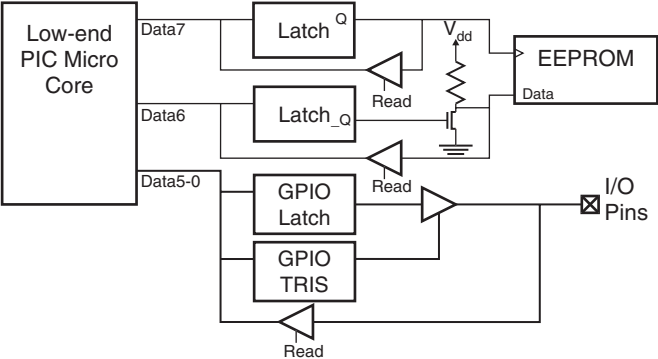


Figure 16.1 The low-end PIC microcontroller EEPROM solutions consist of the MCU interfacing to the EEPROM via I2C.

In this figure, GPIO bits 6 and 7n do not have TRIS control bits. In addition, bit 6 (the PIC12Cxx EEPROM bit SDA) has an open-drain driver. This driver circuit is designed to let both the PIC microcontroller and the EEPROM drive the data line at different intervals without having to disable the PIC microcontroller’s write of the EEPROM.

Information is written to the EEPROM device using the waveform shown in Fig. 16.2. Note that for timing I use instruction cycles for a PIC microcontroller running at 4 MHz.

The start and stop bits are used to indicate the beginning and end of an operation and can be used halfway through to halt an operation. The start and stop bits actually are invalid cases (data cannot change while one clock is active or high).

This operation means that the GPIO port must be accessed carefully; always make sure that the SDA and SCL GPIO bits have a 1 in them or else the built-in EEPROM

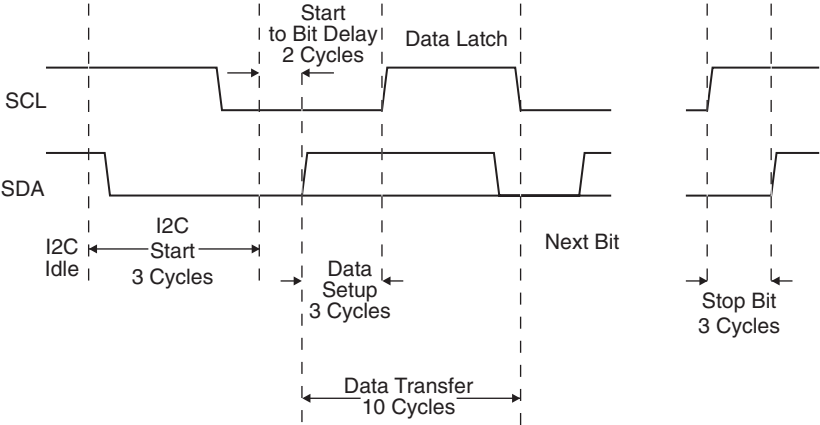


Figure 16.2 The basic data waveforms that must be produced by the low-end PIC microcontroller to access the built in EEPROM.

may be accessed incorrectly, causing problems with subsequent reads. You should never use the instruction

```
clrf    GPIO
```

Data is written to the most significant bit first, which is probably backwards to most applications. Before any transfer, a control byte has to be written. The control byte data is in the format

```
0b1010000R
```

where R is the `Read/_Write` byte (indicating what is coming next). If this bit is set, then a read of the EEPROM at the current address pointer will take place. If a write is to take place, the read/write bit is reset.

After a byte is sent, the SDA line is pulled low to indicate an acknowledgment (ACK or just A in the bitstream representations below). This bit is set low (as an acknowledgment) when the operation has completed successfully. If the acknowledgment bit is high (NACK), it does not necessarily mean there was a failure; if it is issued by the EEPROM, then it indicates a previous write has not completed. The PIC microcontroller will issue the acknowledgment to stop the EEPROM from preparing to send additional bytes out of its memory in a multibyte read.

There are five operations that can be carried out with the EEPROM that is built into the PIC12CE5xx. They are

- 1** Current address set
- 2** Current address set/data byte write
- 3** Data byte read at current address
- 4** Sequential (multibyte) read at current address
- 5** Write completion poll

The EEPROM in the PIC12CE5xx is only 16 bytes in size. Each byte is accessed using a 4-bit address. This address is set using a control byte, with the R bit reset followed by the address. The bitstream looks like this:

```
idle – Start – 1010000A – 0000addrA – DataByteA – Stop – idle
```

In the second byte sent, the `0b00000addr` pattern indicates that the four `addr` address bits become the address to set the EEPROM's internal address pointer to for subsequent operations.

After the 2 bytes have been sent, the SCL and SDA lines are returned to IDLE for three cycles using the instruction

```
movlw  0xC0
iorwf  GPIO, f           ; set SDA /SCL
```

before another operation can complete.

The address data write is similar to the address write but does not force the two lines into IDLE mode, and it passes along a data byte before stopping the transfer:

```
Idle – Start – 10100000A – 0000addrA – DataByteA – Stop – idle
```

Data bytes can be read singly or sequentially depending on the state of ACK from the PIC microcontroller to the EEPROM after reading a byte. To halt a read, when the last byte to be read has been received, the PIC microcontroller issues a NACK (or N in the bitstream listing) to indicate that the operation has completed.

A single-byte read looks like this:

```
idle – Start – 10100001A – DataByteN – Stop – idle
```

whereas a 2-byte read looks like this:

```
idle – Start – 10100001A – DataByteA – DataByteN – Stop – idle
```

The last operation is sending dummy write control bytes to poll the EEPROM to see whether or not a byte write has completed (10 ms is required). If the write has completed, then an ACK will be returned; otherwise, a NACK will be returned.

This is a pretty cursory explanation of how the PIC12CE5xx's built-in EEPROM works. In later chapters I will include a more comprehensive explanation of accessing I2C and provide you with code examples to do it.

I do want to make one point on the `flash15x-ASM` code you will see referenced in the 12CE5xx datasheet and on the Microchip web page. This file is designed to be linked into your application and provide the necessary I2C routines to access the EEPROM memory. Unfortunately, this file is quite difficult to set up correctly, and there are no instructions for using it.

If you do want to use the `flash15x.ASM` file, then there are a few things to do:

- 1 Install it so that it occupies memory in the first 256 bytes of the PIC microcontroller. The file should not be put at the start of program memory because this will interfere with the PIC microcontroller's reset.
- 2 Declare `EEADDR` and `EEDATA` in your file register variable declarations.
- 3 Make sure that the `#define emulated` line is commented out. If this line is left in, code will be generated that will attempt to write to the SDA and SCL bits (which don't exist) and in the process will set all the GPIO bits to output.

TMR1

Along with TMR0, many PIC microcontrollers have an additional 16-bit (TMR1) and 8-bit (TMR2) timer built into them. These timers are designed to work with the compare/capture program hardware feature. Along with enhancing this module, they also can be used as straight timers within the application. TMR1 (Fig. 16.3 shows the block diagram

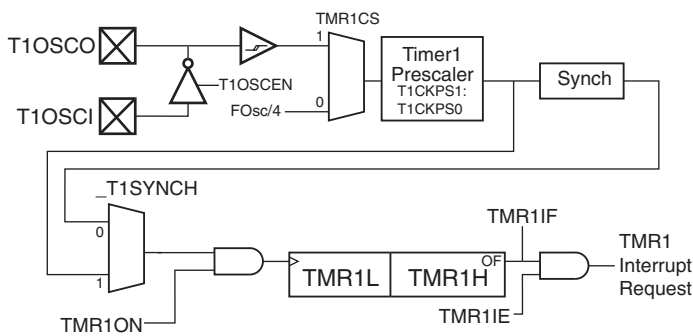


Figure 16.3 TMR1 block diagram.

of the timer) is a 16-bit timer that has four possible inputs. What is most interesting about TMR1 is that it can use its own crystal to clock the timer. This allows TMR1 to run while the PIC microcontroller’s processor is “asleep.”

To access TMR1 data, the TMR1L and TMR1H registers are read and written. Just as in TMR0, if the TMR1 value registers are written, the TMR1 prescaler is reset. A TMR1 interrupt request (TMR1IF) is made when TMR1 overflows. TMR1 interrupt requests are passed to the PIC microcontroller’s processor when the TMR1IE bit is set.

TMR1IF and TMR1IE normally are located in the PIR and PIE registers. To request an interrupt, along with TMR1IE and GIE being set, the INTCON PIE bit also must be set. To control the operation of TMR1, the T1CON register is accessed with its bits defined as shown in Table 16.2.

The external oscillator is designed for fairly low-speed real-time clock applications. Normally, a 32.768-kHz watch crystal is used, along with two 33-pF capacitors. Additionally,

TABLE 16.2 T1CON REGISTER BIT DEFINITION	
BIT	DESCRIPTION
7–6	Unused
5–4	T1CPS1–T1CPS0—Select TMR1 prescaler value
	11—1:8 prescaler
	10—1:4 prescaler
	01—1:2 prescaler
	00—1:1 prescaler
3	T10SLEN—Set to enable TMR1’s built-in oscillator.
2	T1SYNCH—When TMR1CS is reset, the MR1 clock is synchronized to the instruction clock.
1	TMR1CS—When set, external clock is used.
0	TMR1ON—When set, TMR1 is enabled.

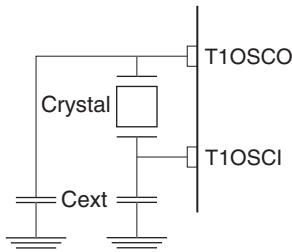


Figure 16.4 TMR1 can be driven by its own separate 32.768-kHz oscillator.

100- or 200-kHz crystals could be used with TMR1, but the capacitance required for the circuit changes to 15 pF. The TMR1 oscillator circuit is shown in Fig. 16.4.

When TMR1 is running at the same time as the processor, the T1SYNCH bit should be reset. This bit will cause TMR1 to be synchronized with the instruction clock. If the TMR1 registers are to be accessed during processor execution, resetting T1SYNCH will make sure that there are no clock transitions during TMR1 access. T1SYNCH must be set (no synchronized input) when the PIC microcontroller is in sleep mode. In sleep mode, the main oscillator is stopped, stopping the synchronization clock to TMR1.

In the PIC18 devices, TMR1 can be specified as the processor clock. This feature is one way to implement a low-current operating mode (the PIC microcontroller will run while drawing less than 1 mA of current) without disabling the entire device and its built-in functions. Note that returning to the normal program oscillator will require the 1024-instruction-cycle and optional 72-ms power-up reset delay that occurs when the PIC microcontroller clock starts up.

The TMR1 prescaler allows 24-bit instruction cycle delay values to be used with TMR1. These delays can be either a constant value or an overflow, similar to TMR0. To calculate a delay, the formula

$$\text{Delay} = (65,536 - \text{TMR1Init}) \times \text{prescaler} / \text{T1frequency}$$

is used, where the T1frequency can be the instruction clock, TMR1 oscillator, or an external clock driving TMR1. Rearranging the formula, the TMR1Init initial value can be calculated as

$$\text{TMR1Init} = 65,536 - (\text{Delay} \times \text{T1Frequency} / \text{prescaler})$$

When calculating delays, the prescaler will have to be increased until the calculated TMR1Init is positive—this is similar as to how the TMR0 prescaler and initial value are calculated for TMR0.

TMR2

TMR2 is used as a recurring event timer (see Fig. 16.5). When it is used with the CCP module, it is used to provide a PWM timebase frequency. In normal operations, it can be used to create a 16-bit instruction cycle delay.

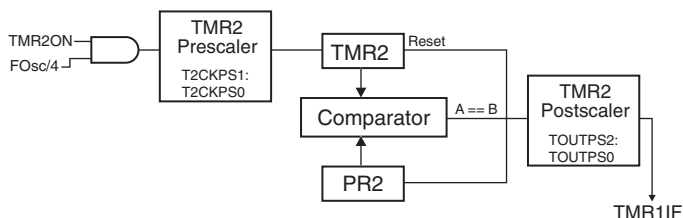


Figure 16.5 TMR2 block diagram.

TMR2 is continually compared against the value in PR2. When the contents of TMR2 and PR2 match, TMR2 is reset, and the event is passed to the CCP as TMR2 Reset. If the TMR2 is to be used to produce a delay within the application, a postscaler is incremented when TMR2 overflows and eventually passes an interrupt request to the processor. TMR2 is controlled by the T2CON register, which is defined in Table 16.3.

The TMR2 register can be read or written at any time with the usual note that writes cause the prescaler and postscaler to be zeroed. Updates to T2CON do not affect the TMR2 prescaler or postscaler.

The timer itself is not synchronized with the instruction clock like TMR0 and TMR1 because it can be used only with the instruction clock. This means that TMR2 can be incremented on a 1:1 instruction clock ratio.

PR2 contains the reset, or count up to value. The delay before reset is defined as

$$\text{Delay} = \text{prescaler} \times (\text{PR2} + 1) / (\text{Fosc} / 4)$$

TABLE 16.3 T2CON REGISTER BIT DEFINITION

BIT	DESCRIPTION
7	Unused
6–5	TOUTPS3–TOUTPS0—TMR2 postscaler select 1111—16:1 postscaler 1110—15:1 postscaler . . . 0000—1:1 postscaler
2	TMR2ON—When set, TMR2 is enabled
1–0	T2CKPS—TMR2 prescaler selection bits 1x—16:1 prescaler 01—4:1 prescaler 00—1:1 prescaler

If PR2 is equal to zero, the delay is

$$\text{Delay} = (\text{prescaler} \times 256) / (\text{Fosc} / 4)$$

I do not usually calculate TMR2 delays with an initial TMR2INIT value. Instead, I take advantage of the PR2 register to provide a repeating delay and just reset TMR2 before starting the delay.

To calculate the delay between TMR2 overflows (and interrupt requests), the following formula is used:

$$\text{Delay} = (\text{prescaler} \times [\text{PR2} + 1 | 256]) / ((\text{Fosc} / 4) \times \text{postscaler})$$

Interrupts use the TMR2IE and TMR2IF bits that are similar to the corresponding bits in TMR1. These bits are located in the PIR and PIE registers. Because of the exact interrupt frequency, TMR2 is well suited for applications that provide “bit banging” functions such as asynchronous serial communications and PWM signal outputs.

Compare/Capture/ PWM (CCP) Module

Included with TMR1 and TMR2 is a control register and a set of logic functions (known as the *CCP*) that enhances the operation of the timers and can simplify your applications. This hardware may be provided singly or in pairs, which allows multiple functions to execute at the same time. If two CCP modules are built into the PIC microcontroller, then one is known as *CCP1* and the other as *CCP2*. In the case where two CCP modules are built in, then all the registers are identified with the *CCP1* or *CCP2* prefix. The CCP hardware is controlled by the CCP1CON (or CCP2CON) register, which is defined in Table 16.4.

The most basic CCP mode is capture, which loads the CCPR registers (CCPR14, CCPR1C, CCPR2H, and CCPR2L) according to the mode the CCP register is set in. This function is illustrated in Fig. 16.6 and shows that the current TMR1 value is saved when the specified compare condition is met.

Before enabling the capture mode, TMR1 must be enabled (usually running with the PIC microcontroller clock). The “edge detect” circuit in the figure is a 4:1 multiplexor, which chooses between the prescaled rising-edge input or a falling-edge input and passes the selected edge to latch the current TMR1 value and optionally request an interrupt.

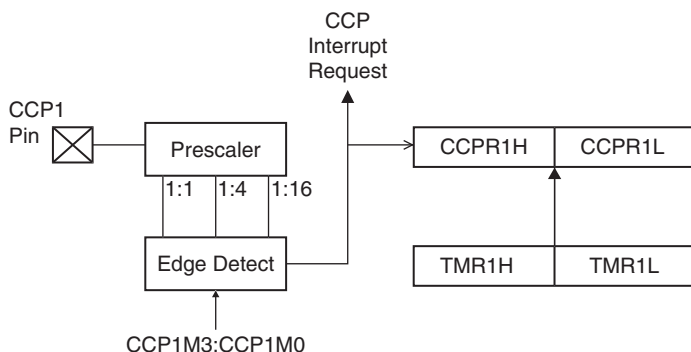
In capture mode, TMR1 is running continuously and is loaded when the condition on the CCPx pin matches the condition specified by the CCPxMS:CCPxM0 bits. When a capture occurs, then an interrupt request is made. This interrupt request should be acknowledged and the contents of CCPRxH and CCPRxL saved to avoid having them written over and the value in them lost.

TABLE 16.4 CCPXCON REGISTER BIT DEFINITIONS

BIT	FUNCTION
7–6	Unused
5–4	DC1B1–DC1B0—CEPST significant 2 bits of the PWM compare value.
3–0	CCP1M3–CCP1M0—CCP module operating mode
	11xx—PWM mode
	1011—Compare mode, trigger special event
	1010—Compare mode, generate software interrupt
	1001—Compare mode, on match, CCP pin low
	1000—Compare mode, on match, CCP pin high
	0111—Capture on every sixteenth rising edge
	0110—Capture on every fourth rising edge
	0101—Capture on every rising edge
	0100—Capture on every falling edge
	00xx—CCP off

Capture mode is used to time-repeating functions or in determining the length of a PWM pulse. If a PWM pulse is to be timed, then when the start value is loaded, the polarity is reversed to get to the end of the pulse. When timing a PWM pulse, the TMR1 clock must be fast enough to get a meaningful value with a high enough resolution that there will be an accurate representation of the timing.

Compare mode changes the state of the CCPx pin of the PIC microcontroller when the contents of TMR1 match the value in the CCPRxM and CCPRxL registers as shown in Fig. 16.7. This mode is used to trigger or control external hardware after a specific delay.

**Figure 16.6** Block diagram of CCP capture circuitry.

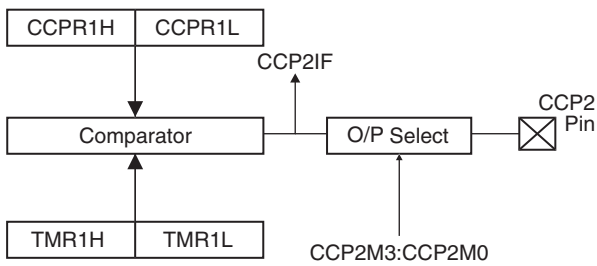


Figure 16.7 Block diagram of CCP compare circuitry.

The most interesting use I’ve seen for the compare mode of the CCP is to turn the PIC microcontroller into a “watchdog” for a complex system. As is shown in Fig. 16.8, the PIC microcontroller controls reset to the system processor. On power-up, the PIC microcontroller holds the processor reset until Vcc has stabilized, and then the TMR1 is reset each time the system writes to the PIC microcontroller. System reset is enabled if after a time-out delay Vcc falls below a specific level.

Using event-driven code, the PIC microcontroller application would look like this:

```
PowerUpEvent() // PIC microcontroller Power Up
{
    TMR1 = 0; TMR1 = on; // Start TMR1

    CCPRx = PowerUpDelay; // Put in Watchdog Delay
    CCPxCON = 0b000001000; // Drive Pin Low and /then High
    // on Compare Match
    ADCIE = on; // Start ADC Check of Vcc
} // End PowerUpEvent

CompareMatchEvent( ) // TMR1 = Compare / WDT T/O.
{
```

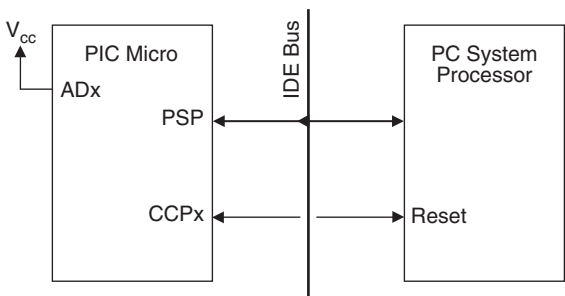


Figure 16.8 PC watchdog timer using PIC microcontroller with the CCP compare circuitry enabled.

```

    CCPxCON = 0;                // Turn off compare.
    CCPx = 1;                   // Reset system

} // End CompareMatchEvent

PSPWriteEvent( )               // PSP Written to Reset WDT
{
    // Count

    TMR1 = 0;

} // End PSPWriteEvent

ADCIFEvent( )                 // ADC Finished Vcc check
{

    if (ADC < OperatingMinimum) {
        CCPxCON = 0;           // Turn Off ADC
        CCPx = high;           // Reset system program
    }

    ADCIF = 0;                 // Reset Interrupt Request

} // End ADCIFEvent

```

PWM OPERATION

Of the three CCP modes, I find the PWM signal generator to be the most useful. This mode outputs a PWM signal using the TMR2 reset at a specific value capability. The block diagram of PWM mode is shown in Fig. 16.9. The mode is a combination of the normal execution of TMR2 and capture mode; the standard TMR2 provides the PWM period, whereas the compare control provides the “on” time specification.

When the PWM circuit executes, TMR1 counts until its most significant 8 bits are equal to the contents of PR2. When TMR2 equals PR2, TMR2 is reset to 0, and the CCPx pin is set high. TMR2 is run in a 10-bit mode (the 4:1 prescaler is enabled before PWM operation). This 10-bit value is then compared with a program value in CCPRxM (along with the two DCxBx bits in CCPxCON), and when they match, the CCPx output pin is reset low.

To set up a 65 percent duty cycle in a 20-kHz PWM executing in a PIC microcontroller clocked at 4 MHz, the following steps are taken: First, the CCPRxM and PR2 values are calculated for TMR2; the 4:1 prescaler must be enabled, resulting in a delay of

$$\text{Delay} = (\text{PR2} + 1) * 4 / (\text{frequency}/4)$$

$$\begin{aligned}
 \text{PR2} &= \text{delay} * \text{frequency} - 1 \\
 &= 50\text{msec} * 4\text{mHz} - 1 \\
 &= 200 - 1 \\
 &= 199
 \end{aligned}$$

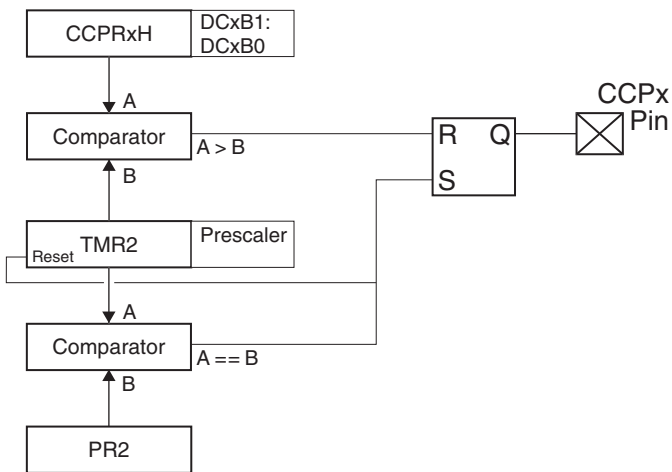


Figure 16.9 CCP PWM generation circuitry block diagram.

Then, 65 percent of 200 is 130, which is loaded into CCPRxM.

The code for creating the 65 percent 20-kHz PWM is

```
movlw 199
movwf PR2                                ; Set up TMR2 Operation
movlw (1 << TMR2on) + 1
movwf T2CON                               ; Start it Running with a 50 msec
                                           ; Period
movlw 130                                ; 65% of the Period
movwf CCPRxH
movlw (1<<DCxB1) + 0x00F
movwf CCPxCON                             ; Start PWM

; PWM is operating
```

Note that in this code I don’t enable interrupts or have to monitor the signal output. In addition, you should notice that I don’t use the fractional bits. To use the 2 least significant bits, I assume that they are fractional values. For the preceding example, if I wanted to fine-tune the PWM frequency to 65.875 percent, I would recalculate the value as a fraction of the total period.

For a period of 200 TMR2 counts with a prescaler of 4, the CCPRxH value becomes 131.75. To operate the PWM, I would load 130 into CCPRxh (subtracting 1 to match TMR2’s zero start) and then the fractional value 0.75 into DCxB1 and DCxB0 bits. I assume that DCxB1 has a value of 0.50 and that DCxB0 has a fractional value of 0.25. Thus, to get a PWM in this case, CCPRxH is loaded with 130, and DCxB1 and DCxB0 are both set. Table 16.5 gives the fractional DCxBX bit values.

TABLE 16.5 CCP DCXBX BIT DEFINITION

FRACTION	DCXB1–DCXB0
0.00	00
0.25	01
0.50	10
0.75	11

The least significant 2 bits of the PWM obviously are not that important unless a very small PWM “on” period is used in an application. A good example of this is using the PWM module for an R/C servo. In this case, the PWM period is 20 ms with an “on” time of 1 to 2 ms. This gives a PWM pulse range of 5 to 10 percent, which makes the DCxB1 and DCxB0 bits important in positioning the servo accurately.

Serial I/O

As with many microcontrollers, the PIC microcontroller has optional built-in serial I/O interfacing hardware. These interfaces, which are available on certain PIC microcontroller part numbers, allow a PIC microcontroller to interface with external memory and enhanced I/O functions (such as ADCs) or communicate with a PC using RS-232. As with other enhanced peripheral features, the serial I/O hardware is available on different PIC microcontrollers, and the hardware may be available differently in different devices.

SYNCHRONOUS SERIAL PORT (SSP) COMMUNICATIONS MODULE

In discussing the synchronous serial port (SSP), I am first going to discuss its basic operations, followed by the I2C operations in the next section. There are two reasons why I break operation of the SSP into two parts, the first being that I2C is quite a complex operation that I feel would be best served by a discussion in its own section.

The second reason for splitting out the I2C function is the inability of two SSI versions to provide the full range of I2C operations. The SSP and BSSP modules, which are available in many PIC microcontroller part numbers, do not have I2C master mode capabilities. This limits their usefulness in working with I2C (where typically the PIC microcontroller is a master) as compared with PIC microcontrollers equipped with the MSSP (master SSP) module, which does have I2C multimaster capabilities.

SPI (its data stream is shown in Fig. 16.10) is an 8-bit synchronous serial protocol that uses 3 data bits to interface with external devices. Data is clocked out, with the most

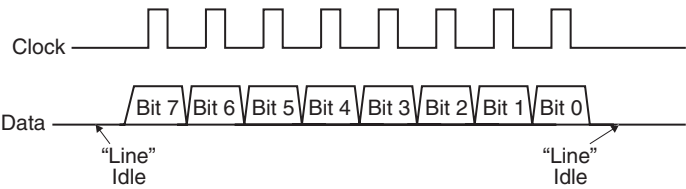


Figure 16.10 SPI data and clock waveform.

significant bit first, on rising or falling edges of the clock. The clock itself is generated within the PIC microcontroller (master mode), or it is provided by an external device and used by the PIC microcontroller (slave mode) to clock out the data.

The clock can be positive, as shown in the figure with a 0 idle or negative (high line idle) with a 1 idle and the clock pulsing to 0 and back again. The data receive latch is generally on the return to idle state transition.

The BSSP module is the basic SSP module and provides data pulling on the return to idle clock edge. The original SSP module provides the ability to vary when data is output and read.

Controlling the operation of the different SSP modules is the SSPCON register. In describing the operational bits, note that I only describe the SPI-specific operations in Table 16.6.

TABLE 16.6 SSP/BSSP SSPCON REGISTER BIT DEFINITION	
BIT	FUNCTION
7	WCOL—Write collision, set when new byte written to SSPBUF while transfer is taking place.
6	SSPOV—Receive overflow, indicates that the unread byte is SSPBUF overwritten while in SPI slave mode.
5	SSPEN—Set to enable the SSP module.
4	CKP—Clock polarity select, set to have a high idle.
3–0	SSPM3–SSPM0—SPI mode select 1xxx—I2C and reserved modes 011x—I2C slave modes 0101—SPI slave mode, clock = SCK pin, _SS not used 0100—SPI slave mode, clock = SCK pin, _SS enabled 0011—SPI master mode, TMR2 clock used 0010—SPI master mode, INСК/16 0001—SPI master mode, INСК/4 0000—SPI master mode, INСК

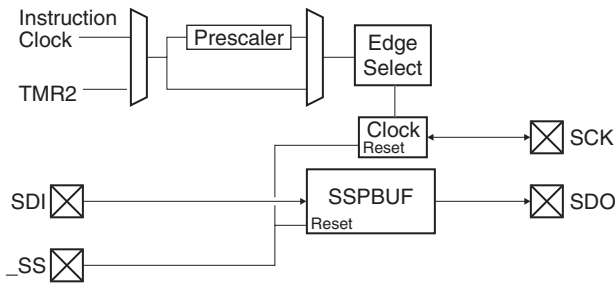


Figure 16.11 Block diagram of SSP SPI module.

The block diagram for the SSP module is shown in Fig. 16.11. In master mode, when a byte is written to SSPBUF, an 8-bit most-significant-bit-first data transfer process is initiated. The status of the transfer can be checked by the SSPSTAT register BF flag; the SSPSTAT register is defined as shown in Table 16.7.

The connection of a PIC microcontroller to an SPI bus is quite straightforward. In Fig. 16.11, two PIC microcontrollers are shown with the SDO and SDI sides connected. To initiate a byte transfer, a byte is written to the SSPBUF of the master. Writing to the SSPBUF of the slave will not initiate a transfer. When SPI mode is enabled, the SDI, SDO, and SCK bits' TRIS functions are set appropriately. This is shown in Fig. 16.12.

The SSP SPI transfers can be used for single-byte synchronous serial transmits of receivers with serial devices. In Fig. 16.13 I show the circuit to TX a byte to a 74LS374

TABLE 16.7 SSP/BSSP SSPSTAT BIT DEFINITION

BIT	FUNCTION
7	SMP—Set to have data sampled after active to idle transition; reset to sample at active-to-idle transition; not available in BSSP.
6	CKE—Set to TX data on idle-to-active transition; else TX data on active-to-idle transition; not available in BSSP
5	D/_A—Used by I2C.
4	P—Used by I2C.
3	S—Used by I2C.
2	R/_W—Used by I2C.
1	UA—Used by I2C.
0	BF—Busy flag, reset while SPI operation active.

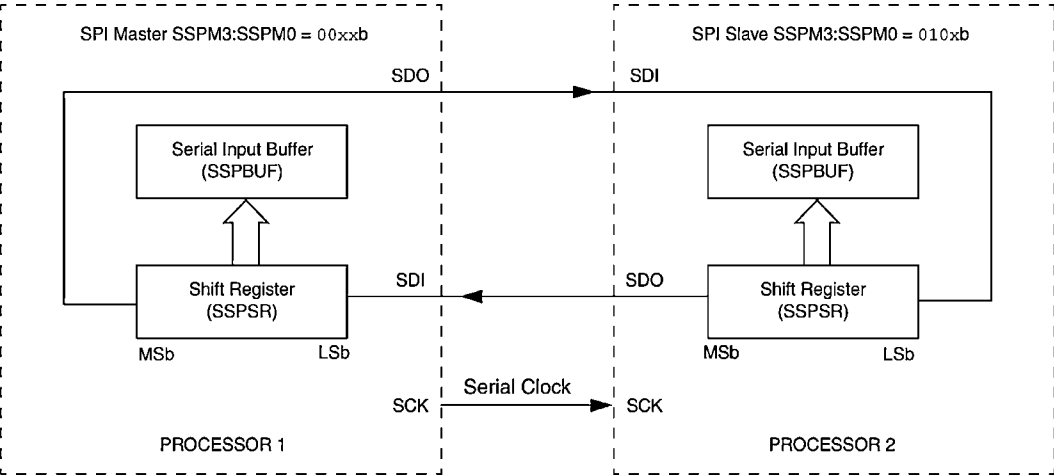


Figure 16.12 SPI master/slave connection.

wired as a serial in, parallel out shift register. In Fig. 16.14, I show a 74LS374 being used with a 74LS244 as a synchronous parallel in, serial out register. Both these operations are initiated by a write to SSPBUF.

The SPI data receive operation may be immediately obvious. To latch data into the 74LS374, the I/O pin is driven high; this disables the 74LS374’s drivers, allowing the parallel data to be latched in. When the I/O pin is low, the 74LS244’s drivers are disabled, and the 74LS374 behaves like a shift register.

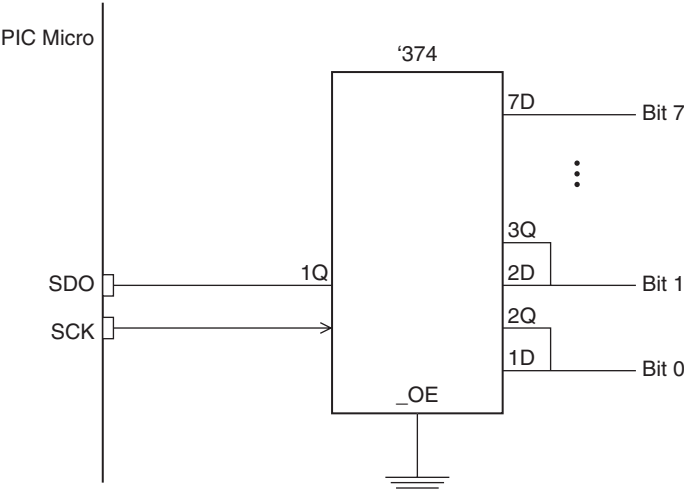


Figure 16.13 Wiring connection to pass SPI data from the PIC microcontroller to a serial-to-parallel converter.

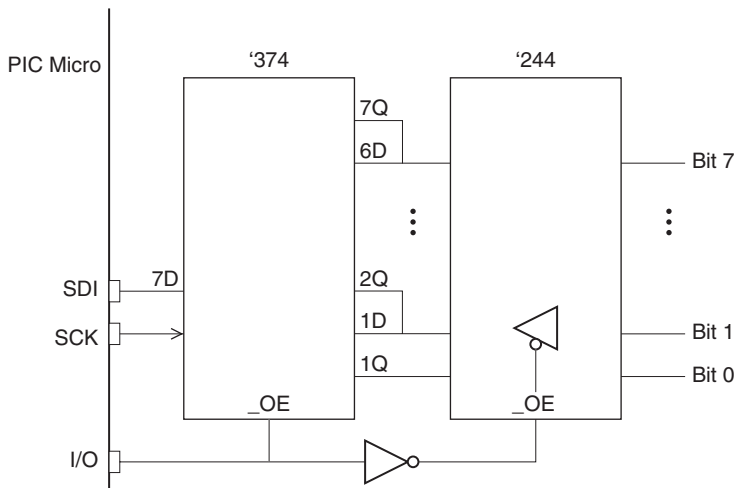


Figure 16.14 Wiring to bring data from a parallel-to-serial converter into a PIC microcontroller.

To show this, here is some code to read the input state of Fig. 16.14. Note that I disable the SSPEN bit when a transfer is not taking place to allow the I/O pin and SCK to strobe in the data.

```

bsf      IOPin                ; Want to Latch Data into the '374
bcf      SCK
bsf      STATUS, RPO
bcf      IOPin
bcf      SCK
bcf      STATUS, RPO
bsf      SCK                ; Latch the Data into the '374
bcf      SCK
bcf      IOPin              ; Disable '244 output, Enable '374
movlw    (I << SMP) + (I << CKE)
movwf    SSPSTAT            ; Set up the SSP Shift In
movlw    (I << SSPEN) + (I << CKP) + 0x000
movwf    SSPCON
movf     TXData, f          ; Load the Byte to Send
movwf    SSPBUF            ; Start Data Transfer
btfss    SSPTAT, BF
    goto $ - 1              ; Wait for Data Receive to Complete
; Data Ready in SSPBUF when Execution
; Here
bcf      SSPCON, SSPEN      ; Turn off SSP

```

When using the SSP, the data rate either can be selected as a multiple of the executing clock or use the TMR2 overflow output. The actual timing depends on the hardware the PIC microcontroller SSP master is communicating with.

When in slave mode, along with an external clock being provided, there is a transmit reset pin known as `_SS`. When this pin is asserted high, the SSP output is stopped (the SDO TRIS bit is changed to input mode), and the SSP is reset with a count of zero. When the bit is reset, the clock will start up again, and the original most significant bit is reset, followed by the remaining 7 bits.

MASTER SSP AND I2C OPERATION

When I wrote the first edition of this book, one of the most significant concerns people had with the PIC microcontroller's built-in hardware was the lack of master and multimastering I2C capability. This concern has been resolved with the availability of the MSSP (master SSP) module that is included in new PIC microcontroller devices. The original SSP and BSSP will continue to be available in devices that currently have them, but the enhanced MSSP will be designed into all new devices that have the SSP module.

When you look at the MSSP datasheets, you'll see that there are 33 pages documenting how the function works. When you actually work with the MSSP, you will find that very few instructions are actually required to implement the function, and their use is quite easy to understand. In this section I will concentrate on a single master I2C interface and point out the issues that you will have to be aware of when working in a multimaster system.

Five registers are accessed for MSSP I2C operation; they are the SSP control registers (SSPCON and SSPCON2 in Tables 16.8 and 16.9), the SSP status register (SSPSTAT), the SSP receive/transmit register (SSPBUF), and the SSP address register (SSPADD). These registers are available in the SSP and BSSP but are slightly different for MSSP.

The status of the transfer can be checked by the SSPSTAT register BF flag; the SSPSTAT register is defined in Table 16.10.

I2C connections between the PIC microcontroller's I2C SDA (data) and SCL (clock) pins is very simple, with just a pull-up on each line, as shown in Fig. 16.15. I typically use a 1-k Ω resistor for 400-kHz data transfers and a 10k Ω resistor for 100-kHz data rates.

Note that before any of the I2C modes are used, the TRIS bits of the respective SDA and SCL pins *must* be in input mode. Unlike many of the other built-in advanced I/O functions, MSSP does not control the TRIS bits. Not having the TRIS bits in input mode will not allow the I2C functions to operate.

In master mode, the PIC microcontroller is responsible for driving the clock (SCL) line for the I2C network. This is done by selecting one of the SPI master modes and loading the SSPADD register with a value to provide a data rate that is defined by the formula

$$\text{I2C Data Rate} = F_{\text{osc}} / (4 * (\text{SSPADD} + 1))$$

This can be rearranged to

$$\text{SSPADD} = (F_{\text{osc}} / (4 * \text{I2C Data Rate})) - 1$$

Thus, in a 4-MHz PIC microcontroller, to define a 100-kHz I2C data rate, the preceding formula would be used to calculate the value loaded into SSPADD:

TABLE 16.8 MSSP SSPCON BIT DEFINITION

BIT	FUNCTION
7	WCOL—Write collision, set when new byte written to SSPBUF while transfer is taking place.
6	SSPOV—Receive overflow, indicates that the unread byte is SSPBUF overwritten.
5	SSPEN—Set to enable the SSP module
4	In I2C modes, if bit is reset, the I2C SCL clock line is low—Keep this bit set.
3–0	SSPM3–SSPM0 SPI mode select 1111—I2C 10-bit master mode/start and stop bit interrupts 1110—I2C 7-bit master mode/start and stop bit interrupts 1101—Reserved 1100—Reserved 1011—I2C master mode with slave idle 1010—Reserved 1001—Reserved 1000—I2C master mode with SSPADD clock definition 0111—I2C slave mode, 10-bit address 0110—I2C slave mode, 7-bit address 0101—SPI slave mode, clock = SCK pin, _SS not used 0100—SPI slave mode, clock = SCK pin, _SS enabled 0011—SPI master mode, TMR2 clock used 0010—SPI master mode, INSCCK/16 0001—SPI master mode, INSCCK/4 0000—SPI master mode, INSCCK

$$\begin{aligned}
 \text{SSPADD} &= (\text{Fosc} / (4 * \text{I2C Data Rate})) - 1 \\
 &= (4 \text{ MHz} / (4 * 100 \text{ kHz})) - 1 \\
 &= (4 \times 10^6 / (4 * 100 \times 10^3)) - 1 \\
 &= 10 - 1 \\
 &= 9
 \end{aligned}$$

In a PIC microcontroller running at 4 MHz, a value of 9 must be loaded into the SSPADD to have a 100-kHz I2C data transfer.

TABLE 16.9 MSSP SSPCON2 BIT DEFINITION	
BIT	FUNCTION
7	GCEN—Enable interrupt when “general call address” (0x0000) is received.
6	ACKSTAT—Received acknowledge status; set when acknowledge was received.
5	ACKDT—Acknowledge value driven out on data write.
4	ACKEN—Acknowledge sequence enable bit, which when set will initiate an acknowledge sequence on SDA/SCL; cleared by hardware.
3	RCEN—I2C receive enable bit
2	PEN—Stop condition initiate bit; when set, stop condition on SDA/SCL; cleared by hardware.
1	RSEN—Set to initiate the repeated start condition on SDA/SCL; cleared by hardware.
0	SEN—When set, a start condition is initiated on the SDA/SCL; cleared by hardware.

To send data from the PIC microcontroller to an I2C device using the MSSP, the following steps must be taken:

- 1 The SDA/SCL lines must be put into input mode (i.e., their respective TRIS bits must be set).
- 2 I2C master mode is enabled. This is accomplished by setting the SSPEN bit of SSPCON and writing 0b01000 to the SSPM3–SSPM0 bits of the SSPCON register.

TABLE 16.10 MSSP SSPSTAT BIT DEFINITION	
BIT	FUNCTION
7	SMP—Set to have data sampled after active-to-idle transition; reset to sample at active-to-idle transition; not available in BSSP.
6	CKE—Set to TX data on idle-to-active transition; else TX data on active-to-idle transition; not available in BSSP.
5	D/_A—Used by I2C.
4	P—Used by I2C.
3	S—Used by I2C.
2	R/_W—Used by I2C.
1	UA—Used by I2C.
0	BF—Busy flag; reset while SPI operation active.

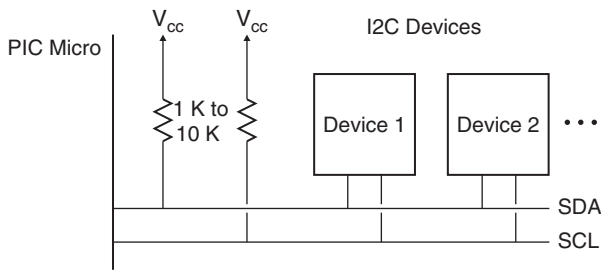


Figure 16.15 Typical I2C connection to a PIC microcontroller with separate pull-up resistors.

- 3 A start condition is initiated by setting the SEN bit of SSPCON2. This bit is then polled until it is reset.
- 4 SSPBUF is loaded with the address of the device to access. Note that for many I2C devices, the least significant bit transmitted is the read/write bit. The R/_W bit of SSPSTAT is polled until it is reset (which indicates that the transmit has been completed).
- 5 The ACK bit from the receiving device is checked by reading the ACKDT bit of the SSPCON2 register.
- 6 SSPBUF is loaded with the first 8 bits of data or a secondary address that is within the device being accessed.
- 7 The R/_W bit of SSPSTAT is polled until it is reset.
- 8 The ACK bit from the receiving device is checked by reading the ACKDT bit of the SSPCON2 register.
- 9 A new start condition may have to be initiated between the first and subsequent data bytes. This is initiated by setting the SEN bit of SSPCON2. This bit is then polled until it is reset.
- 10 Operations 6 through 8 are repeated until all data is sent or a NACK (negative acknowledge) is received from the receiving device.
- 11 A stop condition is initiated by setting the PEN bit of SSPCON2. This bit is then polled until it is reset.

This sequence of operations is shown in Fig. 16.16. Note that in the figure the SSPIF interrupt request flag operation is shown. In the preceding sequence, I avoid interrupts, but the SSPIF bit can be used either to request an interrupt or to avoid the need to poll different bits to wait for the various operations to complete.

To receive data from a device employs a similar set of operations, with the only difference being that after the address byte(s) have been sent, the MSSP is configured to receive data when the transfer is initiated:

- 1 The SDA/SCL lines *must* be put into input mode (i.e., their respective TRIS bits must be set).
- 2 I2C master mode is enabled. This is accomplished by setting the SSPEN bit of SSPCON and writing 0b01000 to the SSPM3–SSPM0 bits of the SSPCON register.

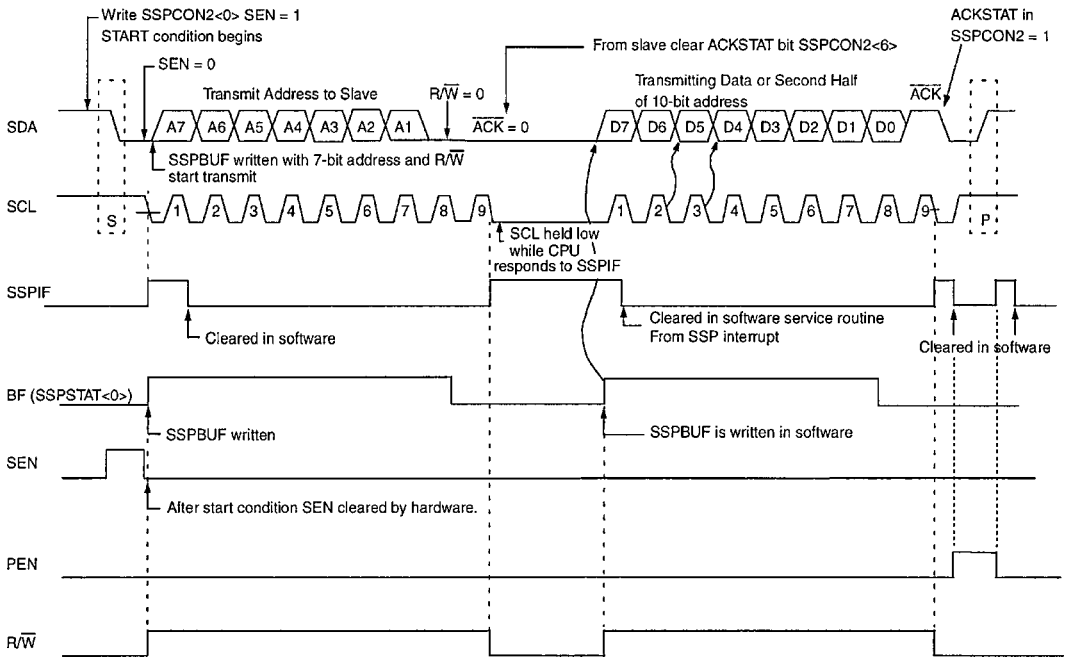
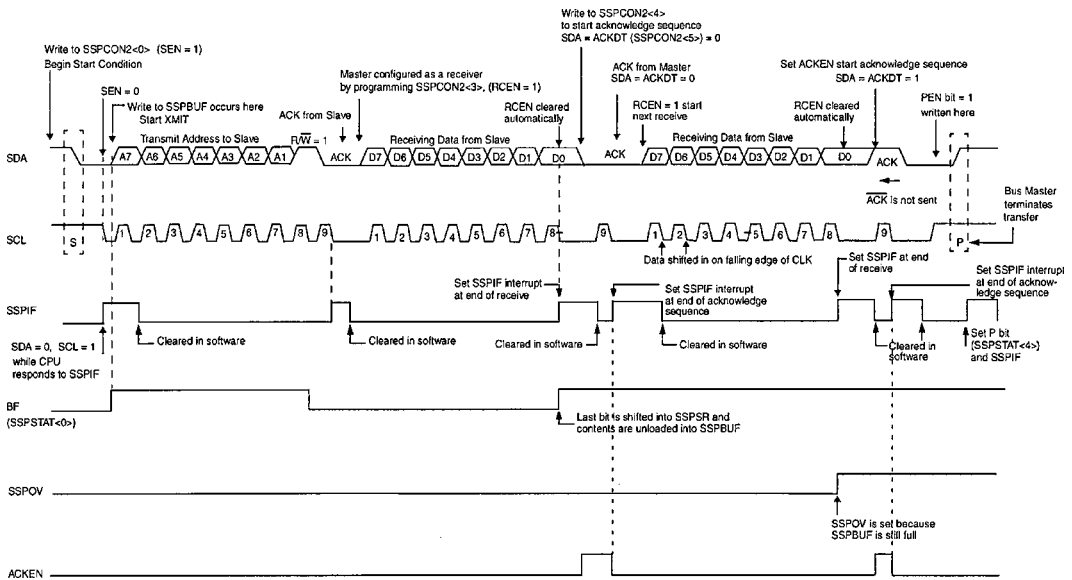


Figure 16.16 MSSP I2C data address/transmission.

- 3** A start condition is initiated by setting the SEN bit of SSPCON2. This bit is then polled until it is reset.
- 4** SSPBUF is loaded with the address of the device to access. Note that for many I2C devices, the least significant bit transmitted is the read/write bit. The R/_W bit of SSPSTAT is polled until it is reset (which indicates that the transmit has been completed).
- 5** The ACK bit from the receiving device is checked by reading the ACKDT bit of the SSPCON2 register.
- 6** SSPBUF is optionally loaded with the secondary address within the device being read from. The R/_W bit of SSPSTAT is polled until it is reset.
- 7** If a secondary address was written to the device being read from, reading the ACKDT bit of the SSPCON2 register checks the ACK bit from the receiving device.
- 8** A new start condition may have to be initiated between the first and subsequent data bytes. This is initiated by setting the SEN bit of SSPCON2. This bit is then polled until it is reset.
- 9** If the secondary address byte was sent, then a second device address byte (with the read indicated) may have to be sent to the device being read. The R/_W bit of SSPSTAT is polled until it is reset.
- 10** The ACKDT will be set (NACK) or reset (ACK) to indicate whether or not the data byte transfer is to be acknowledged in the device being read.
- 11** The RCEN bit in the SSPCON2 register is set to start a data byte receive. The BF bit of the SSPSTAT register is polled until the data byte has been received.

**Figure 16.17** MSSP I2C data address/read.

- 12** Operations 10 and 11 are repeated until all data is received, and a NACK (negative acknowledge) is sent to the device being read
- 13** A stop condition is initiated by setting the PEN bit of SSPCON2. This bit is then polled until it is reset.

Figure 16.17 show the data receive operation waveform

Along with the single master mode, the MSSP is also capable of driving data in multimaster mode. In this mode, if a data write “collision” is detected, it stops transmitting data and requests an interrupt to indicate that there is a problem. An I2C *collision* is the case where the current device is transmitting a high data value, but there is a low data value on the SDA line. This condition is shown in Fig. 16.18. The WCOL bit of the SSPCON register indicates that the collision has taken place.

When the collision occurs, the I2C software must wait some period of time (I use the time required to transmit 3 bytes) before polling the SDA and SCL lines to ensure that they are high and then initiating a “repeated start condition” operation. A repeated start condition is the process of restarting the I2C data transfer right from the beginning (even if it was halfway through when the collision occurred).

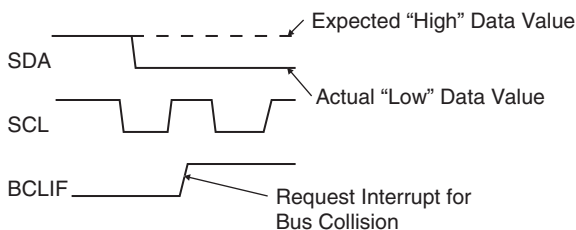


Figure 16.18 I2C MSSP collision response.

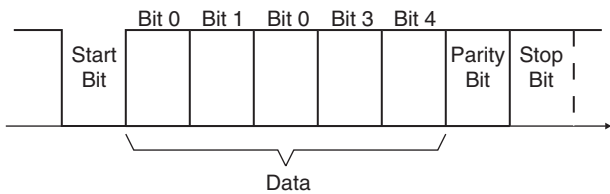


Figure 16.19 Baudot asynchronous serial data.

USART ASYNCHRONOUS SERIAL COMMUNICATIONS

The PIC microcontroller’s universal asynchronous synchronous receiver transmitter (USART) hardware allows you to interface with serial devices such as a PC using RS-232 or for synchronous serial devices, with the PIC microcontroller providing the clock or having an external clock drive the data rate. The USART module is best suited for asynchronous serial data transmission, and in this section, I will be concentrating on its capabilities.

Asynchronous data has been discussed elsewhere in more detail in this book. The PIC microcontroller transmits and receives NRZ (no return to zero) asynchronous data in the format shown in Fig. 16.19. The figure shows 5 bits of serial data—the PIC microcontroller can transfer 8 or 9 bits although by setting the high-order bits of the output word; smaller data packets can be sent.

If the USART is used for synchronous data, the bits can be latched into the destination on the falling edge of the clock. In both these cases, a byte is sent within a packet. While I have discussed packet decoding in detail elsewhere in this book, in this section I’ll tend to treat the packet encoding and decoding as a “black box” part of the USART and deal with how the data bytes are transmitted and received.

There are three modules to the USART: the clock generator, the serial data transmission unit, and the serial data reception unit. The two serial I/O units require the clock generator for shifting data out at the write interval. The clock generator’s block diagram is shown in Fig. 16.20.

In the clock generator circuit, the SPBRG register is used as a comparison value for the counter. When the counter is equal to the SPBRG register’s value, a clock “tick” output is made, and the counter is reset. The counter operation is gated and controlled by the SPEN (serial port enable) bit along with the synch (which selects whether the port is in synchronous or asynchronous mode) and BRGH (which selects the data rate) bits.

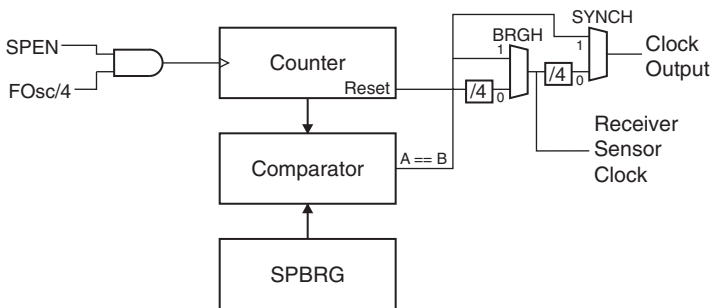


Figure 16.20 USART clock block diagram.

Unfortunately, in the PIC microcontroller USART, the bits used to control the operation of the clock generator, transmit unit, and receive unit are spread between the TXSTA and RCSTA registers, along with the interrupt enable and acknowledge registers. The individual bits will be defined at the end of this section, after the three functions of the USART are explained.

For asynchronous operation, the data speed is specified by the formula

$$\text{Data Rate} = F_{\text{osc}} / (16 \times (4^{**} (1 - \text{BRGH})) \times (\text{SPBRG} + 1))$$

This formula can be rearranged so that the SPBRG value can be derived from the desired data rate:

$$\text{SPBRG} = F_{\text{osc}} / (\text{Data Rate} \times 16 \times (4^{**} (1 - \text{BRGH}))) - 1$$

Thus, for a PIC microcontroller running at 4 MHz, the SPBRG value for a 1,200 bps data rate with BRGH reset is calculated as

$$\begin{aligned} \text{SPBRG} &= F_{\text{osc}} / (\text{Data Rate} \times 16 \times (4^{**} (1 - \text{BRGH}))) - 1 \\ &= 4 \text{ MHz} / (1200/\text{sec} \times 16 \times (4^{**} (1 - 0))) - 1 \\ &= 4 (10^{**6}) / (1200 \times 16 \times 4) - 1 \\ &= 52.0833 - 1 \\ &= 51.0833 \end{aligned}$$

With 51 stored in SPBRG, there will be an actual data rate of 1,201.9 bps, which has an error rate of 0.16 percent to the target data rate of 1,200 bps. This error is well within limits to prevent any bits from being read in error.

There is one thing that I should note about the USART clock generator, and that is that for many “early” PIC microcontroller part numbers, the BRGH bit does not work properly when it is set. This is not an issue with PIC microcontroller part numbers issued after 2000, but you should be aware of this if you are working with something like a PIC16C74, which was released around 1996—the USART will not work properly with the BRGH bit set. If you are working with EPROM (C technology indicator in the part number), I recommend that you always develop your applications with the BRGH bit reset. If you need data rates faster than what is possible for the PIC microcontroller clock (2,400 bps is the maximum for a 4-MHz clock), I recommend that you increase the PIC microcontroller’s clock speed rather than risk setting BRGH in a device in which it may not work properly.

The transmission unit of the USART can send 8 or 9 bits in a clocked (synchronous) or unclocked (asynchronous) manner. The block diagram of the hardware is shown in Fig. 16.21. If the synch bit is set, then data is driven out on the PIC microcontroller’s RX pin, with the data clock being either driven into or out of the TX pin. When data is loaded into the TXREG, if CSRC is reset, then an external device will clock it out. If CSRC can be shifted 8 or 9 bits at a time, with the operation stopping when the data has been shifted out. An interrupt can be requested when the operation is complete.

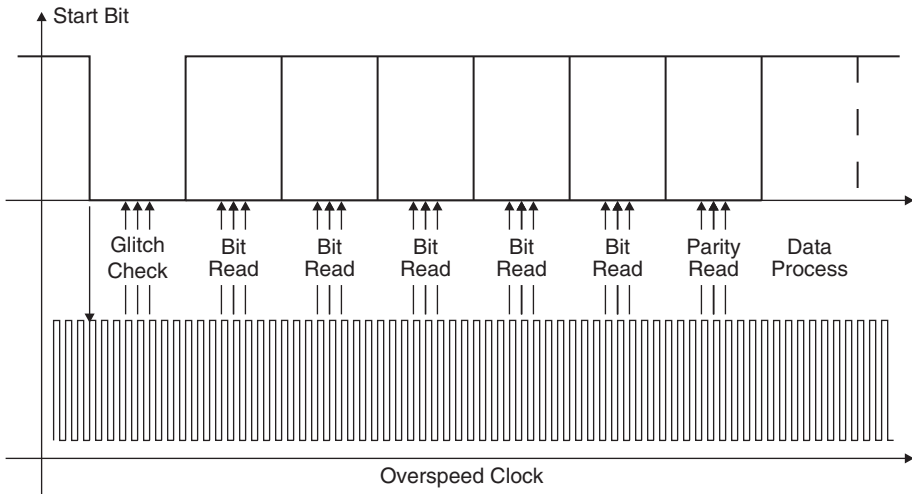


Figure 16.23 Asynchronous packet detection and data reading are implemented using a clock that runs at 16 times the data rate and samples the data at what it calculates as the middle of each data bit.

control unit provides a polling clock for the hardware. When the input data line is low for three receive sensor clock periods, data is then read in from the “middle” of the next bit, as shown in Fig. 16.23. When data is being received, the line is polled three times, and most states read is determined to be the correct data value. This repeats for the 8 or 9 bits of data, with the stop bit being the final check.

As with the TX unit, the RX unit has a holding register, so if data is not processed immediately and an incoming byte is received, the data will not be lost. However, if the data is not picked up by the time the next byte has been received, then an *overflow* error will occur. Another type of error is the *framing error*, which is set if the stop bit of the incoming NRZ packet is not 0. These errors are recorded in the RCSTA (receiver status) register and have to be reset by software.

In some PIC microcontrollers, the USART receive unit also can be used to receive two synchronous bytes in the format `data:address`, where `address` is a byte destined for a specific device on a bus. When the `adden` bit of the RCSTA register is set, no interrupts will be requested until both the address and data bytes have been received. To distinguish between the bytes, the ninth address bit is set (while the ninth bit of data bytes is reset). When this interrupt request is received, the interrupt handler checks the device address for its value before responding to the data byte.

To control the USART, two registers are used explicitly. The TXSTA (transmitter status) register is defined in Table 16.11, and the RCSTA (receiver status) register is defined in Table 16.12.

To set up an asynchronous serial communication transmit, the following code is used:

```
bsf    STATUS, RPO
bcf    TXSTA, SYNCH      ; Not in Synchronous mode
bcf    TXSTA, BRGH      ; BRGH = 0
```

TABLE 16.11 USART TXSTA BIT DEFINITION	
BIT	DEFINITION
7	CSRC—Clock source select used in synchronous mode; when set, the USART clock generator is used.
6	TX9—Set to enable 9-bit serial I/O.
5	TXEN—Set to enable data transmission.
4	SYNC—Set to enable synchronous transmission.
3	Unused
2	BRGH—Used in asynchronous mode to enable fast data transmission; it is recommended to keep this bit reset.
1	TRMT—Set if the transmission shift register is empty.
0	TXD—Nine-bit of transmitted data.

```
movlw   DataRate           ; Set USART Data Rate
movwf   SPBRG

bcf     STATUS, RPO        ; Enable serial port
bsf     RCSTA ^ 0x080, SPEN
bsf     STATUS, RPO
bcf     TXSTA, TX9         ; Only 8 bits to send
bsf     TXSTA, TXEN        ; Enable Data Transmit
bcf     STATUS, RPO
```

TABLE 16.12 USART RCSTA BIT DEFINITION	
BIT	DEFINITION
7	SPEN—Set to enable the USART.
6	RX9—Set to enable 9-bit USART receive.
5	SREN—Set to enable single-byte synchronous data receive; reset when data has been received.
4	CREN—Set to enable continuous receive.
3	ADDEN—Set to receive data address information; may be unused in many PIC microcontroller part numbers.
2	FERR—Framing error bit.
1	OERR—Overrun error bit.
0	RX9D—Received ninth bit.

To send a byte in WREG, use the following code:

```

btfss    TXSTA, TRMT
    goto  $ - 1          ; Wait for Holding Register to
                        ; become Free/Empty
movwf    TXREG           ; Load Holding Register

                        ; When the Transmit Shift Register
                        ; is Empty, byte will be sent

```

In the data send code, the TRMT bit, which indicates when the TX holding register is empty, is polled. When the register is empty, the next byte to send is put into the transmit shift register. This polling loop can be eliminated by setting the TXIE bit in the interrupt control register and then in your interrupt handler, checking to see if the TXIF flag is set before saving a byte in TXREG.

To set up an asynchronous read, the following code is used:

```

bsf      STATUS, RPO
bcf      TXSTA, SYNCH    ; Want Asynch Communications
bcf      TXSTA, BRGH     ; Low Speed Clock
movlw    DataRate        ; Set Data Rate
movwf    SPBRG
bsf      RCSTA ^ 0x080, SPEN ; Enable Serial Port
bcf      TCSTA ^ 0x080, RX9 ; Eight Bits to Receive

```

To wait to receive data and then store the incoming byte in WREG, you can use the following code:

```

btfss    PIR1, RXIF      ; Wait for a Character to be
    goto  $ - 1          ; Received
movf      RCREG, w        ; Get the byte Received
bcf      RXIF            ; Reset the RX byte Interrupt
                        ; Request Flag

```

Analog I/O

Depending on your experience level, you may feel that the PIC microcontroller has quite limited analog I/O capabilities. This is especially true if you are looking for high-speed analog operation, including signal analysis. The Microchip PIC mid-range microcontrollers actually have relatively good analog I/O capabilities, although for high-speed analog I/O you may want to look at external ADCs and DACs (or even other microcontrollers or circuits) that do provide high-speed capabilities.

The ADC built into the PIC microcontrollers can sample and process signals as fast as 25 kHz or so accurately. Looking at the ADC's specifications, you might feel that the best analog signal frequency that can be processed is 50 kHz (because the examples in

the datasheet show a 19-ms acquisition/processing time). I specify 25 kHz because of Nyquist's sampling theorem, which says that to sample an analog signal properly, you must sample at twice the highest data frequency expected in the signal.

Thus 25 kHz may seem like a reasonably fast signal to sample; after all, speech only requires 2.5 kHz, and full-spectrum audio is only up to about 18 kHz. For most electronic signals, though, 25 kHz is actually quite a low speed and not very useful (e.g., the AM radio band starts at 66 kHz and NTSC monochrome composite video runs at 3.5 MHz).

Along with the slow ADC sampling and processing speeds, digital signal processing (DSP) algorithms are difficult to implement on the PIC microcontroller because of the processor's ability to interface with only 8 bits of data and lack of multiply instructions except in the PIC18. Limited DSP functions can be implemented, but they will be challenging for data input waveforms that are faster than 1 kHz or so. For these reasons, I don't recommend that the PIC microcontroller's built-in ADCs be used for anything other than measuring dc voltages. With up to 12 bits available with built-in ADCs, the PIC microcontroller is very well suited for making accurate measurement of dc analog voltages.

Personally, I find the Microchip documentation to be quite complex and difficult to figure out how to use the built-in ADC hardware for applications. In the following sections I'll go through how the analog input and processing works on the PIC microcontroller, and I provide some hints for using the features without having to wade through all the documentation.

VOLTAGE COMPARATORS

The simplest way of inputting analog voltages in the PIC microcontroller is to use the optional comparators that indicate when a voltage is greater than another voltage. The inputs compared can be switched between different I/O pins as well as ground or a reference voltage that can be generated inside the PIC microcontroller chip. Enabling built-in comparators is a very straightforward operation, with the only prerequisite being that the pins used for the analog compare must be in input mode. Comparator response is virtually instantaneous, which allows alarm or other fast responses from changes in the comparator inputs.

The comparator works very conventionally, as shown in Fig. 16.24. If the value of the `+input` is greater than the `-input`, the output is high. There are two comparators in the PIC16C82X controlled by the CMCON register, which is defined in Table 16.13. The CIS and CM2–CM0 bits work together to select the operation of the comparators, and the resulting I/O pin configurations are listed in Table 16.14.

On power-up, the comparator CM bits are all reset, which means that RA0 to RA3 are in analog input mode. If you want to disable analog input, the CM bits must be set (write 0x007 to CMCON).

Interrupts can be enabled that will interrupt the processor when one of the comparator output changes. This is enabled differently for each PIC microcontroller with built-in comparators. Like the PORTB change on interrupt, after a comparator change interrupt request has been received, the CMCON register must be read to reset the interrupt handler.

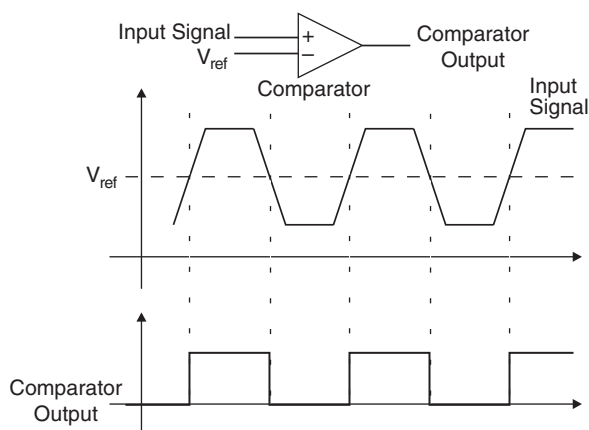


Figure 16.24 A comparator returns a digital value each time an analog voltage is at a specific value.

Along with comparing to external values, the PIC16C62x also can generate a reference voltage (V_{ref} in the preceding table) using its own built-in 4-bit digital-to-analog converter. The digital-to-analog converter circuit is shown in Fig. 16.25.

The V_{ref} control bits are found in the VRCON register and are defined in Table 16.15.

The V_{ref} output depends on the state of the VRR bit. The V_{ref} voltage output can be expressed mathematically if VRR is set as

$$V_{ref} = V_{dd} * (Vfcon \& 0x00F) / 24$$

or if it is reset as

$$V_{ref} = V_{dd} * (8 + (Vrcon \& 0x00F)) / 32$$

Note that when VRR is set, the maximum voltage of V_{ref} is 15/24 of V_{dd} , or just less than two-thirds V_{dd} . When VRR is reset, V_{ref} can be almost three-quarters of V_{dd} .

TABLE 16.13 COMPARATOR CMCON REGISTER BIT DEFINITIONS

BIT	DESCRIPTION
7	C2OUT—Comparator 2 output (high if + > -).
6	C1OUT—Comparator 1 output (high if + > -).
5–4	Unused
3	CIS—Comparator input switch.
2–0	CM2–CM0—Comparator mode.

TABLE 16.14 COMPARATOR MODULE I/O SPECIFICATION					
CM	CIS	COMP +INPUT	COMP -INPUT	COMP 2 -INPUT	COMP 2 -INPUT
000	X	RA0	RA3 ^a	RA2 ^b	RA1 ^c
001	0	RA2	RA0	RA2	RA1
001	1	RA2	RA3	RA2	RA1
010	0	Vref	RA3	Vref	RA1
010	1	Vref	RA3	Vref	RA2
011	X	RA2	RA0 ^d	RA2	RA1
100	X	RA3	RA0 ^d	RA2	RA1
101	X	Don't care	Don't care	RA2	RA1
110	X	RA2	RA0 ^e	RA2	RA1 ^f
111	X	RA3	RA0 ^g	RA2	RA1 ^h

^aFor CM2–CM0 equal to 000, RA3 through RA0 cannot be used for digital I/O.

^bFor CM2–CM0 equal to 000, RA2 and RA1 cannot be used for digital I/O.

^cRA0 and RA3 can be used for digital I/O.

^dRA3 can be used for digital I/O.

^eRA3 is a digital output, same as comparator 1 output.

^fRA4 is the open-drain output of comparator 2.

^gRA0 and RA3 can be used for digital I/O.

^hRA1 and RA2 can be used for digital I/O.

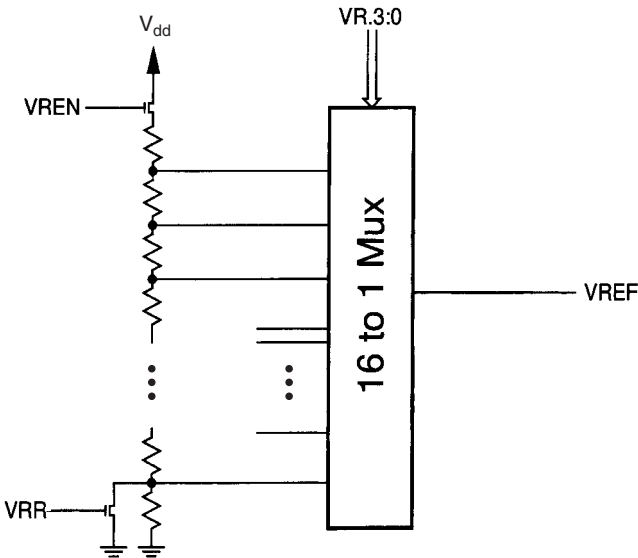


Figure 16.25 The comparator Vref circuit selects an analog voltage from a resistor divider.

TABLE 16.15 COMPARATOR VRCON REGISTER BIT DEFINITIONS

BIT	DESCRIPTION
7	VREN—Vref, enable when set.
6	VROE—Vref output, enable when set RA2—Vref.
5	VRR—Vref range select. 1—Low range. 0—High range.
4	Unused.
3–0	VR3–VR0—Voltage selection bits.

Vref can be output on RA2, but you probably will want it to be used internally only with the comparators because using it to drive RA2 requires extra current from the application. If an analog voltage output is required, I have used the comparator module's Vref in several applications, and I also have used a PWM output driving a resistor and capacitor network. In either case, the resulting analog voltage will have to be buffered using something like a unity gain op-amp circuit because the current output capabilities are very modest.

ANALOG INPUT

When I first started using the analog-to-digital converter (ADC) built into the PIC16C7x devices, I felt like the feature was very complex and difficult to work with. When you read through the Microchip datasheets on the ADC that is built into the different PIC microcontroller part numbers, you will find that there are multiple 20-page descriptions of the ADC. Each of these descriptions is slightly different in terms of register locations and bit operations depending on the part number and its features (such as the number of I/O pins that can provide ADC input), but they all work essentially the same way. In this section I want to give you a brief overview of the basics of ADCs, along with the important concepts that you will have to know to use them in your applications. At the end of this section I have provided some sample code to help guide you in using the ADC successfully in your own applications.

Some time ago, it was easy to tell which PIC microcontroller part numbers had an ADC built into them; they were the ones with a 7 as the character after the C or F of the part number. Now, many different PIC microcontrollers have a built-in ADC that will indicate an analog voltage level from 0 to Vdd with 8- or 10-bit accuracy, and you will have to look at their datasheets to understand which ones have this capability.

The ADC inputs usually are situated in the PORTA I/O pins and can be used as either digital I/O or analog inputs. The actual bit accuracy, utilization of pins, and operating speed are a function of the PIC microcontroller part number and the clock speed at which

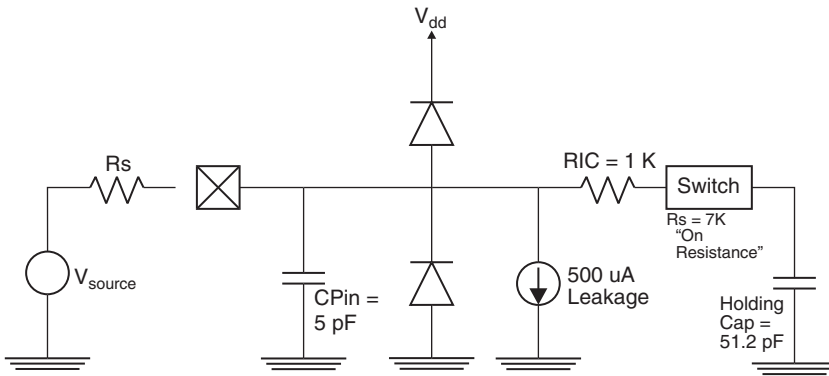


Figure 16.26 The equivalent to the PIC microcontroller ADC input circuitry.

the PIC microcontroller runs. When a pin is configured for analog input, it follows the models shown in Fig. 16.26.

R_s in the V_{source} circuit is the in-line resistance of the power supply. In order to get reasonable times for charging the ADC's holding capacitor, this value should be less than 10 k Ω . If you look through the ADC documentation, you will find that the time required for the holding capacitor to load the analog voltage and to stabilize is

$$T_{ack} = 5 \text{ ms} + [(\text{temp} - 25^\circ\text{C}) \times 0.05 \text{ ms}/^\circ\text{C}] + (3.19\text{C} \times 10^{**7}) \times (8\text{k} + R_s)$$

which works out to anywhere from 7.6 to 10.7 μs at room temperature. I usually avoid this calculation altogether and assume that at least 12 μs is required for the holding capacitor voltage to stabilize to the input voltage.

Once the voltage is stabilized within the holding capacitor, a test for each bit is made; 9.5 cycles are required to do an 8-bit conversion. The bit conversion cycle time (known as TAD) can be anywhere from 1.6 to 6.4 μs and can use either the PIC microcontroller's instruction clock or a built-in 250-kHz RC oscillator. To get a valid TAD time using the PIC microcontroller's instruction clock, a 2, 8, or 32 prescaler is built into the ADC. For example, a 4-MHz clock using the divide by 8 prescaler will have a 2- μs TAD time, which is acceptable for the ADC. If the divide by 2 counter were used, the TAD would be 500 ns, which is much too fast for the ADC to work correctly.

The built-in 250-kHz oscillator is used to carry out the ADC conversion when the PIC microcontroller is asleep. Microchip recommends that the PIC microcontroller be put to sleep during the ADC conversion for maximum accuracy (and minimum internal voltage or current upsets). If the PIC microcontroller is put to sleep, then the minimum conversion time is much longer than what is possible using the built-in clock because the PIC microcontroller has to restart when the ADC completion interrupt has been received.

The minimum conversion time is defined as the total time required for the holding capacitor to stabilize at the input voltage and for the ADC operation to complete. Assuming that a 12- μs holding time could be implemented along with a 15- μs ADC conversion time, the maximum time is about 27 μs , or 37,000 ADC samples per second can be implemented.

TABLE 16.16 PORTA BIT MODES AS DEFINED BY THE ADCON1 BITS OF A PIC16C71

ADCON1 BITS	AN3	AN2	AN1	AN0
11	Digital	Digital	Digital	Digital
10	Digital	Digital	Analog	Analog
01	Vref (Input)	Analog	Analog	Analog
00	Analog<P>	Analog	Analog	Analog

This is not fast enough for most electronics operations and probably not fast enough for audio decoding (especially with the slow digital processing capabilities of the PIC microcontroller's processor). I'm pointing this out to indicate that the PIC microcontroller's ADC is best used for relatively slowly changing inputs.

To measure analog voltages, the analog input pins of the PIC microcontroller, which are usually in port A, have to be set to analog input on power-up; the analog input pins normally are set to analog input and not digital I/O. To specify the modes, the ADCON1 register is written to. For example, in Table 16.16 I have listed the different ADCON1 bit values and corresponding PORTA pin operational modes in the PIC16C71.

When I use an ADC-equipped PIC microcontroller in an application where all the PORTA pins have to be digital I/O, I normally set the pins to digital operations right at the start of the application, before writing to the TRISA register or initializing the state of the pins. Until the pins are changed to digital I/O, they always will return 0 and cannot be set to an output value of 1.

Normally, when the ADC is used in a PIC microcontroller, the voltage reference is from ground to Vdd. If this range is not acceptable, or if the power supply is unreliable, a new reference voltage can be specified. In some devices that are equipped with ADCs, the lower-voltage reference can be specified externally as well. The bit definition of ADCON1 is part-number-specific and changes based on the device part number, number of PORTA pins, and the number of bit resolution provided by the ADC. There is one generality that you should be aware of: When the ADCON1 bits are all set, the PORTA I/O pins are all put into digital operation.

The ADCON0 register is used to control operation of the ADC. The bits of the register typically are defined as in Table 16.17.

The ADC consumes power even when it is not being used, and for this reason, if the ADC is not being used, ADON should be reset.

If the PIC microcontroller's ADC is capable of returning a 10-bit result, the data is stored in the two ADRESH and ADRESL registers. When 10-bit ADC results are available, the data can be stored in ADRESH/ADRESL in two different formats. The first is to store the data "right justified" with the most significant 6 bits of ADRESH loaded with 0 and the least two significant bits loaded with the two most significant bits of the result. This format is useful if the result is going to be used as a 16-bit number, with all the bits used to calculate an average.

TABLE 16.17 ADCON0 REGISTER BIT DEFINITIONS	
BIT	FUNCTION
7–6	ADCS1–ADCS0 bits, used to select the TAD clock. 11—Internal 250-kHz oscillator 10—FOSC/32 01—FOSC/8 00—FOSC/2
5–3	CHS2–CHS0—Bits used to select which analog input is to be measured; these bits and their operation are part number specific.
2	GO/_DONE—Set bit to start ADC conversion; reset by hardware when ADC conversion is complete.
1	ADIF—Set on completion of ADC conversion and requests an interrupt.
0	ADON—Set to enable the ADC.

The second 10-bit ADC result format is “left justified,” and the 8 most significant bits are stored in ADRESH. This format is used when only an 8-bit value is required in the application, and the two least significant bits can be “lopped” off or ignored.

To do an analog to digital conversion, the following steps should be taken:

- 1 Write to ADCON1 indicating what the digital I/O pins are and which are the analog I/O pins. At this time, if a 10-bit conversion is going to be done, then set the format flag in ADCON 1 appropriately.
- 2 Write to ADCON0, setting ADON, resetting ADIF and GO/_DONE, and specifying the ADC TAD clock and the pin to be used.
- 3 Wait for the input signal to stabilize.
- 4 Set the GO/_DONE bit. If this is a high-accuracy measurement, ADIE should be enabled for interrupts and then the PIC microcontroller put to “sleep.”
- 5 Poll GO/_DONE until it is reset (conversion done).
- 6 Read the result from ADRES and optionally ADRESH.

To read an analog voltage from the RAO pin of a PIC167C1 running a 4-MHz PIC microcontroller, the following code would be:

```
bsf      STATUS, RPO
movlw    0x002
movwf    ADCON1 ^ 0x080           ; AN1/AN0 are Analog Inputs
bcf      STATUS, RPO
movlw    0x041                   ; Start up the ADC
movwf    ADCON0
```



```

movlw    5
addlw    0x0FF                ; Delay 20 µsec for Holding
btfss    STATUS, Z            ; Capacitor to Stabilize
goto     $ - 2

bsf       ADCON0, GO          ; start the ADC conversion

btfsc    ADCON0, GO           ; Wait for the ADC Conversion
goto     $ - 1                ; to End

movf      ADRES, w             ; Read the ADC result

```

As you read the Microchip datasheets on the ADC, you will see that there are methods of implementing shorter, less accurate conversions. I do not recommend implementing these conversions because they decrease the accuracy of the ADC conversion but do not affect the biggest delay to doing the ADC conversion—the delay for the holding capacitor. This means that while the ADC can operate with a modest increase in speed, the total number of samples per second that can be made with the ADC cannot be increased substantially.

Parallel Slave Port (PSP)

One of the most interesting features of the 40-pin mid-range and PIC18 PIC microcontrollers is the parallel slave port (PSP) that is built into the PORTD and PORTE I/O pins. This feature allows the PIC microcontroller to act like an intelligent peripheral to any 8-bit data bus device.

The PSP is very easy to wire up with separate chip select and read/write pins for enabling the data transfer. The block diagram of the PSP is shown in Fig. 16.27.

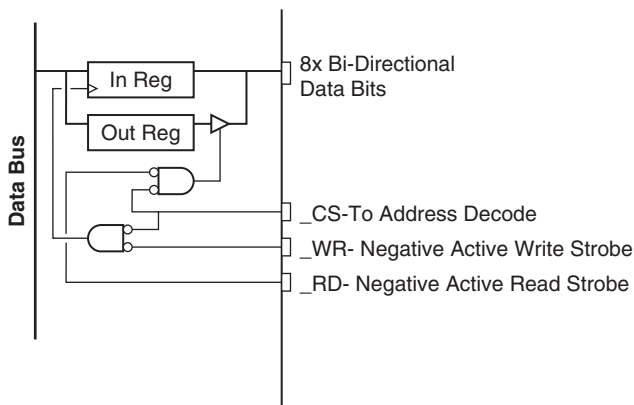


Figure 16.27 The hardware internal to the PIC microcontroller that implements the parallel slave port (PSP).

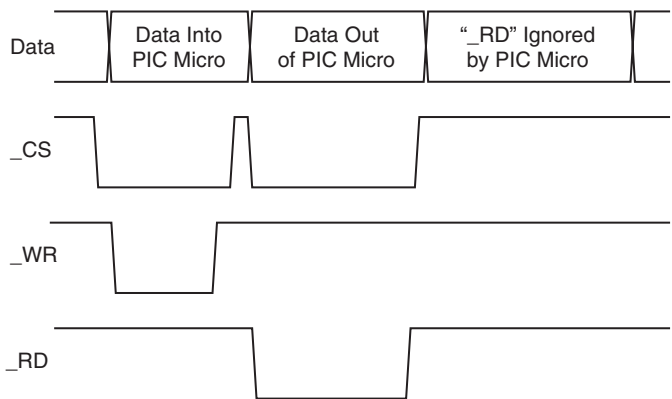


Figure 16.28 The waveform input from the host processor to the PSP-enabled PIC microcontroller.

The actual read/write I/O operations take place as you would expect for a typical I/O device connected to a microprocessor’s address/data control bus. A read and write operation waveform is shown in Fig. 16.28. The minimum access time is one clock (not instruction clock) cycle. For a PIC microcontroller running at 20 MHz, the minimum access time is 50 ns.

To enable the parallel slave port, the PSP mode bit of the TRISE register must be set. When this bit is set, port D becomes driven from the _CS, _RD, and _WR bits, which are RE2, RE1, and RE0, respectively. When the PSP mode bit is set, the values in PORTD, PORTE, TRISD, and TRISE are ignored.

PSP mode should be enabled the whole time the PIC microcontroller is active. Changing the pins between modes could cause bus operation problems with the controlling device and the device connected to PORTD and PORTE bits. In addition, the contents of PORTD and PORTE are unknown on return from PSP mode.

When PSP mode is enabled and _CS and _RD are active, PORTD drives out the contents of OUTREG. When OUTREG (which is at PORTD’s address) is written to, the OBF (*output buffer full*) bit of TRISE is set. This feature, along with the input data flags in TRISE, is not available in all devices. The PBF bit will become reset automatically when the byte in the OUTREG is read by the device driving the external parallel bus. When a byte is written into the parallel slave port (_CS and _WR are active), the value is saved in INREG until it is overwritten by a new value. If the optional status registers are available, the IBF bit is set when the INREG is written to and cleared when the byte in INREG is read. If the byte is not read before the next byte is written into INREG, the IBOV bit, which indicates the overwrite condition, is set.

In older PIC microcontrollers that have a PSP port, the IBF, OBF, and IBOV bits are not available in TRISE. While I recommend only using parallel slave port devices that have the IBF, OBF, and IBOV flags, there will be times when this is not possible. If you use a part that doesn’t have these bits, make sure that you create a method or protocol for sending data that ensures that no data byte transfers are missed. This can be done by sending the complement of the previous byte to the PIC microcontroller before the

next byte is sent and responding to reading the byte in OUTREG by writing its complement back into INREG.

With the parallel slave port working, all the other PIC microcontroller resources are available. This means that you can use ADCs (making sure that the PORTE bits are not set for analog input, which will cause problems with the parallel slave port), serial I/O, and other features that allow advanced I/O to and from the PIC microcontroller. In Chap. 21 I use the PSP to implement a custom serial interface to a PC's ISA bus.

In-Circuit Serial Programming (ICSP)

Microchip was one of the first manufacturers to produce microcontrollers that could be programmed after being wired into an application. This capability was first provided in mid-range PIC microcontrollers but since has become a feature in all new PIC microcontroller's devices. ICSP also can be used for parts that have not yet been soldered into a circuit, which minimizes the cost of creating a PIC microcontroller device programmer.

The ICSP connector for mid-range PIC microcontrollers is a 5-pin, 0.100" "spacing" IDC connector with the pin-out listed in Table 16.18. The ICSP connector can be added to any application circuit, and it allows even simple programmers to load PIC microcontrollers with application code without the need of ZIF sockets or SMT part adapters.

In Chap. 4, operation of the pins was described; I do want to bring to your attention the LVP pin, which allows the PIC microcontroller to be programmed without a high programming voltage. This feature can be useful if you have enough I/O pins available for your application because both the `_MCLR` and LVP pins will be unusable, whereas with high-voltage programming, LVP will act as a normal digital I/O pin, and `_MCLR` could be used as an input (if this feature is available in the PIC microcontroller part number).

TABLE 16.18 ICSP CONNECTOR PINOUT

PIN	FUNCTION
1	<code>_MCLR/Vpp</code> (programming voltage)
2	Vcc (normally +5 V)
3	Gnd
4	Data
5	Clock
6	LVP (low-voltage programming mode control)

This page intentionally left blank

PIC MCU INPUT AND OUTPUT DEVICE INTERFACING

If you were to look around your home, you would see a fairly limited variety of output devices that are used to help you to interface with the electronics in your home. Regardless of how technical they are, just about everyone has seen a light-emitting diode (LED) or pressed a button that produces a simple sound. These basic interfaces can be interfaced easily with PIC[®] microcontrollers and allow you to create your own applications. Many of these interfaces can be added to an application with very little circuitry or software effort, and they do a lot to enhance an application and make it easier to work with. Under the covers of the various products around your home are another set of very common devices with standard interfaces. These devices and their interfaces provide additional functionality to the microcontrollers built into the products, and being standard devices with standard interfaces, the amount of effort to add them to applications is actually quite minimal.

In this chapter I will be introducing you to a number of user interface and hardware function expansion devices that you can use in your own applications. Later in this book when I start introducing you to different applications, you will see these devices in action. The purpose of this chapter is to give you a more generic perspective on this aspect of PIC microcontroller application development and help you to understand the requirements of these devices and what impact they will have on the application development process.

LEDs

The most common form of output from a microcontroller is the LED. As an output device, it is cheap and easy to wire to a microcontroller. Generally, LEDs require anywhere from 5 to 25 mA of current to light (which is often within the output sink/source specification for most microcontrollers). What you have to remember, though, is that LEDs are diodes, which means current flows in one direction only. The typical circuit that I use to control an LED from a PIC microcontroller input-output (I/O) pin is shown in Fig. 17.1.

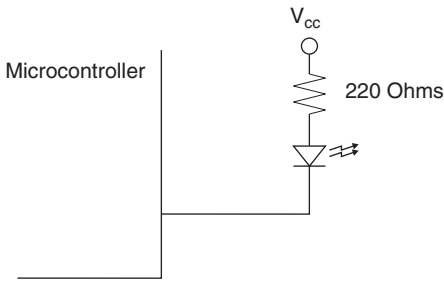


Figure 17.1 The basic circuit for driving an LED consists of a current-limiting resistor and a PIC microcontroller I/O pin capable of sinking enough current to turn on the LED.

With this circuit, the LED will light when the microcontroller's output pin is set to 0 (ground potential). When the pin is set to input or outputs, a 1, the LED will be turned off. This is a general convention for operation that came about owing to simple transistor logic and the Intel 8051, which could not source a significant amount of current; it could only sink enough current to turn on an LED. The popularity of this approach has led to a generation of engineers who design circuitry that turns on LEDs when the logic output is low. If you are new to electronics, this will definitely seem counterintuitive, but it is something that you are going to have to accept (like the fact that current is measured in the direction opposite to electron flow).

The 220- Ω resistor is used for current limiting and will prevent excessive current that can damage the microcontroller, LED, and power supply. The reason why I use a 220- Ω resistor is because when I was a student, I was told that I could never go wrong with it—this is true, but it is also equally true for 330 and 470 Ω in most applications. When you are designing your own applications, you should look at the LED's datasheet to understand its forward voltage as well as “on current” to properly calculate the best current-limiting resistor value.

To calculate the correct voltage, start with the formula:

$$V_{\text{applied}} = V_{\text{LED}} + I_{\text{LED}} \times R_{\text{current limiting}}$$

or rearranged to find $R_{\text{current limiting}}$, the formula becomes

$$R_{\text{current limiting}} = (V_{\text{applied}} - V_{\text{LED}})/I_{\text{LED}}$$

Using this formula for an LED that has a 1.5-V forward voltage and lights at 5 mA in a system that provides 3.3 V of power, the current-limiting resistor can be calculated:

$$R_{\text{current limiting}} = (3.3 \text{ V} - 1.5 \text{ V})/5 \text{ mA} = 360 \Omega$$

In this situation, I would use a 330- Ω current-limiting resistor.

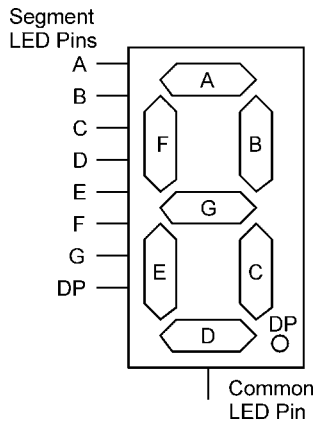


Figure 17.2 The seven-segment LED can have a common anode or cathode.

MULTISEGMENT LED DISPLAYS

Probably the easiest way to output numeric (both decimal and hex) data is via *seven-segment LED displays*. These displays were very popular in the seventies (if you're old enough, you're first digital watch probably had seven-segment LED displays) but have been largely replaced by LCDs.

However, seven segment LED displays (Fig. 17.2) are still useful devices that can be added to a circuit without a lot of software effort. By turning on specific LEDs (each of which lights up a “segment” in the display), the display can be used to output decimal numbers.

Each one of the LEDs in the display is given an identifier, and a single pin of the LED is brought out of the package. The other LED pins are connected together and wired to a common pin. This common LED pin is used to identify the type of seven-segment display (as either common cathode or common anode). Wiring one display to a microcontroller is quite easy—it is typically wired as seven [or eight if the decimal point (DP) is used] LEDs wired to individual pins. The most important piece of work you'll do when setting up seven-segment LED displays is matching and documenting the microcontroller bits to the LEDs. Spending a few moments at the start of a project will simplify wiring and debug of the display later.

The typical method of wiring multiple seven-segment LED displays together is to wire them all in parallel and then control the current flow through the common pin. Because the current is generally too high for a single microcontroller pin, a transistor is used to pass the current to the common power signal. This transistor selects which display is active. In Fig. 17.3, four common-cathode seven-segment displays are shown connected to a microcontroller.

In this circuit, the microcontroller will shift between the displays, showing each digit in a very short time slice. This is usually done in a timer interrupt handler. The basis for the interrupt handler's code is listed below:

Int:

- Save Context Registers
- Reset Timer and Interrupt

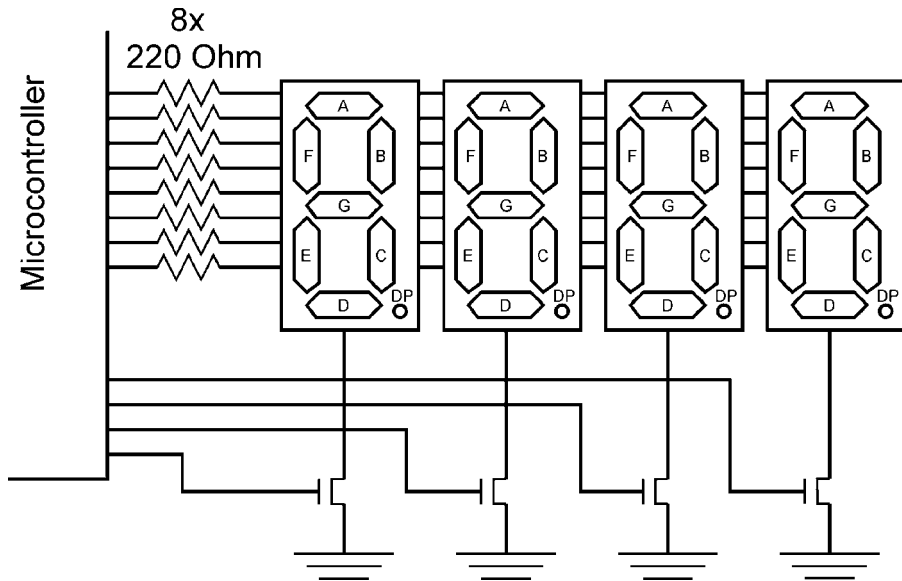


Figure 17.3 Multiple seven-segment LED displays can have common segment wiring with their common pins individually controlled, and the application scan through them faster than the human eye can perceive.

```

- LED_Display = 0                                ; Turn Off all the LEDs
- LED_Output = Display[ Cur ]
- Cur = (Cur + 1) mod #Displays                    ; Point to Next
Sequence Display
- LED_Display = 1 << Cur                            ; Display LED for
Current Display
- Restore Context Registers
- Return from Interrupt

```

This code will cycle through each of the digits (and displays), having current go through the transistors for each one. To avoid flicker, I generally run the code so that each digit is turned on/off at least 50 times per second. The more digits you have, the faster you have to cycle the interrupt handler (i.e., eight seven-segment displays must cycle at least 400 digits per second, which is twice as fast as four displays).

You may feel that assigning a microcontroller bit to select each display LED to be somewhat wasteful (at least I do). I have used high-current TTL demultiplexor (i.e., 74S138) outputs as the cathode path to ground (instead of discrete transistors). When the output is selected from the demultiplexor, it goes low, allowing current to flow through the LEDs of that display (and turning them on). This actually simplifies the wiring of the final application as well. The only issue is to make sure that the demultiplexor output can sink the maximum of 140 mA of current that will come through the common-cathode connection.

Along with seven-segment displays, there are 14- and 16-segment LED displays available that can be used to display alphanumeric characters (A–Z and 0–9). By following the same rules as used when wiring up a seven-segment display, you shouldn't

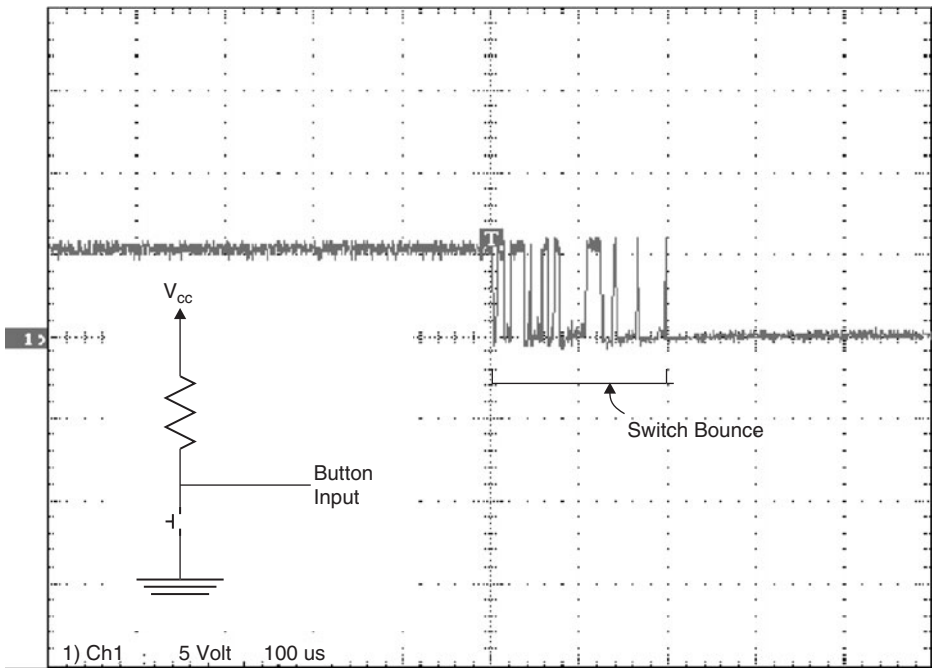


Figure 17.4 When a switch makes or breaks a connection, the contacts bounce against each other, possibly being interpreted as multiple button presses.

have any problems with wiring the display to a PIC microcontroller. In Chap. 21 I show how 7- and 16-segment LEDs can be used to display letters and numbers.

Switch Bounce

When a button is opened or closed, we perceive that as a “clean” operation that really looks like a step function. In reality, the contacts of a switch bounce when they make contact, resulting in the jagged signal shown in Fig. 17.4.

When this signal is passed to a PIC microcontroller, the microcontroller can recognize this as multiple button presses, which will cause the application software to act as if multiple, very fast button presses have taken place. To avoid this problem so that the switch press is treated like an idealized press, the step function I mentioned earlier and show in Fig. 17.5 is used. The signal will have to be *debounced*. There are two common methods used for debouncing button inputs.



Figure 17.5 The ideal waveform to come out of a switch is a clean transition like this one.

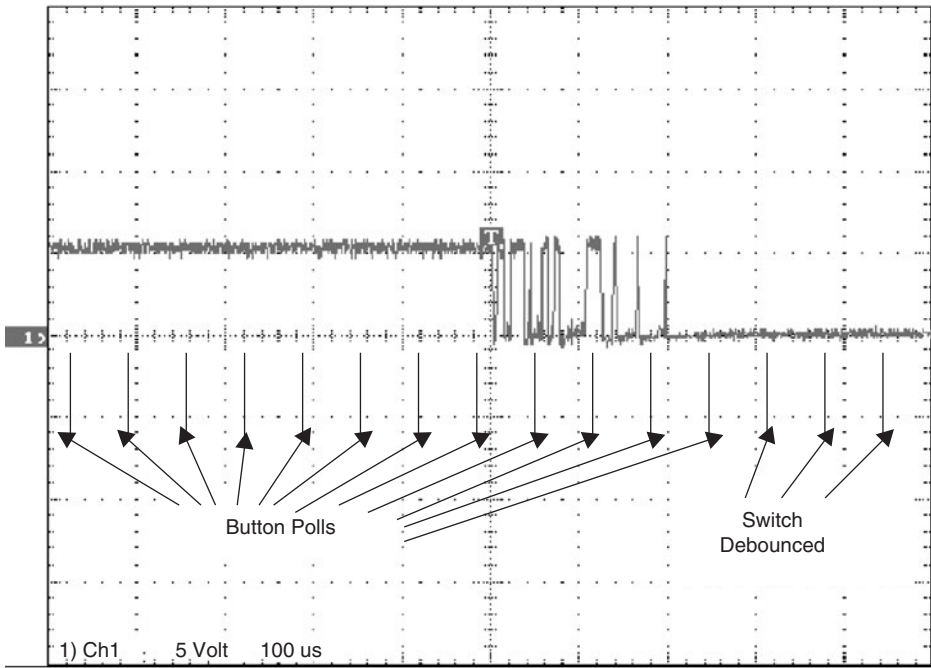


Figure 17.6 Polling the line continuously to wait for the bouncing to stop.

The first is to poll the switch line at short intervals until the switch line stays at the same level for an extended period of time. A button is normally considered debounced if it does not change state for 20 ms or more. By polling the line every 5 ms, this debouncing can be conceptualized quite easily, as shown in Fig. 17.6.

The advantage of this method is that it can be done in an interrupt handler, and the line can be scanned periodically with a flag set if the line is high and another flag if the line is low. For the indeterminate stage, neither bit would be set. This method is good for debouncing keyboard inputs.

The second method is to poll the line continually and wait for 20 ms to go by without the line changing state. The algorithm I use for this function is

```
do;
  while (Button == High);           // Poll Until Button is Pressed
  for (Dlay = 0; (Dlay < 20 ms) and (Button == Low); Dlay++);
until (Dlay >= 20 ms);
```

This code will wait for the button to be pressed and then poll it continuously until either 20 ms has passed or the switch has gone high again. If the switch goes high, the process is repeated until it is held low for 20 ms. This method is well suited to applications that don't have interrupts and only have one button input and no need for processing while polling the button. As restrictive as it sounds, there are many applications that fit these criteria.

This method also can be used with interrupt inputs along with TMR0 in the PIC microcontroller, which eliminates the restrictions I just detailed. The interrupt handler behaves

like the pseudocode below when one of the “port changes on interrupt” bits is used for the button input:

```
interrupt ButtonDebounce()      // Set Flags According to the
{                               // Debounced State of the Button

    if (T0IF == 1) {           // TMR0 Overflow, Button Debounced
        T0IF = 0; T0IE = 0;    // Reset and Turn off TMR0 Interrupts
        if (Button == High) {
            Pressed = 0; NotPressed = 1; // Set the State of the Button
        } else {
            Pressed = 1; NotPressed = 0;
        } // fi
    } else {                   // Port Change Interrupt
        Pressed = 0; NotPressed = 0; // Nothing True
        RBIF = 0;              // Reset the Interrupt
        TMR0 = 20msDlay;       // Reset Timer 0 for 20 msec
        T0IF = 0; T0IE = 1;    // Enable the Timer Interrupt
    } // fi
} // End ButtonDebounce
```

This code waits for the input pin to change state and then resets the two flags indicating the button state and starts TRM0 to request an interrupt after 20 ms. After a port-change interrupt, notice that I reset the button state flags to indicate to the mainline that the button is in a transition state and is not yet debounced. If TMR0 overflows, then the button is polled for its state, and the appropriate button state flags are set and reset.

The mainline code should poll the `Pressed` and `NotPressed` flags when it is waiting for a specific state. In Chap. 20 I will show how this method using TMR0 and interrupts can be implemented with or without interrupts.

If you don't want to use the software approaches, you can use a capacitor to filter the bouncing signal and pass it into a Schmidt trigger input. Schmidt trigger inputs have different thresholds depending on whether the signal is rising or falling. For rising edges, the trigger point is higher than for falling edges. Schmidt trigger inputs have the symbol put in the buffer shown in the circuit presented in Fig. 17.7.

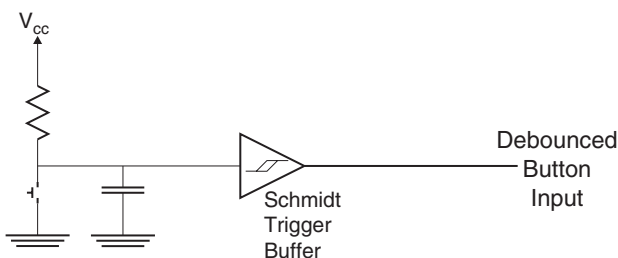


Figure 17.7 Bouncing into the PIC microcontroller can be eliminated by adding a filter to the button input line.

This method is fairly reliable but requires a Schmidt trigger gate in your circuit. There may be a Schmidt trigger input available in your PIC microcontroller, but check the datasheet to find out which states and peripheral hardware functions can take advantage of it.

One comment I want to make about debouncing switches is to choose buttons with a positive “click” when they are pressed and released. These have reduced bouncing and are a lot easier to work with than other switches that don’t have this feature. I have used a number of switches over the years that don’t have this click, and they can be a real problem in circuits with intermittent connections and unexpected bouncing that occurs while the button is pressed and held down.

Matrix Keypads

Switch matrix keyboards and keypads are really just an extension to the simple buttons of the preceding section with many of the same concerns and issues to watch out for. The big advantage that a matrix keyboard gives you is the ability to handle a large number of pins for a relatively small number of PIC microcontroller pins. The PIC microcontroller is well designed for simply implementing switch matrix keypads, which, like liquid-crystal displays (LCDs) explained in the next section, can add a lot to your application with a very small investment in hardware and software.

A switch matrix is simply a two-dimensional matrix of wires with switches at each vertex. The switch is used to interconnect rows and columns in the matrix, as can be seen in Fig. 17.8. This diagram may not look like the simple button, but it will become more familiar when I add pull-ups on the rows and switchable ground connections on the columns, as I show in Fig. 17.9.

In this case, by connecting one of the columns to ground, if a switch is closed, the pull-down on the row will connect the line to ground. When the row is polled by an I/O pin, a 0, or low voltage, will be returned instead of a 1 (which is what will be returned if the switch in the row that is connected to the ground is open).

As a rule of thumb, the number of PIC microcontroller I/O pins required to implement a switch matrix keypad is twice the square root of the number of keys rounded up to the nearest whole number. For example, if you want to implement a 29-button keypad

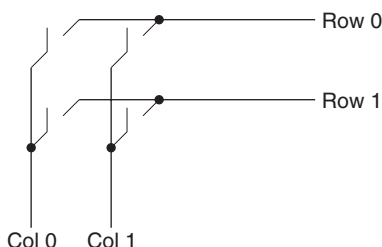


Figure 17.8 Multiple buttons can be wired together in a matrix to minimize the number of I/O pins required.

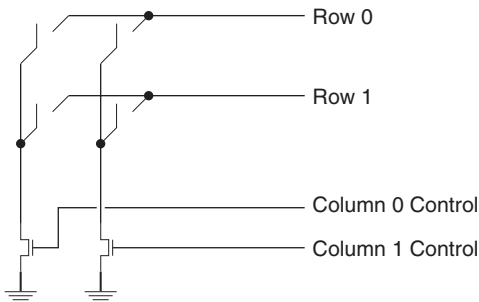


Figure 17.9 To read the state of the buttons, the columns are brought to ground, and the pulled-up rows are polled.

in your application, you would find the square root of 29 (which is 5.4) and round it up to the nearest whole number (6) and double it to discover that you should plan for 12 PIC I/O pins for the keypad. Again, this is a basic rule of thumb; I'm sure that you're thinking that the optimal number of PIC I/O pins would be 11. The error comes in because 29 is less than halfway between 25 and 36 (two evenly square rooted numbers); if it were 31, you would need 12 I/O pins to support the keypad.

As I said earlier, the PIC microcontroller is well suited to implementing switch matrix keyboards with PORTB's internal pull-ups and the ability of the I/O ports to simulate the open-drain pull-downs of the columns, as shown in Fig. 17.10. Normally, the pins

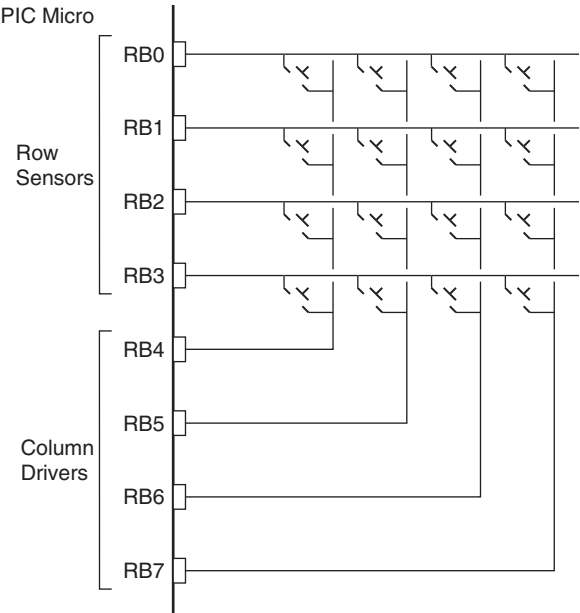


Figure 17.10 A 16-button keypad would be wired as shown here, requiring only eight PIC I/O pins.

connected to the columns are left in tristate (input) mode. When a column is being scanned, the column pin is output-enabled, driving a 0, and the four input bits are scanned to see if any are pulled low.

In this case, the keyboard can be scanned for any closed switches (buttons pressed) using the code

```

int KeyScan() // Scan the Keyboard and Return
when a
{ // key is pressed

int i = 0;
int key = -1;

while (key == -1) {

for (i = 0; (i < 4) & ((PORTB & 0x00F) == 0x0F0); i++);

switch (PORTB & 0x00F) { // Find Key that is Pressed
    case 0x00E: // Row 0
        key = i;
        break;
    case 0x00D: // Row1
    case 0x00C:
        key = 0x04 + i;
        break;
    case 0x00B: // Row2
    case 0x00A:
    case 0x009:
    case 0x008:
        key = 0x08 + i;
        break;
    else // Row3
        key = 0x0C + i;
        break;
} // hctiws
} // elihw

return key;

} // End KeyScan

```

The **KeyScan** function will only return when a key has been pressed. This routine will not allow keys to be debounced or for other code to execute while it is executing.

These issues can be resolved by putting the key scan into an interrupt handler that executes every 5 ms:

```
Interrupt KeyScan( )      // 5 msec Interval Keyboard Scan
{
```

```

int i = 0;
int key = -1

for (i = 0; (i < 4) & ((PORTB & 0x00F) == 0x00F)); i++);

if (PORTB & 0x00F) != 0x00F) { // Key Pressed
    switch (PORTB & 0x00F) { // Find Key that is Pressed
        case 0x00E: // Row 0
            key = i;
            break;
        case 0x00D: // Row1
        case 0x00C:
            key = 0x04 + i;
            break;
        case 0x00B: // Row2
        case 0x00A:
        case 0x009:
        case 0x008:
            key = 0x08 + i;
            break;
        else // Row3
            key = 0x0C+i;
            break;
    } // hctiws
    if (key == KeySave) {
        keycount = keycount + 1; // Increment Count
        // <-- Put in Auto Repeat Code Here
        if (keycount == 4)
            keyvalid = key; // Debounced Key
    } else
        keycount = 0; // No match - Start Again
    KeySave = key; // Save Current key for next
                // 5 ms
    } // fi // Interval
} // End KeySave

```

This interrupt handler will set `keyvalid` variable to the row/column combination of the key button (which is known as a *scan code*) when the same value comes up four times in a row. This time scan is the debounce routine for the keypad. If the value doesn't change for four intervals (20 ms in total), the key is determined to be debounced.

There are two things to notice about this code. The first is that in both routines I handle the row with the highest priority. If multiple buttons are pressed, then the one with the highest bit number will be the one that is returned to the user. The second point is that this code can have an autorepeat function added to it very easily. To do this, a secondary counter has to be first cleared and then incremented each time the `keycount`

variable is 4 or greater. To add an autorepeat key every second (200 intervals), the following code is added in the interrupt handler

```
if (keycount == 4) {
    keyrepeat = keyrepeat - 1; // Decrement the Key Auto Repeat Value
    if (keyrepeat == 0) {
        keyrepeat = 200;      // Restart the 1 second Auto
                               // Repeat count
        keycount = 3;         // Reset the counter
        keyvalid = key;       // Return the key
    }
} else // Reset the Auto Repeat Counter
    keyrepeat = 1;           // End Outputting the Value with Auto
                               // Repeat
```

LCDs

LCDs can add a lot to your application in terms of providing a useful interface for the user, debugging an application, or just giving it a professional look. The most common type of LCD controller is the Hitachi 44780, which provides a relatively simple interface between a processor and an LCD. Using this interface is often not attempted by new designers and programmers because it is difficult to find good documentation on the interface, initializing the interface can be a problem, and the displays themselves are expensive.

I have worked with Hitachi 44780-based LCDs for a while now, and I have to say that I don't believe any of these perceptions. LCDs can be added quite easily to an application and use as few as three digital output pins for control. As for cost, LCDs often can be pulled out of old devices or found in surplus stores for less than a dollar.

The purpose of this section is to give you a brief tutorial on how to interface with Hitachi 44780-based LCDs. I have tried to provide the all the data necessary for adding LCDs successfully to your application. In the book I use Hitachi 44780-based LCDs for a number of different projects.

The most common connector used for 44780-based LCDs is 14 pins in a row, with pin centers 0.100 in apart, with the pinout listed in Table 17.1.

As you probably would guess from this description, the interface is a parallel bus that allows simple and fast reading/writing of data to and from the LCD. This waveform to write an ASCII byte out to the LCD's screen is shown in Fig. 17.11. The ASCII code to be displayed is 8 bits long and is sent to the LCD either 4 or 8 bits at a time. If 4-bit mode is used, two nybbles of data (sent high 4 bits and then low 4 bits with an E clock pulse with each nybble) are sent to make up a full 8-bit transfer. The E clock is used to initiate the data transfer within the LCD.

Sending parallel data as either 4 or 8 bits is the primary mode of operation. While there are secondary considerations and modes, deciding how to send the data to the LCD is the

TABLE 17.1 HITACHI 44780 LCD DRIVER PINOUT

PIN	DESCRIPTION
1	Ground
2	Vcc
3	Contrast voltage
4	R/S—Instruction/register select
5	R/W—Read/write LCD registers
6	E—Clock
7–14	D0–D7—Data pins

most critical decision to be made for an LCD interface application. Eight-bit mode is best used when speed is required in an application and at least 10 I/O pins are available. Four-bit mode requires a minimum of 6 bits. To wire a microcontroller to an LCD in 4-bit mode, just the top 4 bits (DB4–7) are written to, with first the high nybble followed by the low nybble (which will be described in the next section).

The R/S bit is used to select whether data or an instruction is being transferred between the microcontroller and the LCD. If the bit is set, then the byte at the current LCD cursor position can be read or written. When the bit is reset, either an instruction is being sent to the LCD or the execution status of the last instruction is read back (whether or not it has completed).

The different instructions available for use with the 44780 are shown in Table 17.2, and the bit descriptions for the different commands are

Set cursor move direction:

ID—Increment the cursor after each byte written to display if set

S—Shift display when byte written to display

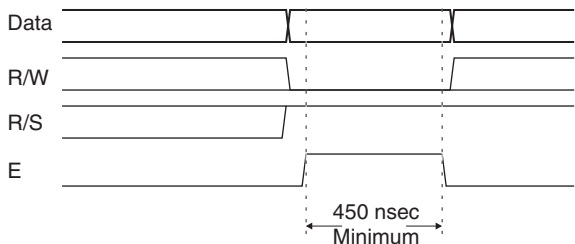


Figure 17.11 The Hitachi 44780-based LCD data write waveform.

TABLE 17.2 HITACHI 44780—BASED LCD CONTROLLER COMMANDS										
R/S	R/W	D7	D6	D5	D4	D3	D2	D1	D0	INSTRUCTION/DESCRIPTION
4	5	14	13	12	11	10	9	8	7	Pins
0	0	0	0	0	0	0	0	0	1	Clear display
0	0	0	0	0	0	0	0	1	*	Return cursor and LCD to home position
0	0	0	0	0	0	0	1	ID	S	Set cursor move direction
0	0	0	0	0	0	1	D	C	B	Enable display/cursor
0	0	0	0	0	1	SC	RL	*	*	Move cursor/shift display
0	0	0	0	1	DL	N	F	*	*	Reset/set interface length
0	0	0	1	A	A	A	A	A	A	Move cursor to CGRAM
0	0	1	A	A	A	A	A	A	A	Move cursor to display
0	1	BF	*	*	*	*	*	*	*	Poll the busy flag
1	0	H	H	H	H	H	H	H	H	Write hex character to the display at the current cursor position
1	1	H	H	H	H	H	H	H	H	Read hex character at the current cursor position on the display

*Not used/ignored. This bit can be either 1 or 0.

Enable display/cursor:

D—Turn display on(1)/off(0)

C—Turn cursor on(1)/off(0)

B—Cursor blink on(1)/off(0)

Move cursor/shift display:

SC—Display shift on(1)/off(0)

RL—Direction of shift right(1)/left(0)

Set interface length:

DL—Set data interface length 8(1)/4(0)

N—Number of display lines 1(0)/2(1)

F—Character font 5 × 10(1)/5 × 7(0)

Poll the busy flag:

BF—This bit is set while the LCD is processing

Move cursor to CGRAM/display:

A—Address

Read/write ASCII to the display:

H—Data

Reading data back is best used in applications that require data to be moved back and forth on the LCD (such as in applications that scroll data between lines). The busy flag can be polled to determine when the last instruction sent has completed processing.

For most applications, there really is no reason to read from the LCD. I usually tie R/W to ground and just wait the maximum amount of time for each instruction (4.1 ms for clearing the display or moving the cursor/display to the home position; 160 μ s for all other commands). As well as making my application software simpler, this also frees up a microcontroller pin for other uses. Different LCDs execute instructions at different rates, and to avoid problems later on (such as if the LCD is changed to a slower unit), I recommend just using the maximum delays listed here.

In terms of options, I have never seen a 5×10 LCD display. This means that the F bit in the `SetInterface` instruction always should be reset (equal to 0).

Before you can send commands or data to the LCD module, the module must be initialized. For 8-bit mode, this is done using the following series of operations:

- 1** Wait more than 15 ms after power is applied.
- 2** Write 0x030 to LCD and wait 5 ms for the instruction to complete.
- 3** Write 0x030 to LCD and wait 160 μ s for the instruction to complete.
- 4** Write 0x030 *again* to LCD and wait 160 μ s or poll the busy flag.
- 5** Set the operating characteristics of the LCD:
 - Write `SetInterface` length.
 - Write 0x010 to turn off the display.
 - Write 0x001 to clear the display.
 - Write “set cursor move direction” to set cursor behavior bits.
 - Write “enable display/cursor” to enable display and optional cursor.

In describing how the LCD should be initialized in 4-bit mode, I will specify writing to the LCD in terms of nybbles. This is so because initially, just single nybbles are sent (and not two, which make up a byte and a full instruction). As I mentioned earlier, when a byte is sent, the high nybble is sent before the low nybble, and the E pin is toggled each time 4 bits are sent to the LCD. To initialize in 4-bit mode, the following sequence of commands is sent:

- 1** Wait more than 15 ms after power is applied.
- 2** Write 0x03 to LCD and wait 5 ms for the instruction to complete.
- 3** Write 0x03 to LCD and wait 160 μ s for instruction to complete.
- 4** Write 0x03 *again* to LCD and wait 160 μ s (or poll the busy flag).
- 5** Set the operating characteristics of the LCD:
 - Write 0x02 to the LCD to enable 4-bit mode.

All following instruction/data writes require two-nybble writes:

- Write `SetInterface` length.
- Write `0x01/0x00` to turn off the display.
- Write `0x00/0x01` to clear the display.
- Write “set cursor move direction” to set cursor behavior bits
- Write “enable display/cursor” to enable display and optional cursor.

Once the initialization is complete, the LCD can be written to with data or instructions as required. Each character to display is written like the control bytes, except that the R/S line is set. During initialization, by setting the S/C bit during the “move cursor/shift display” command, after each character is sent to the LCD, the cursor built into the LCD will increment to the next position (either right or left). Normally, the S/C bit is set (equal to 1) along with the R/L bit in the “move cursor/shift display” command for characters to be written from left to right (as with a teletype video display).

One area of confusion is how to move to different locations on the display and, as a follow-on, how to move to different lines on an LCD display. Table 17.3 lists how different LCD displays that use a single 44780 can be set up with the addresses for specific

TABLE 17.3 HITACHI 44780—BASED LCD TYPES AND CHARACTER LOCATIONS

LCD	TOP LEFT	NINTH	SECOND LINE	THIRD LINE	FOURTH LINE	COMMENTS
8 × 1	0	N/A	N/A	N/A	N/A	Note 1
16 × 1	0	0x040	N/A	N/A	N/A	Note 1
16 × 1	0	8	N/A	N/A	N/A	
8 × 2	0	N/A	0x040	N/A	N/A	Note 1
10 × 2	0	0x008	0x040	N/A	N/A	Note 3
16 × 2	0	0x008	0x040	N/A	N/A	Note 2
20 × 2	0	0x008	0x040	N/A	N/A	Note 2
24 × 2	0	0x008	0x040	N/A	N/A	Note 2
30 × 2	0	0x008	0x040	N/A	N/A	Note 2
32 × 2	0	0x008	0x040	N/A	N/A	Note 2
40 × 2	0	0x008	0x040	N/A	N/A	Note 2
16 × 4	0	0x008	0x040	0x020	0x040	Note 2
20 × 4	0	0x008	0x040	0x020	0x040	Note 2
40 × 4	0	N/A	N/A	N/A	N/A	Note 4

Notes:

¹Single 44780/no support chip.

²44780 with support chip.

³44780 with support chip. This is quite rare.

⁴Two 44780s with support chips. Addressing is device-specific.

character locations. The LCDs listed are the most popular arrangements available, and the layout is given as number of columns by number of lines.

The “ninth character” is the position of the ninth character on the first line. Most LCD displays have a 44780 and support chip to control the operation of the LCD. The 44780 is responsible for the external interface and provides sufficient control lines for 16 characters on the LCD. The support chip enhances the I/O of the 44780 to support up to 128 characters on an LCD in two lines of eight. From the table, it should be noted that the first two entries (8×1 and 16×1) only have the 44780 and not the support chip. This is why the ninth character in the 16×1 does not appear at address 8 and shows up at the address that is common for a two-line LCD.

I’ve included the 40-character by 4-line (40×4) LCD because it is quite common. Normally, the LCD is wired as two 40×2 displays. The actual connector is normally 16 bits wide, with all 14 connections of the 44780 in common, except for the E (strobe) pins. The individual E strobe lines are used to select between the areas of the display used by the two devices. The actual pinouts and character addresses for this type of display can vary among manufacturers and display part numbers. Note that when using any kind of multiple-44780 LCD display, you probably should display the cursor of only one of the 44780s at a time.

Cursors for the 44780 can be turned on as a simple underscore at any time using the “enable display/cursor” LCD instruction and setting the C bit. I don’t recommend using the B (block mode) bit because this causes a flashing full-character square to be displayed, and it really isn’t that attractive.

The LCD can be thought of as a teletype display because in normal operation, after a character has been sent to the LCD, the internal cursor is moved one character to the right. The “clear display” and “return cursor and LCD to home position” instructions are used to reset the cursor’s position to the top right character on the display. An example of moving the cursor is shown in Fig. 17.12.

To move the cursor, the “move cursor to display” instruction is used. For this instruction, bit 7 of the instruction byte is set, with the remaining 7 bits used as the address

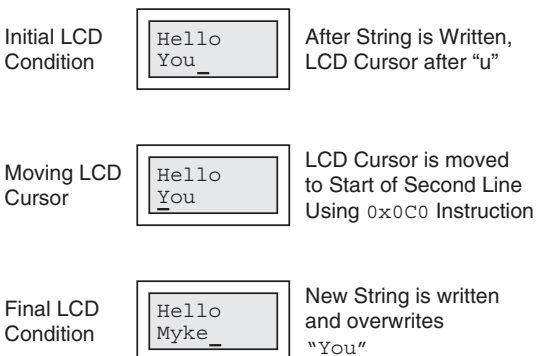


Figure 17.12 Text appears at the current LCD cursor position; to overwrite characters, the cursor can be moved.

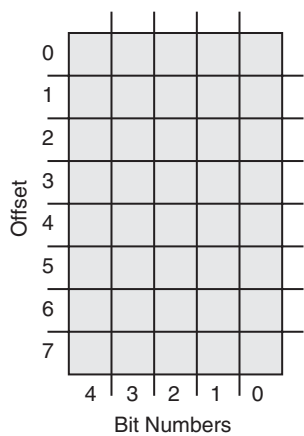


Figure 17.13 The custom LCD characters have to fit in a 7 × 5 pixel box.

of the character on the LCD to which the cursor is to move. These 7 bits provide 128 addresses, which matches the maximum number of LCD character addresses available. Table 17.3 should be used to determine the address of a character offset on a particular line of an LCD display.

The character set available in the 44780 is basically ASCII. I say *basically* because some characters do not follow the ASCII convention fully (probably the most significant difference is 0x05B, or \, is not available). The ASCII control characters (0x008 to 0x01F) do not respond as control characters and may display funny (Japanese) characters. Eight programmable characters are available and use codes 0x000 to 0x007. They are programmed by pointing the LCD’s cursor to the character generator RAM (CGRAM) area at eight times the character address. The next 8 bytes written to the RAM are the line information of the programmable character starting from the top. The “character box” representation is shown in Fig. 17.13.

I like to represent this as eight squares by five, as is shown in the diagram to the right. Earlier I noted that most displays were 7 pixels by 5 for each character, so the extra row may be confusing. Each LCD character is actually 8 pixels high, with the bottom row normally used for the underscore cursor. The bottom row can be used for graphic characters, although if you are going to use a visible underscore cursor and have it at the character, I recommend that you don’t use it (i.e., set the line to 0x000).

Using this box, you can draw in the pixels that define your special character and then use the bits to determine what the actual data codes are. When I do this, I normally use a piece of graph paper and then write hex codes for each line, as I show in Fig. 17.14, to produce a diagram of a simple “smiley face.”

For some “animate” applications, I use character rotation for the animations. This means that instead of changing the character each time the character moves, I simply display a different character. Doing this means that only 2 bytes (moving the cursor to the character and the new character to display) have to be sent to the LCD. If animation were accomplished by redefining the characters, then 10 characters would have to be sent to

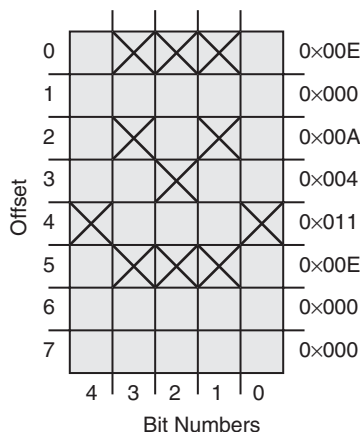


Figure 17.14 A custom character definition.

the LCD (one to move into the CGRAM space, the eight defining characters, and an instruction returning to display RAM). If multiple characters are going to be used or more than eight pictures for the animation, then you will have to rewrite the character each time.

The user-defined character line information is saved in the LCD's CGRAM area. This 64-byte space of memory is accessed using the "move cursor into CGRAM" instruction in a similar manner to that of moving the cursor to a specific address in the memory with one important difference. This difference is that each character starts at eight times its character value. This means that user-definable character 0 has its data starting at address 0 of the CGRAM, character 1 starts at address 8, character 2 starts at address 0x010 (16), and so on. To get a specific line within the user-definable character, its offset from the top (the top line has an offset of 0) is added to the starting address. In most applications, characters are written to all at one time with character 0 first. In this case, the instruction 0x040 is written to the LCD, followed by all the user-defined characters.

The last aspect of the LCD to discuss is how to specify a contrast voltage to the display. I typically use a potentiometer wired as a voltage divider (Fig. 17.15). This will provide an easily variable voltage between ground and V_{CC} that will be used to specify the contrast (or darkness) of the characters on the LCD screen. You may find that different LCDs work differently, with lower voltages providing darker characters in some and higher voltages doing the same thing in others.

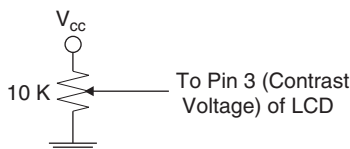


Figure 17.15 Contrast control can be provided by a pot wired as a voltage divider.

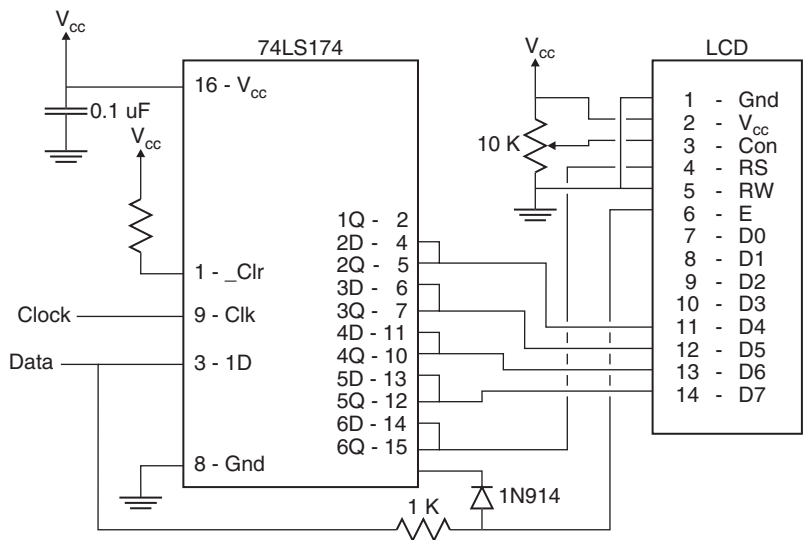


Figure 17.16 With this circuit, you can control an LCD with just two wires.

TWO-WIRE LCD CONTROL

There are a number of different ways to wire up an LCD. Earlier I noted that the 44780 could interface with 4 or 8 bits. To simplify the demands of microcontrollers, a shift register is often used to reduce the number of I/O pins to three. This can be further reduced by using the circuit shown in Fig. 17.16, in which the serial data is combined with the contents of the shift register to produce the E strobe at the appropriate interval. This circuit ANDs (using the 1-kΩ resistor and IN914 diode) the output of the sixth D flip-flop of the 74LS174 and the data bit from the device writing to the LCD to form the E strobe. This method requires one less pin than the three-wire interface and a few more instructions of code.

I normally use a 74LS174 wired as a shift register (as shown in the schematic diagram) instead of a serial-in/parallel-out shift register. This circuit should work without any problems with a dedicated serial-in/parallel-out shift register chip, but the timings/clock polarities may be different. When the 74LS174 is used, note that the data is latched on the rising edge (from logic low to high) of the clock signal. Figure 17.17 is a timing diagram for the two-wire interface and shows the 74LS174 being cleared, loaded, and then the E strobe when the data is valid and 6Q and incoming data are high.

In the diagram to the right I have shown how the shift register is written to for this circuit to work. Before data can be written to it, loading every latch with zeros clears the shift register. Next, a 1 (to provide the E gate) is written, followed by the R/S bit and the 4 data bits. Once the latch is loaded correctly, the data line is pulsed to strobe the E bit. The biggest difference between the three-wire and the two-wire

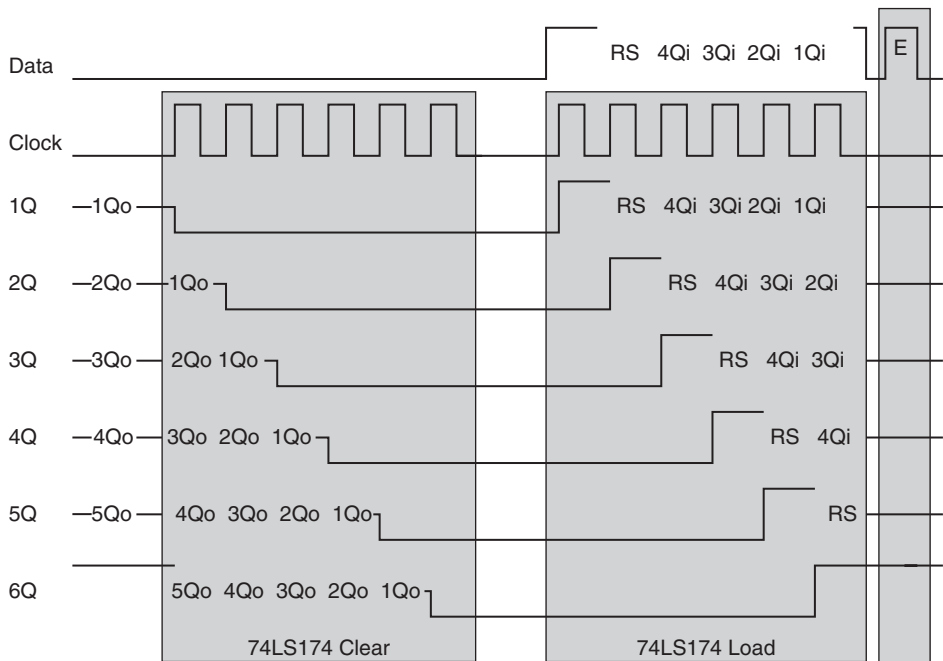


Figure 17.17 The synchronous serial waveform of the data coming in showing the bits propagating through the shift register.

interface is that the shift register has to be cleared before it can be loaded, and the two-wire operation requires more than twice the number of clock cycles to load 4 bits into the LCD.

I've used this circuit with the PIC microcontroller, BASIC Stamp, 8051, and AVR, and it really makes the wiring of an LCD to a microcontroller very simple. A significant advantage of using a shift register, as in the two circuits shown here, is the lack of timing sensitivity that will be encountered. The biggest issue to watch for is to make sure that the E strobe's timing is within specification (i.e., greater than 450 ns); the shift register loads can be interrupted without affecting the actual write. This circuit will not work with open-drain-only outputs (something that catches up many people).

One note about the LCD's E strobe is that in some documentation it is specified as high level active, whereas in others it is specified as falling-edge-active. It *seems* to be falling-edge-active, which is why the two-wire LCD interface presented below works even if the line ends up being high at the end of data being shifted in. If the falling edge is used (as in the two-wire interface), then make sure that before the E line is output on 0, there is at least a 450-ns delay with no lines changing state.

The following C routine could be used to write to the two-wire LCD interface:

```
LCDNybble(char Nybble, char RS)
{
    int i;
```

```
Data = 0; // Clear the '174
for (i = 0; i < 6; i++) { // Repeat for six bits
    Clock = 1; Clock = 0; // Write the "0"s into the '174
} // rof

Data = 1; // Output the "AND" Value
Clock = 1; Clock = 0;

Data = RS; // Output the RS Bit Value
Clock = 1; Clock = 0;

for (i = 0; i < 4; i++) { // Output the Nybble
    if ((Nybble & 0x008) != 0) // Output the High Order Bit
        Data = 1;
    else
        Data = 0;
    Clock = 1; Clock = 0; // Strobe the Clock
    Nybble = Nybble << 1; // Shift up Nybble for Next Byte
} // rof

Clock = 1; Clock = 0; // Toggle the "E" Clock Bit

} // End LCDNybble
```

Analog I/O

In the following sections I want to introduce you to some of the practical aspects of working with analog data with the PIC microcontroller. This includes position sensing using potentiometers. At first glance, you may think that an ADC-equipped microcontroller is required for this operation, but there are a number of ways of doing this with strictly digital inputs. Along with discussing how it can be done with ADCless microcontrollers, I'll also show how the IBM PC carries out the function (it doesn't use an ADC either).

For analog output, I will focus on the theory and operation behind PWM analog control signals. This method of control is very popular and is a relatively simple way to provide analog control of a device. It also can be used for communication of analog values between devices without needing any type of communication protocol. The PIC microcontroller has some built-in hardware that makes the implementation of PWM input and output quite easy to do.

While I discuss audio I/O, I want to make it clear that audio input and output capabilities cannot be provided in the PIC microcontroller without significant front-end signal processing and filtering. Output from the PIC microcontroller is limited to simple beeps without special hardware.



Figure 17.18 A potentiometer wired as a voltage divider can be read using the PIC microcontroller's ADC input.

READING POTENTIOMETERS

One of the more useful human input devices is the dial. Rather than relying on some kind of digital data such as a button or character string, the dial allows users a freer range of inputs, as well as positional feedback information, in a mechanical device. For most people, reading a potentiometer value requires setting the potentiometer as a voltage divider and reading the voltage between the two extremes at the wiper, as shown in Fig. 17.18. However, there is a very elegant way of reading a potentiometer's position using the digital input of a PIC microcontroller, as I will show in this section.

Note that I consider the measurement to be of the potentiometer's position, not its resistance. This is an important semantic point; as far as using a potentiometer as an input device is concerned, I do not care what the actual resistance is of its position, just what the position is. The method of reading a potentiometer's position using a digital I/O pin that I am going to show you depends very much on the parts used and will vary significantly between implementations.

The method of reading a potentiometer uses the characteristics of a charged capacitor discharging through a resistor. If the charge is constant in the capacitor, then the time to discharge varies according to the exponential curve shown in Fig. 17.19.

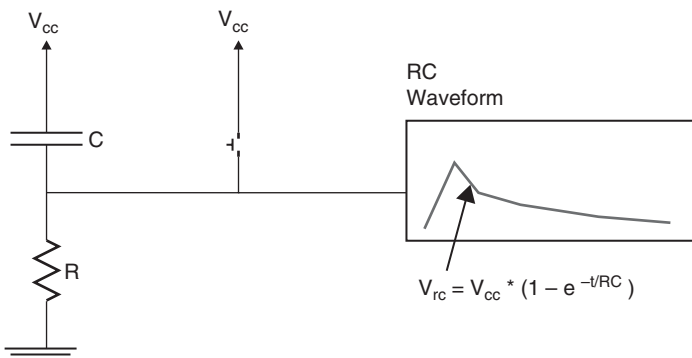


Figure 17.19 A capacitor's charge-decay response can be modified by a potentiometer, and the time taken can be measured by a PIC microcontroller.

The charge in a capacitor is proportional to its voltage. If a constant voltage (i.e., from a PIC microcontroller I/O pin) can be applied to a capacitor, then its charge will be constant. This means that in the voltage-discharge curve shown in Fig. 17.19, if the initial voltage is known, along with the capacitance and resistance, then the voltage at any point in time can be predicted.

The equation in the figure, i.e.,

$$V(t) = V_{\text{start}}(1 - e^{-t/RC})$$

can be reworked to find R if V , V_{start} , t , and C are known:

$$R = -t/C * \ln((V_{\text{start}} - V)/V_{\text{start}})$$

Rather than calculate the value, though, you can make the approximation of 2 ms for a resistance of 10 k Ω and a capacitance of 0.1 μF with a PIC microcontroller that has a high-to-low threshold of 1.5 V. To measure the resistance in a PIC microcontroller, I use the circuit shown in Fig. 17.20. In this circuit, the PIC microcontroller's I/O pin outputs a high that charges the capacitor (with some leakage through the potentiometer resistor).

After the capacitor is charged, the pin is changed to input, and the charge in the capacitor draws through the resistor with a voltage determined by the $V(t)$ formula. When the pin first changes state, the voltage across the resistor will be greater than the threshold for some period of time. When the voltage across the potentiometer falls below the voltage threshold, the input pin value returned to the software will be 0. If the time required for voltage across the pin to go from a 1 to a 0 is recorded, it will be proportional to the resistance between the potentiometer's wiper and the capacitor.

The pseudocode for carrying out the potentiometer read is

```
int ReadPot()                                // Return the
Potentiometer's Position
{
    int i;
```

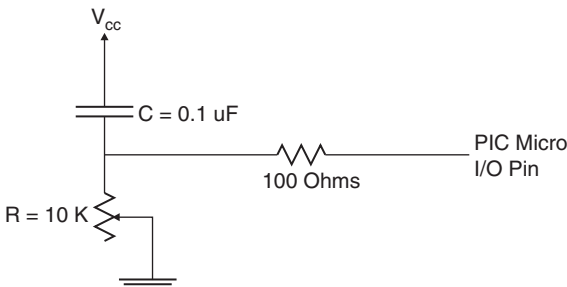


Figure 17.20 A circuit that can measure the resistance of a potentiometer by the time it takes for a capacitor to discharge through it.

```

pin = output; pin = 1;           // Charge the Capacitor
for (i = 0; i < charge; i++);
pin = input;                     // Let the Capacitor Discharge
for ( i = 0; pin == 1; i++);
return I ;

} // End ReadPot

```

The PIC microcontroller assembly-language code for implementing this potentiometer read is not much more complex than this pseudocode. Later in this book I will give you some examples of how potentiometer reads are actually accomplished.

The 100- Ω resistor between the PIC microcontroller pin and the RC network is used to prevent any short circuits to ground if the potentiometer is set so that there is no resistance in the circuit at all. This method of reading a potentiometer's position is very reliable, but it is not very accurate, nor is it particularly fast. When setting this up for the first time in a specific circuit, you will have to experiment to find the actual range it will display. This is due to part variances (including the PIC microcontroller) and the power-supply characteristics. For these reasons, I do not recommend using the potentiometer/ capacitor circuit in any products. Tuning the values returned will be much more expensive than the cost of a PIC microcontroller with a built-in ADC.

PWM I/O

The PIC microcontroller, like most other digital devices, does not handle analog voltages very well. This is especially true for situations where high-current voltages are involved. The best way to handle analog voltages is to use a string of varying-wide pulses to indicate the actual voltage level. This string of pulses is known as a PWM analog signal and can be used to pass analog data from a digital device, control dc devices, or even output an analog voltage. In this section I want to discuss PWM signals and how they can be used with the PIC microcontroller. In the discussion of TMR1 and TMR2 earlier in this book, I presented how PWM signals were implemented and read using the CCP built-in hardware of the PIC microcontroller. For this section I will show how PWM signals can be used for I/O in PIC microcontrollers that do not have the CCP module built in.

A PWM signal is a repeating signal that is on for a set period of time that is proportional to the voltage that is being output. A PWM signal is shown in Fig. 17.21. I call the on time the *pulse width* in the figure, and the *duty cycle* is the percentage of the on time relative to the PWM signal's *period*.

To output a PWM signal, the following code could be implemented, although there is no way of changing the values while it is running (unless you were to include an interrupt handler):

```

Period = PWMPeriod;           // Initialize the Output
On = PWMOn;                   // Parameters

```

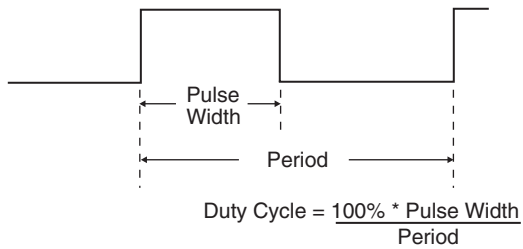


Figure 17.21 The basic PWM waveform with its features labeled.

```
while (1 == 1) {

    PWM = ON;                                // Start the Pulse
    for (i = 0; i < On; i++ );                // Output ON for "On" Period of
                                                // Time
    PWM = off;                                // Turn off the Pulse
    For ( ; i < PWMPeriod; i++ )              ; // Output off for the rest of the
                                                // PWM Period
} // end while
```

To avoid the problem of all the resources being devoted to the PWM code, I would recommend using the TMR0 interrupt to create the PWM timing:

```
Interrupt PWMOutput() // When Timer Overflows, Toggle "On" and "Off"
{                     // and Reset Timer to correct delay for Value

    if (PWM == ON) {  // If PWM is ON, Turn it off and Set Timer
        PWM = off;    // Value
        TMR0 = PWMPeriod - PWMOn;
    } else {          // If PWM is off, Turn it ON and Set Timer
        PWM = ON;     // Value
        TMR0 = PWMOn;
    } // fi

    INTCON.TOIF = 0;  // Reset Interrupts

} // End PWMOutput TMR0 Interrupt Handler
```

This code is quite easy to port into PIC microcontroller assembly language. For example, if the PWM period were 1 ms (executing in a 4-MHz PIC microcontroller), a divide by four prescaler value could be used with the timer, and the interrupt handler assembly-language code would be

```
org      4
Int:     ; Interrupt Handler
```

```

movwf  _w          ; Save Context Registers
movf   STATUS, w   ; - Assume TMR0 is the only enabled Interrupt
movwf  _status
btfsc  PWM         ; Is PWM O/P Currently High or Low?
goto   PWM_ON
nop                    ; Low – Nop to Match Cycles with High

bsf    PWM         ; Output the Start of the Pulse

movlw  6 + 6       ; Get the PWM On Period
subwf  PWMOn, w    ; Add to PWM to Get Correct Period for
; Interrupt Handler Delay and Missed cycles
; in maximum 1024  $\mu$ sec Cycles
goto   PWM_Done

PWM_ON:              ; PWM is On – Turn it Off

bcf    PWM         ; Output the “Low” of the PWM Cycle

movf   PWMOn, w    ; Calculate the “Off” Period
sublw  6 + 6       ; Subtract from the Period for the Interrupt
; Handler Delay and Missed cycles in maximum
; 1024  $\mu$ sec Cycles
goto   PWM_Done

PWM_Done:            ; Have Finished Changing the PWM Value
sublw  0           ; Get the Value to Load into the Timer
movwf  TMR0

bcf    INTCON, T0IF ; Reset the Interrupt Handler

movf   _status, w  ; Restore the Context Registers
movwf  STATUS
swapf  _w, f
swapf  _w, w

retfie

```

In this code, TMR0 is loaded in such a way that the PWM period is always 1 ms (or a count of 250 “ticks” with the prescaler value of 4). To get the value added and subtracted from the total, I first took the difference between the number of ticks to get 1 ms (250) and the full timer range (256). Next, I counted the total number of instruction cycles of the interrupt handler (which is 23) and divided it by 4 and added the result to the 1-ms difference. The operation probably seems confusing because I was able to optimize the time for the PWM signal off:

Time Off = Period – ON

to

```

movf    ON, w
sublw   6 + 6
sublw   0
movwf   TMR0

```

I realize that this doesn't make any sense the first time you look at it, and I will go through it by showing how it works. Using the original equation, you should note that this calculates the number of cycles to be delayed by TMR0, but the actual value to be loaded into TMR0 is calculated as

```

TMR0 Delay Value = 0x100 - (Time Off)
                  = 0x100 - (Period - ON)
                  = 0x100 - (256 - 250 + Interrupt Execution- ON)
                  = 0x100 - (6 + 6 - ON)
                  = 0x100 - (12 - ON)
                  = 0x100 - 12 + ON
                  = 0xF4 + ON

```

Going back to the three instructions that load TMR0, you can show that they execute as

```

movf    ON, w          ; w = ON
sublw   6 + 6          ; w = 6 + 6 - w
                        ;   = 12 - ON
                        ;   = 12 + 0xFF ^ ON + 1
sublw   0              ;   = 13 + 0xFF ^ ON
                        ; w = 0 - w
                        ;   = 0 - (13 + 0xFF ^ ON)
                        ;   = 0 + 0xFF ^ (13 + 0xFF ^ ON) + 1
                        ;   = 0xFF ^ 13 + 0xFF ^ 0xFF ^ ON + 1
                        ;   = 0xFF ^ 13 + ON + 1
                        ;   = 0xF4 + ON

```

which is (surprisingly enough) the same result that was found with the “TMR0 delay value” equation earlier. The formula in itself is not that impressive—except that it dovetails very well with the PWM on half of the code. This optimization probably belongs in another chapter on optimization, but to be honest with you, I came up with it using nothing but trial and error along with the “feeling” that this kind of optimization were possible. This is an example of what I mean when I say that you should look for opportunities when processing data in the PIC microcontroller. More often than not, you will come up with something like these few instructions that are very efficient and integrate different cases.

Note that in this code, the PWM signal will never fully be on (a high dc voltage) or fully off (a ground level dc voltage). This is so because when the routine enters the sub-routine handler, it changes the output regardless of whether or not it is required for the length of the interrupt handler. In actuality, if you time it out, you will see that the 23 instruction cycles that the interrupt handler takes between changing the value works out to a 2.4 percent loss of full-on and full-off. This should not be significant in most

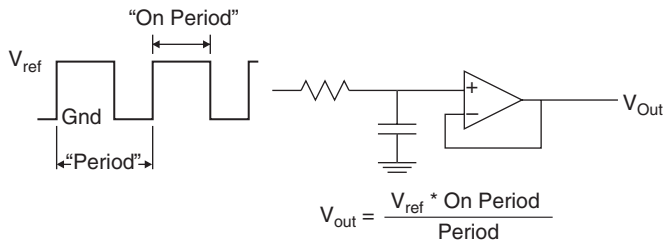


Figure 17.22 The PWM output can be converted to an analog voltage using an RC low-pass filter.

applications and will serve as a “heartbeat” to let the receiver know that the PIC microcontroller is still functioning, even though the output is “stuck” at an extreme.

In the preceding example I expended quite a bit of energy in making sure that the period remains the same regardless of the on time. This was done to make sure that the changes in the duty cycle remained proportional to the changes in the on period. This is important if the PWM output is going to be passed through a low-pass filter, as shown in Fig.17.22, to output an analog voltage.

In many applications where a PWM signal is communicating with another digital device, this effort to ensure that the period is constant is not required. In these cases, a timer is used to time the on period. This can be shown as the pseudocode

```
Int TimeOn()                                // Time the Width of an incoming Pulse
{
    int i = 0;

    while (PWMIP == off);                    // Wait for the Pulse to Start

    for ( ; PWMIP == ON; i++ );              // Time the Pulse Width

    return i;                                // Return the Pulse Width
} // end TimeOn
```

with the actual PIC microcontroller assembly-language code being quite simple but dependent on the maximum pulse width value being timed—very long pulses will require large counters or delays in between the PWM input (PWMIP in TimeOn above) poll.

Passing analog data back and forth between digital devices in any format is not going to be accurate owing to the errors in digitizing the value and restoring it. This is especially true for PWM signals, which can have very large errors owing to the sender and receiver not being properly synched up and the receiver not starting to poll at the correct time interval. In fact, the measured value could have an error of upward of 10 percent from the actual value. This loss of data accuracy means that analog signals should not

be used for data transfers. However, as I will show later in this book, PWM signals are an excellent way to control analog devices such as LEDs and motors.

When using a PWM signal for driving an analog device, it is important to make sure that the frequency is outside the range of what a human can perceive. As noted in the LED section earlier, this frequency is 30 Hz or more. For motors and other devices that may have an audible whine, however, the PWM signal used should have a frequency either below 50 Hz or 20 kHz or more to ensure that the signal does not bother the user (although it may cause problems with the user's dogs). The lower PWM frequency is probably surprising, but there are a lot of small motors that work very well with the low PWM frequency.

The problem with the higher frequencies is that the granularity of the PWM signal decreases. This is due to the inability of the PIC microcontroller (or whatever digital device is driving the PWM output) to change the output in relatively small time increments from on to off relative to the size of the PWM signal's period. In the preceding example code, four instruction cycles (of 1 μ s each) are the lowest level of granularity for the PWM signal that results in about 250 unique output values. If the PWM signal's period was decreased to 100 μ s from 1 ms for a 10-kHz frequency, the same code would have only 25 or so unique values that could be output. In this case, to retain the original code's granularity, the PIC microcontroller would have to be sped up 10 times (not possible for most applications) or another way of implementing the PWM would have to be found.

Audio Output

When I discuss the PIC microcontroller's processing capabilities with regard to audio, I tend to be quite disparaging. The reason for this is the lack of hardware multipliers in low-end and mid-range PIC microcontrollers and the inability of all the devices to natively handle greater than 8 bits in a floating-point format. The PIC microcontroller processor has been optimized for responding to digital inputs and cannot implement the real-time processing routines needed for complex analog I/O.

Despite this, you can implement some surprisingly sophisticated audio output that goes beyond simple beeps and boops using a circuit such as the one shown in Fig. 17.23.

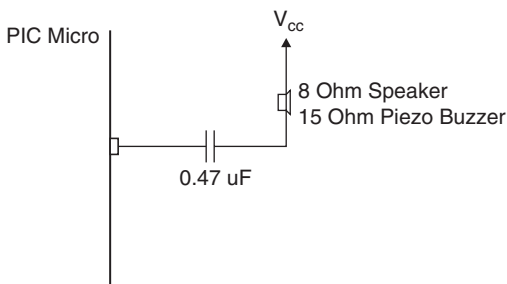


Figure 17.23 The PIC microcontroller can drive a simple piezo speaker.

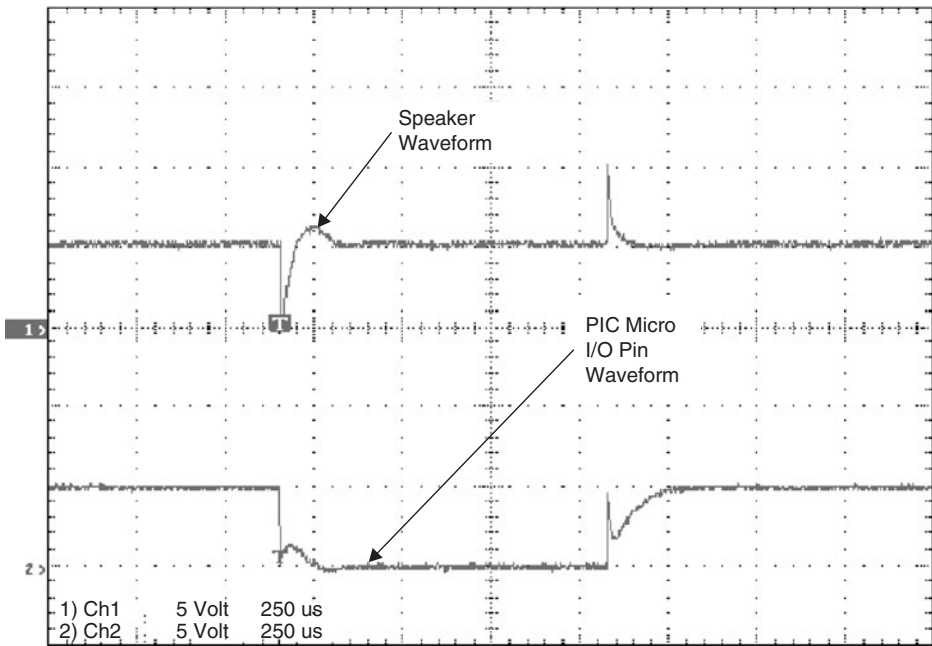


Figure 17.24 The voltage waveform across the speaker is probably not what you expect.

This passes dc waveforms through the capacitor (which filters out the kickback spikes) to the speaker or piezo buzzer. When a tone is output, your ear will hear a reasonably good tone, but if you were to look at the actual signal on an oscilloscope, you would see the waveform shown in Fig. 17.24 both from the PIC microcontroller's I/O pin and from the piezo buzzer itself.

The PIC microcontroller pin, capacitor, and speaker are actually quite a complex analog circuit. Note that the voltage output on the PIC microcontroller's I/O pin is changed from a straight waveform. This is due to the inductive effects of the piezo buffer. The important thing to note in this figure is that the upward spikes appear at the correct period for the output signal.

Timing the output signal generally is accomplished by toggling an output pin at a set period within the TMR0 interrupt handler or using the CCP module to produce a PWM tone. To generate a 1-kHz signal in a PIC microcontroller running at 4 MHz, you can use the following code (which does not use the prescaler) for TMR0 and the PIC microcontroller's interrupt capability:

```
org      4
int:
    movwf    _w          ; Save Context Registers
    bcf      INTCON, TOIF ; Reset the Interrupt
    movlw    256 - (250 - 4)
    movwf    TMR0        ; Reset TMR0 for another 500 μsecs
```

```
btfsf    SPKR        ; Toggle the Speaker
    goto    $ + 2
bsf      SPKER        ; Speaker Output High
    goto    $ + 2
bcf      SPKER        ; Speaker Output Low
swapf    _w, f        ; Restore Context Registers
swapf    _w, w
retfie
```

There are two points to notice about this interrupt handler: The first is that I don't bother saving the STATUS register's contents because neither the zero, carry, nor digit carry flags are changed by any of the instructions used in the handler. The second point to notice is the reload value of TMR0 to generate a 1-kHz output in a 4-MHz PIC microcontroller (an instruction clock period of 1 μ s); I have to delay 500 cycles for the wave's high and low. Because TMR0 has a divide by two counts on its input, I have to wait a total of 250 ticks. When I record TMR0, note that I also take into account the cycles taken to get to the reload (which is 7 or 8), divide them by 2, and take them away from the reload value.

For this handler, the reload value may be off by one cycle depending on how the main-line executes, for a worst-case error of 0.2 percent, or 2,000 ppm. This level of accuracy is approximately the same as what you would get for a ceramic resonator; and the change in the frequency should not cause a noticeable warbling (changes in the frequency) of the output.

When developing applications that output audio signals, I try to keep the tone within the range of 500 Hz to 2 kHz. This is well within the range of human hearing and is quite easy to create the software for. When you look at the "Christmas tree" in Chap. 21, you can see how this is done for creating simple tunes on the PIC microcontroller.

Relays and Solenoids

Some real-life devices that you may have to control by a microcontroller are electromagnetic relays, solenoids, and motors. These devices cannot be driven directly by a microcontroller because of the current required and the noise they generate. This means that special interfaces must be used to control electromagnetic devices.

The simplest method of controlling these devices is to just switch them on and off and by supplying power to the coil in the device. The circuit shown in Fig. 17.25 is true for relays (as is shown), solenoids (which are coils that draw an iron bar into them when they are energized), and dc motors (which will only turn in one direction).

In this circuit, the microcontroller turns on the Darlington transistor pair, causing current to pass through the relay coils, closing the contacts. To open the relay, the output is turned off (or a 0 is output). The shunt diode across the coil is used as a kickback suppressor. When the current is turned off, the magnetic flux in the coil will induce a large back EMF (voltage) that has to be absorbed by the circuit or there may be a voltage spike that can damage the relay power supply and even the microcontroller. This diode *never* must be

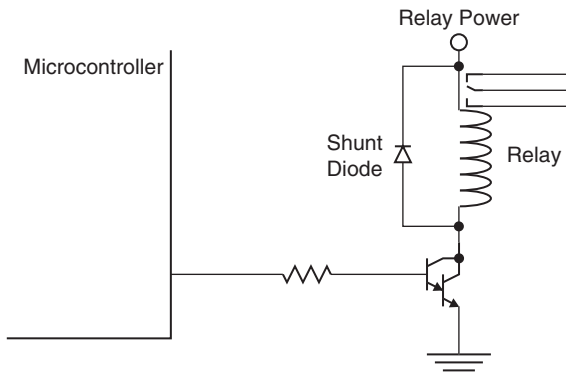


Figure 17.25 A relay can be controlled using a high-current transistor and a kickback-suppression diode.

forgotten in a circuit that controls an electromagnetic device. The kickback voltage is usually on the order of several hundred volts for a few nanoseconds. This voltage causes the diode to break down and allows current to flow, attenuating the induced voltage.

Rather than designing discrete circuits to carry out this function, I like to use integrated chips for the task. One of the most useful devices is the ULN2003A (Fig. 17.26) or the ULN2803 series of chips, which have Darlington transistor pairs and shunt diodes built in for multiple drivers.

Asynchronous (NRZ) Serial Interfaces

Asynchronous long-distance communications came about as a result of the Baudot teletype. This device mechanically (and, later, electronically) sent a string of electrical signals (which we would call a *packet of bits*, shown in Fig. 17.27) to a receiving printer.

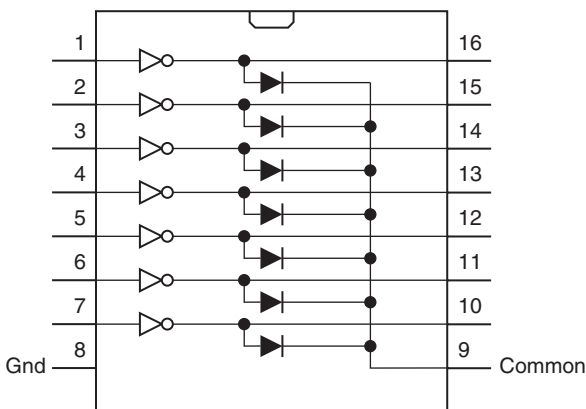


Figure 17.26 The ULN2003A (along with modern variants) is an efficient way to control multiple relays.

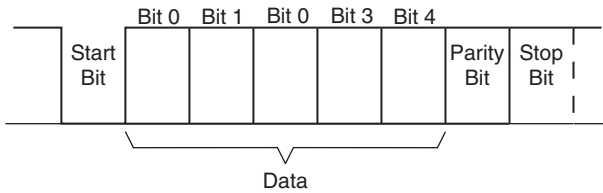


Figure 17.27 Every NRZ asynchronous packet consists of a start bit, a stop bit, and data bits.

With the invention of the teletype, data could be sent and retrieved automatically without having to have an operator sitting by the teletype all night unless an urgent message was expected. This data-packet format is still used today for the electrical asynchronous transmission protocols described below.

Before going on, there is one point that some people get unreasonably angry about, and that's the definition and use of the terms *data rate* and *baud rate*. The baud rate is the maximum number of possible data-bit transitions per second. This includes the start, parity, and stop bits at the ends of the data packet shown in the figure, as well as the 5 data bits in the middle. I use the term *packet* because we are including more than just data (there is also some additional information in there as well), so *character* and *byte* (if there were 8 bits of data) are not appropriate terms. This means that for every 5 data bits transmitted, 8 bits in total are transmitted (which means that nearly 40 percent of the data transmission bandwidth is lost in teletype asynchronous serial communications).

The data rate is the number of data bits that are transmitted per second. For this example, if you were transmitting at 110 baud (which is a common teletype data speed), the actual data rate would be 68.75 bps (or assuming 5 bits per character, 13.75 characters per second).

I tend to use the term *data rate* to describe the *baud rate*. This means that when I say *data rate*, I am specifying the number of bits of *all* types that can be transmitted in a given period of time (usually 1 second). I realize that this is not absolutely correct, but it makes sense to me to use it in this form, and I have used it consistently throughout this book (and I have not used the term *baud rate*).

With only 5 data bits, the Baudot code could transmit only up to 32 distinct characters. To handle a complete character set, a specific five-digit code was used to notify the receiving teletype that the next 5-bit character would be an extended character. With the alphabet and most common punctuation characters in the primary 32 characters, this second data packet wasn't required very often.

As discussed in Chap. 16, when waiting for a character, the PIC microcontroller USART receiver polls the line repeatedly at 1/16 bit period intervals until a 0 (space) is detected. The receiver then waits half a cycle before polling the line again to see if a glitch was detected and not a start bit. Once the start bit is validated, the receiver hardware polls the incoming data once every bit period multiple times (again, to ensure that glitches are not read as incorrect data).

The stop bit was provided originally to give both the receiver and the transmitter some time before the next packet is transferred (in early computers, the serial datastream was created and processed by the computers and not by custom hardware, as in modern

computers). It should be noted that the stop bit is always a 1, which is the reason for asynchronous communications being known as NRZ, or never return zero—at the end of the packet, the stop bit makes the line a 1, and the receiver knows that new data is coming in when a 0 start bit is detected.

The parity bit is a crude method of error detection that was first brought in with teletypes. The purpose of the parity bit is to indicate whether the data was received correctly. An odd parity bit meant that if all the mark bits in a packet after the start and before the stop bits were counted, the result would be an odd number. Even parity is checking all the data and parity bits and seeing if the number of mark bits is an even number. Along with even and odd parity, there are mark, space, and no parity. Mark parity means that the parity bit is always set to a 1, space parity is always having a 0 for the parity bit, and no parity is eliminating the parity bit altogether.

The most common form of asynchronous serial data packet is 8-N-1, which means 8 data bits, no parity, and 1 stop bit. This reflects the capabilities of modern computers to handle the maximum amount of data with the minimum amount of overhead and with a very high degree of confidence that the data will be correct.

I stated that parity bits are a crude form of error detection. I said this because they can only detect one bit error (i.e., if 2 bits are in error, the parity check will not detect the problem). If you are working in a high-induced-noise environment, you may want to consider using a data protocol that can detect (and, ideally, correct) multiple bit errors.

RS-232

In the early days of computing (the 1950s), while data could be transmitted at high speed, it couldn't be read and processed continuously. Thus a set of “handshaking” lines and protocols were developed for what became known as *RS-232 serial communications*.

With RS-232, the typical packet contained 7 bits (which is the number of bits in an ASCII character). This simplified the transmission of human-readable text but made sending object code and data (which were arranged as bytes) more complex because each byte would have to be split up into two nybbles (which are 4 bits long). Further complicating this is the fact that the first 32 characters of the ASCII character set are defined as special characters (i.e., carriage return, back space, etc.). This meant that the data nybbles would have to be converted (shifted up) into valid characters (this is why if you ever see binary data transmitted from a modem or embedded in an e-mail message, data is either sent as hex codes or as the letters A to Q). With this protocol, to send a single byte of data, 2 bytes (with the overhead bits resulting in 20 bits in total) would have to be sent (and surprisingly enough, to send data would take twice as long as sending a text file of the same length).

As I pointed out earlier, modern asynchronous serial data transmission is normally 8 bits at a time, which will avoid this problem and allow transmission of full bytes without breaking them up or converting them.

The actual RS-232 communications model is shown in Fig. 17.28, and in RS-232, the different devices are wired according to the functions they perform. DTE stands for *data terminal equipment* and is meant to be the connector used for computers (the PC uses this type of connection). DCE, or *data communications equipment*, was meant for modems that transfer data to other long-distance devices.

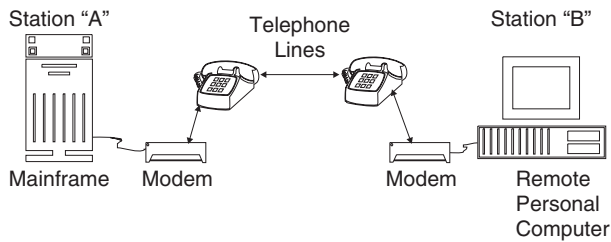


Figure 17.28 The PC communications model.

Understanding how different equipment fits in the RS-232 model is critical to connecting two devices successfully by RS-232. With a pretty good understanding of the serial data, we can now look at the actual voltage signals. As I mentioned earlier, when RS-232 was first developed into a standard, computers and the electronics that drove them were still very primitive and unreliable. Because of this, we’ve got a couple of legacies to deal with.

The first is the voltage levels of the data. A mark (1) is actually -12 V , and a space (0) is $+12\text{ V}$. From the figure, you should see that the hardware interface is not simply a TTL or CMOS level buffer. Later in this section I will introduce you to some methods of generating and detecting these interface voltages. Voltages in the switching region ($\pm 3\text{ V}$) may or may not be read as a 0 or 1 depending on the device. You always should make sure that the voltages going into a PIC microcontroller RS-232 circuit are in the valid regions.

Of more concern are the handshaking signals. These six additional lines (which are at the same logic levels as the transmit/receive lines and are shown in Fig. 17.29) are used to interface between devices and control the flow of information between computers. The “request to send” (RTS) and “clear to send” (CTS) lines are used to control data flow between the computer (DCE device) and the modem (DTE device). When the PC is ready to send data, it asserts (outputs a mark) on RTS. If the DTE device is capable of receiving data, it will assert the CTS line. If the PC is unable to receive data (i.e., the buffer is full or it is processing what it already has), it will deassert the RTS line to notify the DTE device that it cannot receive any additional information.

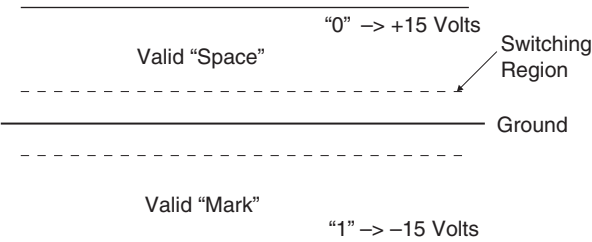


Figure 17.29 RS-232 data is specified at unusually high levels to and from which your circuit will have to translate.

The “data transmitter ready” (DTR) and “data set ready” (DSR) lines are used to establish communications. When the PC is ready to communicate with the DTE device, it asserts DTR. If the DTE device is available and ready to accept data, it will assert DSR to notify the computer that the link is up and ready for data transmission. If there is a hardware error in the link, then the DTE device will deassert the DSR line to notify the computer of the problem. Modems will deassert the DSR line if the carrier between the receivers is lost.

There are two more handshaking lines that are available in the RS-232 standard that you should be aware of, even though chances are that you will never connect anything to them. The first is the “data carrier detect” (DCD), which is asserted when the modem has connected with another device (i.e., the other device has picked up the phone). The “ring indicator” (RI) is used to indicate to a PC whether or not the phone on the other end of the line is ringing or is busy. These lines are used very rarely in PIC microcontroller applications because the AT command set provides a text message for these functions.

There is a common ground connection between the DCE and DTE devices. This connection is critical for the RS-232 level converters to determine the actual incoming voltages. The ground pin never should be connected to a chassis or shield ground (to avoid large current flows or be shifted and prevent accurate reading of incoming voltage signals). Incorrect grounding of an application can result in the computer or device with which it is interfacing resetting or having the power supplies blow a fuse or burn out. The latter consequences are unlikely, but I have seen it happen. To avoid these problems, make sure that chassis and signal grounds are separate or connected by a high-value (hundreds of kilohm) resistor.

Before going too much farther, I should expose you to an ugly truth: The handshaking lines are almost never used in RS-232 (and not just PIC microcontroller RS-232) communications. Normally, three-wire RS-232 connections are implemented as in Fig. 17.30. I normally accomplish this by shorting the DTR/DSR and RTS/CTS lines

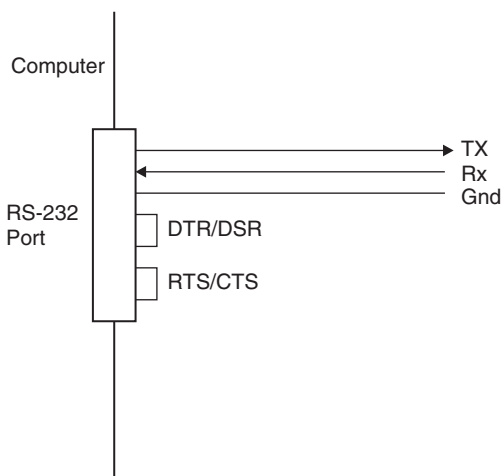


Figure 17.30 In most applications, only three wires are actually used to transmit data between devices.

together at the PIC microcontroller end. The DCD and RI lines are left unconnected. With the handshaking lines shorted together, data can be sent and received without having to develop software to handle the different handshaking protocols.

A couple of points on the three-wire RS-232: The first is that it cannot be implemented blindly; in about 20 percent of RS-232 applications that I have had to do over the years, I have had to implement some subset of the total seven-wire (transmit, receive, ground, and four handshaking lines) protocol lines. Interestingly enough, I have never had to implement the full hardware protocol. This still means that four out of five times if you wire the connection as shown in Fig. 17.30, the application will work.

With the three-wire RS-232 protocol, there may be applications where you don't want to implement the hardware handshaking (the DTR, DSR, RTS, and CTS lines); you may want to implement software handshaking. There are two primary standards in place. The first is known as the *XON/XOFF protocol*, in which the receiver sends an XOFF (DC3 or character 0x013) when it can't accept any more data. When it is able to receive data, it sends an XON (DC1 or character 0x011) to notify the transmitter that it can receive more data.

The final aspect of the RS-232 I want to discuss is the speeds at which data is transferred. When you first see the speeds (such as 300, 2,400 and 9,600 bps), they seem rather arbitrary. The original serial data speeds were chosen for teletypes because they gave the mechanical device enough time to print the current character and reset before the next one came in. Over time, these speeds have become standards, and as faster devices have become available, the speed have been doubled continually (i.e., 9,600 bps is 300 bps doubled five times).

To produce these data rates, the PIC microcontroller's USART uses a clock divider to produce a clock 16 times the data rate. The PIC microcontroller's operating clock is divided by integers to get the nominal RS-232 speeds. This might seem like it won't work out well, but because of RS-232's strange relationship with the number 13, the situation isn't as bad as it may seem.

If you invert (to get the period of a bit) the data speeds and convert the units to microseconds, you will discover that the periods are almost exactly divisible by 13. This means that you can use an even megahertz oscillator in the hardware to communicate over RS-232 at standard data rates. For example, if you had a PIC microcontroller running with a 20-MHz instruction clock and you wanted to communicate with a PC at 9,600 bps, you would determine the number of cycles to delay by

- 1** Finding the bit period in microseconds. For 9,600 bps, this is 104 μ s.
- 2** Dividing this bit period by 13 to get a multiple number. For 104 μ s, this is 8.

Now, if the external device is running at 20 MHz (which means a 200-ns cycle time), you can figure out the number of cycles as multiples of 8×13 in the number of cycles in 1 μ s. For 20 MHz, 5 cycles execute per microsecond. To get the total number of cycles for the 104- μ s bit period, you simply evaluate

$$20 \text{ cycles}/\mu\text{s} \times 13 \times 8 \mu\text{s/bit} = 1,300 \text{ cycles/bit}$$

When implementing an RS-232 interface, you can make your life easier by doing a few simple things. The first is the connection. Whenever I do an application, I standardize on using a 9-pin D-shell with the DTE interface (the one that comes out of the PC) and use standard straight-through cables. By making the external device DCE always and using a standard pinout, I don't have to fool around with null modems or making my own cables.

When I am creating the external device, I also loop back the DTR/DSR and CTS/RTS data pairs inside the external device rather than at the PC or in the cable. This allows me to use a standard PC and cable without having to do any wiring on my own or any modifications. It also looks a lot more professional.

Tying DTR/DSR and CTS/RTS also means that I can take advantage of built-in terminal emulators. Virtually all operating systems have a built-in "dumb" terminal emulator that can be used for debugging the external device without requiring the PC code to run, but it requires the handshaking lines to be working.

As I went through the RS-232 electrical standard earlier in this section, you probably were concerned about interfacing standard, modern technology (i.e., TTL and CMOS) devices to other RS-232 devices. This is actually a legitimate concern because without proper voltage-level conversion, you will not be able to read from or write to external TTL or CMOS devices. Fortunately, this conversion isn't all that difficult, or rather I should say that there are methods that can make it quite easy.

What I want to present to you is three methods that you can choose from for converting RS-232 signal levels to TTL/CMOS (and back again) when you are creating PIC microcontroller RS-232 projects. These three methods do not require ± 12 V and in fact just require the +5-V supply that is used for logic power.

The first method is to use an RS-232 converter that has a built-in charge pump to create the ± 12 V required for the RS-232 signal levels. Probably the most well-known chip that is used for this function is the Maxim MAX232 (Fig. 17.31). This chip is ideal for implementing three-wire RS-232 interfaces (or adding a simple DTR/DSR or RTS/CTS

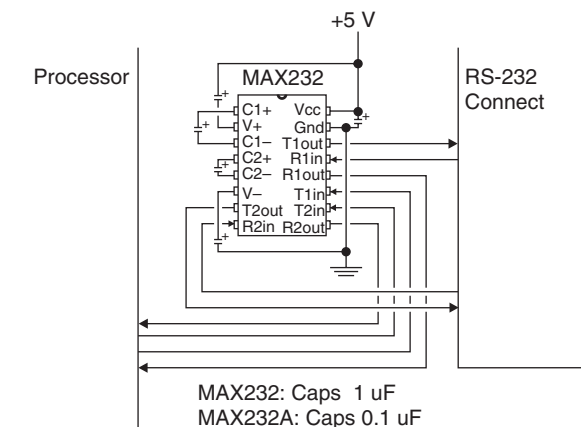


Figure 17.31 The Maxim MAX232 is a commonly used chip that generates its own ± 12 V for RS-232 communications.

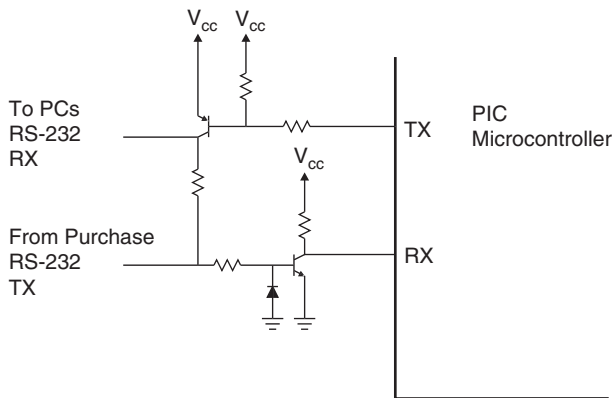


Figure 17.32 Voltage-stealing RS-232 interface that inverts the RS-232 signals to and from the PIC microcontroller.

handshaking interface). Ground for the incoming signal is connected to the “processor” ground (which is not the case’s ground).

Along with the MAX232, Maxim and some other chip vendors have a number of other RS-232 charge-pump-equipped devices that will allow you to handle more RS-232 lines (to include the handshaking lines). Some charge-pump devices that are also available do not require the external capacitors that the MAX232 chip does, which will simplify the layout of your circuit (although these chips do cost quite a bit more).

The next method of translating RS-232 and TTL/CMOS voltage levels is to use the transmitter’s negative voltage. The circuit in Fig. 17.32 shows how this can be done and will be demonstrated later in this book. This circuit relies on the RS-232 communications only running in half-duplex mode (i.e., only one device can transmit at a given time). When the external device wants to transmit to the PC, it sends the data either as a mark (leaving the voltage returned to the PC as a negative value) or as a space by turning on the transistor and enabling the positive-voltage output to the PC’s receivers. If you go back to the RS-232 voltage specification drawing, you’ll see that +5 V is within the valid voltage range for RS-232 spaces. This method works very well (consuming just about no power) and is obviously a very cheap way to implement a three-wire RS-232 bidirectional interface.

As an aside, before going on to the last interface circuit, I should point out a big advantage of the preceding circuit. The advantage results from the fact that the PIC microcontroller’s RS-232 transmitter circuit receives its negative voltage from the PC’s RS-232 transceiver *if* the external device (with this circuit) is connected to a PC. This means that the PC’s TX and RX are connected together, and it can “ping” (send a command that is ignored by the external device) via the RS-232 port to see if an external device is connected to it.

To do this, you need to specify the ping character as something that the external device can recognize and modify so that the PC’s software can recognize that the interface is working. The method that I have employed in the past is to use a microcontroller with “bit banging” software to change some mark bits when it recognizes that a ping character is being received.

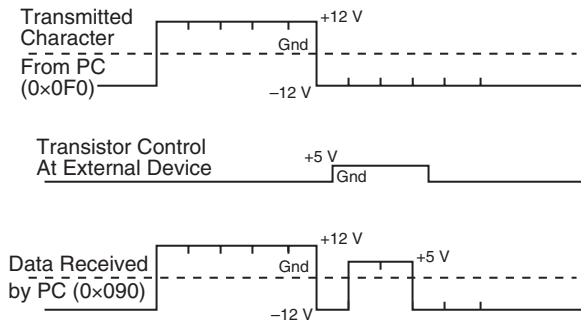


Figure 17.33 Sending 0xF0 and letting the receiver modify it is one way of detecting whether or not there is a receiver and if it is active.

In Fig. 17.33 I show a ping character of 0xF0 that is modified by the external device (by turning on the transistor) to change some bits into spaces. If the PC receives nothing at all, then there is nothing attached to it, and if it receives 0xF0, then the external device is not active.

With the availability of many CMOS devices requiring very minimal amounts of current to operate, you might be wondering about different options for powering your circuit. One of the most innovative that I have come across is using the PC's RS-232 ports themselves as powering devices that are attached to it using the circuit shown in Fig. 17.34.

When the DTR and RTS lines are outputting a space, a positive voltage (relative to ground) is available. This voltage can be regulated and the output used to power the devices attached to the serial port (up to about 5 mA). For extra current, the TX line also can be added into the circuit as well, with a break being sent from the PC to output a positive voltage.

The 5 mA is enough current to power the transistor/resistor type of RS-232 transmitter and a PIC microcontroller running at 4 MHz, along with some additional hardware (such as an LCD). You will not be able to drive an LED with this circuit, and you may find that some circuits that you normally use for such things as pull-ups and pull-downs will consume too much power, and you'll have to specify different resistance values.

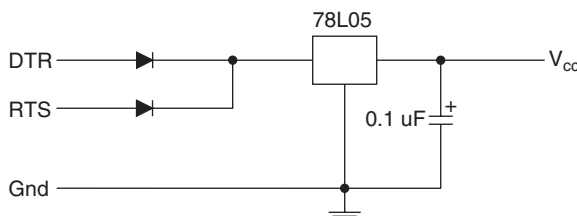


Figure 17.34 The unused handshaking lines can be used to provide power for your PIC microcontroller application.

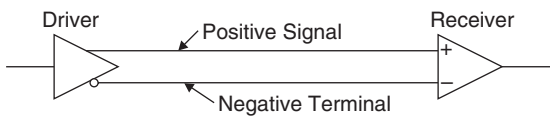


Figure 17.35 For very long distances, you can transmit data using differential-pair wiring.

Now, with this method of powering the external device, you do not have use of the handshaking lines, but the savings of not having to provide an external power supply (or battery) will outweigh the disadvantages of having to come up with a software ping-pong and handshaking protocol. Externally powering a device attached to the RS-232 port is ideal for input devices such as serial mice that do not require a lot of power.

RS-485/RS-422

So far in this book I have discussed single-ended asynchronous serial communications methods such as RS-232 and direct NRZ device interfaces. These interfaces work well in home and office environments but can be unreliable in environments where power surges and electrical noise can be significant. In these environments, a double-ended or differential-pair connection is optimal to ensure the most accurate communications.

A *differential-pair* serial communications electrical standard consists of a balanced driver with positive and negative outputs that are fed into a comparator that outputs a 1 or a 0 depending on whether or not the positive line is at a higher voltage than the negative line. Figure 17.35 shows the normal symbols used to describe a differential-pair connection.

There are several advantages to this data-connection method. The most obvious one is that the differential pair doubles the voltage swing sent to the receiver, which increases its noise immunity. This is shown in Fig. 17.36; when the positive signal goes high, the negative voltage goes low. The change in the two receiver inputs is 10 V rather than the 5 V of a single line. This is assuming that the voltage swing is 5 V for the positive and negative terminals of the receiver. This effective doubling of the signal voltage reduces the impact the electrical interface has on the transmitted signal.

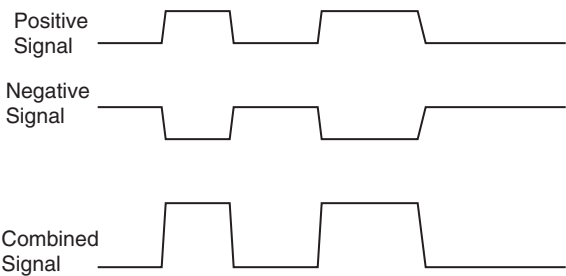


Figure 17.36 Differential data transmission consists of two opposite lower-voltage signals creating a high-quality signal.

Another benefit of differential-pair wiring is that if one connection breaks, the circuit will operate (although at reduced noise-reduction efficiency). This feature makes differential pairs very attractive in cars, aircraft, and spacecraft, where loss of a connection could be catastrophic.

To minimize ac transmission-line effects, the two wires should be twisted around each other. Twisted-pair wiring can either be bought commercially or made simply by twisting two wires together; twisted wires have a characteristic impedance of $75\ \Omega$ or greater.

A common standard for differential-pair communications is RS-422. This standard, which uses many commercially available chips, provides

- 1 Multiple-receiver operation
- 2 Maximum data rate of 10 Mbps
- 3 Maximum cable length of 4000 m (with a 100-kHz signal)

Multiple receiver operation, as shown in Fig. 17.37A, allows signals to be broadcast to multiple devices. The best distance and speed changes with the number of receivers of the differential pair, along with its length. The 4000 m at 100 kHz or 40 m at 10 MHz

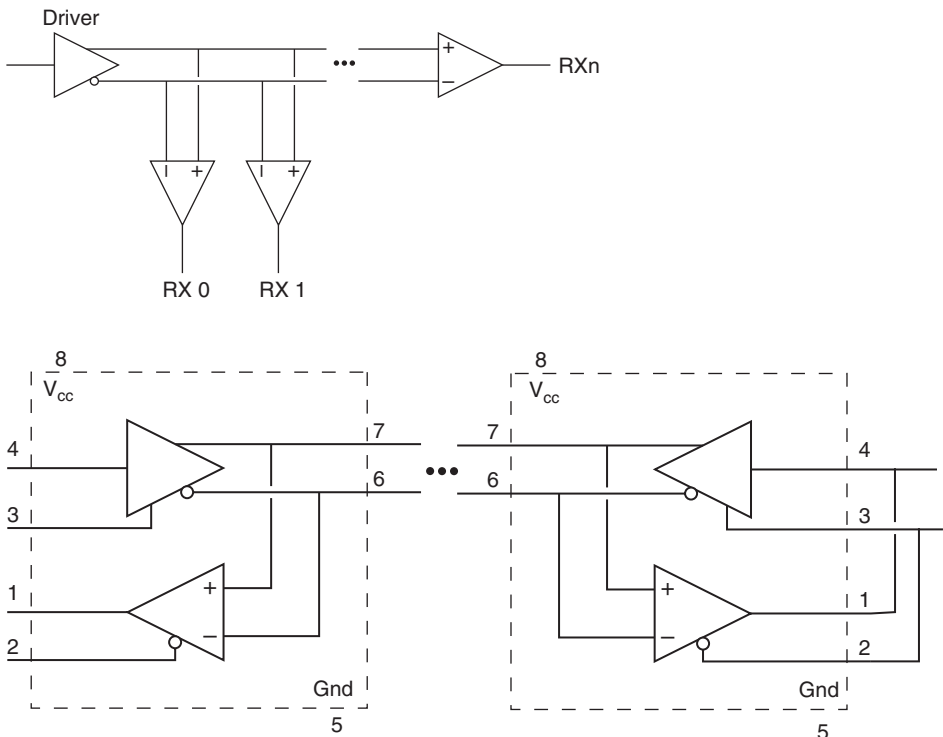


Figure 17.37 A. The RS-422 protocol allows multiple receivers on the same differential-pair wiring. B. The 75176 driver chip is used to create a single RS-485 differential-pair communications medium.

are examples of this balancing between line length and data rate. For long data lengths, a few-hundred-ohm terminating resistor may be required between the positive terminal and the negative terminal at the end of the lines to minimize reflections coming from the receiver and affecting other receivers.

RS-422 is not as widely used as you might expect; instead, RS-485 is much more popular. RS-485 is very similar to RS-422, except that it allows multiple drivers on the same network. The common chip is the 75176, which has the ability to drive and receive on the lines, as shown in Fig. 17.37.

In the right 75176 of Fig. 17.37, I show the RX and TX and two enables tied together. This results in a two-wire differential I/O device. Normally, the 75176s are left in RX mode (pin 2 reset) unless they are driving a signal onto the bus. When the unused 75176s on the lines are all in receive mode, anyone can take over the lines and transmit data.

As with the RS-422, multiple 75176s (up to 32) can be on the RS-485 lines with the capability of driving or receiving. When all the devices are receiving, a high (1) is output from the 75176. This means that the behavior of the 75176 in the RS-485 (because these are multiple drivers) is similar to that of a dotted AND bus; when one driver pulls down the line, all receivers are pulled low. For the RS-485 network to be high, all unused drivers must be off, or all active drivers must be transmitting a 1. This feature of the RS-485 is taken advantage in small system networks such as CAN.

The only issue to be on the lookout for when creating RS-485/RS-422 connections is to keep the cable polarities correct (positive to positive and negative to negative). Reversing the connectors will result in lost signals and misread transmission values.

Synchronous Serial Interfaces

For synchronous data communications in a microcontroller, a clock signal is sent along with serial data, as shown in Fig. 17.38. The clock signal strobes the data into the receiver, and the transfer can take place on the rising or falling edge of the clock. This method of transmission is different from the asynchronous protocol by the provision of the clock line from the transmitter; in asynchronous communications, the receiver is expected to provide a clock for timing the incoming data. The clock used in asynchronous transmission must be very precise and closely matched to the receiver, whereas in synchronous transmission there is only one clock, and the receiver does not need to be precisely tuned for accepting the transmitter's data.



Figure 17.38 Synchronous communications consist of data being latched into the receiver using a clock from the transmitter.

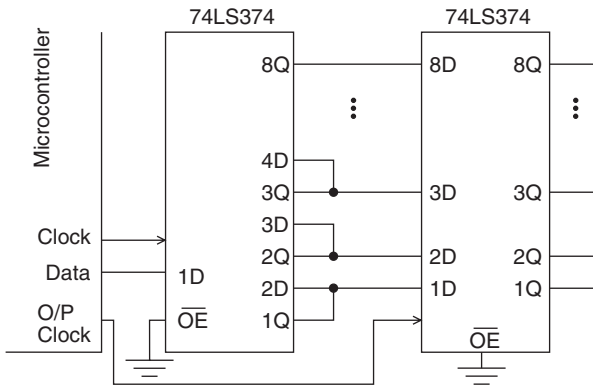


Figure 17.39 Converting synchronous serial data to parallel bits is quite easy to do.

A typical circuit using discrete devices could be that in Fig. 17.39. This circuit converts serial data into eight digital outputs that all are available at the same time (when the O/P clock is strobed). For most applications, the second 374 providing the parallel data is not required. This serial-to-parallel conversion also can be accomplished using serial-to-parallel chips, but I prefer using 8-bit registers because they are generally easier to find than other TTL parts.

There are two very common synchronous data protocols—Microwire and SPI. These methods of interfacing are used in a number of chips (such as the serial EEPROMs used in the BASIC Stamps). While the Microwire and SPI standards are quite similar, there are a number of differences that should be noted.

I consider these protocols to be methods of transferring synchronous serial data rather than microcontroller network protocols because each device is addressed individually (even though the clock/data lines can be common between multiple devices). If the chip select for the device is not asserted, the device ignores the clock and data lines. With these protocols, only a single master can be on the bus.

If a synchronous serial port is built into the microcontroller, the data transmit circuitry might look like that in Fig. 17.40. This circuit will shift out 8 bits of data. For protocols such as Microwire, where a start bit is sent initially, the start bit is sent using direct reads and writes to the I/O pins. To receive data, a similar circuit would be used, but data would be shifted into the shift register and then read by the microcontroller.

The Microwire protocol is capable of transferring data at up to one megabit per second. Sixteen bits are only transferred at a time. After selecting a chip and sending a start bit, the clock strobes out an 8-bit command byte (labeled OP1, OP2, A5 to A0 in the diagram above), followed by (optionally) a 16-bit address word transmitted and then another 16-bit word either written or read by the microcontroller.

The SPI protocol is similar to Microwire, but with a few differences:

- 1** SPI is capable of up to 3 Mbps data transfer rate.
- 2** The SPI data word size is 8 bits.

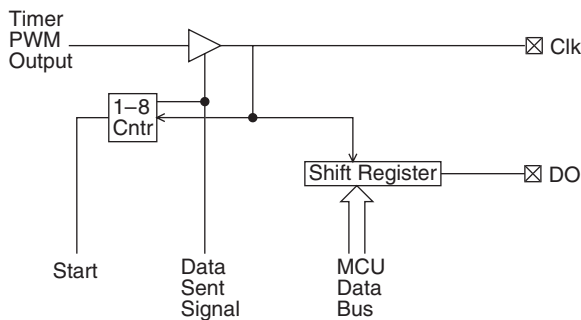


Figure 17.40 The synchronous serial hardware built into the PIC microcontroller allows data to flow through the input and output pins.

- 3 SPI has a “hold” that allows the transmitter to suspend data transfer.
- 4 Data in SPI can be transferred as multiple bytes known as “blocks” or “pages.”

Like Microwire, SPI first sends a byte instruction to the receiving device. After the byte is sent, a 16-bit address is optionally sent, followed by 8 bits of I/O. As noted earlier, SPI does allow for multiple-byte transfers. An SPI data transfer is shown in Fig. 17.41.

The SPI clock is symmetric (an equal low and high time). Output data should be available at least 30 ns before the clock line goes high and read 30 ns before the falling edge of the clock.

When wiring up a Microwire or SPI device, there is one trick that you can do to simplify the microcontroller connection, and that is to combine the DI and DO lines into one pin. Figure 17.42 is identical to what was shown earlier in this chapter when interfacing the PIC microcontroller into a circuit where there is another driver. In this method of connecting the two devices, when the data pin on the microcontroller has completed sending the serial data, the output driver can be turned off, and the microcontroller can read the data coming from the device. The current-limiting resistor between the data pin

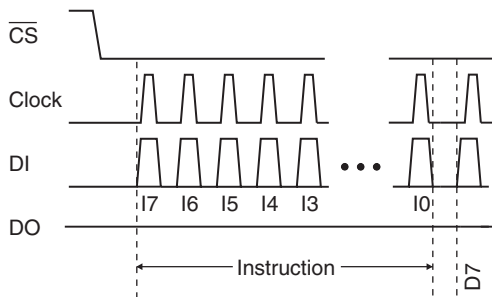


Figure 17.41 The SPI protocol is a very typical synchronous serial protocol with a device chip select.

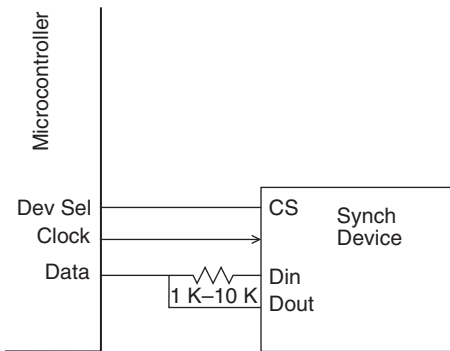


Figure 17.42 The input and output bits can be combined to simplify the wiring between devices.

and DI/DO limits any current flows when both the microcontroller and the device are driving the line.

I²C

The most popular form of microcontroller network is I²C (also known as I²C or “eye-squared-see”), which stands for “inter-intercomputer communications.” This standard was developed originally by Philips in the late seventies as a method to provide an interface between microprocessors and peripheral devices without wiring full address, data, and control busses between devices. I²C also allows sharing of network resources between processors (which is known as *multimastering*).

The I²C bus consists of two lines, a clock line (SCL) that is used to strobe data (from the SDA line) from or to the master that currently has control over the bus. Both these bus lines are pulled up (to allow multiple devices to drive them).

An I²C-controlled home entertainment system might be wired as in Fig. 17.43. The two bus lines are used to indicate that a data transmission is about to begin, as well as pass the data on the bus.

To begin a data transfer, a master puts a “start condition” on the bus. Normally, when the bus is in the idle state, both the clock and the data lines are not being driven (and are

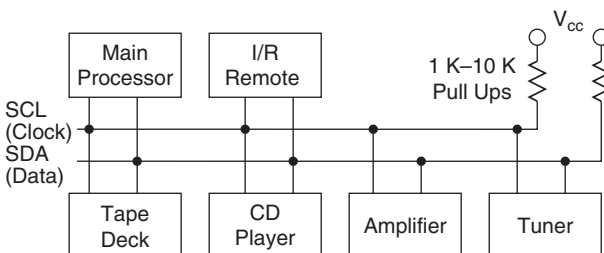


Figure 17.43 The I²C protocol is designed to support multiple devices on a single network wiring.

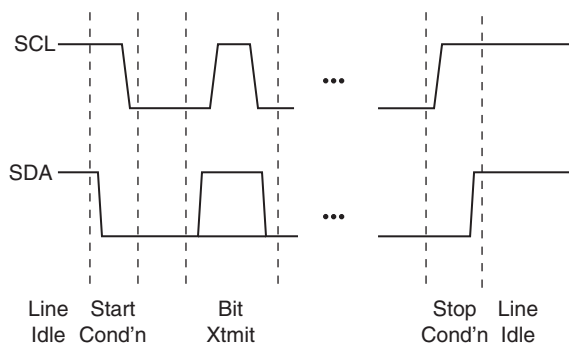


Figure 17.44 The I2C waveforms are similar to other synchronous serial protocols.

pulled high). To initiate a data transfer, the master requesting the bus pulls down the SDA bus line, followed by the SCL bus line. During data transmission, this is an invalid condition because the data line is changing while the clock line is active/high.

Each bit is then transmitted to or from the slave (the device the message is being communicated with by the master), with the negative clock edge being used to latch in the data, as shown in Fig. 17.44. To end data transmission, the reverse is executed; the clock line is allowed to go high, which is followed by the data line.

Data is transmitted in a synchronous (clocked) fashion. The most significant bit is sent first, and after 8 bits are sent, the master allows the data line to float (it doesn't drive it low) while strobing the clock to allow the receiving device to pull the data line low as an acknowledgment that the data was received. After the acknowledge bit, both the clock and the data lines are pulled low in preparation for the next byte to be transmitted or a stop/start condition is put on the bus. Figure 17.45 shows the data waveform.

Sometimes the acknowledge bit will be allowed to float high, even though the data transfer has completed successfully. This is done to indicate that the data transfer has completed and the receiver (which is usually a slave device or a Master that is unable to initiate data transfer) can prepare for the next data request.

There are two maximum speeds for I2C (because the clock is produced by a master, there really is no minimum speed)—standard mode runs at up to 100 kbps, and fast mode

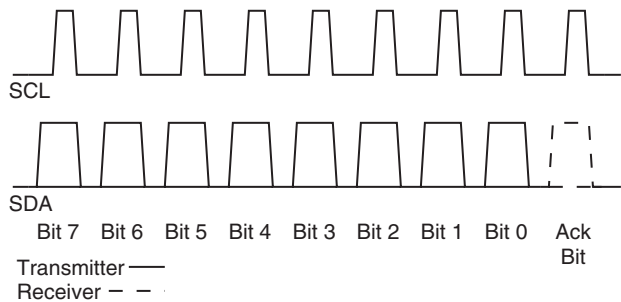


Figure 17.45 An I2C data packet.

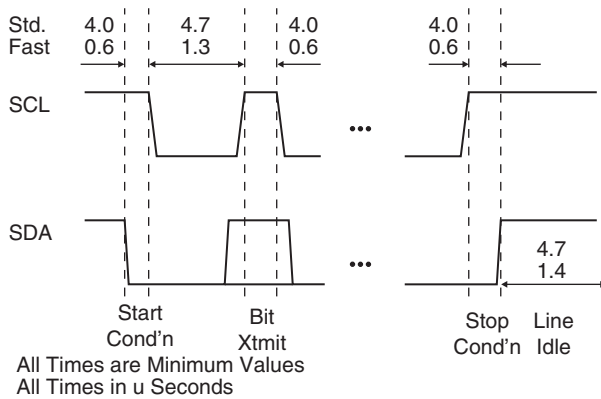


Figure 17.46 Bit timing for I2C data.

can transfer data at up to 400 kbps. Figure 17.46 shows the timing specifications for both the standard (“std.” or 100-kHz data rate) and fast (400-kHz data rate).

A command is sent from the master to the receiver in the format shown in Fig. 17.47. The receiver address is 7 bits long and is the bus address of the receiver. There is a loose standard to use the most significant 4 bits to identify the type of device, whereas the next 3 bits are used to specify one of eight devices of this type (or further specify the device type).

As I just said, this is *loose standard*. Some devices require certain patterns for the second 3 bits, whereas others (such as some large serial EEPROMS) use these bits to specify an address inside the device. In addition, there is a 10-bit address standard in which the first 4 bits are all set, the next bit reset, and the last 2 bits are the most significant 2 bits of the address, with the final 8 bits being sent in a following byte. All this means is that it is very important to map out the devices to be put on the bus and all their addresses.

This is really all there is to I2C communication, except for a few points. In some devices, a start bit has to be resent to reset the receiving device for the next command (i.e., in a serial EEPROM read, the first command sends the address to read from and the second reads the data at that address).

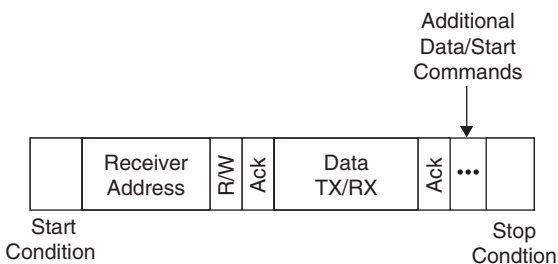


Figure 17.47 Multiple packets make up an I2C message.

The last point to note about I2C is that it's a multimastering device, which is to say that multiple microcontrollers can initiate data transfers on the bus. This obviously results in possible collisions on the bus (which is when two devices attempt to drive the bus at the same time). Obviously, if one microcontroller takes the bus (sends a "start condition") before another one attempts to do so, there is no problem. The problem arises when multiple devices initiate the "start condition" at the same time.

Actually, arbitration in this case is really quite simple. During the data transmission, hardware (or software) in both transmitters synchronize their clock pulses so that they match each other exactly. During the address transmission, if a bit is expected to be a 1 by a master is actually a 0, then it drops off the bus because another master is on the bus. The master that drops off will wait until the "stop condition" and then reinitiate the message. I realize that this is hard to understand with just a written description.

A "bit banging" I2C interface can be implemented in software of the PIC microcontroller quite easily. However, owing to software overhead, the fast mode probably cannot be implemented—even the standard mode's 100 kbps will be a stretch for most devices. I find implementing I2C in software to be best when the PIC microcontroller is the single master in a network. In this way, it doesn't have to be synchronized to any other devices or accept messages from any other devices that are masters and are running a hardware implementation of I2C that may be too fast for the software slave.

MOTOR CONTROL

Normally, when people think about the devices that the PIC[®] microcontroller can control and interface with, they think of other electronic devices that tend to be low power such as light-emitting diodes (LEDs), liquid-crystal displays (LCDs), buttons, memory chips, and so on. This is unfortunate because I believe that this restricts your perspective on some of the applications for which the PIC microcontroller can be used. A good example of this restriction is motor control—the PIC can provide many of the control signals necessary to drive different types of motors in different situations. In this chapter, as well as in Chaps. 20 and 21, I'm going to give you some information on how PIC microcontrollers can control different types of motors and provide you with some of the basic circuitry that would be required for doing so.

As part of the explanation of how the PIC microcontroller can be used to control motors is also a listing of the different types of motors that are available for use. Most developers tend to focus on only one type of device when there are a number of different types of motors that you can choose from for your application. Table 18.1 lists the most common types of motors, along with their characteristics and some sample applications in which they are used. This list is not necessarily complete—I have not listed ac motors that can be controlled by the PIC microcontroller, although they require a lot more effort and the drivers tend to be very specific to the actual motors.

Dc Motors

The basic dc motor (see Figs. 18.1 and 18.2) is an ideal choice for learning the basics of motor control. It is easy to find small, inexpensive motors; there are a number of different toys and development kits that can be used along with the motors to help you understand how they work and how to control them. Designing drivers and writing the control software for them are not trivial exercises—the basic theory behind their design is something that I will be going through over the next few sections. The basic information needed to work with a dc motor will be applied when the other types of motors are discussed.

TABLE 18.1 DIFFERENT TYPES OF ELECTRIC MOTORS THAT ARE EASILY CONTROLLED BY PIC MICROCONTROLLERS			
MOTOR TYPE	DRIVE HARDWARE	CONTROL SIGNALS	OPERATING CHARACTERISTICS
Dc motors	Transistor switches and H-bridges	Pulse width modulated (PWM)	Timed rotation
Bipolar stepper motors	H-bridges	Step sequence	Precise movement
Unipolar stepper motors	Transistor switches	Step sequence	Precise movement
Radio-control servos	Internal to servo	Pulse train	Positioning

An electric motor is really a number of electromagnets that can be switched on and off according to the position of the electromagnet rotor, as shown in Fig. 18.3. I have drawn the motor as having three electromagnets (which is common for small dc motors), and in the left drawing, electromagnet #1 is pointing upward, with electromagnet #2 producing a “South” pole that is drawn to the permanent magnet’s “North” pole. Electromagnet #3 produces a “North” pole, is repelled from the permanent magnet’s “North” pole, and is drawn to the permanent magnet’s “South” pole. In the drawing to the right, the motor’s axle has turned 60 degrees, and now electromagnet #1 is turned on and producing a “South” pole, whereas electromagnet #2 is facing the permanent magnet’s “North” pole and is turned off. Electromagnet #3 is still producing a “North”

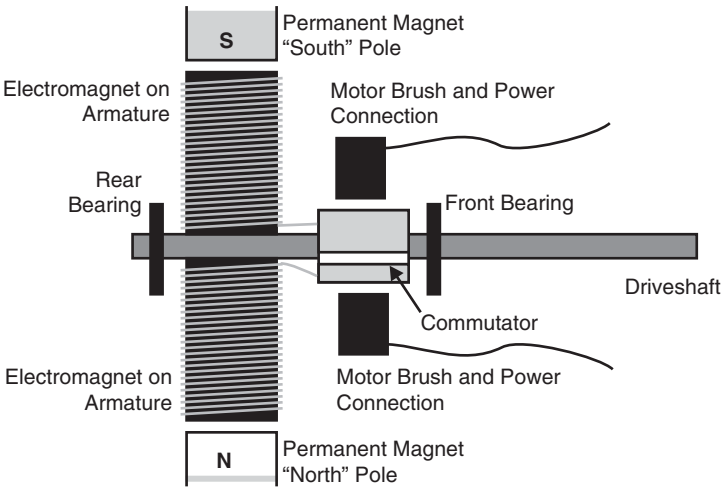


Figure 18.1 The interior parts of the basic dc motor.

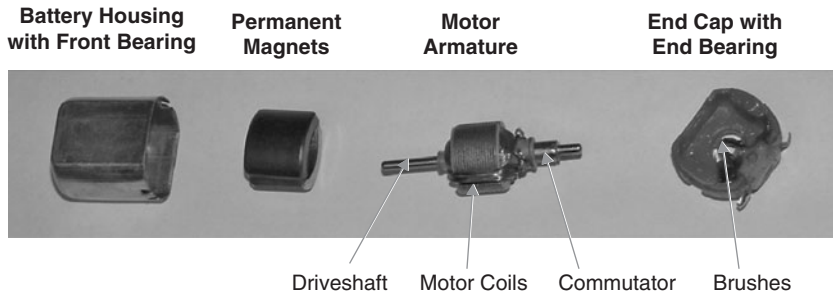


Figure 18.2 An actual dc motor disassembled with the parts labeled.

pole and continues to be drawn to the permanent magnet's "South" pole. As the motor's axle has turned 60 degrees in the figure, electromagnets #1 and #2 have changed operation to ensure that there continues to be a force drawing the electromagnets to the permanent magnets and providing torque on the axle. The motion of the axle along with the torque created by the electromagnets is passed out of the motor and used to drive whatever it is connected to.

A relay could be used for controlling the dc motor, but it really isn't an efficient method, and it cannot turn the motor on and off very quickly (which is a requirement that will be presented later in this chapter). The most common electrical switch for dc motors is the NPN transistor shown in Fig. 18.4. Along with the NPN transistor, I have included a kick-back diode that will absorb the energy stored in the motor's coils when power is removed from the motor; as with any magnetic device, a dc motor produces a voltage spike when power is removed, and protection for the rest of the application must be in place.

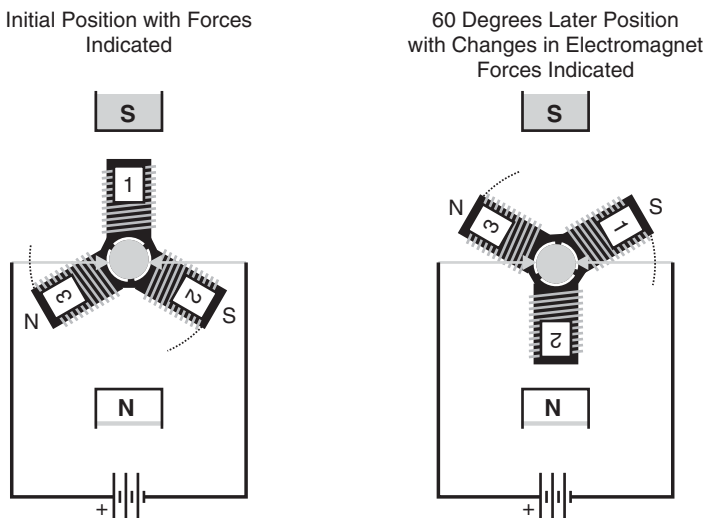


Figure 18.3 The changing fields in the electromagnets cause the dc motor to turn.

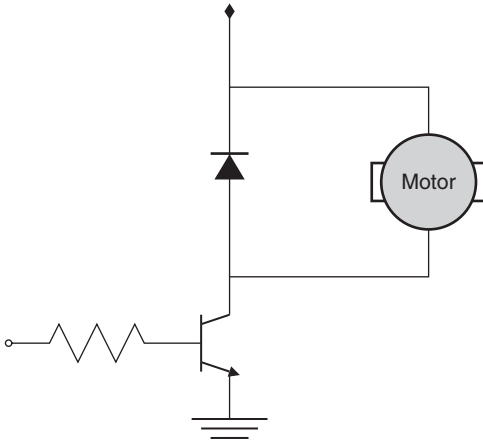


Figure 18.4 An NPN transistor can be used to control the flow of current to a dc motor.

Choosing appropriate power and transistors for controlling a dc motor requires that you understand what is the typical operating voltage and current for the motor and that you select a transistor that will be able to handle the maximum expected current and ensure that the voltage drop across the motor remains in the manufacturer's specified range. I typically want to find a transistor that can carry three or four times the normal operating current of the motor to ensure that the motor startup (which draws more current than it is rated at) is smooth and that any temporary increases in mechanical resistance will not cause the motor to stall.

In the preceding sentence there was an important caveat in saying that *temporary* increases in mechanical resistance should not cause the motor to stall. There is an obvious question here, and that is how to know if there is an increase in resistance that causes the current drawn by the motor to increase? This can be done by adding a low-value resistor in series with the motor and PIC microcontroller analog-to-digital converter (ADC) inputs at either end of the resistor that measures its voltage, as shown in Fig. 18.5.

In a "real" application, there would not be a direct connection from the motor current-monitoring resistor (a voltage divider would be used to reduce the voltages to a level that the ADC could measure efficiently) to the PIC microcontroller, and the ADC in the PIC microcontroller would have an external reference voltage to ensure that the voltages were measured correctly.

The formula to use to determine the amount of current passing through the resistor (and the motor) is

$$I_{\text{resistor}} = (V_{\text{ADC1}} - V_{\text{ADC2}})/R$$

The resistor value obviously should be very small, and you will have to specify a power rating that will not cause the resistor to burn out. For example, if you have a $0.5\text{-}\Omega$ resistor

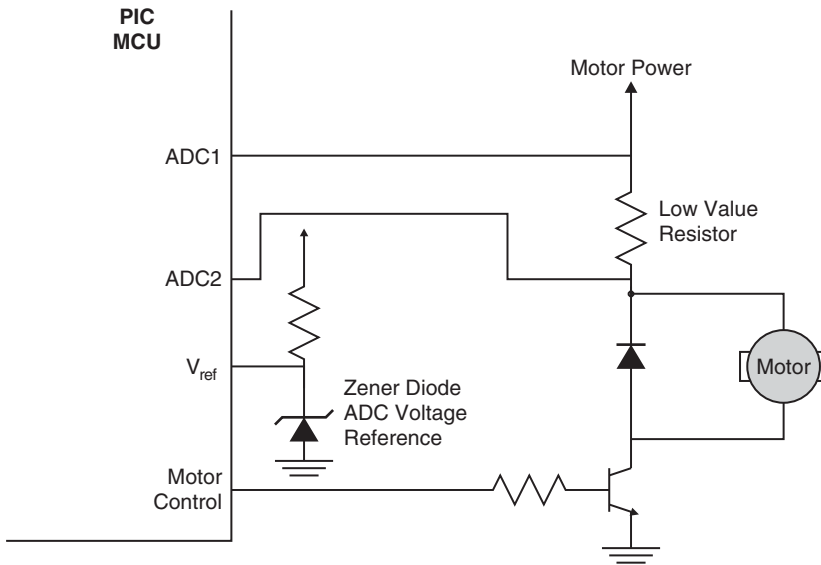


Figure 18.5 The current drawn by the motor can be monitored by adding a low-value resistor in series and monitoring the voltage drop across it.

and the motor normally draws 2 A, then I would recommend a 5-W resistor for situations where the motor starts up (and is in a high-current-draw situation) as well as when the motor is being stalled, in which case the current could double or triple. For this example, the normal operating difference of the two voltage sensors should be 1 V, but in the case of a stalled motor, the voltage difference could be as high as 3 or 4 V (which would indicate a stalled motor that should be shut down).

H-BRIDGE MOTOR CONTROL

In the preceding section I introduced you to the dc motor and presented you with a simple control along with a method of monitoring operation of the motor. This is a very basic level of control and is not useful for most situations; what you will really need is the ability to change the direction of the motor. The dc motor turns in the direction of the current flow through it, so to change the direction of the motor, you have to change the direction of the current flow. This is normally accomplished using a network of switches known as an *H-bridge* and is shown in Fig. 18.6.

In this circuit, if all the switches are open, no current will flow, and the motor won't turn. If switches 1 and 4 are closed, the motor will turn in one direction. If switches 2 and 3 are closed, the motor will turn in the other direction. Both switches on one side of the bridge *never* should be closed at the same time because this will cause the motor power supply to burn out, or a fuse will blow because there is a current path directly from motor power to ground.

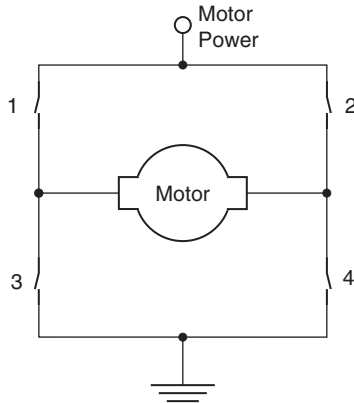


Figure 18.6 The H-bridge allows current to flow through a dc motor in either direction.

An H-bridge could be wired using a collection of transistors, but when you are first starting out, I recommend that you use something like the 293D (see Fig. 18.7) or 298 H-bridge driver chips that are designed for controlling a small dc motor. The 293D chip can control two motors (one on each side) connected to the buffer outputs (pins 3, 6, 11, and 14). Pins 2, 7, 10, and 15 are used to control the voltage level (the switches in the H-bridge diagram) of the buffer outputs. Pins 1 and 9 are used to control whether or not the buffers are enabled. These can be PWM inputs, which makes control of the motor speed very easy to implement. V_s is +5 V used to power the logic in the chip, and V_{ss} is the power supplied to the motors and can be anywhere from 4.5 to 36 V. A maximum of 500 mA can be supplied to the motors. As with the ULN2003A, the 293D

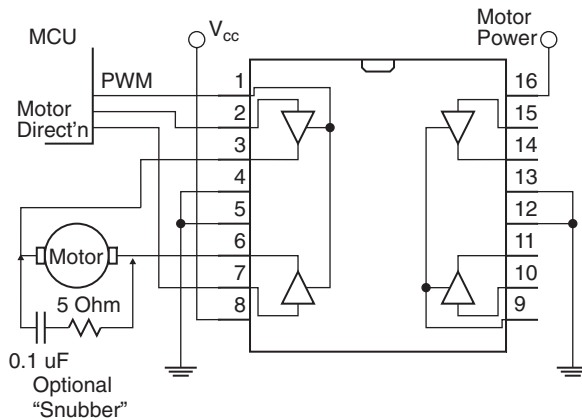


Figure 18.7 The 293D quad half H-bridge chip is an efficient way of driving two dc motors.

contains integral shunt diodes. This means that to attach a motor to the 293D, no external shunt diodes are required.

In the example circuit in Fig. 18.7, you'll notice that I've included an optional snubber resistor and capacitor. These two components, wired across the brush contacts of the motor, will help to reduce electromagnetic emissions and noise spikes from the motor. In the motor control circuits that I have built, I have never found them to be necessary. If you find erratic operation from the microcontroller when the motors are running, however, you may want to put in the 0.1- μ F capacitor and 5- Ω (2-W) resistor snubber across the motor's brushes, as shown in the circuit above.

You must remember that the 293D and 298 motor controller chips are bipolar devices with a 0.7-V drop across each driver (or 1.4 to 1.5 V for a dual-driver circuit, as is shown in Fig. 18.7). This drop must be factored in when selecting the motor to be used, and the "motor power" voltage applied to the chips, along with the current drawn by the motor, results in a fairly significant amount of power dissipation within the driver. The 293D is limited to 1 A total output, and the 298 is limited to 3 A. For these circuits to work best, a significant amount of heat sinking is required.

A modification of the full bridge is the half H-bridge shown in Fig. 18.8. While requiring literally twice the power supply of the full H-bridge, it only requires half the number driver transistors as the full H-bridge. This circuit has advantages over the full H-bridge in that it does not require complementary PNP or P-channel MOSFETs to the NPN or N-channel MOSFET drivers, which means that there will be less power loss through the control transistors.

MOTOR SPEED CONTROL USING A SOFTWARE PWM

With the H-bridge we have a circuit that can be used to control motor direction, so the only aspect of the motor that is left to control is the speed at which the motor turns. This

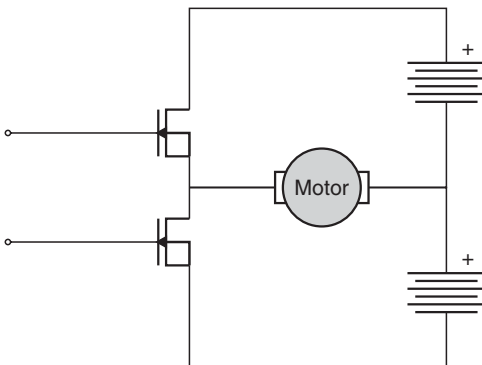


Figure 18.8 The half H-bridge driver is used in applications where the motor can be expected to turn equally in either direction.

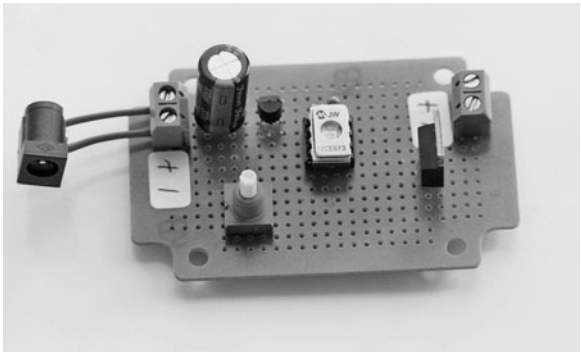


Figure 18.9 A completed basic motor control circuit using an ADC-equipped PIC microcontroller to specify the PWM duty cycle.

is accomplished by using a PWM signal—varying the amount of power being passed to the motor by turning it on and off rapidly (at a speed above human hearing so that any noise made by the motor turning on and off cannot be heard). This can be accomplished a number of different ways, which will be explored and demonstrated in this section. The circuit shown in Fig. 18.9 (with schematic in Fig. 18.10 and bill of materials in Table 18.2) is very fast to build and a good way to experiment with controlling dc motors using the PIC microcontroller.

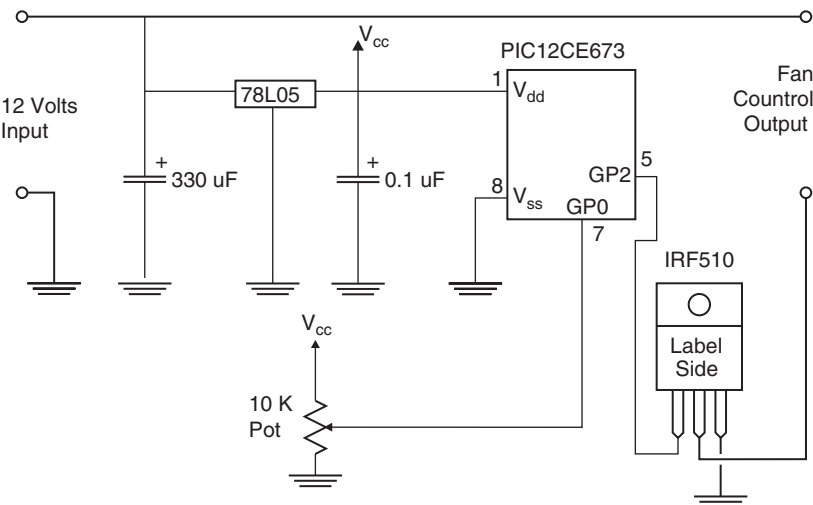


Figure 18.10 The dc motor control circuit consists of an ADC-equipped PIC microcontroller and a high-current MOSFET transistor to switch the motor on and off.

TABLE 18.2 BILL OF MATERIALS FOR DC MOTOR CONTROLLER CIRCUIT

PART	DESCRIPTION
PIC12F675	PIC12F675–04/IP
78L05	78L05 +5-V regulator in a TO-92 package
IRF510	IRF510 high-current N-channel MOSFET transistor
330- μ F	330- μ F, 50-V electrolytic capacitor
0.1- μ F	0.1- μ F, 16-V tantalum capacitor
10-k Ω pot	10-k Ω single-turn potentiometer
Misc.	Prototyping PCB, two screw terminals, wiring

I built my prototype on a fully drilled printed circuit board (PCB) prototyping card, and it took me less than half an hour to solder together the various components. I did this because the circuit is so simple that if I had made any mistakes, I could rewire and add components very easily. Note in the photograph of the project that I used a 2.5-mm power plug for the input, where I simply solder some 20-gauge solid-core wire to the power plug and wired it to the screw terminals used for the power connection.

The potentiometer is used with the PIC microcontroller's ADC to provide the operating level for the motor. I used the GP0 pin for the analog voltage input, with the remaining five pins used as digital input-output (I/O). The potentiometer will provide voltages anywhere from 0 to V_{cc} (5 V).

The motor control simply consists of an IRF510 N-channel MOSFET driver transistor. This transistor is quite inexpensive and can handle up to 5.6 A with a 10-V input with an on resistance (between the drain and source) of 0.54 Ω . This transistor is used as a switch for the PWM output.

The actual application code is very simple and only requires 42 instructions. It was designed to output a 10-kHz PWM signal, with the ADC being polled in between each ADC pulse. This polling and the software associated with it are the reason why there is the inactive period at full speed when you look at the full-power PWM waveform.

The pseudocode for the application is

```
main()                                // Fan Control Application
{

    int  ADCON = 32;
    int  ADCoff = 1;
    int  i;

    OSCCAL = CalibrationValue;
```

```

ADCON1 = 6;                // All I/O Pins Digital Except for GP0
ADCON0 = 0x041;            // Enable the ADC

TRISIO = 0x03F ^ (1 << 2); // GP2 is the PWM Output

while (1 == 1) {           // Loop Forever

    GP2 = 1;                // Output the PWM Signal while ADC
                            // Capacitor is Stabilizing to the Pot
                            // Input
    for (i = 0; i < ADCON; i++ );
    GP2 = 0;
    for (i = 0; i < ADCoff; i++ );

    ADCON0 = ADCON0 | Go;    // Start the ADC Operation

    GP2 = 1;                // Output the PWM Signal while ADC
                            // is processing the Pot Input
    for (i = 0; i < ADCON; i++ );
    GP2 = 0;
    for (i = 0; i < ADCoff; i++ );

    ADCON = (ADRES >> 3) + 1; // Get the ADC Value and Scale it for
    ADCoff = 33 - ADCON;      // the Application

}
} // End Fan Control Application

```

In this application, note that I output the PWM signal twice for each ADC sample loop. During the first PWM signal output, the ADC's input capacitor is allowed to stabilize. During the second PWM output, the ADC operation is allowed to take place. I set the ADC with the divide by eight TAD clock. The actual assembly-language code is not much more complex and can be found in the `code\motorctrl` folder.

In this application, there are a few things that I want to bring to your attention. The first is that the PWM output operation, which was coded in assembly language as

```

bsf      PWMOut              ; Output the First Value
decfsz   ADCON, f
goto     $ - 1
bcf      PWMOut
decfsz   ADCoff, f
goto     $ - 1

```

takes 99 instruction cycles under all cases, and I have written the ADC scaling so that the value always will be between 1 and 32. By making the value one-based instead of zero-based, I avoided the problem where if one of the ADC time values were equal to zero, I would end up looping 256 times.

The second issue is that the scaling routine takes more cycles than I would have liked:

```

rrf    ADRES, w           ; Read the ADC result
movwf  ADCON
rrf    ADCON, f           ; Convert to a 1-32 Value
rrf    ADCON, f
movlw  0x01F
andwf  ADCON, f
incf   ADCON, f

```

The problem is that the ADCON scaling has to be

$$\text{ADCON} = (\text{ADRES} \gg 3) + 1;$$

for the PWM code to work properly. To get the higher PWM frequency, the application's clock would have to be increased. To keep the application simple, I elected to use the PIC microcontroller's built-in 4-MHz RC oscillator. If I were to increase the operating frequency, I would have had to use an external oscillator.

MOTOR SPEED CONTROL USING THE CCP PWM

In this application I want to jump right to creating a fairly complex control application that allows forward and reverse, as well as speed control, of a dc motor using the full H-bridge circuit of Fig. 18.11 with the bill of materials in Table 18.3. The direction and speed of the motor are specified by the potentiometer; when the potentiometer is in centered, the motor is stopped. When the potentiometer is turned toward one extreme, the

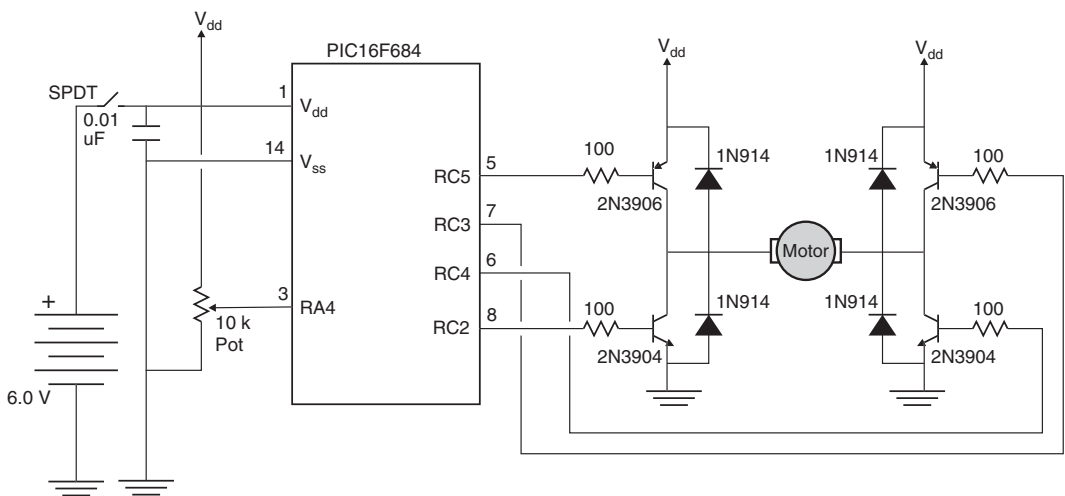


Figure 18.11 A complete dc motor control application.

TABLE 18.3 PWM DC MOTOR CONTROL BILL OF MATERIALS	
PART	DESCRIPTION
PIC16F684	PIC16F684–04/P
2N3904	2N3904 NPN bipolar transistors
2N3906	2N3906 PNP bipolar transistors
1N914	1N914 or 1N4148 silicon diodes
100	100 Ω ¼ W
10-k Ω pot	10-k Ω single-turn potentiometer
0.01- μ F	0.01 μ F capacitor, any type
SPDT	PCB-mountable single-pole double-throw switch
Misc.	PCB board, 3x “AA” battery clip, dc motor, two screw terminals

motor will start turning and speed up as the potentiometer reaches the end of its travel. If the potentiometer is turned in the opposite direction from center, then the motor will turn in the opposite direction, starting off slowly and moving much faster as the wiper approaches the other stop. This is a fairly intuitive interface and one that will allow you see the operation of the H-bridge as well as the PWM signal that controls the motor’s speed.

The circuit (see Fig. 18.11) was designed for using the PWM capabilities of the PIC16F684’s ECCP, which requires RC5–RC2 to interface to four motor drivers and run a single motor in both directions with PWM speed control. I used a small hobby motor rated at 4 V that I bought at an electronics store.

As I indicated earlier, one of the purposes of this experiment is to control the motors using the built-in ECCP PWM along with a potentiometer polled once every 100 ms for controlling the speed and direction of the motor. A PWM frequency of 15 kHz was chosen because it allowed a 64-step PWM control output and still ran above the range of most people’s hearing. Generally, a PWM control output of 20 kHz is desired because it ensures that it is above the range of hearing and will not affect other devices. To produce a PWM of 20 kHz, a TMR2 reset value of 48 would have had to be produced, which would require passing one-third the value of the ADC rather than one-half, as I have done in this application.

The ECCP PWM is quite easy to set up but does have an issue with the bipolar transistors used in this experiment—the high output of the PIC16F684 is less than V_{dd} applied to the PNP transistor, and when the PIC MCU output is high (turning off the PNP transistor), there still is some current flow. To eliminate this current flow (as well as any potential current flow to the NPN transistors when they are supposed to be off), I make the pin connected to the base of the unused transistor an input. This application code is called `cMotor.c`.

```
#include <pic.h>
```

```
/* cMotor - Control a DC Motor using a Potentiometer
```

This Program Monitors a Pot at RA3 (RA3) and moves a DC Motor Accordingly. Values less than 0x80 move the motor in reverse while Values greater than 0x80 move the motor forwards. When the Pot is at an extreme, the ECCP PWM moves at full speed.

Hardware Notes:

PIC16F684 running at 4 MHz Using the Internal Clock

RA4 - Pot Command

RC5/P1A - Motor Forwards High

RC4/P1B - Motor Reverse PWM (on Low)

RC3/P1C - Motor Reverse High

RC2/P1D - Motor Forwards PWM (on Low)

myke predko

05.01.10

*/

```
__CONFIG(INTIO & WDTDIS & PWRTEN & MCLRDIS & UNPROTECT \
    & UNPROTECT & BORDIS & IESODIS & FCMDIS);
```

```
int Dlay;                // LED Time on Delay Variable
char ADCValue;
```

```
main()
```

```
{
```

```
    PORTC = 0;
    CMCON0 = 7;           // Turn off Comparators
    ANSEL = 1 << 3;      // RA4 (AN3) is the ADC Input

    ADCON0 = 0b00001101; // Turn on the ADC
                        // Bit 7 - Left Justified Sample
                        // Bit 6 - Use VDD
                        // Bit 4:2 - RA4
                        // Bit 1 - Do not Start
                        // Bit 0 - Turn on ADC
    ADCON1 = 0b00010000; // Select the Clock as Fosc/8

    TMR2 = 0;            // TMR2 Provides PWM Period
    PR2 = 64;            // 15 kHz PWM Frequency
```

```
T2CON = 0b00000100;    // Enable TMR2
CCPR1L = 0;             // 0 Duty Cycle to Start Off

while(1 == 1)
{
    NOP();
    for (Dlay = 0; Dlay < 6666; Dlay++);    // 100 ms between
    NOP();                                   // Samples

    GODONE = 1;                            // Read Pot Value
    while (GODONE);

    ADCValue = ADRESH;    // Read in ADC Value
    if (ADCValue > 0x80)  // go Forwards
    {
        CCPR1L = (ADCValue - 80) >> 1;
        CCP1CON = 0b01001110;
        TRISC = 0b011011; // RC5/RC2 Output, RC3/RC4 Input
    }
    else                                // Go in Reverse
    {
        CCPR1L = (ADCValue ^ 0x7F) >> 1;
        CCP1CON = 0b11001110;
        TRISC = 0b100111; // RC5/RC2 Output, RC3/RC4 Input
    } // fi
} // elihw
} // End cMotor
```

There are a few things that you should be aware of for this experiment. The first is that the circuit is meant to work with a wide variety of different small hobby motors. You may find that the circuit does not have a very long battery life, the transistors get warm owing to excessive current passing through them, or the motors do not produce a lot of torque. As I indicated earlier, the motor driver given in this circuit is for the general case, and by understanding your motor's parameters, along with specifying a power supply and driver transistors with appropriate current, voltage and resistance parameters will help them perform at maximum efficiency. The four kickback suppression diodes are not optional (even if you have a very small motor), and if your circuit acts strangely, you may wish to add a 10- to 47- μ F electrolytic capacitor to the PIC MCU's power pins and a 0.1- μ F capacitor across the motor to help reduce the electrical noise produced by the motor.

Lastly, do not disassemble this circuit when you are finished; it will be required for the next experiment.

Stepper Motors

I'm always surprised to see stepper motors being used for just about any application that an electric motor would be considered for. This is surprising to me because I would consider stepper motors to be ideal for specific applications and suboptimal for

others. For example, I would consider a stepper motor to be ideal as a positioning device for a disk-drive read/write head; the ability to locate the head at a specific location (after a set number of steps) makes the stepper motor ideal for this type of application because there is a guaranteed power supply and the mass of the head isn't very large, so the amount of current needed to move the head and keep it in position is minimal. I would not consider stepper motors to be appropriate in robot-arm or motion-drive mechanisms because of their requirement for more power than standard dc motors; but I see them used in this application all the time. Similarly, in other applications I see stepper motors come up when they are clearly not the optimal type of dc motor.

The reason why stepper motors are so popular is they are so easy to use and program. This is not to say that dc motors or servos are not easy to work with, just that stepper motors are much easier to work with precisely. The ability to move a set number of steps allows precise positioning of mechanical devices, and in a mobile robot, you can track how far the robot has gone quite easily. These requirements can be met in the other types of motors, but the stepper motor makes it quite a bit easier to accomplish.

UNIPOLAR STEPPER MOTOR CONTROL

The unipolar stepper motor is subtly different from the bipolar motor in how its four coils are wired. Instead of current passing through two coils at a time, the unipolar stepper motor is designed with a common connection between each set of parallel-wired coils so that each coil can be turned on or off individually. The normal wiring configuration of the unipolar stepper motor is shown in Fig. 18.12, with the common wires

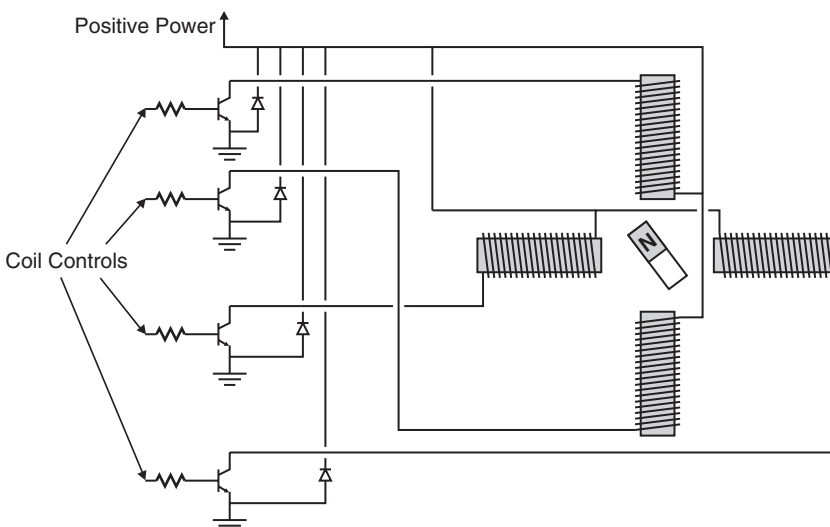


Figure 18.12 The unipolar stepper motor is built from multiple coils with a common connection.

connected to positive power and the individual coil wires being passed to open-collector drivers, which turn the coils on in sequence, drawing the shaft toward one coil at a time.

The unipolar stepper motor is usually differentiated by the number of wires coming out of it, whereas in Fig. 18.12 I imply that there are five wires coming from the unipolar stepper motor, there are usually six. Each pair of coils has its own common wire. There are some stepper motors that have five wires coming out, these seem to be a hybrid unipolar/bipolar stepper motor with each pair of coils wired differently. In such cases, you should treat the stepper motors as simply bipolar. The unipolar stepper motor can be used in exactly the same applications as the bipolar motor when the common wires are disconnected, and the four wires leading to an individual pair of coils are used.

There are a couple of differences that you should be aware of between the two types of stepper motors. Because the unipolar motor has only one coil active instead of the bipolar motor's two, it doesn't have the same torque as a bipolar motor. On the plus side, the bipolar motor control circuitry is a lot simpler to program, simply pulsing each coil in sequence.

Figure 18.13 shows the circuit that I came up with for testing the unipolar servo motor, and Table 18.4 lists the materials. In the figure you can see the six-pin header to which I soldered the six unipolar stepper motor wires, with the common wires (found with a DMM resistance check) being placed in the middle of the connector. Along with having to solder the stepper motor wires to a header, you should also Krazy glue a cardboard pointer to the shaft of the stepper motor so as to be able to observe its motion when you test it.

Testing the unipolar stepper motor is accomplished in exactly the same way as testing the bipolar stepper motor: `cStepper 3.c` will sequence through the coils, hopefully

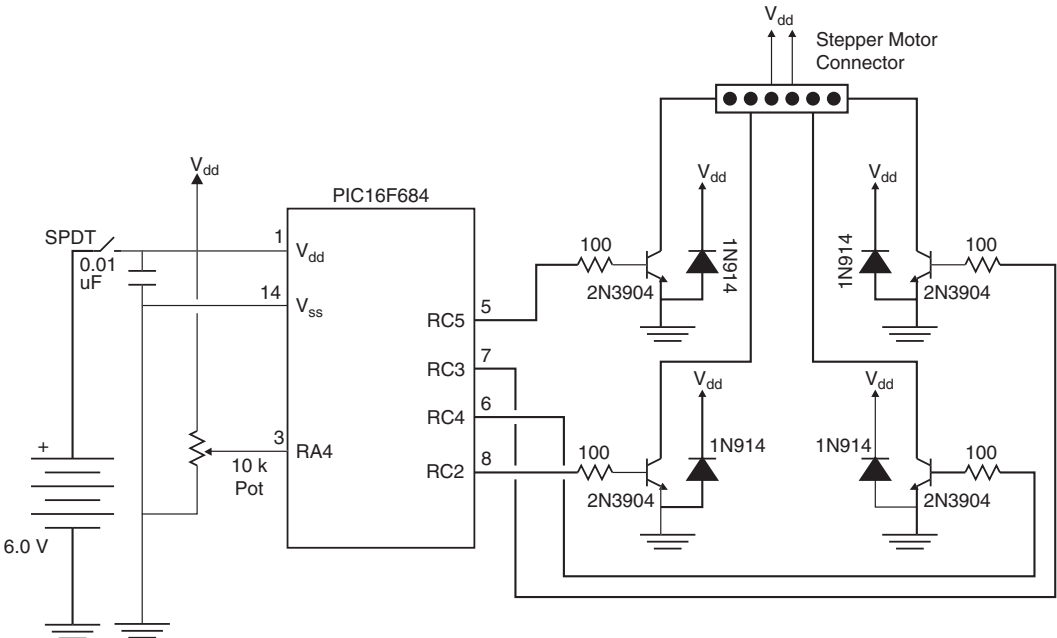


Figure 18.13 Unipolar stepper motor control circuit.

TABLE 18.4 BILL OF MATERIALS FOR UNIPOLAR STEPPER MOTOR CIRCUIT

PART	DESCRIPTION
PIC16F684	PIC16F684-04/P
3904	2N3904 NPN bipolar transistors
1N914	1N914 or 1N4148 silicon diodes
10-k Ω pot	10-k Ω single-turn potentiometer
100- Ω	100 Ω , 1/4 W resistor
0.01- μ F	0.01- μ F capacitor, any type
SPDT	PCB mountable single-pole double-throw switch
Stepper motor	Unipolar stepper motor (see text)
Misc.	Prototyping PCB, 4 $\times$ "AA" battery clip, six screw terminal headers, cardboard arrow (see text)

moving the cardboard pointer continuously. Again, if the pointer doesn't move, then move the wires to the control transistors until it does. A simpler way of testing and decoding the wiring is to touch the base connection (through the 100- Ω resistor) of each transistor to the Vdd rail of the breadboard to figure out which circuit is wired to which pin and then attach them in sequence (starting at RC2 and going to RC5) to the PIC MCU.

```
#include <pic.h>
/*  cStepper 3.c - Turn a Unipolar Stepper Motor

Hardware Notes:
PIC16F684 Running at 4 MHz with Internal Oscillator
RC5:RC2 - Stepper Motor Outputs

myke predko
05.01.15

*/

__CONFIG(INTIO & WDTDIS & PWRTEN & MCLRDIS & UNPROTECT \
& UNPROTECT & BORDIS & IESODIS & FCMDIS);

unsigned int j;
unsigned char OutputVal = 1 << 2;

main()
{
```

```

    PORTC = 0;
    CMCON0 = 7;                // Turn off Comparators
    ANSEL = 0;                 // Turn off ADC
    TRISC = 0b000011;         // RC5:RC2 Outputs

    while(1 == 1)              // Loop Forever
    {
        NOP();
        for (j = 0; j < 21000; j++);
        NOP();

        OutputVal = (OutputVal & 0x3C) << 1;
        if ((1 << 6) == OutputVal)
            OutputVal = 1 << 2;
        PORTC = OutputVal;

    } // elihw
} // End cStepper 3

```

I wrote code to control the movement of the unipolar stepper motor using a potentiometer; the control software is almost identical to that used in the preceding experiment with a 2- to 257-ms delay in the stepper motor movements. The C language version was called `cStepper 4.c`:

```

#include <pic.h>
/*  cStepper 4.c - Control a Unipolar Stepper Motor Using a Pot

Hardware Notes:
PIC16F684 Running at 4 MHz with Internal Oscillator
RC5:RC2 - Stepper Motor Outputs
RA4 - Potentiometer Cotrol

myke predko
05.01.15

*/

__CONFIG(INTIO & WDTDIS & PWRTEN & MCLRDIS & UNPROTECT \
    & UNPROTECT & BORDIS & IESODIS & FCMDIS);

char OutputVal = 1 << 2;
unsigned int j;
unsigned char Period;
const int Onems = 83;
main()
{

```



```

PORTC = 0;
CMCON0 = 7;           // Turn off Comparators
ANSEL = 1 << 3;       // RA4 (AN3) is the ADC Input

ADCON0 = 0b00001101; // Turn on the ADC
                        // Bit 7 - Left Justified Sample
                        // Bit 6 - Use VDD
                        // Bit 4:2 - RA4
                        // Bit 1 - Do not Start
                        // Bit 0 - Turn on ADC
ADCON1 = 0b00010000; // Select the Clock as Fosc/8
TRISC = 0b000011;    // RC5:RC2 Outputs

while(1 == 1)         // Loop Forever
{
    NOP();
    for (j = 0; j < Onems; j++);
    NOP();
    GODONE = 1;        // Start ADC
    for (j = 0; j < Onems; j++);
    Period = ADRESH;   // Read Value
    if (0x80 != Period) // Only Move if Something There
    {
        if (0x80 < Period) // Forwards
        {
            Period = (Period - 0x80) ^ 0x7F;
            OutputVal = (OutputVal & 0x3C) << 1;
            if ((1 << 6) == OutputVal)
                OutputVal = 1 << 2;
        }
        else // Reverse - "Period" OK
        {
            OutputVal = (OutputVal & 0x3C) >> 1;
            if ((1 << 1) == OutputVal)
                OutputVal = 1 << 5;
        } // fi

        PORTC = OutputVal; // Move Stepper

        while (0 != Period) // Delay at New Position
        {
            for (j = 0; j < Onems; j++);
            Period = Period - 1;
        } // elihw
    } // fi
} // End cStepper 4

```

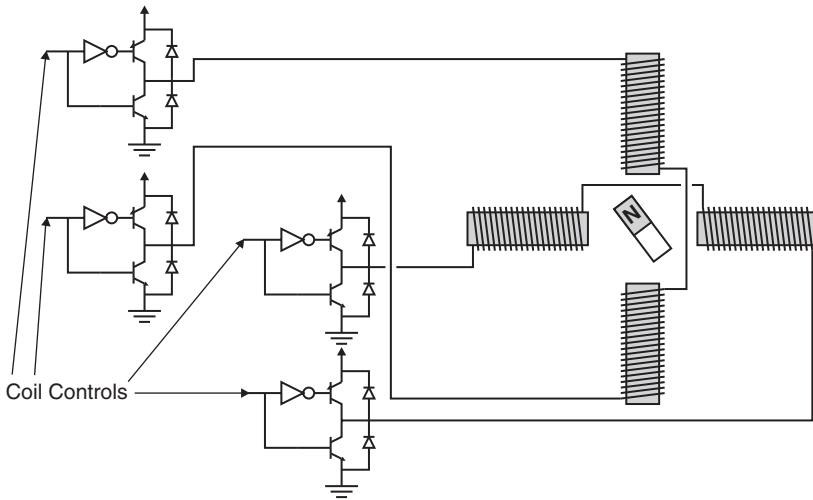


Figure 18.14 Two sets of coils pull the bipolar stepper motor output shaft into position.

BIPOLAR STEPPER MOTOR CONTROL

There are a few characteristics of bipolar stepper motors that you should be aware of. They consist of four (or more) coils arranged perpendicularly to each other, as I have drawn in Fig. 18.14. These four coils surround a magnetized shaft that will be either attracted or repelled when the coils are energized. To turn the stepper motor on, the coils are energized in a pattern that will cause it to turn in one direction or another. Because of the time required to energize the coils along with the inertia (as well as any load resistance) of the shaft and reduction gearing placed on the shaft output, the speed of the stepper motor is much more limited than that of the dc motor. The reduction gearing reduces the movement of the motor output from 45 or 90 degrees for each change in position of the shaft to just a couple of degrees or so to maximize the torque output of the motor. Along with the slower speed of the stepper motor, the need to keep at least one coil energized at any one time will draw more current than the dc motor.

The high-current push-pull drivers of the H-bridge are used to alternatively turn the two sets of coils on and off as well as to change polarities. In Table 18.5 I have listed the polarities for the different coils to move the shaft by 45 degrees at a time. This is known as *half-stepping* and requires that one or two coils be energized at any time. *Full-stepping* moves the shaft by 90 degrees at a time, and only one set of coils is energized at any time.

In the table, the “North” and “South” specifications are arbitrary and are used to indicate that the polarity of the coils’ magnetic fields changes over the course of the sequence. Also note that in the table I have emphasized the test of coil changing (one coil changes in each step).

To test out the information in Table 18.5, I created the circuit shown in Fig. 18.15 with the bill of materials listed in Table 18.6 to drive a bipolar stepper motor and soldered it to a prototyping PCB. With the stepper motor that I used, the connector used

TABLE 18.5 HALF-STEP COIL ENERGIZATION PATTERN FOR A BIPOLAR STEPPER MOTOR		
STEP	UP-DOWN COIL	EAST-WEST COIL
1	South	Off
2	South	South
3	Off	South
4	North	South
5	North	Off
6	North	North
7	Off	North
8	South	North

with to it attached to a double in-line connector that could be plugged into the breadboard (similar to the ones used in the servo experiments shown later in this chapter). Chances are that you will not be so lucky, and you will have to solder the individual pins to a single-pin in-line header that can be plugged into the breadboard.

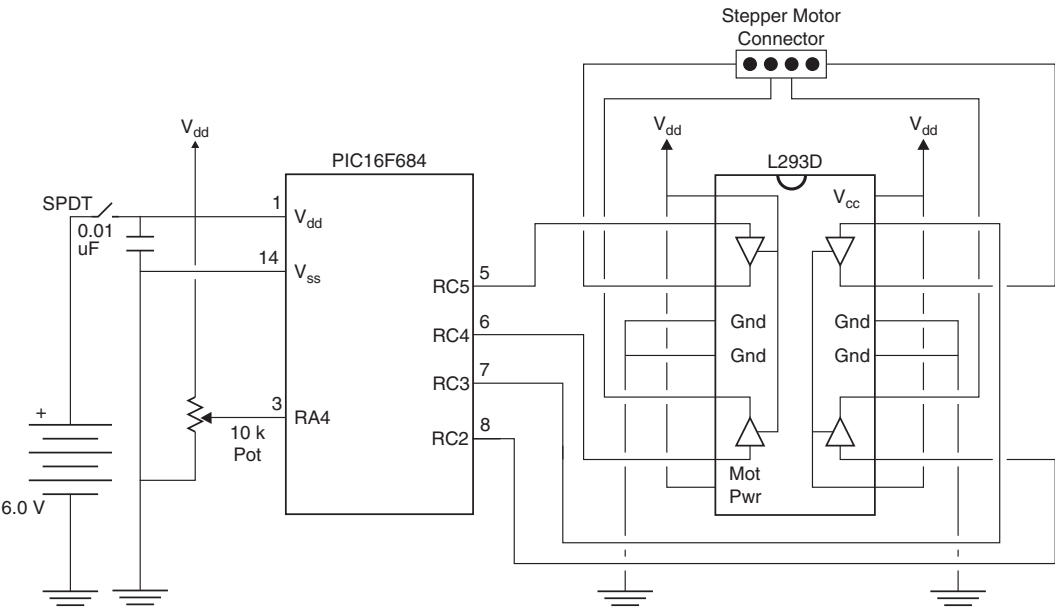


Figure 18.15 An L293D is used to provide four half-bridges to control the movement of the bipolar stepper motor.

TABLE 18.6 BILL OF MATERIALS FOR BIPOLAR STEPPER MOTOR CIRCUIT

PART	DESCRIPTION
PIC16F684	IC16F684-04/P
L293D	L293D motor-driver chip
10-k Ω pot	10-k Ω single-turn potentiometer
100- Ω	100 Ω , 1/4 W resistor
0.01- μ F	0.01- μ F capacitor, any type
SPDT	PCB-mountable single-pole double-throw switch
Stepper motor	Bipolar stepper motor (see text)
Misc.	Prototyping PCB, 4 $\times$ "AA" battery clip, four 0.100-in terminal block (see text), cardboard arrow (see text)

Before burning the PIC16F684 with the following software, I again want to suggest that you cut a sliver of cardboard as a pointer and Krazy glue it to the end of the stepper motor's output shaft so that you can clearly observe the movement of the stepper motor.

When the circuit is built, you can burn a PIC16F684 with `cStepper.c`, which takes the information from Table 18.5 and uses it to create a simple table for half-step driving the bipolar stepper motor. In between steps, there is a quarter-second delay, and if your application is wired correctly, you will see the pointer you glued to the stepper motor shaft turning through 360 degrees (a degree or so at a time). If you do not see this pattern, then you will have to rearrange the wires on the terminal block until the motor starts working correctly.

```
#include <pic.h>
/*  cStepper.c - Turn a Stepper Motor

Hardware Notes:
PIC16F684 Running at 4 MHz with Internal Oscillator
RC5:RC2 - Stepper Motor Outputs

myke predko
05.01.15

*/

__CONFIG(INTIO & WDTDIS & PWRTEN & MCLRDIS & UNPROTECT \
& UNPROTECT & BORDIS & IESODIS & FCMDIS);

unsigned int i = 0, j;
```

```

const char StepperTable[] = {0b011100, 0b010100, 0b000100, 0b100100,
                             0b100000, 0b101000, 0b111000, 0b011000};

main()
{
    PORTC = 0;
    CMCON0 = 7;                // Turn off Comparators
    ANSEL = 0;                 // Turn off ADC
    TRISC = 0b000011;         // RC5:RC2 Outputs

    while(1 == 1)              // Loop Forever
    {
        NOP();
        for (j = 0; j < 21000; j++);
        NOP();

        PORTC = StepperTable[i];

        i = (i + 1) % 8;

    } // elihw
} // End cStepper

```

R/C Servo Control

Servos designed for use in radio-controlled airplanes, cars, and boats can be interfaced easily to a PIC microcontroller. They are often used for robots and applications where simple mechanical movement is required. The output of an R/C servo is usually a wheel that can be rotated from 0 to 90 degrees. (There are also servos available that can turn from 0 to 180 degrees, as well as servos with very high torque outputs for special applications.) Typically, they only require +5 V, ground, and an input signal.

An R/C servo is an analog device that takes a varying length pulse, times it, and moves the output to the specified position. For most RC servos, the pulse (shown in Fig. 18.16) is between 1.0 and 2.0 ms long and repeats every 20 ms. The position of the servo is determined by the time between 1.0 and 2.0 ms; the longer the pulse, the further the shaft will turn.

While the PIC microcontroller's PWM is capable of outputting the correct pulse signal for an R/C servo, it will be unlikely that this will be accurate enough for precisely positioning the shaft. It is good enough for mobile robot applications, where the servo is modified to move in either direction indefinitely, and does not use a PWM signal, controlling a servo could be considered very easy, although the TMR2 output probably will not give you the positional accuracy that you will want.

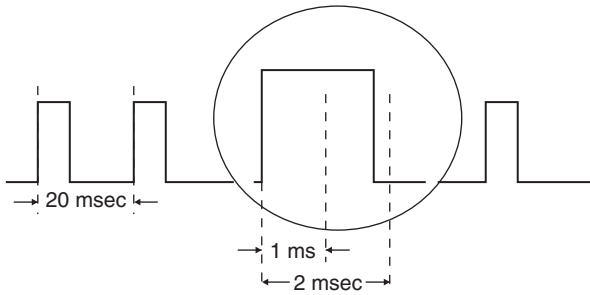


Figure 18.16 The length of the pulse sent to an R/C servo determines the position of the output shaft.

For producing a PWM signal using a PIC microcontroller, I normally use a timer interrupt (set every 20 ms) that outputs a 1.0- to 2.0-ms PWM signal using the following pseudocode:

```
Interrupt() {                                     // Interrupt Handler Code

int i = 0;

    BitOutput( Servo, 1);                         // Output the Signal

    for (i = 0; i < (1 msec + ServoDlay); i++ );

    BitOutput( Servo, 2);

    for (; i < 2 msec; i++ );                     // Delay full 2 msecs

} // End Interrupt Handler
```

This code can be expanded easily to control more than one servo (by adding more output lines and `ServoDlay` variables). This method of controlling servos is also nice because the `ServoDlay` variables can be updated without affecting the operation of the interrupt handler. The interrupt handler takes 2 ms out of every 20 ms. This means that there is a 10 percent cycle overhead for providing the PWM function (and this doesn't change even if more servo outputs are added to the device).

To demonstrate how a servo operates, I created the simple application `asmServo.asm` that reads the voltage from a potentiometer voltage divider and converts this value into a pulse width that is then sent to the servo. Figure 18.17 is the circuit's schematic, and Table 18.7 lists the bill of materials for the application.

In this experiment, I use a potentiometer and the PIC16F684's ADC to specify the position of the servo. The value returned from the potentiometer is displayed on 8 of the 10 LEDs built into an LED display bargraph. This function was originally put in to debug the ADC operation, but I left it in because I liked seeing the LEDs move with the servo.


```
; Hardware Notes:
; PIC16F684 running at 4 MHz Using the Internal Clock
; RA4 - Pot Command
; RA5 - Servo Connection
#define ServoPin PORTA, 5
; RC4:RC0 - Bits 7:3 of LED Output
; RA2:RA0 - Bits 2:0 of LED Output
;
;
; Myke Predko
; 04.12.26
;
LIST R=DEC
INCLUDE "p16f684.inc"

__CONFIG _FCMEN_OFF & _IESO_OFF & _BOD_OFF & _CPD_OFF &
__CP_OFF & _MCLRE_OFF & _PWRTE_ON & _WDT_OFF & _INTOSCIO
; Variables
CBLOCK 0x020
Temp, Dlay
ServoCount
ServoState
ENDC

PAGE
; Mainline

org      0

nop                                ; For ICD Debug

clrf     PORTA
clrf     PORTC
movlw    7                        ; Turn off Comparators
movwf    CMCON0
movlw    b'00001101'             ; Enable ADC on RA4
movwf    ADCON0

bsf      STATUS, RP0
movlw    0xD1                    ; Enable TMR0 with 4x Prescaler
movwf    OPTION_REG ^ 0x80
movlw    1 << 3                  ; RA4 (AN3) ADC Input
movwf    ANSEL ^ 0x80
movlw    b'00010000'             ; Select ADC Clock as Fosc/8
movwf    ADCON1 ^ 0x80
movlw    b'011000'               ; RA4/RA3 As Inputs
```



```

movwf    TRISA ^ 0x80
clrfs    TRISC ^ 0x80          ; All PORTC Outputs
bcf      STATUS, RP0

clrfs    ServoState           ; Use Simple Servo State M/C

movlw    0x80;                ; Start with Servo in Middle
movwf    ServoCount

movlw    HIGH ((20000 / 5) + 256)
movwf    Delay
movlw    LOW ((20000 / 5) + 256)
addlw    -1                    ; Wait for ADC Input to be Valid
btfsc    STATUS, Z
    decfsz Delay, f
    goto  $ - 3                ; 5 Cycle Delay Loop for 20 ms

Loop:
    bsf    ServoPin            ; Output a Servo Signal
    clrfs    TMR0
    bcf     INTCON, T0IF       ; Wait for Overflow
    bsf     INTCON, T0IE
    btfsc   ServoState, 0      ; Calculate Value in 1 ms Pulse
    goto    ReadADC
StartADC:
    bsf     ADCON0, GO         ; Start ADC
    bsf     ServoState, 0
    goto    ADCDone
ReadADC:
    ; Read ADC Value
    movf    ADRESH, w
    movwf   ServoCount
    andlw   b'00000111'       ; Display the ADC Value
    iorlw   1 << 5             ; Make Sure ServoPin Stays High
    movwf   PORTA
    rlf     ServoCount, w      ; Need Top 5 Bits
    movwf   Temp
    swapf   Temp, w
    andlw   0x0F
    btfsc   STATUS, C
    iorlw   0x10               ; Add Top Bit
    movwf   PORTC
    bcf     ServoState, 0      ; Repeat
ADCDone:
    movf    ServoCount, w      ; Get Read with Servo Value
    btfsc   STATUS, Z
    movlw   1                  ; If Zero, Make 1
    sublw   0                  ; Take it away from 256

```

```
    btfss    INTCON, T0IF
    goto     $ - 1
    movwf    TMR0
    bcf      INTCON, T0IF          ; Wait for Overflow
    movf     ServoCount, w        ; Repeat to get 2 ms Delay
    btfss    INTCON, T0IF
    goto     $ - 1
    bcf      ServoPin              ; Finished with the Servo
    movwf    TMR0
    bcf      INTCON, T0IF          ; Wait for Overflow
    btfss    INTCON, T0IF
    goto     $ - 1

    movlw    HIGH ((18000 / 5) + 256)
    movwf    Dlay
    movlw    LOW ((18000 / 5) + 256)
    addlw    -1                    ; Want 20 ms Loop
    btfsc    STATUS, Z
    decfsz   Dlay, f
    goto     $ - 3                ; 5 Cycle Delay Loop for 20 ms
    goto     Loop                 ; Repeat

end
```

PRACTICAL PC INTERFACING

Chances are that the device you are most interested in interfacing the PIC[®] microcontroller to is the PC. Unfortunately, this is also the most difficult device that you probably will try to interface to owing to the hardware complexities of modern systems and understanding the different paths data takes to get from one part of the PC to another. Further complicating the task of interfacing the PIC microcontroller to a PC is the sophistication of the Microsoft Windows operating systems now in use, with their high degree of separation between hardware and software that requires a very high level of education to develop hardware interfaces and the device drivers needed to access hardware within the PC.

When people ask me about interfacing to a PC using the PIC microcontroller, I generally recommend that just the RS-232 serial ports be used as a connection between the PC and the PIC microcontroller. As I will discuss later in this chapter, there are some very interesting capabilities that you can exploit (including using the software interface with the USB port). The serial port interface is well supported by terminal emulators such as Hilgreave's HyperTerminal, and basic programming interfaces can access the serial ports directly by a number of different methods.

When designing all port interfaces, I recommend developing the interface code to pass data in a "human readable" format and use standard "text" data transfers rather than compressed data. If this is serial data, it will allow a terminal emulator such as HyperTerminal to be used when you are debugging the PIC microcontroller interface—allowing the PC application to be debugged separately, after the PIC microcontroller interface, which eliminates the two variables when something doesn't work. The time penalty is actually quite small for the text transfers and is more than offset by the ease with which applications can be debugged.

The Universal Serial Bus ("USB") is the PC (and Macintosh) interface that I expect to have the most capabilities with in the future. This interface is dynamically reconfigurable and provides 100 mA at 5 V for peripheral devices. Right now, I believe that there is still some reluctance to take advantage of the USB interface, but as time goes on, I expect there to be more and easier interfaces available for hobbyists and small companies to take advantage of for their products.

There are many ways in which each PC interface can be wired to the PIC microcontroller, and for this reason, this chapter really is more of a theory chapter with information specific to the PC only provided. PIC microcontroller interface application code and circuitry can be found in Chap. 21 of this book.

PC Software Application Development Tools

As I indicated in the introduction to this chapter, the PC is becoming an increasingly complex device, as well as continually insulating the user from the actual hardware and systems running inside it. This trend is also continuing with the latest generation of application software development tools now on the market—these tools tend to separate the developer more and more from the hardware, making developers rely on existing products that may or may not provide them with the functions necessary for their ideas. Another factor is the high cost of high-quality integrated development tools. The entire situation is unfortunate, and I believe that it is a major issue when it comes to students being able to learn about working with hardware, as well as hobbyists and small companies being able to develop applications.

One of the most popular application software development tools currently used in the market is one that hasn't been supported for several years—Visual Basic 6.0. This compiler, with a user-friendly integrated development environment, has some easy-to-work-with serial port interfacing capabilities, and while it is not particularly fast and produces code that is reasonably sized, it is still the choice of hobbyists and small businesses. Microsoft is promoting Visual Studio.Net (even to the point of making student editions of the development tools free for download), but these tools are geared more toward developing web and server applications instead of allowing people to experiment with their own ideas. Copies of Visual Basic 6.0 are available from web outlets such as EBay and in some dusty corners of computer stores—if you can get your hands on a copy, I suggest that you do so. In this book I spend a bit of time explaining the MSComm serial port interface that is available within Visual Basic 6.0.

FREE TOOLS

I realize that the preceding section was somewhat political, but I do believe that the decisions being made in companies such as Microsoft with regard to PC development tools, operating system, and hardware interfaces have been wrong and short-sighted. They're particularly short-sighted because the open-source community has been hard at work providing some very high-quality tools that can be used instead of Visual Studio or some of the other software tool vendors' products for nothing more than the cost of a download. In this section I would like to introduce you to the Cygwin development environment that will allow you to create your own PC applications that can access operating system APIs and allow you to create your own applications.

The Cygwin package can be found on the web at www.cygwin.com. The package is an overlay to the Windows operating system that will give you a Linux command-line prompt from which you can run basic packages like GCC (GNU C Compiler) and use the DDD integrated development environment with a source-code debugger. From the command prompt, you also can run such Unix utilities as `grep`, `kdifff`, and others that are favorites with developers.

Installing Cygwin is something of a chore, and before you do it, you will have to do some research on the total package that you will want to install. At a basic level, you would be looking at installing

- 1** BASH or some other interface shell script
- 2** GCC
- 3** DDD or other GDB-based (GNU Debug) debugging tools
- 4** Editors (many people still use VI)
- 5** Basic Unix utilities (`grep` and others)
- 6** Windows operating system API header files

When the tools are installed, you then would be looking at creating a batch file to start up Cygwin; the one that I use is

```
@ echo off
C:
chdir C:\cygwin\bin

bash --login -i
```

followed by a number of different batch files and scripts for compiling and debugging. For example, the batch file `wingcomp.bat` will create a complete Windows application with a dialog box user interface:

```
windres %1.rc -O coff -o %1.res
gcc -mwindows -mno-cygwin -g -o %1.exe %1.c %1.res -lgdi32
-luser32
```

The first line of the batch file converts the dialog box `.rc` file into a `.res` resource file, which then is compiled and linked along with the source file (or files) into a Windows 32-bit application. Along with the executable, this batch file also provides GDB information that will allow DDD to be used to debug the application code. The `mwindows` parameter is absolutely required—if it is not present, you will end up creating a Linux executable and not a Windows one.

Using the Cygwin tool to develop PC applications should start with creating simple console applications and then moving on to Windows dialog box–based applications. To develop dialog boxes, you will have to create `.rc` files in which each control and its position on the dialog box are specified explicitly—if you are familiar with Visual Studio, then you will feel like this process is like stepping back in time about 15 years—and you’d be right; the development process is not as slick as Visual Studio, and you

probably will be spending more time placing controls than you would in a more modern application development environment.

The upside of using this tool is that you will have the access and capabilities that will make it possible for you to precisely control different aspects of the PC and be able to explore much more of its capabilities than you would if you were to just rely on the current tools.

Lastly, the title of this section (“Free Tools”) is misleading. Open-source tools are not free; they are the result of many thousands of people doing their best to make them better and to keep up with the latest operating system and hardware products. If you are able, please work at learning the tools and help to improve them. If you do not have the skills to support compilers and Linux emulation, please spend some time contributing to the overall packages in terms of header files, library functions, example applications, and instructions to simplify setups for others. I don’t see the next revolution in computing coming from a big company or university; I believe that it will come from a hobbyist or someone with a passion to do things differently—please help us achieve that goal.

Serial Port

The device you are most likely to interface to is the PC, and its serial ports consist of basically the same hardware and BIOS interfaces that were first introduced with the first PC in 1981. Since that time, a 9-pin connector has been specified for the port (in the PC/AT), and one significant hardware upgrade was introduced when the PS/2 was announced. For the most part, the serial port has changed the least of any component in the PC over the past 25-plus years.

Either a male 25- or 9-pin connector is available on the back of the PC for each serial port. These connectors are shown in Fig. 19.1 and are wired according to Table 19.1.

The 9-pin standard was developed originally for the PC/AT because the serial port was put on the same adapter card as the printer port, and there wasn’t enough room for the serial port *and* parallel port to both use 25-pin D-shell connectors. Actually, I prefer the smaller form-factor connector.

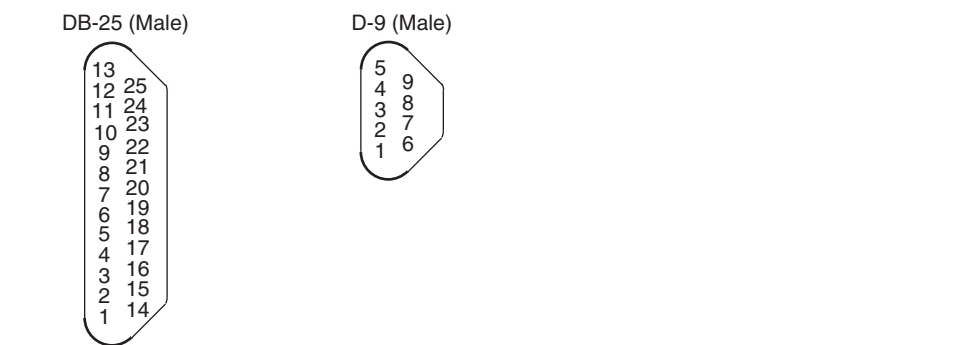


Figure 19.1 The PC normally has a 9-pin RS-232 connector, but some older models may have a 25-pin connector.

TABLE 19.1 THE STANDARD RS-232 CONNECTOR PINOUTS

PIN NAME	25-PIN	9-PIN	I/O DIRECTION
TxD	2	3	Output (O)
RxD	3	2	Input (I)
Gnd	7	5	
RTS	4	7	O
CTS	5	8	I
DTR	20	4	O
DSR	6	6	I
RI	22	9	I
DCD	8	1	I

I feel that the serial ports are the best way to interface a PIC microcontroller to the PC. This is so because the serial port's timing is standard across virtually all PCs, and signal current is very low, which minimizes the chances that a problem with the circuit connected to it will damage the PC. Note that I said *minimizes* the chances that the PC can be damaged—very high voltage inputs can damage the PC's RS-232 interface circuitry or the PC itself. This can be avoided by only using proper RS-232 interfaces on your PIC microcontroller application.

Up to four serial ports can be addressed by the PC, and of these, probably only two will be usable for connecting external devices to the PC. The serial port “base addresses” are listed in Table 19.2, and each base address is used as an initial offset to eight registers that are used by the serial port controller (the 8250). The “interrupt number” is the interrupt vector requested when an interrupt condition is encountered.

The block diagram of the 8250 Universal Asynchronous Receiver/Transmitter (UART) used as the baseline device in the PC is quite simple, and if you were to design your own device, its block diagram probably would look like the 8250's. Figure 19.2 shows the data paths for serial communications. You might want to refer back to this diagram as I explain how the various registers work.

TABLE 19.2 PC SERIAL PORT BASE ADDRESSES

PORT	BASE ADDRESS	INTERRUPT NUMBER
COM1	0x3F8	0xC
COM2	0x2F8	0xB
COM3	0x3E8	0xC
COM4	0x2E8	0xB

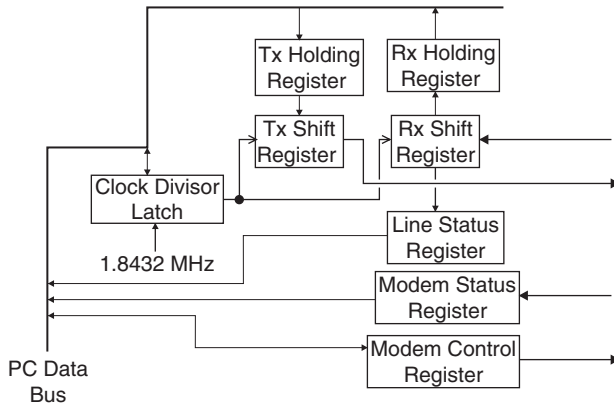


Figure 19.2 Block diagram of the 8250 UART chip used in the original PC and the basis for the hardware used in modern PCs.

When implementing an RS-232 interface, you can make your life easier by doing a few simple things. The first is to simplify the connection. Whenever I do an application, I standardize on using a 9-pin D-shell with the DTE interface (the one that comes out of the PC). As well as standardizing on the DTE connection, I also loop back the DTR/DSR and CTS/RTS data pairs inside the external device rather than at the PC or in the cable. These actions allow me to use a standard PC and cable without having to do any wiring on my own or any modifications. It actually looks a lot more “professional” as well.

Tying DTR/DSR and CTS/RTS also means that I can take advantage of built-in terminal emulators. Virtually all operating systems have a built-in “dumb” terminal emulator that can be used for debugging the external device without requiring the PC code to run. Getting the external device working before debugging the PC application code should simplify the work that you have to do.

The last point is that while I can develop applications that run up to 115,200 bps (by writing a 1 into the two data divisor registers), I typically run at 9,600 or 19,200 bps. By keeping the data rates reasonable, I can run the applications to reasonable lengths (up to about a 1,000 feet with shielded cabling) without requiring special protocols because of bit errors.

A PIC microcontroller application that is appropriate for PC serial communications has the following characteristics:

- 1** A standard PC serial port is to be used.
- 2** Only two computing devices are connected together.
- 3** These two devices may be an arbitrary distance apart (from inches to miles to astronomical distances).
- 4** Relatively small amounts of data need to be transferred in real time (on the order of hundreds of kilobytes per hour).
- 5** Human-readable commands are transmitted between the devices, and data is transmitted using standard protocols.

If an application does not fit all these criteria, then you probably should be looking for another method of communicating between the PC and the PIC microcontroller.

HYPERTERMINAL

Depending on when you bought your PC and the operation system that is running on it, you may or may not have Hilgraeve's HyperTerminal ASCII terminal emulator loaded on to your PC under the Communications pull-down. In this section I want to introduce you to the application and show you its different features as an example of what you should be looking for in a terminal emulator to which you will interface a PIC microcontroller. As I said in the preceding section, I feel that the serial port is the best way to interface to a PIC microcontroller. As part of the interface, I recommend that the communications are to be as "human readable" as possible. This allows the PIC microcontroller application to be tested and debugged separate from the PC application to eliminate as many variables as possible.

In demonstrating PC terminal emulators, I am going to focus on Hilgraeve's HyperTerminal. If you do not have a copy of HyperTerminal or want to upgrade to the latest level, you can download and install HyperTerminal for free for personal use from the Hilgraeve web site: www.hilgraeve.com/.

To configure the terminal emulator, first "disconnect" it (the program "connects" itself automatically) by clicking on the telephone with the receiver off, or click on "Call" and select "Disconnect." Now select "File" and then "Properties" to display the "Properties" dialog box. This dialog box specifies how you would like HyperTerminal to work for you. Click on the "Settings" tab, and make sure that "Terminal Keys," "ANSI," "Emulator," and "500" (line) backspace buffer are selected.

The "Terminal Setup" menu will allow you to define the cursor settings used in the HyperTerminal dialog box. "ASCII setup" will allow you to tailor the data to the application. These parameters are really user- and application-specific. For the most part, I leave these at the default values. Finally, make sure that you have a "direct connection" and that the program does not attempt to access the system through a modem.

Next, click on "Configure," and look at the parameters to set up. For PIC microcontroller applications that connect serially in this book, select "Eight" bits, "None" for parity, "1" stop bit, and "None" for flow control. The data rate ("bits per second") is the value you want to use (usually 1200 bps for the projects presented in this book). Selecting "None" for flow control means that a three-wire RS-232 can be used. Don't worry about "advanced"; it is normally used for specifying the serial port's "first in/first out" data buffers (FIFOs). To be on the safe side, select "No FIFO operation."

Finally, click on "View," and select "Font"; select a "Monospace" font such as "Courier New," which is easier to follow than a proportional spaced font when you are sending formatted data between the PC and an external device.

With your setup finished, click on "File" and then "Save As" so that you can save these parameters. The default directory should be "desktop." For the file name, enter something descriptive such as

DIRECT COM 1 — 1200 bps

and click on “SAVE.” If you minimize all the active windows now, you’ll see that a new tab is on your desktop that will bring up the HyperTerminal with the parameters that you just entered.

When HyperTerminal has the desktop’s “Focus” and is “Connected,” when you press a key, the ASCII code for it will be transmitted out of the PC’s serial port. Data coming in on the serial port will be directed to HyperTerminal’s display window.

Files also can be transferred; this is an excellent way to create test cases instead of repeatedly typing them in manually. Doing a “Send File” will send the file exactly as it is saved on disk. “Send Text File” is preferable because it transfers the file’s ASCII data in the same format as if it were typed in, including the “Carriage Return”/“Line Feed” line-end delimiter.

You can use other terminal emulators instead of HyperTerminal, but if you do, you should be selecting an application that has the following capabilities:

- 1** TTY and ANSI terminal emulation
- 2** Varying data rates
- 3** 8–N–1 data format
- 4** “Monospace” fonts
- 5** User-selectable com port access
- 6** User-selectable “handshaking”
- 7** Configuration save
- 8** Text file transfer

This is a pretty basic list of features, but if your terminal emulator provides these capabilities, you will be able to use it to interface to the PIC microcontroller applications presented here, as well as to any that you create on your own.

VISUAL BASIC MSCOMM SERIAL CONTROL

I really like Microsoft’s Visual Basic 6.0 as a quick and dirty Microsoft Windows application-development tool. I find that I can create Visual Basic applications very quickly and can update them as the application gets more complex. It is also an excellent tool for experimenting (although my experiments usually turn into applications). If you are willing to prowl around and look through what kind of extra controls are available, you will find that there is the MSComm serial communications control. This control allows you direct access to the serial ports within the PC and allows you to interface directly with the serial port hardware without having to load in device drivers.

The MSComm control itself is very easy to use; the biggest problem that you will have is trying to figure out how to enable it. When you first load up Visual Basic, you are given a basic number of controls in the Toolbox down the left hand side of the development screen. These controls, as I pointed out earlier in this chapter, are a basic number needed to execute most initial (beginner’s) Visual Basic applications. The basic controls can be expanded with not only the MSComm serial port controls but also with Microsoft file objects, ActiveX, and OLE controls, Kodak Image, Macrovision Shockwave controls along with a lot of other controls and objects that you can use in your application.

For adding the MSComm serial port control to the available selection, you can click on “Project,” followed by “Component” and then “Apply” “Microsoft Comm Control.” With the control added to the toolbox, you can now use MSComm with your applications. The YAP programmer is a fairly complex programmer that was designed to interface only with PC and workstation serial ports to program PIC microcontrollers. When I originally designed the YAP, I designed it for use with a generic terminal emulator. By using MSComm, I was able to come up with a reasonably attractive Windows front end that runs quickly and easily for the application.

To work with the MSComm control, after loading the control onto the toolbox, I place MSComm’s “Telephone” icon on the dialog box, similarly as I would with the timer. When the application is executing, the “Telephone” is invisible to the user.

To initialize the MSComm control, I used the recommended sequence that consists of

- 1** Specify the hardware serial port to be used.
- 2** Set the speed and data format to be used.
- 3** Define the buffer size.
- 4** Open the port and begin to use it.

The code used to perform these functions is placed in the `Form_Load` subroutine, which means that the port is enabled before the primary dialog box is executing.

```
Private Sub Form_Load()
' On Form Load, Setup Serial Port 3 for YAP Programmer

MSComm3.CommPort = 3

MSComm3.Settings = "1200,N,8,1"

MSComm3.InputLen = 0

MSComm3.PortOpen = True

Text1.Text = "Turn on YAP Programmer"

End Sub
```

With the port initialized and executing, I use a 50-ms timer to continually poll the serial port and display data in the “Text” box when it is received:

```
Private Sub Timer1_Timer()
' Interrupt every 50 msec and Read in the Buffer
Dim InputString

InputString = MSComm3.Input

If (InputString <> "") Then
```

```
    If (Text1.Text = "Turn on YAP Programmer") Then
        Text1.Text = "" ' Clear the Display Buffer
    End If
    Text1.Text = Text1.Text + InputString
End If
```

```
End Sub
```

This application code first prompts the user to turn on the programmer, and if it is done when data is received, the “Text” display is cleared, and data is placed in sequence on the display.

For specialized operations (such as selecting Flash versus EPROM control store types), I use `CommandButton` controls, which send data to the YAP via the serial port using the code

```
Private Sub Command1_Click()
' Put the Programmer into "Flash" Mode

Text1.Text = ""

MSComm3.Output = "f" + Chr$(13)
```

```
End Sub
```

With these controls, the YAP can be controlled using the buttons and the mouse with the dialog much more quickly and efficiently (i.e., little chance for error) than if the commands were entered manually by the user. One nice feature of this application is the text box that is continually updated by the timer interrupt routine, showing what is actually happening with the YAP and allowing the user to debug problems very quickly.

Further enhancing the usefulness of the `MSComm` control is the `OnComm` event. This feature is similar to an interrupt because it is requested after specified events in the serial port. The `CommEvent` property contains the reason code for the event. These codes are listed in Table 19.3.

These values can be processed in the `OnComm` event handler like this:

```
Private Sub Object_OnComm()
' Handle Serial Port Events

Select Case Object.CommEvent
    Case comEventBreak ' Handle a "Break" Received
        Beep
    : ' Handle other events
End Select
End Sub
```

To identify the serial port “object,” I have italicized the word *Object* in the `OnComm` event handler to make the label used for the serial port more noticeable.

TABLE 19.3 VISUAL BASIC MSCOMM “ONCOMM” EVENT SUMMARY

COMMEVENT IDENTIFIER	COMMEVENT CODE	DESCRIPTION
comEvSend	1	Specified number of character sent
comEvReceive	2	Specified number of characters received
comEvCTS	3	Change in the CTS line
comEvDSR	4	Change in the DSR line
comEvCD	5	Change in the “carrier detect” line
comEvRing	6	Ring detect is active
comEvEOF	7	“End of file” character received
comEventBreak	1001	Break signal received
comEventFrame	1004	Framing error in incoming data
comEventOverrun	1006	Receive port overrun
comEventRxOver	1008	Receive buffer overflow
comEventRxParity	1009	Parity error in received data
comEventTxFull	1010	Transmit buffer is full
comEventDCB	1011	Unexpected device control block error

For basic OnComm events, to allow the OnComm handler to respond to the problems, the OnComm handler code simply has to be written. If the handler is not present and the error or event takes place, then it simply will be ignored by Visual Basic.

Parallel Port

Several years before the IBM PC first became available, Centronics built and sold printers using a simple parallel bus interface. This bus was used to pass data from a computer to a printer and poll the printer status, waiting until additional characters could be sent. As part of the format, a special connector also was used. This connector format became very popular and was adopted by a number of printer manufacturers and quickly became an industry standard. The Centronics printer port’s advantages were that its hardware could be replicated by using a few simple components, it was relatively fast compared with RS-232 ports, and software could be written easily for it. Today, the parallel port is the first device most people look to

when simple input-output (I/O) expansion must be implemented in the PC. I consider this unfortunate because this port is actually poorly designed for the purpose, and if you are looking for efficient digital input/digital output in the PC, I suggest that a USB DI/DO card should be considered first.

The parallel port itself is very simple; the design used in the PC/AT consists of just seven TTL chips and provides a simple, byte-wide parallel bidirectional interface into the PC. Over the last 20 years, as PCs have gotten more complex, so have their printers. When the PC was first introduced, the standard printer was a relabeled Epson (with an IBM Badge) dot-matrix Centronics-compatible graphics printer that used the parallel port's data and handshaking lines to control the data transfer a byte at a time from the PC.

The early printer interfaces in the PC, after sending a byte, would wait for a handshaking line to indicate that the printer was ready for the next character before sending the next one. As you could imagine, this method was very slow and took up all the PC's cycles in printing a file. As printers have improved, data buffers have been built into them to allow faster data transfers, as well as byte-wide checking of data information. Specialized devices, such as scanners, have been added to the PC's parallel port because of the reasonably high bandwidth that can be obtained using this port. This method of passing data to a printer printing works reasonably well but can be inefficient for large volumes of data, and it can be difficult to create drivers that work under Windows to share the printer port with another device.

To model the parallel port, I usually go right to the base circuit shown in Fig. 19.3. This diagram shows the parallel port connector pin out along with the registers involved with passing data and the appropriate bits for the different functions. The "control

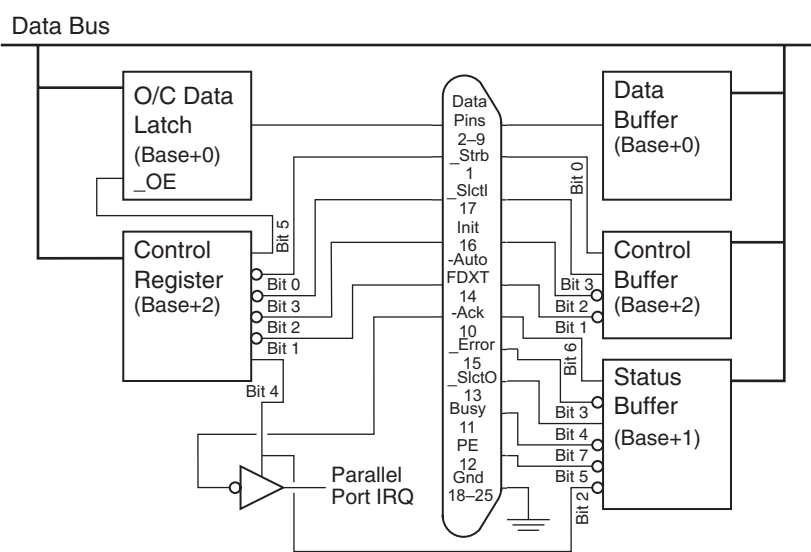


Figure 19.3 The original PC's parallel port was implemented using standard TTL chips.

register” is used to enable the data output latch drivers and enable interrupt requests from the parallel port hardware.

In this figure I have assumed that the parallel port is a 25-pin DB-25 male connector. The true Centronics printer connector is a 36-pin shell, but this shell is connected to the PC’s DB-25M parallel port connector via a female connector and several feet of cable. When developing hardware that interfaces to the PC, I normally use straight-through DB-25M to DB-25F (female) cable or a “printer extension” cable that is a DB-25F to DB-25F. The “straight through” cable normally is used as straight-through serial cables and extensions to the Centronics connector cables. The “printer extension” cable is used for connecting PCs up to a selector box for inexpensively sharing PCs. The advantage of using these types of cables is that the output can be brought from your PC to your bench and not be translated in any way. This is an advantage in applications where the hardware interface will be connected directly to the parallel port on the PC.

If you look at the PC/AT *Technical Reference* manual, you will see that the parallel port is designed with 74LS TTL logic that is capable of driving 20-mA or more loads. This was later changed to provide the data pins with pulled-up open-collector outputs, which is the standard assumed by most bidirectional ports built into the SuperIO chips built into modern PCs. This means that you can only assume that the parallel port, at best, will only be able to source a couple of milliamps or so.

When the parallel port passes data to a printer, the I/O pins create the basic waveform shown in Fig. 19.4, with 0.5- μ s minimum delays between the edges of the waveforms. It is important to note that the printer BIOS routines will not send a new character to the printer until it is no longer busy. When **Busy** is no longer active, the **Ack** line is pulsed active, which can be used to request an interrupt to indicate to data output code that the printer is ready to accept another character.

When interfacing to the parallel port, because the different port pins are seemingly inverted at random, I use a set of functions that I created a number of years ago to eliminate the confusion. These routines change all the input and output lines to being positively active (to simplify trying to figure out what is happening).

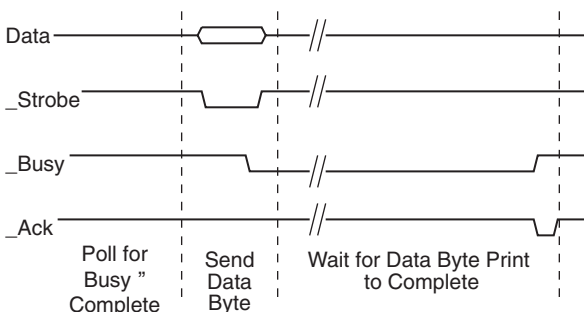


Figure 19.4 The parallel port write waveform showing the data being strobed out and the system waiting for the **_Busy** line to become inactive.

```
PPortOut( int BaseAddr )    // Enable Data Bit Drivers
{

    outp( BaseAddr + 2, inp( BaseAddr + 2 ) & 0x0DF );

} // End PPortOut

PPortIn( int BaseAddr )      // Disable Data Bit Drivers
{

    outp( BaseAddr + 2, inp( BaseAddr + 2 ) | 0x020 );

} // End PPortIn

PPortIRQEn( int BaseAddr ) // Enable the Parallel Ports Interrupt
{
                                // Requesting Hardware

    outp( BaseAddr + 2, inp( BaseAddr + 2 ) | 0x010 );

} // End PPortIRQEn

PPortIRQDis( int BaseAddr ) // Disable the Parallel Ports Interrupt
{
                                // Requesting Hardware

    outp( BaseAddr + 2, inp( BaseAddr + 2 ) & 0x0EF );

} // End PPortIRQDis

PPortHiSLCT( Int BaseAddr ) // Set "SLCT In" (Pin 17) to an
{
                                // Electrical "High"

    outp( BaseAddr + 2, inp( BaseAddr + 2 ) | 0x008 );

} // End PPortHiSLCT

PPortLoSLCT( Int BaseAddr ) // Set "SLCT In" (Pin 17) to an
{
                                // Electrical "low"

    outp( BaseAddr + 2, inp( BaseAddr + 2 ) & 0x0F7 );

} // End PPortLoSLCT

PPortHiInit( Int BaseAddr ) // Set "Init" (Pin 16) to an
{
                                // Electrical "High"
```



```

    outp( BaseAddr + 2, inp( BaseAddr + 2 ) & 0x0FB );

} // End PPortHiInit

PPortLoInit( Int BaseAddr ) // Set "Init" (Pin 16) to an
{                               // Electrical "low"

    outp( BaseAddr + 2, inp( BaseAddr + 2 ) | 0x004 );

} // End PPortLoInit

PPortHiAuto( Int BaseAddr ) // Set "Auto FDXT" (Pin 14) to an
{                               // Electrical "High"

    outp( BaseAddr + 2, inp( BaseAddr + 2 ) & 0x0FD );

} // End PPortHiAuto

PPortLoAuto( Int BaseAddr ) // Set "Auto FDXT" (Pin 14) to an
{                               // Electrical "low"

    outp( BaseAddr + 2, inp( BaseAddr + 2 ) | 0x002 );

} // End PPortLoAuto

PPortHiStrobe( Int BaseAddr )    // Set "Strobe" (Pin 1) to an
{                               // Electrical "High"

    outp( BaseAddr + 2, inp( BaseAddr + 2 ) & 0x0FE );

} // End PPortHiStrobe

PPortLoStrobe( Int BaseAddr )    // Set "Strobe" (Pin 1) to an
{                               // Electrical "low"

    outp( BaseAddr + 2, inp( BaseAddr + 2 ) | 0x001 );

} // End PPortLoStrobe

```

In status bit read routines, note that I have not included reads for bits that are driven in from the port. I assume that the control register latches are good and that the device connected to the port is not holding it high or low. Also, for the status bit read routines, the result returned is either 0 or 1 and inverted, if appropriate.

```
PPortRdBusy( Int BaseAddr ) // Read the "Busy" (Pin 11) handshaking
{
    // line

    return 1 ^ (( inp( BaseAddr + 1 ) & 0x080 ) >> 7 );
} // End PPortRdBusy

PPortRdError( Int BaseAddr ) // Read the "Printer Error" (Pin 12)
{
    // handshaking line

    return 1 ^ (( inp( BaseAddr + 1 ) & 0x020 ) >> 5 );
} // End PPortRdBusy

PPortRdSLCTO( Int BaseAddr ) // Read the "SLCT Out" (Pin 13)
{
    // handshaking line

    return 1 ^ (( inp( BaseAddr + 1 ) & 0x010 ) >> 4 );
} // End PPortRdSLCTO

PPortRdAck( Int BaseAddr ) // Read the "Ack" (Pin 10) handshaking
{
    // line

    return ( inp( BaseAddr + 1 ) & 0x008 ) >> 3;
} // End PPortRdAck
```

With these routines, external digital hardware can be controlled by the 4 output bits driven to the parallel port, and data is either read from or written to, and an additional 4r input pins are available.

Earlier in this chapter I went on at length about the modern operating systems and development tools; you cannot access individual registers, which means that these functions cannot be implemented in a modern system. There are two ways that you can get around this issue. The first is to use an alternative application development system such as Cygwin, which was discussed at the start of this chapter.

Another way is to connect a PSP-equipped PIC microcontroller to the PC's parallel port and print data to the MCU. This method isn't as audacious as it might first appear. From your applications, data can be sent to the printer port device driver. The printer type is defined as part of the system, and if the PIC microcontroller and PSP port were defined as a basic printer, with the basic capabilities of taking in data and pulsing the `_Busy` and `_Ack` lines to simulate the basic printer, you could send data one way to the PIC microcontroller very efficiently.

PIC MICROCONTROLLER APPLICATION

BASICS

I'm an experimenter by nature. As I work through computer and electronic projects, I'll often create small applications to understand a new microcontroller or to help clarify features in devices with which I already am familiar. In this chapter I will be going through many of the different aspects of the PIC[®] microcontroller on an experimental basis to help you to understand how applications execute in the processor and how hardware interacts with the PIC microcontroller. At the end of each of these little applications you will have a piece of code and some knowledge that you can apply in your own applications.

Remember that the three PIC microcontroller families have a great deal of commonality in architectures, instructions and how they execute. In the following sections I will concentrate on the mid-range PIC microcontroller architecture, but much of this code can be used throughout the other two architectures with, in the worst case, only minor modifications. I recommend that you keep this commonality in mind when you are looking at other people's code and applications; something written for the low end often can be ported directly to the high end, and vice versa.

Jumping Around

Execution change is probably an area that you didn't expect that you would have to learn with the PIC microcontroller. In most processors, the jump instructions and their operation are quite straightforward and usually do not require the additional support of the PIC microcontroller. Most processors have conditional jump instructions and jump instructions that can execute anywhere in the processor's instruction space. The PIC microcontroller instructions work at a much lower level than those of many other processors. From the perspective of conditional jumping, there are no specific instructions for executing a jump based on STATUS register contents, but there is the mechanism to

conditionally execute based on the state of any bit in the PIC microcontroller. The PIC microcontroller's capabilities are actually quite a bit more powerful and flexible than what you would get with a processor with conditional jumps built in. The problem with them is that they are somewhat more difficult to learn to use effectively.

The requirement for page bit and register setup before interpage jumping is really not that unusual for processors. For example, the 8088, used in the IBM PC, has short, intrasegment, and intersegment jumps, each one giving the application programmer different options for changing execution to a different location anywhere in the system's memory. In the PIC microcontroller, writing jumps that go across page boundaries and require changes to PCLATH and the STATUS registers (depending on the processor architecture) are not that difficult, but there are some rules that should be followed when doing them.

I must point out that the PIC18 architecture does have the ability to jump anywhere in its program memory space and does have some conditional jumps. These instructions are unique to this architecture, but remember that the basic conditional execution instructions and methodology used in the other architectures are implemented in the PIC18, and code written for lower-level devices should work in a PIC18 processor with little, if any, modification.

LowGoto: LOW-END JUMPING BETWEEN PROGRAM MEMORY PAGES

Low-end devices do not have a PCLATH register, but the page selection bits are in the STATUS register and have to be set or reset according to the page in which the address being jumped to is located. Changing the STATUS page selection bits in low-end devices is really only slightly different from doing it in mid-range's PCLATH register writes, with the majority of the issues being in how the subroutine and table calls work.

The LowGoto application can be found in the `code\LowGoto` folder and is very simple, with only seven instructions executed to jump within the initial page (page 0) and then to jump to and from page 1.

```
title "LowGoto - Low-End Jumping Around."
;
; In this Application, Jumps Between Device Pages is
; Demonstrated.
;
;
; 99.12.25 - Created for the Second Edition
;
; Myke Predko
;
LIST R=DEC
INCLUDE "p16c5x.inc"      ; <- Note PIC microcontroller is PIC16C56

; Registers
```

```

; Macros
MyLGoto MACRO Label
    if ((Label & 0x0200) != 0)
        bsf    STATUS, PA0
    else
        bcf    STATUS, PA0
    endif
    if ((Label & 0x0400) != 0)
        bsf    STATUS, PA1
    else
        bcf    STATUS, PA1
    endif
    goto    Label & 0x01FF      ; Jump to Label Without Page Selections
endm

__CONFIG _CP_OFF & _WDT_OFF & _XT_OSC

PAGE
; Mainline of LowGoto

org      0

    goto    Page0Label        ; Goto an Address Within the Same Page

Page0Label:                    ; Label in Page 0

    MyLGoto Page1Label

org      0x0200
Page1Label:                    ; Label in Page 1
    lgoto   Page0Label

end

```

The 16C56 is a low-end PIC microcontroller with 1,024 instructions of EPROM program memory. This is actually two low-end instruction pages and works well for this experiment, as well as for later ones discussing how interpage calls and program counter updates work in low-end devices.

The first thing to note about this experiment is what happens when you click on the “PIC Microcontroller Reset” icon in MPLAB. If you have set up your `LowGoto` project as I have shown so far, you’ll see that the black highlight line is displayed on the first `goto Page0Label`. This is actually what you expect based on your experience with the 16F84 and other mid-range devices.

Even though this is what you are used to, it is not 100 percent correct. To show what I mean, add the stopwatch to the application by clicking on “Window” and then “Stopwatch.” Notice that after clicking on the “Reset Processor” icon on the MPLAB

toolbar, the number of cycles shown executed is 1. This shouldn't seem right because before the first instruction is executed, the timer should be 0.

The reason for this discrepancy is how reset works in the low-end PIC microcontroller processors and how I have discussed how applications should be programmed for them. In low-end devices, the reset vector is always the last address of program memory, not the first, as in the other PIC microcontroller processor families. To make the devices “appear” more common to the other processor families (with the mid-range family in particular), ignore the last instruction as to when all the bits are set; it is `xorlw 0xFF`, which inverts the bits in the `w` register. This is not an issue because the value in `w` register is undefined at power-up anyway.

For the two jumps, the `MyLGoto` macro sets the PA0 and PA1 bits of the STATUS register according to where the destination label is located. The PA0 bit is set according to the state of bit 9 of the destination address, and PA1 is set according to the state of bit 10 of the destination address. In the `MyLGoto` macro, instead of setting or resetting these bits according to the label address, I could have used the following code:

```
movlw  (1 << PA0) | (1 << PA1)
andlw  STATUS, w          ; Clear All the Bits But PA0 and PA1
iorlw  ((HIGH Label) & 0x006) << 4
movwf  STATUS
goto   (Label & 0x01FF) | ($ & 0x01800)
; Jump to the Address within the Bank
```

which loads in the contents of the STATUS register, clears PA0 and PA1, and then loads them with the value directly from label. After PA0 and PA1 are set, I then jump to the address within the instruction bank. This method takes twice as many instructions as the method that I used to no advantage over the `MyLGoto` macro. Actually, I would also say that the method shown above is suboptimal to the macro's version because it is also much harder to understand, and it changes the `w` register (whereas the macro doesn't).

Using the Microchip-supplied `lgoto` directive is something that you may want to consider in your applications when you are definitely jumping between banks. The `lcall` pseudo-instruction is *not* recommended for use at all because it does not restore the state of the PA0 and PA1 bits (or PCLATH in mid-range devices) after return. I should point out that the `lgoto` pseudoinstruction should not be used after a conditional `skip` instruction because it adds extra instructions (which are not “seen” in the source code) that will affect the operation of `skip`. Ideally, conditional branching should not take place across pages.

MidGoto: MID-RANGE JUMPING BETWEEN PROGRAM MEMORY PAGES

In mid-range PIC microcontrollers, jumping between the pages is accomplished by setting the PCLATH register before executing a `goto` or `call` instruction or changing the PIC microcontroller's program counter via the PCL register. This feature may

seem intimidating when you first start working with it, but as you gain experience with the PIC microcontroller, it actually will be quite easy to work through.

To show how jumping between program memory pages is accomplished, I created the MidGoto application, which can be found in the code\MidGoto folder:

```

title "MidGoto - Low-End Jumping Around."
;
; In this Application, Jumps Between Device Pages is
; Demonstrated.
;
;
; 99.12.25 - Created for the Second Edition
;
; Myke Predko
;
LIST R=DEC
INCLUDE "p16c73b.inc"          ; ← Note the Changed Processor

; Registers

; Macros
MyLGoto MACRO Label
    movlw HIGH Label
    movwf PCLATH
    goto Label & 0x07FF        ; Jump to Label Without Page
Selections
endm

__CONFIG _CP_OFF & _WDT_OFF & _XT_OSC & _BODEN_OFF & _PWRTE_ON

PAGE
; Mainline of LowGoto

org      0

    goto    Page0Label        ; Goto an Address Within the Same Page

Page0Label:                    ; Label in Page 0

    MyLGoto Page1Label

org      0x0800
Page1Label:                    ; Label in Page 1
    MyLGoto Page0Label

end

```

This application first jumps to a label within the current page (0) and then does an interpage jump using the `MyLGoto` macro. This macro sets up the `PCLATH` register before executing the `goto` instruction. To demonstrate the interpage jumps, I wanted to use a PIC microcontroller with more than one code page. One of the most basic devices for doing this is the PIC16C73B, which is a 28-pin mid-range PIC microcontroller with a number of ADC inputs.

In mid-range (and higher-end) PIC microcontrollers, the lower 8 bits of the program counter are available in `PCL`, whereas the upper bits use a `PCLATH` (and optionally a `PCLATU` in the PIC18 microcontrollers) register. When a `goto` or `call` instruction is executed or `PCL` is updated, the contents of `PCLATH` are loaded into the high bits of the PIC microcontroller's program counter along with the new address.

To demonstrate how this works, after you have simulated the application and watched the `PCLATH` register change in the 16C73B “Watch” window, comment out the line that updates `PCLATH` (`movwf PCLATH`) in the `MyLGoto` macro, rebuild, and then step through the code. This macro differs from the one demonstrated in the preceding section in that it changes the value of the `w` register—a similar, conditionally executing macro that just changes the appropriate bits of `PCLATH` also could be created.

The `MyLGoto` macro will allow the jump between pages to execute correctly, but there are a few caveats that you should be aware of. If you want to have interpage `call` instructions, check in the experiments for my comments on this. Second, this macro *never* should be used with conditional `skip` instructions. As I will note in the next experiment, only nonconditional jumps should be done between pages as you are learning the PIC microcontroller. The third issue is that you will get a message if you are jumping from a page in which a `PCLATH` page selection bit is set to an address where the page selection bit is reset. This isn't a big problem—by changing the macro `goto` instruction to

```
goto    (Label & 0x07FF) | ($ & 0x01800)
```

you will add the appropriate high level bits for the current page to the `goto` instruction, eliminating the message.

CondJump: **CONDITIONAL JUMPING**

Conditional jumping in the PIC microcontroller is something that probably won't seem all that intuitive, but once you understand how it can be used not only for executing according to the state of the `STATUS` register bits but also for *all* the register bits in the processor, you will begin to see how you can write applications for the PIC microcontroller that are a lot more efficient than in other chips. Instead of having instructions that are based on the state of one of the execution `STATUS` bits (zero, carry, and digit carry), the conditional instructions are based on the state of any bit accessible to the processor. This is quite a profound method of operation and one that you really have to sit down and think about because it can allow you to create applications that are startling in their efficiency and their ability to work through complex comparisons.

Elsewhere in this book I have described the “skip on bit condition” instructions, and I want to take this opportunity to demonstrate their use in a few cases, along with discussing

how operations can be written to avoid taking up extra cycles and instructions for operations that just require one instruction. The three conditional cases in `CondJump.asm` may seem to be quite limited in their scope, but used with the STATUS register experiments presented earlier in this chapter and used with flags, they are really all you have to be aware of for most of your applications.

When I created the `CondTest` application, it was created as an assembly-language version of the C application:

```

Main()
{

    int  i, j, k;
    bit  TestFlag;

    i = 5;                // Initialize the Variables
    j = 7;
    TestFlag = 0;

    if (I < j)             // Set TestFlag Based on "i" and "j"
        TestFlag = 1;

    if (TestFlag == 1)     // If "TestFlag" set, Jump
        goto LongJump;

    while (1 == 1);        // Loop Forever

    :

LongJump:                // Label in another Page

    for (k = 0; k < j; k++ )    // Repeatedly increment "I"
        i = i + 1;

    while (1 == 1);        // Loop Forever

}

```

The `CondJump` code demonstrates an 8-bit comparison, a bit test, and a counting loop that you will use in your own applications. The source code can be found in the `code\CondJump` folder. Note that the code is written for a PIC16C73B, which has more than one page of instructions. This was done to show you how to implement inter-page conditional jumps.

```

title  "CondJump - Conditional Jumping"
;
; This application shows how conditional jumping can be
; implemented in a variety of different situations,

```

```
; including interpage jumps.
;
; Myke Predko
; 99.12.29
;
; Hardware Notes:
; Simulated 16C73B
; No I/O
;
LIST R=DEC
INCLUDE "p16c73b.inc"           ; ← Note PIC16C74B

; Registers
CBLOCK 0x020
Flags
i, j
ENDC

#define TestFlag Flags, 0        ; Define a File Register Flag

__CONFIG __CP_OFF & __WDT_OFF & __RC_OSC

PAGE
; Mainline of CondJump

org 0

clrf  Flags                    ; No Flag is Set

movlw 5                        ; Setup Test Variables
movwf i
movlw 7
movwf j

; if (i < j)
; TestFlag = 1;

movf j, w                      ; Subtract "j" from "i"
subwf i, w                     ; And Look at Carry Result
btfsc STATUS, C
goto $ + 2
bsf TestFlag

; if (TestFlag == 1)
; goto LongJump

movlw HIGH LongJump            ; Set up PCLATH for the Long Jump
movwf PCLATH
```

```

    btfsc TestFlag
    goto LongJump & 0x07FF
    movlw HIGH $                ; Restore PCLATH to the Current
    movwf PCLATH                ; Page

    goto $                      ; Endless Loop if TestFlag == 0

org    0x0800
LongJump:

; for (j = 0; j < 7; j++ )
;   i = i + 1;

    incf i, f                    ; Increment "i", "j" Times
    decfsz j, f
    goto $ - 2

    goto $

end

```

To explain the code, I want to work through each test individually, looking back to the original C source and comparing it with the source code just presented.

The variable initialization should not be of any surprise to you. In this code, I load `i` with 5 and `j` with 7 and clear the `FLAGS` register, clearing `TestFlag` at the same time.

The first comparison is executing the next instruction if `i` is less than `j`. This comparison was discussed in previous experiments as well as the text and has the basic form

```

    movf    Parameter1, w
    subwf   Parameter2, w
    btfss|c STATUS, Z|C          ; "Test" in Table Below
    goto    NotTrue
; Code Executed if Condition is "True"

```

NotTrue:

With `Parameter1`, `Parameter2`, and whether or not the carry or zero flag is specified, the following conditional execution code can be implemented according to Table 20.1.

In `CondJump`, I followed this format instruction explicitly, but I could have simplified it to

```

movf    j, w                    ; Subtract "j" from "i"
subwf   i, w                    ; And Look at Carry Result
btfss   STATUS, C
    bsf   TestFlag

```

TABLE 20.1 THE DIFFERENT PARAMETERS FOR THE FOUR-INSTRUCTION CONDITIONAL EXECUTION TEST			
CONDITION	Parameter1	Parameter2	TEST INSTRUCTION
A == B	A	B	btfss STATUS, Z
A != B	A	B	btfsc STATUS, Z
A > B	A	B	btfsc STATUS, C
A >= B	B	A	btfss STATUS, C
A < B	B	A	btfsc STATUS, C
A <= B	A	B	btfss STATUS, C

because there is only one instruction that is executed conditionally, and the jump over the code that executes if the condition is true consists of one instruction that can be handled by the skip on bit state instructions.

The next conditional jump is interesting for two reasons. The first is that it is jumping to another page, and the second is that the jump is based on the condition of the flag. In a traditional processor, a “jump on flag” condition probably would involve the operations

```
if (((Flag & (1 << Bit)) != 0)
    goto Label;
```

which, in PIC microcontroller assembler, would look like

```
movf    Flags, w
andlw   1 << Bit
btfsc   STATUS, Z
goto    Label
```

and is not that efficient compared with the code that can be generated. The PIC microcontroller itself can access the bits in the processor, so the test on the bit can be done in one instruction and not by ANDing the test register with a mask value:

```
btfsc   TestFlag
goto    Label
```

which is much simpler and faster.

This simple bit test and jump (or execute a single conditional instruction) also can be applied to the various hardware register bits in the PIC microcontroller. In fact, when one of the processor STATUS condition bits is tested and a jump made from its state, this is exactly what is happening. By eliminating the need to isolate a bit, as I show in the inefficient code above, the PIC microcontroller can carry out some very fast and efficient operations.

Interpage conditional jumps are somewhat difficult to conceptualize. While I recommend avoiding them as much as possible, sometimes they have to be done. The format that I have used here, where I initialize PCLATH before the test and jump and then reset it after the test and jump, seems to be the simplest way of doing it. Because `movlw` and `movwf` do not change the processor STATUS registers, they can be used before the conditional skip instruction.

For the comparison operation that is the first conditional jump, the code could be changed to

```

movf      Parameter1, w
subwf     Parameter2, w
movlw     HIGH PastTrue
movwf     PCLATH
btfss|c   STATUS, Z|C          ; "Test" in Table 20.1
goto      (NotTrue & 0x07FF) | ($ & 0x01800)
movlw     HIGH $
movwf     PCLATH

; Code Executed if Condition is "True"

```

NotTrue:

if the Code Executed if Condition is "True" straddles a page boundary.

Another way of doing this is to jump around the `goto PastTrue`, as in

```

movf      Parameter1, w
subwf     Parameter2, w
btfsc|s   STATUS, Z|C          ; "Test" in Table Above
goto      $ + 4
movlw     HIGH PastTrue
movwf     PCLATH
goto      NotTrue

; Code Executed if Condition is "True"

```

NotTrue:

But this code only saves one instruction and requires you to think through the negative condition to jump over. Personally, I prefer the first method that doesn't require any negative thinking to work through.

The last conditional operation is the `for` loop at the end of the application. In the PIC microcontroller, this can be done most efficiently with the `decfsz` instruction, as is shown in `CondJump`, rather than with something like

```

clrf      k                      ; k = 0
ForLoop:
movf      j, w                  ; is "k" == "j"?

```

```
    subwf    k, w
    btfsc    STATUS, Z
    goto     ForEnd

; Execute Code Here

    incf     k, f      ; Increment Counter
    goto     ForLoop   ; Repeat the Test
ForEnd:
```

which actually performs the same function but requires quite a few more instructions. To implement a loop for a set number of loops, the number is loaded into a variable, and then the `decfsz` instruction is used until it is equal to 0.

The only advantage I can see of using the preceding format is if the intermediate values of the counter are required. The intermediate value in the `decfsz` for loop can be found using the instruction snippet

```
movlw    LoopNumber    ; Final Value of "k"
subwf    k, w
```

In these two instructions, the current value is subtracted from the number of loops (`LoopNumber`) to be executed, and the result would be the same as `k` in the `for` loop analog code above for a specific loop.

Looking over this application, you probably will feel like it is pretty artificial and doesn't represent conditional execution accurately. This feeling may be reinforced by a quick look through other examples in this chapter and other example code in this book. If you work at understanding exactly what I have been doing, I think that you will discover that the operation of the conditional code follows the examples I've explained here and in other areas of this book. Often the examples will have considerations for other aspects of code that aren't obvious when you look at them for the first time.

StateMC: EXECUTION CONTROL STATE MACHINES

Execution state machines became popular when the first read only programmable memories became available as a method of algorithmically controlling digital devices without using a microprocessor. The external input data values are combined with the current state of the memory to determine what the outputs will be. A block diagram of this is shown in Fig. 20.1. A typical example application for the state machine is controlling traffic lights. The state machine's read only memory is loaded from a table such as Table 20.2. The inputs are listed along with the current state to get expected outputs for different situations.

For a traffic light, I have assumed that there are two timers, 1 second long and 15 seconds long, that are started by the state machine as one of its inputs. The timers are reset by the application, and when they time out, they provide new inputs to the state machine. The timers are started by a rising edge on the RST lines, and when they overflow (reach their delay values), a 1 is output onto the O/F lines. The block diagram of the state machine can be modified to what is shown in Fig. 20.2.

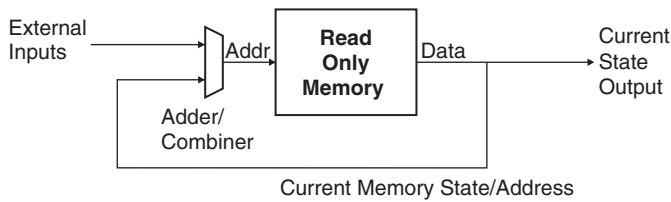


Figure 20.1 A simple hardware state machine implementation.

The table data is then converted to address and corresponding data. In this table, you should be able to see how the address output (which is actually the address input) depends on the state of the timers and advance to the next state when the timers overflow.

Now, as you would expect, the actual hardware and input-output (I/O) states for a state machine are somewhat more complex. The code presented here for the traffic lights only handles them in one direction and does not take into account variables such as cars wanting to turn left or pedestrians pressing a crossing button.

State machines can be implemented in the PIC microcontroller quite simply using tables to decide which routine to jump to next. For example, if you were keying off a file register value (i.e., 0, 1, 2, . . .), you could use the following code:

```

movf      Reg, w           ; Get the Index
addwf     PCL, f           ; Jump to the State Machine Table
                               ; Entry
  
```

TABLE 20.2 TRAFFIC LIGHT STATE MACHINE INPUTS AND OUTPUTS

ADDRESS	1-S O/F	15-S O/F	ADDRESS OUT	1-S RESET	15-S RESET	LIGHTS
Start	x	x	Start + 1	1	0	None
Start + 1	0	0	Start + 1	0	0	None
Start + 1	1	0	Green	0	1	Green
Green	0	0	Green	0	0	Green
Green	0	1	Yellow	1	0	Yellow
Yellow	0	0	Yellow	0	0	Yellow
Yellow	1	0	Red	0	1	Red
Red	0	0	Red	0	0	Red
Red	0	1	Green	0	1	Green

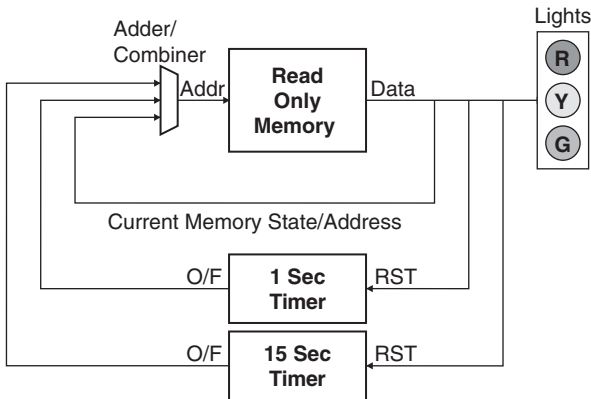


Figure 20.2 Using a state machine to implement a simple traffic light controller.

```

goto    State0                      ; Reg = 0
goto    State1                      ; Reg = 1
goto    State2                      ; Reg = 2

:                                   ; ... And so on ...

```

There are three advantages to using a table in this situation:

- 1** Fewer instructions are required for the decision-making process.
- 2** Each different jump condition takes the same number of cycles. This could be important for some applications.
- 3** The table jumps are a lot easier to understand by looking at just the code.

I define a state machine as a program that uses a single value for executing different functions of the program. This value is set according to the previous state and current environmental conditions (i.e., “if RB0 is set, increment the state and execute the response to RB0 being set”).

State machines are a form of *nonlinear programming*. Nonlinear programming involves different methods of programming when the traditional `if/else/endif` structure of coding is not used or is inappropriate. To demonstrate this in the PIC microcontroller, I created the `StateMC` application that can be found in the `code\StateMC` folder.

```

title  "StateMC - Demonstrating a State Machine"
;
; This Program Demonstrates how a State Machine could work with
; the PIC microcontroller Architecture.
;
; 99.11.14 - Updated for Second Edition
;

```



```

; Myke Predko
; 96.05.14
;
LIST R=DEC
INCLUDE "P16F84.inc"

; Registers
CBLOCK 0x020
i                                ; General Counter
state                           ; Returned Value
Temp                           ; Temporary Storage Variable
ENDC

__CONFIG __CP_OFF & __WDT_OFF & __RC_OSC

PAGE
; Mainline of StateMC

org 0

clrf i                          ; Initialize Variables
clrf state

clrf PORTB                      ; Setup PortB
bsf STATUS, RP0
clrf TRISB ^ 0x080
bcf STATUS, RP0

; Now, Execute the Program

Loop:                            ; Return Here every Execution
    movlw 1                     ; Check the Least Significant Bit
    andwf PORTB, w              ; of PORTB
    movwf Temp
    bcf STATUS, C               ; Now, Shift over the state
    rlf state, w                ; Variable
    addwf Temp, w               ; Add the Least Significant Bit of
; PORTB
    addwf PCL, f                ; Jump to the Correct State
    goto State0                 ; Execution Vector
    goto State0
    goto State10
    goto State11
    goto State2
    goto State2

; State Routines...

```

```
State0:                                ; Increment i to 4
    incf i, f

    movlw 4
    subwf i, w                        ; is "i" greater than 3?
    btfsc STATUS, C
    incf state, f                     ; Yes, Increment the State
; Variable
    goto Loop                        ; Execute the State value again

State10:                               ; Increment the LSB of PORTB if
; it's == 0
    movlw 1
    addwf PORTB, f

    goto Loop

State11:                               ; Shift Over PORTB by one until
; Carry Set
    bcf STATUS, C
    rlf PORTB, f

    btfsc STATUS, C                  ; Is the Carry Set?
    incf state, f                    ; Yes, Go to the Next State

    goto Loop

State2:                               ; Reset Everything and Restart the
    clrf I                           ; Program
    clrf state

    goto Loop

end
```

I realize that `StateMC` is a pretty simple example of a state machine, but it does show how the state is changed with different conditions and the program progresses forward. I realize that this application may not seem so *simple*—especially considering that some of the operations and execution will be quite unexpected. In fact, this can be a problem with state machines—the operation becomes even more confusing as you modify the application over time and see simple changes that can be made. In `StateMC`, an example of this would be how I increment and shift the output value in `PortB` when there are other ways of doing this (that are not quite so complex).

State machines are particularly useful in low-end PIC microcontrollers, where the two-level stack may be a hindrance in traditional programming methods.

Some Basic Functions

Compared with the latest 64-bit processors, the PIC microcontrollers described in this book are extremely simple. Despite this relative simplicity, there are many different ways in which they operate, and there are some quirks to be aware of. In the following sections I want to introduce you to some of these behaviors to make you aware of them as well as give you a better idea of how the PIC microcontroller operates. I'm sure that when you first started working with PIC microcontrollers, you were amazed at their complexity, but as you gain more experience with them, they will start to seem simpler, and you will be able to create applications much faster and with much fewer errors.

CALCULATING CURRENT REQUIREMENTS/CHECKING EXPERIMENTALLY

In an application, if I know the voltage applied to the circuit and the current being drawn by it, I can go back and determine the power being used by the circuit using the formula

$$\text{Power} = V \times I$$

Earlier in the book I stated that I didn't think that my PIC microcontroller current estimations would be very accurate—and that when I was developing the power-supply specification for the application, I derate the calculated current value by 25 to 100 percent. In this experiment I want to check how useful this derating value is and whether or not I can predict accurately how much current the application really requires.

When I look at the PIC16F84 datasheet, I can see that at 4 MHz and the XT oscillator specified, the PIC microcontroller requires a typical intrinsic current of 1.8 mA and a maximum intrinsic current of 4.5 mA. This means that when the light-emitting diode (LED) is off (no current flowing through it), I would expect to see anywhere from 1.8 to 4.5 mA flowing through the circuit.

When the LED is turned on, the current passing through the PIC microcontroller will be increased by the current that is being sunk through the LED. For my typical LED circuits, I assume that the LED has a voltage drop of 0.7 V (the same as any silicon diode) with a maximum current of 20 mA. To provide this current, I have placed a 220- Ω current-limiting resistor in series with the LED.

The 220- Ω resistor was chosen by using Kirchhoff's law, which states that the voltage applied to a circuit is equal to the voltage drops within it. If 5.0 V is applied to the circuit and the LED has a voltage drop of 0.7 V, then the resistor has 4.3 V across it. Knowing that the LED must have a maximum of 20 mA flowing through it, I used Ohm's law to calculate the resistance:

$$\begin{aligned} R &= V/I \\ &= 4.3 \text{ V}/20 \text{ mA} \\ &= 215 \Omega \end{aligned}$$

I used a 220- Ω resistor because that is easily found. The actual current flowing through the LED/resistor and sunk by the PIC microcontroller is then

$$\begin{aligned} I &= V/R \\ &= 4.3 \text{ V}/220 \, \Omega \\ &= 19.5 \text{ mA} \end{aligned}$$

Therefore, when the LED is on, the total current passing through the PIC microcontroller is 1.8 mA typical intrinsic current plus 19.5 mA of LED current. For the power supply for this circuit, I probably would derate this by 50 percent in real life, meaning that I would have to provide a 30-mA, 5-V power supply for this circuit.

I use this value for the total current used by the application when the LED is on. I realize that I am not including the current through the momentary on switch, but this will be 10 μ A (according to Ohm's law) and really doesn't change the total current required by the application in any appreciable manner.

With the application designed and the total current estimated, it is now time to do an empirical check on what the PIC microcontroller actually requires. To do this experiment, I used the same circuit as the previous experiment, except that I broke the ground connection between the PIC microcontroller and ground and wired in my digital multi-meter set on the milliamp reading, as shown in Fig. 20.3 (the bill of materials is shown in Table 20.3). With this circuit, I can now check the actual current drawn by the LED on circuit.

When I first turned on the power to this circuit (LED off), I found that the current passing from the PIC microcontroller to ground was 1.4 mA. This is a bit lower than the typical value quoted by Microchip but only off by 400 μ A. This may have been a

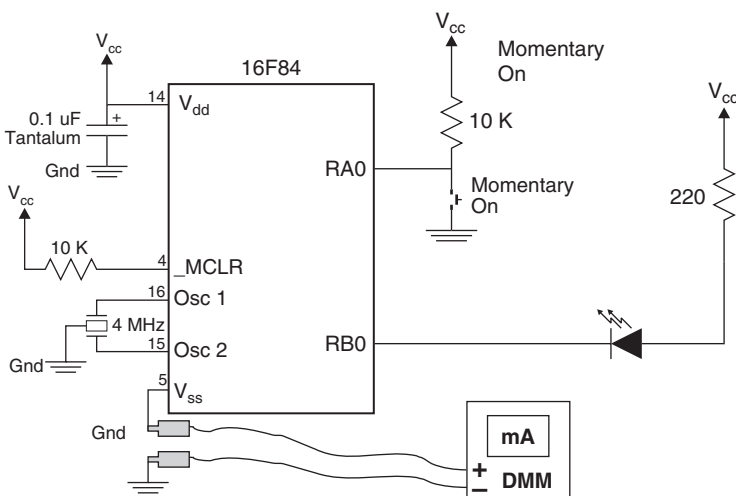


Figure 20.3 Checking the current draw in the simple LED application by measuring the current from V_{ss} to ground.

TABLE 20.3 BILL OF MATERIALS FOR CURRENT-CHECK CIRCUIT

PART	DESCRIPTION
16F85	PIC16F84-04/P
LED	Red LED
1N914	Diode to replace LED (see text)
4 MHz	4-MHz ceramic resonator with internal capacitors
0.1	0.1 Tantalum
10-k	10 k Ω , 1/4 W
220	220 Ω , 1/4 W
Momentary on	Momentary on pushbutton switch
Misc.	Breadboard, wiring, +5-V supply

part variation, but it is still within the minimum derating tolerance I set for PIC microcontroller circuits (which is 25 percent) and at a level that is difficult to measure with most low-cost handheld digital multimeters.

When I turned on the LED, I found that the current jumped to 14.1 mA—which is 28 percent lower than what I calculated above. This was the first time that I had really checked the current drawn by an LED connected to a PIC microcontroller, and I was surprised at the difference—which was much larger than what I expected.

Looking at the operation of the circuit, I found that one of my basic assumptions was incorrect. When I assumed that the LED had a 0.7-V drop, I was wrong. The actual voltage drop across the LED was 2.12 V, and the actual drop across the resistor was 2.6 V, which changes the current calculations dramatically. Going back to Ohm's law for the resistor, the current flowing through the resistor/LED combination is actually 11.8 mA.

Adding 11.8 mA to the observed intrinsic current of 1.4 mA, the total current expected is really 13.2 mA—a difference of only 6 percent from the observed current of 14.1 mA. When I replaced the LED with a 1N914 silicon diode, I found that the actual current from the PIC microcontroller through ground jumped to 18.7 mA, which is very close to my original calculation.

You may be asking yourself why do I still want to derate this value by up to 100 percent? The answer lies in the efficiency and stability of most power-supply circuits. As the load approaches the rated maximum current supplied by a power supply, the power supply may sag or have extra ripple that will affect the performance of the application (i.e., it will be intermittent). As I have said elsewhere, power-supply problems are probably the most difficult to find and fix. Thus, by derating the circuit's current requirements, I have added a safety margin for the power supply that will help it to work properly, even when the application is drawing its full load.

debounce: BUTTON PRESS WITH DEBOUNCE

Depending on the quality of the momentary-on switch you used for power measurement experiment in the preceding section, you may have noticed that the LED flickered while you were pushing down on the switch. This is caused by intermittent contacts being made in the switch. This is one of the reasons why I suggest that you use switches that click when they are pressed and released; the clicking action ensures a good contact and also wipes the switch's contacts clean.

Even with this capability built into the switch, if you were to observe the action of the switch, you probably would see some bouncing, as shown in Fig. 20.4. The jagged edges on the oscilloscope plot are the bounces that happen within the switch when it is pressed. For this experiment, I used the same circuit as in the `1edon` experiment (see Fig. 20.5) and the oscilloscope picture shown in Fig. 20.4 (this picture was taken at RA0). This circuit can be built on a breadboard, as shown in Fig. 20.6.

To eliminate the flickering of the LED, which indicates the switch is bouncing, code must be written to wait for the bouncing to stop for a set amount of time. As I indicated elsewhere in this book, the normal interval of time to wait for the bouncing to stop is 20 ms. This is done by continually polling the switch until it has remained in the same state for 20 ms. This operation is shown in the following pseudocode:

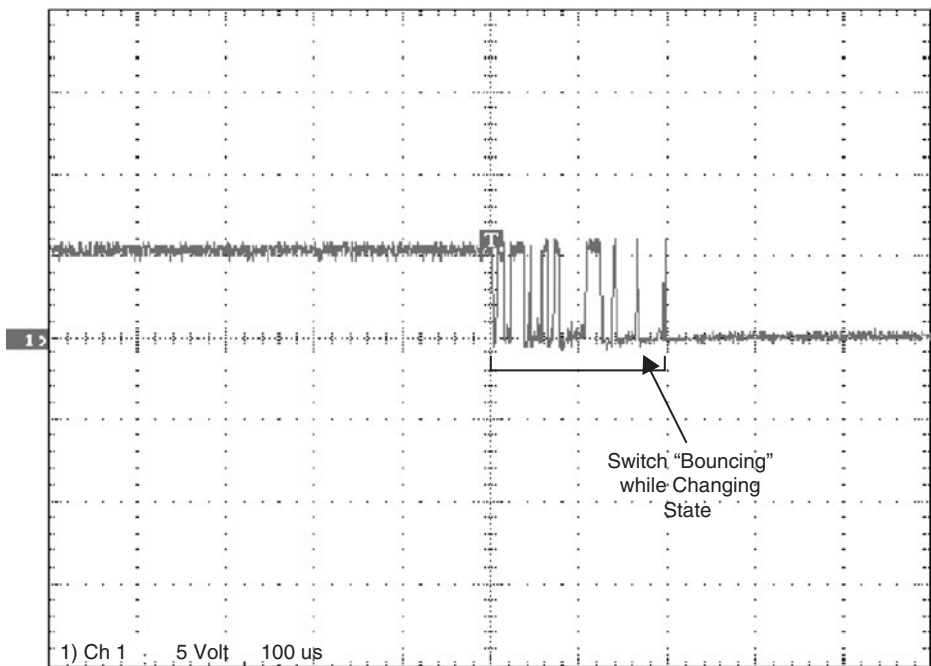


Figure 20.4 Oscilloscope picture of a switch bouncing from a high logic level to a low one.

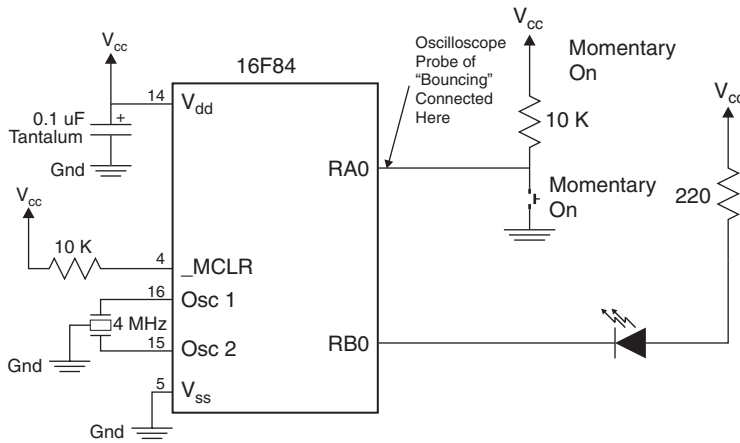


Figure 20.5 Debounce circuit with location of oscilloscope probe shown.

```

Debounce_Down:                // Wait for the Switch to be held down
//   for 20 msec
    while (Switch == Up);      // Wait for the Switch to go down

    for (Dlay = 0; (Switch == Down) && (Dlay < 20msec); Dlay++);
                                // Poll the Switch while Incrementing
                                //   the Delay Timer

    if (Dlay < 20msec)
        goto Debounce_Down;    // If 20 msec has NOT passed, wait for
//   Switch Down and Repeat Process
//   When Execution reaches here, the switch is "Down" and Debounced

```

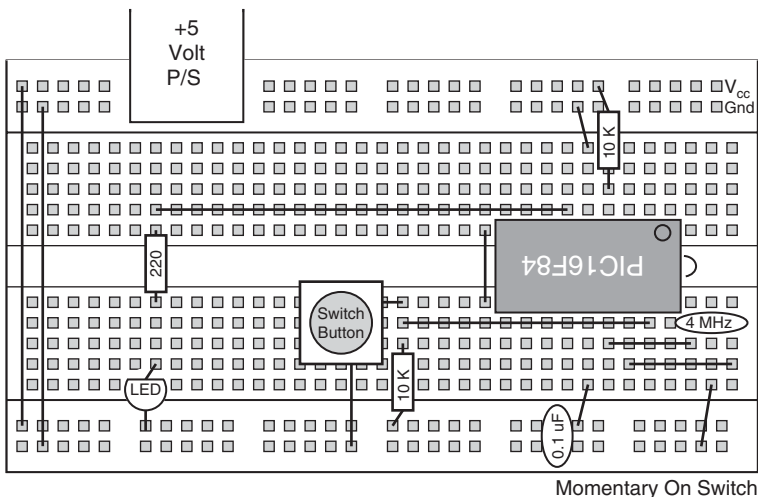


Figure 20.6 Breadboard circuit layout for the debounce circuit.

The actual PIC microcontroller assembly code for a PIC16F84 running at 4 MHz with the button at RA0 is

Debounce_Down:

```
    btfsc  PORTA, 0
    goto   Debounce_Down          ; Wait for Button "Down"

    movlw  0x0100 - 0x0C4          ; Initialize Dlay for a 20 msec
    movwf  Dlay                    ; Delay
    movlw  0x0100 - 0x00A
    movwf  Dlay + 1

    bcf     STATUS, Z              ; Make Sure that Zero is Reset
```

Debounce_Down_Loop:

```
    incfsz Dlay, f
    goto   $ + 2
    incf   Dlay + 1, f
    btfsc  PORTA, 0
    goto   Debounce_Down          ; No - Loop Around Again
    btfss  STATUS, Z              ; Zero Flag Set (20 mSecs Past?)
    goto   Debounce_Down_Loop
```

// When Execution reaches here, the switch is "Down" and Debounced

One aspect of this code that may be confusing is how I came up with the values 0x100–0x0C4 and 0x100–0x00A for the initial values in `Dlay` and how the delay count works in the `Debounce_Down_Loop`. The code in `Debounce_Down_Loop` takes eight instruction cycles to execute—for a 20-ms delay with a 4-MHz clock speed, that means it has to execute 2,500 times.

In decimal, 2,500 is 0x9C4. Because the `incf` instruction is used for the upper byte, I added one cycle to it to make sure that it increments the lower byte 9 times 256 times through and once for 0xC4 (196 decimal) times.

The three instructions for the delay count, i.e.,

```
incfsz Dlay, f
goto   $ + 2
incf   Dlay + 1, f
```

always execute in the same number of cycles and only set the STATUS register's zero flag when the 2,500 loops through `Debounce_Down_Loop` have executed. This code could have been executed differently, specifically using decrements, but I wanted to try something different, and the advantage of this code is that it allows the loop to take a total of eight cycles, which is evenly divisible by 20,000.

This code is useful for checking for button-down, but the button-up also has to be checked. As can be seen in the `debounce` source code snippet below, both the switch


```

Loop:                                ; Loop Here

Debounce Up                          ; Wait for Key to Go Up

Debounce Down                        ; Wait for Key to Go Down

    comf    PORTB, f                ; Toggle the PORTB Value

    goto   Loop

```

The source code for `debounce`, which can be found in the `code\debounce` folder is

[illegible]

```
Down EQU -1                ; Flag Value for Debounce "Down"
; Macros
Debounce MACRO Direction

    if (Direction < 0)      ; Going Down
        btfsc PORTA, 0
    else
        btfss PORTA, 0      ; Wait for Button Released
    endif
    goto $ - 1

    movlw 0x0100 - 0x0C4    ; Initialize Dlay for a 20 msec
    movwf Dlay              ; Delay
    movlw 0x0100 - 0x00A
    movwf Dlay + 1

    bcf STATUS, Z           ; Make Sure that Zero is Reset

    incfsz Dlay, f
    goto $ + 2
    incf Dlay + 1, f
    if (Direction < 0)
        btfsc PORTA, 0
    else
        btfss PORTA, 0      ; Button Still Released?
    endif
    goto $ - 11             ; No - Loop Around Again
    btfss STATUS, Z         ; Zero Flag Set (20 mSecs Past?)
    goto $ - 6

ENDM                        ; End the Macro

__CONFIG _CP_OFF & _WDT_OFF & _XT_OSC & _PWRTE_ON

PAGE
; Mainline of debounce

org 0

nop

bsf PORTB, 0               ; Make LED on RB0 "off" Initially

bsf STATUS, RP0            ; Goto Bank 1 to set Port Direction
bcf TRISB & 0x07F, 0      ; Set RB0 to Output
bcf STATUS, RP0            ; Go back to Bank 0
Loop:                       ; Loop Here
```

```

Debounce Up          ; Wait for Key to Go Up

    nop              ; Location for Stopping after
                    ; "Up" Debounce
Debounce Down        ; Wait for Key to Go Down

    comf    PORTB, f    ; Toggle the PORTB Value

    goto    Loop

end

```

In the `debounce` source code, there are a few things I want to point out. The first is the addition of the `nop` instruction between the two `Debounce` macro invocations. This instruction is used as an MPLAB breakpoint for testing the code. You cannot put a breakpoint at a macro invocation, and there would be no way of easily checking whether or not the `Debounce Up` macro invocation worked correctly unless you were to input a high value on the switch at RA0, wait 20 ms, and then input a low. The `nop` instruction allows you to hang a breakpoint between the two macro invocations during simulation.

For testing this application, I used the “asynchronous input,” just as was used in the `1edon` experiment. To test the debounce capability of the `Debounce` macro, after I had toggled RA0 high, I then waited a few cycles (anywhere from 10 to 15 ms) to test the operation of `Debounce`. In my PC (300-MHz Pentium II), waiting a full 20 ms takes about 2 minutes. If you have a slower PC, you may want to reduce the loop values while you are simulating the application.

In the `debounce` application, note that I debounce the button high (not pressed) before debouncing it low. The reason for doing this is to make sure that there is a definite break between button presses. This check is normally required for most applications that process button inputs.

The `Debounce` macro isn’t bad for applications that require a single button’s input to debounce before proceeding with an application. For applications that cannot spend large periods of time waiting for button inputs, the debounce code will have to be interrupt-based, as is shown in a later experiment.

TimeEnd: TMRO DELAY THAT NEVER ENDS

The `TimeEnd` application (which uses the same circuit as the preceding experiments) is designed to use TMR0 to delay 1 ms after reset and turn on a LED on RB0. Unfortunately, it doesn’t work out that way, as you will discover when you program a PIC16F84 with the `TimeEnd` application (found in the `code\TimeEnd` folder).

```

title    "TimeEnd - Loop While Waiting for TMR0 to End"
;
; This Uses TMR0 with a 4x Prescaler to Provide a 1 msec
; Delay. The Access instruction will cause the Timer to

```

```
; Never Reach Zero
;
;
; Hardware Notes:
; PIC16F84 Running at 4 MHz
; _MCLR is Pulled Up
; PortB0 Pulled up and Connected to LED
;
; Myke Predko
; 99.06.22
;
LIST R=DEC
INCLUDE "p16f84.inc"

; Register Usage
CBLOCK 0x020          ; Start Registers at End of the SFRs
ENDC

PAGE
_CONFIG _CP_OFF & _WDT_OFF & _XT_OSC & _PWRTE_ON
; Mainline of TimeEnd

org    0

nop

bsf    PORTB, 0        ; Turn RB0 off
bsf    STATUS, RP0
clrf   TRISB ^ 0x080
movlw  0x0D1           ; Set TMR0 to Prescaler and 4x
movwf  OPTION_REG ^ 0x080
bcf    STATUS, RP0

movlw  (1024 - 1000) / 4      ; Reset TMR0 to Wait 1 msec
movwf  TMR0

Loop:
movf   TMR0, f            ; Does TMR0 Equal 00? (1 msec Passed)
btfss  STATUS, Z
goto   Loop

bcf    PORTB, 0          ; Turn RB0 ON

goto   $                ; Loop Forever When Done

end
```

The code is very simple; it sets up the prescaler to be used with TMR0 to divide by four along with having TMR0 driven by the instruction clock. Once this is done, TMR0 is loaded with 6 and waits for the timer to overflow to 0. The only problem is that when you build the circuit and program a PIC microcontroller with `TimeEnd`, the application sits there and doesn't turn on the LED.

When you look at the source code, you probably won't see anything amiss. The problem becomes apparent only when you simulate the application and look at the value of TMR0 as the PIC microcontroller executes through `Loop`. Each time `Loop` executes, TMR0's value stays the same at 6. This can be observed using the MPLAB IDE simulator with a TMR0 watch register and single-stepping through the code.

The problem with the application is with the `movf TMR0, f` instruction just after the `Loop` label. This instruction moves the contents of TMR0 through the ALU to test the value against 0 and conditionally sets the zero flag *and then* stores this value back into TMR0. When the value is stored back in TMR0, the prescaler is reset.

If you check the number of cycles used by `Loop`, i.e.,

```
Loop
  movf   TMR0, f           ; Does TMR0 Equal 00? (1 msec Passed)
  btfss  STATUS, Z
  goto   Loop
```

you'll come up with four instruction cycles each time `Loop` executes. This is a problem because the TMR0 prescaler is set to be a divide by four counter. Each time `Loop` executes, the prescaler is reset, never passing a clock tick to TMR0 to increment it. As a result, TMR0 never changes value, and `Loop` becomes an endless loop.

The easiest way to fix the problem is to change the `movf TMR0, f` instruction to `movf TMR0, w`, with the value in TMR0 being stored in the w register instead of back into itself. When you make this change and program the PIC microcontroller with the updated code, you'll find that it works without any problems.

There may be a question as to how likely this problem is. If you're going to use TMR0, chances are that you are going to use it with an interrupt, and there would be no need for polling it at all. This is true for mid-range devices, but what about the low end?

Using TMR0 for setting up delays in low-end PIC microcontrollers is a fairly common practice, and the problem I've shown here is something that causes becomes apparent to many new application developers. The easiest way to avoid this is *never* to write to the TMR0 register except when setting it originally. If you have a value in WREG you don't want changed, then it should be saved before polling TMR0.

WDT: THE WATCHDOG TIMER

If you are new to microcontrollers in general and the PIC microcontroller specifically, you may be unsure as to what the watchdog timer (WDT) offers for your application. A very common informal experiment that is performed is to enable the watchdog timer and see what happens to the application. The result is usually a problem that is difficult to characterize and find by looking externally to the PIC microcontroller.

To demonstrate how the watchdog timer works, I created the WDT.asm application, which can be found in the code\WDT folder.

```
title "WDT - Demonstrate PIC MCU Reset using WatchDog Timer"
;
; This Code Puts a changing value into PIC microcontroller's PortB
; after loading PortA bits 2 and 3 with the _TO and _PD bits.
;
; The PIC microcontroller should be reset during execution by the
; operation the Watchdog timer, which will cause the _TO and _PD
; Display To Change.
;
;
; Hardware Notes:
; PIC16F84 Running at 4 MHz
; _MCLR is Pulled Up
; All 8 bits of PortB are Pulled up and Connected to LEDs
; PORTA.2 is Pulled up and Connected to a LED for _PD
; PORTA.3 is Pulled up and Connected to a LED for _TO
;
; Myke Predko
; 99.12.23
;
LIST R=DEC
INCLUDE "p16f84.inc"

; Register Usage
CBLOCK 0x020 ; Start Registers at End of the Values
BValue
Dlay:2
ENDC

PAGE
__CONFIG __CP_OFF & __WDT_ON & __XT_OSC & __PWRTE_ON

; Mainline of WDT
org 0

nop

movlw 0x0FF
movwf PORTB
movwf BValue

movlw 0x00C
movwf PORTA
```

```

    bsf     STATUS, RP0
    clrf    TRISB ^ 0x080      ; Make All 8 PortB Bits Output
    movlw   0x013             ; Make RA2 and RA3 Outputs
    movwf   TRISA ^ 0x080
    bcf     STATUS, RP0

    rrf     STATUS, w          ; Get the Status Bits
    xorlw   0x00C             ; Invert _PD and _TO
    andlw   0x00C             ; Clear the Other Bits
    movwf   PORTA             ; Set the LEDs Appropriately

Loop:                                     ; Loop Here

    call    Delay

    bcf     STATUS, C          ; Change PORTB
    btfss   BValue, 7
    bsf     STATUS, C
    rlf     BValue, w
    movwf   PORTB
    movwf   BValue

    call    Delay

    movlw   0x0FF             ; Turn OFF LEDs
    movwf   PORTB

    goto    Loop

Delay:                                     ; Delay 1/5 Seconds
    clrf    Dlay
    clrf    Dlay + 1
    decfsz  Dlay, f
    goto    $ - 1
    decfsz  Dlay + 1, f
    goto    $ - 3

    return

end

```

The application itself uses a lot of LEDs (and their current-limiting resistors) to provide you with the status of the application, as shown in Fig. 20.7, with the breadboard wiring shown in Fig. 20.8; the bill of materials is listed in Table 20.4.

When you first run the application, note that the two LEDs that were added to RA2 and RA3 are on initially. The application code will start flashing the LEDs in an increasing

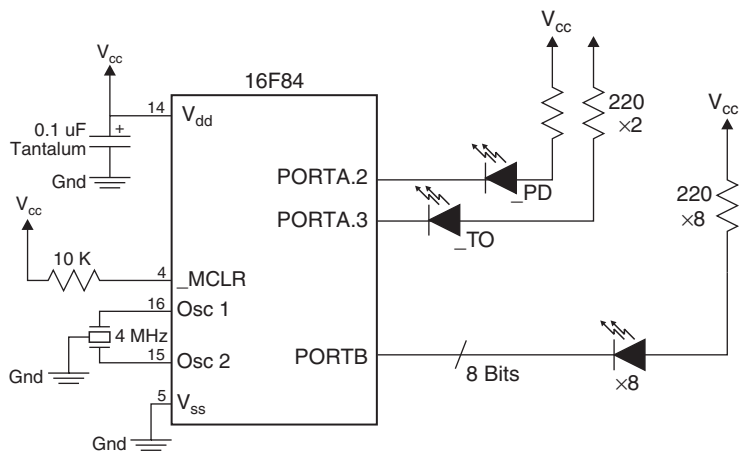


Figure 20.7 WDT: monitor PD and TO during watchdog timer reset.

pattern. After a few seconds, the PIC microcontroller will reset and start executing again. Note that when the PIC microcontroller resets, that the LED connected to RA2 (which outputs the value of **_PD**) is turned off. This indicates that the PIC microcontroller has been reset by a watchdog timer timeout.

Also note that the only thing done to enable the watchdog timer in the application code is to just change the **_WDT_OFF** configuration fuse parameter to **_WDT_ON**. When the watchdog timer is enabled and the PIC microcontroller powers up, the **OPTION** register is loaded with all 1s, which means that the prescaler is dedicated to the watchdog

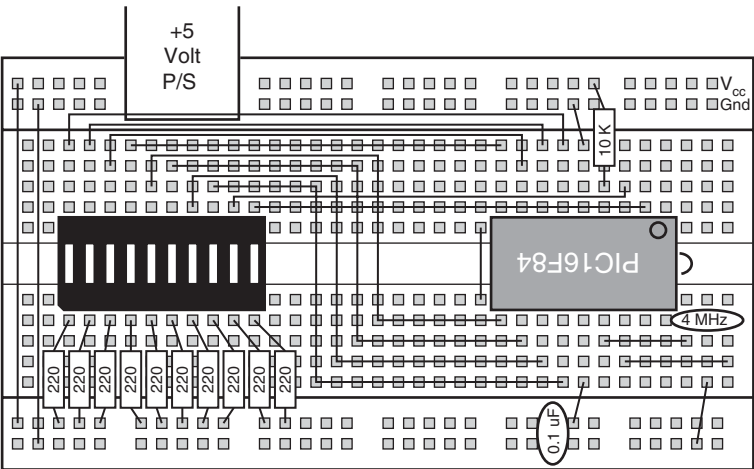


Figure 20.8 The watchdog timer reset circuit built on a breadboard.

TABLE 20.4 BILL OF MATERIALS FOR THE WDT EXPERIMENT

PART	DESCRIPTION
PIC microcontroller	PIC16F84-04/P
4 MHz	4-MHz ceramic resonator with internal capacitors
LED	10-LED bargraph display
10-k	10 k Ω , $\frac{1}{4}$ W
220	220 Ω , $\frac{1}{4}$ W
0.1- μ F	0.1- μ F tantalum
Misc.	Breadboard, wiring, +5-V power supply

timer and is running at the maximum value (which is 127). Because the nominal watchdog timer timeout value is 18 ms, the actual timeout interval becomes

$$\begin{aligned}\text{Watchdog timer timeout} &= \text{nominal timeout interval} \times \text{prescaler} \\ &= 18 \text{ ms} \times 127 \\ &= 2.286 \text{ s}\end{aligned}$$

or the 2.3 seconds that is normally quoted by Microchip as the maximum time for the watchdog timer timeout interval.

To prevent the watchdog timer from resetting the PIC microcontroller, you have to add the `clrwdt` instruction somewhere within the `Loop` code to reset the watchdog timer (and the prescaler). There should be only one `clrwdt` instruction in the code (to avoid an application that is running amok to reset the watchdog timer accidentally), and it should be reset after about 50 percent of the nominal timeout interval has executed. This 50 percent value is a rule of thumb to ensure that code added to the application does not cause a problem with the timer and that any variances in the watchdog timer circuit do not cause a short interval that results in an unwanted reset.

Proper watchdog timer application specification is a bit of a black art. I have not found many applications that really require it—the PIC microcontroller is quite a tough little device from the perspective of electrical interference. The only situations where I would look toward enabling it is in a high-noise environment or high potential for ESD discharge, such as a TV set (the flyback transformer can generate large EMFs), manufacturing floors, automotive applications, and aviation instruments. Personally, I have never had an application upset by what I would consider to be a noise problem when the PIC microcontroller has been decoupled properly.

Having said all this, you are probably not thinking of using the watchdog timer at all for your applications, but there is always the chance that you will accidentally. For many PIC microcontrollers (such as the PIC16F84), enabling the watchdog timer is accomplished by leaving a bit set in the configuration fuses. In these cases, if you forget to put the `_WDT_OFF` parameter into your `__CONFIG` statement, then you will be

inadvertently enabling the watchdog timer. When this happens and you are new to the PIC microcontroller, finding the problem is just about impossible, and as I discussed earlier, the problem can appear to be the same as if the decoupling capacitor was forgotten or incorrectly wired.

This means that if you have an application that seems to start okay but resets itself, you probably should take the PIC microcontroller out of the application and check it in your programmer to see if the watchdog timer is enabled. For some programs (such as the ones presented in this book), you will have to go back to the source and make sure that the `_WDT_OFF` parameter is specified and not forgotten.

TMR0: TMR0 SETUP WITH PRESCALER

As I work through the later experiments and the projects, I will be using the TMR0 and prescaler hardware a lot. This experiment also will give you an idea of the speeds at which the PIC microcontroller operates. For this experiment, you may want to look at the data output with an oscilloscope, but it is not necessary if you have a logic probe.

The TMR0 application (which can be found in the `code\TMR0` folder) is another one that uses PORTB for LED outputs. What TMR0 does is enable TMR0 to run in from the processor's clock and then poll the TMR0 value and write it out to PORTB so that the value of TMR0 can be checked externally. The circuitry is shown in Fig. 20.9 and can be built on a breadboard as in Fig. 20.10. The bill of materials is listed in Table 20.5.

```

title "TMR0 - Demonstrate the Operation of TMR0 with Prescaler"
;
; This Code sets up TMR0 to run from the Instruction Clock and
; uses the Prescaler to divide the incoming clock.
;
;

```

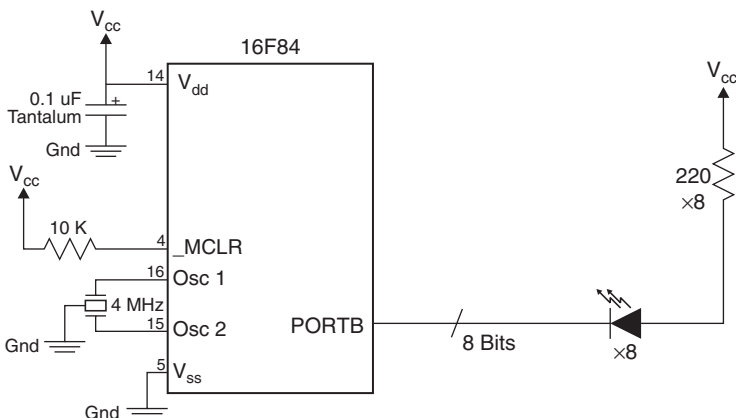


Figure 20.9 TMR0: displaying TMR0 values on eight LEDs.

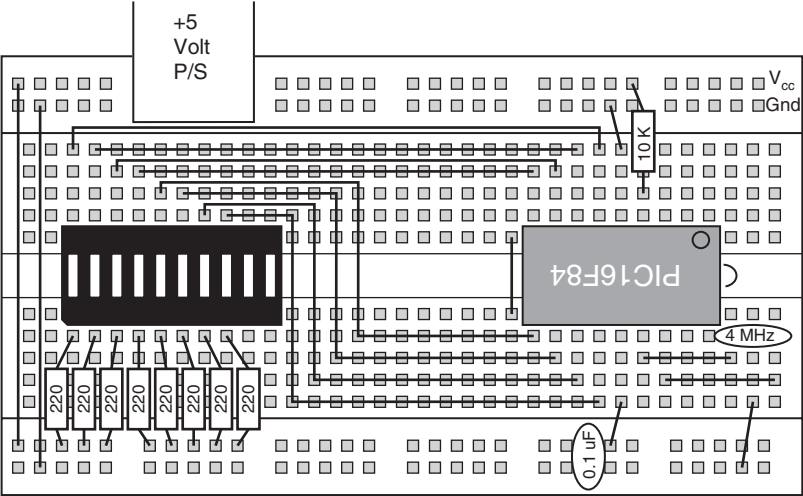


Figure 20.10 The breadboard wiring for the TMR0 application.

```
; Hardware Notes:
; PIC16F84 Running at 4 MHz
; MCLR is Pulled Up
; All 8 bits of PortB are Pulled up and Connected to LEDs
;
; Myke Predko
; 99.12.26
;
LIST R=DEC
INCLUDE "p16f84.inc"

; Register Usage
```

TABLE 20.5 BILL OF MATERIALS FOR THE TMR0 EXPERIMENT	
PART	DESCRIPTION
PIC microcontroller	PIC16F84-04/P
4-MHz	4-MHz ceramic resonator with internal capacitors
LED	10-LED bargraph display
10-k	10 k Ω , $\frac{1}{4}$ W
220	220 Ω , $\frac{1}{4}$ W
0.1- μ F	0.1- μ F tantalum
Misc.	Breadboard, wiring, +5-V power supply

```
CBLOCK 0x020                                ; Start Registers at End of the Values
ENDC

PAGE
__CONFIG __CP_OFF & __WDT_OFF & __XT_OSC & __PWRTE_ON

; Mainline of TMR0
org 0

nop

bsf    STATUS, RP0
clrf   TRISB ^ 0x080                        ; Make All 8 PortB Bits Output
movlw  0x0FF ^ ((1 << T0CS) | (1 << PSA) | 7)
addlw  0                                    ; Put in Prescaler Value
movwf  OPTION_REG ^ 0x080                  ; Load the Option Register Value
bcf    STATUS, RP0

Loop:                                     ; Loop Here

comf   TMR0, w                             ; Output the TMR0 Value
movwf  PORTB

goto   Loop

end
```

The instructions that set up the OPTION register to use the PIC microcontroller's TMR0 clock input and the prescaler for TMR0 will seem a bit cumbersome and hard to understand. Normally, all the bits in the OPTION register are set; to change its operation for specifying TMR0 as using the instruction clock input as well as directing the prescaler to TMR0, the bits have to be reset. In addition, I clear the prescaler's value. To be honest, instead of creating the cumbersome instruction

```
movlw  0x0FF ^ ((1 << T0CS) | (1 << PSA) | 7)
```

I simply use the constant value 0x0D7 as

```
movlw  0x0D7
movwf  OPTION_REG ^ 0x080
```

which is a lot easier to read, and if it is before a `movwf OPTION_REG ^ 0x080` instruction, then I know that I am setting up the OPTION register. I used the more complex format to show how the OPTION register bits are set for this application. The next instruction, the `addlw 0`, is used to set the prescaler to something other than one to one, the reason for which will become obvious.

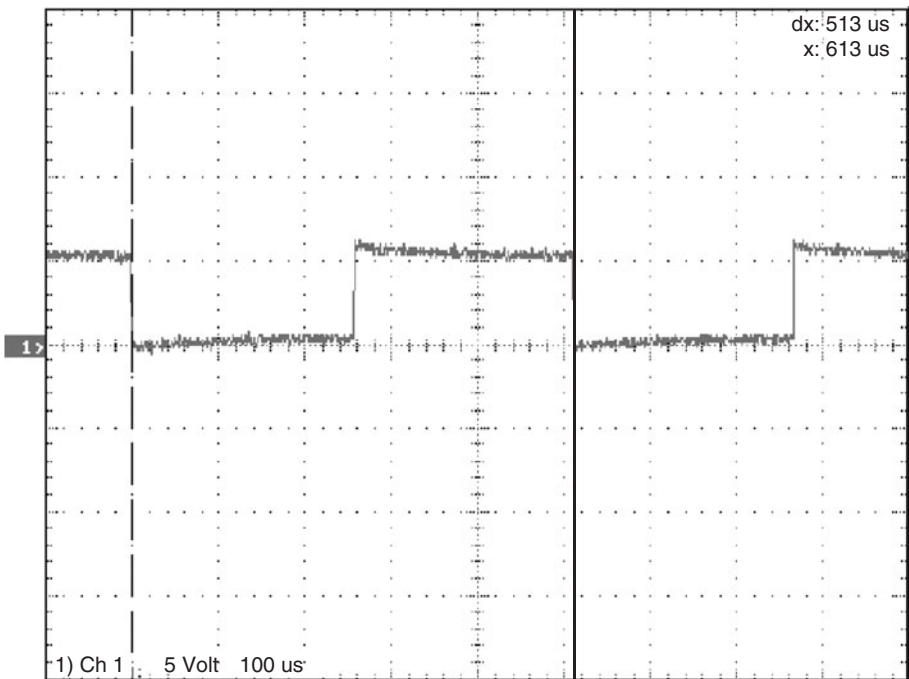


Figure 20.11 TMR0 application RB7 waveform on an oscilloscope with the prescaler equal to zero.

When you first run the circuit, you probably will think that TMR0 is actually 0xFF and is not changing state. Actually, if you check the I/O pins with a logic probe or an oscilloscope, you will discover that the pins are switching, with the most significant bit (RB7) switching at a rate of just less than 2 kHz, as shown in Fig. 20.11. If you check the other PORTB pins, you will find that they are switching at higher frequencies, with RB1 switching at a rate of 125 kHz.

When you look at the I/O pins, you should notice that RB0 does not switch at all. This is because the `Loop` code in `TMR0.asm` takes four instruction cycles, and each time through, the least significant bit of TMR0 is always at the same value for each instruction in the loop and will never change relative to the output. This is something that you can see if you simulate the application.

To actually see TMR0 changing state, change the `addlw 0` instruction in the preceding code above `addlw 7`, which will change the prescaler from a one to one to one to 128. When the code has been reassembled and put into a PIC microcontroller that is in a circuit, the LED connected to RB7 will flash noticeably. If you put an oscilloscope on RB7, you will find that it is flashing at a rate of 16 times per second (16 Hz).

The RB7 LED flashes should not be individually observable by you. Instead, you should see a flashing blur because 16 Hz is at about the limit of your ability to discriminate between events. If you were to slow down the flashing any more, you would

be able to distinguish individual on and off events. If the flashing were any faster (as is the case with RB6), the LED would just appear to be on all the time, although not at the maximum brightness possible because the signal passed to the LED actually would be a PWM signal with a 50 percent duty cycle.

This application should give you an appreciation of how fast the PIC microcontroller actually executes. Note that for changes to the TMR0's most significant bit to be noticeable, a prescaler of 256 had to be assigned, which divides the RB7 by 65,356 times.

Random: RANDOM NUMBER GENERATOR

The `Random.asm` application (found in the `code\Random` subdirectory of the PIC microcontroller directory on your PC) is a collection of the most recent experiments into an application that you should be able to use. Every time the button in the `Random` circuit (shown in Fig. 20.12 with breadboard wiring diagram in Fig. 20.13 and bill of materials in Table 20.6) is pressed, a new random number will be displayed on the eight LEDs connected to PORTB.

The code itself takes advantage of the `Debounce` macro presented earlier, as well as the TMR0 setup provided in the preceding experiment.

```

title "Random - Produce a Random Number Generator with TMR0"
;
; This Code Displays TMR0 on RB0 every time the Button on RA0
; is Pressed.
;
;
; Hardware Notes:

```

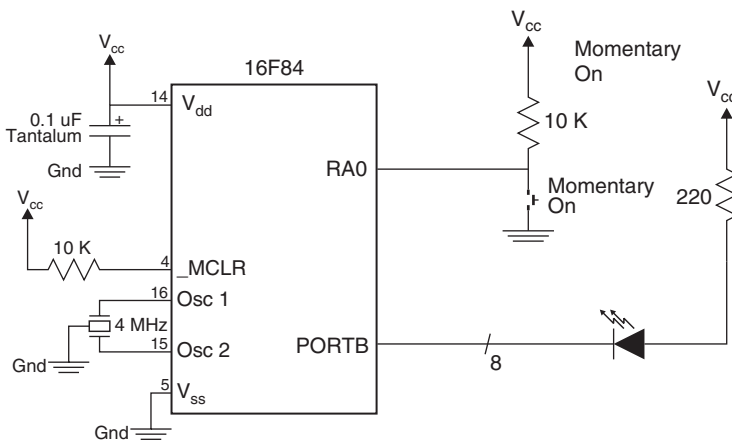


Figure 20.12 Random number generating circuit using a debounced button input.

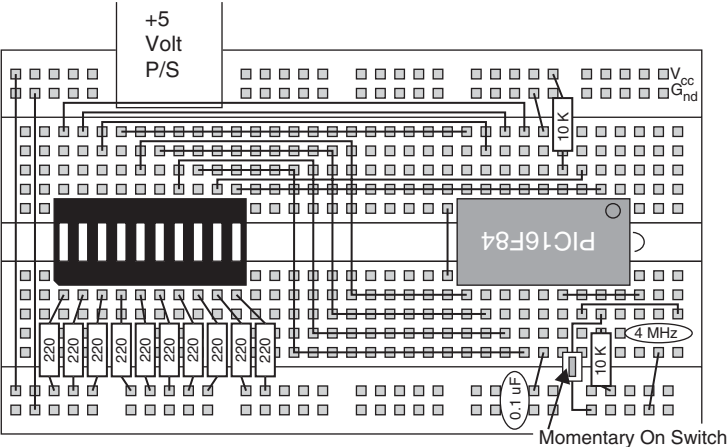


Figure 20.13 Breadboard wiring for the random number generator circuit.

```
; PIC16F84 Running at 4 MHz
; _MCLR is Pulled Up
; All 8 bits of PORTB are Pulled up and Connected to LEDs
; PORTA.0 is Pulled up with a Momentary On Pulling to Ground
;
; Myke Predko
; 99.12.26
;
LIST R=DEC
INCLUDE "p16f84.inc"
```

TABLE 20.6 BILL OF MATERIALS FOR THE RANDOM EXPERIMENT	
PART	DESCRIPTION
PIC microcontroller	PIC16F84–04/P
0.1-μF	0.1-μF tantalum
10-k	10 kΩ, ¼ W
220	220 Ω, ¼ W
4-MHz	4-MHz ceramic resonator with internal capacitors
LED	10-LED bargraph display
Misc.	Breadboard, wiring, +5-V power supply

```
; Register Usage
CBLOCK 0x020          ; Start Registers at End of the Values
Dlay:2                ; Delay Value
ENDC
Up EQU 1              ; Flag Value for Debounce "Up"
Down EQU -1           ; Flag Value for Debounce "Down"

; Macros
Debounce MACRO Direction

    if (Direction < 0) ; Going Down
        btfsc PORTA, 0
    else
        btfss PORTA, 0 ; Wait for Button Released
    endif
    goto $ - 1

    movlw 0x0100 - 0x0C4 ; Initialize Dlay for a 20 msec
    movwf Dlay ; Delay
    movlw 0x0100 - 0x00A
    movwf Dlay + 1

    bcf STATUS, Z ; Make Sure that Zero is Reset

    incfsz Dlay, f
    goto $ + 2
    incf Dlay + 1, f
    if (Direction < 0)
        btfsc PORTA, 0
    else
        btfss PORTA, 0 ; Button Still Released?
    endif
    goto $ - 11 ; No - Loop Around Again
    btfss STATUS, Z ; Zero Flag Set (20 mSecs Past?)
    goto $ - 6

endm ; End the Macro

PAGE
__CONFIG __CP_OFF & __WDT_OFF & __XT_OSC & __PWRTE_ON
; Note that the WatchDog Timer is OFF

; Mainline of Random
org 0

nop
```



```

movlw 0x0FF          ; Turn off all the LEDs initially
movwf PORTB
bsf    STATUS, RP0
clrf   TRISB ^ 0x080 ; Make All 8 PortB Bits Output
movlw  0x0D0          ; Assign Prescaler of 1:1 to TMR0
movwf  OPTION_REG ^ 0x080 ; Load the Option Register Value
bcf    STATUS, RP0

Loop:           ; Loop Here

Debounce Up      ; Wait for Key to Go Up

    nop          ; Location for Stopping after
                  ; "Up" Debounce
Debounce Down    ; Wait for Key to Go Down

    comf  TMR0, w ; Output the TMR0 Value
    movwf PORTB

    goto  Loop

end

```

As I indicated at the start of this experiment, the code simply starts off TMR0 with the prescaler assigned but using a 1:1 value. This means that TMR0 is updated once every two instruction cycles. The `Loop` code waits for the button on RA0 to go high, and then when it is pressed (and debounced), the current value in TMR0 is output onto the eight LEDs.

To demonstrate the random nature of the application, I pressed the button on RA0 16 times and recorded the results in Table 20.7.

As you can see in this table, the numbers are reasonably random. Actually, I was a bit surprised that the same value didn't come up twice. The statistical "birthday test" can be applied to this situation to see how many button presses have to be made before there is better than a 50 percent chance of two numbers being the same. To compute this, if you assume that the TMR0 value returned can be any number from 0 to 255, there are 256 different opportunities for different values. For the second random number generated to not be equal to the first, the chances are 255/256. For the third number not to be equal to the first or second, the chances are $255/256 \times 254/256$. The ultimate product is multiplied by each time the button is pressed. This is computed until there is a 50 percent chance for the number pressed not to be equal to anything before it.

I've always found statistics to be the science of the negative rather than the positive. (If you've taken statistics, you'll know what I'm talking about). In any case, there will have to be 18 presses until there is a 50 percent chance of a repeated random number. It took me 8 additional presses (for a total of 24) to get 0xA9, which matches the fourteenth press.

TABLE 20.7 RANDOM NUMBERS PRODUCED BY RANDOM EXPERIMENT	
BUTTON PRESS	TMRO VALUE
1	0x43
2	0xBA
3	0x75
4	0x12
5	0xC3
6	0xD2
7	0x49
8	0x00
9	0x6A
10	0xE8
11	0xC6
12	0x44
13	0x6C
14	0xA9
15	0xC4
16	0xF9

short: THE SIMPLEST PRACTICAL PIC MICROCONTROLLER APPLICATION POSSIBLE

The title of this experiment probably seems somewhat arrogant, but I think that I have created a small application that does not make a liar out of me. This application is only two instructions long, and I think that it shows off some obscure aspects of the PIC microcontroller, as well how PIC microcontroller programming works. It also will provide you with a simple application that can be used in a variety of situations to check on the health of a PIC microcontroller before an application is programmed in.

The application itself simply flashes an LED on RB7. The circuit that I used is shown in Fig. 20.14 and is wired as in Fig. 20.15. The bill of materials is listed in Table 20.8.

The application code itself is honestly only two instructions long and will cause the LED to flash at about 7.5 times per second when the PIC microcontroller is running at 4 MHz. The application itself can be found in the code\short folder. The two instructions of the application code are just

```
tris    PORTB           ; Save WREG in TRISB
xorwlf  PORTB, f        ; XOR PORTB with the contents of WREG
```

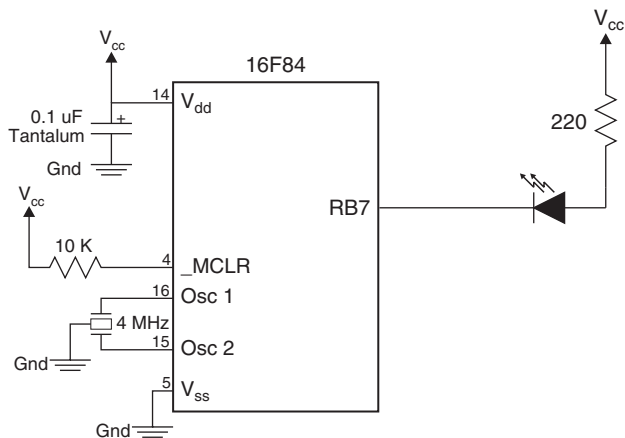


Figure 20.14 Circuit to demonstrate the shortest possible application.

I'm not going to reveal the entire source code just yet because there are some aspects of it I want to discuss later in this write-up. In any case, the unique application code is only these two instructions long, but those instructions take advantage of two features of the PIC microcontroller and its processor.

I realize that elsewhere in this book I discuss how I feel that the `tris` instruction never should be used in the mid-range PIC microcontroller. I use the `tris` instruction in this application simply because it is the method of changing PORTB's TRIS register

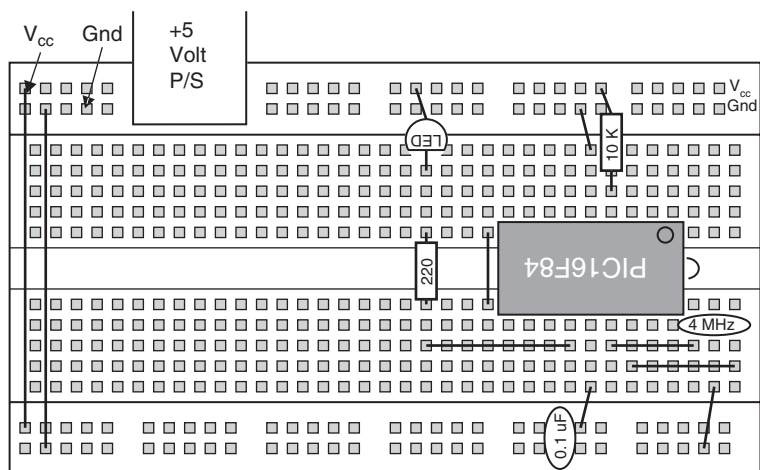


Figure 20.15 Breadboard wiring for the shortest possible application.

TABLE 20.8 BILL OF MATERIALS FOR THE SHORT EXPERIMENT	
PART	DESCRIPTION
PIC microcontroller	PIC16F84–04/P
0.1-μF	0.1-μF tantalum
220	220 Ω, ¼ W
4-MHz	4-MHz ceramic resonator with internal capacitors
LED	Visible-light LED, any color
Misc.	Breadboard, wiring, +5-V power supply

in the fewest number of instructions. If I were to update TRISB properly, I would use the code

```
bsf      STATUS, RP0          ; Access Bank 1
movwf    TRISB ^ 0x080        ; Store the Contents of WREG in TRISB
bcf      STATUS, RP0          ; Return to Bank 0
```

The first feature that the code takes advantage of is that unprogrammed program memory addresses are all set to 1s. This is a function of EPROM and Flash memory; when it is erased, the bits in the memory are all set to 1s. For mid-range PIC microcontrollers, the instruction `addlw 0x0FF` has the bit pattern 0x03FFF, which is all the bits set in an instruction word.

Thus, as the code executes through the unprogrammed program memory, it is adding 0x0FF (or –1) to the contents of the w register. On power-up, the contents of WREG are undefined, but as the code executes through the unprogrammed instructions, the value within it will be incremented continually.

The other aspect of the PIC microcontroller that this application takes advantage of is that when the program counter reaches the end of program memory, it resets and continues executing from address 0.

These two features of the PIC microcontroller mean that the application code, while it is only two instructions long, actually executes as if it were

```
Loop:                                ; Loop Back Here
    call    Delay1019                ; Delay Same Cycles executing
; "addlw 0x0FF" 1022x
    addlw   2                        ; Equivalent to "addlw 0x0FF" 1022x
    tris    PORTB                    ; Save WREG in TRISB
    xorwf   PORTB, f                  ; XOR PORTB with the contents of WREG
    goto    Loop
```

The effective adding of two each time through the loop is why I have put the LED on RB7, the most significant bit in the PORTB register. As the w register is incremented

by two, the most significant register will be toggled at the slowest rate. The LED could be put on any output from RB1 to RB7, but you will find that as you use the least significant bits, the rate at which the LED flashes increases to the point where it cannot be observed.

The actual source code is

```

title "SHORT - Is this the Shortest Possible Application?"
;
; Look at a Two Instruction Application. Continually Update
; w (with the "unprogrammed" ADD 0x0FF) and then use the
; value for "PORTB" and "TRISB".
;
; Hardware Notes:
; 16F84 Running at 4 MHz
; Reset is tied directly to Vcc and PWRT is Enabled.
; PortB is used for Output
;
;
; Myke Predko
;
; 99.10.26 - "Short" Created
;
list R=DEC
include "p16f84.inc"

__CONFIG __CP_OFF & __XT_OSC & __PWRTE_ON & __WDT_OFF

org      0x03FE

    tris    PORTB                ; Save WREG in TRISB
    xorwf   PORTB, f              ; XOR PORTB with the contents of WREG

end

```

This code runs through the PIC microcontroller's program memory repeatedly, adding $-1,022$ to the contents of WREG (which is the same as adding $+2$ when the 8-bit register is taken into account). Next, the new value for WREG is used for the TRISB register and then XORed with the contents of PORTB to change PORTB's output value.

The operation of the application can be observed by enabling the "Program Memory" window (from the "Window" pull-down menu on MPLAB's top pull-downs), as shown in Fig. 20.16. Once you have brought this up, you can single-step through the application to see the execution of the two instructions explicitly programmed into the otherwise erased program memory. I put the two instructions at the end of program memory because you can see that the contents of WREG have been effectively incremented by 2 by the time you get to the end of the program memory.

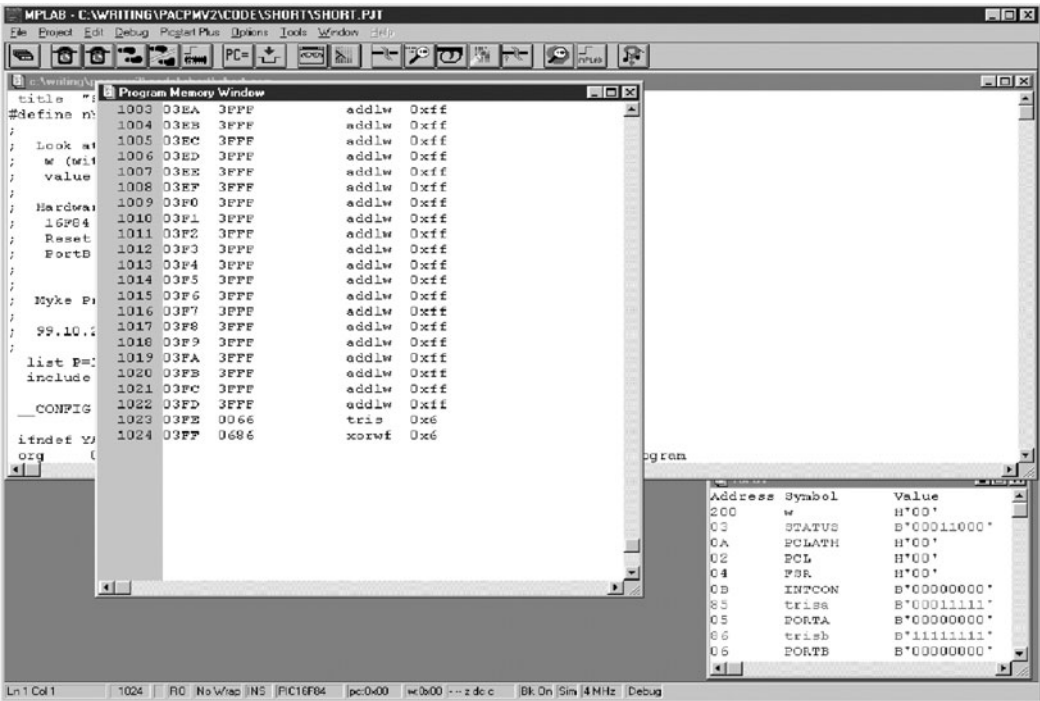


Figure 20.16 The MPLAB IDE program memory dump showing the two instructions of the application at the end of program memory.

Analog Input/Output

When I took electromagnetics and transmission theory in university, there was nothing I wished for more than the world to consist of nothing but ones and zeros. Unfortunately, the world just doesn't work that way—varying voltage levels are available inside the PIC microcontroller's Vdd and Vss voltage range to outside in terms of positive and negative voltages. In addition, the different speeds and waveforms of these voltages can make your life more difficult as well. In the following experiments I want to demonstrate to you some practical examples of how analog I/O interfacing with the PIC microcontroller can be accomplished using the digital I/O functions of the PIC microcontroller to perform the basic tasks of converting an analog resistance to a digital value and a digital value to an analog output voltage.

ADCLess: MEASURING RESISTANCE VALUES WITHOUT AN ADC

For measuring resistance values without an ADC, a simple RC network can be used with the PIC microcontroller, as shown in Fig. 20.17. This method has been taken from the Parallax Basic Stamp 2 and is quite easy to do and works quite well, although there are a few concerns that have to be worked through for this method to be used in an application.

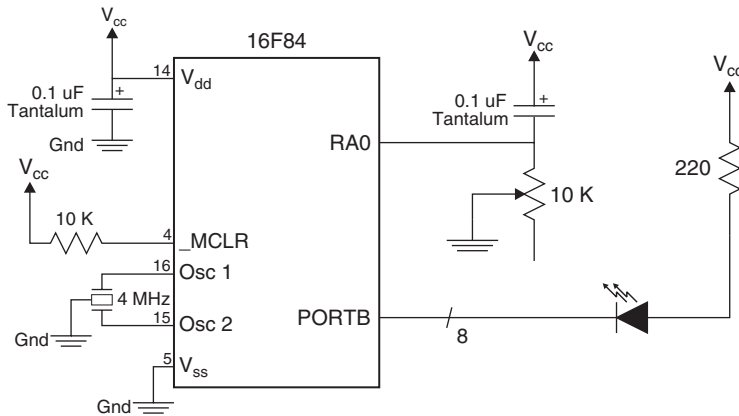


Figure 20.17 Using the PIC microcontroller's digital I/O pins to read a resistor value.

To measure the resistance (assuming that the capacitor is of a known value), the PIC microcontroller first charges the capacitor to 5 V (or its nominal output) using the I/O pin in output mode. Once this is done, the pin changes to input mode and waits for the capacitor to discharge through the potentiometer. Looking at this operation on an oscilloscope, the waveform produced by the circuit looks like Fig. 20.18. In this figure, the “charge” and “discharge” cycles can be seen clearly.

From basic electronic theory, we know that the time required for the capacitor to charge is

$$\text{Time} = R \times C \times \ln(V_{\text{end}}/V_{\text{start}})$$

where V_{start} and V_{end} are the starting and ending voltages that we are interested in. For the PIC microcontroller, we would be interested in the capacitor voltage starting at V_{dd} (after being charged by the PIC microcontroller to 5 V) and then waiting for the capacitor to discharge to the input transition point (2.5 V in the PIC microcontroller).

Because we know the capacitor value along with the voltages and the time it took for the capacitor to discharge, we can rearrange the preceding formula to find R :

$$R = \text{time} / [C \times \ln(V_{\text{end}}/V_{\text{start}})]$$

Therefore, by controlling the voltage applied to the network and knowing the value of the capacitor, we can determine the value of the resistor.

The code used to test the analog I/O uses the following logic:

```
int PotRead()                // Read the Resistance at the I/O Pin
{
    int i;
```

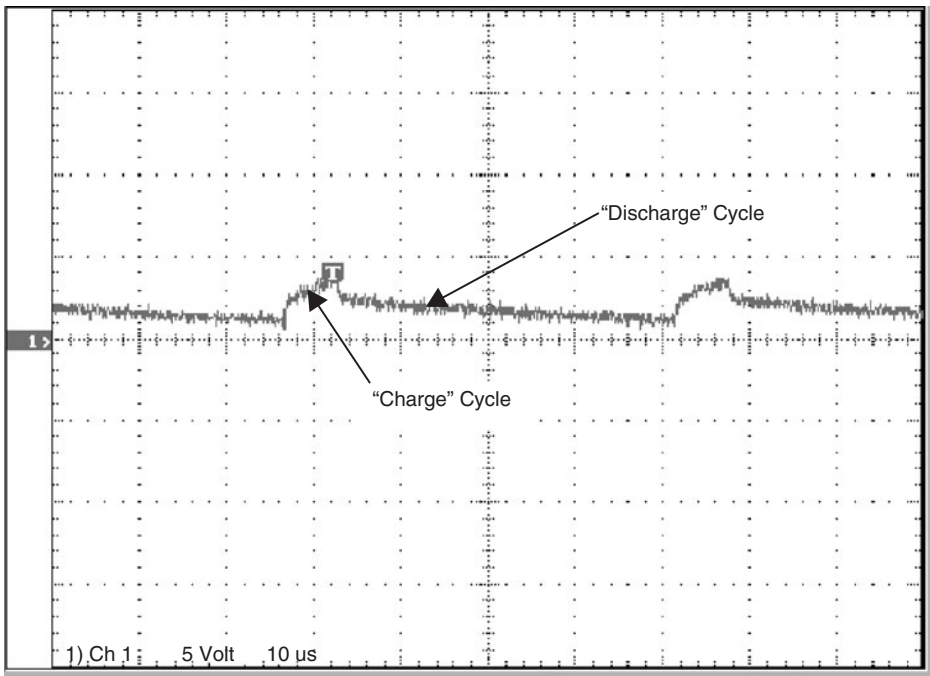


Figure 20.18 The charge/discharge waveform of charging the capacitor and measuring the discharge time.

```

TRIS.Pin = Output;           // Set the Output Mode
Pin = 1;                     // Output a "1" to Charge the Capacitor
for (i = 0; i < 5μsec, i++ );
TRIS.Pin = Input;           // Now, Time How Long it Takes for the
TMR0 = 0;                   // the Capacitor to Discharge through
while (Pin == 1);           // the Potentiometer

return TMR0;                // Return the TMR0 Value for the
                           // Discharge Time

} // end PotRead

```

This code is unique in that no RAM registers are used for the timing; this is done totally within the PIC microcontroller hardware. TMR0 does not have to be used; instead, a simple counter within the `while (Pin == 1)` loop could be incremented.

The source for this experiment is `ADCLess.asm` and can be found in the `code\ADCLess` folder.

```

title "ADCLess - Reading a Resistor Value without an ADC"
;
; This Program copies the "RCTIME" instruction of the Parallax Stamp.

```



```

; A resistor value is read repeatedly and displayed.
;
; This program is a modification of PROG17.ASM
;
; Hardware Notes:
;   PIC16F84 running at 4 MHz
;   Reset is tied directly to Vcc and PWRT is Enabled.
;   A 10K Pot along with a 0.1µF Cap and 100 Ohm Series Resistor on
;   PORTA.0
;   A 220 Ohm Resistor and LED is attached to all the PORTB.7:0
;
; Application Updated: 99.12.26 for 4 MHz PIC16F84.
;
; Myke Predko
; 96.06.02
;
LIST R=DEC
INCLUDE "p16f84.inc"

; Registers

__CONFIG __CP_OFF & __WDT_OFF & __XT_OSC & __PWRTE_ON

PAGE
; Mainline of ADCLess

org      0

nop

movlw 0x0FF
movwf PORTB           ; Turn off all the LED's
clrf  PORTA           ; Use PORTA as the Input

bsf    STATUS, RP0    ; Page 0 to set Port Direction
clrf   TRISB & 0x07F  ; Set all the PORTB bits to Output
movlw 0x0D2           ; Setup the Timer to fast count
movwf OPTION_REG & 0x07F ; Put in Divide by 8 Prescaler for 4x
; Clock
bcf    STATUS, RP0    ; Go back to Page 0

movlw TRISA           ; Have to Set/Read PORTA.0
movwf FSR             ; - Use FSR instead of Changing RP0

Loop:

bsf    PORTA, 0       ; Charge Cap on PORTA.0
bcf    INDF, 0        ; Make PORTA.0 an Output

```


TABLE 20.9 BILL OF MATERIALS FOR THE ADCLess EXPERIMENT

PART	DESCRIPTION
PIC	PIC16F84-04/P
0.1- μ F	0.1- μ F tantalum
10-k	10 k Ω , $\frac{1}{4}$ W
4-MHz	4 MHz with built-in capacitors
10 LEDs	Red LED bargraph display
220	220 Ω , $\frac{1}{4}$ W
10-k pot	10-k Ω potentiometer
Misc.	Breadboard, wiring, +5-V power supply

If you work through different capacitors for the RC network in this experiment, you'll discover how dependent the circuit is on the capacitor value. I found that after trying four different capacitors, I got four different upper limits, with some going beyond 0xFF (the limit that can be returned by the 8-bit TMR0) and one having a maximum value of 0x46. This leads to the biggest problem with this circuit, and that is its dependency on the parts used.

Because of the variance to the capacitor value, I would not recommend this circuit for critical resistance measurements. Yes, a precision cap and power supply, along with characterizing the timer values from the PIC microcontroller, would give accurate results, but as with using precision parts for an RC oscillator, this is not reasonable for volume production.

In the Basic Stamp, a scale value is specified, and the result is returned as a fraction of this value. While this is better than trying to match parts, it still requires some extra work to tune the scale value to the individual circuit.

There is also another problem with this circuit that you probably won't observe unless you put an oscilloscope or digital multimeter (DMM) on the RA0 pin. When the resistor is set to a very low value, the RA0 pin is essentially connected to ground. Many people put in a 100- Ω resistor to prevent a dead short to ground, but this increases the time for the capacitor to become fully charged. In this experiment, I have not included the 100- Ω current-limiting resistor between RA0 and the RC network so that you can observe this issue.

The advantages of this circuit and software are its simplicity and few PIC microcontroller advanced resources used. While not providing high accuracy, the circuit does provide excellent repeatability that can be very useful in many applications. It is not an optimal application because interrupts must be disabled during a resistance value read, and the limits to the circuit must be established first based on the value of the capacitor used in the RC network.

One last thing to notice about this circuit is the use of FSR to point to TRISA (for the switching between output and input in the A/D function). FSR is loaded with the address of TRISA (0x85), and using INDF, TRISA is accessed directly without having to change the default bank (the usual `bsf STATUS, RP0` instruction). This is possible because the FSR register provides up to 8 bits of addressing, which eliminates the need for the RP0 bit when it is addressing registers. This method of using FSR may not be advisable in all applications because dedicating it to an individual function limits its usefulness elsewhere in the application.

vLadder: RESISTOR LADDER OUTPUT

While a single reference voltage produced by a voltage divider may be useful for some applications, a variable-voltage output is much more useful for many other applications. In this experiment, I want to show how multiple analog voltages (approximately 0.55 V apart) can be produced by the PIC microcontroller using the digital I/O pins with a voltage divider.

The output voltage is determined from a voltage divider that has the formula

$$V_{out} = V_{cc} \times [R_n / (R_s + R_n)]$$

where R_n is the resistance between the tap and Vcc and R_s is the resistance between the tap and ground.

A variable resistance voltage divider can be implemented on the PIC as a resistor ladder such as the V_{ref} circuit of the 16C62x devices. The circuit used in this application is shown in Fig. 20.20, and the bill of materials is listed in Table 20.10. Depending

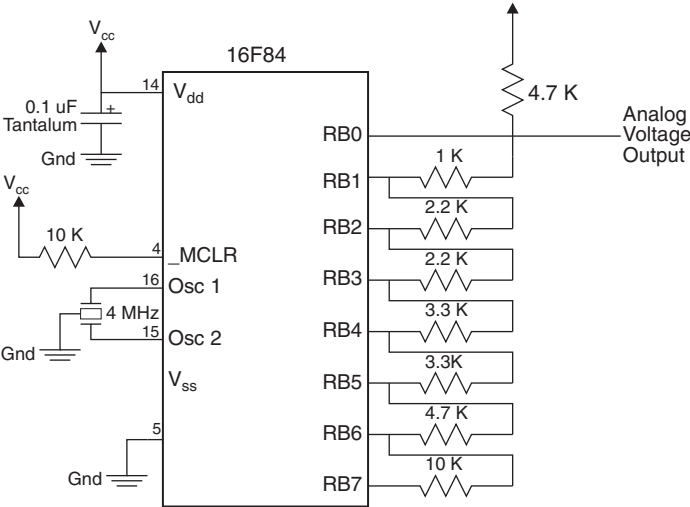


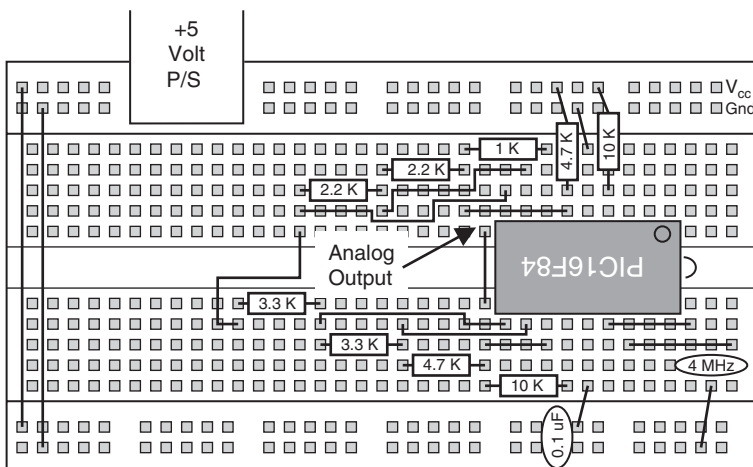
Figure 20.20 Schematic for vLadder.

TABLE 20.10 BILL OF MATERIALS FOR THE vLadder EXPERIMENT

PART	DESCRIPTION
PIC	PIC16F84-04/P
0.1- μ F	0.1- μ F tantalum
10-k	10 k, $\frac{1}{4}$ W resistor
4-MHz	4 MHz with built-in capacitors
4.7-k	4.7 k, $\frac{1}{4}$ W resistor
1-k	1 k, $\frac{1}{4}$ W resistor
2.2-k	2.2 k, $\frac{1}{4}$ W resistor
3.3-k	3.3 k, $\frac{1}{4}$ W resistor
Misc.	Breadboard, wiring, +5-V power supply

on which PORTB output is active, the R_s resistance can be varied to change the output voltage of the circuit. The R_s resistance is varied by changing the PORTB I/O pin to active. All the port pins have been loaded with a 0, and when one of the them is put into output mode, the circuit is grounded at this point, with the resistance being all the resistors between it and the analog voltage output.

Wiring this circuit is actually quite a challenge, and I found that when I redid it for the second edition, I had a number of difficulties getting it right (which will be discussed next). For the breadboard version of the application, the wiring is shown in Fig. 20.21.

**Figure 20.21** Breadboard wiring for vLadder experiment.

Getting the wiring right is not trivial, and in many ways, this was the most difficult application for which to specify the wiring so as to keep it simple enough for you to follow and build your own circuit.

When I created this experiment originally (for the first edition), I rearranged the preceding formula to

$$R_n = (V_{\text{out}}/V_{\text{cc}}) \times R_s / [1 - (V_{\text{out}}/V_{\text{cc}})]$$

By choosing a value for R_s , (say, 10 k Ω), we can easily calculate the value of R_n for a given V_{out} .

The reason why I am going to all this trouble to come up with a formula and calculate it out is to ensure that I can get reasonable linearity in V_{out} for different bit outputs. With the resistor ladder connected to port B, we can have nine different voltages. In selecting these voltages, I have tried to space them 0.55 V evenly apart.

With the circuit shown in Fig. 20.21, you can get nine different voltages very simply. The first is when all the output bits are turned off, which leaves only a pull-up connected to the analog voltage output for a maximum voltage output. If RB0 is in output mode and pulling the output line to ground, this is the low value for the application. R_s values can be added to the circuit by outputting a zero (low voltage) on a pin (terminating the resistor ladder at that point). The remaining seven voltages are selected by grounding intermediate resistors in the ladder using this method.

One *very* important thing to remember is the minimal current (load) output drive characteristics of this circuit. A resistor ladder such as this is very poor at maintaining the output voltage if it has to source or drive current. If there is any type of tangible current flow (more than a few microamperes) outside the voltage divider, the output voltage will be changed from what you expect. The best way to avoid this is to buffer the resistor ladder output using an op-amp, as shown in Fig. 20.22.

When I created this circuit originally, I calculated specific values so that the output would be linear. The same value could be used for each resistor in the ladder, but if we were to plug these into the PIC microcontroller, we would find that we would get the voltage output shown in Fig. 20.23. In the graph shown in this figure, I have plotted what I would consider to be the ideal voltage ramp.

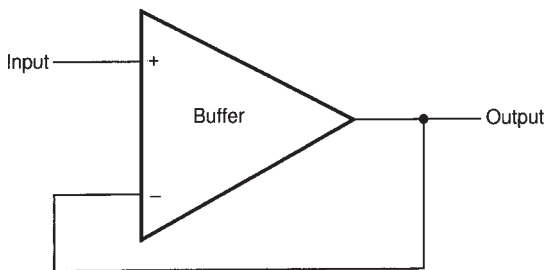


Figure 20.22 Analog voltage buffer.

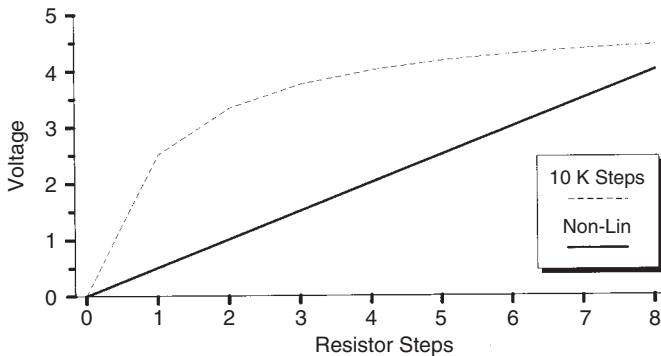


Figure 20.23 Resistor ladder voltage output showing 10k resistor steps versus non linear resistor steps.

The actual output obviously deviates significantly from the desired (linear) output. Finding the correct resistor values is not that difficult. Using the rearranged formula for R_n (from above), we can plug in an R_s value of 10 k Ω , a V_{cc} of 5 V, and then figure how to go from 0 to V_{max} in nine steps.

When I created this application for the first edition of the book, I used a PIC16C84 (which is different from the PIC16F84 used for this edition), and I was able to calculate values very straightforwardly to get a linear output. This was not possible for the PIC16F84, and I found that I had to experiment a bit to get the values specified in Fig. 20.20. These values will produce the reasonable linear steps shown in the oscilloscope picture in Fig. 20.24.

Despite the complexity of wiring this application, one of the really nice things about this circuit is that it can be checked for wiring errors without the PIC installed. This is done by hooking up your DMM and checking the resistances between the I/O pins.

If you're curious, you could rewire the circuit using a constant value for each step in the ladder. If you repeat the experiment with the voltmeter, you will find the asymptotic curve predicted in the preceding graph.

The PIC microcontroller application code (`Vladder.asm`) just runs through the resistor ladder and can be found in the `code\Vladder` folder.

```

title "Vladder - Resistor Ladder Analog Output."
#define DMM
;
; This Program Runs through a Saw Tooth Analog Output from the
; 16C84. The Output is generated by a Resistor Ladder attached
; to PORTB. To set a particular voltage, a bit is output to 0 volts.
;
; Hardware Notes:
; PIC16F84 Running at 4 MHz
; Reset is tied directly to Vcc and PWRT is Enabled.
; The Resistor Ladder is attached to PORTB.7:0
;

```

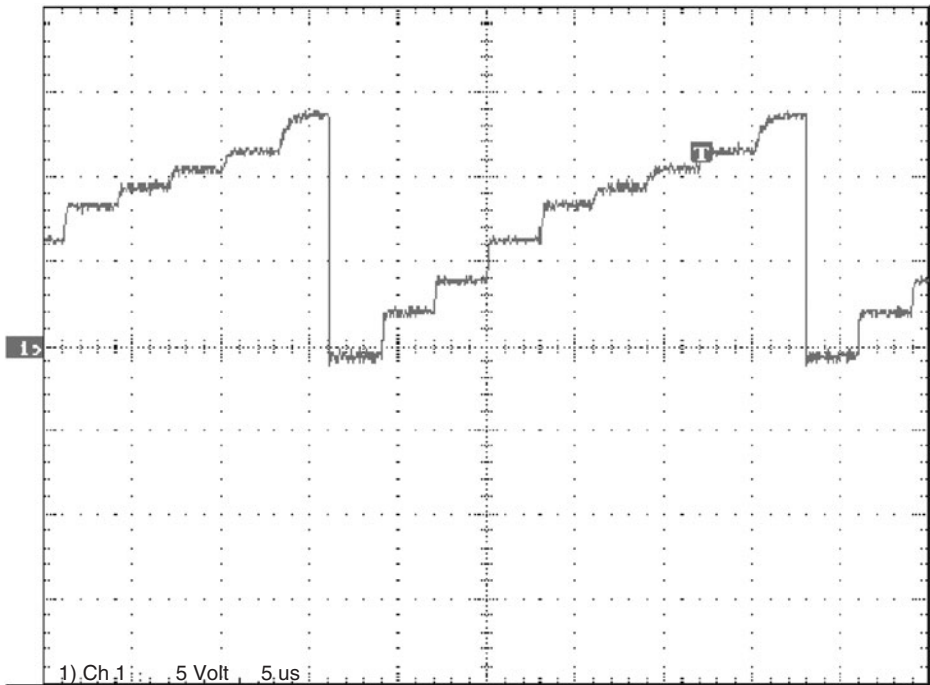


Figure 20.24 The output of the vLadder circuit when looked at with an oscilloscope.

```
; A 4.7K Resistor between PORTB.0 and Vcc (Output is taken from here
; as well)
; A 1K Resistor between PORTB.0 and PORTB.1
; A 2.2K Resistor between PORTB.2 and PORTB.3
; A 2.2K Resistor between PORTB.2 and PORTB.3
; A 3.3K Resistor between PORTB.3 and PORTB.4
; A 3.3K Resistor between PORTB.4 and PORTB.5
; A 4.7K Resistor between PORTB.5 and PORTB.6
; A 10K Resistor between PORTB.6 and PORTB.7
;
; Updated: 99.12.27 - For Second Edition
;
; Myke Predko
; 96.06.27
;
LIST R=DEC
INCLUDE "p16f84.inc"

; Registers
CBLOCK 0x020
Count, Counthi, Countu
```



```

ENDC
__CONFIG __CP_OFF & __WDT_OFF & __XT_OSC & __PWRTE_ON

PAGE
; Code for VLadder

org    0

nop

clrf   PORTB

movlw  TRISB                ; Setup the TRIS Values
movwf  FSR

bcf     STATUS, C           ; Use the Carry as the Skip Value

Loop:                                     ; Loop Around Here to Output Sawtooth

ifdef  DMM                  ; Just Delay if only a DMM Available
    call Delay              ; for Seeing the Output
endif

rlf     INDF, f

goto    Loop

ifdef  DMM
Delay:

    clrf   Count            ; Display for one Second
    clrf   Counthi
    movlw  6
    movwf  Countu
Dlay:
    decfsz Count, f
    goto  Dlay
    decfsz Counthi, f
    goto  Dlay
    decfsz Countu, f
    goto  Dlay

    return
endif

end

```

The code for this application is designed with the DMM conditional code. If the DMM `#define` is used, the 1-second `Delay` subroutine is called so that you can watch the voltage change on a DMM. If DMM is not defined, then the application will run with the voltage changing once every 3 ms, and the waveform can be observed on an oscilloscope, such as the oscilloscope picture shown in Fig. 20.24.

As you step through the code (make sure that you don't define DMM when doing this, or you'll be clicking your mouse a lot, and you will watch the zero shift through TRISB, which causes PORTB to act like an programmable open collector output and change the effective voltage divider resistance. It is important to note that this code takes advantage of the value in the carry flag not changing between `Loop` iterations. This was important when specifying the `Delay` code to make sure that no add or subtract instructions (which could modify the `##`) were used.

When I created this circuit for the second edition, I must confess that I had some significant problems. The first and most obvious one was that the schematic presented in the first edition was not correct and did not match the application code presented with it. This experiment is correct and has been tested in a number of different devices.

I also had a lot of problems replicating what, I saw in the first edition in this edition. It took me several hours to reconcile the actual results, but I can say confidently that the circuit diagrams presented here will work with the application code to produce a high-frequency sawtooth. The information contained in this experiment was used as a base to produce the NTSC composite video output described in Chap. 21.

I/O with Interrupts

So far in these experiments I have focused on the basic hardware that is built into both low-end and mid-range PIC microcontrollers. These experiments are very important because they will help you to understand how the PIC microcontroller works in a variety of situations or how the PIC microcontroller responds to different hardware inputs.

In the next series of experiments, I want to look at using the mid-range PIC microcontroller's interrupt capability to enhance applications and, in many cases, simplify the code and operation of the application. This probably seems surprising to you, but interrupts can make applications much easier to work with. In Chap. 21 I will show how a serial LCD interface board can be built without interrupts or serial interface hardware, but the code is quite complex and very accurately timed.

Also in Chap. 21 I will go through a lot more of the various interrupt sources and operations than I will in this chapter. In this chapter I want to introduce how interrupts work in mid-range PIC microcontrollers and how handlers are written for them.

Cylon: TIMER/INTERRUPT HANDLER WITH CONTEXT SAVING

Throughout this book I have pointed out the advantages of using the TMR0 interrupt source for carrying out background tasks. Later in this chapter I will show a TMR0 interrupt button debounce application that debounces button input without affecting the

operation of the mainline code. To help introduce interrupts and the TMR0 interrupt, I have created a little interrupt handler that moves to lit LEDs back and forth in kind of a “Cylon eye” after a $\frac{1}{8}$ -s delay.

The name “Cylon eye” comes from eye movement of the the metal warriors of *Battlestar Gallactica*. To be honest, the name comes from the original TV show (done in the 1970s), but it has been used for years in different applications to show that the circuit is working properly. I wanted to create the code so that it could be ported into a PIC microcontroller application that supervises the operation of a larger circuit. The “Cylon eye” application uses PORTB connected to a series of LEDs, with TMR0 providing an interrupt request once every sixteenth of a second. The sixteenth of a second is the maximum delay for a PIC microcontroller running at 4 MHz. This leaves the “mainline” or “foreground” execution open to application monitoring code that can stop the “eye” if any problems are discovered.

The circuit itself simply consists of eight LEDs connected to PORTB of the core circuit, as shown in Fig. 20.25 (the bill of materials is listed in Table 20.11). For the breadboard application, I used a bargraph LED to allow simple continuity of the display, as shown in Fig. 20.26.

The application code is `Cylon.asm` and can be found in the `code\Cylon` folder. Along with the context-saving registers, two variables are required for the programmable delay and to keep track of the direction in which the eye is moving.

```

title "Cylon - Output a Cylon Eye in the Background"
#define nDebug
;
; This Application uses TMR0 to Move a "Cylon Eye" back and forth
; Across eight LEDs connected to PORTB.
;

```

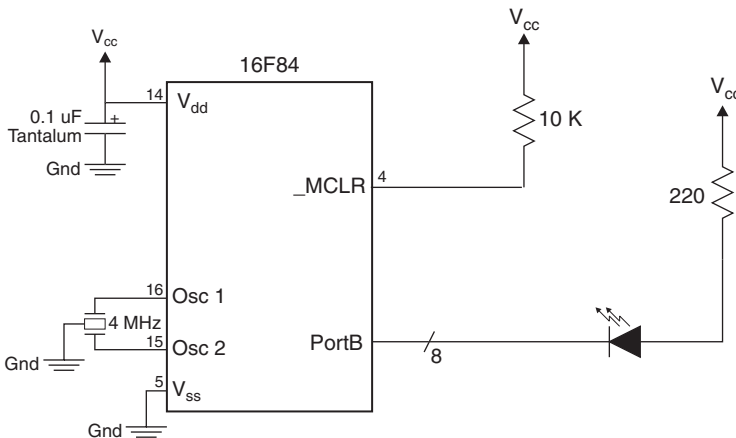


Figure 20.25 Circuit to move a lighted LED like a Cylon warrior’s scanner.

TABLE 20.11 BILL OF MATERIALS FOR THE “CYLON” EXPERIMENT	
PART	DESCRIPTION
PIC	PIC16F84-04/P
0.1-μF	0.1-μF tantalum
10k	10 kΩ, ¼ W
4-MHz	4-MHz with built-in capacitors
LED	10-LED bargraph
220	220-Ω, ¼ W

```
; Hardware Notes:
; This application runs on a 16F84 executing at 4 MHz
; _MCLR is tied through a 10K Resistor to Vcc and PWRT is Enabled
; A 220 Ohm Resistor and LED is connected between PORTB.0 and Vcc
;
; Myke Predko
; 99.12.28
;
LIST R=DEC
INCLUDE "p16f84.inc"

; Registers
CBLOCK 0x020
_w, _status           ; Context Register Save Values
```

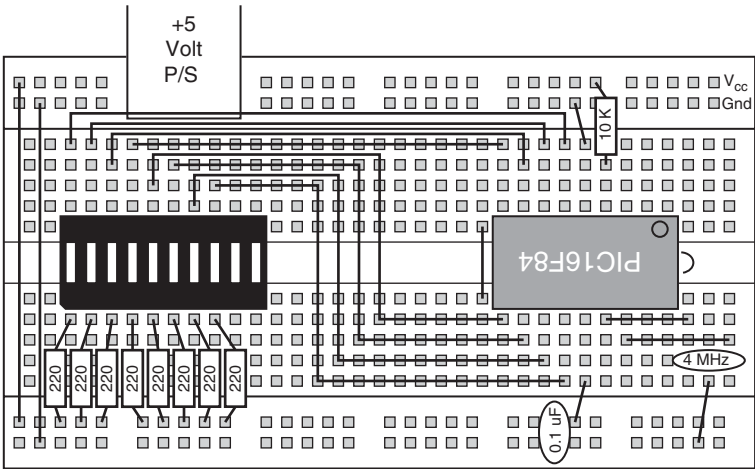


Figure 20.26 Breadboard wiring for the “Cylon eye.”

```

Direction          ; 0 for Up, !0 for Down
Count              ; Count the Number of Times Through
ENDC

__CONFIG _CP_OFF & _WDT_OFF & _XT_OSC & _PWRTE_ON

PAGE
; Mainline of cylon

org      0

nop

movlw    2          ; Setup the Count
movwf    Count

goto     Mainline

org      4
Int:      ; Interrupt Handler
movwf    _w         ; Save Context Registers
movf     STATUS, w   ; - Assume TMR0 is the only enabled Interrupt
movwf    _status

bcf       INTCON, T0IF ; Reset the Interrupt Flag

decfsz   Count, f    ; Execute Once Every Two times
goto     IntEnd

movlw    2          ; Reset the Counter
movwf    Count

btfss    PORTB, 7     ; At the Top?
bsf       Direction, 0

btfss    PORTB, 0     ; At the Bottom?
bcf       Direction, 0

btfsc    Direction, 0 ; Going Up?
goto     Down        ; No, Down

Up:      ; Moving the LEDs Up

bsf       STATUS, C   ; Set the Status Flag
rlf       PORTB, f    ; Shift the Data Up

goto     IntEnd

```

Down:

```
bsf    STATUS, C    ; Shift the Data Down
rrf    PORTB, f
```

IntEnd:

```
movf   _status, w    ; Restore the Context Registers
movwf  STATUS
swapf  _w, f
swapf  _w, w

retfie
```

Mainline: ; Setup PWM And Monitor it, Updating "PWMon"

```
clrf   Direction

movlw  0x0E7          ; Start in the Middle
movwf  PORTB

bsf    STATUS, RP0     ; Goto Bank 1 to set Port Direction
clrf   TRISB ^ 0x080   ; PORTB is Output
ifdef  Debug
movlw  0x0D0           ; Debug - Minimum Timer Delay
else
movlw  0x0D7           ; Normal Operation - Maximum Timer
endif
movwf  OPTION_REG ^ 0x080
bcf    STATUS, RP0     ; Go back to Bank 0

clrf   TMR0           ; Start the Timer from Scratch

movlw  (1 << GIE) + (1 << T0IE)
movwf  INTCON          ; Enable Interrupts
```

Loop: ; Loop Here

```
goto   Loop           ; Let Interrupt Handler Work in the Background
```

end

There are two areas of the application that I would bring to your attention. The first is the use of a prescaler for the application, and the second is the Count variable for delaying the output. I use the prescaler set to the maximum delay for the TMR0 input (including using the instruction clock as the TMR0 input source) to delay the interrupts as long as possible. In this configuration, the delay is one sixty-four-thousandth of the

instruction clock. For a PIC microcontroller running at 4 MHz, this works out to an interrupt interval of 16 times per second. I used the prescaler on this application because it simplified the `Count` variable's operation.

By using the prescaler for TMR0 in this application, I am not indicating that I feel that the watchdog timer should not be used with this application. If the eye were used for indicating the health of a larger circuit, I would presume that the watchdog timer would be used to indicate if the PIC microcontroller has had a problem in its execution. In this case, the prescaler could be assigned to the watchdog timer, and the `Count` variable could be increased to 16 bits to perform the same function.

Use of the `Count` variable is a result of the maximum delay being put on TMR0 as not being acceptable for the application. With the $\frac{1}{16}$ -s maximum delay, I found movement of the eye to be somewhat manic. By placing the `Count` variable and counting down from 2, I have delayed the operation to something reasonable. For PIC microcontrollers that are clocked with higher frequencies, this variable can be given a higher value to maintain the speed of the eye going back and forth. In addition, if the prescaler is to be used with the watchdog timer, then `Count` could be used to increase the number of times TMR0 overflows before the eye's position is updated.

I originally used the prescaler set to the largest value to minimize the impact the interrupt handler had on the mainline code. Eliminating the prescaler input will result in fewer instructions available to the mainline code for system monitoring.

TMR0Int: SIMULATING INPUT PIN INTERRUPT WITH TIMER PIN INPUT

Some time ago a question came up on the PICList asking whether or not TMR0 could be used as an interrupt input pin if it were set up in such a way that the next transition would cause an overflow condition and request that the interrupt be handled. I discovered that it is indeed possible to use TMR0 as an interrupt input pin. Using TMR0 in this fashion is something that could be very desirable in the cases where no more interrupt source pins are left available or if a low-end PIC microcontroller is being used, and rather than polling an input, a latch (such as TMR0) could be used to detect a transition. This experiment shows how TMR0 can be set up to request an interrupt after the first incoming data transition.

The circuit is quite simple, as you can see in the schematic in Fig. 20.27 (the bill of materials is listed in Table 20.12) and the breadboard assembly drawing in Fig. 20.28.

The code is also very simple and is known as `TMR0Int` and can be found in the `code\TMR0Int` folder.

```

title "TMR0Int - Treat TMR0 Input like an Interrupt Input"
;
; This Application uses the TMR0 Input Pin (RA4 in the mid-range
; PIC microcontrollers) as interrupt source. When input changes,
; TMR0 overflows, which causes an interrupt request.
;

```

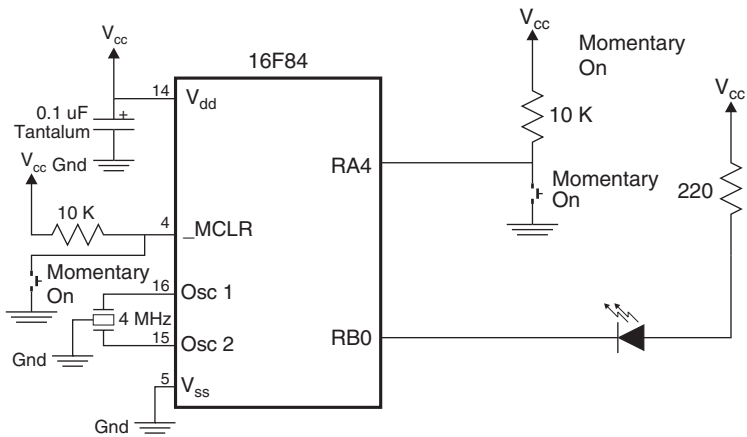


Figure 20.27 Simple circuit to demonstrate the ability to use TMR0 as a digital input that creates an interrupt request.

```
; Hardware Notes:
; This application runs on a PIC16F84 executing at 4 MHz
; _MCLR is tied through a 10K Resistor to Vcc and PWRT is Enabled
; A 10K Pull-Up and a Momentary "On" Switch is Connected to RA4
; A 220 Ohm Resistor and LED is connected between PORTB.0 and Vcc
;
; Myke Predko
; 99.12.28
;
```

TABLE 20.12 TMR0Int BILL OF MATERIALS	
PART	DESCRIPTION
PIC	PIC16F84-04/P
0.1-μF	0.1-μF tantalum
10k	10 kΩ, ¼ W
Momentary-on switch	Breadboard-mountable momentary-on switch
4-MHz	4 MHz with built-in capacitors
LED	Red LED
220	220-Ω, ¼ W
Misc.	Breadboard, wiring, +5-V power supply

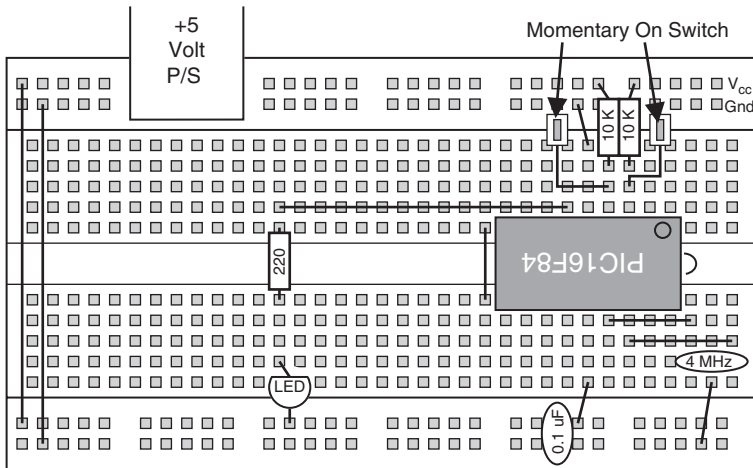


Figure 20.28 The breadboard circuitry for TMR0Int.

```

LIST R=DEC
INCLUDE "p16f84.inc"

; Registers
CBLOCK 0x020
_w, _status                               ; Context Register Save Values
ENDC

__CONFIG __CP_OFF & __WDT_OFF & __XT_OSC & __PWRTE_ON

PAGE
; Mainline of cylon

org    0

nop

movlw  0x0FF
movwf  PORTB

goto   Mainline

org    4
Int:                                       ; TMR0 has Overflowed - New Input

bcf    INTCON, T0IF                       ; Reset the Interrupt Flag

bcf    PORTB, 0                           ; Turn on the LED

```

```
    retfie

Mainline:                                ; Setup TMR0 Interrupt

    bsf    STATUS, RP0      ; Goto Bank 1 to set Port Direction
    bcf    PORTB, 0         ; Enable RB0 for Output
    bcf    STATUS, RP0      ; Go back to Bank 0

    movlw  OPTION_REG       ; Point to the Option Register
    movwf  FSR

    clrf   TMR0             ; Reset the Timer

    movlw  (1 << GIE) | (1 << T0IE)
    movwf  INTCON           ; Enable Interrupts

    movlw  0x0C0            ; Make TMR0 Driven by the Instruction
                                ; Clock
    movwf  INDF

    movlw  0xFF
    movwf  TMR0

    bsf    INDF, T0CS        ; Now, Make TMR0 Driven Externally

Loop:                                ; Loop Here

    goto   Loop             ; Let Interrupt Handler Work in the
                                ; Background

end
```

This code will seem somewhat conventional except for my passing the address of the OPTION register to FSR and updating it from there rather than the usual way of updating the register while bank 1 is being accessed. I used this code because the time required to set TMR0 and then jump to bank 1 would be much more difficult to time than the code I used.

In this example code, TMR0 is initially given the value of 0xFF and waits to be updated by the instruction clock. As I indicated earlier in this book, TMR0 has a two-cycle clock counter (or synchronizer) that must have two inputs before it passes one on to TMR0. This synchronizer, like the prescaler, is reset any time TMR0 is written to. This means that it will overflow and request an interrupt two instructions after it is set. For this application, I want TMR0 to overflow after the first external input, so to set up this condition, I want one instruction between the instruction when TMR0 is loaded with 0xFF and when I change its input source to RA4.

My solution is to use the index register (FSR) to point to `OPTION` and to change the source (by resetting the `T0CS` bit) using the `bcf` instruction. When I have done this before, I have used the `option` instruction, and if I were implementing this code in a low-end PIC microcontroller, I would have to use the `option` instruction. The code here avoids using the not-recommended `option` for mid-range PIC microcontrollers.

When using this code in low-end PIC microcontrollers, remember *not* to use the `movf TMR0, f` instruction and instead use the `movf TMR0, w` instruction. Both will set the `STATUS` register's zero flag, but the first instruction will reload `TMR0` with the current value, and if it is `0xFF`, it will reset the two-cycle timer, which means that two valid interrupt inputs will be required to change the state. What's worse is that if `TMR0` is polled before the two interrupt inputs are received, it will result in the `TMR0` *never* resetting and incrementing for the interrupt source.

This application is very focused on the single task of latching the LED connected to `RB0` on when the input on `RA4` goes low, and for this reason, I didn't bother with a complicated interrupt handler (in fact, I dispensed with context register saving and restoring). In a real application, the interrupt handler code probably would carry out the code for setting `TMR0` as an interrupt source after acknowledging that the interrupt has been received.

LedPWM: TMR0 INTERRUPT USED FOR LED PWM DIMMING

To demonstrate how `TMR0` and interrupts can be used to control the brightness of a LED by producing a PWM signal, I will use the simplest application circuit of this book. This circuit, shown in Fig. 20.29 (with the bill of materials listed in Table 20.13), interfaces with only one LED, which will get brighter and then turn off and repeat using the `LedPWM` code that can be found in the `code\ledpwm` folder.

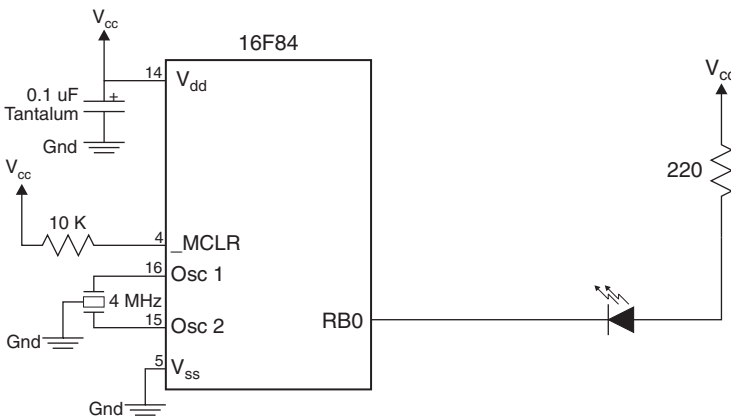


Figure 20.29 Circuit to demonstrate how a PWM can be used to dim an LED.

TABLE 20.13 LedPWM CIRCUIT BILL OF MATERIALS	
PART	DESCRIPTION
PIC	PIC16F84-04/P
0.1-μF	0.1-μF tantalum
10k	10 kΩ, ¼ W
4-MHz	4 MHz with built-in capacitors
LED	Red LED
220	220-Ω, ¼ W
Misc.	Breadboard, wiring, +5-V power supply

On reset, the PIC microcontroller’s RB0 pin is enabled for output, and the TMR0 interrupt is enabled. Once this is done, an incrementing LED-power PWM signal is set and is passed to an interrupt handler that outputs a pulse of varying timing widths every 1 ms.

```
title "LedPWM - Show an LED Changing Brightness"
;
; This Application simply waits for the TMR0 interrupt handler to
; occur and after the LED PWM "ON" is complete, decrement the "ON"
; value. This Application repeats endlessly with the LED getting
; lighter before turning off and starting over.
;
; Hardware Notes:
; This application runs on a 16F84 executing at 4 MHz
; _MCLR is tied through a 4.7K Resistor to Vcc and PWRT is Enabled
; A 220 Ohm Resistor and LED is attached to PORTB.0 and Vcc
; A 10K pull up is connected to RA0 and its state is passed to
; RB0
;
; Myke Predko
; 99.12.14
;
LIST R=DEC
INCLUDE "p16f84.inc"

; Registers
CBLOCK 0x020
_w, _status
PWMOn:2
PWMDouble: 2
ENDC
#define PWM PORTB, 0
```

```
; Context Register Save Values
; PWM "On Value"
; Divide PWM down for Slowing Down
; LED on PORTB.0
```

```

__CONFIG __CP_OFF & __WDT_OFF & __XT_OSC & __PWRTE_ON

PAGE
; Mainline of ledpwm

org      0

nop

clrf     PWMDouble
clrf     PWMDouble + 1

goto     Mainline

org      4
Int:      ; Interrupt Handler

movwf    _w          ; Save Context Registers
movf     STATUS, w   ; - Assume TMR0 is the only
                    ; enabled
movwf    _status     ; Interrupt

btfsc    PWM          ; Is PWM O/P Currently High or Low?
goto     PWM_ON
nop      ; Low - Nop to Match Cycles with
        ; High

bsf       PWM          ; Output the Start of the Pulse

movlw    6 + 6         ; Get the PWM On Period
subwf    PWMOn, w      ; Add to PWM to Get Correct Period
        ; for
        ; Interrupt Handler Delay and
        ; Missed
        ; cycles in maximum 1024  $\mu$ sec
        ; Cycles

goto     PWM_Done

PWM_ON:   ; PWM is On - Turn it Off

bcf       PWM          ; Output the "Low" of the PWM Cycle

movf     PWMOn, w      ; Calculate the "Off" Period
sublw    6 + 6         ; Subtract from the Period for the
        ; Interrupt Handler Delay and
        ; Missed
        ; cycles in maximum 1024  $\mu$ sec
        ; Cycles

```

```
    goto    PWM_Done

PWM_Done:                                ; Have Finished Changing the PWM
                                           ; Value

    sublw   0                            ; Get the Value to Load into the
                                           ; Timer
    movwf   TMR0

    bcf     INTCON, T0IF                  ; Reset the Interrupt Handler

    movf    _status, w                    ; Restore the Context Registers
    movwf   STATUS
    swapf   _w, f
    swapf   _w, w

    retfie

Mainline:                                ; Setup the PWM And then Monitor it,
                                           ; Updating "PWMon"
    bsf     PORTB, 0                      ; Make the LED on PORTB.0 "off"
                                           ; Initially
    bsf     STATUS, RP0                    ; Goto Bank 1 to set Port Direction
    bcf     TRISB ^ 0x080, 0              ; Set RB0 to Output
    movlw   0x0D1                          ; Setup TMR0 with a 4x prescaler
    movwf   OPTION_REG ^ 0x080
    bcf     STATUS, RP0                    ; Go back to Bank 0

    clrf    TMR0                           ; Start the Timer from Scratch

    movlw   (1 << GIE) + (1 << T0IE)
    movwf   INTCON                          ; Enable Interrupts

Loop:                                          ; Loop Here

    btfsc   PWM                            ; Wait for PWM to go Low
    goto    $ - 1

    incfsz  TMR0, w                          ; Wait for TMR0 to Equal 0xFF
    goto    $ - 1

    movf    PWMDouble, f                    ; Decrement PWM Double
    btfsc   STATUS, Z
    decf    PWMDouble + 1, f
    decf    PWMDouble, f
    rrf     PWMDouble + 1, w                ; Divide by 4
    movwf   PWMon + 1
    rrf     PWMDouble, w
    movwf   PWMon
```

```

rrf    PWMOn + 1, w
rrf    PWMOn, w

addlw  255 - 0x0FE          ; Get the High limit
addlw  0x0FE - 0x00E + 1    ; Add Lower Limit to Set Carry
btfss  STATUS, C            ; If Carry Set, then Lower Case
    movlw 0x0FC              ; Carry NOT Set, Reset the
                                ; Character
addlw  0x00E                ; Add Lower Limit to restore the
; Character
movwf  PWMOn                ; Save the New Value

btfss  PWM                  ; Wait for the PWM to Go High
    goto $ - 1

goto   Loop

end

```

The PWM signal-generating code in the interrupt handler can be described using the following pseudocode:

```

Interrupt PWMOutput()      // When Timer Overflows, Toggle "On" and
                           // "Off"
{                           // and Reset Timer to correct delay for
                           // Value

    if (PWM == ON) {       // If PWM is ON, Turn it off and Set Timer
        PWM = off;        // Value
        TMR0 = PWMPeriod - PWMOn;
    } else {               // If PWM is off, Turn it ON and Set Timer
        PWM = ON;         // Value
        TMR0 = PWMOn;
    } // fi

    INTCON.T0IF = 0;       // Reset Interrupts

} // End PWMOutput TMR0 Interrupt Handler

```

This operation first outputs a high on the PWM pin for the specified number of TMR0 ticks (up to 250). When TMR0 overflows and requests an interrupt, the PWM pin is set to low, and TMR0 is loaded with the number of ticks remaining for the 250-instruction-cycle PWM signal period.

Generation of the TMR0 initial value is somewhat unusual and may be somewhat difficult to follow. I suggest that you go back to the “Pulse Width Modulation (PWM) I/O” section of Chap. 17 to see how the add and two subtracts works for loading TMR0 with the correct value.

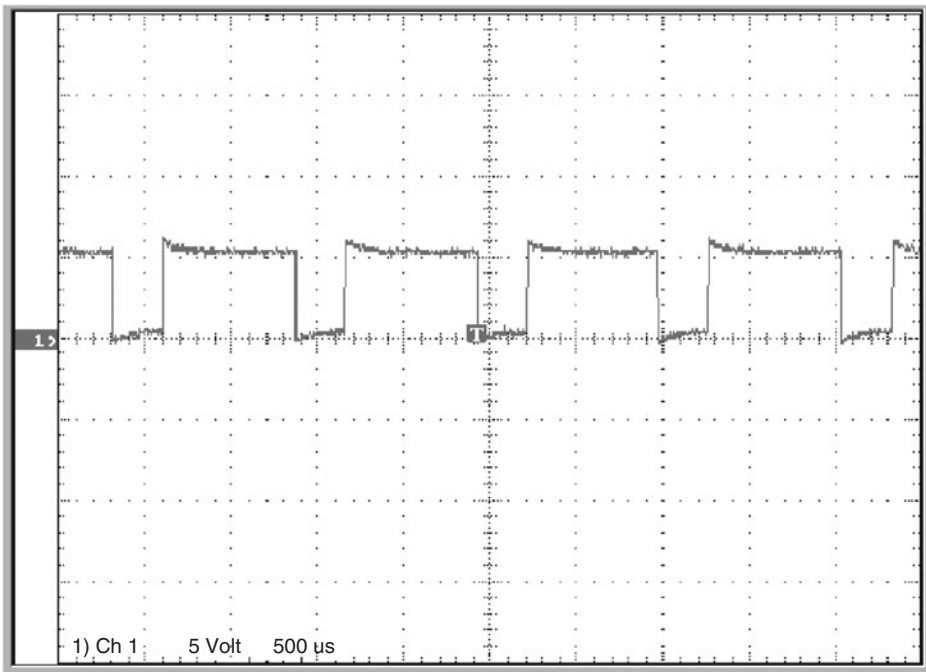


Figure 20.30 The LED PWM waveform.

I would recommend that you first build first and try out this application before reading on. When you run the application, you should see the LED connected to RB0 increase in brightness to a certain point and then turn off and start again. The entire process takes about 1 second. If you were to look at the PWM signal sent to the LED, you would see something like Fig. 20.30 transitioning to a high or low value with short pulses. A mostly low signal will translate to the LED being very bright.

It probably will be surprising to you to discover that the most difficult code that I had to write for the experiment was the code used for updating PWMOn (which is used to select the PWM signal's pulse width). When I created this application originally, I wanted to simply decrement the PWMOn variable to make the off time of the output PWM signal longer as time goes on, which, in turn, makes the LED brighter as time goes on. To do this, I originally wanted to use code that can be shown as the following pseudocode:

```
while (1 == 1) {                                // Loop Forever

    while (PWM == ON);                          // Wait for Pulse to Finish

    PWMOn = PWMOn - 1;                          // Decrement the Pulse Width

    while (PWM == off);                        // Wait for the "Off" to Finish

} // elihw
```


which simply waits for an interval between pulses in which the PWMOn value could be changed. The following assembly-language code was created for this function:

```

Loop:                                ; Return here forever

    btfsc    PWM                    ; Loop while PWM == ON
    goto    $ - 1

    decf     PWMOn, f               ; Decrement the Pulse Width

    btfss    PWM                    ; Loop while PWM == off
    goto    $ - 1

    goto     Loop                  ; Loop Again

```

When I simulated the application, the output seemed to be correct for 10 ms (the length of time I simulated the operation for 10 PWM pulses). When I programmed a PIC microcontroller and tried out the application, I found that the LED flashed briefly on power-up and then stayed dark. When I put an oscilloscope probe on the LED, I found that the PWM signal was high with a few short pulses.

I then simulated the application beyond the first 10 ms and discovered that when the value put into TMR0 was very high (i.e., 0xFF), a new interrupt request from TMR0 would be received before the interrupt handler had returned to the mainline (Loop) code. Execution jumped immediately back into the interrupt handler and changed the polarity of the PWM output pin without giving the mainline code a chance to recognize the change. To fix this, I then monitored the values of PWMOn and made sure that it never became less than 0x0C, at which point PWMOn was reset to 0xF4 to restart itself.

This change to the code did fix the problem, but the LED flashed on and off very quickly (about four times per second). To make the display a bit more user-friendly, I decided to use a 16-bit counter for the PWMOn value (this is PWMDouble) and shift it down to slow down the actual PWM display. I found that I had to do this twice (divide the value by 4) to get an appropriate value. After PWMDouble is divided by 4, I check the range and reset the output to a high value to make the full on time a bit longer. You can see the results of this code in the Loop section at the end of the LedPWM application.

This was not the best way to change PWMOn. Along with being quite complex, requiring three additional file registers for temporary values, it also took me a long time to figure out exactly what should be the values for the range check and reset for the code. A much better way would have been simply to wait 4 ms (the time for four pulses to be output) and then change PWMOn. This method would not require the range checking of the method that I used because the updates do not depend on the changes in the PWM bit.

The code for implementing the changing PWM using the method of simply delaying by 4 ms is

```

Loop:                                ; Loop Here

```

```
    decf    PWMOn, f           ; Decrement the PWM Value

    movlw   4                  ; Delay 4 msecs
    movwf   Dlay
    movlw   0x020
    addlw   0 - 1              ; Decrement the Value
    btfsc   STATUS, Z          ; Using the 5 Cycle Delay Loop
    decfsz  Dlay, f
    goto    $ - 3

    goto    Loop
```

The application source code for `LedPWM` using this method is called `lpbetter` and is located in the `code\ledpwm` folder.

This method uses less than 40 percent of the code of the original method and half the file registers. It also took me about 2 minutes to make the changes to the original application to come up with `lpbetter`.

I've left the original application in place as a monument to stubbornness. To get the original `LedPWM` code working, I spent almost 2 hours trying different tricks before settling on the 16-bit counter, the shift down by 4, and the range check and update. Instead of doing all this work, I really should have been thinking of a better way to implement the code.

The lesson I want to impart to you is that if you are having trouble with something; instead of plowing through and trying to get an application to work, take a few moments and reflect on what you are really trying to do and see if there is a better way of doing it. My problem was that I wanted to be clever and use the interrupt handler's 1-ms operation for the high and low of the PWM interrupt to provide me with a counter for changing the PWM signal control variable (`PWMOn`). The much more efficient method of implementing the changing `PWMOn` value was simply to put in my own delay counter in the mainline and ignore what was happening in the interrupt handler.

Experiences such as this are *lessons*; they are not *mistakes*. Don't get mad at yourself if you miss an obvious opportunity like what I have shown here; instead, remember it for the next time you get in a situation where you are struggling to get an application to work, and try to think up a better way of doing it. Chances are that you'll come up with something pretty spectacular.

IntDeb: DEBOUNCING INPUTS WITH INTERRUPTS

When using interrupts for debouncing applications, the decision has to be made on what is the interface to the mainline. In some applications, it may be appropriate to initiate the action of the button read, whereas in others, the interrupts built into the PIC microcontroller can be used to provide a debounced button function that runs in the background and can be checked as required. The `IntDeb` application uses the `RB0/INT` pin and `TMR0` to provide a background debounced button input that can be polled by the mainline application as required.

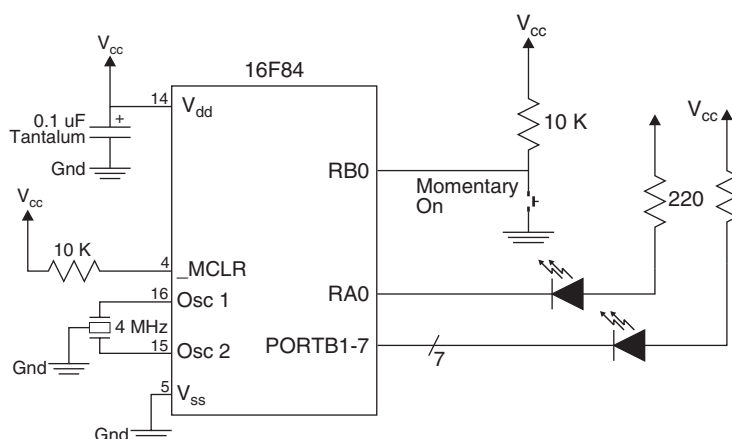


Figure 20.31 Circuit to demonstrate how to debounce inputs with an interrupt and TMR0.

The circuit for `IntDeb` is a bit unusual because for the 8-bit output by the PIC microcontroller, I have split them up between PORTA and PORTB. The reason for doing this is expediency because PORTB.0 is required for the button interrupt input. As will be seen in the application code, this does not cause a major problem for the application. The schematic for the `IntDeb` experiment is shown in Fig. 20.31, and the bill of materials is listed in Table 20.14.

The need for the RB0/INT pin to be used as an input makes wiring this application somewhat different from the other applications with which you have worked. The least significant bit displayed has been moved from RB0 to RA0.

TABLE 20.14 INTERRUPT DEBOUNCE BILL OF MATERIALS

PART	DESCRIPTION
PIC	PIC16F84-04/P
0.1- μ F	0.1- μ F tantalum
10k	10 k Ω , $\frac{1}{4}$ W
4-MHz	4 MHz with built-in capacitors
220	220 Ω , $\frac{1}{4}$ W
LED bargraph	10-LED bargraph
Momentary-on switch	Breadboard-mountable momentary-on switch
Misc.	Breadboard, wiring, +5-V power supply

The `IntDeb.asm` code, found in the `code\IntDeb` folder, is

```
title "IntDeb - Register Contents Int Debounce."
#define nDebug
;
; This Program reads the value in a RAM Register and outputs it
; inverted onto PORTB (which has LEDs to Display the Value). All the
; RAM Registers are Read and Displayed. A button is used as the
; instigator of the next value read. The FSR is Copied into the LEDs
; to Display the current Register Being Displayed.
;
; This program is a modification of PROG18.ASM to use the Interrupt
; Handler to Debounce the Button Input.
;
; Hardware Notes:
; Reset is tied directly to Vcc and PWRT is Enabled.
; A 4.7K Pullup and Switch Pull-Down is attached to PORTB.0
; A 220 Ohm Resistor and LED is attached to PORTB.7:1
; A 220 Ohm Resistor and LED is attached to PORTA.0
;
; Updated for the Second Edition: 99.12.28
;
; Myke Predko
; 97.02.22
;
LIST R=DEC
INCLUDE "p16f84.inc"

; Registers
CBLOCK 0x020
Flags
Reg                                ; Register to Display
ENDC

#define ButUp   Flags, 0           ; Flags Indicating Button State
#define ButDown Flags, 1

__CONFIG __CP_OFF & __WDT_OFF & __XT_OSC & __PWRT_ON

PAGE
; Mainline of IntDeb

org 0

nop

clrf   Flags                        ; No Button Pressed Yet
```

```

    clrf    Reg

    goto    MainLine

    org 4                                ; Interrupt Handler Address
Int:

    btfss   INTCON, T0IF                ; Do we have a Timer Overflow?
    goto    Int_Switch                 ; No - Handle the Switch

    bcf     INTCON, T0IF                ; Yes, Reset Timer Interrupt Request

    movlw   0x001                      ; Assume the Button is Up
    btfss   PORTB, 0                   ; Is the Button Up or Down?
    movlw   0x002                      ; - If PORTB is Low, Button Down
    movwf   Flags

    goto    Int_End

Int_Switch:                             ; Interrupt on the Switch - Reset Timer

    bcf     INTCON, INTF               ; Reset the Interrupt Request

    clrf    Flags                     ; Indicate that Nothing is Valid

    movf    PORTB, 0                  ; What is the Button State?
    andlw   1
    bsf     STATUS, RP0
    bcf     OPTION_REG ^ 0x080, INTEDG
    btfsc   STATUS, Z                  ; Determine the Actual Edge
    bsf     OPTION_REG ^ 0x080, INTEDG
    bcf     STATUS, RP0

    clrf    TMR0                      ; Going to Wait for another Key Press

Int_End:

    retfie

PAGE
MainLine:                             ; Mainline of reading the File Registers

    movlw   0xFF
    movwf   PORTB                     ; Turn off all the Indicator LED's
    movwf   PORTA

    movf    PORTB, w

```

```
    andlw 1                      ; Set 0 if Pin Down

    bsf   STATUS, RP0
    bcf   TRISA ^ 0x080, 0      ; RA0 is Output
    movlw 1                      ; RB0 is Input/Interrupt
    movwf TRISB ^ 0x080        ; Set PORTB.7:1 bits to Output
    ifndef Debug
        movlw 0x090              ; Zero Prescaler for TMR0 if Debug
        btfsc STATUS, Z
        movlw 0x0D0              ; Pin down, Go for the Rising Edge
    else
        movlw 0x096              ; Setup Prescaler for TMR0 to 32.7 msec
        btfsc STATUS, Z
        movlw 0x0D6              ; Pin down, Go for the Rising Edge
    endif
    movwf OPTION_REG ^ 0x080
    bcf   STATUS, RP0

    clrf  TMR0                  ; Reset TMR0

    movlw (1 << GIE) | (1 << T0IE) | (1 << INTE)
    movwf INTCON                ; Setup the Interrupt Delays

Loop:                                ; Loop to Here for Each Register

    btfss ButUp                  ; Wait for the Button to be Debounced UP
    goto $ - 1

    btfss ButDown                ; Wait for Button be to Debounced Down
    goto $ - 1

    incf  Reg, f                 ; Increment the Counter

    comf  Reg, w                 ; Display the Counter Value
    movwf PORTB
    movwf PORTA

    goto Loop

end
```

There should not be too many surprises in this application for you (although there should be one big one that I will discuss below). The mainline code sets up the interrupts along with the prescaler for TMR0 to cause an interrupt after 32.7 ms. I used this interval for the debounce interval (if no button changes have happened within 32.7 ms,

then the `Flags` variable bits are set appropriately). The reason for this interval is that the bouncing is greater than the rule-of-thumb 20 ms and did not require any different values to be set in `TMR0`—I could simply reset it to restart the delay.

The interrupt handler code should not be surprising either. It could be written out in pseudocode as

```
interrupt RB0Debounce()    // Debounce the Button on RB0
{
    if (T0IF != 0) {        // Timer Interrupt (32.7 msec Delay)
        if (Button == 0) {  // Button Pressed Down
            ButUp = 0; ButDown = 1;
        } else {            // Button Released Up
            ButUp = 1; ButDown = 0;
        } // fi
        TMR0 = 0;           // Reset the Timer
        T0IF = 0;
    } else {                // Button State Changed
        if (Button = 0)     // Wait for Next Button Based on Current
                            // State
            OPTION.INTEDG = 1; // Button Down, Wait for going Up
        else
            OPTION.INTEDG = 0;
        INTF = 0;           // Reset Button State
        TMR0 = 0;           // Reset the Timer
    } // fi
} // end RB0Debounce
```

The advantages of this method are that it can be very crisply defined and written in a high level language (this pseudocode could be ported directly to C), and it does not affect the operation of the mainline code or even other interrupts. If you look back at the interrupt handler in `IntDeb` earlier, you should be able to match this code very clearly to the pseudocode above.

Note that in this application I keep track of the current state of the button and change the `INTEDG` flag of the `OPTION` register accordingly. For button debounce routines that are interrupt-based, I would prefer to use the “interrupt on port change” function of the PIC microcontroller’s `PORTB` pins and ignore this. In this case, I could have created two variables that are set with the `PORTB` input states when `TMR0` overflowed and requested an interrupt. In this case, the interrupt handler itself would be a bit simpler, and more than one button could be debounced within an application.

When writing the counter value out to the LEDs, note that I arranged it so that the least significant bit (which is lost in `PORTB` owing to the `RB0/INT` pin) is replicated in `PORTA`. This means that I don’t have to change the bit number of the count register, which simplifies the code and the application quite a bit. Obviously, this action

is further simplified by not using any of the other PORTA pins, but even if they were, the code to update RA0 based on the least significant bit of the counter (Reg) would be

```
comf    Reg, w
movwf   PORTB        ; Write Most Significant 7 Bits of "Reg"
movf    PORTA, w      ; Clear the LSB of PORTA
andlw   0x01E        ; Before setting if LSB of Reg requires it
btfss   Reg, 0        ; If LSB Set, then Output Reset for LED On
iorlw   1             ; LSB Reset, Set output
movwf   PORTA
```

which is not very difficult to implement and does not require any thinking on how to pass register bits back and forth. This is what I mean when I say that optimization takes place during application design and not after the circuit is created and the code is being written.

The problem with the interrupt handler is that I didn't bother putting in context register saves or restores. In the `IntDeb` application, where the button `Flags` values are polled without executing any other code, this is not a problem. However, if this button debounce code were to be used in another application that executed code while buttons could be pressed, this will result in some very difficult to find and debug problems.

What I want to impress you with is the idea that when you create code that could be reused, try to make it as generic as possible. In this case, the context saves are not required, so I skipped the seven instructions or so and two file registers that are required for that function. The problem comes in if I want to use the code in another application, and when it fails intermittently, then the last place I will look is in the debugged code I took from another source.

Serial I/O

If you've read through Chap. 19, you will very definitely get the idea that I believe that the best method for interfacing to PIC microcontroller from a PC or workstation is serially using RS-232. This standard is available in a wide variety of devices and is quite simple to wire—even though the electrical standard seems difficult to connect to. In the following experiments I want to show you how the PIC microcontroller simply can be wired to a PC serially and interfaced using either built-in hardware or the I/O pins using "bit banging" serial algorithms. Lastly, I want to show how a three-wire RS-232 connection can be created that will allow a PC to poll a serial port to determine whether or not a PIC microcontroller is connected and operating.

BasicRS: SIMULATED ASYNCHRONOUS SERIAL I/O HARDWARE WITH PIN STIMULUS FILE

The `BasicRS.asm` application uses a three times clock poll of the incoming data and uses this polling to observe when the data is coming in and then synchs off of this. This

is done by assuming that when the start bit is discovered, that by waiting one cycle (which is three times the data rate), polling now will take place in the middle of the bit (33 to 67 percent from the start of the bit).

This method works very well and allows applications to run in the foreground and treat the data coming in and out like a USART that is built into the PIC microcontroller.

The BasicRS.asm source code can be found in the code\BasicRS folder:

```

title "BasicRS - Simple 3-Wire RS-232 Application"
;
; This Application Implements a simple "3 Wire" RS-232 Link
; to another Computer. An Interrupt Based Bit Banging
; Routine is used instead of a USART.
;
;
; Hardware Notes:
; PIC16F84 Running at 4 MHz
; _MCLR is Pulled Up
; RC7 - Serial Receive Pin
; RC6 - Serial Transmit Pin
;
; Myke Predko
; 99.12.31
;
LIST R=DEC
INCLUDE "p16f84.inc"

; Registers
CBLOCK 0x020
Dlay:2
_w, _status
TXCount, TXByte
RXCount, RXByte
ENDC

#define TX PORTB, 4
#define RX PORTB, 3

PAGE
__CONFIG __CP_OFF & __WDT_OFF & __XT_OSC & __PWRTE_ON

; Mainline of TrueRS
org 0

nop

variable TMRDlay, TMRReset, PreScaler, PreScalerDlay, TMRActual,
RTCAActual, TMRError

```

```
TMRDlay = 4000000 / (1200 * 3 * 4)
```

```
TMRDlay = TMRDlay - 8
```

```
PreScalerDlay = 0
```

```
PreScaler = 2
```

```
TMRReset = TMRDlay / PreScaler
```

```
TMRActual = TMRReset * PreScaler
```

```
TMRError = ((TMRDlay - TMRActual) * 100) / TMRDlay
```

```
RTCActual = TMRActual + 9
```

```
RTCActual = ((RTCActual * 100000) / (4000000 / 4)) * 10
```

```
    goto    Mainline            ; Jump Past the Interrupt Handler

org        4
Int:                               ; PIC microcontroller Interrupt Handler

    movwf   _w                  ; Save the Context Registers
    movf    STATUS, w
    movwf   _status

    bcf     INTCON, T0IF        ; Reset the Timer Overflow Interrupt

    movlw   256 - TMRReset      ; Reset the Timer
    movwf   TMR0

    movf    TXCount, f          ; If Transmitting, Can't Receive
    btfsc   STATUS, Z
    goto    _DoRX

    movlw   0x004                ; Interrupt Transmit Increment Value
    addwf   TXCount, f
    btfss   STATUS, DC          ; Send the Next Byte?
    goto    _TXSendDlayCheck

    bsf     TXCount, 2          ; Want to Increment 3x not Four for each
byte

    bsf     STATUS, C
    rrf     TXByte, f

    movf    PORTB, w            ; Send Next Bit
    andlw   0xFF ^ (1 << 4)
    btfsc   STATUS, C
    iorlw   1 << 4
    movwf   PORTB              ; Cycle 12 is the Bit Send

    goto    _IntEnd
```

```

__TXSendDlayCheck:           ; Don't Send Bit, Check for Start Bit

    btfss TXCount, 0 ; Bit Zero Set (Byte to Send)?
    goto  __TXNothingtoCheck

    goto  $ + 1          ; Send Bit at Start of Cycle 12

    movlw 0x004           ; Setup the Timer to Increment 3x
    movwf TXCount

    bcf   TX               ; Output the Start Bit

    goto  __IntEnd

__TXNothingtoCheck:         ; Nothing Being Sent?

    movf  TXCount, w
    xorlw 0x004           ; Zero (Originally) TXCount?
    btfsc STATUS, Z
    clrf  TXCount

    movf  TXCount, w ; Sent 10 Bits?
    xorlw 0x09C
    btfsc STATUS, Z
    clrf  TXCount          ; Yes, Clear "TXCount" for Next Byte

    goto  __IntEnd

__DoRX:
; ##### - Put in Receive Interrupt Code
;
; RXCount Bit 1 Bit 0
;           0      0

        - Waiting for Character to Come In
;           0      1 - Receiving Character
;           1      0 - Have Received Valid Character
;           1      1 - Error Receiving, Clear Buffer/RXCount
;

    movlw 0x004           ; Check for Bit?
    addwf RXCount, f
    btfss STATUS, DC
    goto  __RXNo          ; Nothing to Check for (Yet)

    movf  RXCount, w      ; At the End?

```

```
xorlw 0x091
btfsc STATUS, Z
goto _RXNo

bcf STATUS, C ; Read the Current Bit
btfsc RX
bsf STATUS, C
rrf RXByte, f

bsf RXCount, 2 ; Start Counting from 4

goto _IntEnd ; Finished Receiving Byte

_RXAtEnd: ; Check Last Bit
btfss RX
goto _RXOverrun ; Not Valid - Error

movlw 2 ; Valid - Save Value
movwf RXCount

goto _IntEnd

_RXNo: ; No Bit to Receive

movf RXCount, w
xorlw 0x09D ; Read with Stop Bit?
btfsc STATUS, Z
goto _RXAtEnd

btfsc RXCount, 0 ; Something Running?
goto _IntEnd ; - Yes, Skip Over

btfss RXCount, 3 ; Checking Start Bits?
goto _RXStart

btfsc PORTB, 3
bcf RXCount, 3 ; Nothing - Keep Waiting

bsf RXCount, 0 ; Mark it has Started

btfss RXCount, 1 ; Something Already Saved?
goto _IntEnd

_RXOverrun: ; Error - Mark the Overrun

movlw 0x003
movwf RXCount

goto _IntEnd
```



```
call    NRZSend

goto    Loop

NRZSend:                                ; Send the Value in WREG Serially
movf    TXCount, f                      ; Wait for Previous Data Sent
btfss   STATUS, Z
goto    $ - 2                          ; Counter = 0 when Sent
movwf   TXByte
bsf     TXCount, 0                      ; Indicate there is Data to Send
return

NRZReceive:                            ; Wait for a New Character to be Received
btfss   RXCount, 1                      ; Bit 0 Set when Data Received
goto    $ - 1
movf    RXByte, w                      ; Get the Received Byte
clrf    RXCount
bcf     STATUS, C                      ; Carry Reset - No Error
return

Delay:

clrf    Dlay
clrf    Dlay + 1
decfsz  Dlay, f
goto    $ - 1
decfsz  Dlay + 1, f
goto    $ - 3

return

end
```

Note that in the code I do not receive anything while I am transmitting. This is because when data is being sent, the DS275 is also passing the sent signal back through the RX lines. This results in the code actually “seeing” its own data coming back as input data and, in the application, converting it and sending it back out. Sending the data back out causes it to be received again, and the application begins to operate as if it were in some kind of feed-back loop, with no new data getting sent and the current byte never ending. Normally, when I write this algorithm, I allow simultaneous (known as *full duplex*) sending and receiving. This is not possible in this application because of operation of the DS275.

The circuit that I used for this application is shown in Fig. 20.32. The breadboard wiring diagram is shown in Fig. 20.33, along with the bill of materials in Table 20.15.

This was the last experiment that I completed, so, of course, it was the one that had the most problems. When I was writing this, I was trying to get Chap. 21 finished on

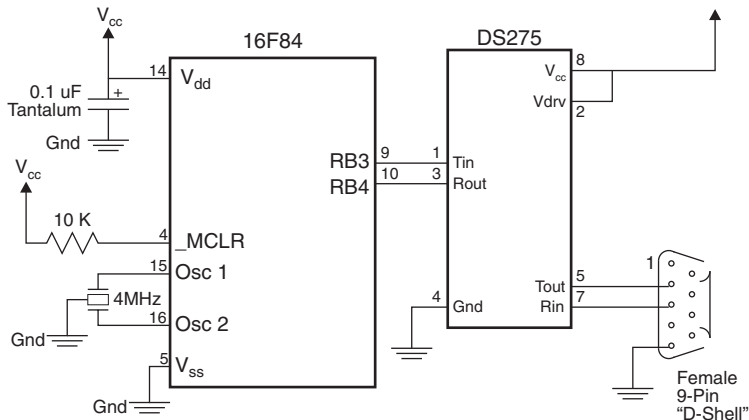


Figure 20.32 Circuit used to demonstrate simple RS-232 communications with the PIC microcontroller.

New Year's Eve 1999 so that I could save the current manuscript on CD-ROM to make sure that I would avoid any Y2K problems without losing anything, and I was starting to get ready to move. This resulted in a lot of pandemonium and three problems with getting the application running. Later in this chapter I am going to discuss debugging an application, but there are a few things that I learned that you should remember when you are in a stressful situation and you feel like you just *have* to get things done.

The first thing is to keep your perspective. The critical aspect of getting the manuscript onto CD-ROM was just that—I should not feel like my back is up against a wall because something isn't finished. What I should have done is made a CD-ROM of the current manuscript and then, when things calmed down, look at the problem again. CD-ROMs cost

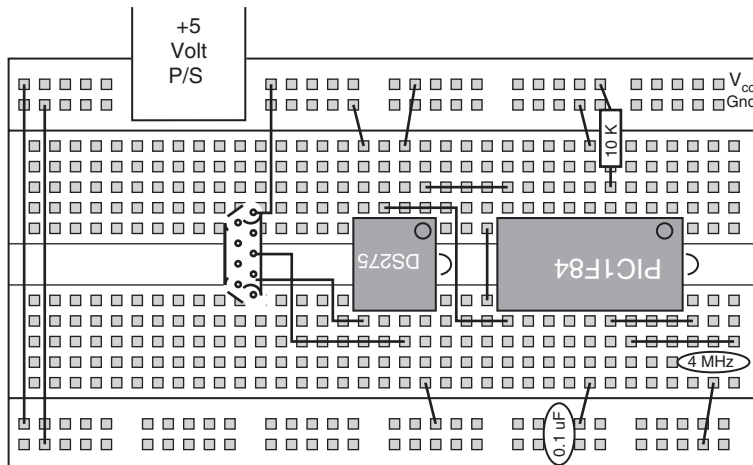


Figure 20.33 Breadboard wiring for BasicRS experiment.

TABLE 20.15 BILL OF MATERIALS FOR THE BasicRS EXPERIMENT	
PART	DESCRIPTION
PIC	PIC16F84-04/P
0.1-μF	0.1-μF tantalum
10k	10 kΩ, ¼ W
4-MHz	4 MHz with built-in capacitors
DS275	DS275 9-Pin DIP RS-232 Interface
Female 9-pin D-shell	DB-9F Female 9-pin, D-shell connector
Misc.	Breadboard, wiring, +5-V power supply

less than 50 cents; if I were able to get the application running before Y2K and written up, I only would have wasted a couple of quarters.

When I was able to realize that the important thing was to make sure that I would be ready for the deadline and not have my back against the wall, my blood pressure went down a lot, and I was able to find and fix the problems.

As I have said, I have implemented this “bit banging” algorithm on a number of different systems, but it turned out that I had not done any with the DS275. This caused me some trouble trying to figure out exactly what the problem was, and it was compounded by my not waiting a full stop bit’s width before sending data back to the PC. This was a problem in that the PC’s UART could not interpret the data coming back and simply ignored the characters. Because I was using HyperTerminal with it, the errors were not reported back to me.

I was able to find the issue with the stop bit with an oscilloscope. When I was able to see that the stop bit of the data going back to the PC was only 520 μs long (it should be at least 833 μs for 1,200 bps), I was able to go back to the code and fix the problem.

Next, I discovered that after I sent the first character, data would be sent repeatedly. This is so because of the negative voltage stealing of the DS275. The problem was that as data was being transmitted, it also was being received and resent over and over again in a digital feedback loop. Normally, as I said, when I implemented this algorithm in a PIC microcontroller in the past, it was always with a MAX232, so the feedback angle wasn’t possible. To fix this problem, I stopped polling the incoming data while a byte transmit was taking place.

Lastly, I discovered that I miswired the DS275. Instead of connecting the Vcc pin (8) to my breadboard Vcc, I accidentally connected it to ground. I found this when I was trying to figure out the other problems, and I discovered that the voltage level was very low (and not between 0 and 5 V, as I expected from my previous experiences with the DS275). This was the keystone to all the problems, and when this was fixed, everything fell into place.

The problem was calming down enough to go back and check every connection against the wiring diagram I produced for it (Fig. 20.33). Once I did this, I was able to

find the wiring problem and then worked back to the other two problems, and I had them fixed in time to save the manuscript on CD-ROM for Y2K.

3RS: INTERFACING A PIC MICROCONTROLLER USING A THREE-WIRE RS-232 INTERFACE

To finish off the experiments, I want to end with one that demonstrates how easy it is to interface a PIC microcontroller with a Microsoft Windows application. If you take a look at the code for both the PC and the PIC microcontroller, you'll probably feel like the application is quite complex, but as I go through the operation, you will see how easy it is to interface a PIC microcontroller serially to PC applications. The experiment itself uses the circuit shown in Fig. 20.34 (the bill of materials listed in Table 20.16).

The serial data connect uses the Ping application described earlier in this book, in which an echoed (by hardware) 0xFF character is changed on its receive to something different. This operation is shown in Fig. 20.35, with the output byte of 0xF0 changed to 0x90. In the actual application, I used a data byte of 0xFF which is changed by the circuit to another character. I chose 0xFF because it is not a valid ASCII symbol and is easily recognized as such by a program such as Hyperterminal.

In the PIC microcontroller application, the data byte 0xFF is waited for, and if it is received, then the next byte (which also should be 0xFF) is modified by changing the character that is echoed back to the host computer, which is a PC in this case. A LED connected to PORTA.2 (RA2) is lit if the connection is active (i.e., 0xFF characters are being received). The 3RS.asm application can be shown as the following pseudocode:

```
main()                // 3RS Pseudo-Code, Used to
{                     // Test and Establish Link to PC Host
  int i;
  char Data;
```

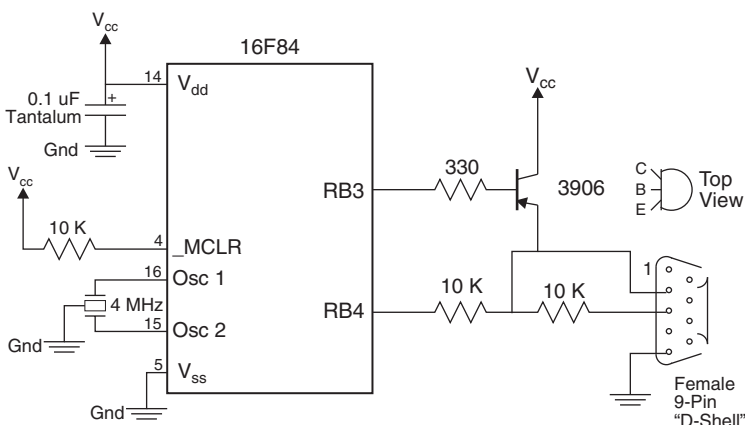


Figure 20.34 Simple three-wire RS-232 interface with voltage stealing.

TABLE 20.16 BILL OF MATERIALS FOR THREE-WIRE RS-232 INTERFACE	
PART	DESCRIPTION
PIC	PIC16F84-04/P
0.1-μF	0.1-μF tantalum
10k	10 kΩ, ¼ W
4-MHz	4 MHz with built-in capacitors
220	220 Ω, ¼ W
1 × LED	LED
330-Ω	330 Ω, ¼ W
3906	2N3906 transistor
1 × Female 9-pin D-shell	DB-9F
Misc.	Breadboard, wiring, +5-V power supply

```
LED = off;                // No Connection, LED Off

while (1 == 1) {          // Loop Forever

    for (i = 0; (i < 600msec) && (SERIN == MARK); i++);
                                // Poll Serial In Line for "Start"
    Bit
    HalfBitDlay();
    if (SERIN == MARK)          // Timeout - Nothing Received
        LED = off;            // Turn off the LED/No Connection
    else {                     // Something Received
        for (I = 0; I < 8; I++) {
```

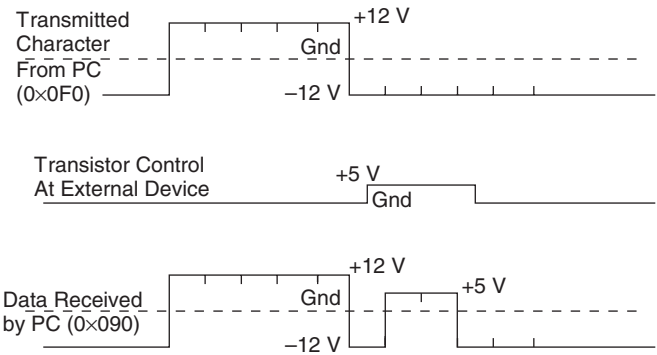


Figure 20.35 Simple test waveform sent to see if receiver is present and active.

```

    Data = (Data >> 1) + (SERIN << 7);
    BitDlay();
} // fi // Read Incoming Byte

if (Data = 0x0FF){ // 0x0FF Received
    LED = on; // Turn on LED to Indicate Data Incoming

    for (i = 0; (i < 600msec) && (SERIN = MARK); i++);
    // Poll Serial Line for Second Byte
    HalfBitDlay();
    if (SERIN == MARK) // Timeout - Nothing Received
        LED = off; // Turn off the LED
    else { // Send Back "Synch"

        for (i = 0; i < 5; i++ )
            BitDlay(); // Wait Past First Four Bits (+ Stop)
        SEROUT = SPACE; // Change the Data Going Out

        For (I = 0; I < 3; I++)
            BitDlay(); // Wait for 3 Bits

        SEROUT = MARK; // Don't Change Anything Else

        BitDlay(); // Don't Change the Stop Bit

    } // fi // End, Second 0x0FF Received
} // fi // End, First 0x0FF Received
} // fi // End, First Byte Received
} // elihw
} // end 3RS

```

The actual PIC microcontroller assembly-language code is 3RS.asm and can be found in the code\3RS folder.

```

title "3RS - Simple 3-Wire Communication Protocol"
;
; This Application Implements a simple Communication
; protocol. The Link is assumed to be down unless the
; the character 0x0FF is received. When this character
; is received, then the next character (which is also
; 0x0FF), is received, the PIC microcontroller, Changes bits 4
; through 6 from "1" to "0" and sends it back to the
; "host". The "RSReceive" and "RSTransmit" were taken
; from "SimpRS".
;
;
; Hardware Notes:
; PIC16F84 Running at 4 MHz

```

```
; _MCLR is Pulled Up
; PORTB.3 is the Transmit Output
; PORTB.4 is the RS-232 Input
; PORTA.2 is an LED indicating the Link is Up
;
; Myke Predko
; 99.12.30
;
LIST R=DEC
INCLUDE "p16f84.inc"

; Register Usage
CBLOCK 0x020          ; Start Registers at End of the Values
Byte, Count           ; Variables for RS-232
Dlay:2                ; Delay Count
OnCount, OffCount     ; Reset On/Off Counts
ENDC

#define TX PORTB, 3
#define RX PORTB, 4
#define LED PORTA, 2

PAGE
__CONFIG __CP_OFF & __WDT_OFF & __XT_OSC & __PWRTE_ON

; Mainline of SimpRS
org    0

nop

bsf    LED           ; LED is "Off"
bsf    TX            ; Start Sending a "1"
bsf    STATUS, RP0
bcf    LED           ; Enable the LED Output
bcf    TX            ; Enable TX for Output
bcf    STATUS, RP0

Loop:

clrf   Dlay          ; Put in a 1/2 Second Delay
clrf   Dlay + 1

btfsc  RX            ; Wait for a Start Bit
goto   HaveBit       ; If Line High, Have the Start Bit
goto   $ + 1
goto   $ + 1
decfsz Dlay, f       ; Loop 64K Times
```

```

    goto $ - 5
    decfsz Dlay + 1, f
    goto $ - 7

    bsf    LED                ; No Start Bit, Start All Over

    goto   Loop

HaveBit:

    call   HalfBitDlay        ; Wait 1/2 a Bit

    btfss  RX                 ; Make Sure Bit is Still Low
    goto   Loop

    movlw  8
    movwf  Count

SynchRXLoop:                ; Loop Here to Read in the Byte

    call   BitDlay            ; Wait a Full Byte

    bcf    STATUS, C          ; Set Carry Accordingly
    btfss  RX                 ; Bit High or Low?
    bsf    STATUS, C

    rrf    Byte, f            ; Shift in the Byte

    goto   $ + 1              ; Make 11 Cycles in Loop
    goto   $ + 1

    decfsz Count, f
    goto   SynchRXLoop

    call   BitDlay            ; Make Sure there is a "Stop" Bit

    bsf    LED                ; Turn Off LED Just in Case

    btfsc  RX
    goto   Loop               ; Not High, Then No Byte - Start Over

    movf   Byte, w
    xorlw  0xFF               ; Is the Value Received 0xFF?
    btfss  STATUS, Z          ; No, LED off, Keep Waiting
    goto   Loop

    bcf    LED                ; Indicate the Link is Up
    clrf   Dlay               ; Put in a 1/2 Second Delay
    clrf   Dlay + 1

```

```
    btfsc  RX                      ; Wait for a Start Bit
    goto  Have2Bit                ; If Line High, Have the Start Bit
    goto  $ + 1
    goto  $ + 1
    decfsz Dlay, f                ; Loop 64K Times
    goto  $ - 5
    decfsz Dlay + 1, f
    goto  $ - 7

    bsf    LED                    ; No Start Bit, Start All Over

    goto  Loop

Have2Bit:

    call   HalfBitDlay            ; Wait 1/2 a Bit

    bsf    LED
    btfss  RX                      ; Make Sure Bit is Still Low
    goto  $ - 1

    bcf    LED

    movlw  9
    movwf  Count
    movlw  5                        ; Start at Bit 4 Coming Back
    movwf  OnCount
    movlw  3
    movwf  OffCount

RXLoop:                                ; Loop Here to Read in the Byte

    call   BitDlay                ; Wait a Full Byte

    decfsz OnCount, f
    goto  $ + 6                    ; No - Decrement and Continue
    decfsz OffCount, f
    goto  $ + 7                    ; Keep Bit Off?
    bsf    TX                      ; Put it On
    bsf    OnCount, 7              ; Yes, Make Sure Never Goes Off Again
    goto  $ + 7
    nop                                ; Not at Turn Off Yet
    goto  $ + 1
    goto  $ + 4

    bcf    TX                      ; Output a Low Bit
    bsf    OnCount, 0              ; Make Sure it Happens Again
    nop
```

```

    decfsz Count, f
    goto RXLoop

    goto Loop

BitDlay:                                ; Delay 833 - 15 Cycles (including
; Call/Return)
    movlw 204
    addlw 0xFF                          ; Take 1 Away from the Loop
    btfss STATUS, Z
    goto $ - 2

    goto $ + 1

    return

HalfBitDlay:                            ; Delay (833 - 15) / 2 Cycles

    movlw 100
    addlw 0xFF                          ; Take 1 Away from the Loop
    btfss STATUS, Z
    goto $ - 2

    return

end

```

To understand this code, I recommend going back over the pseudocode and trying to follow the data coming in. In the pseudocode I have tried to avoid using any definite values simply because they can be confusing (especially with the inverted serial input and positive serial output).

For this experiment, I have used Microsoft's Visual Basic 6.0 as the application development environment for the host code. I have included the Visual Basic source (3RS Visual Basic.frm) and project (3RS Visual Basic.vbp), and the required files can be found in the code\3RS\vb folder. The BASIC source code is

```

' 3RS Visual Basic
'
' This Application Polls a Serial Port for an Active
' PIC microcontroller Connected to it.
'
' When "Start" is Clicked, 0xFF, followed by 0xFF is
' Sent out of the Serial Port. The two Values are
' checked for a Valid Return (which means the Second
' 0xFF is Scrambled by the PIC microcontroller).
'

```

```
' If nothing is returned, then the "No Connect"
' (Label(0)) is Active, if 0xFF and no change, then
' (Label(1)) is Active, if 0xFF is returned and
' Mangled, then (Label(2)) is Active.
'
' 99.12.30
' Myke Predko
Dim oldCombol As Integer      ' Flag of Old Serial Port


---


Private Sub Combol_Click()
' Combol Box is Clicked on. Can Only Change if "Start" is in
' Command2

If (Command1.Caption = "Start") Then
    oldCombol = Combol.ListIndex + 1
End If

End Sub


---


Private Sub Command1_Click()
' "Quit" Button

If (Command2.Caption = "Stop") Then
    MSComm1.PortOpen = False
End If

End

End Sub


---


Private Sub Command2_Click()
' "Start"/"Stop" Button is Active

If (Command2.Caption = "Start") Then
    MSComm1.CommPort = Combol.ListIndex + 1
    On Error GoTo invalidCommPort
    MSComm1.PortOpen = True
    Timer1.Enabled = True
    Command2.Caption = "Stop"
Else
    ' Stop the Action
    MSComm1.PortOpen = False
    Timer1.Enabled = False
    Command2.Caption = "Start"
End If

Exit Sub


---


invalidCommPort:                ' Notify of Invalid Selection
```



```
MsgBox "Selected CommPort Could NOT be Opened. Try Another",
vbExclamation, "Invalid CommPort"
```

```
Exit Sub
```

```
End Sub
```

```
Private Sub Timer1_Timer()
```

```
 ' 1/2 Second "Ping" Timer
```

```
MSComm1.Output = Chr$(&HFF)
```

```
If (MSComm1.InBufferCount <> 0) Then
```

```
temp$ = MSComm1.Input
```

```
If (temp$ = Chr$(&HFF)) Then
```

```
  If (Label1(2).BackColor <> &HFFFF&) Then
```

```
    Label1(1).BackColor = &HFFFF&
```

```
    Label1(0).BackColor = &HFFFFFF
```

```
    Label1(2).BackColor = &HFFFFFF
```

```
  End If
```

```
  Else ' Data Scrambled
```

```
    Label1(2).BackColor = &HFFFF&
```

```
    Label1(0).BackColor = &HFFFFFF
```

```
    Label1(1).BackColor = &HFFFFFF
```

```
  End If
```

```
Else ' Nothing Received
```

```
  Label1(0).BackColor = &HFFFF&
```

```
  Label1(1).BackColor = &HFFFFFF
```

```
  Label1(2).BackColor = &HFFFFFF
```

```
End If
```

```
End Sub
```

```
Private Sub Form_Load()
```

```
  Combo1.AddItem "COM1", 0
```

```
  Combo1.AddItem "COM2"
```

```
  Combo1.AddItem "COM3"
```

```
  Combo1.ListIndex = 0
```

```
  Label1(0).BackColor = &HFFFF&
```

```
  Label1(1).BackColor = &HFFFFFF
```

```
  Label1(2).BackColor = &HFFFFFF
```

```
End Sub
```

The Visual Basic application is available in both source code format and as a package that can be installed on your PC without requiring compilation. Execute `setup.exe` in the `code\3RS\vb\Package` folder to load the 3RS Visual Basic executable application code along with all the necessary device drivers for the RS-232 interface.

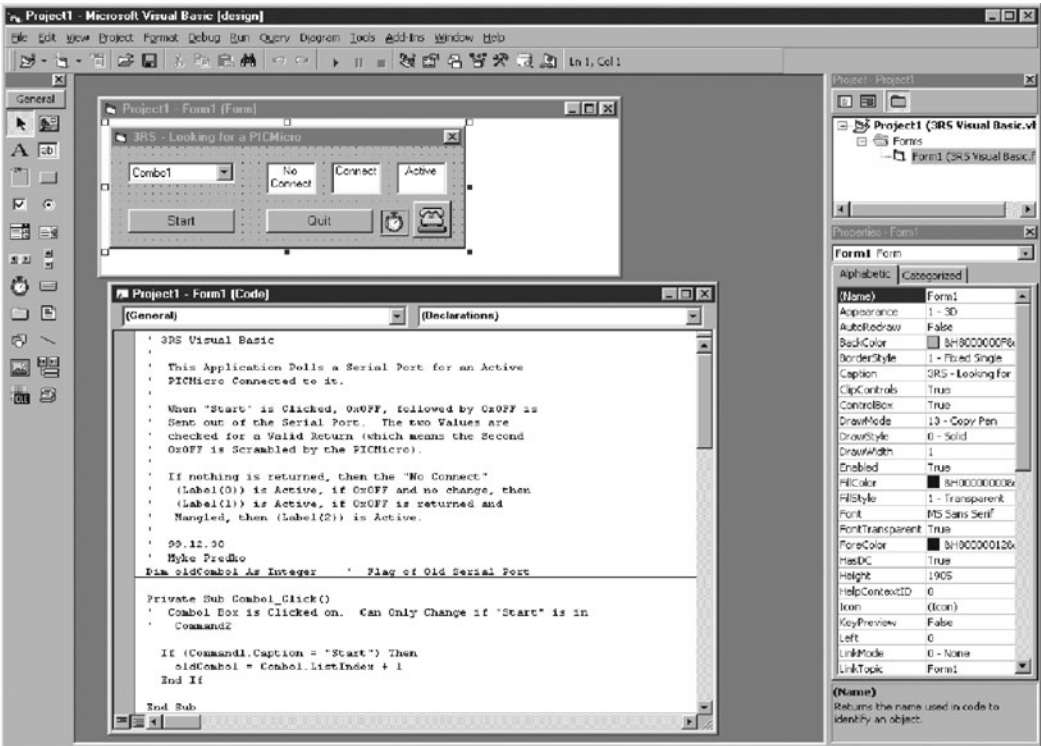


Figure 20.36 The Visual Basic desktop for the 3RS experiment showing the controls used to create the PC application.

I always feel like the source code in Visual Basic (and any Visual Development environment) is only half the story. In Fig. 20.36, the “Dialog Box” or Form that I created for this application is shown and has a “Combo” box to select the serial (COM) port. These controls are used with a Start/Stop, Quit, and Button controls and three “Text” boxes that indicate the current state of the link and are all visible. What you can’t see in the active Form (shown in Fig. 20.37) is the “Timer” and “MSComm” serial Port interface that I have added to control the application.



Figure 20.37 The Visual Basic application active and the PC connected to the working 3RS PIC microcontroller application.

The timer even handler (known as `Timer1` control in the 3RS Visual Basic application) is probably the most important control in the Visual Basic application because it is responsible for periodically sending an `0xFF` out on the specified serial port. `Timer1_Timer` executes every time `Timer1` overflows, which happens every 500 ms, and sends an `0xFF` character out on the specified COM port. If a character is received, then it is checked as either `0xFF` or something else (in which case the “Active” box is made yellow). If nothing is received, then the “No Connect” box is yellow, or if `0xFF` is received back, then “Connect” is returned.

This application violates one of the basic tenets that I try to establish for connecting the PIC microcontroller to host computers via RS-232, and that is that a terminal emulator (such as HyperTerminal) cannot be used for debugging the PIC microcontroller application itself. Despite this, I did not have any trouble testing and debugging the interface.

If you have trouble with the interface, then I would suggest that the ping character should be changed from `0xFF` to something easy to add such as `0x61` (ASCII `a`) to see how the application works. If this is done to help debug the application, make sure that characters are sent within a half second, or the PIC microcontroller application will timeout, and the second character will not be changed (essentially the process is happening all over).

This page intentionally left blank

PROJECTS

When I wrote the first edition of this book, I wanted to provide a good range of sample applications for readers to work through and use as the bases of their own applications. With this goal in mind, I presented the applications as introductions to the PIC[®] microcontroller and different interfacing methods. This actually was quite a big miscalculation on my part. It turned out that most readers wanted to build the applications just as they were. The limited circuit information caused problems for a number of readers who did not have experience in designing their own PIC microcontroller applications and felt like I did not provide enough information to help them wire the applications. To remedy this, I have provided full schematics for each project along with pointers in the downloaded information as to where the code can be found.

I am also pleased to offer twice as many sample applications as were present in the original book. To help you sort through them, I have grouped them according to the PIC microcontroller family to which they belong, and each project has its own HTML page with links to the source code and other pertinent information. In previous books, people have had problems deciphering schematics, so I have included postscript versions of the schematics for download as well.

The projects themselves are generally quite simple and are designed to be built and debugged in the space of a weekend. I have designed the different projects to use a variety of control, display, and power-supply methods. As I describe them, I will point out their pros and cons, as well as discuss their suitability for other applications.

Low-End Devices

As I've gained insight and expertise into working with the different PIC microcontroller devices, I've found that there are many people using the PIC microcontroller architecture families in applications that are inappropriate for their capabilities. Nowhere is this more obvious than in regard to how low-end devices are used. Most of the published

applications that I see in magazines and journals use low-end devices in applications for which I consider them poorly suited. As I indicated earlier in this book, I consider low-end devices best suited for applications that have the following characteristics:

- 1 No interrupts
- 2 No complex dialog-based user input-output (I/O)
- 3 No complex interfaces
- 4 Minimal subroutines

This does not mean that sophisticated applications cannot be implemented with the low-end PIC microcontrollers, just that the code cannot be very long or require interfaces that are best suited for interrupts or the peripherals of mid-range devices. I have targeted the example low-end applications in this chapter to use the 12C5xx and 16C505 PIC microcontrollers, which are very low cost (under a dollar each in reasonably low quantities) and do not require external clocks or reset.

TrainCtl: MODEL TRAIN TRAFFIC LIGHT CONTROL USING A HALL-EFFECT SENSOR

As my kids have gotten older, I have been reintroduced to many of the hobbies and toys that I had when I was young. Many of these activities have changed in ways that would have been unimaginable for me as a kid 30 years ago; for example, Hot Wheels cars can be bought with *computer chips* inside them. If you are buying stuffed toys for babies and toddlers, you will discover that just about all of them have electronic chips inside them to provide light and sound. Finally, remote-control aircraft were something that cost \$500 or more and required a significant level of skill when I was young; today, remote-control aircraft can be purchased for \$30 or less. Despite this, I was disappointed to find that model trains are virtually unchanged from when I was a kid, with many of the products available today being identical to what I had 30 years ago.

This statement is not quite true; there is the DCC protocol, which allows digital commands to be broadcast on the rails to engines, rolling stock, switches, and accessories. A starter-set DCC is very expensive and can be very difficult (*fiddly* is the word I hear most often) to set up and use. What I would like to come up with is a computer-controlled system that is somewhere between the two. The flexibility of a computer system should be able to be added to the train system for relatively low cost and relatively simply.

The project presented here is one of the first of my experiments toward this type of system. It uses the 18-V ac auxiliary power supply built into most electric train systems to power the circuitry as well as the traffic lights that are controlled by it. The prototype circuit is shown in Fig. 21.1.

The circuit shown in Fig. 21.2 draws power from the model train ac power and controls the operation of the three traffic lights through the use of TRIACS. I have to caution you very strongly that while the techniques for determining component values *could* be used for developing applications that control household mains wiring, the component values cannot be used with 110–120- or 220-V ac household power currents. You should not attempt to use the information contained in this project if you are the

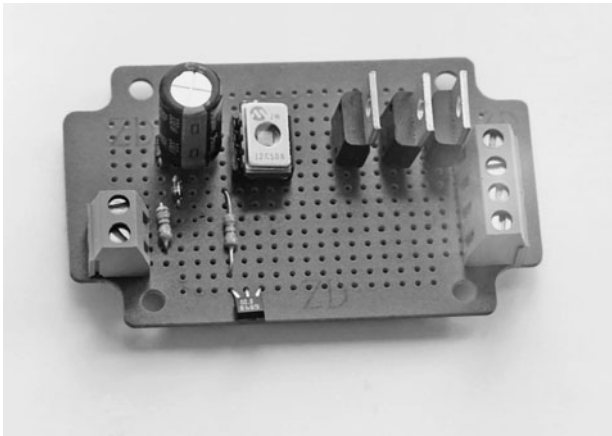


Figure 21.1 The model train three-traffic-light control circuit.

least bit unsure of what you are doing or you are not fully comfortable with working with ac voltages.

The circuit shown in this application is designed only for the 18 V ac that is available from model train transformers for driving accessories. Using 18 V results in very little danger of electrical shock—but short circuits can result in high currents that can burn you or cause fire if you are not careful with the circuit. The bill of materials for the application is listed in Table 21.1.

The circuit has a 5.1-V, 50-mA power supply that also provides half-wave-rectified power from the 18-V ac source. This sounds like a pretty fancy specification, but it actually uses the simply regulator circuit shown in Fig. 21.3. The Zener diode limits the voltage across it to 5.1 V, and this voltage is used for powering the PIC microcontroller that controls the circuit. The silicon diode allows current to pass in only one direction. These two diodes have a combined voltage drop of 5.8 V. Since I want to

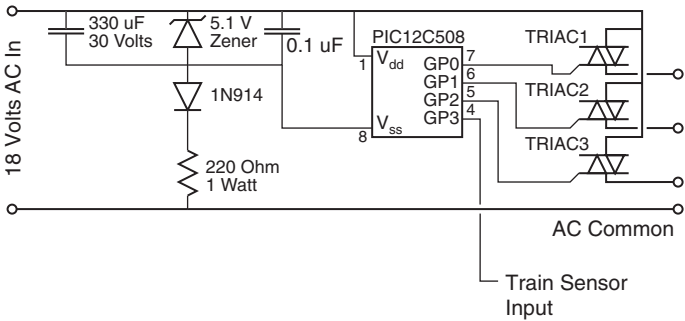


Figure 21.2 The model train light-controller circuit is actually quite simple.

TABLE 21.1 TRAIN CONTROLLER BILL OF MATERIALS	
REFERENCE DESIGNATOR	DESCRIPTION
PIC12C508	PIC12C508–JW
5.1 Zener diode	5.1 V, 1/2 W
1N914 silicon diode	1N914 silicon diode
Gate current TRIAC1–TRIAC3	Forward current 6 A, 25 mA
220	220 Ω, 1 W resistor
330-μF	330 μF, 35 V electrolytic capacitor
0.1-μF	0.1 μF capacitor, any type
Train sensor	Hall-effect sensor (see text)
Miscellaneous	Prototyping PCB, wiring, screw terminals for electrical connections

provide 50 mA to the PIC microcontroller, I use this information to calculate the resistor value needed.

Using Kirchoff’s and Ohm’s laws,

$$\begin{aligned}V_{ac} &= V_{\text{Zener}} + V_{\text{diode}} + V_r \\18\text{ V} &= 5.1\text{ V} + 0.7\text{ V} + V_r \\V_r &= 18 - 5.8\text{ V} \\&= 12.2\text{ V}@50\text{mA} \\R &= V/I \\&= 12.2\text{ V}/50\text{ mA} \\&= 244\text{ }\Omega\end{aligned}$$

I used a 220-Ω resistor in my circuit because it is a standard value.

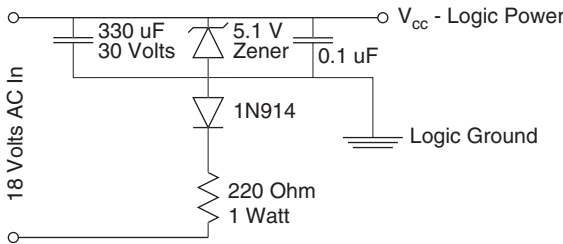


Figure 21.3 Model train 18-V ac to 5-V dc power supply.

With 50 mA of current going through the 220- Ω resistor, the maximum power dissipated is

$$\begin{aligned} P &= V \times I \\ &= 12.2 \text{ V} \times 50 \text{ mA} \\ &= 0.61 \text{ W} \end{aligned}$$

This is actually a reasonable amount of power, and this is why I specified the 1-W resistor. The maximum power going through the diode is

$$\begin{aligned} P &= V \times I \\ &= 0.7 \text{ V} \times 50 \text{ mA} \\ &= 0.035 \text{ W} \end{aligned}$$

which is actually quite low—a small signal diode such as the 1N914 can be used safely.

The large (330-mF) capacitor is used to maintain an even voltage to the PIC microcontroller even when the diode isn't conducting. This capacitor is very effective because I measured a 10-mV ripple on the PIC microcontroller's Vdd relative to Vss (or what I call **logic Gnd**). The 0.1- μ F capacitor is used for decoupling the PIC microcontroller and should be as close to the Vdd and Vss pins as possible.

TRIACs are interesting devices and come under the heading of *thyristors*, which can be used to switch ac signals on and off. TRIACs do not rectify the ac voltage because they consist of two silicon-controlled rectifiers (SCRs), which allow the ac current to pass without any clipping. A typical circuit for TRIACs is shown in Fig. 21.4.

TRIACs do not allow ac current to pass unless their gates are biased relative to the two ac contacts. To do this, I pull the gates to **logic Gnd** by the PIC microcontroller. As noted in the bill of materials (Table 21.1), the current required to close the TRIAC is 25 mA, which can be sunk by most PIC microcontrollers.

The load in this project are colored “grain of wheat” bulbs that are designed for model train applications and can handle the 18-V ac accessory drive provided by the

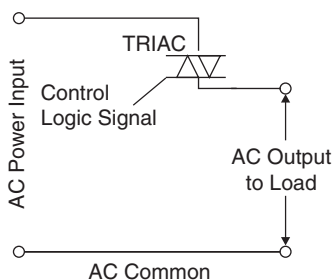


Figure 21.4 The TRIAC acts like a switch, controlling the flow of ac power to a load.


```

while (sensor == no train); // Wait for Train

green = off;           // Tell Traffic to Stop
amber = on;

delay(1 second);

red = on;

DelayCount = 10 seconds;
while (DelayCount != 0) {
    DelayCount = DelayCount - 1;
    if (sensor == train) // Reset if Train still there
        DelayCount = 10 seconds;
} // end while
} // End main - train ctrl

```

My prototype circuit was built on a simple protoboard, as shown in Fig. 21.1. For the 18-V ac lines, I used 20-gauge solid-core signal wires. To facilitate easy insertion/removal into a circuit, I used screw terminals for power and the light signals.

As a final word of warning and caution, this type of construction is adequate for the model train 18-V ac accessory drive; it is not acceptable, nor is it safe, for 110- or 220-V home mains wiring—in these cases, the equations shown in this section can be used, but with the appropriate values used for the voltage levels.

SLI: SERIAL LCD INTERFACE

One of the most useful tools you can build for yourself is a simple LCD interface for use with your projects (Fig. 21.6). In the first edition of this book, I presented a serial LCD interface application that demonstrated how an application written for a mid-range



Figure 21.6 The serial LCD interface in operation.

PIC microcontroller could be ported to a low-end device. It had the interesting capability that it could sense the timing of the incoming signals (and their polarity), which made it useful for some applications where a carriage return character (which was used for timing) was first sent. It made it less useful in applications where the interface was connected to another device, and the data coming out was arbitrary and probably not a carriage return. In the latter cases, the original serial LCD interface could not be used.

The serial LCD interface application presented here is a completely new design and does not use much of the technology from the one presented in the first edition. This application was written exclusively for a low-end PIC microcontroller (the PIC16F505 specifically) and was designed to show off the following characteristics of the part and application design:

- 1** The built-in 4-MHz oscillator of the PIC16F505
- 2** The built-in reset of the PIC16F505
- 3** Using the `_MCLR` pin of the PIC16F505 for serial input
- 4** Combining PORTC and PORTB of the PIC16F505 as an 8-bit I/O bus
- 5** Combining switch inputs with bidirectional pins connecting the PIC microcontroller to the LCD
- 6** Providing a timed interrupt-like interface to read incoming serial data

I chose the PIC16C505 because when the internal 4-MHz oscillator and internal reset are used, the PIC microcontroller provides 11 I/O pins and 1 input pin. The 11 I/O pins are perfect for bidirectional I/O on an LCD, and the single input can be used for the serial input. To set the operating mode (speed and signal polarity), I provided two switches consisting of two pins that can be shorted with a jumper. The circuit that I came up with is shown in Fig. 21.7. The bill of materials for the serial LCD interface is listed in Table 21.2.

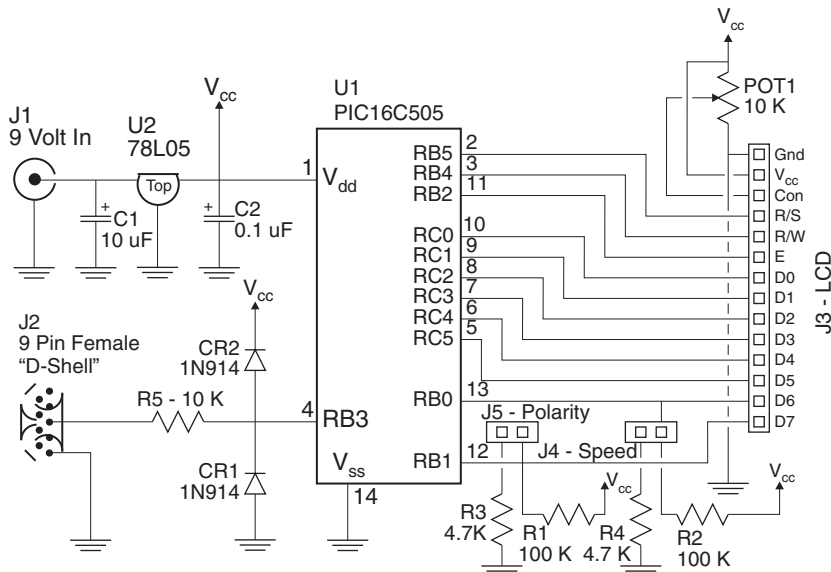


Figure 21.7 The serial LCD interface is based on the PIC16F505, which has an internal oscillator.

TABLE 21.2 SLI BILL OF MATERIALS

REFERENCE DESIGNATOR	DESCRIPTION
U1	PIC16F505-IP
U2	78L05
C1	10 μ F, 16 V electrolytic
C2	0.1 μ F, 16 V tantalum
R1, R2	100 k Ω , $\frac{1}{4}$ W
R3, R4	4.7 k Ω , $\frac{1}{4}$ W
R5	10 k Ω , $\frac{1}{4}$ W
Pot1	10 k Ω , single-turn potentiometer
CR1, CR2	1N914 silicon diode
J1	9-V battery connector
J2	9-pin female D-shell connector with solder tail connections
J3	14 $\times$ 1 header for LCD/LCD is 16 $\times$ 2 Hitachi 44780-based display
J4, J5	2 $\times$ 1 pin header
Misc.	Prototype board, wire

When I built my prototype, I wired it using point-to-point wiring on a simple phenolic prototyping card with the 9-pin D-shell connector pressed onto the card and the mounting holes tied to the card for stability. I mounted the LCD's 14 $\times$ 1 connector in such a way that it covers most of the card, including the two pin headers used to select the speed and polarity of the incoming data.

For the serial LCD interface circuit presented here, note that I have included a simple 5-V power supply consisting of a 10- μ F capacitor and a 78L05. This power supply provides 100 mA for the circuit, and I wanted to use a 9-V radio alkaline battery for powering the serial LCD interface.

To run the PIC16F505 with the internal oscillator and internal reset, I used the `__CONFIG` statement in the source code:

```
__CONFIG _CP_OFF & _WDT_OFF & _IntRC_OSC_RB4EN & _MCLRE_OFF
```

This statement allows RB4 and RB5 to be used as I/O pins instead of OSC2 and OSC1, respectively, and allows RB3 to be used as a digital input (and not `_MCLR/Vpp`). To allow RB5 (which is normally T0CKI—the TMR0 input pin) to be used as an I/O, I had to reset the T0CS bit of the OPTION register, which was accomplished by the two instructions:

```
movlw    0x0FF ^ (1 << T0CS)
option
```

The serial input includes two clamping diodes to ensure that the input voltage on the PIC microcontroller does not exceed any of the specified voltages. When I first implemented this circuit, I only had one clamping diode (CR1 in Fig. 21.7) on RB3. When RS-232 signals were sent to the serial LCD interface with only one clamping diode, I found that the input to the PIC microcontroller ranged from 0.2 to 10 V and placed the PIC microcontroller periodically in program mode.

The second clamping diode (CR2) was added, and the voltage swing on RB3 was reduced from 12 to 4.7 V, and I didn't have any issues with the PIC microcontroller going into program mode.

The source code for this application can be found in the `code\SLI` folder. The final version that is used with the application is `SLI04.asm`.

The two switches caused some problems with the LCD. When I first started working with them, I was surprised to find that the LCD, which did not have any of its I/O pins driven by the PIC microcontroller, was holding down the lines. To allow the PIC microcontroller to read the switch values (which are either pulled up by 100-k Ω resistors or pulled down 4.7-k Ω resistors), the E bit (pin 6) of the LCD has to be pulled down low. To do this, I used the three statements:

```
clrf    PORTB                ; See If PORTB Initialized Helps
movlw   0x0FF ^ ((1 << 5) + (1 << 4) + (1 << 2))
tris    PORTB                ; Drive the LCD Control Bits
```

before the pin read to force the E, R/S, and R/W lines of the LCD low and then to perform the switch read. The need for holding E down in order to read the switches was a surprise and not something that I had expected.

When writing to the LCD, the 8 I/O bits are shared between PORTB and PORTC, with the two most significant bits of the LCD's 8 I/O bits as PORTB bits 0 and 1. To shift up the bits, I used the code

```
movf    char, w              ; Get the Character to Send to the LCD
movwf   Dlay
movwf   PORTC                ; Store the Least Significant bits in PORTC
rlf     Temp, w              ; Convert the Most Significant bits to PORTB's
rlf     Temp, f              ; Least Significant Bits
rlf     Temp, w
rlf     Temp, w
andlw   0x003
movwf   PORTB                ; Output the Data to PORTB
```

This code should be explained a bit because it will not be expected, especially with all the `rlf` statements that seem to continually load the `w` register with the same value.

If you look in the appendices, you will see that the two instructions, namely,

```
rlf     Register, w
rlf     Register, f
```

will do a rotate on the contents of `Register`. The first instruction will load the carry flag with the contents of the most significant bit. The second instruction will shift up the contents of `Register` while passing the most significant bit (which was loaded into the carry flag in the previous instruction) in the least significant bit of `Register`.

The next two instructions (both of which are `rlf Register, w`) will first load the contents of the most significant bit into carry and then load the rotated register into the `w` register. These four instructions (along with the following `andlw 0x003` instruction) moves the most significant 2 bits of the byte to write to the LCD into the 2 least significant bits of `PORTB`.

Another way of doing this could have been as follows:

```
movf    char, w      ; Get the Character to Send to the LCD
movwf   Dlay
movwf   PORTC        ; Store the Least Significant bits in PORTC
swapf   Dlay, f      ; Move the Most Significant Nybble down
rrf     Dlay, f      ; Shift Bits down by 2
rrf     Dlay, w
andlw   0x003
movwf   PORTB        ; Output the Data to PORTB
```

With the circuit and interface code working, I was then able to focus on doing the serial input read and then output to the LCD. As I indicated earlier, I wanted to create an interface that would simulate the operation of a `TMR0` overflow interrupt that polls the serial line three times for each bit.

As I envisioned the serial LCD interface presented here, I wanted to have it work for 9,600 and 1,200 bps. A 9,600 bps data rate has a bit period of 104.167 μs (which I usually round off to 104 μs), and the 1,200 bps data rate has a bit period of 833.333 μs (which I round off to 833 μs).

Using these timings, I had to come up with code that would poll either once every 35 μs (or 35 instruction cycles) for 9,600 bps or 278 μs for 1,200 bps. Polling every 35 μs would give me an effective data rate of 105 μs for 9,600 bps for an error of 0.8 percent. The 278- μs polling rate gave me an effective data rate of 1,199.041 bps with an error of 0.08 percent to an ideal of 1,200 bps. In either case, this error rate is low enough as to not result in incorrect data being read in.

To actually poll the data, I created a macro that would check for data already coming in, and if it were, or if a start bit was detected, it would call a serial processing subroutine. This macro only required three cycles (allowing up to 32 cycles to execute outside of it) to check the incoming data:

```
SerCheck Macro          ; Check Serial Data
    btfss  SerActive    ; Reading to be Done?
    btfss  SerIn
    call   RSSerialPos  ; Yes - Check the RS-232 Line Input
endm
```

In the code, the `SerActive` flag is set to indicate that data is coming in and to jump to the serial read routine. If this flag is not set, then if the incoming data line is a 0 (start bit), the serial read routine is called to start reading the incoming data.

The serial read routine can be written in high-level pseudocode as follows:

```
SerialRead()                                // Read Incoming Serial Data
{
    Temp = inp(PORTB);                       // Get the Contents of the Serial Data
    // for Checking
    if (SerActive == 0)                      // Is this a Start Bit?
        if ((Temp & (1 << SerInBit)) == 0) {

            Delay( 1/3 Bit );                // Delay 1/3 Bit

            if ((inp(PORTB) & (1 << SerInBit)) == 0) {

                SerActive = 1;               // Not a Glitch, Read in Byte

                BitCount = 9;                // Going to Read through 9 Bits

                Delay( 2/3 Bit );            // Delay Full Bit

            }
        } else ;                            // Not Low – Glitch?
    else {                                   // Else, Reading in Data

        BitCount = BitCount - 1;            // Decrement the Bit Count

        if (BitCount != 0)                  // Read Bit?
            RXData = (RXData >> 1) + ((Temp & (1 << SerInBit)) != 0);
        else {                              // Last Bit – Is there a Stop Bit?

            SerActive = 0;                  // In any Case, not Reading Bits

            if ((inp(PORTB) & (1 << SerInBit)) != 0)
                SaveData(RXData);           // Valid Stop Bit – End

        }

        Delay( 1/3 Bit );                  // Delay Remaining Bit

    }

} // End SerialRead
```

In this code, notice that that under all circumstances there is a delay for at least a third of a bit. In the application code, this meant that anytime `SerialRead` is called,

at least two-thirds of a bit period is used in the subroutine. This leaves only one-third of a bit period for actually executing mainline code. This is not a terrible hardship because in the nine bits required for each bit, up to five characters could be sent to the LCD with a polling routine on the busy flag to wait for the LCD to complete writing to its display.

In the start bit, notice that I end up waiting a full bit period before returning to the mainline code. This code performs two glitch checks and makes sure that the data poll takes place in the one-third bit period that is closest to the middle of the bit. As I indicated earlier, the possible bit time errors are anywhere from 0.08 to 0.8 percent, which means that any kind of movement from this position should keep the poll well within the bit. The reason why I capture the serial data at the start of the serial input routine is to make sure that it is polled at the same location during the application's execution.

The assembly code itself is quite a bit more complex, with the most important aspect of it being the delay code. In the preceding pseudocode, I label delays simply like `Delay( 1/3 bit )`, but in the actual application code, it is quite a bit more complex. In the application code, I first debugged the application to run properly at 9,600 bps. To do this, I had to make sure that the line is polled at exactly one-third bit period. To do this, I counted through the number of cycles for each action and then put in a delay to make sure that the number of cycles between the `SerCheck` polling macros was exactly 35 μ s using a delay macro similar to the one described elsewhere in this book.

Once I had RS-232 input (which is essentially the complement of TTL/CMOS NRZ serial input) running at 9,600 bps, I then went ahead and made the changes to the code that would allow the switch selection between 9,600 and 1,200 bps. I was looking ahead at doing this, and to accomplish it, I simply selected between two different time delays. The code I normally used looked something like this:

```

    btfsc    SpeedSet                ; Delay 1/3 Cycle
    goto    Speed3P
Delay 35 - 17 - 4
    goto    Speed4P
Speed3P:
    Delay 278 - 17 - 4
    nop
Speed4P:

```

which executes a delay specific to the speed at which the application is running. If the `SpeedSet` flag is set, then the application is running at 1,200 bps; otherwise, 9,600 bps. In this code, the `- 4` operator is used to include the four cycles that were added to select the actual data speed.

With the application running for RS-232, I put the `SerialRead` code into a macro with the ability to select positive (TTL/CMOS) or negative (RS-232) inputs. This further adds to the complexity of trying to see what is happening inside the source code. If you look over the different copies of the application code, you will see how this macro was developed.

You also will see that I put the serial input and LCD output code in two different pages in the source code. The reason for doing this was that there was not enough space for both to reside in the same page. Even though the actual application is just a few instructions longer than 512 (the low-end PIC microcontroller's page size), it cannot be placed in one page. To simplify its application, I placed the code for each type of polarity in its own page.

Normally, when I am creating an application, I would consider placing the code all together and putting in the comparisons between positive and negative incoming serial data. I didn't in this case because there is no 512-instruction PIC16F505, and I had more than enough program memory space to implement the application this way.

Many serial LCD interfaces on the market today have the capability to accept LCD instructions as well as LCD data. The project given here has this capability as well. To send an LCD instruction, the data byte 0x0FE has to be sent first. The next byte received will be written to the LCD as an instruction rather than a data byte.

The most significant problem I had with getting this application running (once I had figured out the switch problem) was at 1,200 bps with the busy flag polling. I incorrectly left the state of the R/S flag set after a data byte write to the LCD, and when polling the value back, a data read was initiated accidentally.

This was a very confusing problem to find because there was no problem at 9,600 bps. I eventually found the problem by looking at the state of the LCD pins when the E strobe was active. In each data transfer, the first action was correct, the data byte was being written to the LCD, but when I went through all the LCD's bits, I discovered that RS was in the wrong state for polling the busy flag.

ULTRA: ULTRASONIC LCD RANGING

With the serial LCD interface completed, I wanted to try to interface with it using a PIC microcontroller to see how useful it would be. The project that I came up with is a distance-measuring tool using the Polaroid 6500 module and transducer (which can be seen in Fig. 21.8). The resulting project can measure distances using the same ultrasonic transducer as many cameras use to set their lenses. The Polaroid 6500 is very easy to work with, although there are a couple of things to watch out for.

The PIC microcontroller used with this project is the PIC12C509. This application also was an experiment to find out how well two PIC microcontrollers with internal RC oscillators could be used to communicate with each other. The positive NRZ signal input to the serial LCD interface presented as the previous project was used without any type of signal conditioning. The actual communications were not perfect, but they did give me some insight into how the internal oscillators of the PIC microcontrollers work and their relative accuracy.

The ultrasonic transducer I chose for this application is the Polaroid 6500, which was first developed in the mid-1970s for use with Polaroid autofocus cameras. The operation of the device is quite simple. A controller outputs a pulse to the module (which can be up to 100 ms long) and then measures the time for the echo to come back. The interface normally is accomplished using two wires connected to a microcontroller.

The black disk that can be seen in Fig. 21.8 is the ultrasonic transducer that both outputs the ultrasonic signal (which can be heard as a click) and receives the echo when



Figure 21.8 The Polaroid 6500 ultrasonic ranging module.

the sound waves bounce off the first object that they come to. The ultrasonic pulse is triggered by the `INIT` line of the Polaroid 6500, and its `ECHO` line is driven high to indicate that the echoed signal has been received. The time required for the signal to return to the transducer is proportional to the distance from the transducer to the object in its line of sight. The 6500 module is specified to work for ranges from 6 in to 35 ft with an error of 1 percent.

The `INIT` enable and `ECHO` return are shown in Fig. 21.9 for an object placed 16 ft from the transducer. On the oscilloscope picture, I have marked the pulse edges with cursor lines.

Before going too far into this project, there are a few things I should mention. The first is a warning: When the ultrasonic signal is sent, an incredible amount of energy is generated. At the transducer, 400 V is produced, and about 1.2 A is drawn from the power source. This is important to realize because nowhere in the Polaroid or any other documentation I found was there a warning about electrical shocks. For this reason, I insist that you do *not* use both hands when handling the operating transducer. When you do handle the transducer, put your left hand in your pocket to make sure that there is no chance for a current path through your heart.

On the Polaroid 6500, there is a nine-conductor connector that is designed for having a flat flex cable connected to it. One of the first things that I did with the connector was to cut it off, unsolder the connections, and put wires into them. This can be seen in Fig. 21.8. This was one of the smartest things that I did because it really made prototyping and experimenting with the 6500 very easy.

To experiment with the Polaroid 6500, I used a PIC12C509, taking advantage of its internal 4-MHz oscillator and reset. With these functions taken care of, all I required of it was three wires, two for interfacing to the Polaroid 6500 and one for sending serial data to the serial LCD interface. The circuit that I used can be seen in Fig. 21.10. The bill of materials for the project is listed in Table 21.3.

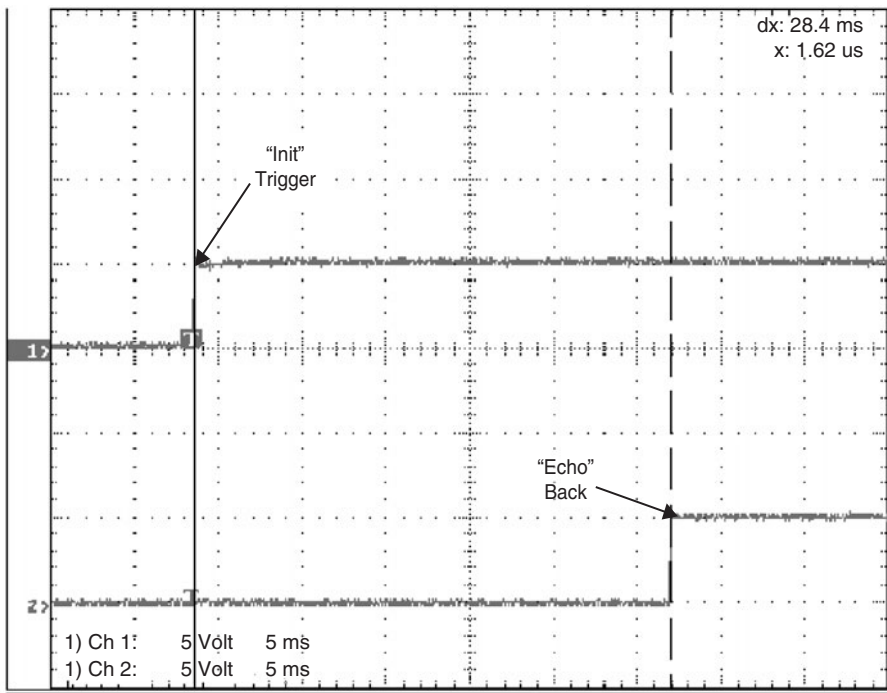


Figure 21.9 The distance from the Polaroid 6500's transducer to some object is found by measuring the time from the INIT trigger to when the ECHO line becomes active.

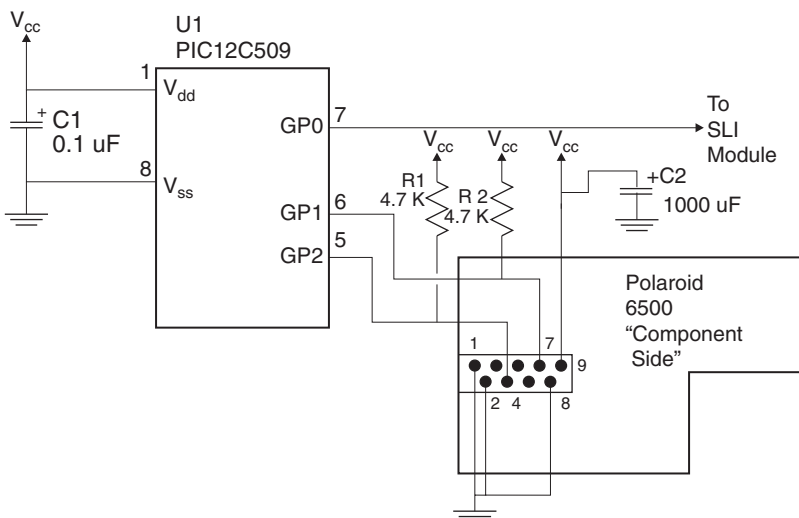


Figure 21.10 The circuit used to measure distances using the Polaroid 6500.

TABLE 21.3 ULTRA BILL OF MATERIALS

REFERENCE DESIGNATOR	DESCRIPTION
U1	PIC12C509/JW
R1, R2	4.7 k Ω , 1/4 W
C1	0.1 μ F tantalum
C2	1,000 μ F, 35 V electrolytic
Polaroid 6500	Polaroid 6500 ultrasonic ranger with transducer
Misc.	Breadboard prototyping system, wiring, 1-A, 5-V power supply

A few comments are necessary about this circuit, the first being the use of a 1-A rated 5-V bench power supply. The Polaroid 6500 is capable of drawing a large amount of current when it is activated. The reasonably large supply and very large (1,000- μ F electrolytic) filter capacitor are to make sure that operation of the Polaroid 6500 does not cause voltage surges that affect the operation of the circuit or power supply.

The `Ultra02.asm` code (found in the `code\Ultra` folder) is only 156 instructions long. The reason why I chose the PIC12C509 and not the PIC12C508 is simply that I had more JW PIC12C509s on hand when I built this project than PIC12C508s. Either PIC microcontroller will work fine without any problems in this application.

This application was really an experiment to learn more about the Polaroid 6500 and the ability of two internally clocked PIC microcontrollers to communicate with each other. To keep the application very simple, I built it on a breadboard, and it only takes 5 to 10 minutes for all five components to be wired together and to the power supply.

In Fig. 21.10 you can get an idea of how I implemented the interface wiring for the Polaroid 6500. After removing the flex socket that comes with the Polaroid 6500, I discovered that the connector vias were in the pattern shown in the schematic, with pin 1 being on the left-hand side and pin 9 being on the right-hand side. The lower pins, halfway between the upper row, are the even pins, and the upper row holds the odd pins. The pinout of the Polaroid 6500 is listed in Table 21.4.

Operation of the 6500 is very straightforward; the `INIT` pin is asserted for typically up to 100 ms, and the controlling processor waits for the `ECHO` to come back. If the surface off of which the ultrasonic burst reflects is too close or too far away, then `ECHO` will never be asserted after `INIT`. The 6500 is specified to work at a minimum distance of 1.33 ft. This can be reduced to about 6 in by asserting the `BINH` line. By asserting `BINH`, any ringing of the transducer will be masked. `BLNK` will ignore any reflected inputs at specific times after `INIT` is asserted. This is useful if there are multiple objects that the 6500 could detect. By blanking out the receiver when echoes are being received, a more accurate reading of the furthest object from the Polaroid 6500 can be made, which

TABLE 21.4 POLAROID 6500 PINOUT		
PIN	FUNCTION	COMMENTS
1	Ground	Connect to 1,000- μ F electrolytic capacitor's cathode
2	BLNK	When asserted, the 6500 will blank out any reflected signal
3	N/C	
4	INIT	When asserted, the 6500 will output the ultrasonic sound burst
5	N/C	
6	OSC	49.4-kHz oscillator output
7	ECHO	Asserted when the returned signal is received
8	BINH	When asserted, the 6500 will ignore any transducer ringing
9	Vcc	+5 V connect to 1,000- μ F electrolytic capacitor's anode

helps to avoid fluctuating readings. For any inputs that are going to be driven, you must pull up the lines with 4.7-k Ω resistors. This is especially important when wiring the 6500 to a microcontroller that has an open drain I/O pin (such as the RA4 in the mid-range PIC microcontrollers) or weak MOSFET pull-ups such as on PORTB. This explanation of the 6500 probably seems quite simplistic. The 6500 can be purchased from a variety of sources on the web.

To simplify operation of the Polaroid 6500, I simply drove operated the ranger with BINH and BLNK pulled to ground. This makes some of the measurements a bit unreliable (i.e., jumping up and down depending on what is the first object to echo back to the Polaroid 6500), but other than this, the application works very well.

Once I had created a simple application for testing the operation of the Polaroid 6500 and getting oscilloscope readings (as in Fig. 21.9), I then wanted to convert the distance measurement to feet and inches and display it on an LCD.

To determine the distance between the Polaroid 6500's transducer and an object, the time between initiation of the INIT pulse and the ECHO return is measured. Using the PIC12C509 running the internal 4-MHz clock, this time was measured in 148- μ s increments.

The 148- μ s increment probably seems like an unusual measurement, but it is based on the speed of sound for ultrasonic frequencies. Assuming that the speed of sound at sea level for a 40-kHz signal is 1,127 ft/s (13,523 in/s), the flight time from the transducer to an object and back (twice this distance) is 147.9 μ s. Using a 148- μ s check results in a 0.068 percent error, which should be insignificant in the actual measurement, and allows the PIC12C509 to be used with the internal oscillator without modification.

The Polaroid 6500 is specified to a maximum distance of 35 ft (which is 420 or 0x01A4 in in the application code). During operation, I wait for the ECHO line to

become active, incrementing a `Distance` counter once every 148 μ s and indicating that there is “no echo” if the counter reaches 420.

The code that does this distance and no echo check is

```

    clrf    Distance          ; clear the Distance Calculation
    clrf    Distance + 1

    bsf     Init              ; Toggle the "Init" Pin High

EchoLoop:                                ; Loop Here until

    movf    Distance, w      ; At 35' (Max Distance)?
    xorlw   0x0A4            ; = 420?
    btfsc   STATUS, Z
    decfsz  Distance + 1, w  ; Low byte == 0x0A4, High Byte == 1?
    goto    $ + 2
    goto    NoEcho           ; - No, Give up

Delay 148 - 12                    ; Make sure each Loop is 148  $\mu$ secs (1 ")

    incf    Distance, f      ; Increment the Distance Count
    btfsc   STATUS, Z
    incf    Distance + 1, f

    btfss   Echo              ; Wait for "Echo"
    goto    EchoLoop

    bcf     Init              ; Turn off "Init" Pin

```

and bears a few comments. In the code after the `EchoLoop` label, I first compare the 2-byte `Distance` variable to see if it is equal to 0x01A4. This is done by first setting the zero flag if the lower byte is 0x0A4 and then optionally decrementing the upper byte using the `decfsz` instruction to check if it is equal to 1 as well. This code is quite efficient, although probably difficult to walk through if you are new to the PIC microcontroller, and it only works if the value being checked for has a high byte of 1. The `Delay` macro is the same one as I developed for the serial LCD interface project.

Once a valid distance has been determined, I then convert the value into feet and inches. To convert the `Distance` into feet, I divide it by 12 using the code

```

    clrf    Feet              ; Clear the Number of Feet to
Display

ConvertLoop:

    movlw   12                ; Is the Value > 12?
    subwf   Distance, w
    btfss   Distance + 1, 0    ; Rolled Over to Second Byte?

```

```
    btfsc STATUS, C          ; Negative Value?
    goto $ + 2
    goto HaveFeet           ; Remaining Distance is Less than 12

    btfss STATUS, C          ; Shift Down?
    decf Distance + 1, f

    movwf Distance          ; Save the New Value

    incf Feet, f             ; Add to the Foot Counter

    goto ConvertLoop

HaveFeet:                    ; "Feet" is Correct, Distance has Inches
```

This code is a dividing a 2-byte variable by a single-byte constant (12 in this case). Elsewhere in this book I have presented you with other division-by-constant algorithms, but I used the repeated-subtraction one here because I knew that the maximum distance would be 35 ft. This results in a maximum number of cycles through the loop of 420 instruction cycles, or 420 μ s in this application.

In this code, note that I save the result in w register before jumping to `HaveFeet`. If the result is greater than 0, then the new value is stored back into `Distance`. When `HaveFeet` is executed, the `Feet` variable has the number of feet found in the `Distance`, whereas the lower byte of `Distance` has the remainder in inches. These two values are then sent to the serial LCD interface.

The NRZ serial output routine is a very standard “bit banging” output routine with the carry flag used to indicate the bit value after the data has been shifted.

When using the Polaroid 6500 PIC12C509 distance-measuring circuit initially with the serial LCD interface, I found that there were a number of errors in the transmitted data. When I looked at the data being sent on an oscilloscope, I found that the bit periods were in error by about 1.7 percent. By programming another PIC12C509, I found that I could reduce the bit period error to about 0, at which point the data presented on the LCD display was perfect.

After doing this, I realized that I probably should have tried to calibrate the built-in oscillator in the PIC16C505 built into the serial LCD interface. I suspect that the part I used in the serial LCD interface also has a reasonably high error that added to the error of the PIC12C509. By choosing a different device, I was able to decrease the combined error to the point where the data was accurate under all circumstances.

Key: SWITCH MATRIX KEY MATRIX

A switch matrix keyboard is a series of switches wired in rows and columns. The switches can be read by pulling up each row individually, then tying down a column to ground, and then seeing if a switch is pulling a row to ground through the column. This is shown in Fig. 21.11. The typical example shown for interfacing a microcontroller to a switch matrix keyboard is a 4×4 keypad. While this can be useful in some applications,

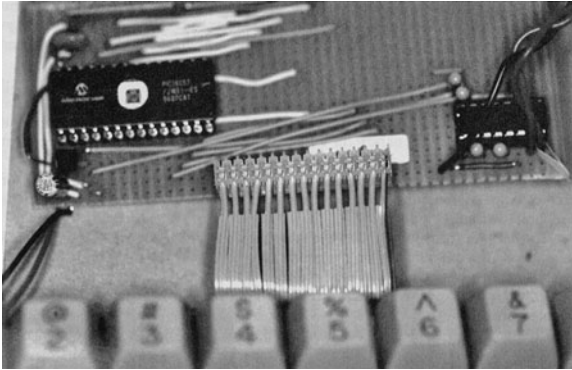


Figure 21.11 The switch matrix keypad circuit was built on a vector prototyping board and provides a wiring interface between the PIC MCU and the switch matrix keypad.

often only a full QWERTY keyboard will do. For this project, I wanted to go through how a switch matrix keyboard could be attached to a PIC microcontroller.

The pull-down transistors shown in Fig. 21.12 can be either discrete transistors or PIC I/O pins individually kept in input mode and changed to output mode with a 0 or low-voltage output to simulate the switch column being pulled down to ground. From the figure it can be seen that when one of the switches is closed and the column it is connected to is pulled down to ground, the receiver will sense a low value. If the switch were open and the column were pulled down to ground, the pull-up would cause the PIC microcontroller to sense a high value. This method of switch sensing is analogous to having multiple open-collector outputs on a single pulled-up line.

To scan the keyboard, each pull-down is enabled individually in series, and any time an input value is low, the key's switch at that row/column address is closed and the key pressed.

With this scheme, there are a few issues to consider:

- 1 How should the keyboard be wired to the controller?
- 2 What about switch bouncing?

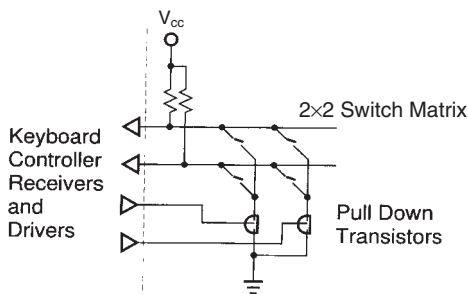


Figure 21.12 Switch matrix keyboard connections.

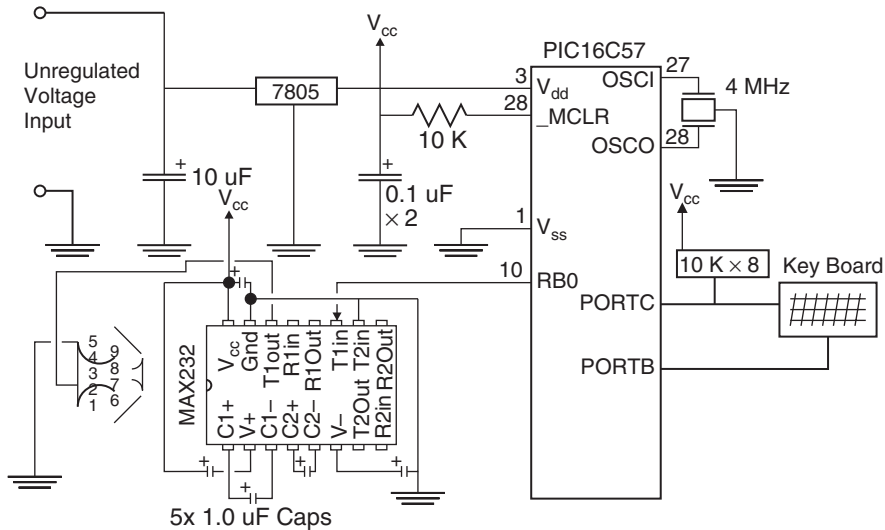


Figure 21.13 Keyboard interface circuitry.

- 3** What about multiple keys pressed at the same time?
- 4** What about Shift/Ctrl/Alt/Function key modifiers?

In developing the first software application for the project **Key1**, I had to understand these issues and have a plan to deal with them. The source for the **Key** application can be found in the `code\Key` folder.

Note that these applications were designed to work with a keyboard that I found for a dollar in a surplus store in Toronto a number of years ago. The keyboard was manufactured by General Instruments for the Texas Instruments TI-1000 personal computer that was last built over 25 years ago. There is no visible part number on the device itself except for the Texas Instruments logo. I doubt that you will be able to find this keyboard, but this still makes this application and the process I went through to decode the keyboard useful as a reference for whatever keyboards or keypads you want to work with.

The schematic of the circuit I used is shown in Fig. 21.13. To assemble my prototype, I used a vector board, which is a PCB that has a series of copper strips on the backside; to make a circuit, wires (or components) are soldered to a specific strip. The strip can be cut to allow multiple circuits to use the same strip. This method of prototyping is somewhat obscure, but it can be very useful when single-bussed devices, such as the keyboard (or an LCD), are being prototyped in the circuit. I have used the vector board in a few of the projects simply because it is reasonable for the applications. The bill of materials for the circuit is listed in Table 21.5.

For my prototype, I used a 9-V alkaline radio battery to the circuit to power it. My original intention was to use the keyboard interface with the serial LCD interface (presented earlier in this chapter) to make a simple 1,200 bps RS-232 TTY terminal. Actually, this still can be done using these two components.

TABLE 21.5 KEY BILL OF MATERIALS	
REFERENCE DESIGNATOR	DESCRIPTION
PIC16C57	PIC16C57-JW
78-5	78L05 in TO-92 package
MAX232	Maxim MAX232
4-MHz	4-MHZ ceramic resonator with built-in capacitors
10-kΩ	10 kΩ, 1/4 W
10-kΩ × 8	10-kΩ single common tap 9-pin SIP
10-μF	10-μF, 35-V electrolytic
2 × 0.1-μF	0.1-μF, 16-V tantalum
RS-232 connector	9-pin female D-shell connector
Misc.	Prototype card, hook-up wire, 15 × 1 IDC connector of keyboard

Trying to figure out how a keyboard should be wired to the PIC microcontroller when you don't have any information is not as daunting a task as it would appear to be. When I bought the keyboard for this project, the only identifying feature on it with regard to electrical connections was a strip indicating pin 1 on a 15-pin IDC connector.

The first thing that I did was set up a matrix and using a digital multimeter. I beeped out every key with the two different connector pins. With this information, I created the matrix shown below:

Pin	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1					"V"	"R"			"4"		"M"	"J"	"F"	"7"	"U"
2					"C"	"E"			"3"		","	"k"	"D"	"8"	"I"
3					"X"	"W"			"2"		"."	"L"	"S"	"9"	"O"
4						Ctrl			Fctn		"="	" "	"Shift		Ent
5						"B"	"Z"								
6							"T"	"Q"							
7									"5"		"N"	"H"	"G"	"6"	"Y"
8									"1"		"/"	";"	"A"	"0"	"P"
9															
10															
11															
12															
13															
14															
15															

Once this was done, I manipulated the table until I could get a good understanding of how the keyboard was wired and what would be the best way to attach it to the PIC microcontroller.

The design point I decided on was setting up eight rows (or register bits) for each column. I defined the *row* as where I put the pull-up and the *column* as the pin I pull to ground. The data was transformed into the following matrix, where the rows and columns are the pin numbers on the connector:

Column	Row ->	5	6	9	11	12	13	14	15
1		"V"	"R"	"4"	"M"	"J"	"F"	"7"	"U"
2		"C"	"E"	"3"	" , "	"K"	"D"	"8"	"I"
3		"X"	"W"	"2"	" . "	"L"	"S"	"9"	"O"
4			Ctrl	Func	"="	" "	Shift		Enter
7		"B"	"T"	"5"	"N"	"H"	"G"	"6"	"Y"
8			"Q"	"1"	" / "	" ; "	"A"	"0"	"P"
10				Caps					

With this information, I was ready to specify the wiring. At this point, the task was simply to wire the rows and columns to the PIC microcontroller. The rows having the pull-up noted above and connected to PORTC and the columns pulled down within PORTB.

I used a PIC16C57 for the keyboard and an RS-232 interface because it had more than enough I/O pins to handle the 15 pins required by the keyboard. I outputted the NRZ serial data (through a MAX232) to my PC's RS-232 HyperTerminal emulator so that I could monitor what was coming out (and, if needed, send debug information as well).

To read the keys, I used the algorithm

```

KeyPreviously = 0          ; Nothing Currently read

Loop:

    Delay4ms                ; Delay for key Debouncing

    if KeyPreviously = 0

        KeyCount = 0        ; Reset # of Times through Loop

        for i = 0 to # columns and No KeyPreviously
            Scan Column      ; Check each Column and Set
                               ; KeyPreviously
        else
            ; Else Key Previously pressed

            if KeyPreviously Still Pressed

                KeyCount = KeyCount + 1    ; Increment the Actual Count

                if KeyCount == 5          ; Do we have the First Press?

```

```

        Send KeyPreviously

    else
        If KeyCount = 128            ; Have we Waited 1 Sec?
            Send KeyPreviously

        else
            if KeyCount = 192        ; have we Waited another 1/2 Sec?
                Send KeyPreviously
                KeyCount = 128      ; Reset for Next AutoSend

    else                            ; Key was lifted
        KeyPreviously = 0          ; Start all over

goto Loop

```

This algorithm handles the key bouncing by requiring that five consecutive polls 4 ms apart sense the key being pressed. When the bit goes high, the key read is reset until the next key press pulls down the row and the counting resumes.

Multiple key presses (which obviously can happen when more than two keys in the same column are pressed at the same time) are resolved within the `ScanColumn` routine, and it is resolved by taking the lowest active bit in the column. The `Send` routine looks up the ASCII code to send by reading the value in a row/column table.

It should be noted that the key modifiers (Ctrl, Func, etc.) are always masked off in `Key1`. I didn't bother to put in the modifiers because I didn't have an application that required the keyboard (and being lazy . . .). They can be implemented easily by adding new tables for each modifier and then, before calling the table to look up the value, adding an offset to the correct table values.

Note that I wouldn't bother debouncing the key modifiers because they are not the action that initiates the action of sending the keys. They are just used to make sure that the proper keys are sent.

The code presented here (`Key1.asm`) could be ported very easily to a mid-range PIC microcontroller with the advantage that the TMR0 interrupt could be used to initiate the scan, allowing the keyboard read code to be run totally in the background. Another advantage of using a mid-range device is the ease in which the TRIS bits could be rotated directly within software rather than keeping track of the current TRIS value, as I have to do in this application.

Along with porting the application to another PIC microcontroller, you also can change the tables I created for interpreting the different scan codes to another switch matrix keyboard easily. This application is a bit high end; if you were to use a 4×4 switch pad, this code could be used quite easily within one port rather than the two presented here. One last enhancement that could be made to this application is to use the built-in pull-ups on PORTB of most PIC microcontroller devices. This would avoid the need for the 9-pin common tap single in-line package (SIP) resistor that I used in this application.

Mid-Range Devices

By far the most popular PIC microcontroller family is the mid-range family. This family is very well designed for sophisticated single-chip applications using the interrupt-enabled processor. Advanced I/O features can simplify applications and avoid the need for “bit banging” applications. The mid-range devices, with these features, often allow simpler applications than what might be possible ordinarily.

In the following example projects, you will see that the feature of the mid-range family that I take the most advantage of is the interrupt capability. As I’ve said elsewhere, this allows more sophisticated applications for the PIC microcontroller and, in many cases, simplifies the final applications. When I use interrupts in an application, note that I rarely use more than one interrupt source at a time, and when I do, the second one is almost always the TMR0 interrupt to provide timing functions in the projects.

The projects presented in the following sections are used not only to demonstrate different I/O and code methods but to also give you some concrete examples of how the different peripheral I/O functions work in an application.

Clock: ANALOG CLOCK

This was one of my first PIC microcontroller applications and one that turned out to be very successful—in fact, it has spent about 5 years on my desk at work keeping just about perfect time (Fig. 21.14). This unique timepiece can be wire-wrapped together in about an afternoon. Seventy-two LEDs (or 84 if you build it the same way I did and use 2 LEDs for the hours) provide a digital replica of an analog clock. For the application, I used a 32.768-kHz quartz watch crystal and a 2.8-V lithium PC backup battery to make sure that I didn’t have to reset the clock if power was lost.

The circuit uses five 74LS154 four to sixteen demultiplexors, as I’ve shown in Fig. 21.15, to drive the LEDs. One minute LED and one hour LED are displayed at any time from

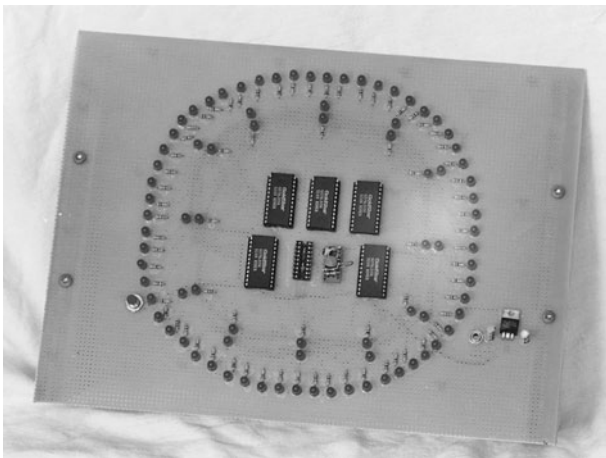


Figure 21.14 The PIC microcontroller-based analog clock is an interesting conversation piece.

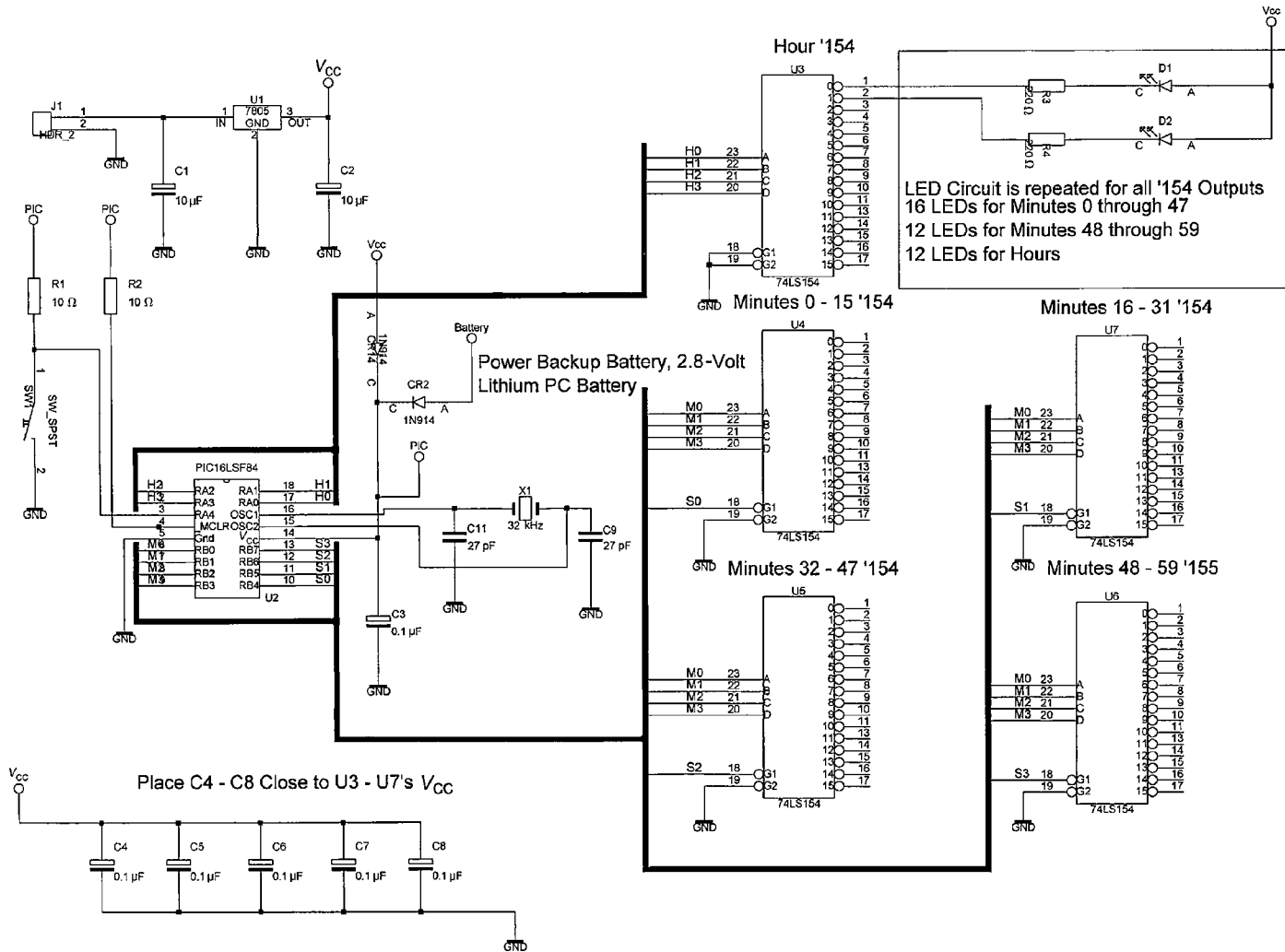


Figure 21.15 The simple PIC microcontroller clock uses the output from the I/O pins to select the demultiplexors that are used to drive the appropriate LED.

TABLE 21.6 CLOCK BILL OF MATERIALS

REFERENCE DESIGNATOR	DESCRIPTION
U1	7805 +5-V regulator in a TO-220 package
U2	PIC16LF84-04/P
U3-U7	74LS154 four to 16 demultiplexor
X1	32.768-kHz watch crystal
CR1-CR2	1N914 silicon diode
D1-D72	5-mm-diameter red LEDs
D73-D84	5-mm-diameter red LEDs used if hours consist of two LEDs each
R1-R2	10 k Ω , 1/4 W
R3-R74	220- Ω , 1/4 W
C1-C2	10- μ F, 35-V electrolytic
C3-C8	0.1- μ F, 16-V tantalum
C9, C11	27 pF any type
J1	2.5-mm power plug
SW1	Momentary-on push-button switch
Battery	2.8-V lithium PC backup battery
Misc	Prototype board, wire wrap sockets, wire, +7-V output PC power supply

PORTB and PORTA or the PIC16LF84, respectively. The bill of materials for the analog clock is listed in Table 21.6.

For my prototype, I point-to-point wired the circuit. For your version, I recommend that you either design an embedded PCB for the circuit or wire-wrap it. Point-to-point wiring took me about a day's worth of effort on this circuit. If you are going to wire-wrap the circuit, I recommend that you wrap directly to the LED posts—this will save you a lot of time and make your life a lot easier.

I specified a low-voltage PIC microcontroller for this circuit because I've included a simple battery backup circuit using a 2.8-V lithium PC battery to make sure that the time isn't lost in the case of occasional power outages. If power is ever lost to the circuit (I powered my prototype from a wall-mounted ac/dc power adapter), the lithium battery will keep the PIC microcontroller running until power is restored. The lithium battery will run the PIC microcontroller for a very long time because the PIC microcontroller's current consumption is on the order of microamps, and the 2.1 V of the PIC power output from the PIC microcontroller is insufficient to cause a parasitic drain in the 74LS154s.

To keep the time, I allow TMRO to run continuously with a divide by 64 prescaler that overflows once every second. In the software, the T0IF bit is polled continually, and if it is set, then it is reset, and the button is checked for being pressed; if it isn't, the seconds counter is updated with, if necessary, the minutes and hours.

The entire application is quite simple and could be written out in pseudocode as

```
main()                                // Simple PIC microcontroller Clock
{

    int  Seconds = 0;
    int  Minutes = 0;
    int  TimeInc = 1;

    OPTION_REG = TMRO - Internal Clock/TMRO - Prescaler/Prescaler = 64;

    PORTB = 0x0E0;                    // PORTB outputs "Minutes"
    TRISB = 0;                        // PORTB All Output
    PORTA = 0;                        // PORTA is the "Hours" Counter
    TRISA = 0x010;                    // RA4 is an Input Pin

    TMRO = 0;                         // Wait 1 Second before Starting

    while (1 == 1) {                  // Loop Forever

        while ((INTCON & (1 << T0IF)) == 0);
        T0IF = 0;                     // One Second has Gone By

        if (RA4 == 0) {               // The Button has been Pressed
            TimeUpdate(TimeInc); // Increment the Time
            TimeInc = ((TimeInc << 1) + 1) ^ 0x03F;
            Seconds = 0;
        } else {                      // Update the Second Counter
            TimeInc = 1;              // Button isn't pressed
            Seconds = Seconds + 1;
            if (Seconds > 59) {        // Minute has gone by
                Timeupdate(TimeInc);
                Seconds = 0;
            }
        }
    }
} // End Simple PIC microcontroller Clock
```

This code actually has been updated for this edition of the book and uses the clock button time-set algorithm I first came up with for *Handbook of Microcontrollers*. When the button is first pressed (it is polled once per second), the time is incremented by 1 minute. After incrementing the time, the increment value is shifted up by 1 and has 1 added to it to 0x03F (63 decimal). When 63 decimal is added to the Minutes, it will roll over

and increment the hours. Using this algorithm, the clock can roll over 12 hours in just 20 seconds—a lot faster than if you were to hold an incrementing minutes button down. In addition, it offers nice capabilities to set the clock accurately because the incrementing period changes according to how long the button is pressed.

The most complex part of the application is the `Timeupdate` subroutine. This subroutine is complex because the upper 4 bits of `PORTB` are used to select which 74LS154 is enabled for output (with the lower 4 bits selecting the 16-bit output of the 74LS154). The code for updating the time uses the `Minutes` variable, and adding the passed value to it appropriately updates both the minutes and hours depending on the result.

```
Timeupdate:                                ; Add contents of "w" to PORTB (Minutes)

    addwf    Minutes, f                    ; Save the Update

    movlw    60                            ; Are the Minutes Rolled Over?
    subwf    Minutes, w
    btfss    STATUS, C
    goto     TUMinutes                    ; No, Just Save the New Value

    clrf     Minutes                       ; Reset the Minutes

    incf     PORTA, w                      ; Do the Hours get Incremented?
    andwf    0x00F
    xorlw    0x00C
    btfsc    STATUS, Z                    ; Are we Up to 12?
    movlw    0x00C                        ; Yes — Reset to Zero
    xorlw    0x00C                        ; Convert the Time Back
    movwf    PORTA

TUMinutes:                                ; Convert Minutes to LED Positions
    movlw    0x00F                        ; Get the 16 Minute selection
    andwf    Minutes, w
    movwf    Temp
    call     Get154                        ; Get the enable for the specific `154
    iorwf    Temp, w                      ; Combine with the Minutes for the `154
    movwf    PORTB

    return

Get154:                                    ; Use the MS 2 Bits of "Minutes" to
    movlw    HIGH Get154Table              ; Get the `154 Enable Setting
    movwf    PCLATH
    swapf    Minutes, w
    andlw    2
    addlw    LOW Get154Table               ; Compute the Table Address
    btfsc    STATUS, C
    incf     PCLATH
    movwf    PCL
```

Get154Table:

```
retlw 0x0E0          ; Minutes 0 – 15
retlw 0x0D0          ; Minutes 16 – 31
retlw 0x0B0          ; Minutes 32 – 47
retlw 0x070          ; Minutes 48 – 59
```

This code takes a maximum of 34 instruction cycles to execute. This code, with the code presented above, means that the vast majority of time the PIC microcontroller spends processing in this application is spent polling the TOIF flag instead of calculating the new LED's display value. This is important to avoid any possibility that the TOIF flag being set is missed (and the time is lost).

xmas : CHRISTMAS DECORATION WITH BLINKING LIGHT AND MUSIC

If you look through the hobbyist electronics magazines in December, you are sure to find a number of flashing-light PIC microcontroller applications. These applications usually use a number of LEDs to flash on and off in a seemingly random pattern. These lights are often put in a Christmas tree decorations or a stand-alone decoration, as shown in the X'mas project (Fig. 21.16), which has 15 flashing LEDs and can play a tune.

This is the third time I've created a lighted Christmas tree with a PIC microcontroller. In the first case, I hand-wired all the connections (which made it truly horrific to wire).

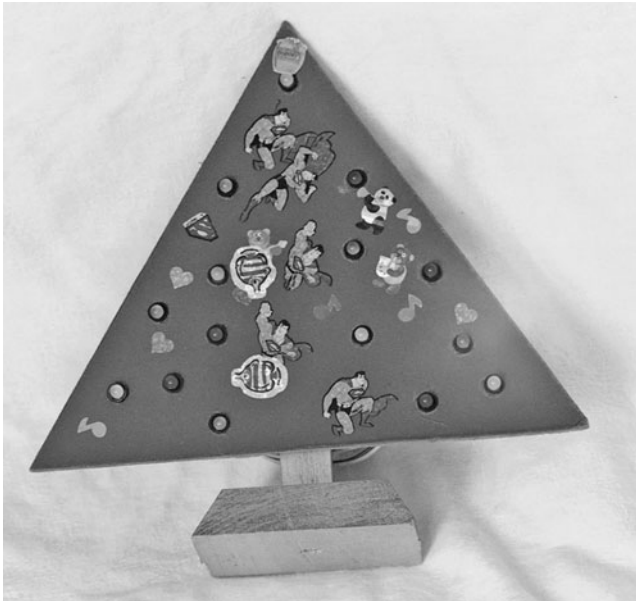


Figure 21.16 The Christmas tree decoration with stickers for ornaments.

The second version used a predesigned PCB card, but the design was lost in a hard file crash (before I started religiously backing up files onto CD-ROM), and the software was written for an early PIC microcontroller compiler that never worked extremely well. I ended up updating it just for the light application (and the tune generator that went with it). Also with the second version, I created an unnecessarily complex tune generator that requires two separate PC applications to run before the code could be built with the tune installed.

This third version avoids the problems of the other two, and by using LEDs with leads soldered to them, the build time for the entire project is just a few hours (not including decoration paint drying time). The application supports up to 16 LED outputs, a speaker with software that can drive a 128-note tune, and a built-in power supply, so the application can be run from a wall-mounted ac/dc adapter. The schematic is shown in Fig. 21.17. The bill of materials is listed in Table 21.7.

There are a couple of things to notice in this application. The first is that there is no on/off switch for power. I left this off because I assumed that once power was applied, the decoration would run continuously. Next, I do not interconnect the 74LS374s, which are used as shift registers, because there are ample pins available for the application, and randomizing can be done in software. Lastly, I used a 3.58-MHz ceramic resonator instead of my typical 4.0-MHz one because I ran out of 4-MHz ceramic resonators. This change did not result in a major problem with timing the application, as will be discussed below. The macro calculator was used for calculating delays, and a 4-MHz ceramic resonator or crystal could be substituted in its place.

The software used to run the application could be considered to be a simple multitasker with two “processes.” The critical process is the tune process, which plays a tune programmed into a tape. The other process is a pseudorandom number generator using as a linear feedback shift register that runs in the foreground. The application can be modeled as

```
main ( ) // Xmas Tree high level code
{
    int TuneDelay = 1;
    int TuneIndex = 0;
    int Data = 0x0FFFF;

    PORTA = output; // Set Up Outputs
    OPTION = TMRO Instruction Clock | TMRO Prescaler; // Set Up Timer
    INTCON = GIE | TOIE; // Enable Timer Interrupt

    While (1 == 1) { // Loop Forever Updating LEDs

        dlay(); // Delay 1 second

        Data = (data << 1) + (Data.15 ^ Data.12); // Create Pseudo Random Number
```

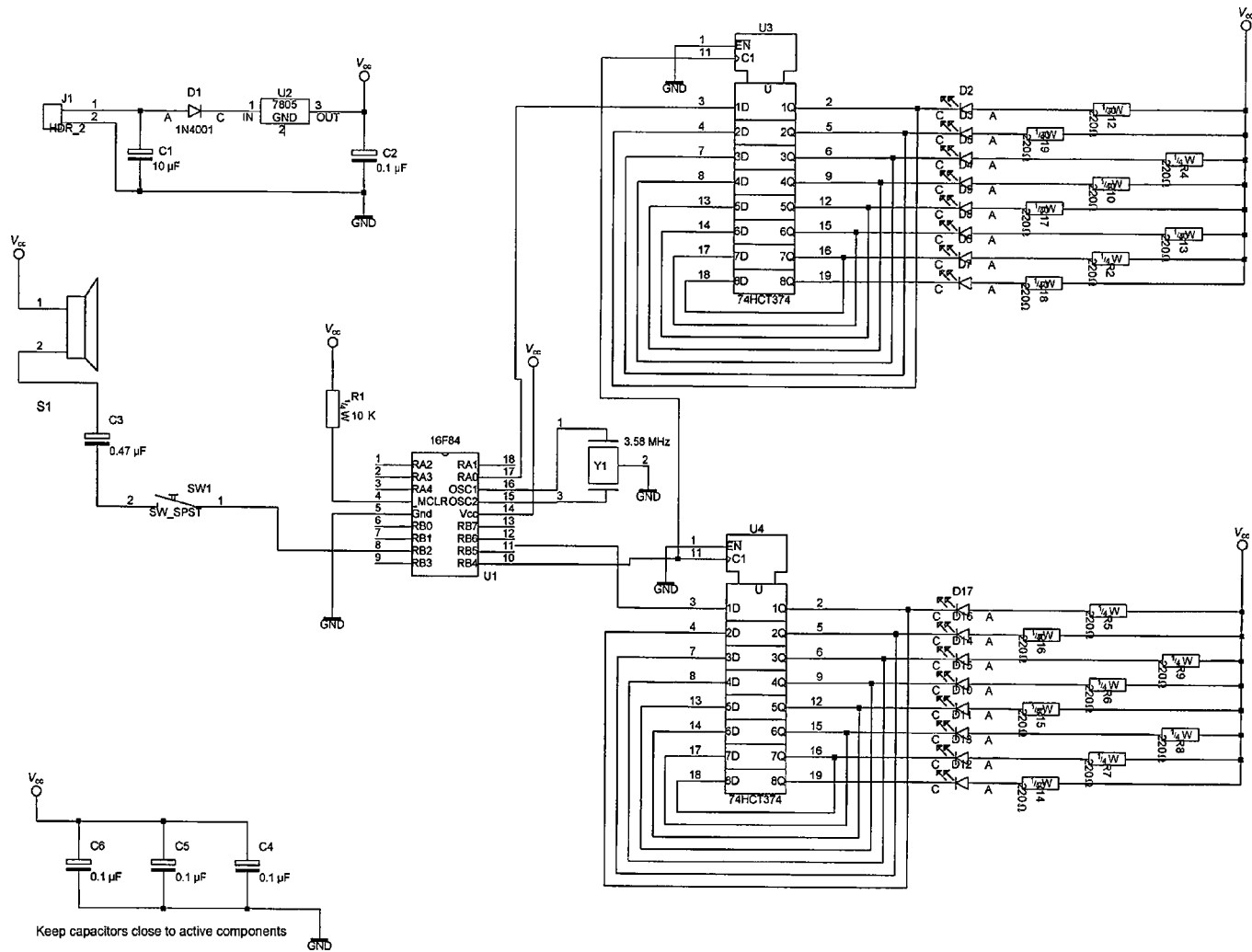


Figure 21.17 Christmas decoration schematic.

TABLE 21.7 XMAS BILL OF MATERIALS	
REFERENCE DESIGNATOR	DESCRIPTION
U1	PIC16F84-P/04
U2	7805
U3-U4	74LS374
Y1	3.58-MHz ceramic resonator with built-in capacitors
D1	1N4001 silicon diode
D2-D17	Multicolored LEDs with leads soldered on
C1	10-μF, 16-V electrolytic
C2, C4-C6	0.1-μF, 16=V tantalum
C3	0.47 μF, any type
R1	10 kΩ, 1/4 W
R2-R19	220 Ω, 1/4 W
S1	Single pole, single throw
S1	Piezo speaker
J1	2.5-mm power connector
Misc	PCB board, wall-mounted 8-V, 500-mA ac/dc power converter

```
        ShiftOut(Data);           //    and Shift it out to the LEDs
    }
} // End Xmas Mainline

Interrupt TMROInt()               // Timer0 Interrupt Handler
{

    TMRO = Note(TuneIndex);        // Reset TMR0 to Delay Again
    TOIF = 0;

    TuneDelay = TuneDelay - 1;

    if (TuneDelay == 0) {    1    // If Tune Delay Finished, Output a

        TuneIndex = TuneIndex - 1; // New Note
        if Note(TuneIndex) == 0)   // At End of the Tune?
            Tune index = 0;         // Yes, Reset and Start Again
```

```

TuneDelay = Delay(TuneIndex);

} else                                     // Toggle Note
    if (Note(TuneIndex) != Pause)
        SpkrOutput = SpkrOutput ^ 1;

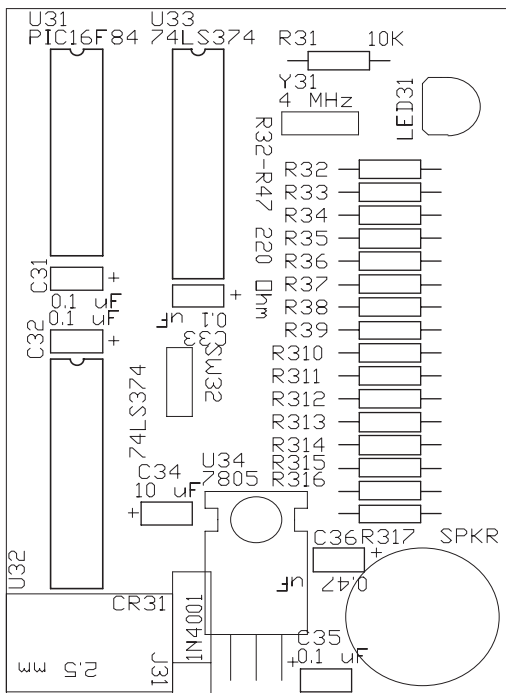
} // End TMR0 int.

```

The table (listed as the “Note” and “Delay” functions in the code above) contains the tune with notes and duration programmed in using the `note add` macro. This macro will allow development of the code needed for a tune very quickly—just transcribe the tune into the note range described below, and program a PIC microcontroller.

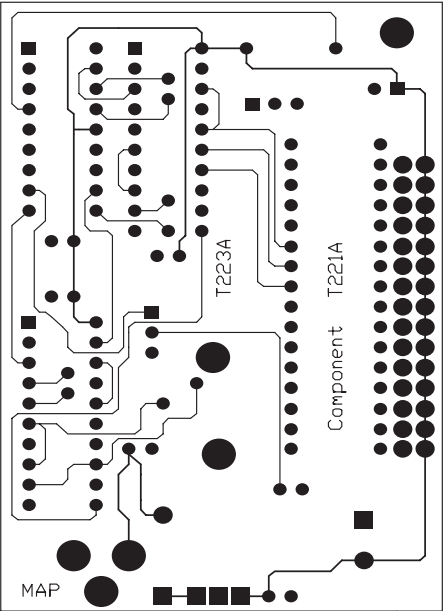
After getting a PCB built, you will be able to solder the application together in less than an hour. The PCB design is shown in Figs. 21.18 through 21.20.

For my application, I created the Christmas tree shown in Fig. 21.16. The PCB has two 0.125-in holes drilled in it for your use in mounting it to a display. When I mounted my board to the Christmas tree, I simply screwed it to the back of the tree using small wood screws. This is shown in Fig. 21.21. When I assembled this project,



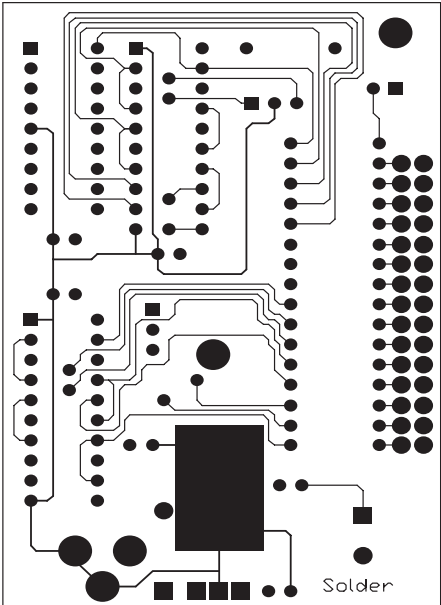
T225A Overlay

Figure 21.18 Silkscreen for component placement on the Christmas decoration.



T225A Top Layer

Figure 21.19 The topside PCB traces for the Christmas decoration.



T225A Bottom Layer

Figure 21.20 The bottom PCB traces for the Christmas decoration.

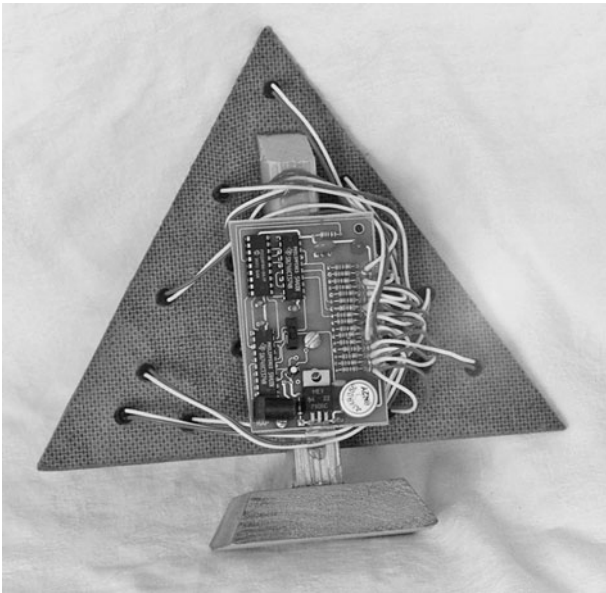


Figure 21.21 The rear of the Christmas decoration showing the PCB and the wiring.

I tried to keep the LED wires away from the PCB to allow removal of the PIC microcontroller for reprogramming, as well as allowing me to fix any potential problems (there weren't any).

For previous Christmas tree projects, I laboriously soldered leads onto LEDs and then glued them into holes on the tree. For this project, I was able to find LEDs that were presoldered onto leads and glued into panel-mount plastic carriers. These parts are a bit more expensive than buying LEDs and wires, but the time saved made the project quite enjoyable rather than a particularly odious chore.

For my Christmas tree, I used a piece of scrap pressboard cut into an equilateral triangle 8 in on a side and a $3/4$ -in piece of wood for the stand, as shown in Fig. 21.22. I painted the triangle forest green and the base and stand gold. To assemble the tree, I simply glued the triangle to the stand and screwed the stand to the base.

After the glue and paint had dried, I inserted the LEDs into the holes and soldered their leads to the PCB. Total time working on the tree was about 2 hours, with a total elapsed time of 3 days (mostly for paint and glue to dry).

When the tree was completed, it seemed to be a bit bare. To help dress it up, I bought a few sheets of small sparkly stickers that had various toys and Christmas decorations and gave them and the tree to my $4\frac{1}{2}$ -year-old daughter to decorate. This turned out to be a lot of fun for her. As a bonus, if enamel paint is used with a smooth surface for the tree triangle, the stickers can be peeled off easily and the tree redecorated the next year.

Creating tunes is a relatively simple chore. Once you have chosen the song you would like the application to play, you will have to transpose it into the notes provided in Table 21.8.

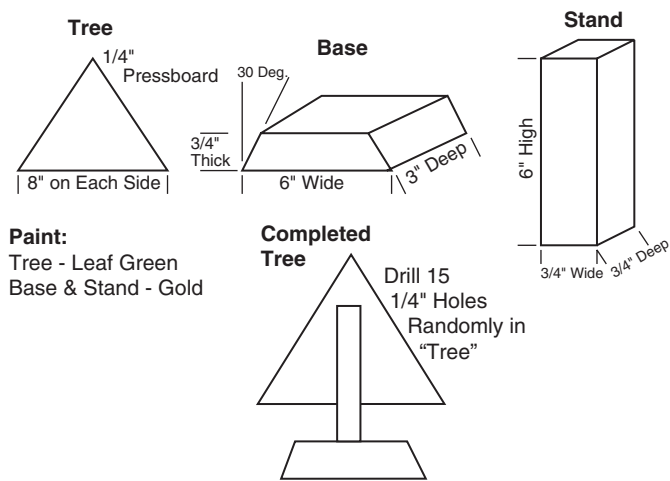


Figure 21.22 The wood pieces for the Christmas decoration.

TABLE 21.8 XMAS NOTE TABLE			
NOTE	LABEL	FREQUENCY, HZ	COMMENTS
B	1nB	494	B below middle C
C	nC	523	
C#	nCS	554	Middle C
D	nD	587	
D#	nDS	622	
E	nE	659	
F	nF	699	
F#	nFS	740	
G	nG	784	
G#	nGS	831	
A	nA	880	
A#	nAS	923	
B	nB	988	
C	hnC	1047	
C#	hnCS	1109	Octave above middle C
D	hnD	1175	
D#	hnDS	1245	
E	hnE	1319	
Pause	nP	N/A	Pause a sixteenth beat

When specifying the time for the note, the delay is given in 16 beats. To specify a tune, the notes are specified along with the number of sixteenth beats. For example, the first line of “Oh Christmas Tree,” which is

“Oh Christ-mas tree, Christ-mas tree, how lovely are . . .”

1/8D 3/16G 1/16G 1/8A 3/16B 1/16B 3/8B 1/8B 1/8A 1/4C . . .

is written out as

nD, 2, nG, 3, nG, 1, nG, 6, nA, 2, nB, 3, nB, 1, nB, 6, nB, 2, nA, 2, nB, 2, hnC, 4 . . .

The application is timed for 72 beats per minute, which allows me to do a pretty good rendition of “O Christmas Tree.”

Other songs can be programmed in very easily; to do this, I recommend buying a cheap beginners piano book with Christmas carols (or other songs) and transcribing the notes. This may be surprising, but the most difficult aspect of this application for me was to find and transcribe a version of “O Christmas Tree” that had notes that fell in the proper range for the application.

For driving the LEDs seemingly randomly, the pseudorandom formula

$$\text{Data} = (\text{data} \ll 1) + (\text{data}.15 \wedge \text{data}.12)$$

shifts the current value up by 1 bit and then fills in the least significant bit with the most significant bit and another XORed together. This seems unbelievably simple (and it is), but it does a very good job of randomizing the data as a very simple linear feedback shift register.

The two 74LS374s are wired as synchronous serial shift registers. I use this format for a variety of different applications to shift data serially out to a parallel I/O. To shift the data out, notice that I perform a shift on the data with the carry up from the lower byte being used as the LSB of the upper (the lower bytes LSB is the pseudorandom value discussed earlier).

The `xmax040.asm` application code is in the `code/Xmas` folder. If you decide to put in your own tune, then change the name of the file so that you don’t overwrite the original application.

IRTank: TV IR REMOTE-CONTROL ROBOT

In this chapter I present two different robot features that can be controlled by a PIC. This section will show you how to use the PIC microcontroller with a TV remote control to send commands to a tracked vehicle. At the end of this chapter I’ll go through a programmable servo controller.

The genesis of this project was an article in *Electronics Now*. In the preceding project I made the observation that hobbyist magazines always have Christmas decorations in their November–December issues. On the other hand, you’ll notice a number of different robot designs throughout the year. The magazine project presented a simple

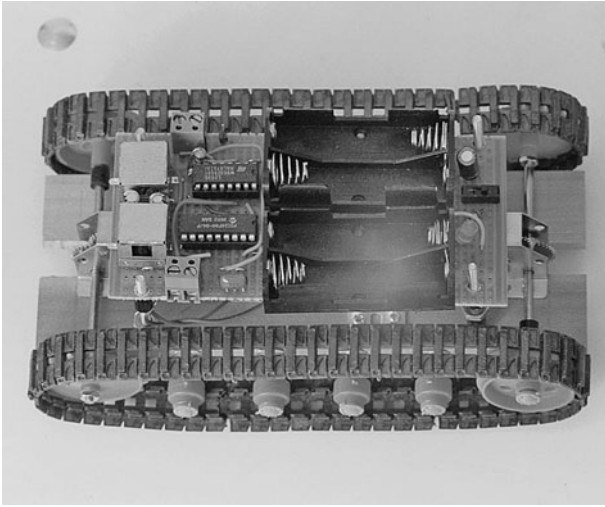


Figure 21.23 The completed IR-controlled tank.

differentially driven robot that could be controlled by a TV remote control. After looking through the article, I decided to build my own robot (Fig. 21.23) because there were a few things that I could improve on.

The project in the article used a single infrared (IR) receiver (typically used with TV remote controls) to control the robot. Two motors (for the robot's left and right sides) were switched on and off by an H bridge motor control made out of discrete transistors. The magazine robot used a PIC 16C5x for control. I felt that by using a mid-range part, I would have the advantage of interrupts to handle the incoming commands (which I'll refer to as *data packets*).

Most (if not all) IR TV remotes use a space-width encoding scheme in which the data bits are embedded in the packet by varying the lengths of certain data levels. This can be seen in Fig. 21.24 from a theoretical perspective and from the practical (oscilloscope output of a TV remote control IR receiver) in Fig. 21.25.

The normal signal coming from an IR receiver circuit is high when nothing is coming (line idle) and then goes low with a **leader** signal to indicate that data is coming in. The data consists of a bit **Synch**, whose value, when complete, is transmitted as the

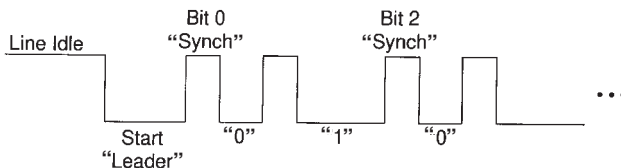


Figure 21.24 The output of a TV remote-control receiver consists of a number of pulses that are separated by different-length spaces that are used to encode data.

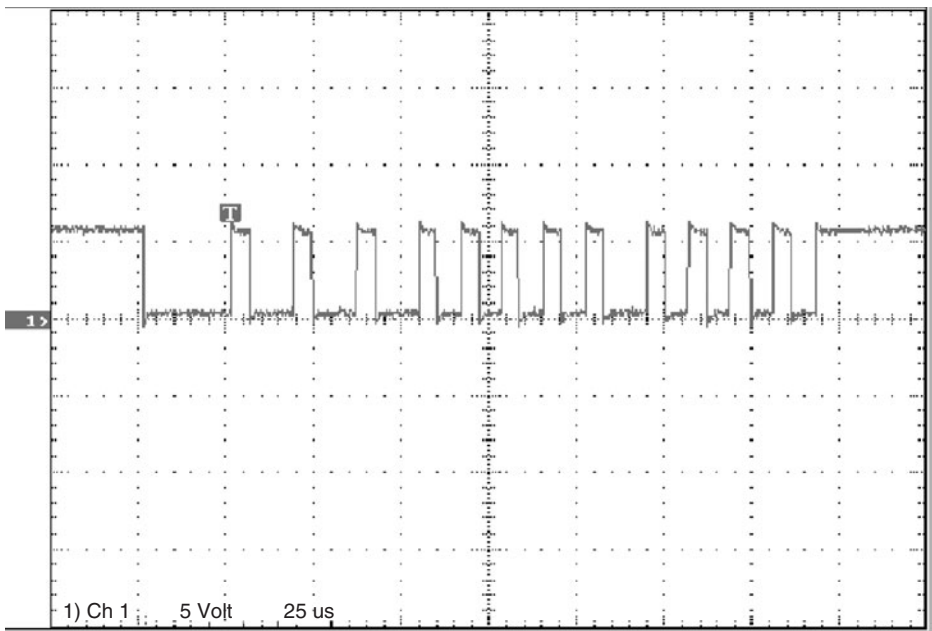


Figure 21.25 An oscilloscope view of the TV IR remote-control receiver output.

length of time before the next bit Synch. The IR robot is designed for Sony TV remotes, which have 12 data bits and a 40-kHz carrier. The timings are listed in Table 21.9 (and use a base timing T of 550 μ s).

To read the incoming signal, I used the following state machine code in an interrupt handler that differentiated between TMR0 overflows and PORTB change on interrupt:

```
interrupt IRTankInt()      // Handle Incoming IR Serial Data
{
    if (INTCON.T0IF != 0) {
        // TMR0 Overflow – Timeout Indicating Invalid
        // Data Being Received
    }
}
```

TABLE 21.9 SONY IR REMOTE-CONTROL TIMING		
FEATURE	T TIMING	ACTUAL LENGTH
Leader	4T	2.20 ms
Synch pulse	T	0.55 ms
0	T	0.55 ms
1	2T	1.10 ms

```
    BitCount = 0;           // Prepare for the Next Packet
    State = 0;

    INTCON = 1 << RBIE;    // Wait for Change on PORTB
Interrupt Request
                                // Also Reset "T0IF"
} else {                     // Else, Handle Incoming Bit Transition

    Temp = PORTB;          // Save the Changed State in PORTB to
Reset
                                // Interrupt Request
    INTCON = (1 << RBIE) + (1 << T0IE);
                                // Reset and Enable TMR0 and RB Interrupts
    switch(State) {         // Handle PORTB Change Based on Current
State
        case 0:            // Expect Leader/Synch Coming In

            TMR0 = 5msec;    // Leader Valid for 2.2 msec -
Put in Timeout

            State = State + 1;

            Value = 0;       // Haven't received anything yet

            break;

        case 1:            // End of Leader/Get "Synch"

            TMR0 = 1 msec;   // "High" Synch Pulse is Valid
for 0.55 msec

            BitCount = 12;   // Expect twelve Incoming Bits

            State = State + 1;

            break;

        case 2:            // Start of Synch Bit - Come back here for
                                // Each One
            TMR0 = 2 msec;   // "Low" Value can be valid for 1.10 msec

            State = State + 1;

            break;

        case 3:            // Bit Ended - Use TMR0 to Get Value

            Value = Value >> 1;
```

```

    If (TMR0 < 0.8 msec)
        Value.msb = 1;
                                // If Less than 0.80 msec has passed, then
                                // "1" was received

    BitCount = BitCount - 1;
    if (BitCount == 0) {
        DataFlag = 1          // Flag Mainline that Data is Available

        INTCON = 1 << RBIE;

    }
} // end Switch
} // end if
} // End IRTankInt

```

This code has the advantage of being able to discriminate between different manufacturers' IR codes (the differences lay in the length of time during the transitions, the number of bits, and the IR carrier frequency).

As I said earlier, one of the features I didn't like about the magazine article robot was its use of a single IR receiver. I was concerned that this would not provide adequate reception in a crowded room where the robot could be turned in different directions relative to the remote control.

For my robot, I decided to use two IR receivers pointing 180 degrees apart (this can be seen in the figure; the IR receivers are the two square metal cans at the front of the robot). This gives almost 360-degree coverage.

By doing this, a new problem came up, and that was the problem of arbitration. I found that one receiver may not pick up the transmitted signal or that it would be a 40-kHz cycle or two ahead or behind the other receiver. The arbitration scheme used is quite simple. The first receiver to transmit the leader to the PIC microcontroller is the one that is "listened" to for the remainder of the data packet.

The application code can be found in the `code\IRTank` folder. The application has been updated from the first edition's code to use a PIC16F84 instead of a PIC16C84. `Irtank.asm` was based on `Test9A` from the first edition. This was the last application in the series of programs written to understand how TV remote controls work and then build up the understanding into the final application. One thing that I am proud of in this application code is that when I created it, I had nothing like an oscilloscope to debug it. I was able to figure out the actual times by writing simple applications that mapped out signal timing, and I was able to differentiate the codes that were being sent. Once I had enough information to understand how the remote control worked, I was able to write the application.

The IR receiver the code was developed using a LiteOn 40-kHz IR remote control receiver module (Digi-Key Part Number LT1060-ND). This receiver was connected up to a PIC microcontroller that had a number of LEDs on PORTB/PORTA to try to decode what was happening. Since the original version of the robot was created, many different

manufacturers of 32-, 36-, 38-, and 40-kHz IR remote-control receivers have come on the scene from which you can choose.

Once I had the IR interface worked out, I thought I was away to the races. Unfortunately, this did not turn out to be the case. As a robot platform, I bought two Tamiya tracked vehicle parts kits (Item 70029). Each one has a single electric motor and gearbox—buying two gave me a motor and gearbox for each side's tracks. Running the motors, I found that they used about 250 mA, which was within the stated current ranges of the H drive used in the magazine robot's circuit. I then went off happily building the magazine circuit's H drive.

After hooking up the recommended circuit, nothing happened except the transistors got very hot (one actually exploded). This led to (literally) several weeks of asking questions and experimenting with different circuits (I even pulled out some of my old transistor textbooks—when I did that, I knew I was *really* desperate) until I determined that I had a circuit that wouldn't work for this application.

To make a long story short, I decided to see what others had done before me. Looking up robotics web sites (such as the Seattle Robotics Society), I discovered the L293D chip, which is a single integrated circuit that provides the H drive function for two motors (making this application much easier to wire than the original circuit that used eight discrete transistors and resistors). The L293D consists of four drivers (with clamping diodes) and is designed to be used as a two-motor H drive control. The circuit in Fig. 21.26 shows how one-half of the L293D can be used to control a motor.

Now, with everything working properly, I came up with the circuit shown in Fig. 21.27. The controller to be used is a universal remote control set to Sony TV. For the electronics, I used the parts listed in Table 21.10.

Four AA alkaline batteries provided power for my prototype. Using nickel-cadmium batteries, I found the robot to be very sluggish. Trying to figure out the cause of the

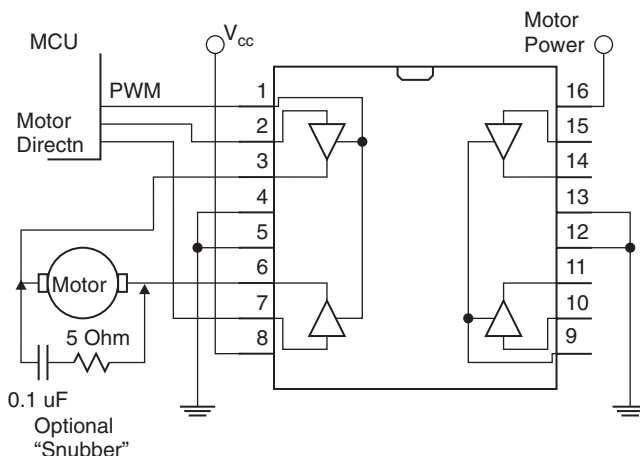


Figure 21.26 The L293D dual H-drive chip is an excellent part to use when you are starting out in robotics and want to have a reliable and efficient motor driver.

keep everything aligned. I've used this kit for a number of robots over the years with much better results (giving my kids something to terrorize our cats with).

IRBetter: ADDENDUM IR ROBOT

Sometime after I completed the IR remote-controlled robot (and the kids were well on their way to demolishing it), a question came up on the PICList concerning alternative methods of reading infrared remote control transmissions. The question was about sampling and learning the received data packet rather than comparing each incoming bit to a predetermined value. The question sparked the idea of representing the codes inside the PIC microcontroller in other ways.

My first attempt at this was to count the number of ones during a sample period. Knowing that a characteristic pattern of ones and zeros would be sampled, I felt that I should be able to get a reasonably precise value for the transmitted packet. This was the beginning of IRLCD_2 in the code/IRBetter folder. To test out the code, I created a breadboard circuit using the schematic in Fig. 21.28.

Note that this circuit uses the serial LCD interface described earlier in this chapter. Using the LCD really made debugging this experiment (I wouldn't go so far as to call it a project) a lot easier and is a good example of the usefulness of a simple UART interface.

The main body of the code, where the IR stream is read/sampled, is

```

movlw 0x0A0                ; Setup the Timer Interrupt
movwf INTCON

Loop:                       ; Loop Here for Each Update of Screen

movlw 200                  ; Wait for the Time Out
subwf IntCount, w
btfss STATUS, Z

```

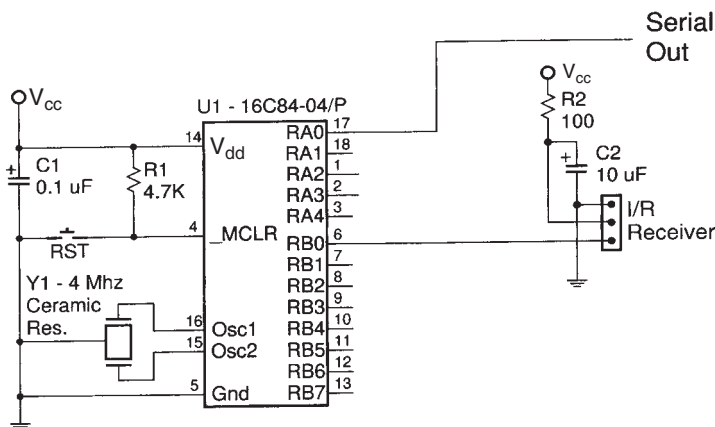


Figure 21.28 Test circuit for better operation of the PIC microcontroller IR receiver application code.

```

        goto Loop                ; Has NOT timed out

movlw 200                        ; Can we Display?
subwf ReadCount, w
btfsc STATUS, Z
    goto Loop_Reset            ; Reset the Count Values

movf ReadCount, w               ; Now, Display what was read in

clrf IntCount                   ; Clear the Display Values
clrf ReadCount

call DispHex                    ; Display the Hex Value

movlw 0x08E                     ; Reset the Cursor for Writing
call WriteINS

goto Loop                       ; Wait for the Next Loop Around

Loop_Reset:

clrf IntCount                   ; Reset the Values
clrf ReadCount

goto Loop

Int:                             ; Interrupt, Check I/R Input

movwf _w                        ; Save the Context Registers
swapf STATUS, w
movwf _status

bcf INTCON, T0IF                ; Clear the Timer Interrupt

incf IntCount                   ; Increment the Count Register

btfsc PORTB, 6                  ; Increment the Read Value?
    incf ReadCount

movlw 256 - 25                   ; Reset the Timer
movwf TMR0

swapf _status, w                ; Restore the Context Registers
movwf STATUS
swapf _w
swapf _w, w
retfie

```

This code simply counts the number of ones and stores the result in `ReadCount` for a given amount of time. The theory behind this method of sampling was that the dead space between packets would be read along with the data, and the result would combine them.

The actual value returned from the program wasn't very repeatable (as was expected). For example, five tries with the 1 key from a universal remote programmed with Sony codes produced these results:

```
0x09F
0x09D
0x08C
0x09D
0x09D
```

Generally, the results from this program were repeatable about 60 percent of the time. This might have been acceptable except for the poor discrimination that this method had. For example, the codes for 2 and 3 are 0x081 and 0x082, respectively. The problem lies in the fact that the two codes have the same number of ones and zeros. The code *may* pick up the differences, but I didn't find this to be the case.

Thus the code for reading the IR packet was changed to

```
    clrf  IntCount           ; Reset the Counters
    clrf  ReadCount

GetPack:                        ; Get the Next Packet Coming In

    movlw 0x088              ; Wait for Port Change Interrupt
    movwf INTCON

Loop:                          ; Loop Here for Each Update of Screen

    movlw 150                 ; Wait for 25 msec of Data from I/R
    subwf IntCount, w
    btfss STATUS, Z
    goto Loop                 ; Has NOT timed out

    clrf  INTCON              ; No more interrupts for a while

    movf  ReadCount, w        ; Get the Read in CRC

    clrf  IntCount            ; Reset for the Next Packet
    clrf  ReadCount

    call  DispHex             ; Now, Display the Character

    movlw 0x08E               ; Reset the Cursor
    call  WriteINS
```

```

        goto    GetPack                ; Wait for the Next I/R Packet

Int:                                ; Interrupt, Check I/R Input

        movwf  _w                    ; Save the Context Registers
        swapf  STATUS, w
        movwf  _status

        movlw  0x020                  ; Just wait for a Timer Interrupt
        movwf  INTCON

        movlw  256 - 20                ; Reset the Timer
        movwf  TMR0

        incf   IntCount                ; Increment the Count Register

        bcf    STATUS, C                ; Now, Figure out what to Add to
LSB
        btfsc  PORTB, 6                ; Is the Incoming Value Set?
        goto  Int_Set

        btfsc  ReadCount, 5            ; Do we Update the Value coming in?
        bsf    STATUS, C

        goto  Int_End

Int_Set:                            ; Incoming Set
        btfss  ReadCount, 5            ; Is the Current Bit Set?
        bsf    STATUS, C                ; No, Turn on the Incoming Bit

Int_End:

        rlf    ReadCount                ; Shift Over with New Input Data

        swapf  _status, w                ; Restore the Context Registers
        movwf  STATUS
        swapf  _w
        swapf  _w, w
        retfie

```

which can be found as IRLCD_3.

The fundamental changes were that the sampling started after the leader was received, and the 1s and 0s were treated as the inputs to a linear feedback shift register. Elsewhere in this book I have discussed how linear feedback shift registers work. For the preceding code, an 8-bit LFSR was used to produce cyclical redundancy check (CRC) codes. In this case, the input isn't the high bit of the shift register—instead, it is the input from the IR receiver.

Using this code, the CRC codes listed in Table 21.11 were generated from the Sony IR transmitter.

TABLE 21.11 IMPROVED IR TANK Read ALGORITHM RESULTS	
KEY	CRC CODE
Power	0x52
Vol+	0x5E
Vol-	0xBB
Ch+	0xDC
0	0x17
1	0x7A
2	0x8D
3	0x33
4	0x1F
5	0x4E
6	0x72
7	0xCC
8	0xB9
9	0x23

The interrupt handler code waits for a port change interrupt (the IR line going low from its nominal state of 1), and once that happens, the line is sampled every 200 μ s, and a CRC is generated from each sample. After 150 samples (30 ms), the CRC is output serially in hex format (i.e., sending the high nybble followed by the low one).

The CRC generated is rock solid (none of the 60 percent repeatability I had with just sampling bits). I don't know if I'm going to go back and update my IR robot code (lack of initiative more than anything else), but this is clearly a *much* more elegant and robust method of handling IR codes.

I did a limited amount of checking for invalid code rejection by reprogramming my universal remote with Panasonic and RCA codes. The CRCs generated were different from the Sony ones shown earlier.

I was never pleased with the XORing used to create the CRC in the preceding code. I felt that it was too confusing to understand. After some thought, I came up with the idea that if I used the same bit number for the PORT input bit as the CRC tap, I could simplify the CRC generator (from `bcf STATUS, C` to `rlf ReadCount` above) to

```
bcb STATUS, C
movf PORTB, w ; Get the Value Read in
xorwf ReadCount, w ; XOR it with the current
andlw 0x040 ; Clear all the bits but the two
```

```

btfss    STATUS, Z           ;   we're interested and if not = 0
bsf      STATUS, C           ;   then make LSB of the CRC = 1
rlf      ReadCount

```

This code only improves the original by two addresses, but it sure is a lot easier to understand. This is `IRLCD_4.asm` in the `IRBetter` folder.

When I implemented this change, I cheated a bit and changed the CRC tap (and not the line coming in, which would have meant that I would have to change the LCD code). But it still ran very well, with unique CRCs generated for each of the keys of the Sony-mimicking universal remote.

I guess the moral of this whole escapade is that tremendous improvements in your code (in terms of size and effort requirements) can be made if you look at a problem from a different angle. The code literally took less than 6 hours to develop and debug (compared with over 2 weeks for the original IR receiver code of the robot). The code takes up about a third of the space of the IR `read` algorithm used in the robot and uses only two 8-bit variables compared with the seven of the robot's code. This is a tremendous improvement!

Discussing this philosophically, it can be seen that this experiment actually restructures the application (reading an IR transmitter) to best fit the PIC microcontroller. The data read is now totally 8-bit, as opposed to the 12/16 bits that had to be handled in the original application.

Thermo: ELECTRONIC THERMOMETER WITH SEVEN SEGMENT LED DISPLAYS

One of the most popular PIC microcontroller projects I have ever created is this electronic thermometer. The basic model shown here has served me well for a number of years (Fig. 21.29), and I have even replicated it in *123 PIC Microcontroller Experiments for the Evil Genius* to demonstrate how the application could be implemented using HT Soft's PICC Lite. The application itself uses the resistor/capacitor network discussed elsewhere in this book for determining the resistance value of a thermistor. The temperature output is displayed on three seven-segment LEDs. The circuit is driven by a PIC16F84 using a 1-MHz crystal. There are a number of possible areas for inaccuracies in this circuit, so I have included the ability to set a calibration value, which is stored in the PIC16F84's data EEPROM, which is used to allow the temperature range to be set accurately.

The thermistor that I used was bought from Radio Shack, and while the part is widely available in North America, it cannot be ordered under the Radio Shack brand name elsewhere in the world. I have described the operation of the thermistor so that you can find the equivalent part numbers in your own location.

One comment/caveat for readers who want to use this project as a basis for other applications: This is not a precision instrument. Despite inclusion of the capability of calibrating the output temperature, you should not assume that this thermometer can be used for precision operations. I am putting in this warning because I know of at least two people who have used this circuit to control the temperature in their barns. While

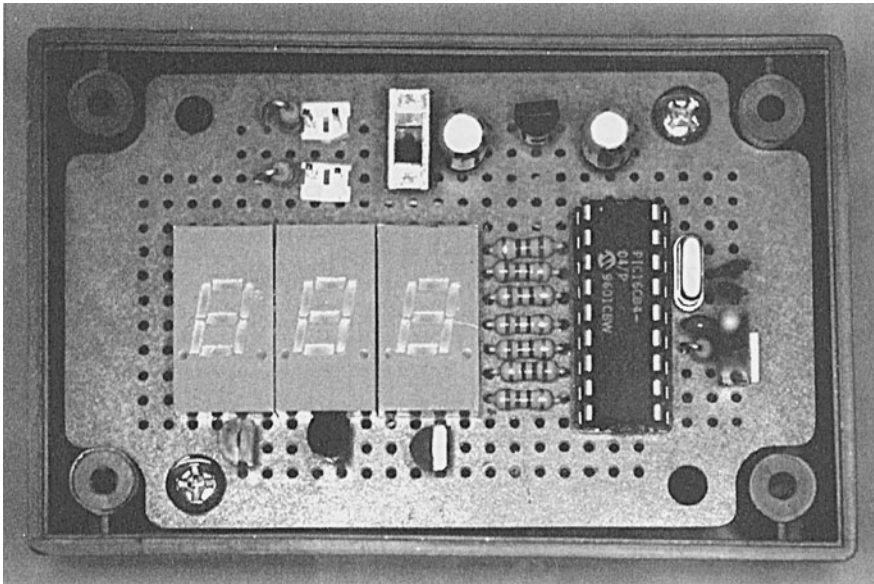


Figure 21.29 The completed digital thermometer.

I have not heard any negative feedback regarding the circuit's operation or accuracy. I should point out that if you are looking for accurate temperature readings, use the DS18B20 digital thermometer, which has an interface application presented later in this chapter. In terms of actual applications, this circuit apparently has been used as the basis for a thermostat in a chicken-hatching incubator without any ill-effects to the chicks. I suspect that this circuit is reasonably good around a small set range, although I would be very wary of the accuracy of results returned over a wide temperature range.

The code displays the current temperature in degrees Celsius. The thermistor was bought from Radio Shack (Part Number 271-110). This part is a 10-k thermistor with a negative temperature coefficient (NTC) of -3.85 percent. This means that for each degree Celsius the thermistor is raised, the resistance within the thermistor drops by 3.85 percent. The base temperature is 25°C , and the thermistor's response to different temperatures is listed in Table 21.12.

To create the thermometer, I created the circuit shown in Fig. 21.30. My prototype was built on a phenolic prototyping board that was bought as part of a prototype box. The bill of materials for this project is listed in Table 21.13.

The seven-segment LED displays used are of a common cathode type (which is to say the cathodes of all the LEDs within the display are connected to common pins) that takes up a 14-pin 0.300-in DIP pattern. The pinout of the conventional seven-segment LED displays is shown in Fig. 21.31. When I first created this application, I couldn't find the standard reference to the display, so I came up with my own standard (which is almost right). In Fig. 21.31 I have included a table to allow you to convert between my number convention and the universal letter convention. DP is for the display's decimal point.

TABLE 21.13 DIGITAL THERMOMETER BILL OF MATERIALS	
REFERENCE DESIGNATOR	DESCRIPTION
U1	PIC16F84-04/P
U2	78L05
Y1	1.00-MHz crystal
C1	10-μF electrolytic
C2, C5	0.1-μF tantalum
C3, C4	30 pF
R1, R12, R13	10 k Ω, 1/4 Wt
R2-R9	220 Ω, 1/4 W
R10	100 Ω, 1/4 W
R11	10 kΩ, -3.85% NTC thermistor (Radio Shack Part Number 271-110)
Q1-Q3	2N7000 N-channel MOSFET in TO-92 package
LED1-LED3	7-segment common cathode LED displays
J1	9-V battery connector
SW1, SW2	2 × 1 0.100-in pin headers
Misc.	Prototype PCB, experimenter's box, wire

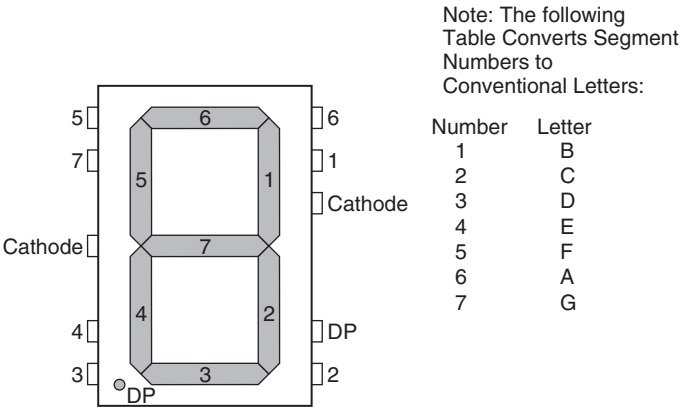


Figure 21.31 Pinout for a seven-segment common-cathode LED.

look at my circuit, notice that I included an SPST switch for turning on and off power to the circuit.

Each LED segment is connected to a PIC pin via a 220-Ω resistor, except for the segment connected to RA4, in which the 220-Ω resistor is attached to Vdd, and the PIC pin pulls it low to turn off the LED. This is to avoid the issue of RA4 not being able to drive positive signals and ensures that there is no possibility for high currents to be sunk by RA4. This was done to avoid both the PIC microcontroller being burned out and to minimize current consumption when RA4 is pulling the line low.

In my application, I wired the segments as

RA2 is connected to segment 6.

RA3 is connected to segment 5.

RA4 is connected to segment 1. Note the comments above.

RB0 is connected to segment 7.

RB1 is connected to segment 4.

RB2 is connected to segment 3.

RB3 is connected to segment 2.

The decimal point (DP) pin of the LED display was left unconnected.

The reason for using these values was strictly to make the wiring easier. If you look at Figure 21.29, you'll see that the seven current-limiting resistors are placed between the PIC and the LED displays. The seven-segment LED display connections use the seven I/O pins on the left side of the PIC microcontroller.

When creating an application such as this, you really have to plan ahead. In this application, the best example of this is wiring the seven left-side PIC microcontroller I/O pins directly to the current-limiting resistors and then to the seven-segment LED displays to avoid having to come up with very complex wiring. To help simplify the wiring even further, I also put the switching transistors along the bottom of the seven-segment LED displays to allow a common ground to be run to each one. Even with this planning, I still had to move around a couple of components and wire them on the backside of the board to get everything to work together. The final mess of wires is shown in Fig. 21.32.

A useful strategy to use in the application development process is to make sure that each subsystem in the application is tested before the final application is created. The source code for the Thermo application (`thermo.asm`) can be found in the `code\Thermo` folder.

The program uses the following formula

$$\text{Time} = R \times C \times -\ln(V_{\text{end}}/V_{\text{start}})$$

to measure the resistance of the thermistor. To initially calculate the actual temperature, I used idealized components and had the temperature looked up from a table (because the thermistor resistance versus temperature is a nonlinear function, and I didn't want

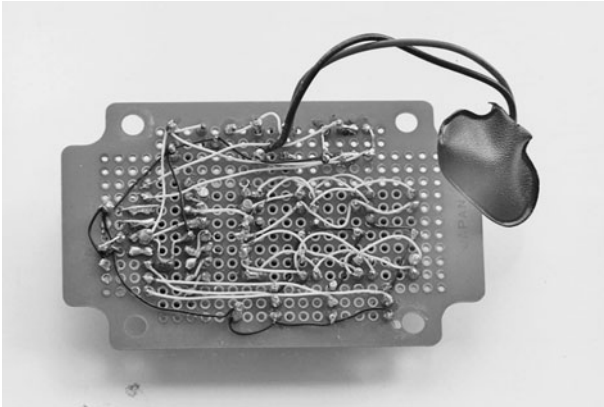


Figure 21.32 The backside of the digital thermometer's PCB.

to have to use an extensive mathematical formula). The table relates the resistance value read (where the time taken for the capacitor to discharge is proportional to the resistance) to an actual temperature. The table values relate back to idealized components (i.e., exact values).

In the actual application, I provide a constant value that will shift the actual value returned into an accurate temperature value. This constant changes the preceding formula to

$$\text{Time} = \text{constant} \times R \times C \times -\ln(V_{\text{end}}/V_{\text{start}})$$

Once the program starts to run, you will see that it displays a value for the current temperature or an error message `uuu` or `^^^` for too cold and too hot, respectively.

While the circuit is up and running, you may want to fool around with it a bit, such as putting the thermistor between your fingers and watching the temperature go up or putting it into a refrigerator/freezer and watching it go down. In doing this, you will discover that the thermistor-based thermometer will seem a lot faster than a mercury-based one. That is so because the thermistor has a lot smaller thermal mass than the mercury thermometer and can reach its environment's temperature faster.

As you look at the displayed temperature, you probably will notice two things. The first is that the temperature is probably wrong. That is so because you are using components that are *not* perfect (ideal)—their values are somewhat off the exact specified values. The second is that you probably will see the temperature creep up if the thermistor is close to the PIC microcontroller as the PIC microcontroller warms up from use. Both these problems can be overcome by setting the calibration constant in the PIC microcontroller.

Instead of working with a constant as part of the conversion formula, I could have added a potentiometer and then calibrated the circuit by changing the RC network's total resistance. The downside to this method is that the trim parts can drift themselves either by material breakdown over time or by the circuit being knocked around.

When the SW1 and SW2 connectors are shorted to ground, the PIC microcontroller pins to which they are connected will be pulled to ground and used to change the constant value. I put in a third-of-a-second delay to allow for debouncing and provide a delay between updating the value and disconnecting the connection to the switch.

A significant feature of the `thermo.asm` application code is how the data is stored in the EEPROM. On power-up, 2 data check bytes are checked for the values 0x0AA and 0x055. These data check bytes are used to indicate that the value in the data EEPROM byte is actually correct. On the first power-up of the PIC microcontroller with this program, these memory locations are invalid, and a separate set of code is executed to initialize the check-sum bytes and the data EEPROM bytes to an idealized constant. Using the check bytes along with value byte means that if you shut off the power to the PIC microcontroller and then when you come back later, the value will still be there and usable.

When I was adding the EEPROM code, I encountered the most significant problem in debugging the code. Every time I would run the application in hardware, only one digit would be displayed, and only the first digit of the EEPROM data check was written by the application. This was very confusing because when I simulated the application, I found that it seemed to be running correctly, but it wouldn't when put into real hardware. After much and protracted debugging, I found that the problem was with initialization of the segment variables. I had copied one line from the previous and not changed its value to the correct variable name. This ended up costing me about 3 weeks of part-time debugging to find the problem.

The problem with the simulating that I was doing was that I had put in a Debug define and had jumped over a 3-second setup delay. On the system that I was working with at the time (a 50-MHz 486), this simulation took over 45 minutes to execute. When I finally gave in and allowed the simulator to run through the full 45 minutes, the problem was obvious and easily fixed, the lesson being that I didn't make sure that my variable initialization was correct before I looked for other problems, and I forgot to be totally naive when looking at the problem and not expecting everything to be correct.

Once I had gotten the testing applications working, I combined them into the `prog32.asm` application that was shipped with the first edition. The calibration value itself is 16 bits long and is multiplied by the actual delay value. The high byte of the resulting 16-bit number is used as the corrected delay value. Using the high byte is the same as dividing the result by 256 (which is what the calibration value is based on). Doing the calculation this way eliminates the requirement to provide a division routine or floating-point routines as part of the calibration.

This method of implementing fractions is discussed elsewhere in this book, but I want to go through it again because it relates to the problem of providing a constant fraction value to this application. For example, if you wanted to find 30 percent of an 8-bit number, you could do this two different ways. The first is to multiply by 3 and divide by 10. To get 30 percent of 123, the operations would be

$$\begin{aligned} 30\% \text{ of } 123 &= (123 \times 3)/10 \\ &= 369/10 \\ &= 36 \text{ or } 37 \text{ (depending on rounding)} \end{aligned}$$

The problem with this method in the PIC microcontroller is that division cannot be implemented easily. Thus, instead of multiplying by a fraction, a constant value for a particular denominator is used. For the PIC microcontroller, a denominator of 256 or 65,536 (0x0100 or 0x010000) is probably the best way of doing this.

Using this method, instead of multiplying 123 by 3 and then dividing by 10, 123 can be multiplied by the fractional value of 256 and then divided by 256 to get the actual value. This is shown below:

$$\begin{aligned}
 30\% \text{ of } 123 &= [123 \times (30\% \text{ of } 256)]/0x0100 \\
 &= (123 \times 77)/0x0100 \\
 &= 9,471/0x0100 \\
 &= 0x024FF/0x0100 \\
 &= 0x024 \\
 &= 36
 \end{aligned}$$

The seven-segment LED driver code and hardware specified within this application can be used for a variety of purposes. Different character sets (i.e., hex codes) and additional and different displays can be implemented easily by modifying or cutting and pasting this code into another application.

MaryaToy: ADDENDUM TO THE ELECTRONIC THERMOMETER

After I built the LED thermometer, my daughter, who was 18 months old at the time, found it absolutely fascinating with its LED displays. In fact, I had a lot of problems trying to keep her from wanting to play with it. The obvious solution for me was to come up with a toy for her that had lights and buttons that would respond to her inputs (Fig. 21.33).

This also was a good chance for me to experiment with other types of LED displays. The display that I used is a 15-segment alphanumeric display. This display is very similar to the seven-segment displays of the electronic thermometer except that it has a lot more

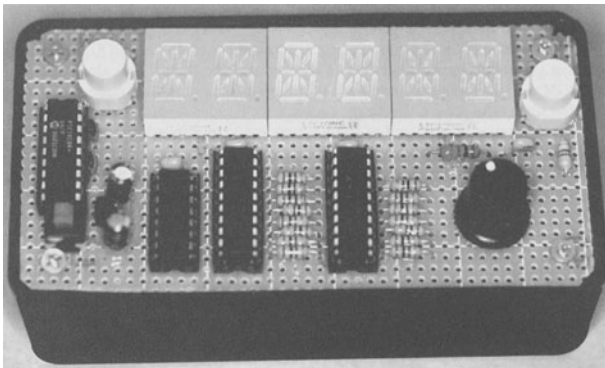


Figure 21.33 The child's toy created using a PIC16F84, some buffers, and six 15-segment LED displays.

segments, as you can see in the ASCII-Art drawing below. In this diagram, I have shown how I numbered each of the segments.

```

      ---1---
      |\  |  /|
      6 91011 2
      |  \ | /  |
      -7- -8-
      |  /|\  |
      5121314 3
      | /  |  \ |
      ---4--- Dot(15)

```

The most significant part of this project was figuring out how to drive these displays. Ideally, I wanted to use a simple PIC microcontroller (a PIC16C84 at the time).

Actually, this was quite easy to do because of the experience I got from doing a “Frosty the Snowman” display that was a precursor to the Christmas tree display I presented earlier in this chapter, along with the digital thermometer that was presented in the preceding section. The circuit I came up with (originally) is shown in Fig. 21.34. Six 15-digit displays were wired in parallel like the three seven-segment display of the digital thermometer, with each digit driven from an output of the 74LS374. For selecting which display is active, I used a 74S138 rather than a single transistor. In this way, up to eight displays can be handled without additional components.

Conceptually, the wiring of the display is very simple; in actuality, when you wire it, you’ll feel like you are going blind. I suggest that you buy a display with multiple digits that just have multiple common-cathode connections for each of the digits within the display. For my prototype, I used displays with two digits built in (and two common cathodes). The bill of materials for this project is listed in Table 21.14.

Despite essentially halving the amount of wiring, I still found that it took me a whole afternoon to point-to-point wire the displays to the current-limiting resistors connected to the 74LS374s. The displays that I used have their pins on the top and bottom of the packages, which makes daisy chaining the wiring quite difficult to do (I was able to do it in the digital thermometer relatively easily).

The mainline code simply updates a 6-byte array called `Disp` that consists of the ASCII codes (from 0x020 to 0x05F) that are currently displayed on the 15-segment LEDs. TMR0 was enabled along with its interrupt request, and each time it overflows (every 512 instruction cycles, or every 512 μ s because the PIC microcontroller is running with a 4-MHz clock), the character in each `Disp` element is output on its respective 15-segment LED display. The 512- μ s interval between digit displays gives an overall display frequency of 325 Hz, which is flicker-free and provides an acceptably bright output. I say that the display is *acceptably bright* because the output is essentially a PWM with the duty cycle being one-sixth or 16 percent of a total cycle.

I found that the interrupt handler required 162 instruction cycles of the 512 instruction cycles available. I am mentioning this because you should be aware of the 30 percent overhead that the display interrupt handler operation places on the PIC microcontroller’s execution. This turned out to be something to be aware of when I ported the code to PICBASIC in the next section.

9-Volt Battery Connector

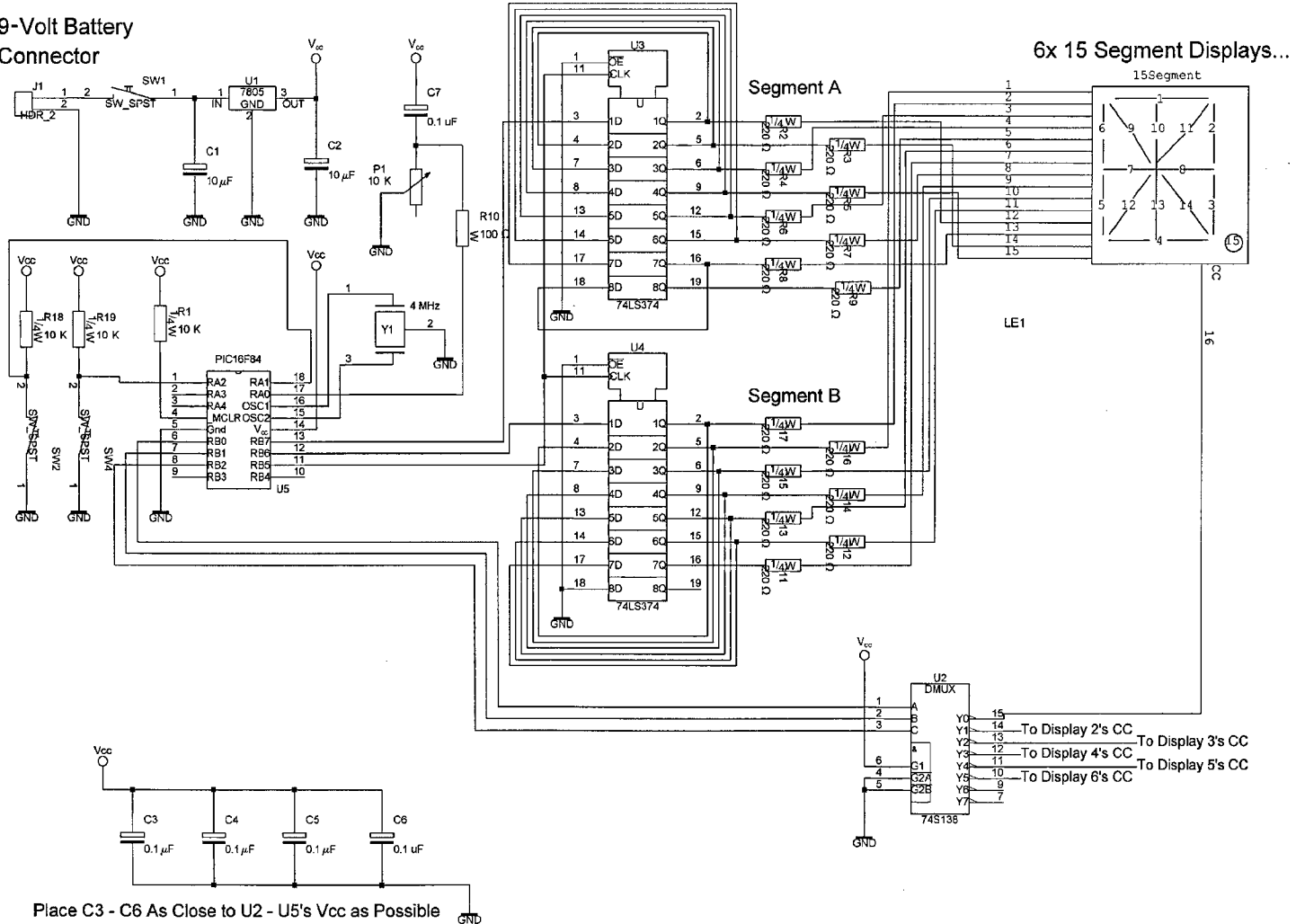


Figure 21.34 The schematic for the toy.

TABLE 21.14 MARYA'S TOY BILL OF MATERIALS

PART	DESCRIPTION
U1	7805 +5-V regulator
U2	74S138, 3 to 8 demultiplexor
U3, U4	74LS374
U5	PIC16F84-4/P
Y1	4-MHz ceramic resonator with internal capacitors
15-segment LED	6 × 15-segment alphanumeric LED displays
R1, R18, R19	10 k Ω , 1/4 W
R2–R9, R11–R17	220 Ω , 1/4 W
R10	100 Ω , 1/4 W
POT1	10-k Ω single-turn potentiometer
C1, C2	10- μ F electrolytic
C3–C7	0.1 μ F, any type
SW1	SPST power switch
SW2, SW4	Momentary-on pushbutton switches
J1	9-V alkaline battery connector
Misc.	Prototype PCB, project case, wiring

The source code is `Prog1.asm` (which can be found in the `code\MaryaToy` folder).

When I built the original, it was my plan to poll the buttons and read the potentiometer (as part of an RC network) to display the alphabet as well as numbers. I didn't do this because my daughter wasn't very interested in it—even though it had the letters “MARYA'S TOY” run across the LEDs. For some reason, she was always a lot more interested in the digital thermometer. The effort to add the additional functions didn't seem to be worthwhile until I started working on the second edition of this book and revisited this project, as I will discuss in the next section.

MaryaBas: PICBASIC PORT OF MARYA'S TOY

After updating the Marya's toy project for the second edition, I wanted to use it as a test bed to test out the capabilities of PICBASIC from the perspective of

- 1 Understanding how easy it is to develop complex applications in PICBASIC using built-in PIC microcontroller hardware features
- 2 Measure the performance of PICBASIC compared with straight assembly-language programming

- 3 Mix a reasonably complex interrupt handler written in assembly language with straight PICBASIC code
- 4 Evaluate the ease with which new features can be added to an application

My overall impressions of PICBASIC from this project are very favorable, but there are a few things that I learned that you should be aware of. I also want to say that this project turned out to be more than I bargained for in terms of both code development and debugging as well as difficulty. I would rate this application as being quite difficult to implement and, when it comes right down to it, quite advanced in terms of the skills I had to apply to get it working.

For this application, I wanted to use a PIC microcontroller with a built-in ADC rather than rely on the RC network as I originally proposed for this project; therefore, I used a PIC16C711 (which has a built-in ADC). I wanted to see how easily the ADC registers could be accessed from PICBASIC. I was happy to find that to use the PIC16C711, all I had to do was specify the processor in the compile statement, and the correct libraries would be specified and loaded automatically.

The original circuit was modified slightly with the RC network change to a simple potentiometer voltage divider and the button wired to RA1 relocated to RA3 because the PIC16C711 cannot just have RA0 as an analog input without RA1 also being an analog input as well. The updated circuit is shown in Fig. 21.35. The bill of material for the circuit is listed in Table 21.15.

The PICBASIC application source code and PICBASIC compiled assembly-language code (which can be run from MPLAB) can be found in the `code\maryabas` folder.

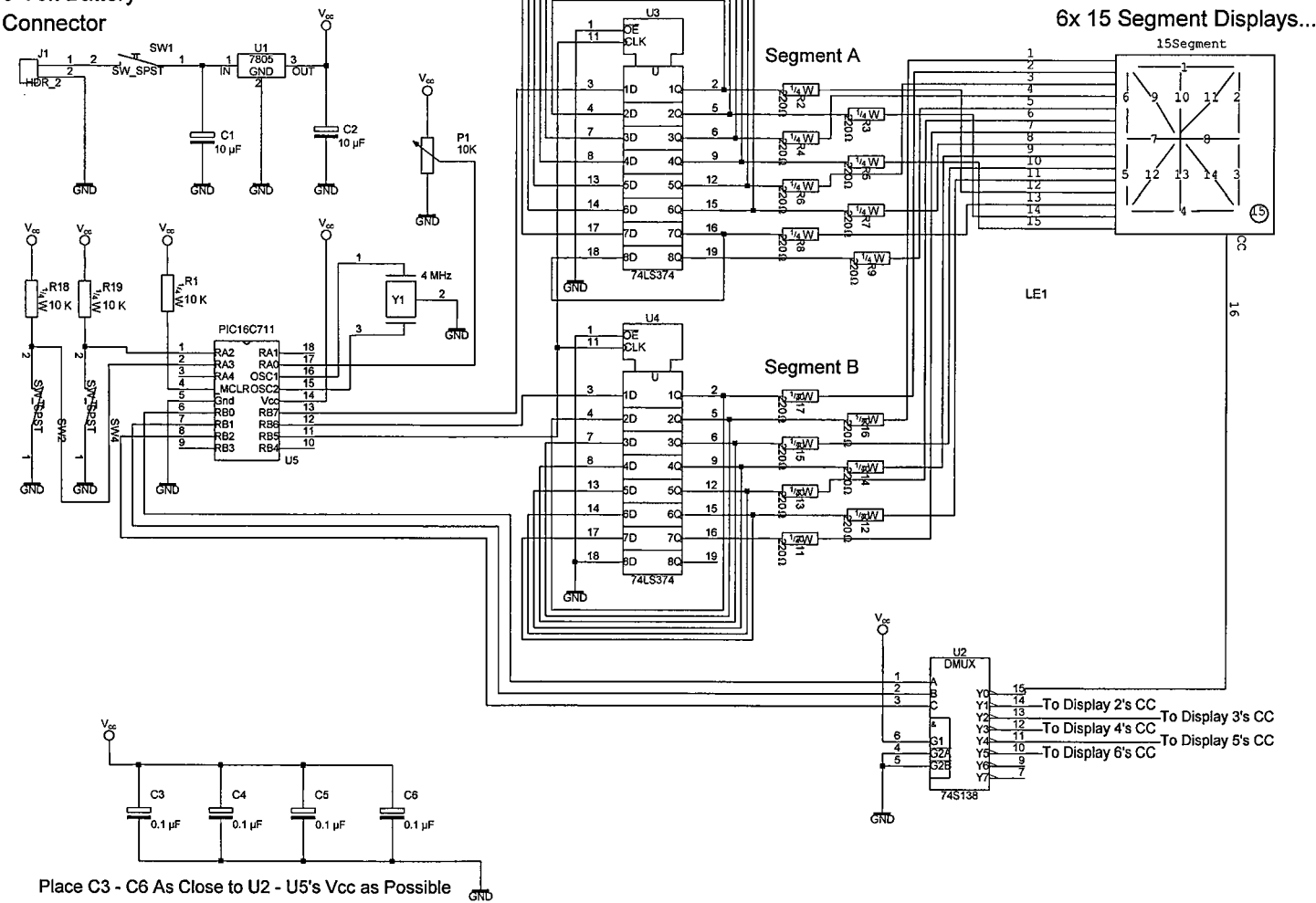
To create the application code, the first thing that I wanted to do was replicate the functions that I had in the original assembly-language application. This application simply displays a scrolling “MARYA’S TOY” string on the six 15-segment LED displays. The first part of the implementation plan that I had for this application was to copy in the interrupt handler from the previous project exactly and just use PICBASIC for creating the display information.

Along with leaving the interrupt handler in assembler, I also would include the tables used for storing the different ASCII characters. I wanted to keep the assembly-language interrupt handler as separate from the PICBASIC application as possible; the only interface would be the 6-byte `Disp` array, which contained the ASCII codes for each display. The PICBASIC mainline would provide the interface functions.

As it turned out, this was an excellent approach to implementing the application that eventually included providing the same initial scrolling display along with polling the two buttons and displaying either the alphabet or the numbers 0 through 9, with the potentiometer selecting the initial digit.

Implementing the initial capabilities is where I first ran into significant problems. It was my intention simply to use the interrupt handler code from the original project, put the PICBASIC handler prefix and postfix that I show in the Appendix E, and add some mainline code to have TMR0 driven from the instruction clock and enable the TMR0 interrupt request. I expected this to be a half-hour exercise, and it turned into 8 hours of frustrating debug.

9 Volt Battery
Connector



Place C3 - C6 As Close to U2 - U5's V_{cc} as Possible

Figure 21.35 The updated schematic for the MaryaBas application.

TABLE 21.15 BILL OF MATERIALS FOR THE UPDATED MARYA'S TOY

PART	DESCRIPTION
U1	7805 +5-V regulator
U2	74S138, 3 to 8 demultiplexor
U3, U4	74LS374
U5	PIC16C711-JW
Y1	4-MHz ceramic resonator with internal capacitors
15-segment LED	6 × 15-segment alphanumeric LED displays
R1, R18, R19	10 kΩ, 1/4 W
R2-R9, R11-R17	220 Ω, 1/4 W
R10	100 Ω, 1/4 W
POT1	10-kΩ single-turn potentiometer
C1, C2	10 μF Electrolytic
C3-C7	0.1 μF, any type
SW1	SPST power switch
SW2, SW4	Momentary-on pushbutton switches
J1	9-V alkaline battery connector
Misc.	Prototype PCB, project case, wiring

When I tried to compile this simple application, I found that I received **Error Line ###: Syntax Error and Bad Token “;”** error statements from the assembler. I went through a lot of gyrations of different formats for data until I tried deleting the tables to look at the absolute smallest problem (and if I could isolate it, send the code to meLabs for them to take a look at). When I deleted the tables, I found that the errors completely went away.

Thinking that I had put the data in the wrong format for the tables, I then went back and started adding the table code back in. I started by adding the table information a line at a time to see where the problem was. After about 10 lines, I started adding blocks of 16 table lines (instructions and comments) at a time, and I didn't get the errors until I had restored the tables to their original size.

Playing around with deleting individual lines, I found that if I deleted four lines from the total, the error messages would go away. Needless to say, this was pretty strange until I remembered a compiler that I worked with years before that had a limited amount of buffer/heap space. This compiler could process only so much source code, after which it started producing strange errors.

As an experiment, I restored the tables to their original size and deleted all the comments to the second table (for the second 74LS374). Amazingly enough, the error messages

disappeared. After a bit of analysis, I found that the PICBASIC version (2.21) could support only about 0x0A00 (or 2,560) bytes of assembler source code—anything above this amount caused a problem. This is something to remember if you are adding assembler code to your own PICBASIC applications.

This was not a trivial problem to find, characterize, and overcome. If you are new to programming, I doubt that you would have been able to find this problem and work out a solution to it. One of the recommendations that I would have from this is to avoid embedding assembly-language code in your PICBASIC applications unless you are comfortable with debugging assembly-language code and have some experience with programming in different environments. This is not a terrible restriction because, as I will discuss later, I found the compiled code produced by PICBASIC to be very efficient, with there being few reasons why you would want to use assembly language with PICBASIC.

I also found that my experiences with writing compilers for the PIC microcontroller were useful for seeing what the compiler produced. In this application, I did spend quite a bit of time looking at the instructions PICBASIC produces from the source code, and I can say that it is probably just as efficient as anything an experienced PIC microcontroller assembly-language programmer would come up with. There are some caveats to this statement, as I will present later when I discuss the PICBASIC application code.

With the PICBASIC version of Marya's Toy at the same level of operation as the assembly-language version (both were displaying the scrolling text string "MARYA'S TOY"), I wanted to take a look at the differences in the two applications. The most obvious thing to check was the number of instructions required for both applications. The assembly-language version uses 336 instructions, whereas the PICBASIC version uses 345—a difference of 2.7 percent. The obvious conclusion from this is that PICBASIC is very efficient in developing its assembler instructions. As I indicated earlier, when I simulated the assembly-language output, I found the code to be quite logical and efficient, which bears out the conclusion reached by the comparison of the number of instructions.

The only execution difference that I noticed between the two versions is that in the PICBASIC version, the scrolling was somewhat slower. When I looked at the reason for this, I explicitly put in a 125-ms delay between character scrolls in the PICBASIC source code. This delay is "stretched" out by the 30 percent overhead of the 15-segment LED display interrupt handler overhead. When I wrote the assembly-language code and was debugging it, I decreased the mainline delay so that the characters would scroll at a rate of one character every 125 ms even with the interrupt handler operating.

With the scrolling speed increased, I wanted to go on and add polling of the two buttons, and when one of the buttons is pressed, another display (either the alphabet or the first 10 numbers) is displayed, and its position could be specified by turning the potentiometer.

When I originally implemented the button polling, I used the PICBASIC `button` function—and specified the information as if it were Parallax Stamp PBASIC. In doing this, I found that the operation of the `button` function is actually quite a bit different

between the two languages. In PICBASIC, `button` polls the specified button, and if it is in the target state, it performs a debounce poll of the `button` for 10 ms before jumping to the destination. The Stamp `button` function is designed to poll the button in a loop around and increment a count each time the button is down (and clear it if it is up).

For the application, I really wanted to implement a 1-ms loop in which each of the two buttons would be polled. To do this, I ended up going away from the PICBASIC `button` function and ended up writing my own Stamp `button` equivalent function:

```
if PORTA.2 = 0 then          ' Debounce Button2
    Button1 = Button1 + 1    ' Count Number of Times Down
else
    Button1 = 0              ' If Up, Reset
endif
if Button1 >= 15 then Destination
```

In my code, each time the button is polled (which happens once every 1.5 ms nominally), if it is down (the bit equal to 0), I increment a counter. If it is high, then the counter is reset. When the counter reaches a preset value (15 for the preceding example code), the button is determined to be reset, and execution jumps to the `Destination` label.

In this application, after one of the buttons is pressed, the application jumps to either an alphabet or a number display that uses the potentiometer to specify what portion of the string is to be displayed. The code I came up with for the Alphabet display is quite elegant and should be discussed:

```
Alphabet:                    ' Put in the Alphabet

    LoopCount = 0
    Button2 = 0              ' Alphabet is From Button1

Loop_2:

    ADCON0.2 = 1             ' Start ADC Operation

    pause 1                  ' delay 1 msec

    if PORTA.3 = 0 then      ' Debounce Button2
        Button2 = Button2 + 1
    else
        Button2 = 0          ' If Up, Reset
    endif
    if Button2 >= 15 then Numbers

    OutPos = ADRES           ' Read the Results of the ADC

    OutPos = OutPos */ 21    ' Get the Scaled Value

    for i = 0 to 5           ' Update the Alphabet
```

```

        lookup OutPos, ["ABCDEFGHIJKLMNOPQRSTUVWXYZ"], j
        ,
        01234567890123456789012345
        Disp[i] = j          ' Get the Next Character
        OutPos = OutPos + 1   ' Increment to the Next Character
    next i

    goto Loop_2

```

Before starting the button polling loop (which executes once per second), I set the GO/_DONE bit of the ADCON0 register of the PIC16C711. This starts the analog-to-digital conversion on the specified pin (RA0 in this case). After the 1-ms delay (the `pause 1` statement), the other button is polled in my debounce routine, and execution jumps to the different display routine if more than 20 ms has passed with the switch not changing. Note that I do not check both switches, just the one for the other function. This was done to avoid the need to have to wait for the pressed button to be debounced high—instead, it is assumed that only one button is pressed at any time.

If the button has not been pressed requiring the jump to the other display routine, I read the potentiometer's position from the ADC and scale it for the display. The scaling statement

```
OutPos = OutPos */ 21
```

probably requires some explanation.

When displaying the alphabet or numbers in this application, I wanted to make sure that all six displays were in use at all times. This means that the last five positions of each string cannot be used as the starting position of the display. For the alphabet, instead of being able to start with 26 different letters, this application can start with only 21—with the last 5 being in the other displays.

If you were to scale a value between 0 and 21 in real life, you would use a calculator and divide by the fraction of 21 over 256. The result's fractional value would be taken off the result, and you would be left with an integer between 0 and 20 (just what you want). Put mathematically, the formula would be

```
OutPos = int(OutPos * 21 / 256)
```

For example, a value of 142 from the ADC would become

```

OutPos = int(OutPos * 21 / 256)
        = int(142 * 21 / 256)
        = int(2,982 / 256)
        = int(11.648)
        = 11

```

The result is correct, but the process taken to get there requires real numbers, which the PIC microcontroller simply does not work with very well. This operation could be done in PICBASIC, but it would be terribly expensive in terms of the number of instructions to complete the task.

What is required, as I've shown in other parts of this book, is a better way to calculate fractions. The best way that I have found is by multiplying by the maximum value of the new range and discarding the lower 8 bits of the product. Put mathematically, this is

```
OutPos = (OutPos * 21) >> 8
```

and can be demonstrated using the previous example as

```
OutPos = (OutPos * 21) >> 8
        = (142 * 21) >> 8
        = 2,982 >> 8
        = 0xBA6 >> 8
        = 0xB
        = 11
```

Multiplication by a constant is very easy in the PIC microcontroller, and shifting a value to the right by 8 bits is accomplished simply by discarding the lower 8 bits. Elsewhere in this book I show how this formula can be implemented very easily in PIC microcontroller assembler.

PicBasic has an operator that makes this operation even simpler than the preceding formula, and that is the `*/` operator, which multiplies two numbers together and lops off the least significant byte (8 bits) exactly in the manner in which we require for this scaling operation.

The reason why I am showing you this is to get you used to the idea of looking for tricks to make your applications more efficient and avoid trying to implement applications using the same techniques as you would have used for your high school math homework.

Going through the `calculator` process above will take away any chances for PICBASIC to produce code that is just as efficient as assembler and will end up being a lot more work for you. If you are going to use a high level language such as PICBASIC, make sure that you understand it reasonably thoroughly and that you can take advantage of special instructions such as the one above.

The other point in using PICBASIC (and the `Alphabet` code above) is to limit the number of `lookup` and `lookdown` statements in your application. This is not to say that `lookup` and `lookdown` are not implemented efficiently—they are actually quite efficiently implemented with table operations in PICBASIC. It is just that they are so useful when programming that you will want to use a lot of them. A lot of them will use a lot of space in the PIC microcontroller where there isn't a lot to begin with.

In the `Alphabet` code above, you might have thought it would be better to do something like

```
Alphabet:                                     ' Put in the Alphabet

for i = 0 to 5                               ' Update the Alphabet
    lookup i, ["ABCDEFGHIIJKLMNOPQRSTUVWXYZ"], j
```



```

'          01234567890123456789012345
  Disp[i] = j          ' Get the Next Character
next i

LoopCount = 0
Button2 = 0          ' Alphabet is From Button1

Loop_2:

  ADCON0.2 = 1          ' Start ADC Operation

  pause 1          ' delay 1 msec

  if PORTA.3 = 0 then    ' Debounce Button2
    Button2 = Button2 + 1
  else
    Button2 = 0          ' If Up, Reset
  endif
  if Button2 >= 15 then Numbers

  OutPos = ADRES          ' Read the Results of the ADC

  OutPos = OutPos */ 21    ' Get the Scaled Value

  for i = 0 to 5          ' Update the Alphabet
    lookup OutPos, ["ABCDEFGHIJKLMNOPQRSTUVWXYZ"], j
    '          01234567890123456789012345
    Disp[i] = j          ' Get the Next Character
    OutPos = OutPos + 1    ' Increment to the Next Character
  next i

  goto Loop_2

```

where the display is given an initial alphabet value before the ADC is read. The problem with this code is that it doubles the amount of space required for the lookup table. While in itself one lookup is not a lot (about 35 instructions), and it makes implementing a table very easy, two lookups take up more space: 70 instructions, or 7 percent of the total available in the PIC microcontroller.

Along with using up more space than is necessary in the PIC microcontroller's program memory, the first lookup is probably not desired because the correct location for the alphabet display is not known, and the display will flash with the arbitrary string that is sent to the 15-segment LED displays. Even though the flash is only for 1 ms, it probably will be picked up by a user and will not be that attractive—the code used will set the correct starting point for the display immediately following the press of the button from the previous display.

In case you are wondering, the comment line that follows the lookup table, that is, 01234567890123456789012345, is just my way of keeping track of all the offsets

for the different characters within the lookup table and making sure that I do not forget any table elements. It is a good idea for you to add little tools like this to make sure that everything is correct and in place and to provide an easy visual way of checking actual to expected values.

I realize that I have gone on *ad nauseum* about application efficiency because in a high level language such as PICBASIC this is where taking the time to look at the best way of doing things is going to pay off the most. When using tables in a PIC microcontroller assembler, the tables can be accessed multiple times, but in the PICBASIC compiler, each lookup, even if it is identical with another one, will produce code for a unique table. Many other examples of this can be found and avoided, making the compiled PICBASIC code just as efficient as well-written PIC microcontroller assembler if you look at the situation and see if there are other ways to implement the function.

In sum, I want to make a few comments about PICBASIC. The first is that it is a very efficient compiler in terms of code size and execution speed. I would not hesitate to recommend it for new PIC microcontroller users. PICBASIC has extremely rich functions and capabilities that will allow you to develop very sophisticated applications quickly and efficiently. Internal PIC microcontroller features and registers can be accessed in line without regard to banks or bit values. In many ways, PICBASIC is an optimal blending of a high level language with assembly language's ease of accessing specific registers and bits. The biggest problem I found with PICBASIC is the difficulty in which code can be simulated (and, by extension, emulated). As I have been working through this third edition, meLabs has created a plug-in for MPLAB IDE that you should ensure that you use to be able to take advantage of the source code debugging (both in the simulator and in the MPLAB ICD 2 debugger).

PCTherm: RS-485 MASTER/SLAVE REMOTE THERMOMETER INTERFACE

I debated whether or not to include this project in the third edition of this book because it uses a PC ISA adapter, and in the years since the second edition was printed, it has become essentially impossible to find PCs with this basic bus, and the current versions of operating systems do not support memory-mapped I/O operations (either in address or register spaces). This means that unless you have an older PC (built before 1997 or so), you won't be able to implement this project as is. You can, however use the circuit with other processors in which a relatively low speed parallel bus is available and you can access devices on it. When I did the project, I also spent quite a bit of time creating the bidirectional RS-485 communications methodology used here that you probably will want to take advantage of. In the end, I decided to leave this project in so that you could review and understand how RS-485 communications can be implemented practically and see a real-world example of how the PIC microcontroller parallel slave port (PSP) is used as well as discuss some of the issues I had to overcome.

As I will describe, this application consists of three quite simple pieces that work together to provide a remote temperature sensor using the PC ISA adapter shown in Fig. 21.36 and the basic circuit in Fig. 21.37. A Windows control program was written in Visual Basic 6.0 and interfaces to the ISA card that has a PIC16F877 with its parallel

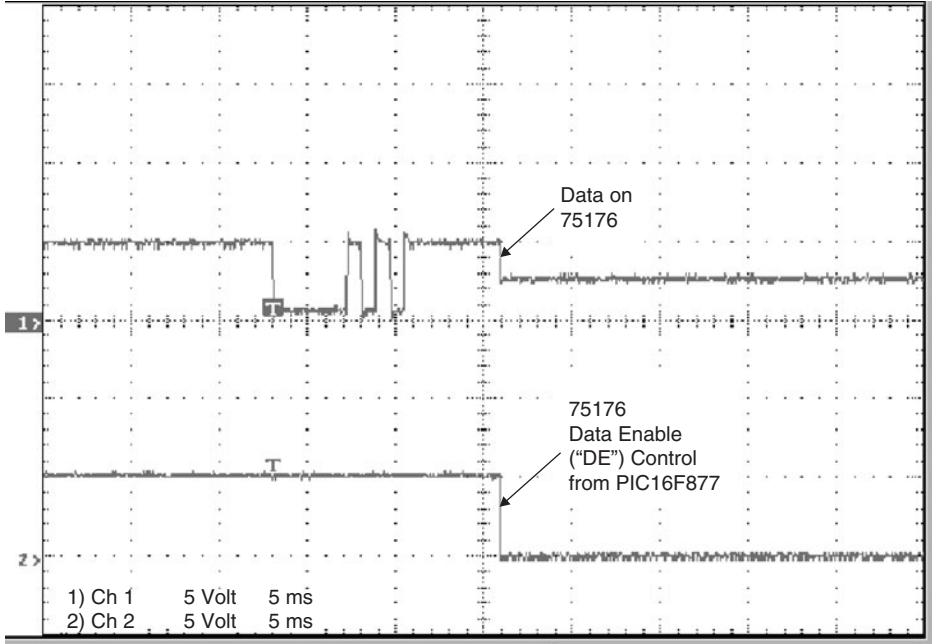


Figure 21.36 The PC ISA adapter card contains the PIC microcontroller with the parallel slave port (PSP) and RS-485 interface.

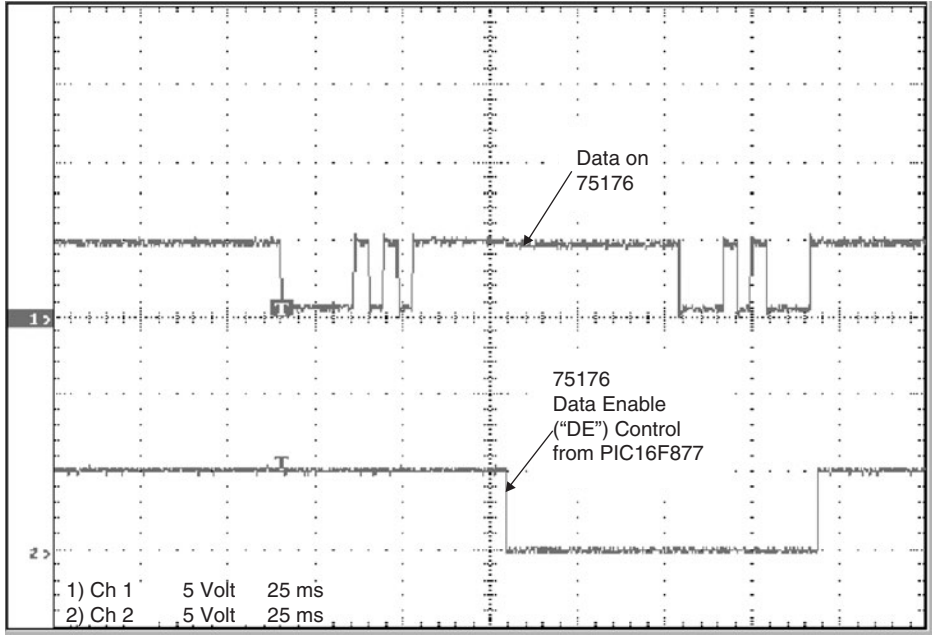


Figure 21.37 The external sensor is a simple circuit powered by 12 V coming from the RS-485 connection with the DS1820 temperature sensor on the far left.

slave port (PSP) enabled. This PIC16F877 uses its USART to communicate to a remote PIC16HV540 that interfaces with a Dallas Semiconductor DS1820 temperature sensor via RS-485. This entire application uses less than 620 lines of code (including comments). The application works very well and could be used as the basis of a home temperature-sensing system.

The first card that I created was the ISA card that the PIC16F877 resided on. In Fig. 21.38 you can see that the circuit is quite simple to implement with only five chips. The circuit itself uses two 74LS85 value comparators and a 74LS138 for address selection. The RS-485 interface is implemented with a 75176, and the PC's +12-V power supply is passed to the remote sensor for its power. In the figure you can see that there are other (unpopulated) sockets on the board; this ISA prototype board was taken from previous projects simply because it is quite expensive, and I wanted to reuse it rather than spend \$100 on a new one. The bill of materials for this circuit is listed in Table 21.16.

There are a number of aspects of this circuit that will seem unusual. I use the PIC microcontroller itself for driving the address that is compared to when writes to the PSP port take place. The circuit was originally an ISA prototype card that I had lying around from previous projects, and rather than rewire the circuit, I used the circuit already on the prototype card. This circuit uses the PIC microcontroller to output the most significant 7 bits of the PC's I/O port address. The lower 3 bits are decoded and selected by the 74LS138. This method of decoding the PC's I/O port addresses allows me to select different addresses according to my own requirements. For this application, the ISA card presented here was the only optional card, so I used address 0x0300 (the first prototype adapter block) for this application.

Note that in this circuit the received data driver control ($\overline{\text{RE}}$) of the 75176 is always pulled low so that the driver is active. I did this so that the PIC microcontroller's RX pin (RC7) is always being driven. The LED was added to provide a visual indicator of when the ISA card received valid data.

When I originally built this card, I used an old 80386 processed PC that I had bought for \$30. This PC had I/O slots that were a "standard" three-quarters of an inch apart. For this project, I used a different PC that only had half an inch between slots. This meant that I had to reposition some cards in the system as well as cut down the ends of the wire-wrap pins. If you have to do this, please be careful and keep track of all the cut pins and clean them off the board—you do not want any of them falling into the PC and shorting out anything on the PC's motherboard.

The ISA card's PIC16F877 source code is `PCTherm2.asm` and can be found in the `code\pctherm` folder. This project does not have to use the PIC16F877; any 40-pin mid-range PIC microcontroller will work (because they all have built-in USARTs and PSPs).

To communicate with this card, I wanted to use a PIC16HV540 ("high voltage") PIC microcontroller to take the 12-V power from the PC and convert it to 5 V for the remote application. In doing this, I wanted the PIC16HV540 to power the 75176/RS-485 interface and the DS1820 digital thermometer. When I started working with this circuit, I discovered that the PIC16HV540 did not have the current drive capabilities required

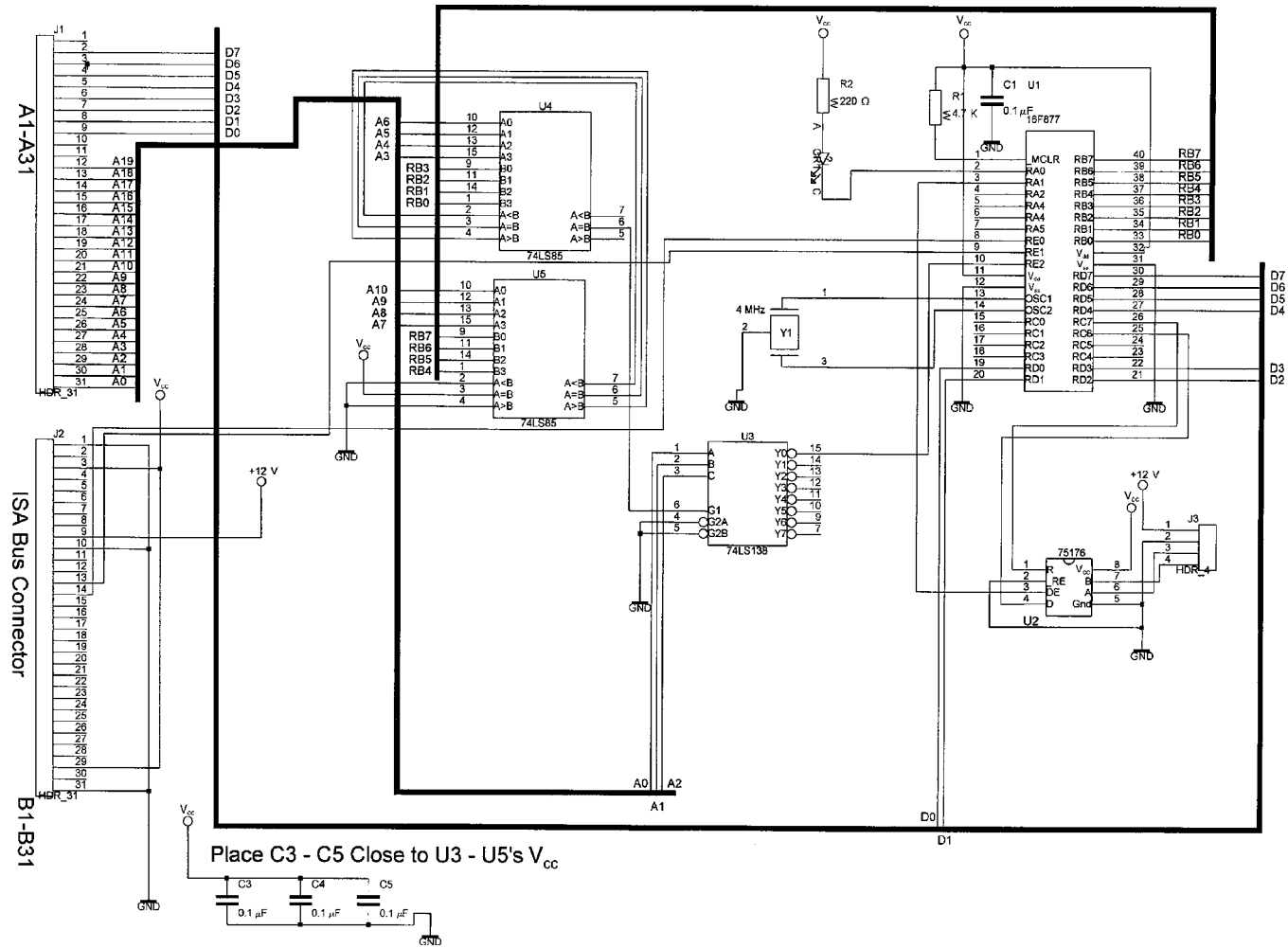


Figure 21.38 PC ISA adapter circuitry.

TABLE 21.16 PC THERMOMETER ISA CARD BILL OF MATERIALS	
REFERENCE DESIGNATOR	DESCRIPTION
U1	PIC16F877-04/P
U2	75176 RS-422/RS-485 interface
U3	74LA138 8 to 1 demultiplexor
U4, U5	74LS85 value comparators
CR1	Red LED
Y1	4-MHz ceramic resonator with internal capacitors
R1	4.7 k Ω , 1/4 W
C1	0.1 μ F tantalum
C3-C5	0.1 μ F any type
J3	4 $\times$ 1 screw terminal
Misc.	ISA prototyping PCB, wire-wrap sockets, write-wrap wire, telephone cable

for the 75176 (especially when it was driving the line low when the ISA card's 75176 was driving it high). The solution to this problem was to add a 7805 to provide power for it and the DS1820. The final circuit is shown in Fig. 21.39 and was built using point-to-point wiring with the bill of materials shown in Table 21.17.

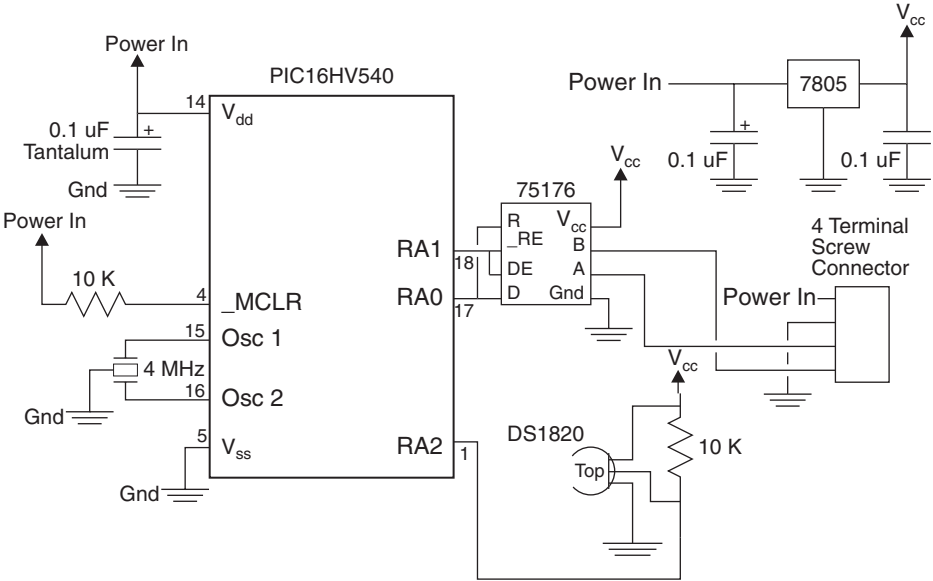


Figure 21.39 The PIC16HC650 microcontroller-based remote thermometer circuit.

TABLE 21.17 PC THERMOMETER REMOTE PIC MICROCONTROLLER CARD BILL OF MATERIALS

REFERENCE DESIGNATOR	DESCRIPTION
PIC16HV540	PIC16HV540-JW
7805	7805 +5-V regulator
75176	75176/RS-422/RS-485 interface
DS1820	DS1820 TO-92 electronic thermometer
4 MHz	4 MHz with internal capacitors
0.1	0.1 μ F tantalum capacitor
10k	10 k Ω , 1/4 W resistor
Connector	Four-terminal screw connector
Misc.	Prototype card, sockets, wire-wrap wire

The PIC16HV540 source code (`RemPIC.asm`) can be found in `code\PCTherm` folder. The code can be converted to any other 18-pin PIC microcontroller without modification.

You also might want to use a PIC16F505 and have it powered from the 7805. The problem with the PIC microcontroller I/O pins being unable to supply power to the different components of the circuit is one that I suspect you will be encountering more and more as more sophisticated chips become available with lower current output I/O pins, and you may find yourself in a situation where you have to go back to earlier designed parts to avoid the need for adding additional power supplies or drivers.

The reason for using the +12-V power from the PC to the remote card was to avoid very long wire voltage drops that would cause problems with the voltage from the PC being high enough to run the PIC microcontroller and the other parts on the card reliably. Using 100 feet of four-conductor telephone cable, I found a 0.35-V drop across the line.

For debugging the application before it was installed in the PC, I connected the ISA card to a bench +5-V power supply and the remote card to a +12-V power supply and connected them with two conductors of the four-conductor telephone cable. Note in both Figs. 21.38 and 21.39 that I have put the two 4×1 screw terminals in the same orientation. This was to avoid issues with keeping track of the wiring. The convention I used for the wiring is listed in Table 21.18.

TABLE 21.18 REMOTE THERMOMETER WIRE CONNECTIONS

WIRE COLOR	FUNCTION
Red	+12 V
Black	Ground
Yellow	A pin (positive RS-485 voltage) of 75176
Green	B pin (negative RS-485 voltage) of 75176

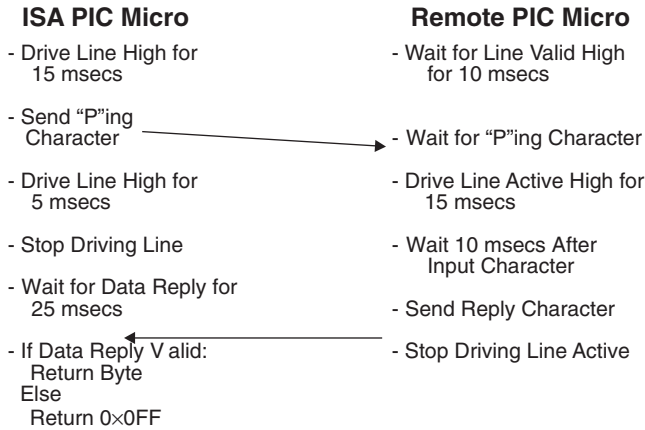


Figure 21.40 The data-flow diagram for the remote thermometer.

Note that there is no physical difference between this cable and a real telephone connected in your home, so if you are going to install this cable alongside telephone cable, make sure that you keep track of which cable is which. Note that plugging phones into this cable or a PC could damage either device (which could be *very* expensive if phone company equipment is damaged).

For the data communication, I used 1,200-bps NRZ serial communications with some special 75176 control timings to ensure that the two PIC microcontrollers do not get into contention with one another. The communication's flow for the two PIC microcontrollers' is shown in Fig. 21.40, and note that when the communication is working correctly, neither device will be pulling the RS-485 line low. When I introduced RS-422 and RS-485, I did note that multiple devices could drive at the same time, but I found that in this application, owing to the method of providing power to the remote PIC microcontroller, I could not get enough current to reliably pull the data line low.

This method of communication works quite well, as you can see in the oscilloscope pictures shown in Figs. 21.41 and 21.42. The top line in both diagrams shows the actual RS-485 positive-voltage signals. Figure 21.41 shows the signals the PIC16F877 master works with, and you can see the 75176 driver being turned off (the lower line) when the temperature data from the remote PIC microcontroller is expected. During this time, the line remains driven, and valid data is received by the PIC16F877.

In Fig. 21.42, the incoming ping character (P or ASCII 0x50) can be seen on the 75176 data line, after which it drives the line, waits 15 ms, and drives the current temperature back on the line. When the transmission has been completed, the remote PIC16HV540 stops driving the RS-485 line (at which point the PIC16F877 resumes driving the line high).

After the remote PIC microcontroller has responded to the PIC16F877, it polls the DS1820 for the current temperature. This temperature is transmitted back to the PIC16F877 the next time the ping character is received. The PIC16F877 polls the remote device once every second, so there is plenty of time for reading the current temperature from the DS1820.

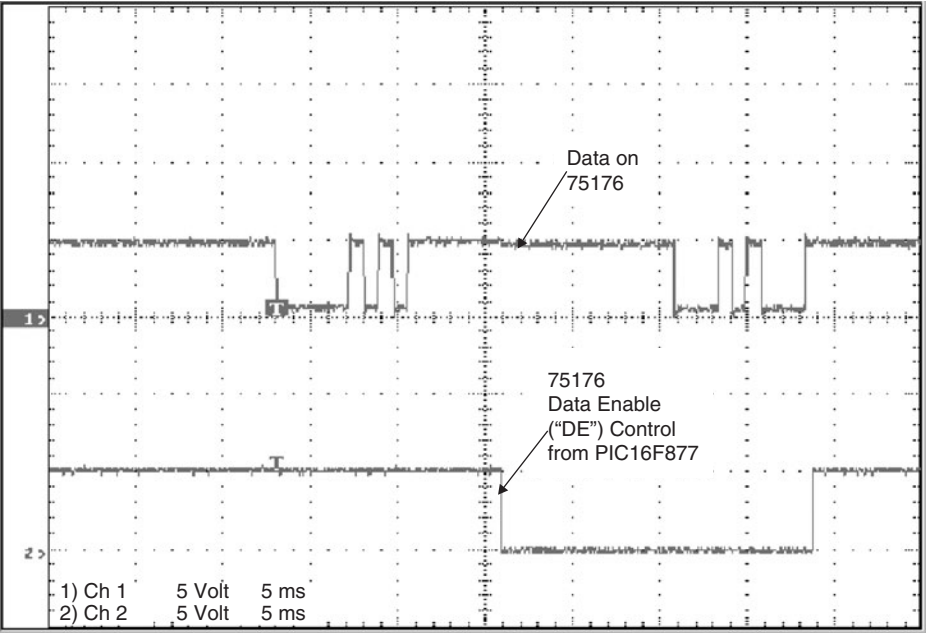


Figure 21.41 Remote thermometer ISA command to remote thermometer circuit.

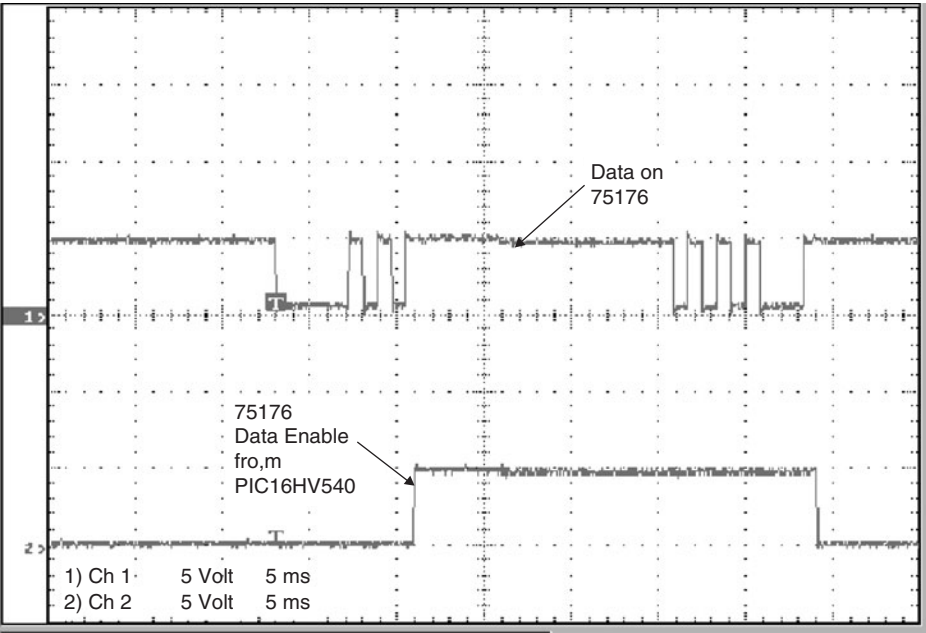


Figure 21.42 ISA command to remote thermometer circuit and temperature returned.

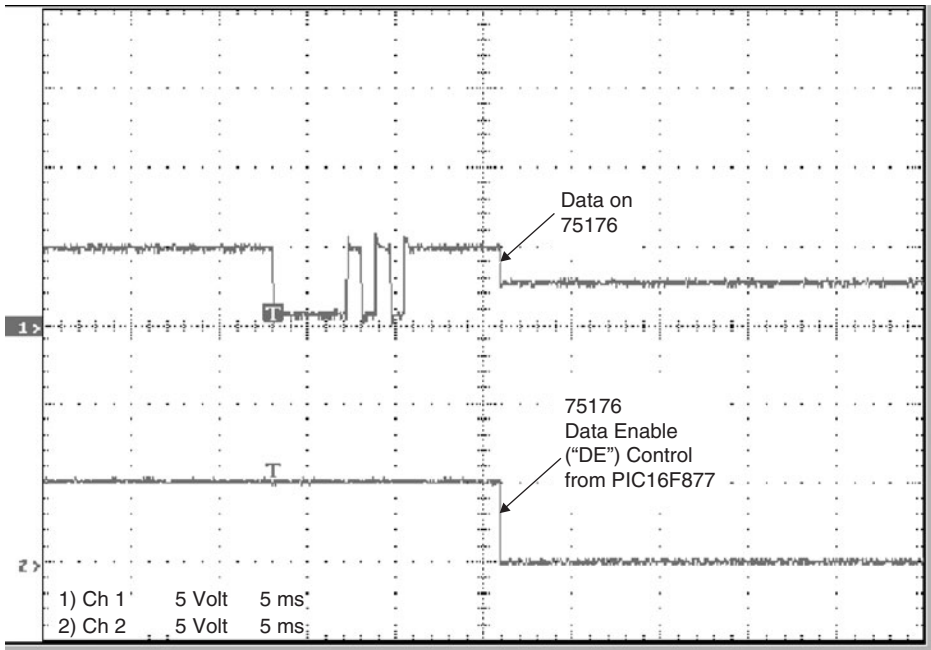


Figure 21.43 The voltage levels on the RS-485 lines when the remote thermometer circuit is disconnected.

If the remote PIC microcontroller is disconnected or cannot respond in any other way, the PIC16F877 will experience the data shown in Fig. 21.43. When the RS-485 voltage goes to the half-voltage level, the 75176 will convert that to a low with periodic voltage spikes. The voltage spikes are caused by noise on the RS-485 lines that differ enough for the differential input drivers to interpret them as high-voltage values. These spikes are the reason why I poll the line for 10 ms before accepting a low-voltage level as a valid NRZ serial character in the remote PIC microcontroller.

When the low data is received by the PIC16F877's USART, the data is interpreted as an invalid character (it is not in 8-N-1 format, and the FERR (framing error) bit of the RCSTA register is set. This bit indicates that when a high value was expected for the Stop bit, a low was received. To clear this error, I reset the USART before transmitting the ping character.

As I've indicated, this is not a terribly hard circuit to build. I recommend that you do test the two cards and their connections on a bench using the power supplies I outlined earlier to avoid the need to power up and power down the PC to debug the application. Something to keep in mind is that it is often quite difficult probing a card inside a complete system, especially if there are other cards in the system. On the bench, with the two cards hooked up and working correctly, you should see the LED flash on and off at a rate of once per second. I was not able to get an LED connected to the PIC16HV540 to work because of the low current source and sink capabilities of the chip. Once the LED is flashing (and you can stop it by disconnecting the remote

PIC microcontroller), you can test it out in your PC. By debugging the communications circuitry outside the system, you will find it to be quite easy to debug the circuit as well.

If you build this circuit for another system, I recommend that you first try to communicate with the remote PIC microcontroller using a debugger or monitor program that allows you to read and write devices on the I/O bus. When I first tested the ISA card and connections in the PC, I used MS-DOS debug rather than going directly to the Visual Basic application. You should be able to peek into the PIC16F877 to see what the current temperature value is. If you have wired up your bus interface correctly, you will be surprised to see temperatures being returned from the PIC microcontroller.

SERVO CONTROLLER

This project really demonstrates what kind of complete applications can be implemented with the PIC microcontroller. The servo-controller project uses a 16C71 to provide a text user interface with an LCD and allows the user to control up to four servos, develop a sequence of events for the servos to run, and allow the user to save a sequence for later execution (Fig. 21.44). It's a pretty impressive application for an 8-bit microcontroller with only 18 I/O pins. This project is designed for controlling the servos used in armature robots or mechanical displays that require moving parts; the servos can be controlled either individually or sequenced. There are a number of companies that make similar applications for sale for several hundreds of dollars—even using new parts, you can do it for \$20 or less.

The schematic is shown in Fig. 21.45, and you should notice that a lot of the lines are actually Vcc or ground. One thing to point out in this schematic is that I did not include the power for the 74LS174. Pin 16 is connected to Vcc and pin 8 to ground. Table 21.19 lists the project's bill of materials.

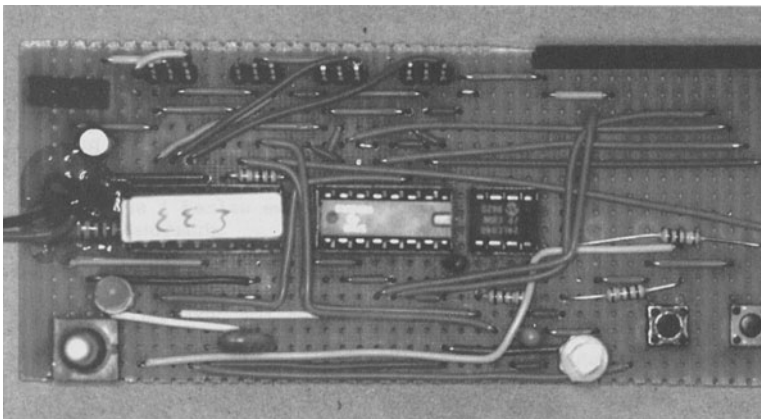


Figure 21.44 The completed four-servo-controller circuit.

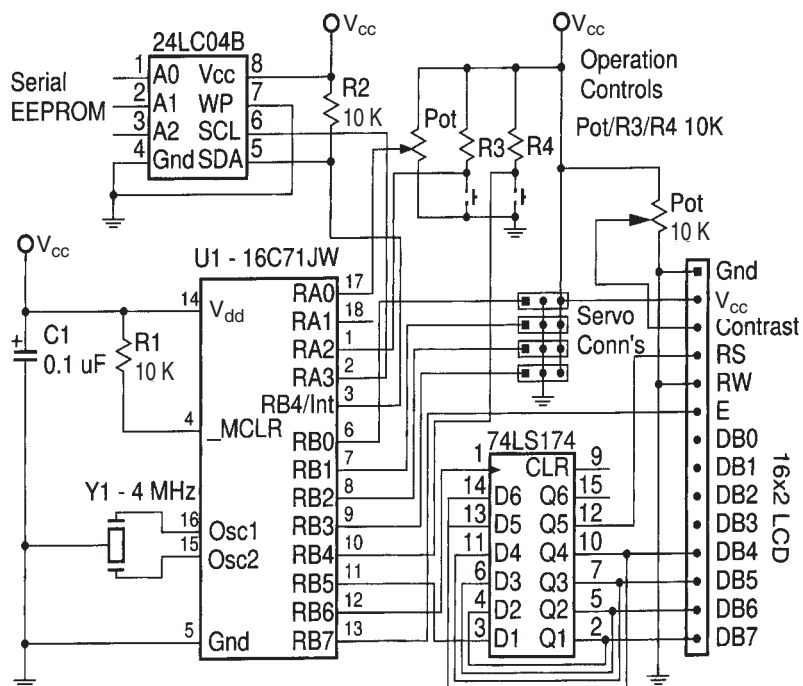


Figure 21.45 The servo-controller circuit consists of a PIC16C71 and a 74LS174 used as a serial-to-parallel converter for the LCD.

TABLE 21.19 SERVO-CONTROLLER BILL OF MATERIALS

PART	DESCRIPTION
U1	PIC16C71/JW
U2	25LC04B I2C serial EEPROM
U3	74LS174 hex D flip-flop
Y1	4-MHz ceramic resonator with internal capacitors
C1–C3	0.1 μ F any type
R1–R4	10 k Ω , $\frac{1}{4}$ W
R5, R6	10-k Ω single-turn potentiometers
SW1, SW2	Momentary-on pushbutton switches
Misc.	Prototyping PCB, wiring, 8, 16, and 18 DIP sockets, 14 $\times$ 1 0.100-in socket for LCD, 4 $\times$ 3 $\times$ 1 0.100-in pin headers for servos, 5-V, 500-mA power supply

When I first developed the circuit (and the preceding schematic), I just went ahead and assembled it on a vector board (as shown in Fig. 21.44). A vector board is a good prototyping tool for this project because of the repeated power and grounds that are easily bussed.

The user interface consists of a pot, two buttons, and a 16-character by two-line LCD. Making it all work together is a menu routine that takes a source message, puts it on the screen, and then handles user input in the form of a pot position and then buttons selecting the action.

The initial screen looks like this:

```
Servo Controller
>Pos< Pgm  Run
```

The button at RB4 is the `Select` button, which will move the cursor (the `>` and `<`) between the actions on the lower line of the LCD. When the `Enter` button (at RA2) is pressed, the display program is ended, and the cursor position is returned to the caller. The pot is used to select arbitrary values (i.e., the `##`).

Using a pot to specify exact values is kind of tricky and will require some practice and patience. An optimal solution would be to use a multiturn pot instead of the single-turn one that I used (you also should put on a knob to make turning the pot easier on the fingers). The reason why I went with the pot in the first place is because the servo position and program delay values are not really precision operations.

The user interface is used to provide a non-language-based programming environment. With the pot `Select` and `Enter` buttons, you can specify an immediate servo position, enter in a program, or run the program (either single steps or running with 20-ms nominal steps).

I used radio-control model servos for this project that rely on a 1- to 2-ms pulse (the duration specifies the position) every 20 ms. The 20-ms cycle is a natural for a TMR0-based interrupt handler. When invoked, the interrupt handler outputs a pulse of 1 ms to each servo and then loops with a counter. When the counter value is greater than the value for a particular servo, the pulse for that servo is turned off. The counter continues to loop until a full 2 ms (to allow full travel of a servo) is complete. TMR0 then is reset to an 18-ms delay (so that the whole cycle is 20 ms long).

The mainline program is responsible for updating the servo positions from user input or the application program in the serial EEPROM.

Note that for this project, the servo position granularity is such that there are 50 steps from stop to stop. This is due to the parallel control of the servos; the count loop takes quite a long period of time checking each servo value to see if it has to be updated. The number of steps can be increased by using either a faster PIC microcontroller clock or fewer servos and taking out the code used to support the unneeded devices.

The last major subblock of this project is the I2C serial EEPROM. The 24LC04B provides 4 kb or 512 bytes of EEPROM. For each program instruction, 2 bytes are used, in the format shown in Table 21.20.

TABLE 21.20 SERVO CONTROLLER I2C MEMORY DATA	
BITS	FUNCTION
15–10	Instruction check sum
9–8	Instruction type
	11 Print character on LCD
	10 Delay n/10 seconds
	01 Goto instructions
	00 Set servo position
7–0	Servo postion/goto location/ delay value/display character

When the program is running, it can be stopped manually by pressing the **Enter** button. The PIC microcontroller itself communicates with the serial EEPROM by behaving as an I2C master. The 24LC04B is an I2C slave device, which means that it responds to instructions directed to it by a master (these instructions are prefaced by the I2C control byte, as described elsewhere in this book).

There are two things to note in this application. The first is with the 24LC04B; the ninth bit of the byte address is in the leading control byte (the least significant 8 bits are passed in the next byte). The other thing to note is that the A0–A2 pins on the EEPROM package are *not* connected to the chip inside. This means that the 24LC04B cannot be used with other EEPROM devices (which typically use the A0–A2 bits to differentiate each other when the control byte is sent) on the I2C bus because of the danger of contention (two devices each trying to transmit data).

The actual code for communicating with the EEPROM is quite simple and, as it’s laid out in `ServoEE3.asm`, makes reading and writing to the EEPROM simply just subroutine calls. The code used in this application was modified to be more general and used as part of the include files presented in the appendices.

THE MIC-II

The one project I did not get any feedback on in the first edition was the MIC; this is a software application that actually provides a complete debug and development system for a simulated microcontroller with a built-in UART in a 1,024-instruction PIC microcontroller. I would have thought that the sheer audacity of such a device would have made a number of people stand up and take notice. Along with the MIC-II application itself, I have also included two sample applications to give you an idea of what can be done with it.

The Parallax BASIC Stamp actually was how I first got interested in PIC microcontrollers; I was fascinated with how an 18-pin microcontroller could be used to load and execute PBASIC applications. The actual hardware was very simple and quite efficient, although I felt that there were three shortcomings with the BASIC Stamp. The first shortcoming I saw was the need for an external chip to store the program in. When I first started to work

with the PIC microcontroller, the device that I really got interested in working with was the PIC16C84. This was an earlier version of the PIC16F84 that is featured in Chap. 20 and has 64 bytes of data EEPROM that could be used for storing applications. The second shortcoming with the Parallax BASIC Stamp was the need to compile applications on a host computer and the limited debug capabilities of the interface. This is not to say that the Parallax development software is not efficient, just that it requires a PC connected to the BASIC Stamp, and that seemed limiting. At the time, I was working with a number of different manufacturers' high-end UNIX workstations, and I wanted to see more PIC microcontroller applications available for them and take advantage of their RS-232 interfaces. The last concern I had about the BASIC Stamp was the relatively slow execution speed (1,000 instructions per second for the BS1 and 2,000 instructions per second for the BS2). I felt that a PIC microcontroller with an internal data EEPROM could run much faster and take up less space.

As good as I think I am with the PIC microcontroller, I did not think I could develop my own BASIC language that could fit into the PIC microcontroller's 1-kB address space. I decided to see what I could do with a simulated/emulated microprocessor executing its own assembly-language instructions. I am very happy with what I have been able to implement with the MIC-II.

The first step in this project was to come up with the emulated device's architecture. I don't know if the way I did it was putting the cart before the horse (I've never designed a computer architecture before), but I based the design on what I wanted the instructions to look like when they were displayed for the programmer. I wanted to take advantage of the PIC microcontroller's hardware features, so I designed the microcontroller to use a similar Harvard architecture to the PIC microcontroller. This means that program memory (which is the PIC16F84's 64 bytes of EEPROM data memory) is separate from the 11 file registers (including the PIC16F84's PORTB). With this basis, I wanted there to be a simple monitor program built into the PIC microcontroller with its own assembler and disassembler. Along with handling source code, the monitor program would control application single-stepping, program execution, and modifying and handling instructions.

The design I came up with is a true 8-bit processor—all data paths are 8 bits wide, and there is only a maximum of 8 address bits. The architecture that I came up with is shown in Fig. 21.46. This architecture is able to handle four addressing modes and can provide four symbolic labels within the application.

A sample application circuit could be similar to the one shown in Fig. 21.47, based on the bill of materials listed in Table 21.21.

To make the instructions more intuitive, I wanted the instructions to be referenced to the accumulator at all times. I felt that this also would greatly simplify the work required to implement the assembly language. I wanted to create instructions in the format

Verb noun

which is the action (arithmetic, data movement, etc.) followed by the object of the action. Thus, for the addition instruction

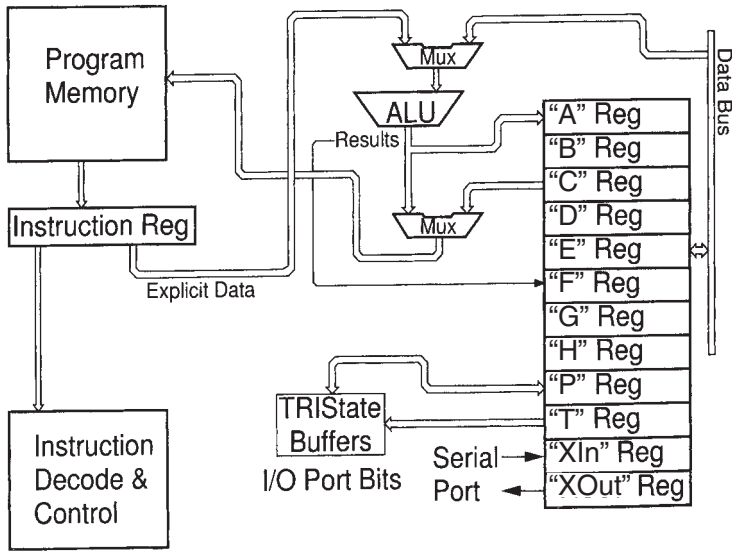


Figure 21.46 The MIC-II processor architecture.

the complete operation would be

```
Accumulator = Accumulator + 0x077
```

With this format, I decided on the 10 instructions for the MIC listed in Table 21.22.

Each one of these instructions has a parameter that is addressed in one of four modes. The four modes, which will be discussed in greater detail next, are immediate, direct, indirect, and label.

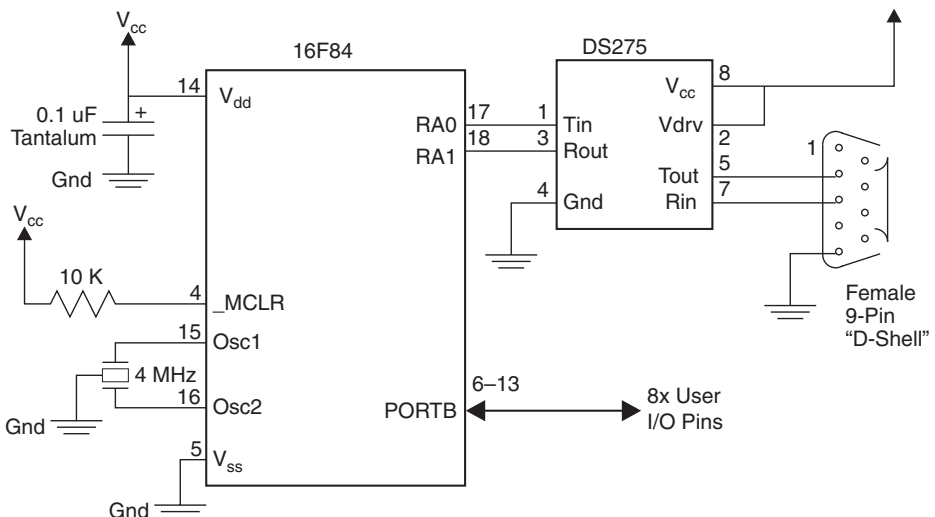


Figure 21.47 The MIC-II circuit is very simple and just relies on a +5-V source.

TABLE 21.21 MIC-II BILL OF MATERIALS

REFERENCE DESIGNATOR	DESCRIPTION
16F84	PIC16F84-04/P
DS275	Dallas semiconductor DS275
C1	0.1 μ F tantalum
R1	10 k Ω , $\frac{1}{4}$ W
Y1	4-MHz ceramic resonator with built-in capacitors
J1	Female 9-pin D-shell
Misc.	Prototyping PCB, wiring, +5-V power supply

The 10 instructions gave me all the necessary functions I could think of (with some caveats).

A shift left of the contents of a register would be accomplished by adding a value to itself:

```
< Reg          ; Load Accumulator with Reg
+ Reg          ; Reg + Reg = Reg << 1
```

“Branch on condition” is implemented by moving the MIC’s STATUS register contents into the accumulator (A register) and then testing the condition of a bit within the register. A skip on carry set would be

```
< F            ; Load the Accumulator with the Flags Register
# 0            ; Skip if Carry Bit Set
```

TABLE 21.22 MIC-II INSTRUCTIONS

INSTRUCTION	DESCRIPTION
>	Load the accumulator
<	Store accumulator in register
+	Add to contents of accumulator (store result in accumulator)
—	Subtract from contents of accumulator (result in accumulator)
	OR with contents of accumulator (store result in accumulator)
&	AND with contents of accumulator (store result in accumulator)
^	XOR with contents of accumulator (store result in accumulator)
/	Shift register to the right by 1 (store result in accumulator)
#	Skip the next instruction if the specified accumulator bit is set
@	Jump to the specified address, store address + 1 in B

You'll also notice there isn't a `call` instruction; instead, any time a `goto (@)` instruction is executed, the incremented program counter is stored in B. Doing subroutines this way saved code and file registers needed for a program counter stack. A traditional subroutine `return` instruction would be implemented using the two instructions

```
< B           ; Get the return value
> C           ; Change the Program Counter
```

This subroutine return will be expanded on in the following examples.

With the instruction set specified, I then decided on how to do the register addressing. I decided on four modes: immediate, direct, indirect, and label. The first three can be shown easily with the following example instructions:

```
+ 37          ; Add 0x037 to contents of Accumulator
+ D           ; Add contents of the "D" register to Accumulator
+ [D]         ; Add contents of register addressed by "D" to the
               ; Accumulator
```

If you look at these instructions, you'll see that each instruction takes data in these formats and acts on the value they represent. Note that when specifying hex values, where the most significant nybble is in the range of A to F, you should put in a leading zero to the value or the PIC microcontroller will assume that the A to F registers are being accessed and code the instruction accordingly.

In the application, there are four labels (A through D) that are assigned during application development. When the application is being passed to the MIC, a label is assigned at a specific address using the `!` directive to identify the address as a label (instead of a register). Accessing this address from within the source code uses the `!` Letter format. For example:

```
@ !A          ; Jump to Address for Label "A"
```

The label is stored as an address in the data EEPROM, and when the application references the label, its value is moved to a register. Loading the accumulator with a label value is accomplished using the instruction

```
< !A          ; Load Accumulator with the Address of Label "A"
```

In specifying the registers, I decided on eight Base registers and three special-purpose registers. The base registers (A–H) all can be written to and read from, but there are some special purposes assigned to some of the registers, as shown in Table 21.23.

The program counter C can only contain values between 0x00 and 0x1D. The last two possible addresses (up to 30 instructions can be programmed into the MIC-II as 16-bit words) are used to store the label addresses as well as provide a check for a valid application in the EEPROM. If a value outside these limits is written to the C register, it is loaded with 0 as a default.

TABLE 21.23 MIC-II REGISTER DEFINITIONS

REGISTER	FUNCTION
A	Accumulator (the destination for all arithmetic operations)
B	Goto return address
C	Program counter
F	Flags register
D, E, G, H	General-purpose registers
P	I/O port data bits
T	I/P port tristate control (TRIS) bits
X	Serial receive/transmit register

The F (or flags) register contains the result of arithmetic STATUS along with the serial receive/transmit status. This register is continually updated after each instruction or as the serial interface receives or transmits data. The flags bits are arranged as shown in Table 21.24.

The high 4 bits are the complement of their low nybble counterpart to allow the “skip on bit set” to work for all conditions. Note that the flags register cannot be written to from either the user interface or the application. If the register is written to in the application a “No instruction” halts execution. This means that a `breakpoint` instruction in the MIC-II is

> F ; Invalid Write to Flags Register

TABLE 21.24 MIC-II FLAG REGISTER DEFINITION

BIT	FUNCTION
0	Carry flag, set on addition greater than 0x100 or subtraction less than 0
1	Zero flag, set on arithmetic result equal to 0
2	Receiver byte waiting
3	Transmitter free
4	Not carry flag
5	Not zero flag
6	Not receiver byte waiting
7	Not transmitter free

A `nop` in the MIC-II is a `goto (@)` instruction pointing to the next instruction in the program memory.

The UART (universal asynchronous receiver/transmitter) built into the MIC-II is a three times sampling NRZ software handler using the PIC16F84's TMR0. The timer algorithm (and code) used is essentially identical to that of the YAP-IIs. When the application is running (using the `R` command), a Ctrl-C (ASCII 0x003) byte will stop the application running, or you can insert the `breakpoint` instruction (`> F`), as discussed earlier. I should point out that at some point it would be interesting to port the MIC-II code to a PIC microcontroller that has a built-in UART to simplify operation of the application code.

The three special-purpose registers are the P, T, and X registers. P and T are the PORTB and TRISB registers of the PIC microcontroller and operate identically as to how they work in the PIC microcontroller. When a 0 is written to a T bit, the corresponding PORTB bit will be put into output mode. When a 1 is written to a T bit, the corresponding PORTB bit will be in input mode.

The X register, when written to, transmits the byte and, when read from, reads the last received character (or 0x000 if there isn't an unread character). Data should not be written to until the "transmitter free" bit is set, and while data is being sent, the "transmitter free" bit will be reset. When a character is available for reading, the "receiver byte waiting" bit will be set. Reading from the X register will reset this bit. As I indicated earlier, the monitor program will poll the incoming data, and when a Ctrl-C is received, execution will end.

With the instructions, architecture, and peripheral devices defined, I then turned my attention toward the user interface. As I indicated earlier, this interface consists of an NRZ serial interface running at 1,200 bps with an 8-N-1 data packet format. Any commercially available RS-232-level translator can be used to interface the circuit to a host computer.

When the MIC-II is executing, it behaves as if it is an emulator controlling the processor. The prompt that I came up with for the user is

```
[Label:...] PC ACC Flags Ins > _
```

where !Label = label at the current address

PC = program counter

ACC = accumulator

Flags = zero and carry flags

Ins = the disassembled instruction at the program counter

You probably will note the similarity to this prompt to the one I used for the EMU-II. The different commands that can be entered into the MIC-II are listed in Table 21.25.

To set the label positions, first specify the address you want the label at. To do this, assign the desired address to the C register. Next, put in the !Label command on the next line. Figure 21.48 shows the process for running the MIC-II application code in a PIC16F84 that is in a YAP-II socket. After starting the MIC-II application running at 4 MHz, I change the program counter C to address 4.

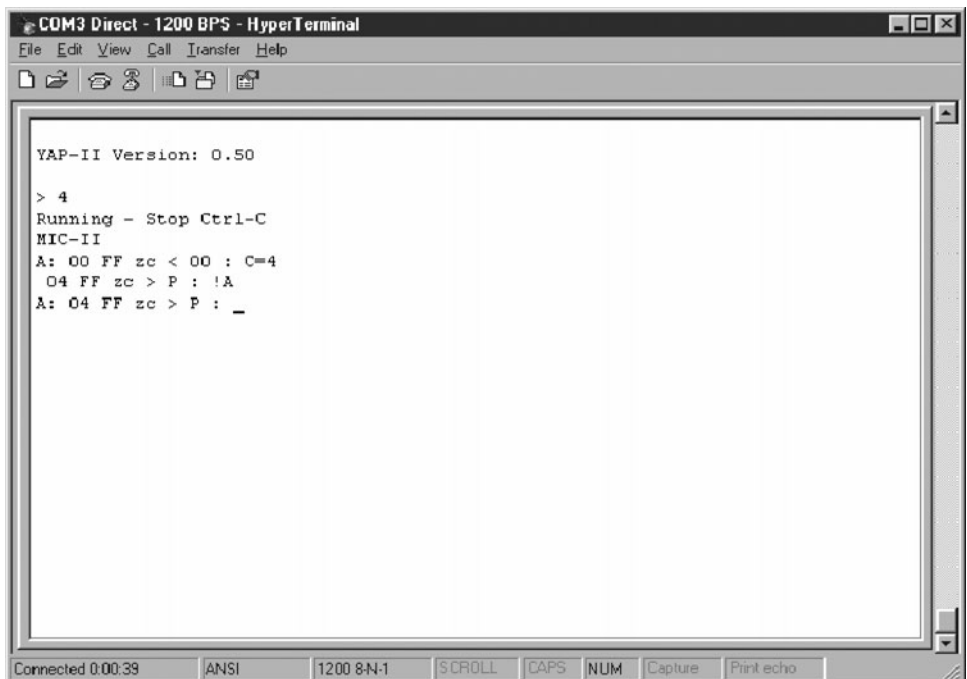
With the instruction at address 4 displayed, I then enter !A to indicate that the ALabel should move to this address. The final line of Fig. 21.48 shows label A assigned to address 4.

TABLE 21.25 MIC-II COMMANDS

COMMAND	OPERATION
Nothing/"1"	Single-step the MIC-II processor. Execute one instruction and update the program counter (C register)
"R"	Run the MIC-II at full speed from the current program counter (C register) location
Register	Return the contents of the specified register
Register = Constant	Set the register to a specific value
!Label	Specify the label position
Instruction Verb Noun	Load the instruction with parameter into the PIC16F84's data EEPROM

Labels are used in the application as in any traditional programming language. Jumping to B from the current execution is accomplished by referencing the label in the instruction as shown in the following code example:

```
@ !B                ; jump on to Label "B"
:
B:
```

**Figure 21.48** MIC-II label-assignment keystrokes.

Here is a simple test program that you may want to try. It is designed to have four LEDs connected to the PIC16F84's RB0 to RB3 pins and will increment them and then restart. I have written it with the labels on the start of the line. As I showed earlier, to put in a label, you will have to enter in the `!Label` command before the instruction. This command does not increment the C register, so you can enter the instruction right after it.

```
A: < 0      ; Enable all the Outputs
   > T
   > D      ; Use "D" as the Output Counter
   ^ 0FF   ; Invert the Data to Turn OFF the LEDs
   > P
   < 10     ; Loop Sixteen Times before Restarting
   > E
B: < D      ; Increment the LED Display
   + 1
   > D
   ^ 0FF   ; Invert the Count for the LEDs
   > P
   < E      ; Decrement the Loop Counter
   - 1
   > E
   < F      ; Check for being Done
   # 1     ; Skip Next if "Zero" is Set
@ !B      ; Not Zero, Repeat at "B"
@ !A      ; Start All Over Again
```

When this application runs, 184 instructions are executed before looping back to the start and beginning again. When I watched this on my oscilloscope (probing RB3 because this cycles once during the execution), it took 19.4 ms for an average execution speed of approximately 9,500 instructions per second.

If you want to watch the LEDs changing, you could try the application listed below. This code calls a subroutine at `C`: that loops 256 times to provide a delay in the application so that it can be observed.

```
A: < 0      ; Enable all the Outputs
   > T
   > D      ; Use "D" as the Output Counter
   ^ 0FF   ; Invert the Data to Turn OFF the LEDs
   > P
B: < D      ; Increment the LED Display
   + 1
   > D
   ^ 0FF   ; Invert the Count for the LEDs
   > P
   @ !C     ; Jump to Label "C"
   @ !B     ; Loop Back to "B" Forever
```

```

C: < B      ; Save the Return Address
   > H
   < 0      ; Loop 256 Times
   > G
D: < G      ; Increment the Counter
   + 1
   > G
   < F      ; Check to See if Zero Set (256x)
   # 1
@ !D
   < H      ; Return to Caller
   > B

```

This application is shown entered into the MIC-II in Fig. 21.49. To produce the listing, I changed the program counter (C register) value to the next instruction.

In the application's subroutine, notice that I save the value in the B register into the H register. I did this because within the subroutine I perform a `goto` instruction that changes the value of the B register to the address following the `@ !D` instruction. This is not the return address for the subroutine. When I first coded the application, I forgot to do this, but I was able to find the problem by stepping through the application (after changing the value loaded into G into 0x0FE or 254 to enable me to quickly execute the code in the subroutine's loop).

```

A: 00 2A zc < 00 : C=1
01 2A zc > T : C=2
02 2A zc > D : C=3
03 2A zc ^ FF : C=4
04 2A zc > P : C=5
B: 05 2A zc < D : C=6
06 2A zc + 01 : C=7
07 2A zc > D : C=8
08 2A zc ^ FF : C=9
09 2A zc > P : C=A
0A 2A zc @ !C : C=B
0B 2A zc @ !B : C=C
C: 0C 2A zc < B : C=D
0D 2A zc > H : C=E
0E 2A zc < 00 : C=F
0F 2A zc > G : C=10
D: 10 2A zc < G : C=11
11 2A zc - 01 : C=12
12 2A zc > G : C=13
13 2A zc < F : C=14
14 2A zc # 01 : C=15
15 2A zc @ !D : C=16
16 2A zc < H : C=17
17 2A zc > C :

```

Figure 21.49 On-terminal emulator MIC-II subroutine example.

NTSC VIDEO OUTPUT

Throughout this book I have used LEDs extensively to provide user feedback on the status of an application or its input. I also have used LCDs to display text data for users to allow complex operations to be explained rather than relying on panels with text or instruction books. For your home electronics, one of the output devices that you are probably most familiar with is the cathode-ray tube (CRT) of your TV, which is used to provide I/O for the TV itself, your VCR, DVD player, and maybe your stereo.

If you've looked at the different PIC microcontroller projects that are available on the Internet, you should not be surprised to find that there are a number of different projects available for the PIC microcontroller in which it can be used to drive NTSC video output. In this project I would like to introduce you to National Television Standards Council (NTSC) composite video and a PIC microcontroller application that will show you how to process PIC microcontroller ADC data along with moving data. While the hardware is very simple and the software is also quite simple, this is one of the most challenging projects in this book.

I do not address other standards, such as PAL or SECAM, but the generation of composite video for these standards is simple, and this application should be reasonably easy to port to these systems.

This project demonstrates how ADC data can be captured while driving an NTSC composite video output along with computing the position of a bouncing ball. If you've looked on in this book, you'll see that the circuit that I came up with for this application is extremely simple, but the code probably took me the most time to develop, and this was probably the most challenging project for me to develop in this book. The reasons for this is the stringent timing required by NTSC and the challenges I had in developing the simple displays used in this application.

Before explaining the application, I should first introduce you to composite NTSC video, as well as give you two warnings. The first warning is this: For this project, I used a \$10 television that I bought at a garage sale and used a \$1.50 video modulator that I bought at a surplus store to convert the composite video that the PIC microcontroller circuit produces—if you can't find a similar modulator, you can use an inexpensive video game modulator/composite video switch that can be found at most stores selling audio/visual equipment. This signal is passed to the TV using a standard 75- Ω coaxial cable.

I did not modify the TV in any way (such as providing a bypass to the video preamp from the tuner), and I don't recommend that you do this either. There are potentially lethal voltages inside a TV, stored in capacitors that are still present even if the TV is turned off and not plugged in. While the circuit here should not produce any voltages or currents that could damage a TV, I don't recommend that you hook this circuit up to the family's large-screen TV (or any TV that you care about). Instead, you should look for an old 12-inch black and white TV that you can pick up for a few bucks. If flames come out of your family's home entertainment system, you only have yourself to blame.

The second warning is: the circuit and software presented here are essentially a video transmitter. The modulator will convert the composite video to a frequency that may be picked up by your neighbors' TVs on channel 3 (which they are probably using for cable converters, VCRs, DVD players, and the like). Please be sensitive to

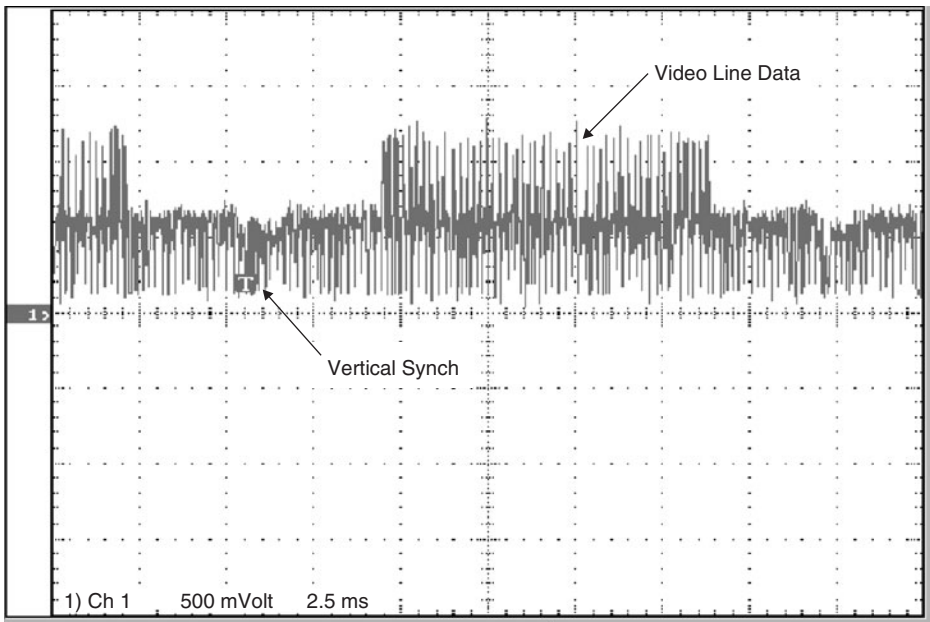


Figure 21.50 The waveform for a complete NTSC frame of data.

whether or not you are interfering with their reception (or you may get a visit from your local FCC representative).

If you find that there are any problems with your home TVs or radios while this circuit is in operation, shut it down and use a different modulator or cable setup. These problems will manifest themselves as snow (white spots randomly on the screen) or audio static. You may find that you are unable to use this application without causing problems—in such a case, go on to the next project.

With the caveats out of the way, let's look at the NTSC composite video signal produced by this application. After the tuner in your TV set has demodulated an incoming signal, the actual video information, called *composite video*, is passed to the video drive electronics. If you were to look at the composite video signal for an entire frame of data on an oscilloscope, you would see something like Fig. 21.50.

In this figure I have identified two features that you will have to become familiar with. The first is the *vertical sync*, which is a series of specialized pulses that tells the video drivers to reset the *raster* (the electron beam that travels across the CRT) to move to the top left-hand corner of the screen. During each line, data is output as an analog voltage. After the vertical sync is sent, video line data is sent, each line of the data being a corresponding line on the TV display.

The description I'm going to give here is for black and white composite video with no colorburst information. The colorburst is a 3.579-MHz sine wave that is output after the horizontal sync to allow the video circuitry to latch onto the phase of the color signal sent to the TV. Along with the brightness (or *luminance*) information, which consists

of dc voltage levels, there is an analog signal attached to the signal as well for the color (or *chrominance*) value for what is displayed on the TV's screen. The color displayed depends on the phase of the 3.579-MHz chrominance signal—by changing the phase of the signal, the color is changed on the display. Detection of the phase difference is accomplished using a phased-locked loop circuit.

Composite video gets its name from the fact that it combines three different signals (the vertical synch, horizontal synch, and video data) all in one line. Special circuitry in the TV (or CRT as I will call it for most of this section) splits this information out to control how the output is displayed. The period of time between vertical synchs is known as the *video field*. For NTSC composite video (which this circuit creates), 59.94 fields are displayed each second. One complete *frame* of video consists of two fields, with the raster scan line of one field overlapping the other (which is known as *interlaced video*). The output produced by the project shown here produces the appropriate timing for the output data to repeat over two interlaced lines.

As part of the vertical synch, a number of unused horizontal lines are passed at the same time and are known as *vertical blanking*. When the vertical synch is recognized within the TV and the raster moves to the top of the screen to begin scanning again, the CRT guns are turned off to prevent any spurious signals from being driven on the screen. The vertical syncing operation is shown in Fig. 21.51.

The analog voltage output used for synch pulses and CRT control are always at a level below the video data black level and are often known as *blacker than black*. The normal synch level is at 0.4 V, whereas the active synch pulse is at 0 V.

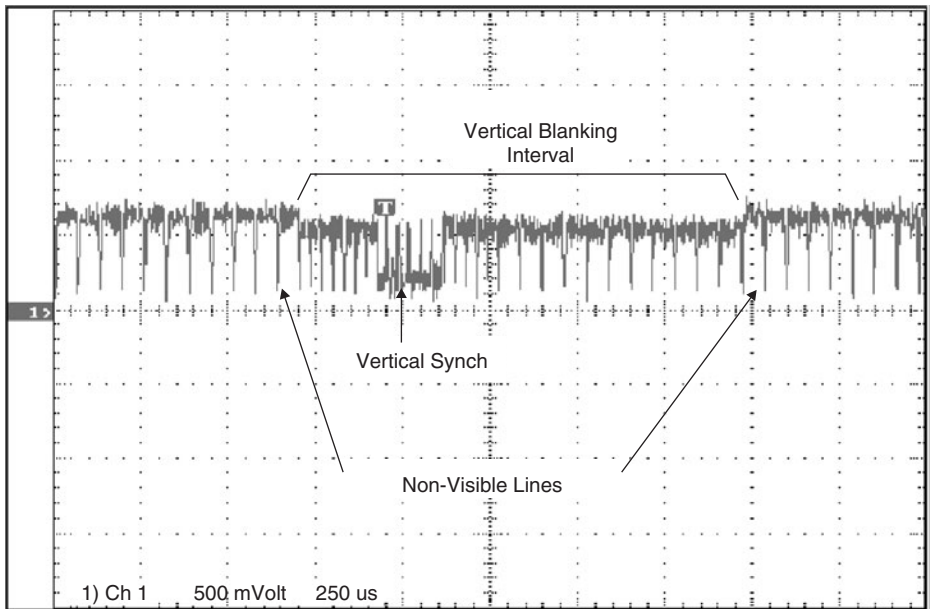


Figure 21.51 NTSC vertical sync signals.

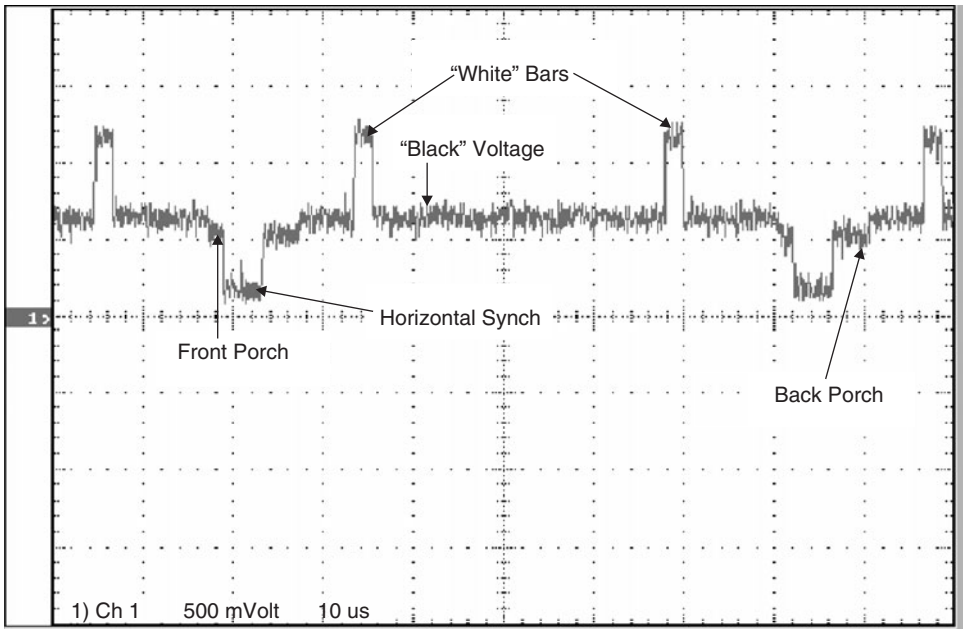


Figure 21.52 A single NTSC raster (line) of data.

The vertical blanking before the vertical synch consists of 6 or 7 half lines (at $31.8 \mu\text{s}$ long) followed by 6 negative half lines (these are the vertical synch pulses) and then another 6 or 7 half lines followed by 10 or 11 full lines (at $63.5 \mu\text{s}$ long).

With the raster gun now pointing to the top left of the CRT, data can be output a line at a time. One of the lines from this circuit is shown in Fig. 21.52. Each line is $63.5 \mu\text{s}$ with a horizontal synch pulse to indicate where the line starts followed by the data to be output on the line. The data output ranges from 0.48 V (black) to 1.20 V (white), with gray being the voltages in between. In the figure you can see the different voltage levels for the horizontal synch, the front porch, the back porch, black, and white.

Note that there is approximately 100 mV of noise on each of the signals. This is largely due to the prototyping construction I used for this application and the poor ground that I have for it. In a “professional” application, I would expect that the noise on the line would be on the order of 10 mV or less. For this application (and the very cheap used TV set that I used), the 100 mV of noise did not cause a problem with image stability (a big consideration for video applications) or the brightness output of different parts of the signal.

The front porch and back porch are 1.4 and $4.4 \mu\text{s}$ in length, respectively, and are at 0.40 V. The synch pulse itself is a 0 voltage active for $4.4 \mu\text{s}$. These signals (with their voltage levels) must be present in any video signal. For the $53.3 \mu\text{s}$ after the synch pulse, the voltage level must be at 0.48 to 1.20 V. If the output dips below 0.48 V, there is a chance the TV (or CRT) will interpret the signal as a new horizontal synch with terrible results (in terms of the output display).

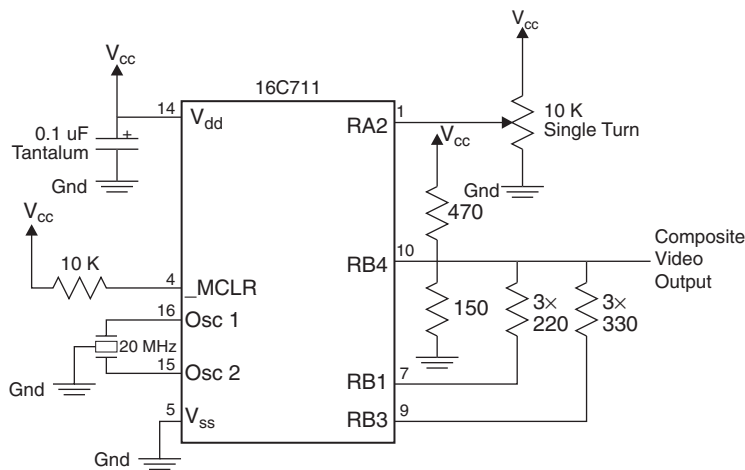


Figure 21.53 Video generator circuit.

For the TV to accept the composite video, the quoted signal lengths must be adhered to as closely as possible. Failure to have the same number of cycles on different lines will result in a “broken” screen. I will discuss this at length later in this project writeup.

The actual circuit I used to create the composite video is amazingly simple (Fig. 21.53), based on the bill of materials listed in Table 21.26.

The composite video voltage output was produced by placing different I/O pins in output mode with an output of 0. The circuit is designed so that only one pin can be

TABLE 21.26 VIDEO GENERATOR BILL OF MATERIALS	
REFERENCE DESIGNATOR	DESCRIPTION
PIC16C711	PIC16C711-JW
20-MHz ceramic resonator	20 MHz with internal capacitors
0.1-μF	0.1 μF tantalum capacitor
10k	10 kΩ, 1/4 W resistor
10-k pot	10 kΩ, single turn potentiometer
470	470 Ω, 1/4 W resistor
150	150 Ω, 1/4 W resistor
220	220 Ω, 1/4 W—note three 220-Ω resistors wired in parallel
330	330 Ω, 1/4 W—note three 330-Ω resistors wired in parallel
Misc.	Prototype board, wiring, +5-V power supply, video modulator, video modulator power supply; 75-Ω coax cabling, TV set

enabled as output at any one time. When a pin is in output mode, the pin that it is connected to is pulled down and allows current to pass through the resistor connected to the pin. This added resistance changes the resistance of the voltage divider output and changes the voltage output from it.

The circuit works very well and provides a very fast specific digital-to-analog conversion. In the oscilloscope pictures of the composite video signals shown in this section, I used the circuit presented in Fig. 21.53.

When I built this project, I used a SimmStick prototyping card for the circuit. One of the advantages of the SimmStick and other prototyping systems in applications such as this is that the unregulated voltage in is available with +5-V regulated voltage for the PIC microcontroller. I used this feature instead of having to come up with a dual power supply circuit for the application.

The video modulator was attached to the prototyping card with hot-melt glue, and the power, ground, and composite video outputs were passed from the SimmStick edge connector. When I first was setting up the project, I used a small Tyco toy video camera as a sample composite video source. An RCA socket was glued to the video modulator using hot-melt glue, the shield soldered to the modulator's case was used for ground, and the signal line was passed to the modulator's input. I originally created this SimmStick card for an Atmel AVR composite video output for the *Handbook of Microcontrollers*, and I've used it on a number of projects since.

Looking at the circuit (shown in Fig. 21.53) for this project, you probably have a few questions. The first is about my use of a ceramic resonator instead of a crystal for the PIC microcontroller clock. This is probably surprising, especially in light of the harping I've done about the importance of an accurate clock so far in this section. When working with a stand-alone video generator like this circuit, the critical parameter is to make sure that the timing is perfectly accurate relative to the various signals and reasonably accurate to the specifications.

I will discuss this in more detail later, but most modern (which is to say, built within the last 30 years) TV sets are able to work with a relatively wide range of input timing parameters. The reason for putting in this tolerance is not to make life easy for people like us but to make the TV set insensitive to changes within itself as the components age. I am continually amazed at the reliability of TV sets (and consumer electronics in general), and one of the reasons for this reliability is the ability of the circuit's designs to continue operating even though their components have degraded. This built-in tolerance makes the lives of experimenters like us much easier.

The second thing you probably will notice in Fig. 21.53 and the bill of materials that go with it is the use of resistors in parallel. The three 220- Ω resistors in parallel result in a resistance of 73 Ω , which is close to half the 150 Ω built into the voltage divider circuit. The three 330- Ω resistors in parallel have an equivalent resistance of 110 Ω . Both these resistances were calculated as part of the composite video output voltage for the 0-, 0.40-, 0.48-, and 1.20-V outputs needed for the composite video output. The actual values needed were 75 and 107 Ω , which are not available as standard values. By applying parallel resistance theory, I was able to approximate them quite closely instead of having to rely on standard values.

The composite video output is driven directly into the video modulator. Most video modulators have a capacitor in series with the input to negate any dc voltage offsets. To avoid any problems with RC delays in the composite video signal, I tried to keep the output impedance as close to $75\ \Omega$ as I could (without having to resort to buffering the analog output from the PIC microcontroller). This is the reason for the relatively low resistances in the digital-to-analog circuit in this project. Normally, when producing analog voltages using a voltage divider circuit (as I use in this project), I would tend to use values in the tens of kilo-ohms. TV circuitry is normally designed to work with low characteristic impedance signals, so resistors in the 10-k Ω range would have problems providing crisp changes in voltage levels to the video modulator with its input capacitor.

To find the resistor levels, I first calculated the proper values in a circuit that connected the resistors to ground directly (and not through an open-drain I/O pin). I then wired the circuit as shown Fig. 21.53 and found that the voltages had changed somewhat (actually, I just had to change the resistor always pulling to ground to 150 Ω).

After calculating the expected values for the resistors, I then put the proposed circuit on a breadboard and wrote a small application to test out the voltage output. This application simply pulled RB1, RB3, and RB4 to ground and allows me to monitor the voltage levels output by the voltage divider. The application that I used was **VLadder** application taken from Chap. 20. Using this application with an oscilloscope allowed me to verify the voltage levels output from the composite video generator circuit.

The I/O pins RB1, RB3, and RB4 may seem strangely arbitrary, but they were used because they simplified the wiring of my prototype on the SimmStick. Depending on the method you use to build your own circuit, you may want to change the specified I/O pins to keep your wiring as simple as possible.

With the circuit built, I then created `video1.asm`, which can be found in the `code\video` folder. `Video1` creates the various signal timings to produce the composite video output for the two fields that make up a frame. As I indicated earlier, 59.94 NTSC composite video fields are sent each second. Each frame is made up of two of these fields, each of the second field's raster placed in between the rasters of the first field. This creates a richer display with fewer visible raster scan lines than if a single field were used for a frame.

To start the vertical blanking interval, a set of half lines is sent. This line is 31.8 μs in length with a 2.2- μs synch at its start. For the first field of the frame, seven of these lines are sent, and six are sent in the second field. Next, a reverse image is sent as the vertical synch, with 2.2- μs porch voltage and the remaining 29.6 μs at 0 V. Six vertical synchs are output for both fields of the frame. Finally, 10 (for the first field) or 11 (for the second field) half lines are sent. For this application, which runs the PIC microcontroller at 16 MHz (with a 250-ns instruction cycle time), 127 instruction cycles were used for each line.

After the vertical blanking interval, 243 video data lines were sent using the specifications I outlined earlier. To produce a 63.5- μs line period, 254 instruction cycles were used for each line.

`Video1` was relatively easy to create, and I was able to get it working in an evening with just a few minutes of debug because I had done a lot of simulation of the application

before trying it out in hardware and driving out to a TV. The biggest issue I had was determining the location of the box so that it would be easily seen on the TV.

Next, I started work on `Video2.asm`, which set up an inverted U-shaped playing field on the TV's display along with a PIC microcontroller-generated paddle. The position of this paddle is determined by the voltage input from the 10-k Ω potentiometer into the PIC microcontroller's ADC. The debug of this program was considerably more challenging than that for `Video1`.

The initial problem I had was with displaying the playing field. This involved creating two routines. The first was used for generating the top line of the display. The second was to create the two vertical bars of the field. While not terribly hard to do, there was a fair amount of work involved in determining values that would result in the playing field properly displayed on my TV's screen.

Next, I did the paddle's position. To read and process the paddle's position, I initiated the PIC microcontroller's ADC in the first half line of the vertical blanking interval. In the second half line I then read out the ADC and then converted the value read into a value that could be displayed.

To do this, I had to know how wide the field was. This is why I did the playing field first. The playing field size that I settled on was 131 instruction cycles wide (32.75 μ s) with 8-instruction-cycle-wide (2 μ s) bars on each side. I wanted to have a bar that was about one-sixth the field wide, so I decided to make it 20 instruction cycles wide. This left 111 instruction cycles for moving the bar. Because I wanted to use the code

```
decfsz  PaddleCount, f
goto    $ - 1
nop
```

to provide a consistent 3-instruction-cycle delay to move the paddle, this gave me a granularity of 37 positions for the paddle on the field.

To scale the 8-bit output of the ADC to the 0 to 37 range of the paddle, I multiplied the 8-bit output by 37 and used the top byte for the paddle position. This value then was multiplied by three and then used to locate the paddle on the screen.

Creating the software to move the paddle accurately was a significant challenge for me (and I still don't feel that I have it 100 percent right). I found that there were five situations in which I had to be able to handle the paddle so as to appear to move linearly to the potentiometer input. These cases were

- Against the left wall
- One position from the left wall
- Two or more positions from the left wall and two or more positions from the right wall
- One position from the right wall
- Against the right wall

I found that each case had to be accurately timed, and this ended up taking most of an evening for me to do.

The final code, which is shown in the photograph at the start of this section, is `video3.asm`. This code incorporates a bouncing ball on the display. The ball moves diagonally until it hits a boundary of the playing field. The obvious original intention of this application was to create a Breakout type of game. The ball itself is three lines by three instruction cycles in size. Each time a new frame (two fields) is displayed, the position of the ball is updated by the size of the ball. There is a 16-bit variable called `BallPos` that is used to keep track of the x and y position of the ball on the screen, as well as the x and y direction in which the ball is moving. Each time the ball hits a playing field wall, the direction changes. The initial position and direction of the ball are determined by the `TMR0` value.

To be able to handle the ball's position, I separated each set of three lines of the playing field. The ball itself, if it is within these lines when the PIC microcontroller generates them, is displayed on the line.

Getting the timing correct for the ball was a daunting task that took me two evenings to do. After getting each case (the same as the paddle earlier plus the case of no ball) to execute in $63.5\ \mu\text{s}$ (or 254 instruction cycles), the movement of the ball seemed to be appropriate.

Unfortunately, in creating the multiple lines for the ball to be displayed, I used up an awful lot of space. This application uses 899 of the 1,024 instructions available in the PIC microcontroller and does not leave enough space for collision detection between the ball and the paddle or the ball and other features on the screen. To follow through and create a game from this application, I would want to use a 2,048-instruction PIC microcontroller such as the PIC16C712 instead of the 1,024-instruction PIC16C711 that I did use.

In the work that was done for this project, I was able to reach a number of conclusions. The first is that the PIC microcontroller is not well suited for generating video signals. Electrically, this was not a huge issue, and gray voltage levels could be added to the voltage divider output quite easily.

The significant problems were in the software used by the project. While the PIC microcontroller is running at a reasonably fast speed (16 MHz), this translated to being able to place a feature on the display arbitrarily with a best-case accuracy of three instruction cycles, or 750 ns. Since $53.3\ \mu\text{s}$ are available in each line for placing features, this gave a resolution of only 71 positions on each line. This positioning accuracy is even further decreased when the playing field width of 131 instruction cycles is available, with the ball being the smallest feature displayable on the screen with up to 43 different positions. Using a PIC18 microcontroller running at 40 MHz would improve the granularity of the ball position to about 107 different positions for each line of the playing field. Ideally, I would like to see something approaching 256 different positions for the display to be as accurate as possible.

With different cases, timing the application is a major headache and a difficult piece of work. This was especially true when I was calculating the ball's position with two playing field borders to also display at the same time. Ideally, a PIC microcontroller video generator application would have only one feature to put on any line at a given time.

This circuit could be used as the basis of a PIC microcontroller-based video data overlay circuit. Instead of generating the video timing signals within the PIC microcontroller, synch separators could be used to indicate the start of a field and a line for the PIC microcontroller to then count from and drive a signal on top of the input signal. An obvious example of an overlay circuit is the text generator used at the end of TV shows to display the credits over some other signal.

To get accurate positioning of the overlay, you probably will have to use some kind of phase locked loop (PLL) circuit to make sure that the PIC microcontroller's clock is locked on the source generator's clock and the overlay appears on the display at the same position each time. This isn't a terribly hard circuit to design and could be useful for placing crosshairs or arrows on a video signal indicating where problems or features to look at are.

PIC18 Devices

To finish off the example applications, I wanted to demonstrate how the PIC18 microcontroller is used in a circuit and how code is written for it. As time goes on, I believe that the PIC18 architecture will become the primary architecture for developing applications, just as the mid-range devices have replaced low-end devices as the basic PIC microcontroller (in the mid-1990s, what are now called mid-range devices actually were the high-end devices, with the low end forming the basis of most applications and products based on the PIC microcontroller. This belief is based on the wide selection of part numbers and features becoming available for the PIC18 parts, along with development tools and the ease in which applications can be developed for them.

FUZZY LOGIC FAN GOVERNOR

To finish off the projects, I want to take a look at using the PIC microcontroller as a digital controller. Elsewhere in this book I have noted that the 8-bit data word and limited mathematical capabilities decrease the PIC microcontroller's attractiveness as a digital signal processor (DSP). I want to take a look at the ability of the PIC microcontroller to implement fuzzy logic control that requires significantly less computational power than DSP algorithms. Along with using fuzzy logic for processing, I want to use built-in hardware features of the PIC microcontroller wherever possible instead of relying on "bit banging" interface methods. This design philosophy allows the hardware interfaces to be created quickly and modified easily when I am debugging the application. For this application, I want to use a PIC18 to test how easily code can be taken from mid-range PIC microcontroller applications and ported to the PIC18. In many ways this application is an experiment with some interesting results, especially in the development of fuzzy logic applications and what I learned about them (see Figs. 21.54 and 21.55).

The circuit used for this project was taken from the motor controllers that I have presented elsewhere in this book. The circuit shown in Figs. 21.56 and 21.57 (with the bill

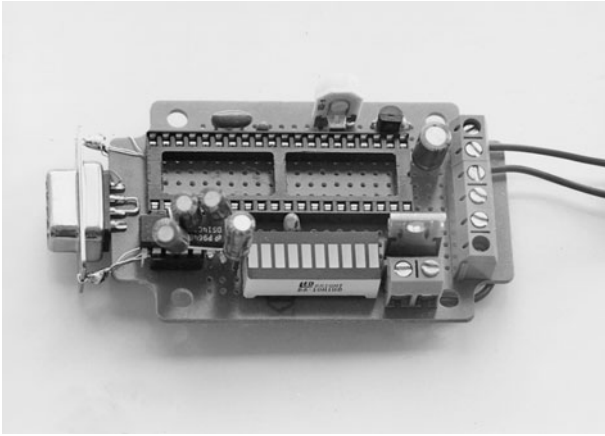


Figure 21.54 The completed fuzzy logic fan governor circuit.

of materials listed in Table 21.27) was built on a small prototyping card, as you can see in Fig. 21.54. Using point-to-point wiring, it took me about 5 hours to wire the application, with a few problems that I'll outline below.

When I built this board originally, I used a Dallas Semiconductor DS-275 voltage-stealing RS-232 interface. This chip had problems because the reflected data on the TX line, returning the data sent on the receiving line (a function of this type of RS-232 interface), caused problems with the fuzzyTECH tool during the fuzzy logic development. To get the RS-232 interface to work correctly, the MAX 232 was substituted. Unfortunately, installing this chip was a major exercise in point-to-point wiring.

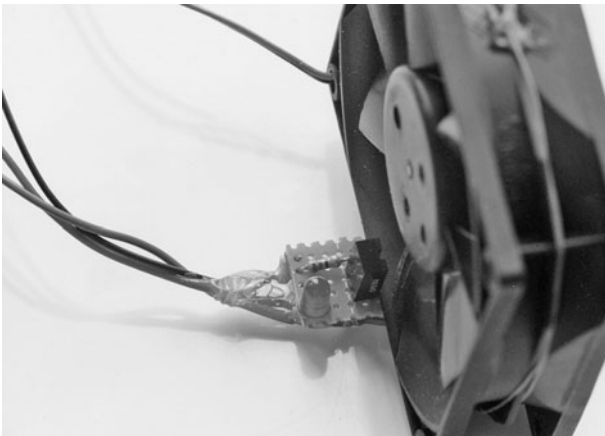


Figure 21.55 The modified fan with the IR interruptor fan blade sensor.

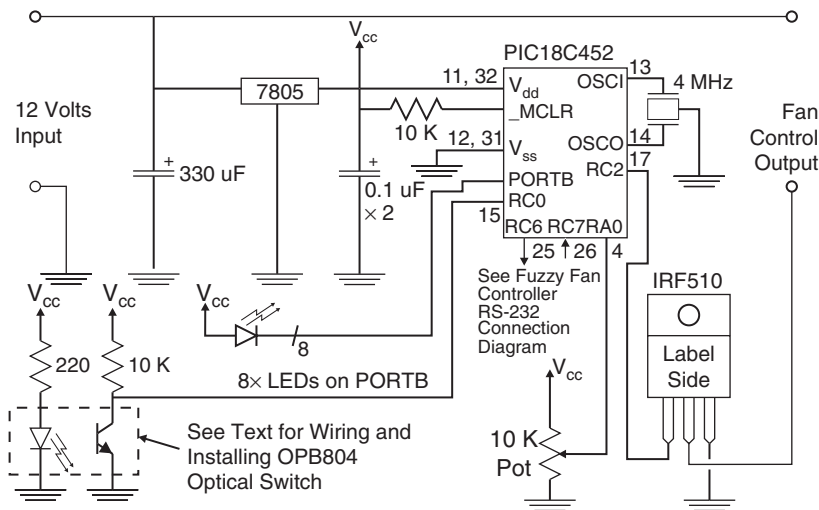


Figure 21.56 PIC microcontroller-based fuzzy logic fan controller.

The resulting circuit is half hanging into the DS275 socket and half hanging in air. Wiring this chip in took me at least as long as building the rest of the entire circuit.

To measure the fan's speed, I used an OPB804 slotted OPTO-isolator switch, as I've outlined in Fig. 21.58. This device consists of an infrared (IR) LED providing a signal to a phototransistor that turns on when the IR signal is received. When the IR signal is blocked, the phototransistor is off. I recommend that you prototype the circuit in the top right corner of Fig. 21.58 to better understand its operation.

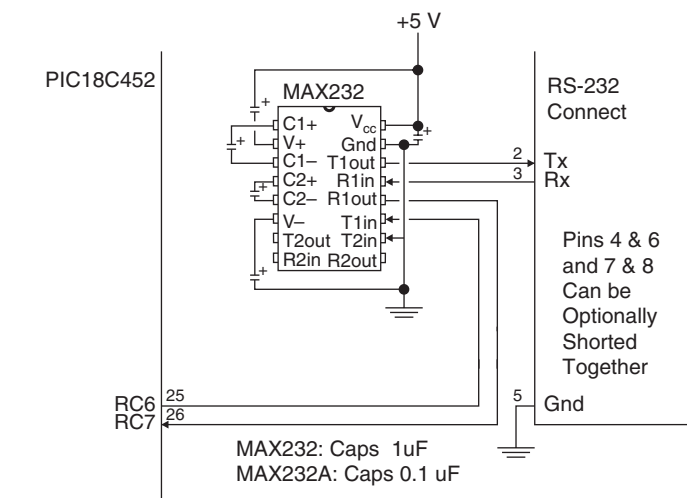


Figure 21.57 The fuzzy logic fan controller uses a MAX232 for the RS-232 interface.

TABLE 21.27 FUZZY LOGIC MOTOR CONTROLLER BILL OF MATERIALS	
REFERENCE DESIGNATOR	DESCRIPTION
PIC16C452	PIC18C452–JW
7805	7805 in TO-220 package
MAX232	MAX232 or MAX232A
IRF510	IRF510 N-channel MOSFET in TO-220 package
OPB804	OPB804 slotted optoisolator switch or equivalent
4 MHz	4 MHz ceramic resonator with built-in capacitors
10k	10 k Ω , 1/4 W resistor
220	220 Ω , 1/4 W resistor
10-k pot	10-k Ω single-turn potentiometer
330- μ F	330 μ F, 35 V electrolytic capacitor
0.1- μ F	0.1 μ F, 16 V tantalum capacitor
LED	Eight individual LEDs or 10 $\times$ bar graph LED display
Misc.	Prototyping board, 8 $\times$ screw terminal, 9-pin female D-shell connector wiring, 12 volt wall-mounted ac/dc power supply

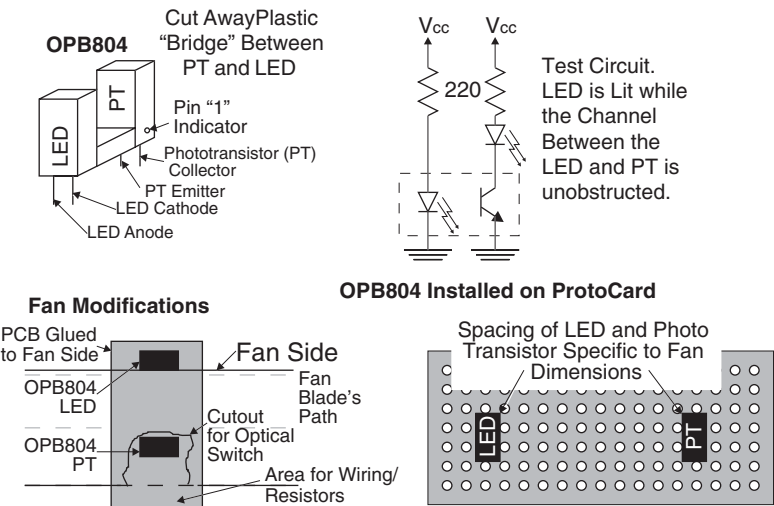


Figure 21.58 The optoisolator is cut apart to create the blade sensor.

To use the OPB804 in this application, carefully cut a notch in the side of the 12-V fan that you are going to use for this application. The notch cannot be cut so far as to interfere with movement of the fan blades and prevent them from turning when the optoisolator switch is installed. For my prototype, I used a nibbler and checked the clearance after every cut. For the fan that I used, I found that I could safely cut out enough material to allow the LED and phototransistor to be within 0.800 in (in the stock OPB805, the distance between the LED and the phototransistor is 0.150 in).

I have a few comments about the OPB804. This part is not critical to operation of the application. If you are going to substitute it for another part, I do have some comments about it, however. The first is to use only an IR slotted switch. I realize that visible-light switches are available, but these should be avoided because of the chance for ambient light reflected off the fan blades to confuse the sensor. The second comment is that you should make sure that the slotted switch that you use will work when the LED and phototransistor are directly apart and there is a direct path between the LED and the phototransistor. Lastly, test the LED and potentiometer on the prototype PCB board before it is mounted on the fan; you don't want to have glued it onto the fan only to discover that it doesn't work.

With the notch cut in the fan, carefully cut away the plastic bridge connecting the LED and phototransistor. There are no signals or circuitry contained within the bridge. When soldering the LED and phototransistor to the prototype board, make sure that they are as square as possible.

You will find that as the distance between the LED and phototransistor increases, the gain of the transistor decreases. When I originally built my prototype, I had an LED connected to the phototransistor, as in the top right-hand corner of Fig. 21.58. When the phototransistor PCB was tested, I found that its gain had decreased markedly, and the output voltage was always above the PIC microcontroller's logic threshold, making counting impossible. But the 10-k Ω pull-up on the OPB804 provided good-quality signals that I could use to count the speed of the fan's rotation.

As I indicated at the start of this section, I wanted to take advantage of the PIC microcontroller's built-in interface hardware for this application; I used the PIC microcontroller's built-in ADC along with a potentiometer wired as a voltage divider to act as a speed control.

To set the power output to the fan, I used the PWM hardware built into TMR2 and the CCP. I used the four times clock multiplier all built into the PIC18C452 to give me a 1-ms instruction cycle and give TMR2 the capability of driving out a 20-kHz PWM with 200 duty cycle levels available.

As an aside, after building this circuit and writing preliminary software for it, I tried it out with a fan that squealed with a 10-kHz PWM control. In this case, there was no squeal, and the fan was almost silent throughout the full speed range. The conclusion I reached from this was that a motor-control PWM should be 20 kHz or more.

To measure the fan speed, I counted the output from the OPB804 IR optoisolator slotted switch with the 10-k Ω pull-up. I used TMR1 because it can be configured as a 16-bit counter, and I had no idea what the speed range of the fan would be (and consequently how many blades would go by in a second).

To better understand what the fan’s speed range would be, I created a simple application to show the analog voltage output by the pot and read by the ADC. This value was scaled to a value between 0 to 199 for easy comparison with the TMR2 CCP PWM control value.

To scale the 10-bit ADC’s output into an 8-bit maximum 199 decimal value, I outputted the ADC values in left-justified mode, which allowed me to read the most significant 8 bits and ignore the least significant 2. To convert the 8-bit value that has the range of 0 to 255 (decimal), I multiplied it by 200 (using the `mullw` instruction) and only used the most significant byte. This is the same as multiplying the value by the fraction 200/256, and it is very easy to implement in the PIC18 microcontroller.

By using the left-justified mode and the multiplication trick, the code required for scaling the 10-bit ADC result to 0–199 was just

```
movf    ADRESH, w
mullw   200
movf    PRODH, w
```

This type of trick will not work for all applications, but knowing these capabilities exist in the PIC18 microcontroller and planning on the numeric ranges that will be required for the application, you can take advantage of them in your design.

When I created the application to find the fan speed, I started working with the ADC and PWM duty cycle and output (via RS-232) both the ADC value as a percentage (which was proportional to the PWM duty cycle) and the fan speed. Table 21.28 lists the results along with what I would consider to be the ideal fan speed. The fan speeds are in blades past per minute (rpm).

TABLE 21.28 FUZZY LOGIC MOTOR CONTROLLER FAN SPEED TABLE		
DUTY CYCLE	FAN SPEED	IDEAL
0%	0	0
10%	360	80
20%	530	160
30%	600	240
40%	660	320
50%	100	400
60%	720	480
70%	750	560
80%	770	640
90%	785	720
100%	800	800

For this application, I decided on a maximum fan speed of 800 blades past per second, and the maximum speed was somewhere around 850. The top speed of 800 blades past per second would give me a margin of control for speeding up the fan. Since there were seven blades on the fan that I used, this translates to 114 revolutions per second, or 6,858 revolutions per minute (rpm).

When I produced the preceding table, I realized something I hadn't before about fuzzy logic: Along with controlling a process or device, it also can be used to *linearize* its response. For this application, the goal was to provide an output using a single input. This (rather belatedly) became the purpose of this project: Provide a fuzzy logic fan motor linearized control as well as governed speed based on the ADC input value.

To create the fuzzy logic rules for this application, I used the Microchip fuzzyTECH fuzzy logic development tool. This product is no longer available from Microchip, but it can be purchased from other sources, or software that mimics the function of the tool can be created quite simply.

To develop a fuzzy logic application, you must create an application that has the structure shown in the following pseudocode:

```
main()                                // Fuzzy Logic Application
{
    initializeIO( );                  // Initialize the I/O

    while (1 == 1) {
        dlay();                      // Delay for the Sample time
        output = fuzzyMod(input);
    }
} // End Fuzzy Logic Application
```

This is the structure I used for this application. Changing the ADC to speed output application into this format was quite simple, but I did run into a few glitches along the way.

The first was with the serial communications. When I first connected the fan controller hardware to fuzzyTECH, I continually got the error “Multiple Data Input,” which I didn't understand. This caused fuzzyTECH to continually shut down and have to be rebooted. When you read the fuzzyTECH manual, you will find that it does not handle invalid serial data very well, and this is the recommended procedure if invalid data is received.

This was perplexing to me because I used both the manual and the fuzzy logic temperature controller as guidelines for the application. I was able to match the fan controller's RS-232 output to the temperature controller's board very closely. I looked at the signal at the PIC microcontroller, and as far as I could tell, it was perfect. After trying to discover the cause of this, I did what I always do in these cases—I slept on it. Sometime in the next day I remembered the operation of the DS275 and that it was a voltage stealer that uses the RX voltage for its TX, and if the RX line is positive, so will be the TX line. I was able to confirm this with an oscilloscope, and I found that every byte from the PC was echoed back to it. To prevent this problem, I replaced the DS275 with the MAX232. After this (laborious) switch, the fuzzyTECH software performed better.

But not perfectly—occasionally, I got the message “Invalid Input.” In the fuzzyTECH manual, it documents the serial data stream as being the decimal representative of the value with an optional + or - at the start, followed by the ASCII decimal representation of the number. Valid numbers are

```
123
+123
-123
+1.23
-1.23
+1.23E-4
```

where in the last example the E represents “10 to the power of.”

I found that if I always sent data with a leading + sign and three data digits, the “Invalid Input” message would go away. In the serial input data handler, I simplified it by not having exponents and by truncating off fractional data—I just handled the integer values of the fuzzyTECH input.

When I first designed this application, I copied the temperature controller application source code that comes with the product almost exactly. This was a mistake based on how the two applications work. The input for the fuzzy controller was the difference between the input control and the motor control PWM. This value, from -20 to +20 was converted into the data string above and passed to fuzzyTECH. The output was a PWM range based on the error rate.

When I started the application, I found that I could not regulate the speed adequately. While not a fully out-of-control situation, I did have wild oscillations that could not be damped down by changing the rules. The difference between the two systems was the amount of inertia that was being controlled in the thermometric example, temperature changes taking place over many seconds; in the fan controller, changes in speed could be implemented in less than a second. While the fuzzy logic controller could keep the oscillations from becoming unbounded, there was still a 20 to 30 percent change in the fan’s speed each second.

The solution to this problem was to change the operation of the fuzzy logic system. While I still retained the input as the difference between the set speed and the actual speed, the output was changed to output differences in the PWM duty cycle from the actual PWM duty cycle. The application code was modeled as

```
main()                                // Fuzzy Logic Fan Motor Controller
{

    initialize();                      // Initialize the Hardware
    PWM = 50 percent;                  // Initialize the PWM to a 50% duty
                                      // cycle

    While (1 == 1) {
        Delay ( );
        Speed - error = ADCinput - countACT;
        PWM = PWM + FuzzyMod(speed - error);
```

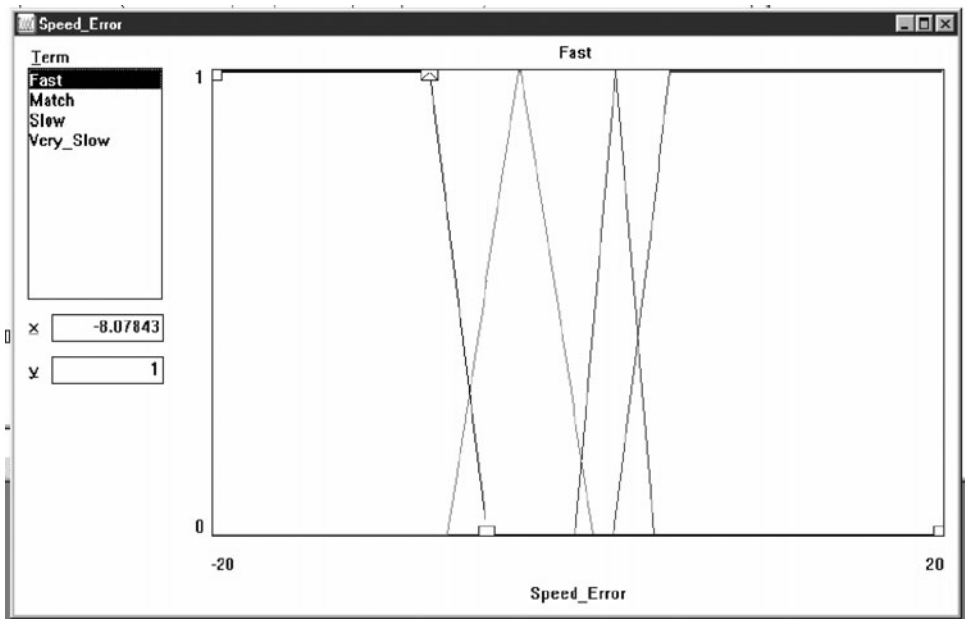



Figure 21.59 Fuzzy logic fan control rules for speed errors.

```

    If (PWM < 0)                // Keep PWM within Valid Range
        PWM = 0;
    else if (PWM > 200)
        PWM = 200;
    }
} // End Fuzzy Logic Fan Motor Controller

```

The speed — error fuzzyTECH input was given the four rules shown in Fig. 21.59. In this set of rules, I identified difference ranges (or patches) for too fast, too slow, and just right. These were matched with the output rules shown in Fig. 21.60. For the input, I used a range of -20 to $+20$, where this value is the difference in counts, to a maximum of 200, between the ADC input (desired speed) and the counter input (actual speed). The change in PWM output is set to the range of -20 to $+20$. Each increase of one digit resulted in about a 1 percent in change of fan speed.

When I started this fuzzy logic system with fuzzyTECH, I found an immediate improvement in the performance of the system. Rather than oscillating between 20 and 30 percent from the target, I found that the motor was regulated to within 8 percent.

While 8 percent was a lot better, I thought I could do even better. My primary strategies were cutting the delay interval by 4 (down to a four times per second sample) and moving the input and output rules. I found that these simple changes resulted in the system being able to keep the fan motor running to within 2 percent; although it occasionally extended to 3 or 4 percent, as you can see in Fig. 21.61. In this figure, the light-colored

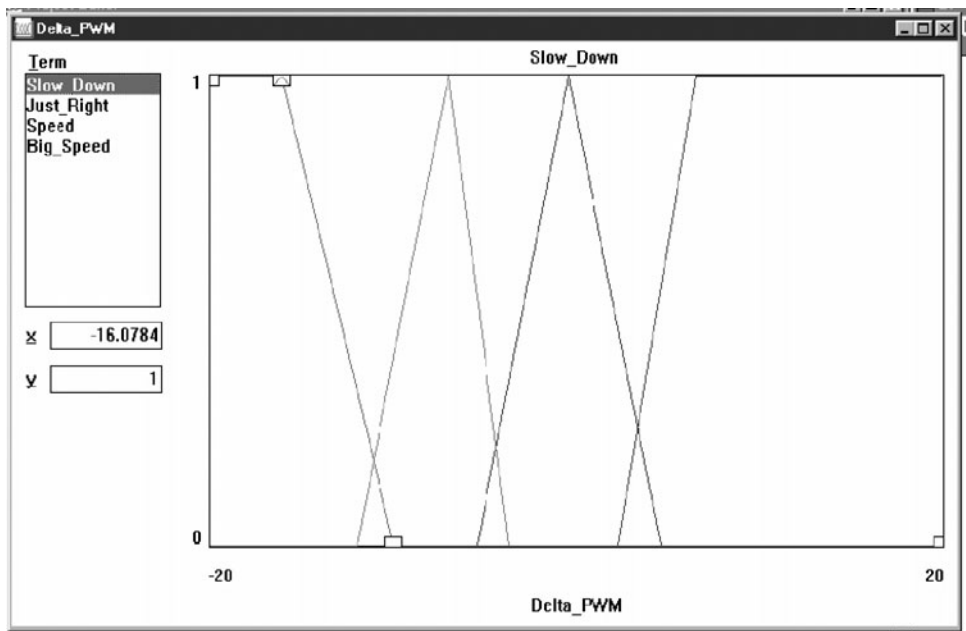


Figure 21.60 Fuzzy fan control PWM rules.

line is the actual speed error, whereas the darker line is the `Delta_PWM`, or change in speed control PWM duty cycle. The changes in the speed control PWM duty cycle are in 0.5 percent duty cycle increments because the total PWM range is 0 to 199.

For this application, this error rate is quite acceptable. You may find that for your own applications, this error rate is either too stringent or too loose. One of the biggest advantages to using fuzzyTECH is that the output's performance can be seen graphically and in real time.

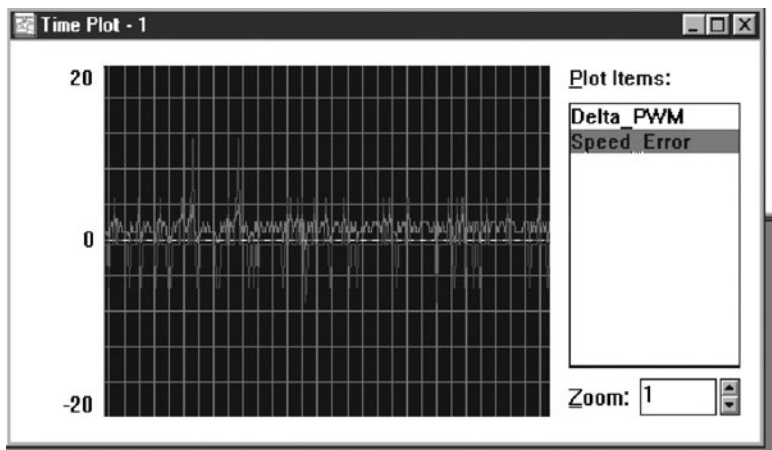


Figure 21.61 Actual fuzzy logic fan controller performance.

The next step in the process was to compile the fuzzy logic system and put the code into the application. I did this to see how well the fuzzy rules would perform in a stand-alone system and how accurate Microchip's statement that the PIC18 was source-code-compatible with the PIC16Cxx (mid-range) devices for which the tool was designed.

I compiled the application and attempted to include it into the `fuzzy2` source. `Fuzzy2ASM` can be found in the `code/fuzzy` folder. After attempting to get the combined source code to assemble without any errors, I gave up with over 200 errors and over 100 warnings left to resolve (and probably many more as I worked through the application).

If you look at the `fuzzy2ASM` source code, you will see that the code works like mid-range source code except for three differences. The first is that I didn't have to worry about register banks for the hardware interface registers. I also kept all the file register variables in the first 128 addresses, which meant that the BSR register never had to be accessed in the application.

The second point to notice is the lack of `goto` instructions. Instead, I used the `branch always` (BRA) because this takes up just one instruction word. It can access all the instructions in the applications easily.

The third change was to replace branches on status flag conditions to branch instructions. An example you'll see in the source code is replacing

```
btfsc STATUS, C  
bra Label
```

with

```
bc Label
```

I found myself often going to the two-instruction format (out of habit), but it was easy to go back and put in the conditional branch statements before I was ready to build the code. This is something that you will have to watch (as well as avoid the `goto` instruction in the PIC18).

This page intentionally left blank

RESOURCES

Microchip

Microchip's corporate headquarters is

Microchip Technology, Inc.
2355 W. Chandler Blvd.
Chandler, AZ 85224
Phone: (480) 786-7200
Fax: (480) 917-4150

The company's website is

www.microchip.com

and contains a complete set of data sheets in .pdf format for download, as well as the latest versions of the MPLAB IDE. Look for the "World Wide Sales and Distribution" page on the website for technical information for your region.

REFERENCE INFORMATION AND TOOLS

As I indicated above, the Microchip website has a wealth of information on PIC[®] microcontroller products, application notes, and articles that can be downloaded free of charge. Also on the website is a link to

<http://buy.microchip.com>

which is Microchip's online ordering system known as Microchip Direct. From this website, you will be able to download the MPLAB IDE and datasheets and order

sample parts (which does have some restrictions). For buying actual PIC microcontroller parts, you will have to contact one of the distributors listed below.

Microchip used to put on a series of seminars throughout the world every year; more recently, the company has created the yearly “Microchip Masters” seminar event in Chandler, Arizona, in which you can get product introductions and tutorials and the chance to meet other developers. I highly recommend going to this event because it is an excellent way to learn more about PIC microcontrollers, meet the field application engineers (FAEs), and find out what is new in the PIC MCU world.

Books to Help You Learn More about the PIC Microcontroller

When the first edition of this book came out, it was one of the PIC microcontroller books available on the market. Here are some others:

123 PIC Microcontroller Experiments for the Evil Genius

Author: M. Predko

ISBN: 0-07-145142-0

Design with PIC Microcontrollers

Author: J. B. Peatman

ISBN: 0-13-759259-0

PIC'n Techniques

Author: D. Benson

ISBN: 0-9654162-3-2

PIC'n Up the Pace

Author: D. Benson

ISBN: 0-9654162-1-6

Serial PIC'n

Author: D. Benson

ISBN: 0-9654162-2-4

Easy PIC'n

Author: D. Benson

ISBN: 0-9654162-0-8

Time'n and Count'n: Using PIC Microcontrollers from Square 1

Author: D. Benson

Publisher: Square 1 Electronics

An Introduction to PIC Microcontrollers

Author: R. A. Penfold

ISBN: 0-85934-394-4

Practical PIC Microcontroller Projects

Author: R. A. Penfold

ISBN: 0-85934-444-4

A Beginners Guide to the Microchip PIC, 2d ed.

Author: N. Gardner

ISBN: 1-899013-01-6

PIC Cookbook

Author: N. Gardner

ISBN: 1-899013-02-4

Useful Books

Here is a collection of books that I have found useful over the years for developing electronics and software for applications. Some of these are hard to find but definitely worth it when you do find them in a used bookstore.

The Art of Electronics (1989)

Authors: P. Horowitz & W. Hill

ISBN: 0-521-37095-7

This is the definitive book on electronics. It's a complete engineering course wrapped up in 1,125 pages. Some people may find it to be a bit too complex, but just about any analog electronics question you could have will be answered in this book. I find that the digital information in this book to be less complete.

Bebop to the Boolean Boogie (1995)

Author: C. Maxwell

ISBN: 1-878707-22-1

This goes somewhat deeper into digital electronics (and less seriously) than *The Art of Electronics*. Clive Maxwell's introduction to electronics stands out with clear and insightful explanations of how things work and why things are done the way they are. I bought my copy when it first became available in 1995 and still use it as a reference when I'm trying to explain how something works. It distinguishes itself from other books by explaining printed wiring assembly (PWA) technology (PCB boards, components, and soldering). This book complements *The Art of Electronics* very nicely.

The Encyclopedia of Electronic Circuits, Vols. 1 to 7

Author: R. Graf

Volume 1 ISBN: 0-8306-1938-0

Volume 2 ISBN: 0-8306-3138-0

Volume 3 ISBN: 0-8306-3348-0

Volume 4 ISBN: 0-8306-3895-4

Volume 5 ISBN: 0-07-011077-8

Volume 6 ISBN: 0-07-011276-2

Volume 7 ISBN: 0-07-015116-4

Rudolf Graf's *Encyclopedia* series on electronic circuits is an excellent resource of circuits and ideas that have been cataloged according to circuit type. Each book contains thousand of circuits and really can make your life easier when you are trying to figure out how to do something. Each volume contains an index listing circuits for the current volume and the previous ones.

CMOS Cookbook (Revised 1988)

Author: D. Lancaster; updated by H. Berlin

ISBN: 0-7506-9943-4

In *CMOS Cookbook*, Don Lancaster introduces the reader to basic digital electronic theory. Also explaining the operation of CMOS gates, he provides hints on soldering and prototyping, lists common CMOS parts (along with TTL pinout equivalents), and provides a number of example circuits (including a good basic definition of how NTSC video works). The update by Howard Berlin has made sure that the chips presented in the book are still available. In the 1970s, Don Lancaster also wrote the *TTL Cookbook* (which also was updated in 1998), but I find the *CMOS Cookbook* to be the most complete and useful for modern applications.

The TTL Data Book for Design Engineers (1988)

I have a couple of copies of the second edition of this book, printed 1981, and they are all falling apart from overuse. The Texas Instruments TTL data books have been used for years by hundreds of thousands of engineers to develop their circuits. Each datasheet is complete with pinouts, operating characteristics, and internal circuit diagrams. While the data books are not complete for the latest HC parts, they will give you just about everything you want to know about the operation of small-scale digital logic. The latest edition I have references for was put out in 1988 and is no longer in print, but you can pick it up in used book stores for relatively modest prices.

The Programmer's PC Source Book, 2d ed. (1991)

Author: T. Hogan

ISBN: 1-55615-118-7

Thom Hogan's 850-page book is just about the best and most complete reference that you can find anywhere on the PC. This book basically ends at the 386 (no 486s, Pentiums of any flavor, PCIs, Northbridges, Southbridges, or SuperIOs or any ASICs of any type), but if you need a basic PC reference that explains BIOS, all the "standard" I/O, DOS, and Windows 3.x interfaces, this is your book. Look for it at your local used bookstore, and if there is a second one, let me know—my copy is falling apart. The only problem with this book is that there are no later editions.

Handbook of Microcontrollers (1998)

Author: M. Predko

ISBN: 0-07-913716-4

I wrote this very thick book as an introduction to and complete reference package for modern 8-bit embedded microcontrollers. As well as providing technical and processor programming information on the Intel 8051, Motorola 68HC05, Microchip PIC microcontroller, Atmel AVR, and Parallax BASIC Stamp, I also have provided datasheets, development tools, and sample applications on the included CD-ROM. To help with your future applications, I explore interfacing to RS-232, I2C, LCD, and other devices, and I devote a fair amount of space to such advanced topics as fuzzy logic, compilers, real-time operating systems (I have included a sample one for the 68HC05), and network communications.

IBM PC and Assembly Language and Programming, 4th ed. (1997)

Author: P. Able

ISBN: 1-1375-6610-7

This is an excellent introduction to assembly-language programming with a fairly low-level approach concentrating on Microsoft's MASM and Borland's TASM. `Debug.com` is used exclusively as a debug tool, which makes this book reasonably inexpensive to get involved with. I bought the first edition in 1983 when I first started working with the PC, and I have kept up with the new editions over the years (largely because the older books fell apart from overuse).

The C Programming Language, 2d ed. (1988)

Authors: B. W. Kernighan and D. M. Ritchie

ISBN: 0-13110-362-8

Brian W. Kernighan and Dennis M. Ritchie's classic text explaining the C programming language has not lost any of its usefulness since its introduction. This book probably has been used by more students in more colleges and universities than any other has. Despite the fact that the book was written originally for a programming course, the writing is crisp, sharp, and easily understandable. This is a book that I've probably owned five copies of, with a couple being worn out and a couple growing legs and walking out of my office.

Recommended PIC Microcontroller Websites

When I have a problem, here is a list of the sites that I turn to in an attempt to find the answers in what other people have already done. Most of these sites are dedicated to the PIC16F84 (and the PIC16C84), but there is a lot of useful information, code, and circuits that you can get from these sites. This list is still current, although it is changing all the time.

As I write this, there is somewhere in the neighborhood of 1,000 web pages devoted to the PIC microcontroller with different applications, code snippets, code development tools, programmers, and other miscellaneous information on the PIC microcontroller and other microcontrollers. To try to simplify your searching, I have included websites that either have unique information and applications or have substantial links to other sites.

My own website has links and information on not only the PIC microcontroller but other devices and subjects as well:

www.myke.com

Bob Blick's website has some interesting PIC microcontroller projects that are quite a bit different from the run-of-the-mill projects:

www.bobblick.com/

Scott Dattalo's highly optimized PIC microcontroller does math algorithms. This is best place to go if you are looking for mathematical functions for the PIC microcontroller:

www.dattalo.com/technical/software/software.html

Along with the very fast PIC microcontroller routines, Scott also has been working on some GNU general purpose license tools designed to run under Linux. More information on these tools can be found in Appendix M. The tools can be downloaded from

www.dattalo.com/gnupic/gpsim.html

www.dattalo.com/gnupic/gpasm.html

Marco Di Leo's PIC Corner has some interesting applications, including information on networking PIC microcontrollers and using them for cryptography:

<http://members.tripod.com/~mdileo/>

On the Dontronics home page, Don McKenzie presents a wealth of information on the PIC microcontroller, as well as other electronic products. There are lots of useful links to other sites, and it is the home of the SimmStick:

www.dontronics.com/

On the Fast Forward Engineering home page, Andrew Warren presents useful information and a highly useful question and answer page on the PIC microcontroller:

<http://home.netcom.com/~fastfwd/>

Steve Lawther's list of PIC microcontroller projects has very many interesting PIC microcontroller (and other microcontroller) projects:

http://ourworld.compuserve.com/homepages/steve_lawther/ucindex.htm

Eric Smith's PIC Page has some interesting projects and code examples to work through:

www.brouhaha.com/~eric/pic/

Rickard's PIC-Wall is a good site with a design for a PIC microcontroller-based composite video game generator:

www.efd.lth.se/~e96rg/pic.html

PicPoint has lots of good projects to choose from, including 5 MB free to anyone that wants to start his or her own PIC microcontroller webpage:

www.picpoint.com/

MicroTronics presents programmer and application reviews:

www.eedevl.com/index.html

Periodicals

Here are a number of magazines that do give a lot of information and projects on PIC microcontrollers. Every month, each magazine has a better than 50% chance of presenting a PIC microcontroller application.

Circuit Cellar Ink

Subscriptions:

P.O. Box 698

Holmes, PA 19043-9613

1 (800) 269-6301

Website: www.circellar.com/

BBS: (860) 871-1988

Nuts & Volts

Subscriptions:

430 Princeland Court

Corona, CA 91719

1 (800) 783-4624

Website: www.nutsvolts.com*Everyday Practical Electronics*

Subscriptions:

EPE Subscriptions Dept.

Allen House, East Borough,

Wimborne, Dorset BH21 1PF

United Kingdom

+44 (0) 1202 881749

Website: www.epemag.wimborne.co.uk

Other Websites of Interest

While none of these are PIC microcontroller-specific, they are a good source of ideas, information, and products that will make your life a bit more interesting and maybe give you some ideas for projects for the PIC microcontroller.

Seattle Robotics Societywww.hhhh.org/srs/

The Seattle Robotics Society has lots of information on interfacing digital devices to such real-world devices such as motors, sensors, and servos. The society also does a lot of exciting things in the automation arena. Most of the applications use the Motorola 68HC11.

List Of Stamp Applications (L.O.S.A)www.hth.com/losa.htm

The List of Parallax BASIC Stamp Applications will give you an idea of what can be done with the BASIC Stamp (and other microcontrollers, such as the PIC microcontroller). The list contains projects ranging from using a BASIC Stamp to give a cat medication to providing a simple telemetry system for model rockets.

Adobe PDF Viewers

www.adobe.com

Adobe .pdf file format is used for virtually all vendor datasheets, including the devices presented in this book (and their datasheets on the CD-ROM).

PKZip and PKUnZip

www.pkware.com

PKWare's zip file compression format is a standard for combining and compressing files for transfer.

Hardware FAQs

http://paranoia.com/~filipg/HTML/LINK/LINK_IN.html

A set of FAQs (frequently asked questions) about the PC and other hardware platforms that will come in useful when interfacing a microcontroller to a host PC.

Innovatus

www.innovatus.com

Innovatus has made available PICBots, an interesting PIC microcontroller simulator that allows programs to be written for virtual robots that will fight among themselves.

Part Suppliers

The following companies supplied components that are used in this book. I am listing them because they all provide excellent customer service and are able to ship parts anywhere you need them.

Digi-Key Corporation

701 Brooks Avenue South

P.O. Box 677

Thief River Falls, MN 56701-0677

Phone: 1 (800) 344-4539 [1 (800) DIGI-KEY]

Fax: (218) 681-3380

www.digi-key.com/

Digi-Key is an excellent source for a wide range of electronic parts. The parts are reasonably priced, and most orders will be delivered the next day. The company is a real lifesaver when you're on a deadline.

Alberta Printed Circuits, Ltd.
No. 3 1112-40th Avenue N.E.
Calgary, Alberta T2E 5T8, Canada
Phone: (403) 250-3406
BBS: (403) 291-9342
E-mail: staff@apcircuits.com
www.apcircuits.com/

AP Circuits will build prototype bare boards from your Gerber files. Boards are available within 3 days. I have been a customer for several years, and the company has always produced excellent quality and been helpful in providing direction to learning how to develop my own bare boards. The website contains the EasyTrax and GCPrevue MS-DOS tools necessary to develop your own Gerber files.

Tower Hobbies
P.O. Box 9078
Champaign, IL 61826-9078
Toll-free ordering in the United States and Canada: 1 (800) 637-4989
Toll-free fax in the United States and Canada: 1 (800) 637-7303
Toll-free support in the United States and Canada: 1 (800) 637-6050
Phone: (217) 398-3636
Fax: (217) 356-6608
E-mail: orders@towerhobbies.com
www.towerhobbies.com/
Excellent source for servos and R/C parts useful in homebuilt robots.

Jameco
1355 Shoreway Road
Belmont, CA 94002-4100
Toll-free in the United States and Canada: 1 (800) 831-4242
www.jameco.com/
Components, PC Parts/Accessories, and hard to find connectors.

JDR Microdevices
1850 South 10th Street
San Jose, CA 95112-4108
Toll-free in the United States and Canada: 1 (800) 538-500
Toll-free fax in the United States and Canada: 1 (800) 538-5005
Phone: (408) 494-1400
E-mail: techsupport@jdr.com
BBS: (408) 494-1430
Compuserve: 70007,1561
www.jdr.com/JDR
Supplies components, PC parts/accessories, and hard-to-find connectors.

Newark
4801 N. Ravenswood
Chicago, IL 60640-4496
(773) 784-5100
Fax: (888) 551-4801
Toll-free in the United States and Canada: 1 (800) 463-9275 [1 (800) 4-NEWARK]
www.newark.com/
Supplies components, including the Dallas line of semiconductors (the DS87C520 and DS275 is used for RS-232 level conversion in this book).

Marshall Industries
9320 Telstar Avenue
El Monte, CA 91731
1 (800) 833-9910
www.marshall.com
Marshall is a full-service distributor of Philips microcontrollers as well as other parts.

Mouser Electronics, Inc.
958 North Main Street
Mansfield, Texas 76063
Sales: (800) 346-6873
Sales: (817) 483-6888
Fax: (817) 483-6899
E-mail: sales@mouser.com
www.mouser.com
Mouser is the distributor for the Seiko S7600A TCP/IP stack chips.

Mondo-tronics Robotics Store
Mondo-tronics, Inc.
524 San Anselmo Ave., No. 107-13
San Anselmo, CA 94960
Toll-free in the United States and Canada: 1 (800) 374-5764
Fax: (415) 455-9333
www.robotstore.com/
Self-proclaimed as “the world’s biggest collection of miniature robots and supplies,” and I have to agree with them. This is a great source for servos, tracked vehicles, and robot arms.

This page intentionally left blank

PIC MICROCONTROLLER SUMMARY

When selecting a PIC[®] microcontroller, there are four things to look for:

- 1** Which PIC MCU part number has all the features that are required by the application
- 2** An instruction set summary that will give the instructions in a condensed format so that it is easy to understand what is happening in each instruction
- 3** A list of the standard register addresses for PIC microcontroller devices and how their bits are defined
- 4** ASCII tables along with an explanation of the corresponding C control characters

This appendix contains much of the necessary PIC microcontroller reference information, and Appendix C provides some reference information that you should find useful.

Feature to Part Number Table

The Table B.1 lists some of the different PIC microcontroller families and the features that are specific to them.

For the PIC16C6x and PIC16C7x devices, the Table B.2 can be used to differentiate the features.

Instruction Sets

The following tables explain the instructions available to the different PIC microcontroller processor architectures algorithmically. This is another way of presenting the instructions, along with visualizing the data flow within the PIC microcontroller's

TABLE B.1 PIC MICROCONTROLLER NUMBER TO FEATURE TABLE

PART NUMBER	FEATURES
PIC10F2xx	6-pin PIC microcontroller, 12-bit (low-end) processor, internal reset and oscillator, optional comparator or ADC
PIC12C5xx	8-pin PIC MCU, 12-bit (low-end) processor, internal reset and oscillator
PIC12F5xx	8-pin PIC microcontroller, 12-bit (low-end) processor, internal reset and oscillator, optional comparator
PIC12C6xx	8-pin PIC microcontroller, 14-bit (mid-range) processor, 8-bit ADC/internal reset and oscillator/optional EEPROM data memory
PIC14000	28-pin PIC MCU, 14-bit (mid-range) processor, advanced ADC/internal voltage reference/on-chip temperature sensor
PIC16C5x	18- to 28-pin PIC microcontroller, 12-bit (low-end) processor
PIC16F5x	18- to 28-pin PIC microcontroller, 12-bit (low-end) processor, Flash program memory
PIC16C505	14-pin PIC MCU, 12-bit (low-end) processor, internal reset and oscillator
PIC16F50x	14-pin PIC MCU, 12-bit (low-end) processor, internal reset and oscillator, Flash program memory, optional comparator
PIC16C6x	18- to 40-pin PIC microcontroller, 14-bit (mid-range) processor, optional TMR1 and TMR2/optional SPI/optional USART/optional PSP
PIC16C62x	18-pin PIC MCU, 14-bit (mid-range) processor, voltage comparators built in with voltage reference/optional EEPROM data memory
PIC16F62x	18-pin PIC microcontroller, 14-bit (mid-range) processor, Flash program memory/voltage comparators built in with voltage reference/internal reset and oscillator
PIC16F63x	14-pin PIC microcontroller, 14-bit (mid-range) processor, Flash program memory, optional ADC and comparators, CCP module
PIC16C642	28-pin PIC MCU, 14-bit (mid-range) processor, voltage comparators built in with voltage reference
PIC16C662	40-pin PIC microcontroller, 14-bit (mid-range) processor, voltage comparators built in with voltage reference
PIC16F69x	20-pin PIC MCU, 14-bit (mid-range) processor, Flash program memory, ADC, comparator and CCP module
PIC16C71x	18-pin PIC MCU, 14-bit (mid-range) processor, 8-bit ADC
PIC16C7x	18- to 40-pin PIC microcontroller, 14-bit (mid-range) processor, 8-bit ADC/optional TMR1 and TMR2/optional SPI/optional USART/optional PSP
PIC16C77x	28- to 40-pin PIC MCU, 14-bit (mid-range) processor, 12-bit ADC/TMR1 and TMR2/USART/I2C/SPI/optional PSP
PIC16F8x	18-pin PIC microcontroller, 14-bit (mid-range) processor, Flash data and program memory

TABLE B.1 PIC MICROCONTROLLER NUMBER TO FEATURE TABLE (CONTINUED)

PART NUMBER	FEATURES
PIC16F87x	28- to 40-pin PIC MCU, 14-bit (mid-range) processor, 10-bit ADC/TMR1 and TMR2/USART/I2C/SPI/optional PSP. MPLAB ICD enabled
PIC16C92x	64-pin PIC microcontroller, 14-bit (mid-range) processor, optional 8-bit ADC/TMR1 and TMR2/LCD controller
PIC18Cx01	64- to 84-pin PIC microcontroller, 16-bit (PIC18) processor, ROMless, USART/I2C/SPI/ADC
PIC18Cxx2	28- to 40-pin PIC MCU, 16-bit advanced (PIC18) processor, USART/I2C/SPI/10-bit ADC
PIC18Fx420	28- to 64-pin PIC microcontroller, 16-bit advanced (PIC18) processor, Flash program memory, USART/I2C/SPI/ADC
PIC18Fx550	28- to 84-pin PIC microcontroller, 16-bit advanced (PIC18) processor, Flash program memory, USB interface, SART/I2C/SPI/ADC

processor or listing the instructions, as is done in the Microchip datasheets. Unless otherwise noted, all instructions execute in one instruction cycle.

LEGEND

Along with the instructions, there are a number of parameters that are needed for the instructions, and these are listed in Table B.3.

TABLE B.2 MID-RANGE PIC MICROCONTROLLER PART NUMBER TO FEATURE BREAKOUT

PART NUMBER	COMMENTS
16Cx1	18-pin PIC microcontroller, 1-kB program memory, no USART/SPI/I2C; PIC16C61 and PIC16C71 are currently "obsoleted."
16Cx2	28-pin PIC microcontroller, 2-kB program memory, SPI/TMR1 and TMR2
16Cx3	28-pin PIC microcontroller, 4-kB program memory, USART/SPI/TMR1 and TMR2
16Cx4	40-pin PIC microcontroller, 4-kB program memory, USART/SPI/PSP/TMR1 and TMR2
16Cx5	40-pin PIC microcontroller, 4-kB program memory, USART/SPI/PSP/TMR1 and TMR2
16Cx6	28-pin PIC microcontroller, 8-kB program memory, USART/SPI/I2C/TMR1 and TMR2
16Cx7	40-pin PIC microcontroller, 8-kB program memory, USART/SPI/I2C/PSP/TMR1 and TMR2

TABLE B.3 INSTRUCTION SYMBOL DEFINITION			
PARAMETER	SYMBOL	OP CODE LETTER	VALUE RANGE
Don't care	N/A	x	
Byte constant	k	k	0 to 0xFF
Register address	Reg	f	PIC microcontroller architecture-specific
Destination	d	d	0 or 1
Selection bit	Bit	b	0 to 7
Destination address	Address	a	0 to 0x7FF
Destination port	Port	p	PORTA (address 0x05) to PORTC (address 0x07)

LOW-END INSTRUCTION SET

Register banks are 32 bytes in size in low-end devices. This makes Reg in Table B.3 in the range of 0 to 0x01F (see Table B.4).

MID-RANGE INSTRUCTION SET

Register banks are 128 bytes in size in low-end devices. This makes Reg in the range of 0 to 0x07F (see Table B.5).

PIC18 INSTRUCTION SET

Microchip, in the PIC18 microcontroller's datasheets, claims that the instruction set is source code-compatible with the mid-range's instruction set. While many of the instructions are similar between the mid-range and PIC18, they really aren't directly compatible owing to the addition of the *a* bit for many of the instructions and the slightly different operation of some of the instructions that have been given the same label (see Table B.6).

MICROCHIP SPECIAL INSTRUCTION MNEMONICS

The following special instructions are macros built into MPASM by Microchip to help make some instructions more intuitive. These instructions are built into MPASM, and their labels *never* should be used for macros, addresses (code or variable), or defines.

Most of these special instructions are made up of one or more standard low-end or mid-range PIC microcontroller instructions. Note that some of these special instructions may change the value of the zero flag (see Table B.7).

LCALL should never be used because the PCLATH bits are not returned to the appropriate value for the code following LCALL. When a goto or call is executed

TABLE B.4 LOW-END PIC MICROCONTROLLER INSTRUCTION SET

DESCRIPTION	INSTRUCTION	OPERATION	OP CODE
Add register contents to the w register and optionally store result in the w register	<code>addwf Reg, d</code>	<pre> if (d == 1) Reg = Reg + w else w = Reg + w endif C = (Reg + w) > 0xFF Z = ((Reg + w) & 0xFF) == 0 DC = ((Reg & 0x0F) + (w & 0x0F)) > 0x0F </pre>	0001 11df ffff
AND immediate with the w register	<code>andlw k</code>	<pre> w = w & k Z = (w & k) == 0 </pre>	1110 kkkk kkkk
AND register contents with the w register and optionally store result in the w register	<code>andwf Reg, d</code>	<pre> if (d == 1) Reg = Reg & w else w = Reg & w endif Z = (Reg & w) == 0 </pre>	0001 01df ffff
AND register contents with the w register and optionally store result in the w register	<code>andwf Reg, d</code>	<pre> if (d == 1) Reg = Reg & w else w = Reg & w endif Z = (Reg & w) == 0 </pre>	0001 01df ffff
Clear the specified bit in the register	<code>bcf Reg, bit</code>	<code>Reg = Reg & (0xFF ^ (1 << Bit))</code>	0100 bbbf ffff
Set the specified bit in the register	<code>bsf Reg, bit</code>	<code>Reg = Reg (1 << Bit)</code>	0101 bbbf ffff
Skip if the specified bit in the register is clear; one instruction cycle if skip not executed, two if it is	<code>btfsc Reg, bit</code>	<pre> if ((Reg & (1 << Bit))) == 0) PC = PC + 1 endif </pre>	0110 bbbf ffff

(Continued)

TABLE B.4 LOW-END PIC MICROCONTROLLER INSTRUCTION SET (CONTINUED)

DESCRIPTION	INSTRUCTION	OPERATION	OP CODE
Skip if the specified bit in the register is set; one instruction cycle if skip not executed, two if it is	btfsc Reg, bit	if ((Reg & (1 << Bit)) != 0) PC = PC + 1 Endif	0111 bbbf ffff
Save the stack pointer and jump to the specified address (two instruction cycles)	call Address	[SP] = PC SP = SP + 1 PC = ((STATUS & 0xE0) << 4) + Address	1001 aaaa aaaa
Clear the specified register	clrf Reg	Reg = 0 Z = 1	0000 011f ffff
Clear the w register	clrw	w = 0 Z = 1	0000 0100 0000
Clear the watchdog timer's counter	clrwdt	WDT = 0 _TO = 1 _PD = 1	0000 0000 0100
Complement the contents of the specified register and optionally store the results in the w register	comf Reg, d	if (d == 1) Reg = Reg ^ 0xFF else w = Reg ^ 0xFF endif Z = (Reg ^ 0xFF) == 0	0010 01df ffff
Decrement the contents of the register and optionally store the results in the w register	decf Reg, d	if (d == 1) Reg = Reg - 1 else w = Reg - 1 endif Z = (Reg - 1) == 0	0011 11df ffff

Decrement the contents of the register and optionally store the results in the w register and skip the next instruction if the results are equal to zero; two instruction cycles taken if skip executed	<code>decfsz Reg, d</code>	<pre> if (d == 1) Reg = Reg - 1 else w = Reg - 1 endif PC = PC + 1 if ((Reg - 1) == 0) PC = PC + 1 Endif </pre>	0010 11df ffff
Jump to the specified address (two instruction cycles)	<code>goto Address</code>	<code>PC = (((STATUS & 0xE0) << 4) + Address</code>	101a aaaa aaaa
Increment the contents of the register and optionally store the results in the w register	<code>incf Reg, d</code>	<pre> if (d == 1) Reg = Reg + 1 else w = Reg + 1 endif Z = (Reg + 1) == 0 </pre>	0010 10df ffff
Increment the contents of the register and optionally store the results in the w register and skip the next instruction if the results are equal to zero; two instruction cycles taken if skip executed	<code>incfsz Reg, d</code>	<pre> if (d == 1) Reg = Reg + 1 else w = Reg + 1 endif PC = PC + 1 if ((Reg + 1) == 0) PC = PC + 1 endif </pre>	0011 11df ffff
OR immediate with the w register	<code>iorlw k</code>	<pre> w = w k Z = (w k) == 0 </pre>	1101 kkkk kkkk
OR register contents with the w register and optionally store result in the w register	<code>iorwf Reg, d</code>	<pre> if (d == 1) Reg = Reg w else w = Reg w endif Z = (Reg w) == 0 </pre>	0001 00df ffff

(Continued)

TABLE B.4 LOW-END PIC MICROCONTROLLER INSTRUCTION SET (CONTINUED)

DESCRIPTION	INSTRUCTION	OPERATION	OP CODE
Check register contents equal to zero and optionally store result in the w register	movf Reg, d	if (d == 0) w = Reg endif Z = Reg == 0	0010 00df ffff
Load w register with an immediate value	movlw k	w = k	1100 kkkk kkkk
Store the value in the w register	movwf Reg	Reg = w	0000 001f ffff
Waste one instruction cycle	nop		0000 0000 0000
Move the contents of the w register into the OPTION register	option	OPTION = w	0000 0000 0010
Resume execution after subroutine and place a constant value in the w register (two cycles used)	retlw k	w = k SP = SP - 1 PC = [SP]	1000 kkkk kkkk
Resume execution after subroutine and place zero in the w register (two cycles used); this is actually a retlw 0 instruction that MPLAB provides	return	w = 0 SP = SP - 1 PC = [SP]	1000 0000 0000
Rotate the register left through carry and optionally save the result in the w register	rlf Reg, d	Temp = C C = (Reg >> 7) & 1 if (d == 1) Reg = (Reg << 1) + Temp else w = (Reg << 1) + Temp endif	0011 01df ffff
Rotate the register right through carry and optionally save the result in the w register	rrf Reg, d	Temp = C C = Reg & 1 if (d == 1) Reg = (Reg >> 1) + (Temp << 7)	0011 00df ffff

		<pre> else w = (Reg >> 1) + (Temp << 7) endif </pre>	
Go into standby mode (indeterminate number of cycles before execution resumes)	sleep	<pre> _TO = 1 _PD = 0 </pre>	0000 0000 0011
Subtract w register contents from register and store result in the w register	subwf Reg, d	<pre> if (d == 1) Reg = Reg + (w ^ 0xFF) + 1 else w = Reg + (w ^ 0xFF) + 1 endif C = (Reg + (w ^ 0xFF) + 1) > 0xFF Z = ((Reg + (w ^ 0xFF) + 1) & 0xFF) == 0 DC = ((Reg & 0x0F) + ((w ^ 0xFF) & 0x0F) + 1) > 0x0F </pre>	0000 10df ffff
Swap the upper and lower nibbles of a register and optionally store result in the w register	swapf Reg, d	<pre> if (d == 1) Reg = ((Reg & 0xF0) >> 4) + ((Reg & 0x0F) << 4) else w = ((Reg & 0xF0) >> 4) +((Reg & 0x0F) << 4) endif </pre>	0011 10df ffff
Move the contents of the w register into the tristate control register of the port	tris Port	<pre> TRIS(Port) = w </pre>	0000 0000 0ppp
XOR immediate with the w register	xorlw k	<pre> w = w ^ k Z = (w ^ k) == 0 </pre>	1111 kkkk kkkk
XOR register contents with the w register and optionally store result in the w register	xorwf Reg, d	<pre> if (d == 1) Reg = Reg ^ w else w = Reg ^ w endif Z = (Reg ^ w) == 0 </pre>	0001 10df ffff

TABLE B.5 MID-RANGE PIC MICROCONTROLLER INSTRUCTION SET

DESCRIPTION	INSTRUCTION	OPERATION	OP CODE
Add immediate to the w register	<code>addlw k</code>	$w = w + k$ $C = (w + k) > 0xFF$ $Z = ((w + k) \& 0xFF) == 0$ $DC = ((w \& 0x0F) + (k \& 0x0F)) > 0x0F$	11 111x kkkk kkkk
Add register contents to the w register and optionally store result in the w register	<code>addwf Reg, d</code>	$\text{if } (d == 1)$ $\text{Reg} = \text{Reg} + w$ else $w = \text{Reg} + w$ endif $C = (\text{Reg} + w) > 0xFF$ $Z = ((\text{Reg} + w) \& 0xFF) == 0$ $DC = ((\text{Reg} \& 0x0F) + (w \& 0x0F)) > 0x0F$	00 0111 dfff ffff
AND immediate with the w register	<code>andlw k</code>	$w = w \& k$ $Z = (w \& k) == 0$	11 1001 kkkk kkkk
AND register contents with the w register and optionally store result in the w register	<code>andwf Reg, d</code>	$\text{if } (d == 1)$ $\text{Reg} = \text{Reg} \& w$ else $w = \text{Reg} \& w$ endif $Z = (\text{Reg} \& w) == 0$	00 0101 dfff ffff
Clear the specified bit in the register	<code>bcf Reg, bit</code>	$\text{Reg} = \text{Reg} \& (0xFF \wedge (1 \ll \text{Bit}))$	01 00bb bfff ffff
Set the specified bit in the register	<code>bcf Reg, bit</code>	$\text{Reg} = \text{Reg} \mid (1 \ll \text{Bit})$	01 01bb bfff ffff
Skip if the specified bit in the register is clear; one instruction cycle if skip not executed, two if skip executed	<code>btfsc Reg, bit</code>	$\text{if } ((\text{Reg} \& (1 \ll \text{Bit}))) == 0$ $\text{PC} = \text{PC} + 1$ endif	01 10bb bfff ffff

Skip if the specified bit in the register is set; one instruction cycle if skip not executed, two if it is	btfsc Reg, bit	if ((Reg & (1 << Bit)) != 0) PC = PC + 1 endif	01 11bb bfff ffff
Save the stack pointer and jump to the specified address (two instruction cycles)	call Address	[SP] = PC SP = SP + 1 PC = ((PCLATH << 8) & 0x1800) + Address	10 0aaa aaaa aaaa
Clear the specified register	clrf Reg	Reg = 0 Z = 1	00 0001 1fff ffff
Clear the w register	clrw	w = 0 Z = 1	00 0001 0xxx xxxx
Clear the watchdog timer's counter	clrwdt	WDT = 0 _TO = 1 _PD = 1	00 0000 0110 0100
Complement the contents of the specified register and optionally store the results in the w register	comf Reg, d	if (d == 1) Reg = Reg ^ 0xFF else w = Reg ^ 0xFF endif Z = (Reg ^ 0xFF) == 0	00 1001 dfff ffff
Decrement the contents of the register and optionally store the results in the w register	decf Reg, d	if (d == 1) Reg = Reg - 1 else w = Reg - 1 endif Z = (Reg - 1) == 0	00 0011 dfff ffff
Decrement the contents of the register and optionally store the results in the w register and skip the next instruction if the results are equal to zero; two instruction cycles taken if skip executed	decfsz Reg, d	if (d == 1) Reg = Reg - 1 else w = Reg - 1 endif if ((Reg - 1) == 0) PC = PC + 1 Endif	00 1011 dfff ffff

(Continued)

TABLE B.5 MID-RANGE PIC MICROCONTROLLER INSTRUCTION SET (CONTINUED)

DESCRIPTION	INSTRUCTION	OPERATION	OP CODE
Jump to the specified address (two instruction cycles)	goto Address	PC = (PCLATH << 8) & 0x1800) + Address	10 1aaa aaaa aaaa
Increment the contents of the register and optionally store the results in the w register	incf Reg, d	if (d == 1) Reg = Reg + 1 else w = Reg + 1 endif Z = (Reg + 1) == 0	00 1010 dfff ffff
Increment the contents of the register and optionally store the results in the w register, and skip the next instruction if the results are equal to zero; two instruction cycles taken if skip executed	incfsz Reg, d	if (d == 1) Reg = Reg + 1 else w = Reg + 1 endif if ((Reg + 1) == 0) PC = PC + 1 endif	00 1111 dfff ffff
OR immediate with the w register	iorlw k	w = w k Z = (w k) == 0	11 1000 kkkk kkkk
OR register contents with the w register and optionally store result in the w register	iorwf Reg, d	if (d == 1) Reg = Reg w else w = Reg w endif Z = (Reg w) == 0	00 0100 dfff ffff
Check register contents equal to zero and optionally store register contents in the w register	movf Reg, d	if (d == 0) w = Reg endif Z = Reg == 0	00 1000 dfff ffff
Load w register with an immediate value	movlw k	w = k	11 00xx kkkk kkkk
Store the value in the w register	movwf Reg	Reg = w	00 0000 1fff ffff

Waste one instruction cycle	<code>nop</code>		00 0000 0xx0 0000
Move the contents of the w register into the OPTION register; use of this instruction is not recommended	<code>option</code>	OPTION = w	00 0000 0110 0010
Resume execution after interrupt (two cycles used)	<code>retfie</code>	GIE = 1 SP = SP - 1 PC = [SP]	00 0000 0000 1001
Resume execution after subroutine, and place a constant value in the w register (two cycles used)	<code>retlw k</code>	w = k SP = SP - 1 PC = [SP]	11 01xx kkkk kkkk
Resume execution after subroutine (two cycles used)	<code>return</code>	SP = SP - 1 PC = [SP]	00 0000 0000 1000
Rotate the register left through carry and optionally save the result in the w register	<code>rlf Reg, d</code>	Temp = C C = (Reg >> 7) & 1 if (d == 1) Reg = (Reg << 1) + Temp else w = (Reg << 1) + Temp endif	00 1101 dfff ffff
Rotate the register right through carry and optionally save the result in the w register	<code>rrf Reg, d</code>	Temp = C C = Reg & 1 if (d == 1) Reg = (Reg >> 1) + (Temp << 7) else w = (Reg >> 1) + (Temp << 7) endif	00 1100 dfff ffff
Go into standby mode (indeterminate number of cycles used)	<code>sleep</code>	_TO = 1 _PD = 0	00 0000 0110 0011
Subtract the w register contents from immediate and store the result in the w register	<code>sublw k</code>	w = k + (w ^ 0xFF) + 1 C = (k + (w ^ 0xFF) + 1) > 0x0F Z = ((k + (w ^ 0xFF) + 1) & 0xFF) == 0 DC = ((k & 0x0F) + ((w ^ 0xFF) & 0x0F) + 1) > 0x0F	11 110x kkkk kkkk

(Continued)

TABLE B.5 MID-RANGE PIC MICROCONTROLLER INSTRUCTION SET (CONTINUED)

DESCRIPTION	INSTRUCTION	OPERATION	OP CODE
Subtract the w register contents from register and optionally store result in the w register	subwf Reg, d	<pre> if (d == 1) Reg = Reg + (w ^ 0xFF) + 1 else w = Reg + (w ^ 0xFF) + 1 endif C = (Reg + (w ^ 0xFF) + 1) > 0xFF Z = ((Reg + (w ^ 0xFF) + 1) & 0xFF) == 0 DC = ((Reg & 0x0F) + ((w ^ 0xFF) & 0x0F) + 1) > 0x0F </pre>	00 0010 dfff ffff
Swap the upper and lower nibbles of a register and optionally store result in the w register	swapf Reg, d	<pre> if (d == 1) Reg = ((Reg & 0xF0) >> 4) + ((Reg & 0x0F) << 4) else w = ((Reg & 0xF0) >> 4) + ((Reg & 0x0F) << 4) endif </pre>	00 1110 dfff ffff
Move the contents of the w register into the tristate control register of the port; use of this instruction is not recommended	tris Port	TRIS(Port) = w	00 0000 0110 0ppp
XOR immediate with the w register	xorlw k	<pre> W = w ^ k Z = (w ^ k) == 0 </pre>	11 1010 kkkk kkkk
XOR register contents with the w register and optionally store results in the w register	xorwf Reg, d	<pre> If (d == 1) Reg = Reg ^ w else w = Reg ^ w endif Z = (Reg ^ w) == 0 </pre>	00 0110 dfff ffff

TABLE B.6 PIC18C INSTRUCTION SET

INSTRUCTION	FORMAT	OPERATION	BIT PATTERN
Add a constant to the w register and store the result in the w register	addlw Constant	<pre> wreg = wreg + Constant if ((wreg > 0) & (Constant > 0)) & (wreg + Constant) > 0x07F)) OV = 1 else OV = 0 endif if ((wreg + Constant) > 0x0FF) C = 1 else C = 0 endif if ((wreg + Constant) & 0x080) != 0) N = 1 else N = 0 endif if (((wreg & 0x0F) + (Constant & 0x0F)) > 0x0F) DC = 1 else DC = 0 endif if (((wreg + Constant) & 0x0FF) == 0x000) Z = 1 else Z = 0 endif </pre>	0000 1111 kkkk kkkk

(Continued)

TABLE B.6 PIC18C INSTRUCTION SET (CONTINUED)

INSTRUCTION	FORMAT	OPERATION	BIT PATTERN
Add w register to the contents of Reg and store the result according to “d” result in w register; if “a” is set, then BSR used for Reg, else access bank is used.	addwf Reg, d, a	<pre> if (“d” == 1) wreg = wreg + Reg else Reg = wreg + Reg endif if (((wreg > 0) & (Reg > 0)) & ((wreg + Reg) > 0x07F)) OV = 1 else OV = 0 endif if ((wreg + Reg) & 0x080) != 0) N = 1 else N = 0 endif if ((wreg + Reg) > 0x0FF) C = 1 else C = 0 endif if ((wreg & 0x0F) + Reg & 0x0F)) > 0x0F) DC = 1 else DC = 0 endif if (((wreg + Reg) & 0x0FF) == 0x000) Z = 1 else Z = 0 endif </pre>	0010 01da ffff ffff

Add w register to the contents of Reg and C, store the result according to “d” result in w register; if “a” is set, then Reg is in the BSR bank, else Reg is in the access bank

addwfc Reg, d,a

```

if ("d" == 1)
    wreg = wreg + Reg + C
else
    Reg = wreg + Reg + C
endif
if (((wreg > 0) & (Reg > 0))
    & ((wreg + Reg + C)
    > 0x07F))
    OV = 1
else
    OV = 0
endif
if ((wreg + Reg + C)
    & 0x080) != 0)
    N = 1
else
    N = 0
endif
if ((wreg + Reg + C)
    > 0x0FF)
    C = 1
else
    C = 0
endif
if (((wreg & 0x0F) + Reg
    & 0x0F) + C) > 0x0F)
    DC = 1
else
    DC = 0
endif
    if (((wreg + Reg + C)
    & 0x0FF) == 0x000)
        Z = 1
    else
        Z = 0
    endif

```

0010 00da ffff
ffff

TABLE B.6 PIC18C INSTRUCTION SET (CONTINUED)

INSTRUCTION	FORMAT	OPERATION	BIT PATTERN
AND a constant to the w register and store the result in the w register	andlw Constant	<pre>wreg = wreg & Constant if ((wreg & Constant) == 0x000) Z = 1 else Z = 0 endif if ((wreg & Constant) & 0x080) != 0) N = 1 else N = 0 endif</pre>	0000 1011 kkkk kkkk
AND w register to the contents of Reg and store the result according to "d" result in w register; if "a" is set, then Reg is in the BSR bank, else it is in the access bank	andwf Reg, d, a	<pre>if ("d" == 1) wreg = wreg & Reg else Reg = wreg & Reg endif if ((wreg & Reg) = 0x000) Z = 1 else Z = 0 endif if ((wreg & Reg) & 0x080) != 0) N = 1 else N = 0 endif</pre>	0001 01da ffff ffff

Branch if the carry flag is set; label is a two's complement offset; one instruction cycle if branch not executed, two if branch executed	bc Label	if (C == 1) PC = PC + 2 + Label endif	1110 0010 kkkk kkkk
Clear the specified bit in Reg; if "a" is set, then Reg is in the BSR bank, else Reg is in the access bank	bcf Reg, Bit, a	Reg = Reg & (0xFF ^ 1 << Bit)	1001 bbba ffff ffff
Branch if negative flag is set; label is a two's complement offset; one instruction cycle if branch not executed, two if branch executed	bn Label	if (N == 1) PC = PC + 2 + Label endif	1110 0110 kkkk kkkk
Branch if the carry flag is reset; label is a two's complement offset; one instruction cycle if branch not executed, two if branch executed	bnc Label	if (C == 0) PC = PC + 2 + Label endif	1110 0011 kkkk kkkk
Branch if negative flag is reset; label is a two's complement offset; one instruction cycle if branch not executed, two if branch executed	bnn Label	if (N == 0) PC = PC + 2 + Label endif	1110 0111 kkkk kkkk
Branch if overflow flag is reset; label is a two's complement offset; one instruction cycle if branch not executed, two if branch executed	bnov Label	if (OV == 0) PC = PC + 2 + Label endif	1110 0101 kkkk kkkk
Branch if the zero flag is reset; label is a two's complement offset; one instruction cycle if branch not executed, two if branch executed	bnz Label	if (Z == 0) PC = PC + 2 + Label endif	1110 0001 kkkk kkkk
Branch if overflow flag is set; label is a two's complement offset; one instruction cycle if branch not executed, two if branch executed	bov Label	if (OV == 1) PC = PC + 2 + Label endif	1110 0100 kkkk kkkk

(Continued)

TABLE B.6 PIC18C INSTRUCTION SET (CONTINUED)

INSTRUCTION	FORMAT	OPERATION	BIT PATTERN
Branch always; label is a two's complement offset; two instruction cycles	<code>bra Label</code>	$PC = PC + 2 + \text{Label}$	1110 0kkk kkkk kkkk
Set the specified bit in Reg; if "a" is set, then Reg is in the BSR bank, else Reg is in the access bank	<code>bsf Reg, Bit, a</code>	$\text{Reg} = \text{Reg} \mid (1 \ll \text{Bit})$	1000 bbba ffff ffff
Test the specified bit in Reg and skip if clear; if "a" is set, then the BSR is used for Reg, else the access bank is used; one instruction cycle if skip not executed, two if skip executed	<code>btfsc Reg, Bit, a</code>	<pre>if ((Reg & (1 << Bit)) == 0) PC = NextIns endif</pre>	1011 bbba ffff ffff
Test the specified bit in Reg and skip if set; if "a" is set, then the BSR is used for Reg, else the access bank is used; one instruction cycle if skip not executed, two if skip executed.	<code>btfss Reg, Bit, a</code>	<pre>if ((Reg & (1 << Bit)) != 0) PC = NextIns endif</pre>	1010 bbba ffff ffff
Toggle the specified bit in Reg; if "a" is set, then the BSR is used for Reg, else the access bank is used	<code>btg Reg, Bit, a</code>	$\text{Reg} = \text{Reg} \wedge (1 \ll \text{Bit})$	0111 bbba ffff ffff
Branch if the zero flag is set; label is two's complement offset; one instruction cycle if branch not executed, two if branch executed	<code>bz Label</code>	<pre>if (Z == 1) PC = PC + 2 + Label endif</pre>	1110 0000 kkkk kkkk
Call the 20-bit label address; if "s" is set, save the context registers (two instruction cycles)	<code>call Label, s</code>	<pre>PUSH(PC) if (s == 1) PUSH(W, STATUS, BSR) endif PC = Label</pre>	1110 110s kkkk kkkk 1111 kkkk kkkk kkkk

Clear the specified register; if “a” is set, then the BSR is used for Reg, else the access bank is used	<code>clrf Reg, a</code>	<pre> Reg = 0 Z = 1 </pre>	0110 101a ffff ffff
Clear the watchdog register and STATUS flags	<code>clrwdt</code>	<pre> WDT = 0 WDT Postscaler = 0 _TO = 1 _PD = 1 </pre>	0000 0000 0000 0100
Complement the contents of the specified register; if “a” is set, then the BSR is used for Reg, else the access bank is used	<code>comf Reg, d, a</code>	<pre> if (“d” == 0) wreg = Reg ^ 0xFF else Reg = Reg ^ 0xFF endif if ((Reg ^ 0xFF) == 0x000) Z = 1 else Z = 0 endif if ((Reg ^ 0xFF) & 0x080) != 0) N = 1 else N = 0 endif </pre>	0001 11da ffff ffff
Compare the specified register with the w register and skip if the register = w register; if “a” is set, then the BSR is used for Reg, else the access bank is used; one instruction cycle if skip not executed, two if skip executed	<code>cpfseq Reg, a</code>	<pre> if ((Reg - wreg) == 0) PC = NextIns endif </pre>	0110 001a ffff ffff

(Continued)

TABLE B.6 PIC18C INSTRUCTION SET (CONTINUED)

INSTRUCTION	FORMAT	OPERATION	BIT PATTERN
Compare the specified register with the w register and skip if the register > w register; if “a” is set, then the BSR is used for Reg, else the access bank is used; one instruction cycle if skip not executed, two if skip executed	cpfsgt Reg, a	if ((Reg - wreg) > 0) PC = NextIns endif	0110 010a ffff ffff
Compare the specified register with the w register and skip if the register < w register; if “a” is set, then the BSR is used for Reg, else the access bank is used; one instruction cycle if skip not executed, two if skip executed	cpfslt Reg, a	if ((Reg - wreg) < 0) PC = NextIns endif	0110 000a ffff ffff
Do a decimal adjust after addition of two BCD values	daw	if ((wreg & 0x0F) > 9) wreg = (wreg & 0x0F) + 0x010 endif	0000 0000 0000 0111
Decrement the contents of the specified register; if “a” is set, then Reg is in BSR bank, else access bank is used	decf Reg, d, a	if (“d” == 0) wreg = Reg - 1 else Reg = Reg - 1 endif if ((Reg - 1) == 0x000) Z = 1 else Z = 0 endif if (((Reg > 0) & ((Reg - 1) < 0x080)) OV = 1 else	0000 01da ffff ffff

```

    OV = 0
endif
if ((Reg - 1) & 0x080) != 0)
    N = 1
else
    N = 0
endif
if ((Reg & 0x00F) - 1)
    & 0x080) != 0)
    DC = 0
else
    DC = 1
endif
if ((Reg - 1) < 0)
    C = 0
else
    C = 1
endif

```

Decrement the contents of the specified register and skip the next instruction if result = 0; if “a” is set, then the BSR is used for Reg, else the access bank is used; one instruction cycle if skip not executed, two if skip executed

decfsz Reg, d, a

```

if (“d” == 0)
    wreg = Reg - 1
else
    Reg = Reg - 1
endif
if ((Reg - 1) == 0x000)
    PC = NextIns
endif

```

0010 11da ffff ffff

Decrement the contents of the specified register and skip the next instruction if result = 0; if “a” is set, then the BSR is used for Reg, else the access bank is used; one instruction cycle if skip not executed, two if skip executed

dcfsnz Reg, d, a

```

if (“d” == 0)
    wreg = Reg - 1
else
    Reg = Reg - 1
endif
if ((Reg - 1) != 0x000)
    PC = NextIns
endif

```

0100 11da ffff ffff

TABLE B.6 PIC18C INSTRUCTION SET (CONTINUED)

INSTRUCTION	FORMAT	OPERATION	BIT PATTERN
Goto the 20-bit label address (two instruction cycles)	goto Label	PC = Label	1110 1111 kkkk kkkk 1111 kkkk kkkk kkkk
Increment the contents of the specified register; if “a” is set, then Reg is in the BSR bank, else access bank is used.	incf Reg, d, a	<pre> if (“d” == 0) wreg = Reg + 1 else Reg = Reg + 1 endif if ((Reg + 1) == 0x000) Z = 1 else Z = 0 endif if (((Reg > 0) & ((Reg + 1) > 0x07F)) OV = 1 else OV = 0 endif if ((Reg + 1) & 0x080) != 0) N = 1 else N = 0 endif if (((Reg & 0x00F) + 1) & 0x010) != 0) DC = 1 else DC = 0 endif if ((Reg + 1) == 0x0100) C = 1 </pre>	0010 10da ffff ffff

Increment the contents of the specified register and skip the next instruction if result = 0; if “a” is set, then the BSR is used for Reg, else the access bank is used; one instruction cycle if skip not executed, two if skip executed

`incfsz Reg, d, a`

```

else
    C = 0
endif
if (“d” == 0)
    wreg = Reg + 1
else
    Reg = Reg + 1
endif
if ((Reg + 1) == 0x000)
    PC = NextIns
endif

```

0011 11da ffff ffff

Increment the contents of the specified register and skip the next instruction if result = 0; if “a” is set, then the BSR is used for Reg, else the access bank is used; one instruction cycle if skip not executed, two if skip executed

`infsnz Reg, d, a`

```

If (“d” == 0)
    wreg = Reg + 1
else
    Reg = Reg + 1
endif
if ((Reg + 1) != 0x000)
    PC = NextIns
endif

```

0100 10da ffff ffff

OR a constant to the w register and store the result in the w register

`iorlw Constant`

```

wreg = wreg | Constant
if ((wreg | Constant)
    == 0x000)
    Z = 1
else
    Z = 0
endif
if ((wreg | Constant)
    & 0x080) != 0)
    N = 1
else
    N = 0
endif

```

0000 1001 kkkk kkkk

(Continued)

TABLE B.6 PIC18C INSTRUCTION SET (CONTINUED)

INSTRUCTION	FORMAT	OPERATION	BIT PATTERN
OR w register to the contents of Reg and store the result according to “d” result in w register; if “a” is set, then Reg is in the BSR bank, else it is in the access bank	<code>iorwf Reg, d, a</code>	<pre> if (“d” == 1) wreg = wreg Reg else Reg = wreg Reg endif if ((wreg Reg) = 0x000) Z = 1 else Z = 0 endif if ((wreg Reg) & 0x080) != 0) N = 1 else N = 0 endif </pre>	0001 00da ffff ffff
Load the specified FSR register with the constant (two instruction cycles)	<code>lfsr f, Const</code>	<code>FSR(f) = Const</code>	1110 1110 00ff kkkk 1111 0000 kkkk kkkk
Move data from 256 address register data to primary register set; if “a” is set, then the BSR is used for Reg, else the access bank is used	<code>movf Reg, d, a</code>	<pre> if (d == 0) wreg = Reg endif if (Reg == 0) Z = 1 else Z = 0 endif if ((Reg & 0x080) != 0) N = 1 else N = 0 endif </pre>	0101 00da ffff ffff

Move contents of the source register into the destination register; the full 12-bit addresses are specified (two instruction cycles)	movff	Regd = Regs	1100 fffs fffs fffs 1111 fffd fffd fffd
Move constant into low nybble of BSR	movlb Constant	BSR(3:0) = Constant	0000 0001 kkkk kkkk
Move constant into the w register	movlw Constant	wreg = Constant	0000 1110 kkkk kkkk
Move contents of the w register into Reg; if “a” is set, then the BSR is used for Reg, else the access bank is used	movwf Reg, a	Reg = wreg	0110 111a ffff ffff
Multiply constant by the w register		PRODH:PROGL = Constant * wreg	0000 1101 kkkk kkkk
Multiply register by the w register; if “a” is set, then the BSR is used for Reg, else the access bank is used	mullwf Reg	PRODH:PROGL = Reg * wreg	0000 0010a ffff ffff
Negate the contents of Reg and store the result back in Reg; if “a” is set, then Reg is in the BSR bank, else Reg is in the access bank.	negw Reg, a	Reg = -Reg if (-Reg < 0x080) OV = 1 else OV = 0 endif if ((-Reg & 0x080) != 0) N = 1 else N = 0 endif if (-Reg > 0xFF) C = 1 else C = 0 endif	0110 110a ffff ffff

(Continued)

TABLE B.6 PIC18C INSTRUCTION SET (CONTINUED)

INSTRUCTION	FORMAT	OPERATION	BIT PATTERN
		<pre> if ((-Reg & 0x0F) > 0x0F) DC = 1 else DC = 0 endif if (-Reg == 0x000) Z = 1 else Z = 0 endif </pre>	
Do nothing for one instruction cycle; note two different op codes.	<code>nop</code>		0000 0000 0000 0000 1111 1111 1111 1111
Pop the top of the instruction pointer stack and discard the result	<code>pop</code>	<code>POP()</code>	0000 0000 0000 0110
Push the top of the instruction pointer stack.	<code>push</code>	<code>PUSH(PC + 2)</code>	0000 0000 0000 0101
Call the 11-bit two's complement offset (two instruction cycles)	<code>rcall Label</code>	<pre> PUSH(PC) PC = PC + 2 + Label </pre>	1101 1kkk kkkk kkkk
Reset the PIC microcontroller processor and all the registers affected by <code>_MCLR</code> reset	<code>Reset</code>	<pre> _MCLR = 0 _MCLR = 1 </pre>	0000 0000 1111 1111
Return from interrupt handler; if "s" is set, restore the w, STATUS, and BSR registers (two instruction cycles)	<code>retfie, s</code>	<pre> PC = POP() GIE = 0 if (s == 1) wreg = POP() STATUS = POP() BSR = POP() endif </pre>	0000 0000 0001 000s

Return from subroutine with new value in w register (two instruction cycles)	retlw Constant	wreg = Constant PC = POP()	0000 1100 kkkk kkkk
Return from subroutine; if “s” is set, restore the w, STATUS, and BSR registers (two instruction cycles)	return, s	PC = POP() if (s == 1) wreg = POP() STATUS = POP() BSR = POP() endif	0000 0000 0001 001s
Rotate left through the carry flag; if “a” is set, then Reg is in BSR bank, else Reg is in the access bank	rlcf Reg, d, a	if (d == 0) wreg(7:1) = Reg(6:0) wreg(0) = C C = Reg(7) else Reg(7:1) = Reg(6:0) Reg(0) = C C = Reg(7) endif if (Reg(6) != 0) N = 1 else N = 0 endif	0011 01da ffff ffff
Rotate left; if “a” is set, then Reg is in the BSR bank, else Reg is in the access bank	rlcnf Reg, d, a	if (d == 0) wreg(7:1) = Reg(6:0) wreg(0) = Reg(7) else Reg(7:1) = Reg(6:0) Reg(0) = Reg(7) endif if (Reg(6) != 0) N = 1 else N = 0 endif	0100 01da ffff ffff

(Continued)

TABLE B.6 PIC18C INSTRUCTION SET (CONTINUED)

INSTRUCTION	FORMAT	OPERATION	BIT PATTERN
Rotate right through the carry flag; if “a” is set, then Reg is in the BSR bank, else Reg is in the access bank	<code>rrcf Reg, d, a</code>	<pre> If (d == 0) wreg(6:0) = Reg(7:1) wreg(7) = C C = Reg(0) else Reg(6:0) = Reg(7:1) Reg(7) = C C = Reg(0) endif if (Reg(0) != 0) N = 1 else N = 0 endif </pre>	0011 00da ffff ffff
Rotate right; if “a” is set, then Reg is in the BSR bank, else Reg is in the access bank	<code>rrcnf Reg, d, a</code>	<pre> if (d == 0) wreg(6:0) = Reg(7:1) wreg(7) = Reg(0) else Reg(6:0) = Reg(7:1) Reg(7) = Reg(0) endif if (Reg(0) != 0) N = 1 else N = 0 endif </pre>	0100 00da ffff ffff
Set the specified register and optionally the w register; if “a” is set, then the BSR is used for Reg, else the access bank is used	<code>setf Reg, s, a</code>	<pre> Reg = 0x0FF if (s == 0) wreg = 0x0FF endif </pre>	0110 100a ffff ffff

Put the PIC microcontroller in a power-down state	sleep	<pre> WDT = 0 WDT Postscaler = 0 _TO = 1 _PD = 0 PIC microcontroller Power Down </pre>	0000 0000 0000 0011
Subtract the contents of Reg and C from the w register and store the result according to "d" result in w register; if "a" is set, then Reg is in the BSR bank, else it is in the access bank	subwfb Reg, d, a	<pre> if ("d" == 1) wreg = wreg - Reg - !C else Reg = wreg - Reg - !C endif if (((Reg > 0) & (wreg > 0)) & (wreg - Reg - !C) < 0x080)) OV = 1 else OV = 0 endif if (((wreg - Reg - C) & 0x080) != 0) N = 1 else N = 0 endif if ((wreg - Reg - !C) > 0x0FF) C = 1 else C = 0 endif if (((wreg & 0x0F) - (Reg & 0x0F) - !C) > 0x0F) DC = 1 else DC = 0 endif if (((wreg - Reg - !C) & 0x0FF) == 0x000) </pre>	0101 01da ffff ffff

(Continued)

TABLE B.6 PIC18C INSTRUCTION SET (CONTINUED)

INSTRUCTION	FORMAT	OPERATION	BIT PATTERN
Subtract the w register from a constant and store the result in the w register	sublw Constant	<pre> Z = 1 else Z = 0 endif wreg = Constant - wreg if (((wreg < 0) & (Constant < 0)) & ((Constant - wreg) < 0x080)) OV = 1 else OV = 0 endif if (((Constant - wreg) & 0x080) != 0) N = 1 else N = 0 endif if ((Constant - wreg) > 0xFF) C = 1 else C = 0 endif if (((Constant & 0x0F) - (wreg & 0x0F)) > 0x0F) DC = 1 else DC = 0 endif if (((Constant - wreg) & 0xFF) == 0x000) </pre>	0000 1000 kkkk kkkk

Subtract the w register from the contents of Reg and store the result according to “d” result in the w register; if “a” is set, then Reg is in the BSR bank, else Reg is in the access bank

```
subwf Reg,
d, a
```

```

    Z = 1
else
    Z = 0
endif
if ("d" == 1)
    wreg = Reg - wreg
else
    Reg = Reg - wreg
endif
if (((wreg < 0) & (Reg < 0))
    & ((Reg - wreg) < 0x080))
    OV = 1
else
    OV = 0
endif
if (((Reg - wreg) & 0x080) != 0)
    N = 1
else
    N = 0
endif
if ((Reg - wreg) > 0x0FF)
    C = 1
else
    C = 0
endif
if (((Reg & 0x0F) - (wreg & 0x0F))
    > 0x0F)
    DC = 1
else
    DC = 0
endif
if (((Reg - wreg) & 0x0FF) == 0x000)
    Z = 1
else
    Z = 0
endif

```

0101 11da ffff ffff

(Continued)

TABLE B.6 PIC18C INSTRUCTION SET (CONTINUED)

INSTRUCTION	FORMAT	OPERATION	BIT PATTERN
Subtract the w register from the contents of Reg and C, and store the result according to “d” result in the w register; If “a” is set, then Reg is in the BSR bank, else Reg is in the access bank	subwfb Reg, d, a	<pre> if (“d” == 1) wreg = Reg - wreg - !C else Reg = Reg - wreg - !C endif if (((Reg > 0) & (wreg > 0)) & ((Reg - wreg - !C) < 0x080)) OV = 1 else OV = 0 endif if (((Reg - wreg - C) & 0x080) != 0) N = 1 else N = 0 endif if ((Reg - wreg - !C) > 0x0FF) C = 1 else C = 0 endif if (((Reg & 0x0F) - (wreg & 0x0F) - !C) > 0x0F) DC = 1 else DC = 0 endif if (((Reg - wreg - !C) & 0x0FF) == 0x000) Z = 1 else Z = 0 endif </pre>	0101 10da ffff ffff

Swap the the contents of Reg and store the result according to “d” result in the w register; if “a” is set, then Reg is in the BSR bank, else Reg is in the access bank	swapf Reg, d, a	<pre> if (d == 1) Reg = ((Reg & 0xF0) >> 4) + ((Reg & 0x0F) << 4) else w = ((Reg & 0xF0) >> 4) + ((Reg & 0x0F) << 4) endif </pre>	0011 10da ffff ffff
Read the program memory contents at the table pointer and execute as option specifies (two instruction cycles)	tablr Option Option: nn Action 00 * 01 *+ 10 *- 11 +*	<pre> switch(Option) case * TABLAT = ProgMem(TBLPTR) case *+ TABLAT = ProgMem(TBLPTR) TBLPTR = TBLPTR + 1 case *- TABLAT = ProgMem(TBLPTR) TBLPTR = TBLPTR - 1 case +* TABLAT = ProgMem(TBLPTR) TBLPTR = TBLPTR + 1 TABLAT = ProgMem(TBLPTR) </pre>	0000 0000 0000 10nn
Write the contents of the table latch into program memory based on the option specification pointer; if the destination is internal program memory, the instruction does not end until an interrupt (two instruction cycles or many if program memory EPROM/Flash write)	tablw Option Option: nn Action 00 * 01 *+ 10 *- 11 +*	<pre> switch(Option) case * ProgMem(TBLPTR) = TABLAT case *+ ProgMem(TBLPTR) = TABLAT TBLPTR = TBLPTR + 1 case *- ProgMem(TBLPTR) = TABLAT TBLPTR = TBLPTR + 1 case +* ProgMem(TBLPTR) = TABLAT TBLPTR = TBLPTR + 1 ProgMem(TBLPTR) = TABLAT </pre>	0000 0000 0000 11nn
Compare the specified register zero and skip if the register = 0; if “a” is set, then the BSR is used for Reg, else the access bank is used; one instruction cycle if skip not executed, two if skip executed	tstfsz Reg, a	<pre> if (Reg == 0) PC = NextIns endif </pre>	0110 011a ffff ffff

(Continued)

TABLE B.6 PIC18C INSTRUCTION SET (CONTINUED)

INSTRUCTION	FORMAT	OPERATION	BIT PATTERN
XOR a constant to the w register and store the result in the w register	xorlw Constant	<pre>wreg = wreg ^ Constant if ((wreg ^ Constant) == 0x000) Z = 1 else Z = 0 endif if ((wreg ^ Constant) & 0x080) != 0) N = 1 else N = 0 endif</pre>	0000 1010 kkkk kkkk
XOR the w register to the contents of Reg and store the result according to "d" result in the w register; if "a" is set, then Reg is in the BSR bank, else Reg is in the access bank	xorwf Reg, d, a	<pre>if ("d" == 1) wreg = wreg ^ Reg else Reg = wreg ^ Reg endif if ((wreg ^ Reg) == 0x000) Z = 1 else Z = 0 endif if ((wreg ^ Reg) & 0x080) != 0) N = 1 else N = 0 endif</pre>	0001 10da ffff ffff

after an `LCALL` statement and the `PCLATH` bits are not set appropriately for the current page, execution will jump into the `LCALL` page.

For low-end PIC microcontrollers, `LCALL` should be

```
bcf/bsf    STATUS, PA0
bcf/bsf    STATUS, PA1
bcf/bsf    STATUS, PA2
```

TABLE B.7 MICROCHIP SPECIAL MNEMONICS

FUNCTION PROVIDED	Equivalent Instruction	Actual Inserted Instructions	Function Operation
Add carry to file register	<code>addcf Reg, d</code>	<code>btfsc STATUS, C</code> <code>incf Reg, d</code>	if ($C == 1$) if ($d == 1$) $Reg = Reg + 1$; else $w = Reg + 1$
Add digit carry to file register	<code>adddcf Reg, d</code>	<code>btfsc STATUS</code> <code>incf Reg, d</code>	if ($DC == 1$) if ($d == 1$) $Reg = Reg + 1$; else $w = Reg + 1$;
Branch to label	<code>B Label</code>	<code>goto Label</code>	$PC = ((PCLATH \ll 8) \& 0x01800) + Label$;
Branch on carry set	<code>BC Label</code>	<code>btfsc STATUS, C</code> <code>goto Label</code>	if ($C == 1$) $PC = ((PCLATH \ll 8) \& 0x01800) + Label$;
Branch on digit carry set	<code>BDC Label</code>	<code>btfsc STATUS, DC</code> <code>goto Label</code>	if ($DC == 1$) $PC = ((PCLATH \ll 8) \& 0x01800) + Label$;
Branch on carry reset	<code>BNC Label</code>	<code>btfss STATUS, C</code> <code>goto Label</code>	if ($C == 0$) $PC = ((PCLATH \ll 8) \& 0x01800) + Label$;
Branch on digit carry reset	<code>BNDC Label</code>	<code>btfss STATUS, DC</code> <code>goto Label</code>	If ($DC == 0$) $PC = ((PCLATH \ll 8) \& 0x01800) + Label$;
Branch on zero reset	<code>BNZ Label</code>	<code>btfss STATUS, Z</code> <code>goto Label</code>	If ($Z == 0$) $PC = ((PCLATH \ll 8) \& 0x01800) + Label$;
Branch on zero set	<code>BZ Label</code>	<code>btfsc STATUS, Z</code> <code>goto Label</code>	If ($Z == 1$) $PC = ((PCLATH \ll 8) \& 0x01800) + Label$;

(Continued)

TABLE B.7 MICROCHIP SPECIAL MNEMONICS (CONTINUED)

FUNCTION PROVIDED	Equivalent Instruction	Actual Inserted Instructions	Function Operation
Clear carry	clrc	bcf STATUS, C	C = 0;
Clear digit carry	clrdc	bcf STATUS, DC	DC = 0;
Long call— do <i>not</i> use, as described above	lcall Label	Low-End: bcf/bsf STATUS, PA0 bcf/bsf STATUS, PA1 bcf/bsf STATUS, PA2 call Label MidRange: bcf/bsf PCLATH, 3 bcf/bsf PCLATH, 4 call Label	
Long goto	lgoto Label	Low-End: bcf/bsf STATUS, PA0 bcf/bsf STATUS, PA1 bcf/bsf STATUS, PA2 goto Label Mid-Range: bcf/bsf PCLATH, 3 bcf/bsf PCLATH, 4 goto Label	
Load w register with contents of Reg	movfw Reg	movf Reg, w	W = Reg if (Reg == 0) Z = 1; else Z = 0;
Negate a file register— <i>only</i> use if “d” equals 1 (putting result back into the file register)	negf Reg, d	comf Reg, f incf Reg, d	Reg = Reg ^ 0xFF if (d == 0) w = Reg + 1; else Reg = Reg + 1;

TABLE B.7 MICROCHIP SPECIAL MNEMONICS (CONTINUED)

FUNCTION PROVIDED	Equivalent Instruction	Actual Inserted Instructions	Function Operation
Set carry	setc	bsf STATUS, C	C = 1;
Set digit carry	setdc	bsf STATUS, DC	DC = 1;
Set zero	setz	bsf STATUS, Z	Z = 1;
Skip the next instruction if the carry flag is set	skpc	btfs STATUS, C	if (C == 1) PC = PC + 1;
Skip the next instruction if the digit carry flag is set	skpdc	btfs STATUS, DC	if (DC == 1) PC = PC + 1;
Skip the next instruction if the carry flag is reset	skpnc	btfs STATUS, C	if (C == 0) PC = PC + 1;
Skip the next instruction if the digit carry flag is reset	skpndc	btfs STATUS, DC	if (DC == 0) PC = PC + 1;
Skip the next instruction if the zero flag is reset	skpnz	btfs STATUS, Z	if (Z == 0) PC = PC + 1;
Skip the next instruction if the zero flag is set	skpz	btfs STATUS, Z	if (Z == 1) PC = PC + 1;
Negate a file register	negf Reg, d	comf Reg, f incf Reg, d	Reg = Reg ^ 0x0FF; if (d == 0) w = Reg + 1; else Reg = Reg + 1;
Subtract carry from file register	subcf Reg, d	btfs STATUS, C decf Reg, d	if (C == 1) if (d == 1) Reg = Reg - 1; else w = Reg - 1;
Subtract digit carry to file register	adddcf Reg, d	btfs STATUS, DC decf Reg, d	if (DC == 1) if (d == 1) Reg = Reg - 1; else w = Reg - 1;
Load Z with 1 if contents of Reg equal 0	movfw Reg	movf Reg, f	if (Reg == 0) Z = 1; else Z = 0;

```
call      (Label & 0x1FF) + ($ & 0xE00)
bsf/bcf   STATUS, PA0
bsf/bcf   STATUS, PA1
bsf/bcf   STATUS, PA2
```

and for mid-range devices, LCALL should be

```
bcf/bsf   PCLATH, 3
bcf/bsf   PCLATH, 4
call      (Label & 0x7FF) + ($ & 0x1800)
bsf/bcf   PCLATH, 3
bsf/bcf   PCLATH, 4
```

`negf` never should be used unless the destination is back into the file register source. If the destination is the `w` register, note that the contents of the file register source will be changed with the complement of the value. Because of this added complexity, use of this special instruction is not recommended.

I/O Register Addresses

The different PIC microcontroller architecture families each have a set of registers at specific addresses. These conventions allow code to be transferred between PIC MCUs designed with the same processors very easily. Over the past few years, the register labels have been made as similar as possible and match the MPASM assembler codes to ensure that applications can be ported between devices within and without the current PIC microcontroller architecture family.

While the register addresses are very similar between PIC microcontrollers of the same architecture family, remember that the bits in the different registers may change function with different PIC microcontroller part numbers. To be absolutely sure of the bits and their function inside a register, consult the Microchip part datasheet.

The register addressing information contained in the rest of this appendix is provided to give you a reference on how the different PIC microcontroller family architecture's registers are addressed.

LOW-END PIC MICROCONTROLLERS

The low-end PIC microcontroller devices have five register bank address bits for up to 32 unique file register addresses in each bank. Up to four register banks can be available in a low-end PIC microcontroller, with the first 16 addresses of each bank being common throughout the banks and the second 16 addresses being unique to the bank. This is shown in Fig. B.1. Using this scheme, low-end PIC microcontrollers have anywhere from 25 to 73 unique file registers available to an application (see Table B.8).

There are a few things to note with low-end register addressing:

- 1 The `OPTION` and `TRIS` registers can be written to only by the `option` and `tris` instructions, respectively.

Bank 0	Bank 1	Bank 2	Bank 3	
Addr - Reg	Addr - Reg	Addr - Reg	Addr - Reg	
00 - INDF 01 - TMR0 02 - PCL 03 - STATUS 04 - FSR 05 - PORTA* 06 - PORTB 07 - PORTC 08-0F Shared File Regs	20 - INDF 21 - TMR0 22 - PCL 23 - STATUS 24 - FSR 25 - PORTA* 26 - PORTB 27 - PORTC 28-2F Shared File Regs	40 - INDF 41 - TMR0 42 - PCL 43 - STATUS 44 - FSR 45 - PORTA* 46 - PORTB 47 - PORTC 28-2F Shared File Regs	60 - INDF 61 - TMR0 62 - PCL 63 - STATUS 64 - FSR 65 - PORTA* 66 - PORTB 67 - PORTC 68-8F Shared File Regs	Shared Registers
10-1F Bank 0 File Regs	30-3F Bank 1 File Regs	50-4F Bank 2 File Regs	70-7F Bank 3 File Regs	Bank Unique Registers

* - "OSCCAL" may take place of "PORTA" in PICMicros with Internal Oscillators

OPTION - Accessed via "option" Instruction

TRIS# - Accessed via "TRIS PORT#" Instruction

Figure B.1 The Low-End PIC microcontroller processor architecture is limited to four banks of up to 32-byte-wide registers.

- 2** If the device has a built-in oscillator, the OSCCAL register is located in address 5, which is normally the PORTA address.
- 3** The STATUS and OPTION registers are always the same for low-end devices.
- 4** The low-end PIC microcontroller FSR register can never equal zero.

MID-RANGE PIC MICROCONTROLLER REGISTERS

If you look at the different mid-range PIC microcontroller devices, you will see that there is a great diversity in the register sets available to the various part numbers. This is quite a bit different from the other three PIC MCU families, in which the registers can be found at specific locations across the family. The diversity in the mid-range PIC microcontroller family is caused by the myriad of different features that have been released over the past few years, along with the number of different pin counts of the various devices.

Despite this diversity, there are some standard addresses (listed in Table B.9) that you can always count on with mid-range PIC microcontrollers. I always start with the block of registers in bank 0 and bank 1 listed in the table and then add to them the features that are built into the specific PIC microcontroller part number.

From these basic addresses, peripheral I/O registers (discussed below) are added to the register banks, with file registers starting at either offset 0x0C or 0x20. For most modern mid-range PIC microcontrollers, the file registers start at address 0x20 of the bank.

The specific part number datasheets will have to be checked to find where the file registers that are shared across the banks are located.

TABLE B.8 LOW-END PIC MICROCONTROLLER REGISTER DEFINITIONS			
ADDRESS	REGISTER	BITS	BIT FUNCTION
0x003	STATUS	7	GPWUF—in PIC12C5xx and PIC16C505, when set, reset from sleep on pin change; when set, power up or _MCLR reset; in other devices, bit 7 is unused.
		6–5	PA1–PA0—select the page to execute out of: 00—page 0 (0x0000–0x01FF) 01—page 1 (0x0200–0x03FF) 10—page 2 (0x0400–0x05FF) 11—page 3 (0x0600–0x07FF)
		4	_TO—set after power up and clrwdt and sleep instructions.
		3	_PD—set after power up and clrwdt instruction; reset after sleep instruction.
		2	Z—set if the 8-bit result is equal to zero.
		1	DC—set for low-order nybble carry after addition or subtraction instruction.
		0	C—set for carry after addition or subtraction instruction.
N/A	OPTION	7	_GPWU—in PIC12C5xx or PIC16C505, reset to enable wake-up on pin change; in other devices, bit 7 is unused.
		6	_GPPU—in PIC12C5xx or PIC16C505, enable pin pull-ups; in other devices, bit 6 is unused.
		5	T0CS—TMR0 clock source select; when set, T0CKI pin is source; when reset, instruction clock.
		4	T0SE—TMR0 edge select; when reset, increment TMR0 on rising edge; when set, increment TMR0 on falling edge.
		3	PSA—prescaler assignment bit; when set, the prescaler is assigned to the watchdog timer, else TMR0.
		2–0	PS2–PS0—prescaler rate select bits Bit TMR0 rate: 111—256:1 110—128:1 101—64:1 100—32:1 011—16:1 010—8:1 001—4:1 000—2:1

TABLE B.8 LOW-END PIC MICROCONTROLLER REGISTER DEFINITIONS (CONTINUED)

ADDRESS	REGISTER	BITS	BIT FUNCTION
			WDT rate:
			64:1
			32:1
			16:1
			8:1
			4:1
			2:1
			1:1

The STATUS register in mid-range PIC microcontroller is defined as listed in Table B.10.

The OPTION register (which has the label OPTION_REG in the Microchip include files) is defined in Table B.11.

Many devices have the PCON register (see Table B.12) that enhances the returned information contained in the _TO and _PD bits of the STATUS register.

The PCLATH register's contents (see Table B.13) are written to the program counter each time a goto or call instruction is executed or if the contents of PCL are changed.

Some mid-range devices are now available with built-in RC oscillators. To make the operation of the oscillators more accurate, the OSCCAL register is written to with a factory-specified *calibration value* register as presented in Table B.14.

TABLE B.9 MID-RANGE BANK 0/BANK 1 REGISTER DEFINITIONS

OFFSET	BANK 0	BANK 1	COMMENTS
0x00	INDF	INDF	
0x01	TMR0	OPTION	
0x02	PCL	PCL	
0x03	STATUS	STATUS	
0x04	FSR	FSR	
0x05	PORTA	TRISA	
0x06	PORTB	TRISB	
0x07	PORTC	TRISC	Available in 28/40-pin parts
0x08	PORTD	TRISD	Available in 40-pin parts
0x09	PORTE	TRISE	Available in 40-pin parts
0x0A	PCLATH	PCLATH	
0x0B	INTCON	INTCON	

TABLE B.10 MID-RANGE STATUS REGISTER DEFINITION	
BIT	FUNCTION
7	IRP—FSR select between the high and low register banks
6–5	RP1–RP0—direct addressing select banks (0–3)
4	_TO—Time-out bit; reset after a watchdog timer reset
3	_PD—Power-down active bit; reset after sleep instruction
2	Z—set when the 8-bit result is equal to zero
1	DC—set when the low nybble of addition/subtraction result carries to the high nybble
0	C—set when the addition/subtraction result carries to the next byte; also used with the rotate instructions

TABLE B.11 MID-RANGE OPTION REGISTER DEFINITION			
BIT	FUNCTION		
7	_RBPU—when reset, the PORTB pin pull-up is enabled.		
6	INTEDG—when set, interrupt request on rising edge of RB0/INT pin.		
5	T0CS—when set, TMR0 is incremented from the T0CKI pin, else by the internal instruction clock.		
4	T0SE—when set, TMR0 is incremented on the high to low (falling edge) of T0CKI.		
3	PSA—prescaler assignment bit; when set, the prescaler is assigned to the watchdog timer, else to TMR0.		
2–0	PS2–PS0—prescaler rate select.		
	Bit	TMR0 Rate	WDT Rate
	111	256:1	128:1
	110	128:1	64:1
	101	64:1	32:1
	000	32:1	16:1
	011	16:1	8:1
	010	8:1	4:1
	001	4:1	2:1
	000	2:1	1:1

TABLE B.12 MID-RANGE PCON REGISTER DEFINITION

BITS	FUNCTION
7	MPEEN—set if there is a memory parity error; this capability is built into a small number of PIC microcontrollers.
6–3	Unused
2	_PER—reset when there was a program memory parity error; this capability is built into a small number of PIC microcontrollers.
1	_POR—reset when execution is from a power-on reset.
0	_BOR—reset when execution is from a brown-out reset.

Interrupts are controlled from the INTCON register (see Table B.15), which controls the basic mid-range PIC microcontroller interrupts as well as access to enhanced interrupt features.

Bit 6 of INTCON may be a peripheral device interrupt enable/request bit, or it can be PEIE, which when set will enable peripheral interrupts set in the PIR and PIE registers. The PIR register(s) contains the F bits (interrupt request active), whereas PIE contains the E bits (interrupt request enable). As I work through the different peripherals, the E and F bits will be listed, but their actual location is part number-specific, and the datasheet will have to be consulted.

Data EEPROM is accessed via the EEADR and EEDATA registers, with EECON1 (see Table B.16) and EECON2 providing the access control. EECON2 is a pseudoregister, and the act of writing to it is used to verify that the operation request is valid.

The data EEPROM write interrupt request bit (EEIE) is either in a PIE register or INTCON. The parallel slave port (PSP; available only in 40-pin mid-range PIC microcontrollers and listed in Table B.17) is enabled by setting the PSPMODE bit. Interrupt request are enabled by the PSPIE flag and requested by the PSPIF flag of the PIE and PIR registers, respectively. The parallel slave port is controlled from TRISE. Note that when the parallel slave port is enabled, PORTD and PORTE cannot be used for I/O.

TABLE B.13 MID-RANGE PCLATH REGISTER DEFINITION

BITS	FUNCTION
7–5	Unused.
4	Select high and low pages.
3	Select odd or even pages.
2–0	Select the 256-instruction address block within current page; this data is used when PCL is written to directly.

TABLE B.14 MID-RANGE OSCCAL REGISTER DEFINITION	
BIT	FUNCTION
7–4	CAL3:CAL0—16-bit calibration value.
3	CALFST—increase the speed of the RC oscillator.
2	CALSLW—decrease the speed of the RC oscillator.
1–0	Unused.

Along with TMR0, some mid-range PIC microcontrollers have TMR1 and TMR2, which are used for basic timing operations as well as CCP (compare, capture, and PWM) I/O. TMR1 is a 16-bit-wide register (accessed via TMR1L and TMR1H) that will request an interrupt on overflow (TMR1IF) if the TMR1IE bit is set. The T1CON register (shown in Table B.18) controls the operation of TMR1.

TMR2 is an 8-bit register that is continually compared against a value in the PR2 register. To have TMR2 operate like TMR0 as an 8-bit timer with a range of 0x000 to 0x0FF, then the PR2 (the register TMR2 is compared against) is set to 0x000. The TMR2 output can be used to drive a PWM signal out. Interrupts (TMR2IF) can be requested after the TMR2 overflow has passed through a postscaler and TMR2IE is set. The T2CON register (see Table B.19) controls the operation of TMR2.

TMR1 and TMR2 are used with one of the two CCP (capture/compare/PWM) modules for advanced I/O. TMR1 is used for capture and compare, and TMR2 is used for PWM output. The CCPR2x registers are used for storing compare/capture values, and the CCPx register specifies the pin used for CCP. The CCPxCON register (shown in Table B.20) is used for controlling CCP operation.

TABLE B.15 MID-RANGE INTCON REGISTER DEFINITION	
BIT	FUNCTION
7	GIE—global interrupt enable; for any interrupt requests to be acknowledged, this bit must be set.
6	Device-specific interrupt enable (see below).
5	TOIE—TMR0 interrupt overflow request enable.
4	INTE—RB0/INT pin interrupt request enable.
3	RBIE—PORTB change interrupt request enable.
2	TOIF—TMR0 interrupt overflow request.
1	INTF—RB0/INT pin interrupt request.
0	RBIF—PORTB change interrupt request.

TABLE B.16 MID-RANGE EECON1 REGISTER DEFINITION

BIT	FUNCTION
7–5	Unused.
4	EEIF—EEPROM write complete interrupt request.
3	WRERR—bit set when EEPROM write was invalid.
2	WREN—set to enabling writing to EEPROM.
1	WR—write control bit.
0	RD—set to allow an EEPROM data read.

TABLE B.17 MID-RANGE PSP REGISTER DEFINITION

BIT	FUNCTION
7	IBF—bit set when a word has been written into the PIC microcontroller and has not been read.
6	OBF—bit set when a byte has been written to the PORTD output register and has not been read.
5	IBOV—bit set when a word has been written into the PIC microcontroller before the previous one has been read.
4	PSPMODE—bit set to enable parallel slave port.
3	Unused.
2	TRISE2—TRIS bit for E2.
1	TRISE1—TRIS bit for E1.
0	TRISE0—TRIS bit for E0

TABLE B.18 MID-RANGE T1CON REGISTER DEFINITION

BIT	FUNCTION
7–6	Unused. T1CKPS1–T1CKPS2—TMR1 input prescaler select.
3	T1OSCEN—set to enable external TMR1 oscillator.
2	_T1SYNCif external clock used for TMR1, then synchronize to it when this bit is reset.
1	TMR1CS—when set, TMR1 is driven by external clock/TMR1 oscillator.
0	TMR1ON—set to enable TMR1.

CCP interrupts are requested via the CCPxIF flag and enabled by the CCPXIE flag, where x is 1 or 2 depending on the active CCP module. There are three different SSP modules built into the PIC microcontroller. Each one provides somewhat different options, and understanding how they work will be critical to your applications and if I2C is going to be used with them. The basic SSP modules (SSP and BSSP) provide a full SPI interface and I2C slave mode interface. The SSPBUF register provides simple

TABLE B.19 MID-RANGE T2CON REGISTER DEFINITION																																			
BIT	FUNCTION																																		
7	Unused.																																		
6–3	TOUTPS3–TOUTPS0—TMR2 output postscaler select. <table><tr><th>Bit</th><th>Postscaler</th></tr><tr><td>1111</td><td>16:1</td></tr><tr><td>1110</td><td>15:1</td></tr><tr><td>1101</td><td>14:1</td></tr><tr><td>1100</td><td>13:1</td></tr><tr><td>1011</td><td>12:1</td></tr><tr><td>1010</td><td>11:1</td></tr><tr><td>1001</td><td>10:1</td></tr><tr><td>1000</td><td>9:1</td></tr><tr><td>0111</td><td>8:1</td></tr><tr><td>0110</td><td>7:1</td></tr><tr><td>0101</td><td>6:1</td></tr><tr><td>0100</td><td>5:1</td></tr><tr><td>0011</td><td>4:1</td></tr><tr><td>0010</td><td>3:1</td></tr><tr><td>0001</td><td>2:1</td></tr><tr><td>0000</td><td>1:1</td></tr></table>	Bit	Postscaler	1111	16:1	1110	15:1	1101	14:1	1100	13:1	1011	12:1	1010	11:1	1001	10:1	1000	9:1	0111	8:1	0110	7:1	0101	6:1	0100	5:1	0011	4:1	0010	3:1	0001	2:1	0000	1:1
Bit	Postscaler																																		
1111	16:1																																		
1110	15:1																																		
1101	14:1																																		
1100	13:1																																		
1011	12:1																																		
1010	11:1																																		
1001	10:1																																		
1000	9:1																																		
0111	8:1																																		
0110	7:1																																		
0101	6:1																																		
0100	5:1																																		
0011	4:1																																		
0010	3:1																																		
0001	2:1																																		
0000	1:1																																		
2	TMR2ON—set to enable TMR2.																																		
1–0	T2CKPS1–T2CKPS0—TMR2 input prescaler select. <table><tr><th>Bit</th><th>Prescaler</th></tr><tr><td>1x</td><td>16:1</td></tr><tr><td>01</td><td>4:1</td></tr><tr><td>00</td><td>1:1</td></tr></table>	Bit	Prescaler	1x	16:1	01	4:1	00	1:1																										
Bit	Prescaler																																		
1x	16:1																																		
01	4:1																																		
00	1:1																																		

TABLE B.20 MID-RANGE CCPXCON REGISTER DEFINITION

BIT	FUNCTION
7–6	Unused.
5–4	DCxB1–DCxB0—PWM duty cycle bit 1 and bit 0; these bits are only accessed by the PWM for its low-output values.
3–0	CCPxM3–CCPxM0—CCPx mode select.
Bit	Function
11xx	PWM mode
1011	Compare mode, trigger special event
1010	Compare mode, trigger on compare match
1001	Compare mode, initialize CCP pin high; on compare, match force CCP low
1000	Compare mode, initialize CCP pin low; on compare, match force CCP high
0111	Capture on every 16th rising edge
0110	Capture on every 4th rising edge
0101	Capture on every rising edge
0100	Capture on every falling edge
001x	Unused
0001	Unused
0000	Capture/compare/PWM off

buffering, with the SSPADD buffers providing the received address for comparing against I/O operations. To control the operation of the SSP, the SSPCON register (defined in Table B.21) is used.

The SSPSTAT register (see Table B.22) is also used to control the SSP.

The master SSP (MSSP) accesses similar registers for the same functions with a second SSPCON register. The important difference between the MSSP and the other SSP modules is the enabled I2C master hardware in the MSSP. The MSSP's SSPCON1 register is defined as shown in Table B.23.

SSPCON2 is used for I2C master mode and is defined in Table B.24.

The SSPSTAT register for MSSP is shown in Table B.25.

Interrupts are requested from the SSP via the SSPIF bit and enabled by the SSPIE bit. Nonreturn to zero (NRZ) asynchronous serial communications are accomplished by the built-in USART. This circuit also can be used for synchronous serial communications. The clock speed is determined by SPBRG. The TXREG and RCREG registers

TABLE B.21 MID-RANGE SSPCON REGISTER DEFINITION	
BIT	FUNCTION
7	WCOL—set if SSPBUF was written to while transmitting data or not in correct mode for transmit.
6	SSPOV—set when SSP receive overflow occurs.
5	SSPEN—enables pins for SSP mode.
4	CKP—in SPI, set for idle clock high; in I2C mode, set to enable clock.
3–0	SSPM3–SSPM0—SSP mode select: 1111—I2C slave mode, 10-bit address 1110—I2C slave mode, 7-bit address 110x—Reserved 1011—I2C firmware-controlled master 1010—Reserved 1001—Reserved 1000—Reserved 0111—I2C slave mode, 10-bit address 0110—I2C slave mode, 7-bit address 0101—SSP slave, _SS disabled 0100—SSP slave, _SS enabled 0011—SPI master, clock = TMR2 0010—SPI master, Fosc/64 0001—SPI master, Fosc/16 0000—SPI master, Fosc/4

are used to transfer data. The RCSTA is the primary USART control register and is defined in Table B.26. TXSTA is defined in Table B.27.

The RCIF interrupt request bit, when set, means that there is a character received in the USART. RCIF is enabled by RCIE. TXIF is set when the TX holding register is empty and is enabled by TXIE.

Comparator-equipped PIC microcontrollers have a built-in reference voltage source that is controlled by the VRCON register (see Table B.28).

The voltage reference output is defined by the formula:

$$V_{ref} = [1/4 * V_{dd} * (1 - VRR)] + V_{dd} * (VR3:VR0 / \{24 + [8 * (1 - VRR)]\})$$

For V_{dd} equal to 5.0 V, Table B.29 lists different V_{ref} values.

The voltage reference is normally used with the voltage comparator, which is controlled by the CMCON Register defined in Table B.30.

TABLE B.22 MID-RANGE SSPSTAT REGISTER DEFINITION

BITS	FUNCTION
7	SMP—data sampled at end of data output time if set, else middle.
6	CKE—data transmitted on rising edge of SCK when set.
5	D/_A—used by I2C; when set, indicates last byte transferred was data; when reset, indicates last byte transferred was address.
4	P—set when stop bit detected.
3	S—set when start bit indicated.
2	R/_W—set when command received was a read.
1	UA—set when application must update SSPADD register.
0	BF—set when buffer is full in RX and when TX is in process.

Interrupts requested by change on comparator outputs are specified by CMIF and enabled by CMIE. There are also some analog-to-digital converter (ADC) options that can be used with the PIC microcontroller. Operation of the ADC is controlled by the ADCON0 register (see Table B.31).

Selecting the PORTA, analog/digital functions, there are a number of different formats of ADCON1 that you should be aware of. For basic 18-pin PIC microcontroller ADCs, ADCON1 is defined in Table B.32.

TABLE B.23 MID-RANGE MSSP SSPCON1 REGISTER DEFINITION

BITS	FUNCTION
7	WCOL—set if SSPBUF was written to while transmitting data or not in correct mode for transmit.
6	SSPOV—set when SSP receive overflow occurs.
5	SSPEN—enables pins for SSP mode.
4	CKP—in SPI, set for idle clock high; in I2C mode, set to enable clock.
3–0	SSPM3–SSPM0—SSP mode select.
	1xx1—Reserved
	1x1x—Reserved
	1000—I2C master mode, clock = $F_{osc}/[4 * (SSPADD + 1)]$
	0111—I2C slave mode, 10-bit address
	0110—I2C slave mode, 7-bit address
	0101—SSP slave, _SS disabled.

TABLE B.24 MID-RANGE MSSP SSPCON2 REGISTER DEFINITION	
BIT	FUNCTION
7	GCEN—set to enable interrupt when general call address is received.
6	ACKSTAT—set when acknowledge received from I2C slave device.
5	ACKDT—reset to send acknowledge at the end of a byte receive.
4	ACKEN—acknowledge I2C sequence when set.
3	RCEN—set to enable I2C receive mode.
2	PEN—reset to initiate stop condition on I2C clock and data.
1	RSEN—set to initiate repeated start condition on I2C clock and data.
0	SEN—set to initiate start condition on I2C clock and data.

TABLE B.25 MID-RANGE MSSP SSPSTAT REGISTER DEFINITION	
BIT	FUNCTION
7	SMP—data sampled at end of data output time if set, else middle.
6	CKE—data transmitted on rising edge of SCK when set.
5	D/_A—used by I2C; when set, indicates last byte transferred was data; when reset, indicates last byte transferred was address.
4	P—set when stop bit detected.
3	S—set when start bit indicated.
2	R/_W—set when command received was a read.
1	UA—set when application must update SSPADD register.
0	BF—set when buffer is full in RX and when TX is in pr.

TABLE B.26 MID-RANGE RCSTA REGISTER DEFINITION	
BIT	FUNCTION
7	SPEN—set to enable the USART.
6	RX9—set to enable 9-bit serial reception.
5	SREN—set to enable single receive for synchronous mode.
4	CREN—set to enable continuous receive mode.
3	ADDEN—enables address detection in asynchronous mode.
2	FERR—framing error bit.
1	OERR—set after overrun error.
0	RX9D—ninth bit of data received.

TABLE B.27 MID-RANGE TXSTA REGISTER DEFINITION

BITS	FUNCTION
7	CSRC—set for synchronous clock generated internally.
6	TX9—set to enable 9-bit data transmission
5	TXEN—set to enable transmit.
4	SYNC—set to select synchronous mode.
3	Unused.
2	BRGH—Set to Select the High Baud Rate
1	TRMT—set when transmit shift register is empty.
0	TX9D—Ninth bit of transmit data.

For more advanced 18-pin PIC microcontrollers, ADCON1 is defined as shown in Table B.33.

Both 28- and 40-pin PIC microcontrollers have the ADCON1 register, as defined in Table B.34.

The result of the ADC operation is stored in ADRES, and ADIF is set on completion of the ADC operation to request an interrupt if ADIE is set. Moreover, 10-bit ADCs are also available in the PIC microcontroller, with a different ADCON1 register (see Table B.35).

In the case of 10-bit ADCs, the result is stored in ADRESL and ADRESH. This mid-range register list does not include the PIC16C92x's LED control registers. This, as well as any other I/O hardware registers that were not available when this appendix was written, can be found in the Microchip datasheets.

TABLE B.28 MID-RANGE VRCON REGISTER DEFINITION

BITS	FUNCTION
7	VREN—set to turn on voltage reference circuit
6	VROE—set to output voltage reference externally.
5	VRR—set for low-voltage reference range; reset for high-voltage reference range.
4	Unused.
3–0	VR3–VR0—Select the reference voltage output

TABLE B.29 VRCON VALUES TO VOLTAGE OUTPUT		
VR3:VR0	VRR = 1	VRR = 0
1111	3.13 V	3.59 V
1110	2.92 V	3.44 V
1101	2.71 V	3.28 V
1100	2.50 V	3.13 V
1011	2.29 V	2.97 V
1010	2.08 V	2.81 V
1001	1.88 V	2.66 V
1000	1.67 V	2.50 V
0111	1.46 V	2.34 V
0110	1.25 V	3.19 V
0101	1.04 V	2.03 V
0100	0.83 V	1.88 V
0011	0.63 V	1.72 V
0010	0.42 V	1.56 V
0001	0.21 V	1.41 V
0000	0.00 V	1.25 V

PIC18

The unique hardware registers built into the PIC18 are defined in Table B.36. Note that these registers are accessed either via the access bank or by using the BSR set to 0x0F. If the registers are to be accessed using the FSR register, then the high nybble is set to 0x0F. For this reason, I have set the first nybble of the 12-bit address as # in Table B.36. If the access bank is used, then there is no high nybble to the address. If the BSR or FSR registers are used for addressing, then # is F.

The PIC18 microcontroller chips are designed with many of the same macros as the mid-range devices. This means that the peripheral functions generally are constructed and accessed in exactly the same way as in the mid-range chips. In the interests of brevity, I have not listed the specific I/O registers in the PIC18 register list because the register/function definitions can be found in the preceding section.

Device Pinouts

In the following sections of this appendix I have tried to generalize the pinouts for various PIC microcontroller part numbers. These graphics are meant to represent how the pins are specified for the different part numbers and do not reflect the actual dimensions of the parts.

TABLE B.30 MID-RANGE CMCON REGISTER DEFINITION

BIT	FUNCTION					
7	C2OUT—set when C2Vin+ is greater than C2Vin–.					
6	C1OUT—set when C1Vin+ is greater than C1Vin–.					
5–4	Unused.					
3	CIS—comparator input switch; see CM2–CM0.					
2–0	CM2–CM0—comparator mode select bits:					
	CIS	C1Vin+	C1Vin–	C2Vin+	C2Vin–	Comments
111	x	Gnd	Gnd	Gnd	Gnd	Comparators off AN3 = C1OUT, RA4 = C2OUT
110	x	AN2	AN0	AN2	AN1	
101	x	Gnd	Gnd	AN2	AN1	
100	x	AN3	AN0	AN2	AN1	
011	x	AN2	AN0	AN2	AN1	
010	1	Vref	AN3	Vref	AN2	
010	0	Vref	AN0	Vref	AN1	
001	1	AN2	AN3	AN2	AN1	
001	0	AN2	AN0	AN2	AN1	
000	x	AN3	AN0	AN2	AN1	Comparators off

As a rule of thumb, pin-through-hole (PTH) parts (P and JW) are standard 0.300- and 0.600-in widths with pins 0.100 in apart in dual in-line packages. The height of the device depends on the package used. I use PTH parts for all the applications presented in this book because of the ease with which they can be handled, programmed, and assembled into circuits.

Surface-mount-technology (SMT) parts are either in dual in-line packages (SO) or in quad plastic chip carriers (PT, PQ, and L).

For actual device dimensions, check the datasheets (on the Microchip web site) for the PIC microcontroller that you are planning on using. Different packages for different PIC microcontrollers have different via, pad, and clearance specifications.

LOW-END

When describing low-end PIC microcontrollers, I also include the PIC12C50x and the PIC16C505, which do use the low-end PIC microcontroller processor architecture but are programmed using the mid-range's ICSP protocol. There are no PLCC or QFP packages used for low-end devices, and the pinouts remain the same whether or not the PIC microcontroller is in an SMT or PTH package.

TABLE B.31 MID-RANGE ADCON0 REGISTER DEFINITION	
BIT	FUNCTION
7–6	ADCS1–ADCS0—ADC conversion clock select: 11—internal RC oscillator 10—divide PIC microcontroller clock by 32 01—divide PIC microcontroller clock by 8 00—divide PIC microcontroller clock by 2
5–3	CHS2–CHS0—ADC conversion channel select bits: 111—AN7 110—AN6 101—AN5 100—AN4 011—AN3 010—AN2 001—AN1 000—AN0
2	GO/_DONE—set to start A/D conversion; reset by hardware when conversion before.
1	Unused.
0	ADON—set to turn on the ADC function unused.

MID-RANGE

The mid-range devices have the widest range of pinouts of any of the PIC microcontroller families. In the following figures, I have given the 8-, 14-, 18-, 28-, and 40-pin packages for the most popular devices, as well as the SMT packaging for the 40-pin devices.

TABLE B.32 MID-RANGE BASIC ADCON1 REGISTER DEFINITION					
BIT	FUNCTION				
1–0	Unused.				
	PCFG1–PCFG0—A/D select:				
	Bit	AN3	AN2	AN1	AN0
	11	D	D	D	D
	10	D	D	A	A
	01	Vref+	A	A	A
	00	A	A	A	A

TABLE B.33 MID-RANGE ADVANCED ADCON1 REGISTER DEFINITION

BIT	FUNCTION				
7–3	Unused.				
2–0	PCFG2–PCFG0—A/D select:				
	Bit	AN3	AN2	AN1	AN0
	111	D	D	D	D
	110	D	D	D	A
	101	D	D	Vref+	A
	100	D	D	A	A
	011	D	A	Vref+	A
	010	D	A	A	A
	001	A	A	Vref+	A
	000	A	A	A	A

For many of the devices, the pinout is similar, but the pin functions may be different. In these cases I have marked the pins with an asterisk to show that these pins have optional other purposes. If you are not sure of what a PIC microcontroller pin is for, check the datasheets included on the CD-ROM that came with this book or the datasheets at the Microchip web site.

TABLE B.34 MID-RANGE TWENTY-EIGHT/FORTY PIN PIC MICROCONTROLLER ADCON1 REGISTER DEFINITION

BIT	FUNCTION								
7–3	Unused.								
2–0	PCFG2–PCFG0—A/D select:								
	Bit	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0
	11x	D	D	D	D	D	D	D	D
	101	D	D	D	D	Vref+	D	A	A
	100	D	D	D	D	A	D	A	A
	011	D	D	A	A	Vref+	A	A	A
	010	D	D	D	A	A	A	A	A
	001	A	A	A	A	Vref+	A	A	A
	000	A	A	A	A	A	A	A	A

TABLE B.35 MID-RANGE 10-BIT ADC ADCON1 REGISTER DEFINITION											
BIT	FUNCTION										
7–6	Unused.										
5	ADFM—when set, the result is right-justified, else left-justified.										
4	Unused.										
3–0	PCFG3–PCFG0—A/D select;										
	Bits	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0	VR+	VR–
	1111	D	D	D	D	VR+	VR–	D	A	AN3	AN2
	1110	D	D	D	D	D	D	D	A	Vdd	Vss
	1101	D	D	D	D	VR+	VR–	A	A	AN3	AN2
	1100	D	D	D	A	VR+	VR–	A	A	AN3	AN2
	1011	D	D	A	A	VR+	VR–	A	A	AN3	AN2
	1010	D	D	A	A	VR+	A	A	A	AN3	Vss
	1001	D	D	A	A	A	A	A	A	Vdd	Vss
	1000	A	A	A	A	VR+	VR–	A	A	AN3	AN2
	011x	D	D	D	D	D	D	D	D	N/A	N/A
	0101	D	D	D	D	VR+	D	A	A	AN3	Vss
	0100	D	D	D	D	A	D	A	A	Vdd	Vss
	0011	D	D	D	D	VR+	A	A	A	AN3	Vss
	0010	D	D	D	A	A	A	A	A	Vdd	Vss
	0001	A	A	A	A	VR+	A	A	A	AN3	Vss
	0000	A	A	A	A	A	A	A	A	Vdd	Vss

The PIC14000, which is designed for “mixed signals,” uses the 28-pin packaging of the standard devices, but the pinouts are different, as shown in Fig. B.13.

The PIC16C92x LCD driver microcontrollers are fairly high pin count devices. Figure B.14 shows the 64-pin dual in-line package (DIP) part. There is also a PLCC and TQFP package for the parts as well.

PIC18

There is a lot of similarity between the mid-range PIC microcontroller’s pinouts and the PIC18 parts, as will be seen in the following figures.

TABLE B.36 PIC18 REGISTER DEFINITION

ADDRESS	REGISTER	FUNCTION/BIT DEFINITION																
0x0#80	PORTA	PORTA read/write register; pin options are as follows: <table><tr><th>Bit</th><th>Function</th></tr><tr><td>7</td><td>Unused</td></tr><tr><td>6</td><td>OSC2</td></tr><tr><td>5</td><td>Slave select/optional AN4</td></tr><tr><td>4</td><td>Open-drain output/Schmidt-trigger input</td></tr><tr><td>3–0</td><td>Optional AN3–AN0</td></tr></table>	Bit	Function	7	Unused	6	OSC2	5	Slave select/optional AN4	4	Open-drain output/Schmidt-trigger input	3–0	Optional AN3–AN0				
Bit	Function																	
7	Unused																	
6	OSC2																	
5	Slave select/optional AN4																	
4	Open-drain output/Schmidt-trigger input																	
3–0	Optional AN3–AN0																	
0x0#81	PORTB	PORTB read/write register; I/O pins can be pulled by software; pin options are defined as follows: <table><tr><th>Bit</th><th>Function</th></tr><tr><td>7–6</td><td>ICSP programming pins/interrupt on pin change</td></tr><tr><td>5</td><td>Interrupt on pin change</td></tr><tr><td>4</td><td>Interrupt on pin change</td></tr><tr><td>3</td><td>CCP2 I/O and PWM output</td></tr><tr><td>2</td><td>Interrupt source 3</td></tr><tr><td>1</td><td>Interrupt source 2</td></tr><tr><td>0</td><td>Interrupt source 1</td></tr></table>	Bit	Function	7–6	ICSP programming pins/interrupt on pin change	5	Interrupt on pin change	4	Interrupt on pin change	3	CCP2 I/O and PWM output	2	Interrupt source 3	1	Interrupt source 2	0	Interrupt source 1
Bit	Function																	
7–6	ICSP programming pins/interrupt on pin change																	
5	Interrupt on pin change																	
4	Interrupt on pin change																	
3	CCP2 I/O and PWM output																	
2	Interrupt source 3																	
1	Interrupt source 2																	
0	Interrupt source 1																	
0x0#82	PORTC	PORTC read/write registers; I/O pins have Schmidt-trigger inputs; pin options are as follows: <table><tr><th>Bit</th><th>Function</th></tr><tr><td>7</td><td>UART receive pin</td></tr><tr><td>6</td><td>UART transmit pin</td></tr><tr><td>5</td><td>Synchronous serial port data</td></tr><tr><td>4</td><td>SPI data or I2C data</td></tr><tr><td>3</td><td>SPI clock or I2C clock</td></tr><tr><td>2</td><td>CCP1 I/O and PWM output/TMR1 clock output</td></tr><tr><td>0</td><td>TMR1 clock input</td></tr></table>	Bit	Function	7	UART receive pin	6	UART transmit pin	5	Synchronous serial port data	4	SPI data or I2C data	3	SPI clock or I2C clock	2	CCP1 I/O and PWM output/TMR1 clock output	0	TMR1 clock input
Bit	Function																	
7	UART receive pin																	
6	UART transmit pin																	
5	Synchronous serial port data																	
4	SPI data or I2C data																	
3	SPI clock or I2C clock																	
2	CCP1 I/O and PWM output/TMR1 clock output																	
0	TMR1 clock input																	

(Continued)

TABLE B.36 PIC18 REGISTER DEFINITION (CONTINUED)

ADDRESS	REGISTER	FUNCTION/BIT DEFINITION										
0x0#83	PORTD	PORTD only available on 40-pin PIC18 devices; Schmidt-trigger inputs; used for data slave port.										
0x0#84	PORTE	PORTE only available on 40-pin PIC18; Schmidt-trigger inputs for I/O mode; used for data slave port as follows: <table><tr><th>Bit</th><th>Function</th></tr><tr><td>7–3</td><td>Unused</td></tr><tr><td>2</td><td>Negative active chip select</td></tr><tr><td>1</td><td>Negative active write enable to PIC18Cxx</td></tr><tr><td>0</td><td>Negative active output enable ($\overline{\text{RD}}$) from PIC18Cxx</td></tr></table>	Bit	Function	7–3	Unused	2	Negative active chip select	1	Negative active write enable to PIC18Cxx	0	Negative active output enable ($\overline{\text{RD}}$) from PIC18Cxx
Bit	Function											
7–3	Unused											
2	Negative active chip select											
1	Negative active write enable to PIC18Cxx											
0	Negative active output enable ($\overline{\text{RD}}$) from PIC18Cxx											
0x0#89	LATA	Data output latch/bypassing PORTA										
0x0#8A	LATB	Data output latch/bypassing PORTB										
0x0#8B	LATC	Data output latch/bypassing PORTC										
0x0#8C	LATD	Data output latch/bypassing PORTD; only available on 40-pin PIC18										
0x0#8D	LATE	Data output latch/bypassing PORTE; only available on 40-pin PIC18										
0x0#92	TRISA	I/O pin tristate control register; set bit to 0 for output mode.										
0x0#93	TRISB	I/O pin tristate control register; set bit to 0 for output mode.										
0x0#94	TRISC	I/O pin tristate control register; set bit to 0 for output mode.										
0x0#95	TRISD	I/O pin tristate control register; only available on 40-pin PIC18; set bit to 0 for output mode.										
0x0#96	TRISE	I/O pin tristate control register; only available on 40-pin PIC18; set bit to 0 for output mode; special function bits specified as follows; <table><tr><th>Bit</th><th>Function</th></tr><tr><td>7</td><td>IBF—set when PSP is enabled and a byte has been written to the PIC microcontroller</td></tr><tr><td>6</td><td>OBF—set when PSP is enabled and a byte output has not been read from the chip</td></tr></table>	Bit	Function	7	IBF—set when PSP is enabled and a byte has been written to the PIC microcontroller	6	OBF—set when PSP is enabled and a byte output has not been read from the chip				
Bit	Function											
7	IBF—set when PSP is enabled and a byte has been written to the PIC microcontroller											
6	OBF—set when PSP is enabled and a byte output has not been read from the chip											

0x0#9D	PIE1	5	IBOV—set when PSP is enabled and the byte written has been overwritten by a subsequent byte
		4	PSPMODE—set to enable PIC microcontroller's PSP I/O port
		3	Unused
		2	TRISE2—TRIS bit for RE2
		1	TRISE1—TRIS bit for RE1
		0	TRISE0—TRIS bit for RE0
		Peripheral interrupt enable register:	
		Bit	Function
		7	PSPIE—set to enable PSP interrupt request on read/write
		6	ADIE—set to enable interrupt request on completion of A/D operation
		5	RCIE—set to enable interrupt request on USART data receive
		4	TXIE—set to enable interrupt request on USART transmit holding register empty
		3	SSPIE—master synchronous serial port interrupt enable bit
		2	CCP1IE—set to enable CCP1 interrupt request enable
		1	TMR2IE—TMR2 to PR2 match interrupt request enable
0x0#9E	PIR1	0	TMR1IE—TMR1 overflow interrupt request enable
		Peripheral interrupt request register:	
		Bit	Function
		7	PSPIF—set on PSP read/write
		6	ADIF—set when A/D complete
		5	RCIF—set on USART data receive
		4	TXIF—set on USART transmit holding register empty
		3	SSPIF—set on synchronous serial port data transmission/reception complete
		2	CCP1IF—set on TMR1 capture or compare match
		1	TMR2IF—set on TMR2 to PR2 match
		0	TMR1IF—set on TMR1 overflow

(Continued)

TABLE B.36 PIC18 REGISTER DEFINITION (CONTINUED)

ADDRESS	REGISTER	FUNCTION/BIT DEFINITION
0x0#9F	IPR1	Peripheral interrupt priority register: Bit Function 7 PSPIP—set to give PSP interrupt request on read/write priority 6 ADIP—set to give interrupt request on completion of A/D operation priority 5 RCIP—set to give interrupt request on USART data receive priority 4 TXIP—set to enable interrupt request on USART transmit holding register empty priority 3 SSPIP—master synchronous serial port interrupt priority when set 2 CCP1IP—set to give CCP1 interrupt request priority 1 TMR2IP—TMR2 to PR2 match interrupt request priority when set 0 TMR1IF—TMR1 overflow interrupt request priority when set
0x0#9A	PIE2	Peripheral interrupt enable register: Bit Function 7–4 Unused 3 BCLIE—bus collision interrupt request enabled when set 2 LVDIE—low-voltage detect interrupt request enabled when set 1 TMR3IE—TMR3 overflow interrupt request enabled when set 0 CCP2IE—CCP2 interrupt request enabled when set
0x0#9B	PIR2	Peripheral interrupt request register: Bit Function 7–4 Unused 3 BCLIF—set for bus collision interrupt request 2 LVDIF—set for low-voltage detect interrupt request 1 TMR3IF—set for TMR3 overflow interrupt request 0 CCP2IF—set for CCP2 interrupt request

0x0#9C	IPR2	Peripheral interrupt priority register: Bit Function 7–4 Unused 3 BCLIF—set for bus collision interrupt given priority 2 LVDIF—set for low-voltage detect interrupt given priority 1 TMR3IF—set for TMR3 overflow interrupt request given priority 0 CCP2IF—set for CCP2 interrupt request given priority
0x0#AB	RCSTA	USART receive status and control register: Bit Function 7 SPEN—set to enable the USART 6 RX9—set to enable 9-bit serial reception 5 SREN—set to enable single receive for synchronous mode 4 CREN—set to enable continuous receive mode 3 ADDEN—enables address detection in asynchronous mode 2 FERR—framing error bit 1 OERR—set after overrun error 0 RX9D—ninth bit of data received
0x0#AC	TXSTA	USART transmit status and control register: Bit Function 7 CSRC—set for synchronous clock generated internally 6 TX9—set to enable 9-bit data transmission 5 TXEN—set to enable transmit 4 SYNC—set to select synchronous mode 3 Unused 2 BRGH—set to select the high baud rate 1 TRMT—set when transmit shift register is empty 0 TX9D—ninth bit of transmit data
0x0#AD	TXREG	USART transmit buffer register
0x0#AE	RCREG	USART receive holding register
0x0#AF	SPBRG	USART clock divisor register

(Continued)

TABLE B.36 PIC18 REGISTER DEFINITION (CONTINUED)

ADDRESS	REGISTER	FUNCTION/BIT DEFINITION																														
0x0#B1	T3CON	TMR3 control register: <table><tr><th>Bit</th><th>Function</th></tr><tr><td>7</td><td>RD16—enable read/write of TMR3 as a 16-bit operation</td></tr><tr><td>6–3</td><td>T3CCP2–T3CCP0—TMR3 and TMR1 to CCPx enable bits</td></tr><tr><td></td><td>1x—TMR3 is CCP clock source</td></tr><tr><td></td><td>01—TMR3 is CCP2 clock source/TMR1 is CCP1 clock source</td></tr><tr><td></td><td>00—TMR1 is CCP clock source</td></tr><tr><td>5–4</td><td>T3CKPS1–T3CKPS0—TMR3 input clock</td></tr><tr><td colspan="2">Prescaler Control</td></tr><tr><td></td><td>11—1:8 prescaler</td></tr><tr><td></td><td>10—1:4 prescaler</td></tr><tr><td></td><td>01—1:2 prescaler</td></tr><tr><td></td><td>00—1:1 prescaler</td></tr><tr><td>2</td><td>_T3SYNC—when reset, TMR3 external clock is synchronized</td></tr><tr><td>1</td><td>TMR3CS—set to select external clock for TMR3; reset to select instruction clock</td></tr><tr><td>0</td><td>TMR3ON—set to enable TMR3</td></tr></table>	Bit	Function	7	RD16—enable read/write of TMR3 as a 16-bit operation	6–3	T3CCP2–T3CCP0—TMR3 and TMR1 to CCPx enable bits		1x—TMR3 is CCP clock source		01—TMR3 is CCP2 clock source/TMR1 is CCP1 clock source		00—TMR1 is CCP clock source	5–4	T3CKPS1–T3CKPS0—TMR3 input clock	Prescaler Control			11—1:8 prescaler		10—1:4 prescaler		01—1:2 prescaler		00—1:1 prescaler	2	_T3SYNC—when reset, TMR3 external clock is synchronized	1	TMR3CS—set to select external clock for TMR3; reset to select instruction clock	0	TMR3ON—set to enable TMR3
Bit	Function																															
7	RD16—enable read/write of TMR3 as a 16-bit operation																															
6–3	T3CCP2–T3CCP0—TMR3 and TMR1 to CCPx enable bits																															
	1x—TMR3 is CCP clock source																															
	01—TMR3 is CCP2 clock source/TMR1 is CCP1 clock source																															
	00—TMR1 is CCP clock source																															
5–4	T3CKPS1–T3CKPS0—TMR3 input clock																															
Prescaler Control																																
	11—1:8 prescaler																															
	10—1:4 prescaler																															
	01—1:2 prescaler																															
	00—1:1 prescaler																															
2	_T3SYNC—when reset, TMR3 external clock is synchronized																															
1	TMR3CS—set to select external clock for TMR3; reset to select instruction clock																															
0	TMR3ON—set to enable TMR3																															
0x0#B2	TMR3L	Low byte of TMR3																														
0x0#B3	TMR3H	High byte of TMR3																														
0x0#BA	CCP2CON	CCP2 control register: <table><tr><th>Bit</th><th>Function</th></tr><tr><td>7–6</td><td>Unused</td></tr><tr><td>5–4</td><td>DC2BX1–DC2BX0—two least significant bits for the 10-bit PWM</td></tr><tr><td>3–0</td><td>CCP2M3–CCP2M0—CCP2 mode select bits:</td></tr><tr><td></td><td>11xx—PWM mode</td></tr><tr><td></td><td>1011—trigger special event compare mode</td></tr><tr><td></td><td>1010—generate interrupt on compare match</td></tr><tr><td></td><td>1001—initialize CCP2 high and force low on compare match</td></tr><tr><td></td><td>1000—initialize CCP1 high and force high on compare match</td></tr></table>	Bit	Function	7–6	Unused	5–4	DC2BX1–DC2BX0—two least significant bits for the 10-bit PWM	3–0	CCP2M3–CCP2M0—CCP2 mode select bits:		11xx—PWM mode		1011—trigger special event compare mode		1010—generate interrupt on compare match		1001—initialize CCP2 high and force low on compare match		1000—initialize CCP1 high and force high on compare match												
Bit	Function																															
7–6	Unused																															
5–4	DC2BX1–DC2BX0—two least significant bits for the 10-bit PWM																															
3–0	CCP2M3–CCP2M0—CCP2 mode select bits:																															
	11xx—PWM mode																															
	1011—trigger special event compare mode																															
	1010—generate interrupt on compare match																															
	1001—initialize CCP2 high and force low on compare match																															
	1000—initialize CCP1 high and force high on compare match																															

		0111—capture on every 16th rising edge 0110—capture on every 4th rising edge 0101—capture on every rising edge 0100—capture on every falling edge 0011—reserved 0010—toggle output on compare match 0001—reserved 0000—capture/compare/PWM off
0x0#BB	CCPR2L	Least significant capture/compare/PWM2 register
0x0#BC	CCPR2H	Most significant capture/compare/PWM2 register
0x0#BD	CCP1CON	CCP1 control register:
		Bit Function 7–6 Unused 5–4 DC1BX1–DC1BX0—two least significant bits for the 10-bit PWM 3–0 CCP1M3–CCP1M0—CCP1 mode select bits: 11xx—PWM mode 1011—trigger special event compare mode 1010—generate interrupt on compare match 1001—initialize CCP2 high and force low on compare match 1000—initialize CCP1 high and force high on compare match 0111—capture on every 16th rising edge 0110—capture on every 4th rising edge 0101—capture on every rising edge 0100—capture on every falling edge 0011—reserved 0010—toggle output on compare match 0001—reserved 0000—capture/compare/PWM off
0x0#BE	CCPR1L	Least significant capture/compare/PWM1 register
0x0#BF	CCPR1H	Most significant capture/compare/PWM1 register

(Continued)

TABLE B.36 PIC18 REGISTER DEFINITION (CONTINUED)

ADDRESS	REGISTER	FUNCTION/BIT DEFINITION
0x0#C1	ADCON1	A/D control register1: Bit Function 7 ADFM—set to return result in right-justified format; reset to return result in left-justified format 6 ADCS2—upper bit of A/D conversion clock select; see ADCON0 for bit definition 5–4 Unused 3–0 PCFG3–PCFG0—A/D pin configuration select bits: Bit AN7 AN6 AN5 AN4 AN3 AN2 AN1 AN0 VR+ VR– 1111 D D D D VR+ VR– D A AN3 AN2 1110 D D D D D D D A Vdd Vss 1101 D D D D VR+ VR– A A AN3 AN2 1100 D D D A VR+ VR– A A AN3 AN2 1011 D D A A VR+ VR– A A AN3 AN2 1010 D D A A VR+ A A A AN3 Vss 1001 D D A A A A A A Vdd Vss 1000 A A A A VR+ VR– A A AN3 AN2 011x D D D D D D D D N/A N/A 0101 D D D D VR+ D A A AN3 Vss 0100 D D D D A D A A Vdd Vss 0011 D D D D VR+ A A A AN3 Vss 0010 D D D A A A A A Vdd Vss 0001 A A A A VR+ A A A AN3 Vss 0000 A A A A A A A A Vdd Vss
0x0#C2	ADCON0	A/D control register2: Bit Function 7–6 ADCS1–ADCS0—ADC conversion clock select, with ADCS2 from ADCON1: 111—internal RC oscillator 110—divide PIC microcontroller clock by 64

		101—divide PIC microcontroller clock by 16
		100—divide PIC microcontroller clock by 4
		011—internal RC oscillator
		010—divide PIC microcontroller clock by 32
		001—divide PIC microcontroller clock by 8
		000—divide PIC microcontroller clock by 2
5–3	CHS2–CHS0	ADC conversion channel:
		Select Bits
		111—AN7
		110—AN6
		101—AN5
		100—AN4
		011—AN3
		010—AN2
		001—AN1
		000—AN0
	2	GO/_DONE—set to start A/D conversion; reset by hardware when conversion before
	1	Unused
	0	ADON—set to turn on the ADC function
0x0#C3	ADRESL	Low byte of the ADC result
0x0#C4	ADRESH	High byte of the ADC result
0x0#C5	SSPCON2	MSSP control register2:
	Bit	Function
	7	GCEN—set to enable interrupt when general call address is received
	6	ACKSTAT—set when acknowledge received from I2C slave device
	5	ACKDT—reset to send acknowledge at the end of a byte receive
	4	ACKEN—acknowledge I2C sequence when set
	3	RCEN—set to enable I2C receive mode
	2	PEN—reset to initiate stop condition on I2C clock and data

(Continued)

TABLE B.36 PIC18 REGISTER DEFINITION (CONTINUED)

ADDRESS	REGISTER	FUNCTION/BIT DEFINITION
0x0#C6	SSPCON1	1 RSEN—set to initiate repeated start condition on I2C clock and data 0 SEN—set to initiate start condition on I2C clock and data MSSP control register1: Bit Function 7 WCOL—set if SSPBUF was written to while transmitting data or not in correct mode for transmit 6 SSPOV—set when SSP receive overflow occurs 5 SSPEN—enables pins for SSP mode 4 CKP—in SPI, set for idle clock high; in I2C mode, set to enable clock 3–0 SSPM3–SSPM0—SSP mode select: 1111—I2C slave mode, 10-bit address 1110—I2C slave mode, 7-bit address 110x—reserved 1011—I2C firmware-controlled master 1010—reserved 1001—reserved 1000—I2C master, $F_{osc}/[4 * (SSPAD + 1)]$ 0111—I2C slave mode, 10-bit address 0110—I2C slave mode, 7-bit address 0101—SSP slave, $\_SS$ disabled 0100—SSP slave, $\_SS$ enabled 0011—SPI master, clock = TMR2 0010—SPI master, $F_{osc}/64$ 0001—SPI master, $F_{osc}/16$ 0000—SPI master, $F_{osc}/4$
0x0#C7	SSPSTAT	MSSP status register:

		Bit	Function
		7	SMP—data sampled at end of data output time if set, else middle
		6	CKE—data transmitted on rising edge of SCK when set
		5	D/_A—when set, indicates last byte transferred was data; when reset, indicates last byte transferred was address
		4	P—set when stop bit detected
		3	S—set when start bit indicated
		2	R/_W—set when command received was a read
		1	UA—set when application must update SSPADD register
		0	BF—set when buffer is full in RX and when TX is in process
0x0#C8	SSPADD	MSSP address compare register	
0x0#C9	SSPBUF	MSSP data buffer	
0x0#CA	T2CON	TMR2 control register:	
		Bit	Function
		7	Unused
		6–3	TOUTPS3–TOUTPS0—TMR2 output postscaler:
			1111—16x
			1110—15x
			1101—14x
			1100—13x
			1011—12x
			1010—11x
			1001—10x
			1000—9x
			0111—8x
			0110—7x
			0101—6x
			0100—5x
			0011—4x
			0010—3x
			0001—2x
			0000—1x

(Continued)

TABLE B.36 PIC18 REGISTER DEFINITION (CONTINUED)

ADDRESS	REGISTER	FUNCTION/BIT DEFINITION
		2 TMR2ON—set to enable TMR2 1–0 T2CKPS1–T2CKPS0—TMR2 prescaler select bits: 1x—prescaler is 16 01—prescaler is 4 00—prescaler is 1
0x0#CB	PR2	TMR2 period compare register
0x0#CC	TMR2	TMR2 register
0x0#CD	T1CON	TMR1 control register: Bit Function 7 RD16—when set, enables 16-bit TMR1 operations 6 Unused 5–4 T1CKPS1–T1CKPS0—TMR1 input clock Prescaler Select 11—1:8 prescaler 10—1:4 prescaler 01—1:2 prescaler 00—1:1 prescaler 3 T1OSCEN—set to enable TMR1 oscillator 2 _T1SYNC—set to synchronize external clock input 1 TMR1CS—TMR1 clock source select; set to select external clock 0 TMR1ON—set to enable TMR1
0x0#CE	TMR1L	Low byte of TMR1
0x0#CF	TMR1H	High byte of TMR1
0x0#D0	RCON	Power-up status register Bit Function 7 IPEN—set to enable priority levels on interrupts 6 LWRT—set to enable tblwt instruction to internal memory

		5	Unused
		4	_RI—reset when the Reset instruction in software
		3	_TO—set after power-up, <code>clrwdt</code> , or sleep instruction
		2	_PD—set by power-up or <code>clrwdt</code> instruction; reset by sleep instruction
		1	_POR—reset if a power-on reset has occurred
		0	_BOR—reset if a brown-out reset has occurred
0x0#D1	WDTCN	Watchdog timer control register:	
		Bit	Function
		7–1	Unused
		0	SWDTEN—set to enable the watchdog timer if <code>_WDT_ON</code> is specified in <code>__CONFIG</code> .
0x0#D2	LVDCN	Low-voltage detect control register:	
		Bit	Function
		7–6	Unused
		5	IRVST—set to indicate low-voltage detect logic and will generate to interrupt
		4	LVDEN—set to enable low voltage detect
		3–0	LVDL3–LVDL0—specify the low-voltage detect limits.
			1111—External voltage used (LVDIN)
			1110—4.5 V min to 4.77 V max
			1101—4.2 V min to 4.45 V max
			1100—4.0 V min to 4.24 V max
			1011—3.8 V min to 4.03 V max
			1010—3.6 V min to 3.82 V max
			1001—3.5 V min to 3.71 V max
			1000—3.3 V min to 3.50 V max
			0111—3.0 V min to 3.18 V max
			0110—2.8 V min to 2.97 V max
			0101—2.7 V min to 2.86 V max

(Continued)

TABLE B.36 PIC18 REGISTER DEFINITION (CONTINUED)

ADDRESS	REGISTER	FUNCTION/BIT DEFINITION
0x0#D3	OSCCON	0100—2.5 V min to 2.65 V max 0011—2.4 V min to 2.54 V max 0010—2.2 V min to 2.33 V max 0001—2.0 V min to 2.12 V max 0000—1.8 V min to 1.91 V max
		Select PIC microcontroller clock source: Bit Function 7–1 Unused 0 SCS—reset to use primary oscillator; set to use TMR1's oscillator
0x0#D5	T0CON	TMR0 control register: Bit Function 7 TMR0ON—set to enable TMR0 6 T08Bit—set to enable TMR0 as an 8-bit timer; reset to enable TMR0 as a 16-bit timer 5 T0CS—set to make TMR0 clock source T0CKI pin; reset to use instruction clock 4 T0SE—set to make TMR0 increment on falling edge of clock 3 PSA—reset to assign TMR0 prescaler 2–0 T0PS2–T0PS0—TMR0 prescaler value: 111—1:256 prescaler 110—1:128 prescaler 101—1:64 prescaler 100—1:32 prescaler 011—1:16 prescaler 010—1:8 prescaler 001—1:4 prescaler 000—1:2 prescaler
	TMR0L	Low byte of TMR0

0x0#D7	TMR0H	High byte of TMR0
0x0#D8	STATUS	PIC microcontroller processor status register:
		Bit Function
		7–6 Unused
		4 N—set when the result has bit 7 set
		3 OV—set when the result overflows a two's complement number (bit 7 changes polarity inadvertently)
		2 Z—set when the least significant 8 bits of the result are all zero
		1 DC—set when the lower nybble of the addition/subtraction overflows
		0 C—set in addition when the result is greater than 0x0FF; reset in subtraction when the result is negative
0x0#D9	FSR2L	Low byte of FSR register 2
0x0#DA	FSR2H	High byte of FSR register 2
0x0#DB	PLUSW2	INDF2 consisting of FSR2 + WREG for address
0x0#DC	PREINC2	INDF2 with FSR2 incremented before access
0x0#DD	POSTDEC2	To INDF2 with FSR2 decremented after access
0x0#DE	POSTINC2	INDF2 with FSR2 incremented after access
0x0#DF	INDF2	Register pointed to by FSR2
0x0#E0	BSR	Bank select register, select register bank:
		Bit Function
		7–4 Unused
		3–0 BSR3–BSR0—bank select register bits
0x0#E2	FSR1L	Low byte of FSR register 1
0x0#E3	FSR1H	INDF1 consisting of FSR1 + WREG for address
0x0#E4	PLUSW1	INDF1 consisting of FSR1 + WREG for address

(Continued)

TABLE B.36 PIC18 REGISTER DEFINITION (CONTINUED)

ADDRESS	REGISTER	FUNCTION/BIT DEFINITION																		
0x0#E5	PREINC1	INDF1 with FSR1 decremented after access																		
0x0#E6	POSTDEC1	INDF1 with FSR1 incremented after access																		
0x0#E7	INDF1	Register pointed to by FSR1																		
0x0#E8	WREG	PIC microcontroller accumulator																		
0x0#E9	FSR0L	Low byte of FSR register 0																		
0x0#EA	FSR0H	High byte of FSR register 0																		
0x0#EB	PLUSW0	INDF0 consisting of FSR0 + WREG for address																		
0x0#EC	PREINC0	INDF0 with FSR0 incremented before access																		
0x0#ED	POSTDEC0	INDF0 with FSR0 decremented after access																		
0x0#EE	POSTINC0	INDF0 with FSR0 incremented after access																		
0x0#EF	INDF0	Register pointed to by FSR0																		
0x0#F0	INTCON3	Interrupt control register 3: <table><tr><th>Bit</th><th>Function</th></tr><tr><td>7</td><td>INT2IP—INT2 external interrupt priority; set for “high”</td></tr><tr><td>6</td><td>INT1IP—INT1 external interrupt priority; set for “high”</td></tr><tr><td>5</td><td>Unused</td></tr><tr><td>4</td><td>INT2IE—set to enable external INT2</td></tr><tr><td>3</td><td>INT1IE—set to enable external INT1</td></tr><tr><td>2</td><td>Unused</td></tr><tr><td>1</td><td>INT2IF—set when external INT2 requested</td></tr><tr><td>0</td><td>INT1IF—set when external INT1 requested</td></tr></table>	Bit	Function	7	INT2IP—INT2 external interrupt priority; set for “high”	6	INT1IP—INT1 external interrupt priority; set for “high”	5	Unused	4	INT2IE—set to enable external INT2	3	INT1IE—set to enable external INT1	2	Unused	1	INT2IF—set when external INT2 requested	0	INT1IF—set when external INT1 requested
Bit	Function																			
7	INT2IP—INT2 external interrupt priority; set for “high”																			
6	INT1IP—INT1 external interrupt priority; set for “high”																			
5	Unused																			
4	INT2IE—set to enable external INT2																			
3	INT1IE—set to enable external INT1																			
2	Unused																			
1	INT2IF—set when external INT2 requested																			
0	INT1IF—set when external INT1 requested																			
0x0#F1	INTCON2	Interrupt control register 2: <table><tr><th>Bit</th><th>Function</th></tr><tr><td>7</td><td>_RBPU—reset to enable PORTB pull-ups</td></tr><tr><td>6</td><td>INTEDG0—set for external INT0 on rising edge</td></tr></table>	Bit	Function	7	_RBPU—reset to enable PORTB pull-ups	6	INTEDG0—set for external INT0 on rising edge												
Bit	Function																			
7	_RBPU—reset to enable PORTB pull-ups																			
6	INTEDG0—set for external INT0 on rising edge																			

		5	INTEDG1—set for external INT1 on rising edge
		4	INTEDG2—set for external INT2 on rising edge
		3	Unused
		2	TMR0IP—high TMR0 interrupt request priority when set
		1	Unused
		0	RBIP—high PORTB change interrupt request priority when set
0x0#F2	INTCON	Interrupt control register:	
		Bit	Function
		7	GIE/GIEH—when set, enables all interrupt request sources (unmasked sources when IPEN is reset)
		6	PEIE/GEIL—enables all low-priority interrupt request sources when set
		5	TMR0IE—when set, enables TMR0 interrupt requests
		4	INT0IE—when set, enables INT0 interrupt requests
		3	RBIE—when set, enables PORTB change
		2	TMR0IF—when set, TMR0 interrupt request active
		1	INT0IF—when set, INT0 external interrupt request active
		0	RBIF—when set, PORTB change on interrupt request active
0x0#F3	PRODL	Low byte of Multiply instruction product	
0x0#F4	PRODH	High byte of Multiply instruction product	
0x0#F5	TABLAT	Table read and write buffer	
0x0#F6	TBLPTRL	Low byte of program memory table pointer	
0x0#F7	TBLPTRH	Middle byte of program memory table pointer	
0x0#F8	TBLPTRU	High byte of program memory table pointer	
0x0#F9	PCL	Low byte of PIC microcontroller program counter	
0x0#FA	PCLATH	Latched middle byte of PIC microcontroller program counter	
0x0#FB	PCKATHU	Latched high byte of PIC microcontroller program counter	

(Continued)

TABLE B.36 PIC18 REGISTER DEFINITION (CONTINUED)

ADDRESS	REGISTER	FUNCTION/BIT DEFINITION
0x0#FC	STKPTR	Stack pointer/index: Bit Function 7 STKFUL—bit set when stack is full or overflowed 6 STKUNF—bit set when stack underflows 5 Unused 4–0 SP4–SP0—stack pointer location bits
0x0#FD	TOSL	Low byte access to top of program counter stack
0x0#FE	TOSL	Middle byte access to top of program counter stack
0x0#FF	TOSU	High byte access to top of program counter stack

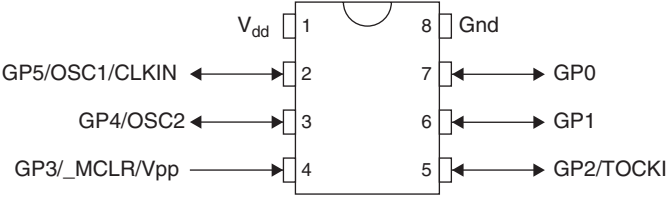


Figure B.2 Pinout for PIC12C50x/PIC12F50x microcontrollers.

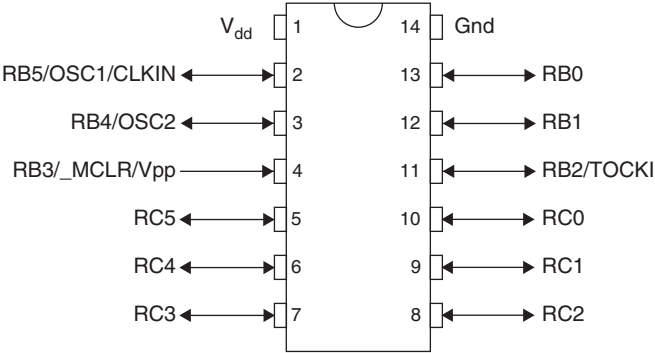


Figure B.3 Pinout for PIC16C505/PIC16F505 microcontrollers.

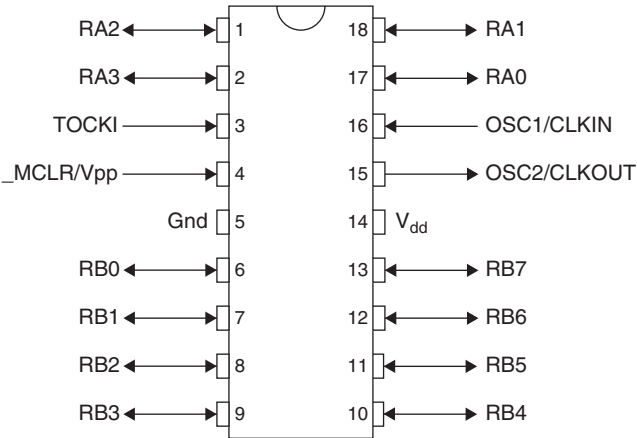


Figure B.4 Pinout for PIC16C54/PIC16F54/PIC16C57 microcontrollers.

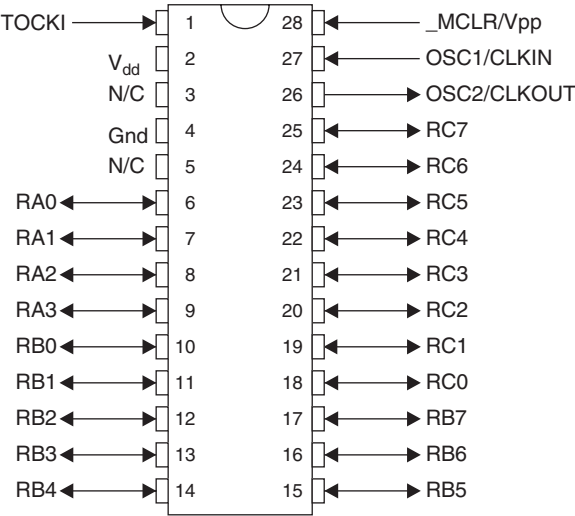
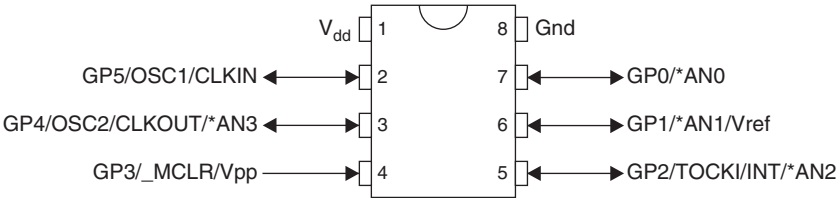
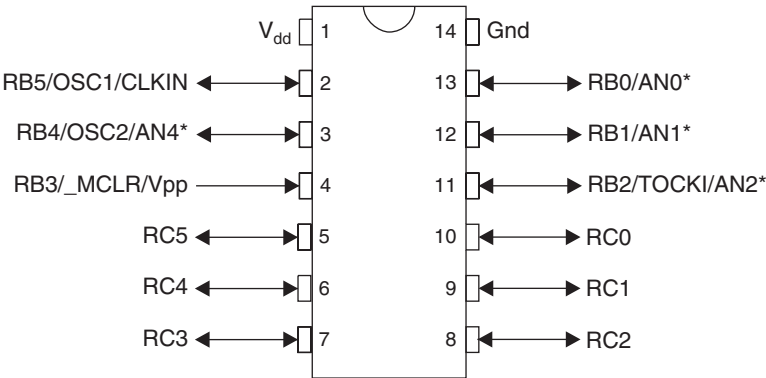


Figure B.5 Pinout for PIC16C55/PIC16C57 microcontrollers.



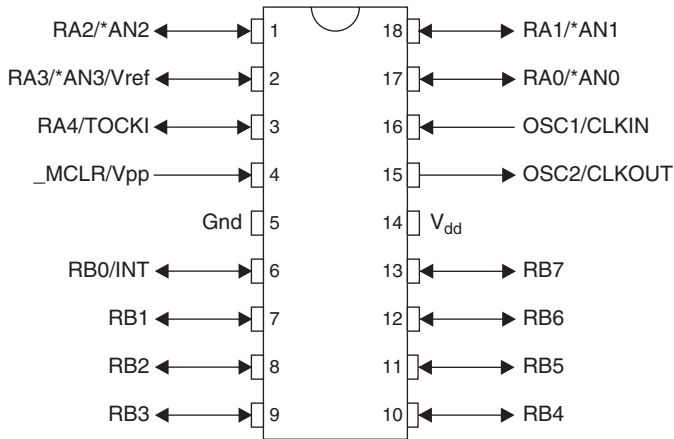
“*” - Indicates Analog I/O Pin

Figure B.6 Pinout for PIC12C6xx/PIC12F6xx microcontrollers.



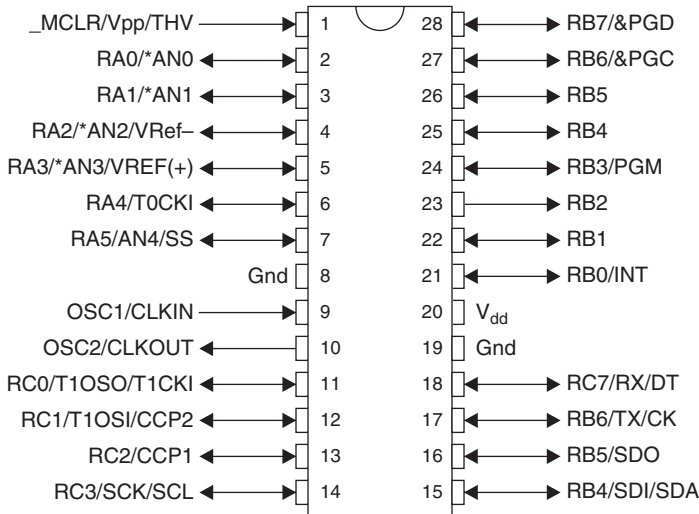
“*” - Indicates Analog I/O Pin

Figure B.7 Pinout for PIC16F69x microcontrollers.



“*” - Indicates Analog I/O Pin
For 16C62x and 16C71(x)

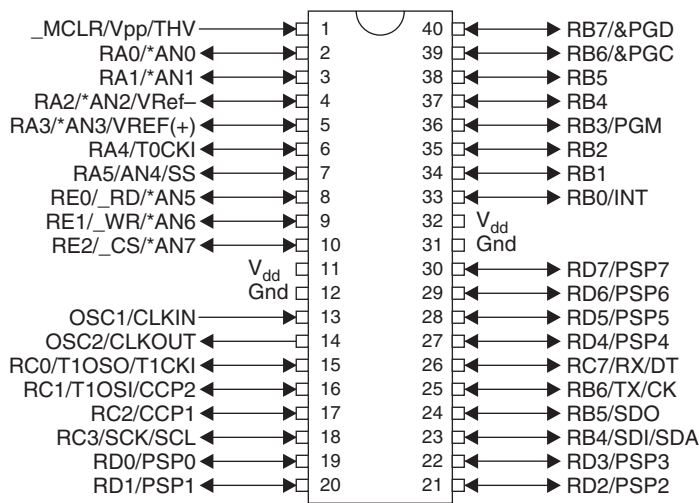
Figure B.8 Pinout for mid-range 18-pin PIC micro-controllers.



“*” - Indicates Analog I/O Pin

“&” - Indicates PIC16F87x Program/Emulator

Figure B.9 Pinout for 28-pin mid-range PIC micro-controllers.



“*” - Indicates Analog I/O Pin
“&” - Indicates PIC16F87x Program/Emulator

Figure B.10 Pinout for 40-pin mid-range PIC microcontrollers.

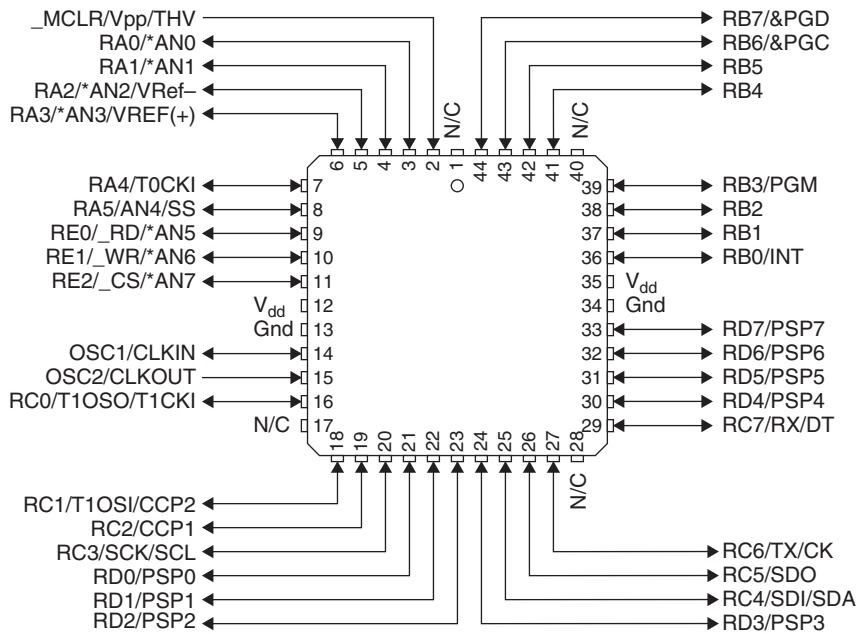


Figure B.11 Pinout for 44-pin PLCC SMT package mid-range PIC microcontrollers.

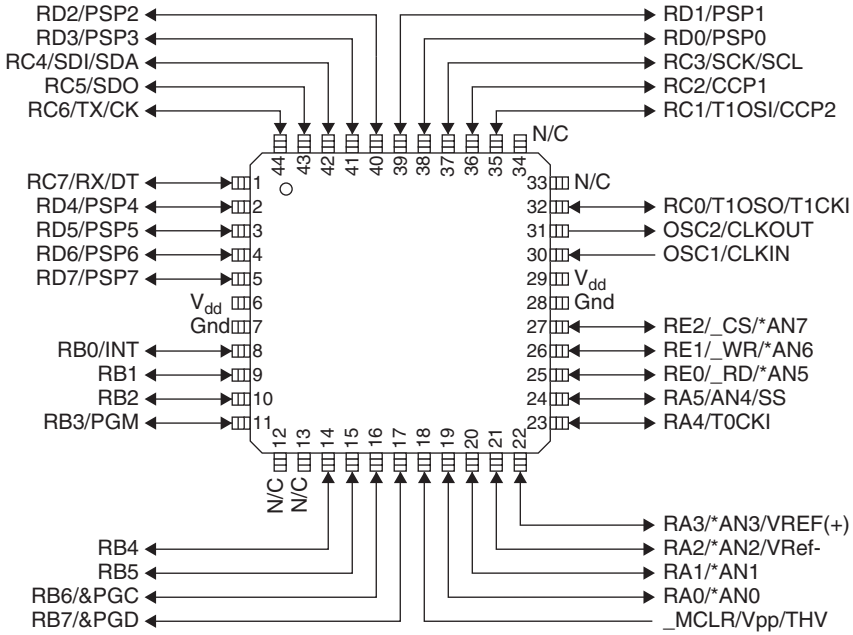


Figure B.12 Pinout for 44-pin QFP SMT package mid-range PIC microcontrollers.

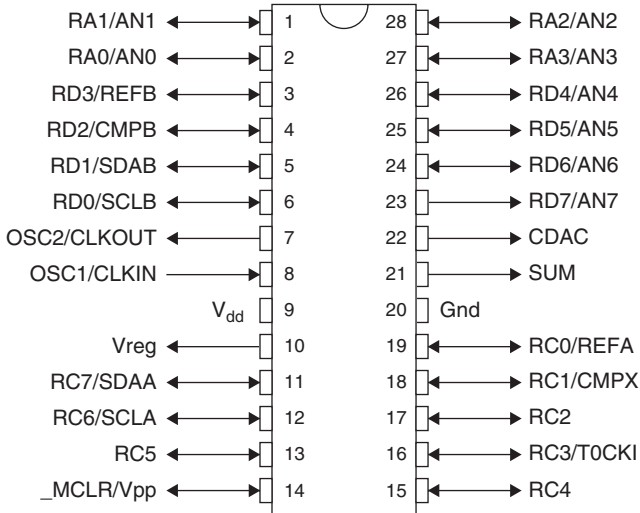


Figure B.13 The PIC14000 is a mixed-signal device with a different pinout than other 28-pin PIC microcontroller part numbers.

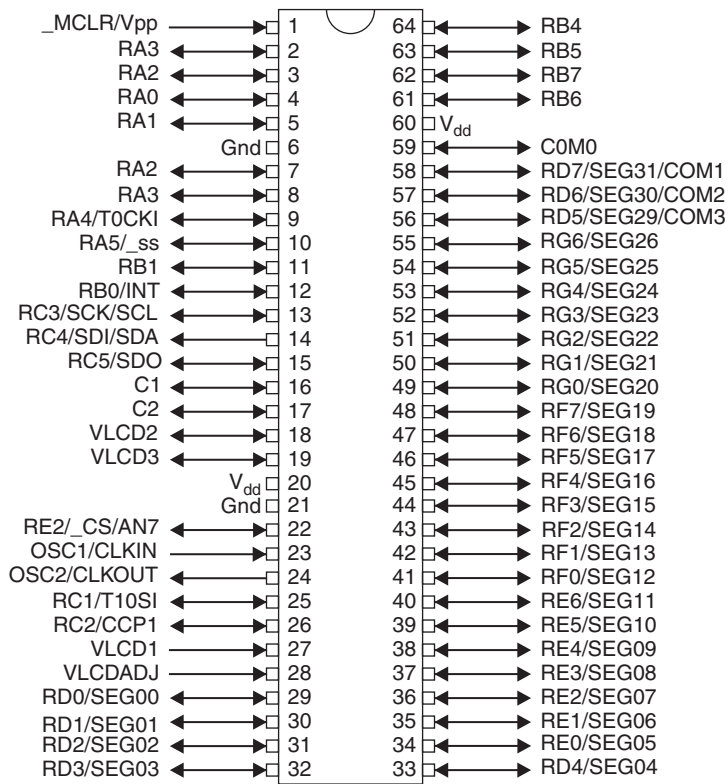


Figure B.14 A 64-pin DIP PIC microcontroller package.

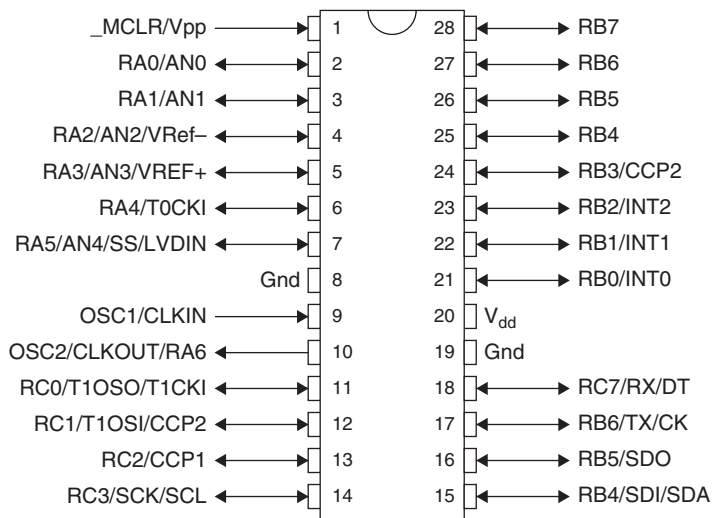


Figure B.15 A 28-pin PIC18 PTH pinout.

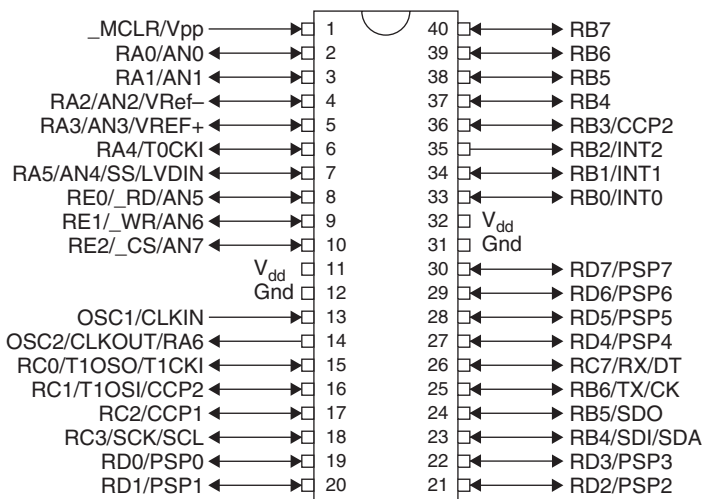


Figure B.16 A 40-pin PIC18 PTH pinout.

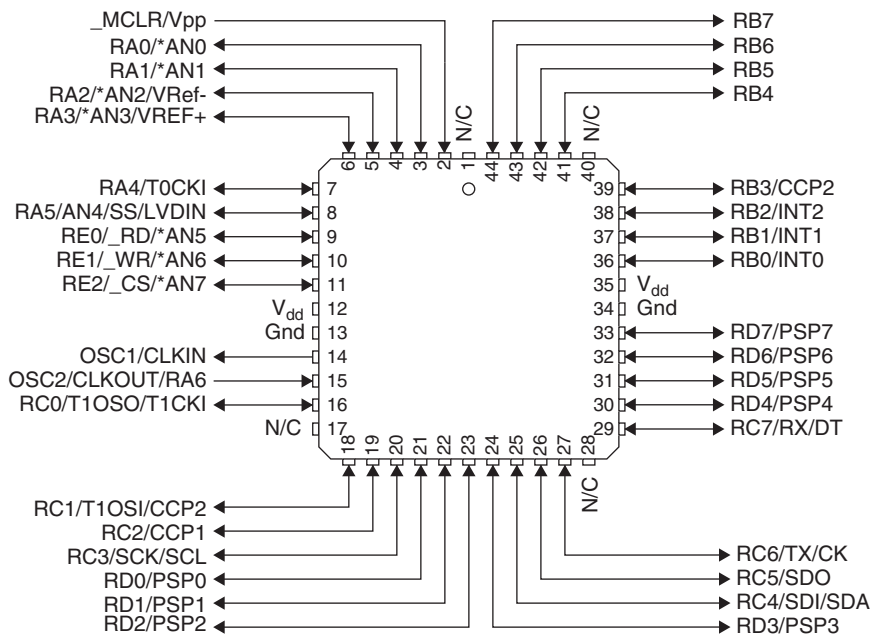


Figure B.17 Pinout for 44-pin PLCC SMT package mid-range PIC18 microcontrollers.

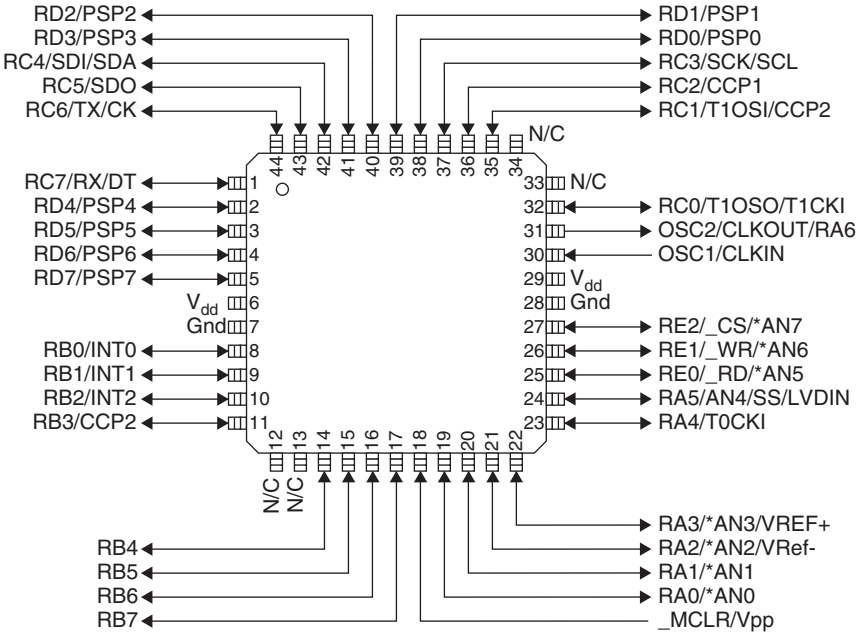


Figure B.18 Pinout for 44-pin QFP SMT package mid-range PIC18 microcontrollers.

USEFUL TABLES AND DATA

TABLE C.1 PHYSICAL CONSTANTS		
SYMBOL	VALUE	DESCRIPTION
AU	149.59787×10^6 km 92,955,628 miles	Astronomical unit (distance from the sun to the earth)
c	2.99792458×10^8 m/s 186,282 mi/s	Speed of light in a vacuum
e	2.7182818285	
ϵ_0	$8.854187817 \times 10^{-12}$ F/m	Permittivity of free space
eV	$1.60217733 \times 10^{-19}$ J	Electronvolt value
g	32.174 ft/s ² 9.807 m/s ²	Acceleration due to gravity
h	6.626×10^{-34} J · s	Planck constant
k	1.380658×10^{-23} J/K	Boltzmann entropy constant
m_e	$9.1093897 \times 10^{-31}$ kg	Electron rest mass
m_n	1.67493×10^{-27} kg	Neutron rest mass
m_p	1.67263×10^{-27} kg	Proton rest mass
pc	2.06246×10^5 AU	Parsec
π	3.1415926535898	Ratio of circumference to diameter of a circle
R	8.314510 J/(K · mol)	Gas constant
σ	5.67051×10^{-8} W/(m ² · K ⁴)	Stefan-Boltzmann constant

(Continued)

TABLE C.1 PHYSICAL CONSTANTS (CONTINUED)		
SYMBOL	VALUE	DESCRIPTION
u	$1.66054 \times 10^{-27} \text{ g}$	Atomic mass unit
$\mu\text{-0}$	$1.25664 \times 10^{-7} \text{ N/A}^2$	Permeability of vacuum
Mach 1	331.45 m/s 1087.4 ft/s	Speed of sound at sea level in dry air at 20°C
None	1480 m/s 4856 ft/s	Speed of sound in water at 20°C

TABLE C.2 AUDIO NOTES	
NOTE	FREQUENCY
G	392 Hz
G#	415.3 Hz
A	440 Hz
A#	466.2 Hz
B	493.9 Hz
C	523.3 Hz
C#	554.4 Hz
D	587.3 Hz
D#	622.3 Hz
E	659.3 Hz
F	698.5 Hz
F#	740.0 Hz
G	784.0 Hz
G#	830.6 Hz
A	880.0 Hz
A#	932.3 Hz
B	987.8 Hz

Notes Around middle C. Note that octave above is twice the note frequency and octave below is one-half the note frequency.

TABLE C.3 TOUCH TONE TELEPHONE FREQUENCIES			
FREQUENCY	1209 HZ	1336 HZ	1477 HZ
697 Hz	1	2	3
770 Hz	4	5	6
852 Hz	7	8	9
941 Hz	*	0	#

Electrical Engineering Formulas

Where V = voltage

i = current

R = resistance

C = capacitance

L = inductance

DC ELECTRONICS FORMULAS

Ohm's law:

$$V = iR$$

Power:

$$P = Vi$$

Series resistance:

$$R_t = R_1 + R_2 + \cdots$$

Parallel resistance:

$$R_t = 1/[(1/R_1) + (1/R_2) + \cdots]$$

Two resistors in parallel:

$$R_t = (R_1 + R_2)/(R_1 + R_2)$$

Series capacitance:

$$C_t = 1/[(1/C_1) + (1/C_2) + \cdots]$$

Parallel capacitance:

$$C_t = C_1 + C_2 + \cdots$$

Wheatstone bridge:

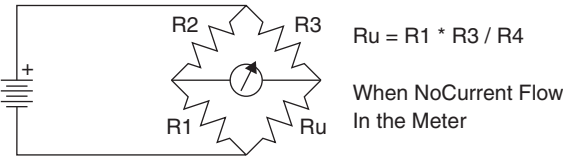


Figure C.1 The Wheatstone bridge is a useful circuit for measuring small resistances.

AC ELECTRONICS FORMULAS

Resonance:

$$\text{Frequency} = 1/[2\pi \times \text{SQRT}(L \times C)]$$

RC time constant:

$$\tau = R \times C$$

RL time constant:

$$\tau = L/R$$

RC charging:

$$V(t) = V_f(1 - e^{-t/\tau})$$
$$i(t) = i_f(1 - e^{-t/\tau})$$

RC discharging:

$$V(t) = V_i \times e^{-t/\tau}$$
$$i(t) = i_i \times e^{-t/\tau}$$

Transformer current/voltage:

Turns ratio = number of turns on primary (*p*) side/number of turns on secondary (*s*) side

$$\text{Turns ratio} = V_s/V_p = I_p/I_s$$

Transmission-line characteristic impedance:

$$Z_o = \text{SQRT}(L/C)$$

Mathematical Formulas

Frequency = speed/wavelength

For electromagnetic waves:

Frequency = c /wavelength

Perfect gas law:

$$PV = nRT$$

BOOLEAN ARITHMETIC

Identify functions:

$$A \text{ AND } 1 = A$$

$$A \text{ OR } 0 = A$$

Output set/reset:

$$A \text{ AND } 0 = 0$$

$$A \text{ OR } 1 = 1$$

Identity law:

$$A = A$$

Double negation law:

$$\text{NOT}(\text{NOT}(A)) = A$$

Complementary law:

$$A \text{ AND NOT}(A) = 0$$

$$A \text{ OR NOT}(A) = 1$$

Idempotent law:

$$A \text{ AND } A = A$$

$$A \text{ OR } A = A$$

Commutative law:

$$A \text{ AND } B = B \text{ AND } A$$

$$A \text{ OR } B = B \text{ OR } A$$

Associative law:

$$(A \text{ AND } B) \text{ AND } C = A \text{ AND } (B \text{ AND } C) = A \text{ AND } B \text{ AND } C$$

$$(A \text{ OR } B) \text{ OR } C = A \text{ OR } (B \text{ OR } C) = A \text{ OR } B \text{ OR } C$$

Distributive law:

$$A \text{ AND } (B \text{ OR } C) = (A \text{ AND } B) \text{ OR } (A \text{ AND } C)$$

$$A \text{ OR } (B \text{ AND } C) = (A \text{ OR } B) \text{ AND } (A \text{ OR } C)$$

De Morgan's theorem:

$$\text{NOT}(A \text{ OR } B) = \text{NOT}(A) \text{ AND } \text{NOT}(B)$$

$$\text{NOT}(A \text{ AND } B) = \text{NOT}(A) \text{ OR } \text{NOT}(B)$$

Note:

AND is often represented as multiplication, nothing between the terms, or the “.” else “*” characters between them.

OR is often represented as addition with “+” between terms.

NOT is indicated with a “-” or “!” character before the term; “~” is usually used to indicate a multibit bitwise inversion.

Mathematical Conversions

1 inch = 2.54 centimeters

1 mile = 1.609 kilometers

1 ounce = 29.57 grams

1 U.S. gallon = 3.78 liters

1 atmosphere = 29.9213 inches of mercury
= 14.6960 pounds per square inch
= 101.325 kilopascals

10,000,000,000 angstroms = 1 meter

1,000,000 microns = 1 meter

Tera = 1,000 giga
Giga = 1,000 mega
Mega = 1,000 kilo
Kilo = 1,000 units
Unit = 100 centi
Unit = 1,000 milli
Unit = 1,000,000 μ

1 hour = 3,600 seconds
1 year = 8,760 hours

ASCII

The ASCII definition uses the 7 bits of each ASCII character

TABLE C.4 ASCII CHARACTER TABLE								
BITS 3:0	BITS 6:4							
	CONTROL		CHARACTERS					
	000	001	010	011	100	101	110	111
0000	NUL	DLE	Space	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	\$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	HS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

ASCII CONTROL CHARACTERS

The ASCII control characters were specified as a means of allowing one computer to communicate with and control another. These characters are actually commands, and if the BIOS or MS-DOS displays or communications APIs are used with them, they will revert back to their original purpose. As I note below when I present the IBM extended ASCII characters, writing these values (all less than 0x020) to the display will display graphics characters.

Normally, only “carriage return” and “line feed” are used to indicate the start of a line. “Null” is used to indicate the end of an ASCIIZ string. “Backspace” will move the cursor back one column to the start of the line. The “Bell” character, when sent to MS-DOS, will cause the PC’s speaker to beep. “Horizontal tab” is used to move the cursor to the start of the next column that is evenly distributed by eight. “Form feed” is used to clear the screen.

TABLE C.5 ASCII CONTROL CHARACTER DEFINITIONS		
HEX	MNEMONIC	DEFINITION
00	NUL	“Null”—used to indicate the end of a string
01	SOH	Message “start of header”
02	STX	Message “start of text”
03	ETX	Message “end of text”
04	EOT	“end of transmission”
05	ENQ	“Enquire” for identification or information
06	ACK	“Acknowledge” the previous transmission
07	BEL	Ring the “bell”
08	BS	“Backspace”—move the cursor on column to the left
09	HT	“Horizontal tab”—move the cursor to the right to the next “tab stop” (normally a column evenly divisible by eight)
0A	LF	“Line feed”—move the cursor down one line
0B	VT	“Vertical tab”—move the cursor down to the next “tab line”
0C	FF	“Form feed” up to the start of the new page; for CRT displays, this is often used to clear the screen
0D	CR	“Carriage return”—move the cursor to the leftmost column
0E	SO	Next group of characters do not follow ASCII control conventions, so they are “shifted out”
0F	SI	The following characters do follow the ASCII control conventions and are “shifted in”

TABLE C.5 ASCII CONTROL CHARACTER DEFINITIONS (CONTINUED)

HEX	MNEMONIC	DEFINITION
10	DLE	“Data link escape”—ASCII control character start of an escape sequence; in most modern applications, “escape” (0x01B) is used for this function
11	DC1	Not defined—normally application-specific
12	DC2	Not defined—normally application-specific
13	DC3	Not defined—normally application-specific
14	DC4	Not defined—normally application-specific
15	NAK	“Negative acknowledge”—the previous transmission was not received properly
16	SYN	“Synchronous idle”—if the serial transmission uses a synchronous protocol, this character is sent to ensure that the transmitter and receiver remain synched
17	ETB	“End of transmission block”
18	CAN	“Cancel” and disregard the previous transmission
19	EM	“End of medium”—indicates end of a file; for MS-DOS files, 0x01A is often used instead
1A	SUB	“Substitute” the following character with an incorrect one
1B	ESC	“Escape”—used to temporarily halt execution or put an application into a mode to receive information
1C	FS	Marker for “file separation” of data being sent
1D	GS	Marker for “group separation” of data being sent
1E	RS	Marker for “record separation” of data being sent
1F	US	Marker for “unit separation” of data being sent

IBM EXTENDED ASCII CHARACTERS

The additional 128 characters shown in Fig. C.3 can do a lot to enhance a character mode application without having to resort to using graphics. These enhancements include special characters for languages other than English, engineering symbols, and simple graphics characters. These simple graphics characters allow lines and boxes in applications to be created.

Hex	0 x	1 x	2 x	3 x	4 x	5 x	6 x	7 x
x0		0	SP	0	@	P	,	p
x1	☺	1	!	1	A	Q	a	q
x2	☹	2	"	2	B	R	b	r
x3	♥	3	!!	3	C	S	c	s
x4	♦	4	\$	4	D	T	d	t
x5	♣	5	%	5	E	U	e	u
x6	♠	6	&	6	F	V	f	v
x7	·	7	'	7	G	W	g	w
x8	◼	8	(	8	H	X	h	x
x9	◻	9	)	9	I	Y	i	y
xA	◐	10	*	:	J	Z	j	z
xB	♂	11	←	+	;	K	[	k
xC	♀	12	└	,	<	L	\	l
xD	♪	13	↔	-	=	M	}	m
xE	🎵	14	▲	·	>	N	^	~
xF	☀	15	▼	/	?	O	—	△

Figure C.2 The first 128 characters of the IBM PC's character set are based on the 128 ASCII character set.

Hex	8 x	9 x	A x	B x	C x	D x	E x	F x
x0	Ç 128	É 144	Á 160	III 176	L 192	⌌ 208	α 224	≡ 240
x1	Ü 129	æ 145	í 161	⏏ 177	⊥ 193	⌌ 209	β 225	± 241
x2	é 130	Æ 146	ó 162	■ 178	⌌ 194	⌌ 210	Γ 226	≥ 242
x3	â 131	ô 147	ú 163	 179	⌌ 195	⌌ 211	π 227	≤ 243
x4	ä 132	ö 148	ñ 164	⌌ 180	— 196	⌌ 212	Σ 228	¡ 244
x5	à 133	ò 149	ñ 165	⌌ 181	⌌ 197	⌌ 213	σ 229	¡ 245
x6	á 134	û 150	ä 166	⌌ 182	⌌ 198	⌌ 214	ω 230	÷ 246
x7	ç 135	ù 151	ó 167	⌌ 183	⌌ 199	⌌ 215	τ 231	≈ 247
x8	ê 136	ÿ 152	ì 168	⌌ 184	⌌ 200	⌌ 216	Φ 232	° 248
x9	ë 137	ö 153	⌌ 169	⌌ 185	⌌ 201	⌌ 217	Θ 233	• 249
xA	è 138	ü 154	⌌ 170	⌌ 186	⌌ 202	⌌ 218	Ω 234	• 250
xB	ï 139	ç 155	½ 171	⌌ 187	⌌ 203	■ 219	δ 235	√ 251
xC	î 140	£ 156	¼ 172	⌌ 188	⌌ 204	■ 220	∞ 236	n 252
xD	ì 141	¥ 157	í 173	⌌ 189	= 205	■ 221	∅ 237	2 253
xE	ä 142	℞ 158	« 174	⌌ 190	⌌ 206	■ 222	∈ 238	■ 254
xF	À 143	f 159	» 175	⌌ 191	⌌ 207	■ 223	∩ 239	255

Figure C.3 The second 128 characters of the IBM PC's character set are used for graphics and special characters.

This page intentionally left blank

MISCELLANEOUS ELECTRONIC REFERENCE INFORMATION

Looking over this book, I realized that there are some topics that are important for you to understand as you work through the material that I have presented here. In this appendix I have provided you with a review of the basic electronic components and the test equipment that you will need to validate and debug your applications.

Basic Electronic Components and Their Symbols

The two basic devices in any circuit are the *power supply* and the *load*. The power supply provides an electron source with the electronics given a specific voltage potential for a maximum rated current. Voltage can be either unchanging (*direct current* or *dc*) or changing (*alternating current* or *ac*) according to a sine wave above and below 0 V.

The symbol used for power will be the “battery” symbol shown in Fig. D.1, with the positive terminal connected to Vcc (or Vdd for CMOS devices such as the PIC[®] microcontroller). The negative terminal is connected to ground (GND) (or Vss for CMOS). The battery symbol is meant to represent the multiple plates in a physical battery; I always remember positive because the symbol kind of looks like an actual cylindrical radio battery with the small ground (which is positive in the physical device) connected backwards.

A *resistor* can characterize the load that in a device that impedes current flow through a circuit. Its symbol is shown in Fig. D.2.

Resistors are not polarized and normally have a series of four or five color bands printed on them. These bands are used to specify the resistance of the circuit using the formula

$$\text{Resistance} = ((\text{Band1} \times 100) + (\text{Band2} \times 10) + (\text{Band3} \times 1)) \times 10^{**} \text{Band4}$$

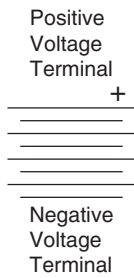


Figure D.1 Battery/power supply symbol.

Table D.1 lists the values for each band and what they mean in the resistor specification formula.

Variable resistors or *potentiometers* are used to change the resistance of a circuit or to provide a user-selected voltage. Its symbol is shown in Fig. D.3.

Pushbuttons normally are referred to as *momentary-on* or *momentary-off switches*, and their symbols (i.e., operation) are shown in Fig. D.4. Momentary-on switches close the circuit when they are pressed and open the circuit when they are released. Momentary-off work in the opposite manner, with the circuit opened when the button is pushed.

Slide switches are used to make a specific contact based on the position in which the switch is left. The most basic switch has one or two *throws* that result in a closed circuit. Figure D.5 shows single- and double-throw switches.

There can be more than one throw, or position, in the switch, and multiple switches can be ganged together as multipole switches. There are many different combinations that can be built out of these basic types of switches. In this book I am mostly concerned with single-pole, single-throw switches (which are known as SPST) for turning on and off power or to control basic signals. The symbol for the SPST is shown in Fig. D.6.

You will have to wire multiple boards together, and this is accomplished by the use of *connectors*. There seems like there are an infinite variety of connectors available. In many cases, the application developer selects the connectors that he or she feels most comfortable with. For this reason, except for some very specific instances where high current or convenience is required, I have not specified the type of connector to be used. The connector symbol name is J.

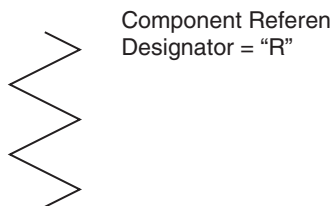


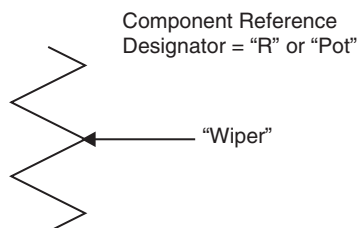
Figure D.2 Resistor symbol and component designation.

TABLE D.1 RESISTOR COLOR CODE CHART

NUMBER	COLOR	BAND 1	BAND 2	BAND 3	BAND 4	OPTIONAL BAND 5
0	Black	N/A	0	0	10^0	N/A
1	Brown	1	1	1	10^1	1% tolerance
2	Red	2	2	2	10^2	2% tolerance
3	Orange	3	3	3	10^3	N/A
4	Yellow	4	4	4	10^4	N/A
5	Green	5	5	5	10^5	0.5% tolerance
6	Blue	6	6	6	10^6	0.25% tolerance
7	Violet	7	7	7	10^7	0.1% tolerance
8	Gray	8	8	8	10^8	0.05% tolerance
9	White	9	9	9	10^9	N/A
N/A	Gold	N/A	N/A	N/A	10^{-1}	5% tolerance
N/A	Silver	N/A	N/A	N/A	10^{-2}	10% tolerance

Fuses can be used within your application to limit current draw and protect circuits, wiring, and power supplies. Like home fuses, electronic fuses are rated in amperes and are available in a number of different board-mountable or in-line packages and cartridges. The symbol is shown in Fig. D.7. Fuses are not rated in regard to the voltage that can be applied to them.

Capacitors are designed for storing electric charge. Their design typically consists of two metal plates separated by a nonconductive material called a *dielectric*. The capacitor symbol is shown in Fig. D.8. The charge placed on a capacitor is measured in farads (symbol F), and are measured in the units of coulombs/volt. A *coulomb* is a measure of 1.6×10^{19} electrons, making 1 F a very large number. Most capacitors are measured in millionths (μF) or trillionths (pF) of a farad.

**Figure D.3** Potentiometer symbol.

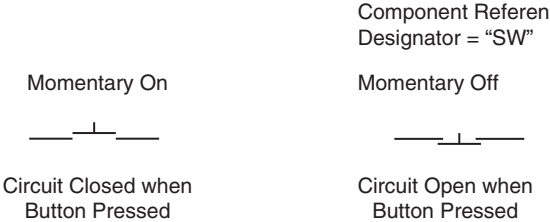


Figure D.4 The two types of momentary switches (on and off).

The type of dielectric employed in a capacitor is usually used to reference the component; the most common types of capacitors are:

- Ceramic disk
- Polyester
- Tantalum
- Electrolytic

Capacitors are either marked with their value or use a three-digit numbering scheme similar to the one used for resistors but with the value being referenced in picofarads. For example, a capacitor marked with 103 is understood to be

$$10 \times 10^3 \text{ pF} = 10,000 \text{ pF} = 0.01 \text{ }\mu\text{F}$$

Where a capacitor stores energy in the form of a charge, *inductors* (or *coils*) store energy in the form of a magnetic field. Inductors (Fig. D.9) are not often used in PIC microcontroller circuits except for switch-mode power supplies, which increase or decrease incoming power using the characteristics of coils to resist changes in current flow. Like the capacitor's farad unit of measurement, the normal values of an inductor are in millionths of a *henry* for the devices to be effective in a circuit.

Modern application switching is accomplished by *semiconductors*, so named because their ability to conduct current can be controlled in different situations. The most basic of these circuits is the *diode* (Fig. D.10), which will pass current in only one direction.

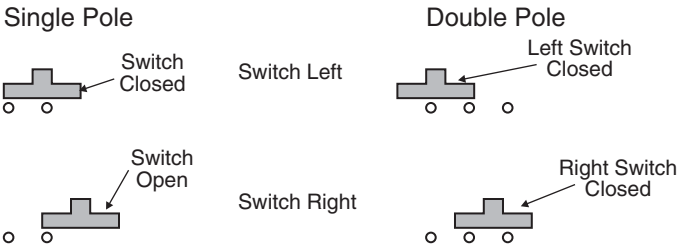


Figure D.5 Throw switches can have either a single set or two sets of contacts.

SPST Switch
Component Referen
Designator = "SW"

Figure D.6 The schematic symbol for a single-pole, single-throw switch.

Component Referen
Designator = "F"

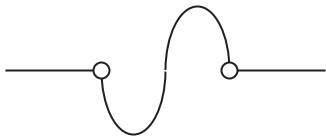
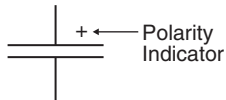


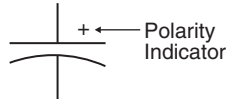
Figure D.7 The fuse's schematic symbol.

Component Referen
Designator = "C"



Polarity Indicator

In Some References, the Symbol is:



Polarity Indicator

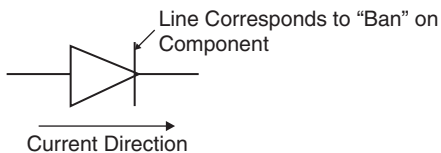
Figure D.8 The capacitor schematic symbol.

Component Referen
Designator = "L"



Figure D.9 The inductor's schematic symbol.

Component Reference
Designator = "CR"



Line Corresponds to "Ban" on Component

Current Direction

Figure D.10 The diode's schematic symbol.
Note that current flow through the inductor is indicated by the band around the device.

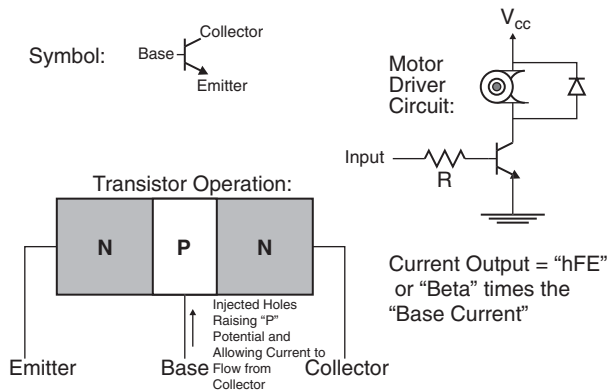


Figure D.11 NPN transistor schematic symbol, example application, and physical device.

The most basic transistor is the *bipolar NPN transistor* (Fig. D.11), which consists of two pieces of N-doped silicon sandwiching a piece of P-doped silicon. N-doped silicon has a material added to it that causes electrons to be available in the crystal lattice. P-doped silicon has a material added that causes there to be an affinity for accepting electrons in the crystal lattice.

Bipolar transistors actually are variable switches in which the amount of current passing through the *collector* and *emitter* can be controlled by controlling the amount of current passed through the *base*. Figure D.11 shows a simple circuit for an inverter. When current is passed through the base to the emitter, the collector will pass the amount current to the emitter defined using the formula

$$I_{ce} = I_{be} \times hFE$$

where I_{ce} is the current through the collector and emitter, and I_{be} is the current through the base and the emitter. hFE , which is also known as *beta*, is the multiplication factor for the current flowing through the base and emitter to the collector and emitter. For a common transistor such as the 2N3904 (of which I use a few of in this book), hFE is in the range of 150 to 200.

Along with the NPN bipolar transistor that is turned on when current is applied to it, there is the *PNP bipolar transistor* that will pass current between its collector and emitter when current is drawn from it. This device is shown in Fig. D.12 and behaves similarly to the NPN transistor, with the amount of current passing through the collector and emitter being proportional to the current being drawn from it.

The most popular form of transistors used today are the *metal oxide silicon field effect transistor* (MOSFET). These components are much smaller and use a fraction of the power of the bipolar transistors. The basic type of MOSFET transistor is the *N-channel device* that is shown in Fig. D.13. When a positive voltage is applied to the *gate*, an electrical field is set up in the P-doped silicon substrate. This field will

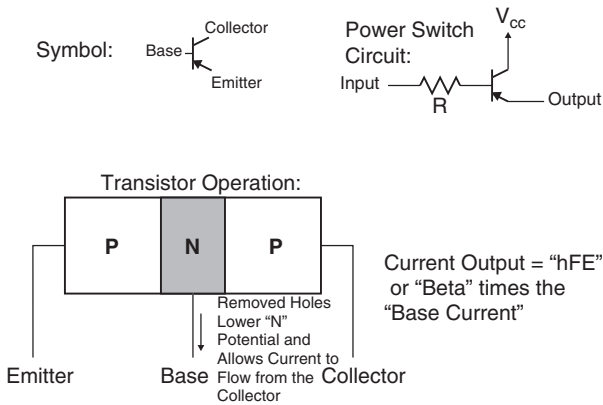


Figure D.12 PNP transistor schematic symbol, example application, and physical device.

cause the electrical characteristics in the substrate below the gate (the *conducting region* in Fig. D.13) to mimic those of N-doped silicon.

The size of the conducting region can be controlled by the amount of voltage applied to the gate. For digital applications, a set amount of voltage is applied constantly, making the N-channel behave like an on/off switch.

MOSFETs usually are defined by the resistance between the drain and the source when the transistor is on or conducting. For an N-channel device, this resistance normally is measured in fractions of an ohm.

N-channel MOSFET transistors have a complementary device, the *P-channel MOSFET*. These transistors normally conduct when a zero voltage is applied to them because a conducting "tub" of N-doped silicon has been placed under the gate, as I've shown in Fig. D.14. When a positive voltage is applied to the gate, the N-doped silicon changes its electrical characteristics to P-doped silicon and stops conducting.

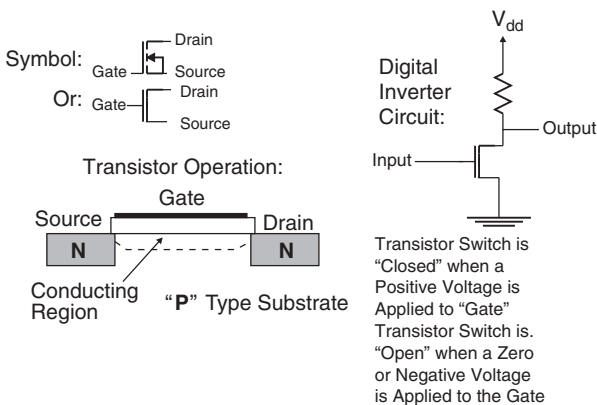


Figure D.13 N-channel MOSFET information with schematic symbols.

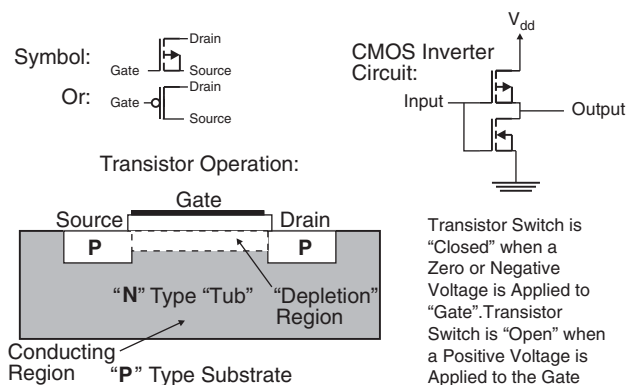


Figure D.14 P-channel MOSFET information.

When the *depletion region* grows to the point where the entire N-doped silicon behaves like P-doped silicon underneath the gate, the transistor is no longer conducting and is turned off.

Varying the amount of voltage applied to the gate can control this “pinching off” of the N-channel “tub.” The P-channel MOSFET has an “on” resistance that at several ohms or more is much higher than that of the N-channel MOSFET, and it can be difficult to match the two devices for analog applications.

Where P-channel MOSFETs have found a niche is in working with N-channel MOSFETs in CMOS (complementary metal oxide silicon) logic. As I’ve shown in Fig. D.14, a P-channel MOSFET can be combined with an N-channel MOSFET to produce an inverter and not require the current-limiting resistor of the NMOS inverter. When you see the symbols for MOSFET transistors, it is easy to forget which is which. Always remember that the N-channel device has the arrow going in (i.e., “iN”). This is not true for NPN transistors, where the arrow indicates current output.

In CMOS circuits, the only real opportunity for current flow is when the gates are switching, and stored charges are passed through the transistors. This accounts for the phenomenally low current (and power) requirements of CMOS circuits and why the current requirement goes up when the clock frequencies go up. As number of switch transitions per second increases, the amount of charge moved within the chips goes up proportionally. This charge movement averages out to a current flow.

Test Equipment

As you gain proficiency in working with electronics in general and the PIC microcontroller specifically, you will find the following tools to be useful in validating and debugging your applications. While these tools may seem specific to debugging hardware, they can be very useful when you have an application software problem that you are trying to debug.

DIGITAL MULTIMETERS

Digital multimeters (DMMs) are invaluable tools for checking voltage (and logic levels) in an application, as well as for measuring the current and the values of some components. Inexpensive DMMs can be bought for as little as \$20 (USD).

The output of a DMM is a three- or four-digit numerical display. In many devices, the measurement is selected via a switch on the DMM, and the display's decimal point moves over. If the value is too large for the display, something like an "I" in the left most digit will be displayed with the other digits blanked out.

Some DMMs have up to six digits, but I must stress that as you work with the DMMs for your own PIC microcontroller or digital electronics projects, never use more than three digits (and ideally not more than two). The extra accuracy is not needed and adds a lot to the cost of the instrument.

Each digit represents a power of 10. For a three-digit display, the value is supposedly accurate to one part in a thousand. Greater than one part per thousand accuracy is required only very rarely in very specialized cases, and when this level of accuracy is required, then precision power supplies, crystals, and components would be used alone with a specially calibrated DMM. It may be interesting to see the differences between devices at ten-millionths of a volt, but this accuracy is not practical for any PIC microcontroller applications that I can think of.

DMMs are not "fast response" devices. You may find that it can take as long as 10 seconds before the display stabilizes on a dc voltage value. The long time needed to stabilize the output means that the DMM is not capable of measuring changing signals unless the signal changes once every few seconds.

Along with the ability to measure current, voltage, and resistance, DMMs are available with the following features and capabilities:

- Perform "autoranging" while measuring a parameter
- Measure capacitance
- Measure temperature
- Measure a bipolar transistor's beta
- Perform diode checks.
- Measure frequency

These features are nice to have but not critical for the projects in this book or most beginning applications.

LOGIC PROBES

Along with a DMM, I consider a *logic probe* to be a basic tool for anyone developing digital circuits. A logic probe is a pencil-like device that can be used to check different parts of a board and return a visual/audio signal that indicates whether or not a signal is logic high, low, or tristated (or not connected to a driver or a pulled up or pulled down input).

There are usually two controls on a logic probe. The Pulse/Continuous switch will cause a light-emitting diode (LED) to flash when a changing signal is encountered or

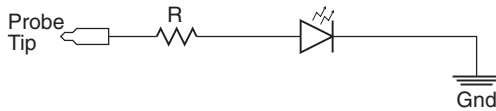


Figure D.15 An LED would seem to be able to provide the basic functions of a logic probe.

a single LED to light for a continuous level. Along with an LED, many logic probes have a speaker in them that will provide beeps and tones so that you can tell what is going on without looking at the LEDs. The speaker audio output is a feature I highly recommend getting.

The other switch is TTL/CMOS and selects which voltage threshold should be used. Normally TTL is 1.4 V, and CMOS is 2.5 V.

Good-quality logic probes can be bought for as little as \$20, and I recommend that you buy one rather than try to build one yourself. You may think that you could build one yourself with a resistor and an LED like the circuit shown in Fig. D.15.

The circuit in Fig. D.15 has three problems:

- 1** It doesn't detect high or low outputs, just high.
- 2** If a circuit doesn't have enough current capacity, the LED won't light.
- 3** There is no indication on level transition.

To detect high, low, and tristate/no connections, another LED will have to be added to the circuit. If the single LED is not lit, then the pin could be attached to a low, tristate, or low-current high output. To fix this, voltage comparators capable of driving a LED should be used in the circuit, as shown in Fig. D.16. In this circuit, if the probe is driven high or low by a pin, one of the LEDs will light. A switch-selectable "threshold reference" voltage divider could be put in for a TTL/CMOS selection.

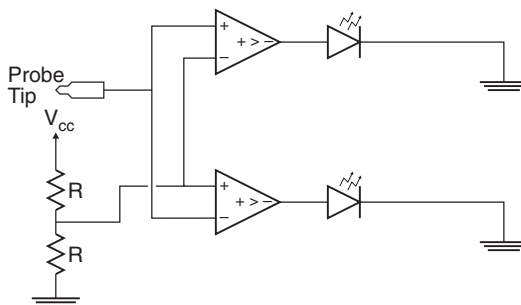


Figure D.16 Adding a comparator creates a logic probe circuit with the capabilities of detecting high and low logic levels.

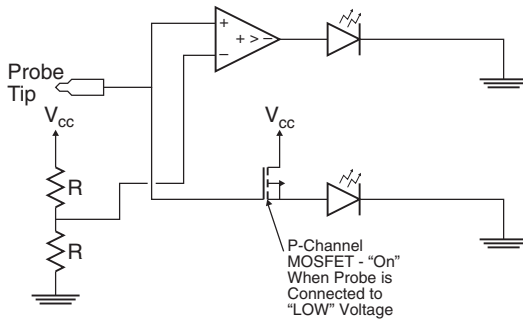


Figure D.17 Enhanced logic probe circuit that uses a P-channel MOSFET to detect low-voltage conditions.

The circuit as it stands will not detect open conditions; for this feature, some kind of MOSFET sensor is required. A P-channel MOSFET could be put in to sense when the logic level is low, as shown in Fig. D.17. When the probe pin is low, the P-channel MOSFET will turn on, passing current to the LED.

To add a transition output, a *single shot* should be triggered by each of the LED drivers. The modification to the logic probe circuit shown in Fig. D.18 will cause a third LED to flash when the LED drivers become active.

Looking at this, you're probably confident that you could build this circuit easily. You probably could, although there may be some problems with getting the circuit to fit in a small enough package to be useful. In addition, if you did attempt to build one yourself; it probably would cost as much in parts as buying a completed unit.

For these reasons, I recommend that you avoid the headaches of building one and just buy one ready made. As unlikely as this sounds, the Radio Shack logic probe contains all the features I've listed here and is quite inexpensive (it is \$20), making it the device you should look at first. I've owned three of them over the years—they are as good as or better than other logic probes costing twice as much or more from more "prestigious" manufacturers.

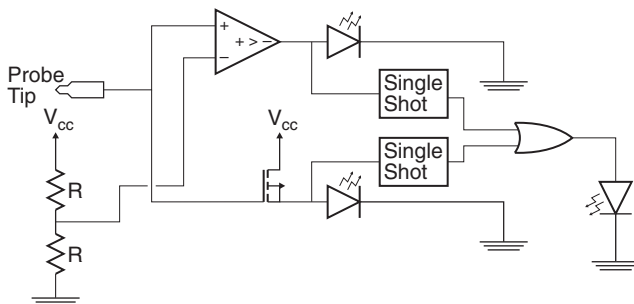


Figure D.18 Final logic probe circuit with single shots to detect logic-level switching.

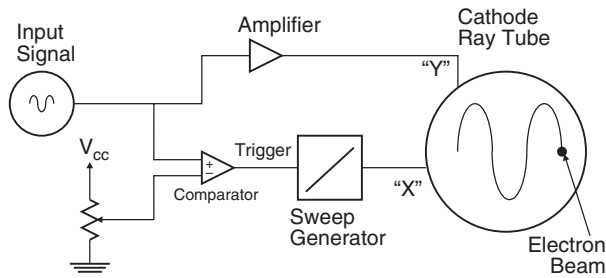


Figure D.19 The basic circuitry of an oscilloscope is actually quite simple.

OSCILLOSCOPES

There are two different types of oscilloscopes. The most basic one is the analog oscilloscope that starts a sweep generator when a voltage trigger level is reached. The sweep generator causes a cathode-ray tube electron beam to move across the screen. The continuing signal is drawn on the circuit screen, deflected up and down proportionately to the voltage of the input signal. This operation is shown in Fig. D.19.

The circuit screen is marked off in *gradicules*, which indicate the time between features on the screen and their magnitudes. The oscilloscope itself is calibrated so that these values can be read off the screen simply by counting the gradicules.

Multiple signals can be displayed by alternating the position of the single electron beam. One way is to alternate the source displayed each time the oscilloscope is triggered. The second method is to change which source is being displayed as the beam sweeps (this is known as *chop*). Figure D.20 shows how these methods work and appear on an oscilloscope display. Neither method is “perfect” for looking at multiple signals, and in either case, important data could be lost or not visible.

There are two problems with the analog oscilloscope when it comes to working with PIC microcontroller (and other digital) circuits. The first is that each time the electron beam sweeps, the signal on the display is very dim and fades quickly. If the intensity is turned up, the phosphors on the backside of the CRT could be damaged. The second problem is that it is very easy to miss single events in multiple sources

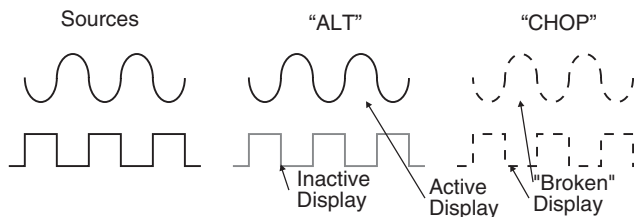


Figure D.20 When displaying multiple traces on an oscilloscope screen, they are either displayed alternately or the display alternates between them.

because the `alt` and `chop` displays miss the changes. These problems get worse with faster sweep speeds and multiple signals.

Over the years, some manufacturers have come up with ways to store signals and improve phosphor performance, but still there are lot of problems working with an analog oscilloscope with digital signals. Analog oscilloscopes are excellent for very low-speed, repeating signals, such as you would find in your stereo with a *reference signal*, but they will not be helpful for digital signals.

Instead of using analog oscilloscope for digital applications, I would recommend using a digital oscilloscope, more properly known as a *digital storage oscilloscope* (DSO) or *digitizer*. These oscilloscopes save incoming signal values digitally and display them in a similar format as an oscilloscope. The block diagram for a digital oscilloscope is shown in Fig. D.21. In this circuit, the incoming circuit is digitized by the Flash ADC and stored into memory. When the triggered event has finished, the data is read from memory and displayed on the oscilloscope's display.

The obvious advantage of a digital oscilloscope is its ability to capture and display an event's multiple signals without any data being lost or difficult to observe. There is also the added advantage of being able to transfer the data from the digital oscilloscope to a PC without having to take a picture of the screen and then digitize it (as you would with an analog oscilloscope). The oscilloscope pictures shown in this book were transferred via RS-232 from my TDS-210 oscilloscope and stored into **TIFF** format files for publishing.

Events also can be displayed more easily on a digital oscilloscope because many models have the capability to sample continuously, and then when the trigger point is reached, the counter just continues to the end of the sample. This is shown as throughout this book with the various oscilloscope pictures that are presented.

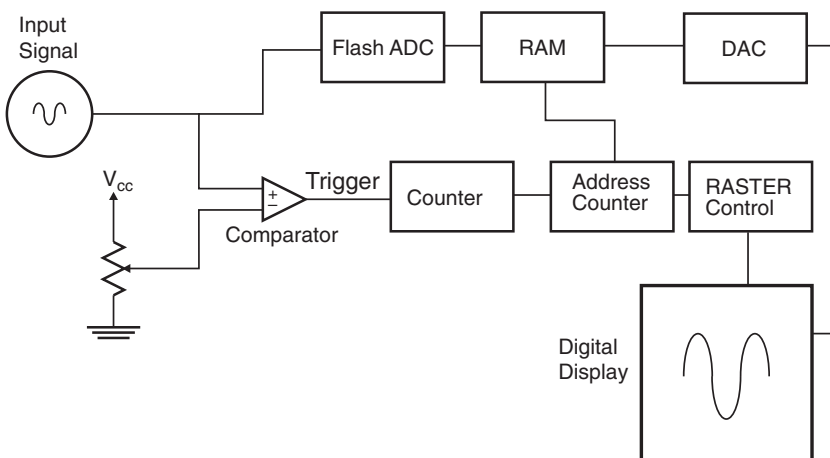


Figure D.21 A digital storage oscilloscope records a waveform in memory, allowing it to be output immediately to the oscilloscope's screen or saved for later analysis.

LOGIC ANALYZERS AND STATE ANALYZERS

A logic analyzer borrows a lot of the technology from the digital oscilloscope discussed in the preceding section. The logic analyzer, as shown in Fig. D.22, uses the incoming multiple-line data pattern for the trigger and then loads its memory with the bit data until it is full. Logic analyzers tend to be quite expensive, although there is a trend today by some manufacturers to include logic analyzer capabilities with a digital oscilloscope.

Instead of recording digital representations of analog data, digital logic values are recorded at various points in the circuit. Logic analyzers are much more difficult to use within a circuit because of the number of connections to be made and the need to label signals on the display and come up with a pattern to compare against to trigger the sample. To help cut down on the need to reenter data, most logic analyzers have a disk or nonvolatile memory built into them to save specific steps.

Logic analyzers can present data in two different ways. A graphical logic display, such as Fig. D.23, is useful in looking at signals and comparing them with simulator output or even circuit drawings.

The second type of display is a *state display*, and a clock is used to strobe data into the memory rather than saving the data according to a logic analyzer internal clock. This type of display is best suited for monitoring the execution of an application, and if the data and addresses are captured, it can be a fast and inexpensive way of creating a processor emulator/tracer.

The two most critical parameters of the logic analyzer are the speed at which data can be processed and the depth of memory behind each pin. The need for fast memory should be obvious.

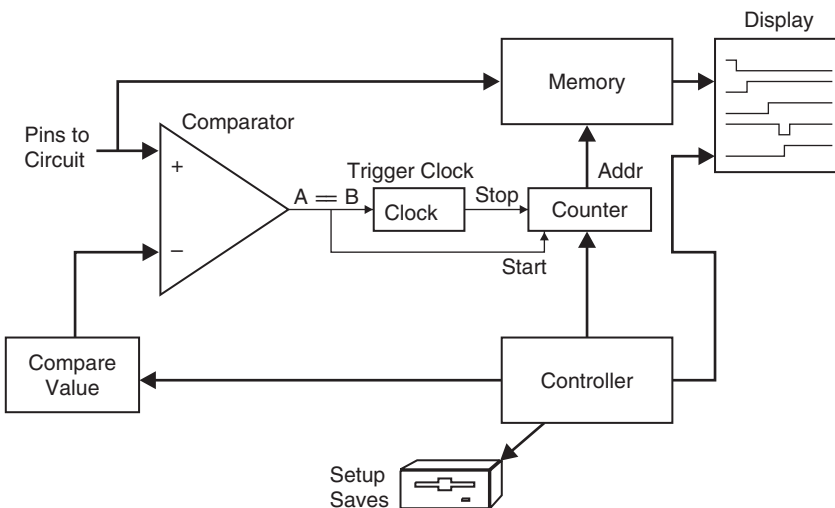


Figure D.22 The logic analyzer's block diagram is very similar to that of the digital storage oscilloscope except that the output of the multiple lines is stored for later display rather than the limited oscilloscope inputs.

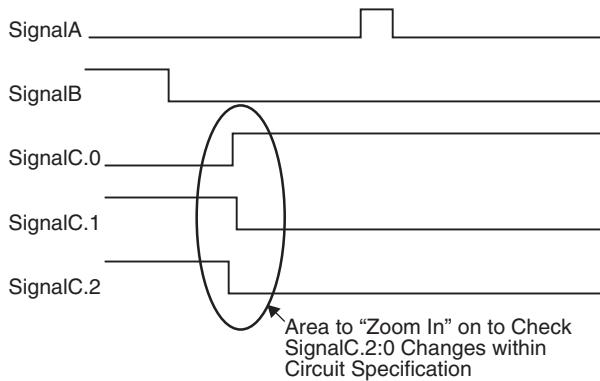


Figure D.23 Depending on the time base, you may find that there are situations in the logic analyzer where you cannot readily determine the relationship between certain signals without changing the time base or getting additional samples.

The depth relates to how much data can be stored and, more important, how much you can zoom in on a problem. In Fig. D.23, channels SignalC1, SignalC2, and SignalC3 are supposed to all change at the same point. To confirm this, you would want to check these transitions by zooming in to see what is actually happening. A logic analyzer with insufficient depth would not be able to display any data except at current resolution. To go to a higher resolution, another "picture" would have to be taken.

This page intentionally left blank

BASIC PROGRAMMING LANGUAGE

The Beginner's All-purpose Symbolic Instruction Code (BASIC) language has been used by new computer programmers for over 30 years. It became a staple of new computer programmers when it was included with the Apple II computer in the mid-1970s. One of the aspects that made BASIC popular was its integrated development environment—the program itself could be used for entering, running, and debugging applications. Owing to the nature of the PIC[®] microcontroller, this capability is rarely available except in deliberate applications such as BASIC87x.

In its original form, the BASIC language is somewhat unstructured, although Microsoft has done a lot to enhance the language and make it easier to work with, starting with GW BASIC that was shipped with the first IBM PCs, QBASIC, and then Visual Basic. In the first part of this appendix I would like to introduce you to the BASIC language, with Microsoft's extensions, and following this, PICBASIC (which is provided by microEngineering Labs).

BASIC variables do not have to be declared except in specialized cases. The variable name itself follows normal conventions of a letter or “_” character as the first character, followed by alphanumeric characters and “_” for variable names. Variable (and address label) names may be case-sensitive depending on the version.

To specify data types, a suffix character is added to the end of the variable name, as shown in Table E.1.

In Microsoft BASIC, the DIM statement can be used to specify a variable type:

```
DIM Variable AS INTEGER
```

without using the suffixes.

To declare arrays, the DIM statement is used like this:

```
DIM Variable([Low TO] High[, [Low TO] High...]) [AS Type]
```

TABLE E.1 BASIC DATA TYPE SPECIFIER SUFFIXES	
SUFFIX	VARIABLE TYPE
\$	String data
%	Integer
&	Long integer (32 bits)—This is a Microsoft BASIC extension.
!	Single precision floating-point number (32 bits)—This is a Microsoft BASIC extension.
#	Double precision floating-point number (64 bits)—This is a Microsoft BASIC extension.

There are a number of built-in statements that are used to provide specific functions to applications, as listed in Table E.2.

For assignment and `if` statements, the operators listed in Table E.3 are available in BASIC.

TABLE E.2 BASIC FUNCTIONS	
STATEMENT	FUNCTION
BASE	Starting array index (default is 0); used in DIM statement
DATA	Data block header
DIM	Variable declaration follows
OPTION	Operating specification
LET	Assignment statement indicator (not mandatory)
RANDOMIZE	Reset random number seed
RND	Return random value
INPUT [Prompt,]	Get console input
PRINT	Output data to console
?	Output data to console (Same as PRINT)
READ	Get DATA information
GOTO	Jump to specified line number or label
GOSUB	Call subroutine starting at line number or label
RETURN	Return execution to statement after GOSUB
IF condition THEN statement	Execute statement if condition is evaluated to be true

TABLE E.2 BASIC FUNCTIONS (CONTINUED)

STATEMENT	FUNCTION
FOR variable = init TO last [STEP inc]... NEXT [variable]	Repeat statements between FOR and NEXT statements while variable does not equal last
ON event GOTO	When the specified event occurs, jump to line number or label
RESTORE	Reset the point to the next DATA value
STOP	Stop program execution; execution may continue via ON event GOTO statement
END	Stop program execution and end application; cannot be restarted
REM	Comment
'	Comment—same as REM
ABS	Return the absolute value of an integer
SGN	Return the sign of an integer or floating-point number
COS	Calculate the cosine of an angle (radian)
SIN	Calculate the sine of an angle (radian)
TAN	Calculate the tangent of an angle (radian)
ATN	Calculate the radian angle of a tangent
INT	Convert a real number to an integer
SQR	Calculate the square root of a number
EXP	Calculate e to the power of the argument
LOG	Calculate the natural logarithm for the argument
TAB	Set tab columns for printer

BASIC's order of operations for complex statements is quite standard for programming languages and is listed in Table E.4.

Table E.5 lists the functions available in Microsoft versions of BASIC for the PC, as well as some BASIC for the PIC microcontroller.

PICBASIC

MicroEngineering Labs' (meLabs') PICBASIC is an excellent tool for learning about the PIC microcontroller before taking the big plunge into assembly-language programming. The source code required by the compiler is similar to Parallax BASIC Stamp BS2's PBASIC with many improvements and changes to make the language easier to work with and support different PIC MCUs.

TABLE E.3 BASIC OPERATORS	
OPERATOR	FUNCTION
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiation
"..."	ASCII string
,	Separator—place tabs in between objects in PRINT statement
;	Print concatenation—no spaces between objects in PRINT statement
\$	String variable identifier
=	Assignment indicator or equals to test operator
<	Less than
<=	Less than or equals to
>	Greater than
>=	Greater than or equals to
<>	Not equal to

In this section I will describe PICBASIC PRO, which is the enhanced version of the compiler that can support the widest range of PIC microcontroller part numbers. PICBASIC PRO is a very complete application development system that is able to convert your single file applications into PIC MCU machine instructions very efficiently.

PICBASIC does not currently have the ability to link together multiple source files, which means that multiple source files must be “included” in the overall source. Assembly-language statements are inserted in-line to the application. PICBASIC produces either assembler source files or completed .hex files. It does not create object files for linking modules together.

TABLE E.4 BASIC'S ORDER OF OPERATIONS FOR COMPLEX STATEMENTS		
OPERATORS	PRIORITY	FUNCTION TYPE
= <> < <= > >=	Highest	Conditional tests
^		Exponentiation
* /		Multiplication and division
+ -	Lowest	Addition and subtraction

TABLE E.5 MICROSOFT EXTENSIONS TO THE BASIC LANGUAGE

FUNCTION	OPERATION
AND	AND logical results
OR	OR logical results
XOR	XOR logical results
EQV	Test equivalence of logical results
IMP	Test implication of logical results
MOD	Get the modulus (remainder) of an integer division
FIX	Convert a floating-point number to an integer
DEFSTR Variable	Define the variable as a string (instead of using the DIM statement)
DEFINT Variable	Define the variable as an integer (instead of using the DIM statement)
DEFLNG Variable	Define the variable as a “long” integer (instead of using the DIM statement)
DEFSNG Variable	Define the variable as a single precision floating-point number (instead of using the DIM statement)
DEFDBL Variable	Define the variable as a double precision floating-point number (without using the DIM statement)
REDIM Variable ([low TO] High[, [low TO] High...]) [AS Type]	Redefine a variable
ERASE	Erase an array variable from memory
LBOUND	Return the first index of an array variable
UBOUND	Return the last index of an array variable
CONST Variable = Value	Define a constant value
DECLARE Function Subroutine	Declare a subroutine/function prototype at program start

(Continued)

TABLE E.5 MICROSOFT EXTENSIONS TO THE BASIC LANGUAGE (CONTINUED)

FUNCTION	OPERATION
DEF FNFunction(Arg[, Arg...])	Define a function (FNFunction) that returns a value; if a single line, then END DEF is not required
END DEF	End the function definition
FUNCTION function(Arg[, Arg...])	Define a function; same operation but different syntax as DEF FNFunction
END FUNCTION	End a function declaration
SUB subroutine(Arg[, Arg...])	Define a subroutine; if a single line, then END DEF is not required
END SUB	End the subroutine definition
DATA Value[, Value...]	Specify file data
READ Variable[, Variable...]	Read from DATA
IF condition THEN statements ELSE statements END IF	Perform a structured IF/ELSE/ENDIF
ELSEIF	Perform a condition test/structured IF/ELSE/ENDIF instead of simply ELSE
ON ERROR GOTO Label	On error condition, jump to handler
RESUME [Label]	Executed at the end of an error handler; can either return to current location, 0 (start of application), or a specific label
ERR	Return the current error number
ERL	Return the line the error occurred at
ERROR #	Execute an application-specific error (number #)
DO WHILE condition statements LOOP	Execute statements while condition is true
DO statements LOOP WHILE condition	Execute statements while condition is true
DO statements LOOP UNTIL condition	Execute statements until condition is true
EXIT	Exit executing FOR, WHILE, and UNTIL loops without executing check

SELECT variable	Execute based on value; CASE statements used to test the value and execute conditionally
CASE value	Execute within a SELECT statement if the variable equals value; CASE ELSE is the default case
END SELECT	End the SELECT statement
LINE INPUT	Get formatted input from the user
INPUT\$(#)	Get the specified number (#) of characters from user
INKEY\$	Check keyboard buffer and return number of pending characters or zero
ASC	Convert the character into an integer ASCII code
CHR\$	Convert the integer ASCII code into a character
VAR	Convert the string into an integer number
STR\$	Convert the integer number into a string
LEFT\$(string, #)	Return the specified number (#) of leftmost characters in string
RIGHT\$(string, #)	Return the specified number (#) of right most characters in string
MID\$(string, start, #)	Return/overwrite the specified number (#) of characters at position start in string
SPACE\$(#)	Returns a string of specified number (#) of ASCII blanks
LTRIM\$	Remove the leading blanks from a string
RTRIM\$	Remove the trailing blanks from a string
INSTR(string, substring)	Return the position of substring in string
UCASE\$	Convert all the lower-case characters in a string to upper-case
LCASE\$	Convert all the upper-case characters in a string to lower-case
LEN	Return the length of a string
CLS	Clear the screen
CSRLIN	Return the current line that the cursor is on

(Continued)

TABLE E.5 MICROSOFT EXTENSIONS TO THE BASIC LANGUAGE (CONTINUED)

FUNCTION	OPERATION
POS	Return the current column that the cursor is on
LOCATE X, Y	Specify the row/column of the cursor (top left is 1,1)
SPC	Move the display the specified number of spaces
PRINT USING "Format"	Print the value in the specified format; +, #, ., and ^ characters are used for number formats
SCREEN mode[, [color][, [page][, visual]	Set the screen mode; color is 0 to display on a color display, 1 to display on a monochrome; page is the page that receives I/O, and visual is the page that is currently active
COLOR [foreground] [, [background][, border]]	Specify the currently active color PALETTE [attribute, color] to change color assignments
VIEW [[SCREEN] (x1,y1)-(x2,y2)[, [color]][, border]]	Create a small graphics window known as a viewport
WINDOW [[SCREEN] (x1,y1)-(x2,y2)]	Specify the viewport's logical location on the display
PSET (x,y)[, color]	Put a point on the display
PRESET (x,y)	Return the point to the background color
LINE (x1,y1)-(x2,y2)[, [color][, [B BF][, style]]]	Draw a line between the two specified points; if B or BF is specified, draw a box (BF is filled)
CIRCLE (x,y), radius[, [color][, [start][, end][, aspect]]]	Draw the circle at center location and with the specified radius; start and end are starting and ending angles (in radians); aspect is the circle's aspect for drawing ellipses
DRAW commandstring	Draw an arbitrary graphics figure; there should be spaces between the commands. Commands: U#—moves cursor up # pixels D#—moves cursor down # pixels

	E#—moves cursor up and to the right # pixels
	F#—moves cursor down and to the right # pixels
	G#—moves cursor down and to the left # pixels
	H#—moves cursor up and to the left # pixels
	I#—moves cursor left # pixels
	R#—moves cursor right # pixels
	Mxy—moves the cursor to the specified x,y position
	B—turns off pixel frawing
	N—turns on cursor and moves to original position
	A#—rotates shape in 90-degree increments
	C#—sets the drawing color
	P#Color#Border—sets the shape fill and border colors
	S#—sets the drawing scale
	T#—Rotates # degrees
LPRINT	Send output to the printer
BEEP	“Beep” the speaker
SOUND frequency, duration	Make the specified sound on the PC's speaker
PLAY notestring	Output the specified string of notes to the PC's speaker
DATE\$	Return the current date
TIME\$	Return the current time
TIMER	Return the number of seconds since midnight
NAME filename AS newfilename	Change the name of a file
KILL FileName	Delete the file

(Continued)

TABLE E.5 MICROSOFT EXTENSIONS TO THE BASIC LANGUAGE (CONTINUED)

FUNCTION	OPERATION
FILES [FileName.Ext]	List the file (MS-DOS dir); FileName.Ext can contain wild cards
OPEN filename [FOR access] AS #handle	Open the file as the specified handle (starting with the # character) Access: I—open for text input O—open for text output A – open to append text B—file is opened to access single bytes R—open to read and write structured variables
CLOSE #handle	Close the specified file
RESET	Close all open files
EOF	Returns “True” if at the end of file
READ #handle, variable	Read data from the file
GET #handle, variable	Read a variable from the file
INPUT #handle, variable	Read formatted data from the file using INPUT, INPUT USING, and INPUT\$ formats
WRITE #handle, variable	Write data to the file
PUT #handle, variable	Write a variable to a file
PRINT #handle, output	Write data to the file using the PRINT and PRINT USING formats
SEEK #handle, offset	Move the file pointer to the specified offset within the file

For additional information and the latest device libraries, look at the microEngineering Labs' web page at <http://www.melabs.com/mel/home.htm>.

PICBASIC PRO is an MS-DOS command-line application that is invoked using the statement

```
PBP [options...] source
```

It also has been compiled to run from the Microsoft Windows protect mode using a full 4.3-GB flat memory model to avoid out-of-memory errors that are possible with the MS-DOS version. To invoke the protect mode version, open an MS-DOS prompt window in Microsoft Windows 95/98/NT/2000 and enter in the command

```
PBPW [options...] source
```

The source is the MS-DOS (maximum eight character) application code source file name with the .bas extension. Options are compiler execution options and are listed in Table E.6.

TABLE E.6 PICBASIC COMMAND LINE OPTIONS

OPTION	FUNCTION
-h/-?	Display the help screen; the help screen is also displayed if no options or source file name is specified.
-ampasm	Use the MPASM assembler and not the PICBASIC assembler.
-c	Insert comments into PICBASIC compiler-produced assembler source file; using this option is recommended if you are going to produce MPASM assembler source from PICBASIC.
-iPath	Specify a new directory "path" to use for include files in PICBASIC.
-lLibrary	Specify a different library to use when compiling; device-specific libraries are provided by PICBASIC when the processor is specified.
-od	Generate a listing, symbol table, and map files.
-ol	Generate a listing file.
-pPICmicro	Specify the PICmicro that the source is to be compiled into; if this parameter is not specified, then a PIC16F84 is used as the processor; PICmicro is in the format: 16F84, where the PIC at the start of the Microchip part number is not specified.
-s	Do not assemble the compiled code.
-v	Turn on verbose mode, which provides additional information when the application is compiled.

TABLE E.7 PICBASIC DEFAULT CONFIGURATION FUSE REGISTER SETTINGS	
FEATURE	PICBASIC SETTING
Code protect	Off
Oscillator	XT—or internal RC if PIC12
WDT	On
PWRT	Off

I have not included a list of the different PIC microcontrollers that can have their application developed by PICBASIC simply because this list changes along with the different parts that are available from Microchip. MeLabs works very hard at making sure that all new PIC MCU part numbers are supported as quickly as possible after they are announced. Information on what part numbers the latest version of PICBASIC supports can be found at meLabs' web site.

PICBASIC does assume a constant set of configuration values. For most PIC microcontrollers, the configuration fuses are set as listed in Table E.7.

When you program the PIC microcontroller, you should make sure that you are comfortable with the oscillator and PWRT value. The watchdog timer should be left enabled because the `NAP` and `SLEEP` instructions use it.

Along with using the MS-DOS command line, PICBASIC can be used with MPLAB. This is the preferred way of working with PICBASIC because it works seamlessly with the Microchip tools. In the text of this book I have included some examples of how to use PICBASIC PRO for the application's source code to demonstrate how PICBASIC works with MPLAB.

To use PICBASIC with MPLAB, after installing PICBASIC, select "Install Language Tool" under MPLAB's "Project" pull-down, and select "microEngineering Labs, Inc." for the "Language Suite," followed by the PICBASIC that you are using. After "PICBASIC" or "PICBASIC PRO" has been selected, click on "Browse" to find PBC.exe or PBPW.exe. Finally, make sure that the "Command Line" radio button is selected, and click on "OK." After doing this, you will be able to create PICBASIC projects as if they were assembler projects.

Note that with PICBASIC, the project and source files must reside in the same subdirectory as the PICBASIC compiler and its associated files. This breaks up the recommended strategy of keeping all the project files in a directory that is different from the tool directory. When you develop PICBASIC applications, I recommend that you copy them into a separate subdirectory when you are finished and delete them from the PICBASIC execution directory. This avoids ending up with large, confusing PICBASIC subdirectories and no idea what is in them.

The starting point of the language is the *label*. Labels are used for specifying execution addresses, variables, and hardware registers, as well as constant strings. As with

most other languages, PICBASIC labels are case-sensitive and can include the characters A to Z and 0 to 9 as well as `_`. A numeric cannot be used as the first character of the label.

Labels in the application code are used to provide jump-to and call addresses. They are indicated by writing them into the application code starting at the leftmost column of the source file and terminating with a colon (`:`). To access the label, the label name is used in a `goto`, `call`, or `branch` statement. For example, to implement an end-less loop, you would use the label declaration and the `goto` statement:

```
Loop:           ' Return here after running through the Code

:              ' Code to Execute inside the loop

goto Loop      ' Jump back to "Loop" and Start Executing
               ' again
```

Absolute addresses are not available within PICBASIC; the labels always should be defined and referenced. The interrupt vector in the mid-range and PIC18 PIC micro-controllers is discussed below.

In the preceding example I have placed a number of comments after the single quote (`'`) character. The compiler will ignore everything to the right of the single quote character, just as it does for the comment character (`;`) in assembly-language programming.

Variables in PICBASIC can be defined in one of two ways. Each way reflects a method that was used by the two different BASIC Stamps. In the BS1, variables are predefined in the system and broken up into either 16-bit words (which are given the identifiers `w0`, `w1`, `w2`, etc.), 8-bit bytes (with the identifiers `b0`, `b1`, etc.), and bits (with the identifiers `Bit#` or just the bit number). The maximum number of variables in the BS1 system is defined by the processor used (in the BS1, which is based on the PIC16C54, only 14 bytes are available as variable memory).

Each byte takes place in one of the words; for example, `b4` is the least significant byte of `w2`. The 16-bit variables are defined as being a part of the 16 bits taken up by `w0` (`b0` and `b1`). This method works well, but care has to be taken to make sure that the overlapping variables are kept track of and not used incorrectly. The most common problem for new BASIC Stamp developers is defining a variable on `b0` and `w0` and having problems when a write to one variable overwrites the other.

To provide these variables to the PICBASIC application, the BASIC Stamp variable declaration files are defined in the two `include` files that are shown within `include` statements below. Only one of these statements can be used in an application.

```
include "bs1defs.bas"
include "bs2defs.bas"
```

A much better way to declare variables is to use the `var` directive to select the different variables at the start of the application and let the PICBASIC compiler determine where the variables belong and how they are accessed (i.e., put in different variable

pages). Along with the `var` directive, the `word`, `byte`, and `bit` directives are used to specify the size of the variable. Some example variable declarations are

```
WordVariable var word      ' Declare a 16 Bit Variable
ByteVariable var byte      ' Declare an 8 Bit Variable
BitVariable var bit        ' Declare a single byte Variable
```

Initial values for the variables cannot be made at the variable declarations.

Along with defining variables, the `var` directive can be used to define variable labels built out of previously defined variables to specify specific data. Using the preceding variables, I can break `WordVariable` up into a top half and a bottom half and `ByteVariable` into specific bytes with the statements

```
WordVariableTop var WordVariable.byte1
WordVariableBottom var WordVariable.byte0

BitVariableMSB var BitVariable.bit7
BitVariableLSB var BitVariable.0
```

Variables also can be defined over registers. When the PICBASIC libraries are merged with the source code, the standard PIC microcontroller register names are available within the application. Using this capability, labels within the application can be made to help make the application easier to work with. For example, to define the bits needed for an LCD, the declarations below could be used:

```
LCDDData var PORTB      ' PORTB as the 8 Bits of Data
LCDE var PORTA.0        ' RA0 is "E" Strobe
LCDRS var PORTA.1       ' RA1 is Data/Instruction Select
LCDRW var PORTA.2       ' RA2 is the Read/Write Select Bit
```

When variables are defined using the `var` and `system` directives, specific addresses can be made in the application. For example, the statement

```
int_w var byte $0C system
```

will define the variable `_w` at address `0x00C` in the system. This reserves address `0x00C` and does not allow its use by any other variables. The bank of a variable can be specified using the `system` directive, as in

```
int_status var byte bank0 system
```

These two options to the `var` directive are useful when defining variables for interrupt handlers, as discussed below.

Along with redefining variables with the `var` statement, PICBASIC also has the `symbol` directive. The `symbol` directive provides the same capabilities as the `var` statement, and it is provided simply for compatibility with the BS1. If you were only

**TABLE E.8 MAXIMUM NUMBER OF ELEMENTS
FOR DIFFERENT ARRAY DATA TYPES**

VARIABLE TYPE	MAXIMUM NUMBER OF ELEMENTS
Word	32
Byte	64
Bit	128

developing PICBASIC applications, I would recommend only using `var` and avoiding the `symbol` directive.

Single-dimensional arrays can be defined within PICBASIC for each of the three data types when the variable is declared as shown below:

```
WordArray var word[10]           ' Ten Word Array
ByteArray var byte[11]           ' Eleven Byte Array
BitArray var bit[12]             ' Twelve Bit Array
```

Note that bits can be handled as an array element, which is a really nice feature to the language. Depending on the PIC microcontroller part number, the maximum array sizes are listed in Table E.8.

As part of the `bit` definition, inout/output (I/O) port pins are predefined within PICBASIC. Up to 16 pins (addressed using the `Pin#` format, where `#` is the pin number) can be accessed, although how they are accessed changes according to the PIC MCU part number for which the application is designed. The pins for different-sized parts are defined in Table E.9.

Note that not all the ports that have all 8 pins specified. For example, accessing `pin6` in an 18-pin device (which does not have an RA6 bit) will not do anything.

TABLE E.9 PICBASIC PIN TO PIC MICROCONTROLLER SPECIFICATIONS

NUMBER OF PIC MCU I/O PINS	PICBASIC PINS 7–0	PICBASIC PINS 15–8
8	GPIO5-0	Mapped onto pins 7–0
14	PORTB/PORTC (depending on part)	PORTA
18	PORTB	PORTA
28	PORTB	PORTC
40	PORTB	PORTC

TABLE E.10 PICBASIC DATA TYPES		
DATA TYPE	MODIFIER	COMMENTS
Decimal	None	PICBASIC default is decimal.
Hex	\$	\$ is placed before the number.
Binary	%	% is placed before the number.
ASCII	"..."	Double quotes are placed around a single character.

Constants are declared in a similar manner to constants but using the `con` directive with a constant parameter:

```
SampleConstant con 3 + 7          ' Define a Sample Constant
```

Constant values can be in four different formats. Table E.10 lists the different formats and the modifiers to indicate to the PICBASIC compiler which data type is being specified.

In Table E.10, note that only an ASCII byte can be passed within double quotes. Some instructions (described below) can be defined with strings of characters that are enclosed within double quotes.

The `define` statement is used to change constants given defaults within the PIC microcontroller when a PICBASIC-compiled application is running. The format is

```
DEFINE Label NewValue
```

The labels, their default values, and their values are listed in Table E.11. The `OSC` define should be specified if serial I/O is going to be implemented in the PIC microcontroller. This value is used by the compiler to calculate the time delays necessary for each bit.

Assembly language can be inserted at any point within a PICBASIC application. Single instructions can be inserted simply by starting the line with an `@` character:

```
@ bcf INTCON, TOIF          ; Reset TOIF Flag
```

Multiple lines of assembly language are prefaced by the `asm` statement and finished with the `endasm`. An example of this is shown below:

```
asm
  movlw 8                    ; Loop 8x
Loop:
  bsf PORTA, 0               ; Pulse the Bit
  bcf PORTA, 0
  addlw $0FF                 ; Subtract 1 from "w"
```


TABLE E.11 PICBASIC INTERNAL OPERATING PARAMETERS

DEFINE	DEFAULT	OPTIONAL PARAMETERS	COMMENTS
BUTTON_PAUSE	10	Any positive integer	Button debounce delay in ms
CHAR_PACING	1000	Any positive integer	Time in μ s between SerOut characters
DEBUG_BAUD	2400	Any	Specified data rate of debug information
DEBUG_BIT	0	7–0	Output pin for debug serial output
DEBUG_MODE	1	0, 1	Polarity of debug NRZ output data; 0, positive; 1, inverted
DEBUG_PACING	1000	Any positive integer	Time in μ s between output characters for DEBUG statements
DEBUG_REG	PORTB	Any I/O port	Port debug output pin
DEBUGIN_BIT	0	7–0	Input pin for debug serial output
DEBUGIN_MODE	1	0, 1	Polarity of debug NRZ input data; 0, positive; 1, inverted
DEBUGIN_REG	PORTB	Any I/O port	Port debug input pin
HSER_BAUD	2400	Any	Hardware serial port's data rate
HSER_SPBRG	25	0-0xFF	Hardware serial port's SPBRG register value
HSER_RCSTA	0x90	0-0xFF	Hardware serial port's RTCA register initialization value; default set for asynchronous communications
HSER_TXSTA	0x20	0-0xFF	Hardware serial port's TXSTA register initialization value; default set for asynchronous communications
HSER_EVEN	1	0, 1	Hardware serial port's parity select value; only used if parity checking is desired
HSER_ODD	1	0, 1	Hardware serial port's parity select value; only used if parity checking is desired
I2C_HOLD	1	0, 1	Stop I2C transmission while the SCL line is held low
I2C_INTERNAL	1	0, 1	Set to use internal EEPROM in PIC12Cxx microcontrollers

(Continued)

TABLE E.11 PICBASIC INTERNAL OPERATING PARAMETERS (CONTINUED)

DEFINE	DEFAULT	OPTIONAL PARAMETERS	COMMENTS
I2C_SCLOUT	1	0, 1	Use a bipolar driver instead of an open-drain I2C clock driver
I2C_SLOW	1	0, 1	Run the I2C at no more than 100 kbps data rate
LCD_BITS	4	4, 8	Number of data bits required for LCD interface
LCD_DBIT	0	0, 4	Specify the nibble used for the first LCD data bit
LCD_DREG	PORTA	Any I/O port	Select the LCD data port
LCD_EBIT	3	7–0	Specify the bit for the LCD E clock bit
LCD_EREG	PORTB	Any I/O port	Specify the port for the LCD E clock bit
LCD_LINES	2	1, 2	Specify the number of lines on the LCD; check information in book on LCDs for how the multiple line specification is used in some single-line LCDs
LCD_RSBIT	4	7–0	LCD RS bit selection
LCD_RSREG	PORTA	Any I/O port	LCD RS bit select register
OSC	4	3, 4, 8, 10, 12, 16, 20	Specify PIC microcontroller operating speed in MHz; note 3 is actually 3.58 MHz
OSCCAL_1K	1	0, 1	Set OSCCAL for PIC12C672
OSCCAL_2K	1	0, 1	Set OSCCAL for PIC12C672
SER2_BITS	8	4–8	Specify number of bits sent with SERIN2 and SEROUT2 instructions

```

    btfss STATUS, Z           ; Do 8x
    goto Loop
endasm

```

Note that labels inside the assembler statements do not have a colon at the end of the string and that traditional assembly-language comment indicator (the semicolon) is used.

Implementing interrupt handlers in PICBASIC can be done in one of two ways. The simplest way of implementing it is to use the `ON INTERRUPT GOTO Label` statement. Using this statement, anytime an interrupt request is received, the `Label` specified in the `ON INTERRUPT` statement will be executed until there is a `resume` instruction that returns from an interrupt. Using this type of interrupt handler, straight PICBASIC statements can be used and assembly-language statements avoided.

The basic operation looks like this:

```

:

ON INTERRUPT GOTO IntHandler

:

IntHandler:

    disable           ' Turn off interrupt and debug requests

    :               ' Process Interrupt

    enable           ' Enable other Interrupts and debug
' requests

    resume           ' Return to the executing code

```

The problem with this method is that the interrupt handler is executed once the current PICBASIC instruction has completed. If a very long PICBASIC instruction is being executed (say, a string serial send), then the interrupt will not be serviced in a timely manner.

The best way to handle an interrupt is to add the interrupt handler as an assembly-language routine. To reference the interrupt handler, the `define INTHAND Label` instruction is used to identify the label where the assembly-language code is listed. The interrupt handler will be moved to start at address 0x004 in mid-range devices.

A code template for generic mid-range PIC microcontroller interrupt handlers is

```

int_w var byte 0x020 system      ' Define the Context Save Variables
int_status var byte bank0 system
int_fsr var byte bank0 system
int_pclath byte bank0 system

:

define INTHAND IntHandler        ' Specify what the Interrupt
' Handler is

```

```
:  
  
' Interrupt Handler – to be relocated to 0x00004  
asm  
IntHandler  
    movwf int_w                ; Save the Context Registers  
    movf STATUS, w  
    bcf STATUS, RP0           ; Move to bank 0  
    bcf STATUS, RP1  
    movwf int_status  
    movf FSR, w  
    movwf int_fsr  
    movf PCLATH, w  
    movwf int_pclath  
  
; ##### - Execute Interrupt Handler Code Here  
  
    movf int_pclath, w        ; Finished, restore the Context  
    movwf PCLATH              ; Registers  
    movf int_fsr, w  
    movwf FSR  
    movf int_status, w  
    movwf STATUS  
    swapf int_w, f  
    swapf int_w, w  
    retfie  
endasm
```

The interrupt template code provides context saving and restoring for the worst-case condition. I am assuming that execution is taking place in something other than bank 0, that the FSR has been given a specific value, and that the PIC microcontroller has more than one page of program memory. If you use this template in your own applications, I do not recommend that the preceding above be changed. The only way you can be sure that the index register is not being used and that execution will be in bank 0 along with the processor taking program memory addresses out of page 0 is to look at the code produced by the compiler. Rather than going to this trouble, you should just use the preceding template and insert your interrupt handler code at the comment with ##### inside it.

There is one issue that you should be aware of when you are adding an assembly-language interrupt handler, and this is that you should make sure that no critically timed PICBASIC operations are executing when the interrupt is acknowledged. If, for example, a serial transfer were taking place when the interrupt request was received and acknowledged, then you would end up changing the timing and the data that is potentially received or sent. To ensure that this doesn't happen, you will have to disable interrupts while the critically timed operation is executing.

Mathematical operators used in assignment statements and PICBASIC instructions are very straightforward in PICBASIC and work conventionally. In BASIC Stamp

PBASIC, you must remember that the operations execute from right to left. This means that the statement

```
A = B * C + D
```

which you would expect to operate as

```
Multiply "B" and "C"
Add the results from 1. to "D"
```

in PBASIC returns the result

```
Get the Sum of "C" and "D"
Multiply the results from 1. With "B"
```

PICBASIC does not follow the PBASIC evaluation convention and returns the expected result from complex statements such as the preceding one. This means that in PICBASIC you do not have to break complex statements up into single operations, as you do in PBASIC, to avoid unexpected expression evaluation. If you are using a BASIC Stamp to prototype PICBASIC applications, then I recommend that you break up the complex statements and use the temporary values.

The mathematical operators used are listed in Table E.12 along with their execution priority and parameters. All mathematical operators work with 16-bit values.

Along with the mathematical operators, the `if` statement uses the test conditions listed in Table E.13. Note that both the BASIC standard labels and the C standard labels are used. `Parm1` and `Parm2` are constants, variables, or statements made up of variables or statements along with the different mathematical operators and test conditions. When a test condition is true, a nonzero is returned; if it is false, then a zero is returned. Using this convention, single-variable parameters can be tested in `if` statements rather than performing comparisons of them to zero.

The PICBASIC instructions are based on the Parallax BASIC Stamp (PBASIC) language, and while there are a lot of similarities, they are really two different languages. In Table E.14 I have listed all the PICBASIC instructions and indicated any special considerations that should be made for them with respect to being compiled in a PIC microcontroller.

These instructions are really *library routines* that are called by the mainline of the application. I am mentioning this because you will notice that the size of the application changes based on the number of instructions that are used in the application. You may find that you can reduce the program memory size drastically by looking at the different instructions that are used and change the statements to assembler or explicit PICBASIC statements.

When I specified the various instructions in Table E.14, note that I used square brackets ([and]) to specify data tables in some instructions. For this reason, I have specified optional values using braces ({ and }), which breaks with the conventions used in the rest of this book.

TABLE E.12 PICBASIC MATHEMATICAL AND BIT OPERATIONS		
PRIORITY	BASIC OPERATOR EXPRESSION	OPERATION
Lowest	Parm1 + Parm2	Return the sum of Parm1 and Parm2.
	Parm1 - Parm2	Return the result of Parm2 subtracted from Parm1.
	Parm1 * Parm2	Return the least significant 16 bits of the product of Parm1 and Parm2; this is often referred to as bytes 0 and 1 of the result.
	Parm1 */ Parm2	Return the middle 16 bits of the product of Parm1 and Parm2; this is often referred to as bytes 1 and 2 of the result.
	Parm1 ** Parm2	Return the most significant 16 bits of the product of Parm1 and Parm2; this is often referred to as bytes 2 and 3 of the result.
	Parm1 / Parm2	Return the number of times Parm2 can be divided into Parm1 evenly
	Parm1 // Parm2	Return the remainder from dividing Parm2 into Parm1; this is known as the <i>modulus</i> .
	Parm1 & Parm2	Return the bitwise value of Parm1 AND Parm2.
	Parm1 Parm2	Return the bitwise value of Parm1 OR Parm2.
	Parm1 ^ Parm2	Return the bitwise value of Parm1 XOR Parm2.
	~ Parm1	Return the inverted bitwise value of Parm1.
	Parm1 &/ Parm2	Return the inverted bitwise value of Parm1 AND Parm2.
	Parm1 / Parm2	Return the inverted bitwise value of Parm1 OR Parm2.
	Parm1 ^/ Parm2	Return the inverted bitwise value of Parm1 XOR Parm2.
	Parm1 << Parm2	Shift Parm1 to the left Parm2 bits; the new least significant bits will all be zeros.
	Parm1 >> Parm2	Shift Parm1 to the right Parm2 bits; the new most significant bits will all be zeros.
	ABS Parm1	Return the magnitude of a number; ABS -4 is equal to ABS 4 and returns 4.
	Parm1 MAX Parm2	Return the higher parameter.
	Parm1 MIN Parm2	Return the lower parameter.
	Parm1 DIG Parm2	Return digit number Parm1 (zero based) of Parm1; 123 DIG 1 returns 2.
	DCD Parm1	Return a value with only the Parm1 bit set; DCD 4 returns %00010000.

TABLE E.12 PICBASIC MATHEMATICAL AND BIT OPERATIONS (CONTINUED)

PRIORITY	BASIC OPERATOR EXPRESSION	OPERATION
	NCD Parm1	Return the bit number of the highest set bit in Parm1.
	Parm1 REV Parm2	Reverse the bits in Parm1 from zero to Parm2; %10101100 REV 4 will return %10100011.
	SQR Parm1	Return the integer square root of Parm1.
	SIN Parm1	Return the trigonometric “sine” of Parm1; the returned value will be based on a circle of radius 127 and 256 degrees (not the traditional circle radius of 1 and having 360 degrees).
Highest	COS Parm1	Return the trigonometric “cosine” of Parm1; the returned value will be based on a circle of radius 127 and 256 degrees (not the traditional circle radius of 1 and having 360 degrees).

TABLE E.13 PICBASIC LOGICAL OPERATIONS

TEST CONDITION	OPERATION DESCRIPTION
Parm1 = Parm2	Return a nonzero if Parm1 equals Parm2.
Parm1 == Parm2	Return a nonzero if Parm1 equals Parm2.
Parm1 <> Parm2	Return a nonzero if Parm1 does not equal Parm2.
Parm1 != Parm2	Return a nonzero if Parm1 does not equal Parm2.
Parm1 < Parm2	Return a nonzero if Parm1 is less than Parm2.
Parm1 <= Parm2	Return a nonzero if Parm1 is less than or equal to Parm2.
Parm1 > Parm2	Return a nonzero if Parm1 is greater than Parm2.
Parm1 >= Parm2	Return a nonzero if Parm1 is greater than or equal to Parm2.
Parm1 AND Parm2	Return a nonzero if Parm1 is nonzero and Parm2 is nonzero.
Parm1 && Parm2	Return a nonzero if Parm1 is nonzero and Parm2 is nonzero.
Parm1 OR Parm2	Return a nonzero if Parm1 is nonzero or Parm2 is nonzero.
Parm1 Parm2	Return a nonzero if Parm1 is nonzero or Parm2 is nonzero.
Parm1 XOR Parm2	Return a nonzero if Parm1 and Parm2 are different logical values.
Parm1 ^^ Parm2	Return a nonzero if Parm1 and Parm2 are different logical values.
Parm1 NOT AND Parm2	Return zero if Parm1 is nonzero and Parm2 is nonzero.
Parm1 NOT OR Parm2	Return zero if Parm1 is nonzero or Parm2 is nonzero.
Parm1 NOT XOR Parm2	Return a nonzero if Parm1 and Parm2 are in the same logical state.

TABLE E.14 PICBASIC BUILT IN FUNCTIONS

INSTRUCTION	DESCRIPTION
BRANCH Index, [Label(,Label...)]	Jump to the Label specified by the value in Index; Index is zero-based, so an Index of 0 will cause execution to jump to the first Label; an Index of 1 will cause execution to jump to the second Label, and so on; this instruction only jumps within the current page; if a PIC MCU with more than one page of program memory is used, then the BRANCHL instruction is recommended.
BRANCHL Index, [Label {,Label...}]	Jump to the Label specified by the value in Index; Index is zero-based, so an Index of 0 will cause execution to jump to the first Label; an Index of 1 will cause execution to jump to the second Label, and so on; this instruction can jump anywhere in PIC MCU program memory.
BUTTON Pin, Down, Delay, Rate, Bvar, Action, Label	Jump to Label when the button has been pressed for the specified number of BUTTON instruction invocations; Down is the pin state for the switch to be assumed to be closed; Delay is the number of invocations the Down state has to be true before the Action is taken; Rate is how many invocations after the first BUTTON jump is true that an autorepeat happens; Bvar is a byte-sized variable only used in this function; Action is whether or not the jump is to take place when the key is pressed (1) or released (0).
CALL Label	Execute the assembly-language call instruction.
CLEAR	Load all the variables with zero.
COUNT Pin, Period, Variable	Count the number of pulses on Pin that occur in Period ms.
DATA @Location, Constant {,Constant...}	Store constants in data EEPROM starting at Location when the PIC MCU is programmed; for data at different addresses, use multiple DATA statements.
DEBUG Value{,Value...}	Define the DEBUG pin as output with the serial output parameters used in the DEBUG defined at reset; when this instruction is executed, pass the parameter data; if an ASCII # (0x023) is sent before a Value, the decimal numeric is sent, rather than the ASCII byte; this instruction (and DEBUGIN) can be used for serial I/O because they take up less space than the SERIN and SEROUT instructions.
DEBUGIN {TimeOut,Label,} [Variable{,Variable...}]	Define the DEBUGIN pin as an input with the serial input parameters used in the DEBUGIN defines at reset; when this instruction is executed, wait for a data byte to come in or jump to the label if the TimeOut value (which is specified in ms) is reached.

DISABLE	Disables interrupts and debug operations; interrupts still will be acknowledged, but ON INTERRUPT GOTO will not execute.
DISABLE INTERRUPT	Disables interrupts and debug operations; interrupts still will be acknowledged, but ON INTERRUPT GOTO will not execute; same as DISABLE.
DTMFOUT Pin, {On, Off,} [Tone{, Tone...}]	Output the touch tone sequence on the specified pin; tones 0 through 9 are the same as on the telephone keypad; tone 10 is the * key, and tone 11 is the # key; tones 12 through 15 correspond to the extended key standards for A to D; filtering is required on the pin output to “smooth” out the signal output.
EEPROM Location, [Constant{[, Constant...}]	Store new values in EEPROM when the PIC MCU is programmed; this instruction is the same as DATA
ENABLE	Enable debug and interrupt processing that was stopped by DISABLE.
ENABLE DEBUG	Enable debug operations that were stopped by DISABLE.
ENABLE INTERRUPT	Enable interrupt operations that were stopped by the DISABLE and DISABLE INTERRUPT instructions.
END	Stop processing the application and put the PIC MCU in a low power sleep mode.
FOR Variable = Start TO Stop {STEP Value} : NEXT {Variable}	Execute a loop, first initializing Variable to the Start value until it reaches the Stop value; the increment value defaults to 1 if no STEP value is specified; when NEXT is encountered, Variable is incremented and tested against the Stop value.
FREQOUT Pin, On, Frequency{, Frequency}	Output the specified Frequency on the Pin for On ms; if a second Frequency is specified, output this at the same time; filtering is required on the pin output to smooth out the signal output.
GOSUB Label	Call the subroutine that starts at address Label; the existence of Label is checked at compile time.
GOTO Label	Jump to the code that starts at address Label.
HIGH Pin	Make Pin an output, and drive it at a high logic level.
HSERIN {ParityLabel,} {TimeOut, Label,} [Variable {[, Variable...}]	Receive one or more bytes from the built-in USART (if present); the ParityLabel will be jumped to if the parity of the incoming data is incorrect; to use ParityLabel, make sure that the HSER_EVEN and HSER_ODD defines have been specified.
HSEROUT [Value {, Value...}]	Transmit one or more bytes from the built-in USART (if present).

(Continued)

TABLE E.14 PICBASIC BUILT IN FUNCTIONS (CONTINUED)

INSTRUCTION	DESCRIPTION
I2CREAD DataPin, ClockPin, ControlByte, {Address,} [Variable {,Variable...}] {,NoAckLabel}	Read a byte string from an I2C device; the ControlByte is used to access the device with block or device select bits; this instruction can be used to access an internal EEPROM in the PIC12CExx devices by entering the define I2C_INTERNAL 1 statement at the start of the application code.
I2CWRITE DataPin, ClockPin, Control, {Address,} [Value{,Value...}] {,NoAckLabel}	Send a byte string to an I2C device; the ControlByte is used to access the device with block or device select bits; this instruction can be used to access internal an EEPROM in the PIC12CExx devices by entering the define I2C_Internal 1 statement at the start of the application code.
IF Comp THEN Label	Evaluate the Comp comparison expression and label; if it is not equal to 0, then jump to Label.
IF Comp THEN : {ELSE :} ENDIF	Evaluate the Comp comparison expression and statement; if it is not equal to 0, then execute the Statements below until either an ELSE or an ENDIF statement is encountered; if an ELSE statement is encountered, then the code after it, to the ENDIF instruction, is ignored; if Comp evaluates to 0, then skip over the Statements after the IF statement are ignored to the ELSE or ENDIF statements, after which any Statements are executed.
INCLUDE "file"	Load in file.bas in the current directory, and insert it at the current location in the source file.
INPUT Pin	Put the specified pin into input mode.
{LET} Assignment	Optional instruction specifier to indicate an assignment statement.
LCDOUT Value{,Value...}	Send the specified bytes to the LCD connected to the PIC MCU; the LCD's operating parameters are set with the LCD defines; to send an instruction byte to the LCD, a \$0FE byte is sent first.
LOOKDOWN offset, [Constant{,Constant...}], Variable	Go through a list of constants with an offset, and store the constant value at the offset in the second Variable; if the offset is greater than the number of constants, then 0 is returned in Variable; offset is zero-based, so the first constant is returned if offset is equal to 0.
LOOKDOWN2 offset, {Test} [Constant{,Constant...}], Variable	Search the list and find the constant value that meets the condition Test. If Test is omitted, then the LOOKDOWN2 instruction behaves like the LOOKDOWN instruction, and the Test is assumed to be an equals sign (=).
LOOKUP Variable, [Constant {,Constant...}], Variable	Compare the first Variable value with a constant string, and return the offset into the constant string in the second Variable; if there is no match, then the second Variable is not changed.

LOOKUP2 Variable, [Value{,Value...}], Variable	Compare the first Variable value with a Value string, and return the offset into the Value string in the second Variable; if there is no match, then the second Variable is not changed; LOOKUP2 differs from LOOKUP because the Values can be 16-bit variable values.																		
LOW Pin	Make Pin an output and drive it at a low logic level.																		
NAP Period	Put the PIC MCU to sleep for the period value that is given below:																		
	<table> <tr> <th>Period</th><th>Delay</th></tr> <tr> <td>0</td><td>18 ms</td></tr> <tr> <td>1</td><td>36 ms</td></tr> <tr> <td>2</td><td>73 ms</td></tr> <tr> <td>3</td><td>144 ms</td></tr> <tr> <td>4</td><td>288 ms</td></tr> <tr> <td>5</td><td>576 ms</td></tr> <tr> <td>6</td><td>1,152 ms</td></tr> <tr> <td>7</td><td>2,304 ms</td></tr> </table>	Period	Delay	0	18 ms	1	36 ms	2	73 ms	3	144 ms	4	288 ms	5	576 ms	6	1,152 ms	7	2,304 ms
Period	Delay																		
0	18 ms																		
1	36 ms																		
2	73 ms																		
3	144 ms																		
4	288 ms																		
5	576 ms																		
6	1,152 ms																		
7	2,304 ms																		
ON DEBUG GOTO Label	When invoked, every time an instruction is about to be invoked, the debug monitor program at Label is executed; two variables, the DEBUG_ADDRESS word and the DEBUG_STACK byte, must be defined as bank 0 system bytes; to return from the debug monitor, a RESUME instruction is used.																		
ON INTERRUPT GOTO Label	Jump to the interrupt handler starting at Label; when the interrupt handler is complete, execute a RESUME instruction.																		
OUTPUT Pin	Put Pin into output mode.																		
PAUSE Period	Stop the PIC MCU from executing the next instruction for Period ms; PAUSE does not put the PIC MCU to sleep like NAP does.																		
PAUSEUS Period	Stop the PIC MCU from executing the next instruction for Period ms.																		
PEEK Address, Variable	Return the value at the register Address in Variable.																		
POKE Address, Value	Write the register Address with the Value.																		

(Continued)

TABLE E.14 PICBASIC BUILT IN FUNCTIONS (CONTINUED)

INSTRUCTION	DESCRIPTION
POT Pin, Scale, Variable	Read a potentiometer's wiper when one of its pins is connected to a capacitor; Scale is a value that will change the returned value until it is in the range of 0 to 0xFF (255).
PULSIN Pin, State, Variable	Measure an incoming pulse width of Pin; State indicates the state of the expected pulse; if a 4-MHz clock is used with the PIC MCU, the time intervals have a granularity of 10 ms.
PULSOUT Pin, Period	Pulse the Pin for the Period; if the PIC MCU is run with a 4-MHz clock, then the pulse Period will have a granularity of 10 ms.
PWM Pin, Duty, Cycle	Output a pulse-width-modulated signal on Pin; each cycle is 5 ms long for a PIC MCU running at 4 MHz; Duty selects the fraction of the cycles (0 to 255) that the PWM is active; Cycle specifies the number of cycles that is output.
RANDOM Variable	Load Variable with a pseudorandom variable.
RCTIME Pin, State, Variable	Measure the absolute time required for a signal to be delayed in an RC network; if a 4-MHz oscillator is used with the PIC MCU, then the value returned will be in 10-ms increments.
READ Address, Variable	Read the byte in the built-in data EEPROM at Address and return its value into Variable; this instruction does not work with the built-in EEPROM of PIC12CExx parts.
RESUME {Label}	Restore execution at the instruction after the ON DEBUG or ON INTERRUPT instruction handler was executed; if a Label is specified, then the hardware is returned to its original state and execution jumps to the code after Label.
RETURN	Return to the instruction after the calling GOSUB.
REVERSE Pin	Reverse the function of the specified Pin; for example, if it were in output mode, it would be changed to input mode.
SERIN Pin, Mode,{Timeout, Label,} {Qual,...} [Variable {,Variable...}]	Receive one or more asynchronous data bytes on Pin; the Pin can be defined at run time; the Qual bytes are test qualifiers that only pass following bytes when the first byte of the incoming string matches; the Timeout value is in ms, and execution jumps to Label when the Timeout interval passes without any data being received; Mode is used to specify the operation of the Pin and is defined next:

```
SERIN2 Pin{\FlowPin},
Mode,{ParityLabel},{
Timeout,Label},{
[Specification]
```

Mode	Baud Rate	State
T300	300	Pos
T1200	1200	Pos
T2400	2400	Pos
T9600	9600	Pos
N300	300	Neg
N1200	1200	Neg
N2400	2400	Neg
N9600	9600	Neg

Receive one or more asynchronous data bytes on Pin; FlowPin is used to control the input of data to the PIC MCU to make sure that there is no overrun; if even parity is selected in the Mode parameter, then anytime an invalid byte is received, execution will jump to the ParityLabel; input timeouts can be specified in 1-ms intervals, with no data received in the specified period causing execution to jump to Label; Mode selection is made by passing a 16-bit variable to the SERIN2 instruction. The bits are defined as

Bit	Function
15	Unused
14	Set if input data is negative
13	Set if even parity is to be used with the data
12–0	Data rate specification found by the formula

$$\text{Rate} = (1,000,000/\text{baud}) - 20$$

The Specification is a string of data qualifiers/modifiers and destination variables that is used to filter and process the incoming data; the qualifiers/modifiers are listed below:

Modifier	Operation
Bin{1...16} Var	Receive up to 16 binary digits and store in Var
DEC{1...5} Var	Receive up to 5 decimal digits and store in Var
HEX{1...4} Var	Receive up to 4 hexadecimal digits and store in Var

(Continued)

TABLE E.14 PICBASIC BUILT IN FUNCTIONS (CONTINUED)

INSTRUCTION	DESCRIPTION			
SEROUT Pin, Mode, [Value{,Value...}]	SKIP #	Skip # received characters		
	STR Array\n\c	Receive a string of <i>n</i> characters and store in Array; optionally ended by character c		
	WAIT("String")	Wait for the specified string of characters		
	WAITSTR Array\n	Wait for a character string <i>n</i> characters long		
	Send one or more asynchronous data bytes on Pin; the Pin can be defined at run time; mode is used to specify the operation of the pin and the output driver [CMOS totem pole or open-drain (OD)] and is defined below:			
	Mode	Baud Rate	State	Driver
	T300	300	Positive	CMOS
	T1200	1200	Positive	CMOS
	T2400	2400	Positive	CMOS
	T9600	9600	Positive	CMOS
	N300	300	Negative	CMOS
	N1200	1200	Negative	CMOS
	N2400	2400	Negative	CMOS
	N9600	9600	Negative	CMOS
	OT300	300	Positive	OD
	OT1200	1200	Positive	OD
	OT2400	2400	Positive	OD
	OT9600	9600	Positive	OD
	ON300	300	Negative	OD
	ON1200	1200	Negative	OD

```
SEROUT2 Pin{\FlowPin},
Mode,{Pace,} {Timeout,
Label,} [Specification]
```

ON2400	2400	Negative	OD
ON9600	9600	Negative	OD

Send one or more asynchronous data bytes on Pin; FlowPin is used to control the output of data to the PIC MCU to make sure that there is no overrun; timeouts can be specified in 1-ms intervals with no flow control on the receiver, the specified period causing execution to jump to Label; the optional Pace parameter is used to specify the length of time (measured in μ s) that the PIC MCU delays before sending out the next character; Mode selection is made by passing a 16-bit variable to the SERIN2 instruction; the bits are defined as

Bit	Function
15	CMOS/open drain driver specification; if set, open drain output
14	Set if input data is negative
13	Set if even parity is to be used with the data
12–0	Data rate specification, found by the formula

Rate = (1,000,000/baud) – 20

The Specification is a string of data qualifiers/modifiers and source values that is used to format the outgoing data; the output format data can be specified with an I prefix to indicate that the data type is to be sent before the data, and the S prefix indicates that a sign (“–”) indicator is sent for negative values; the qualifiers/modifiers are listed below:

Modifier	Operation
Bin{1...16} Var	Receive up to 16 binary digits and store in Var
DEC{1...5} Var	Receive up to 5 decimal digits and store in Var
HEX{1...4} Var	Receive up to 4 hexadecimal digits and store in Var
SKIP #	Skip # received characters
STR Array\n\c	Receive a string of <i>n</i> characters and store in Array; optionally ended by character <i>c</i>

```
WAIT("String")
```

Wait for the specified String of characters.

```
WAITSTR Array\n
```

Wait for a character string *n* characters long.

(Continued)

TABLE E.14 PICBASIC BUILT IN FUNCTIONS (CONTINUED)

INSTRUCTION	DESCRIPTION										
SHIFTIN DataPin, ClockPin, Mode, [Variable{\Bits} {,Variable...}]	Synchronously shift data into the PIC MCU; the Bits parameter is used to specify the number of bits that are actually shifted in (if Bits is not specified, the default is 8); the Mode parameter is used to indicate how the data is to be transferred, and the values are listed below: <table> <tr> <th>Mode</th><th>Function</th></tr> <tr> <td>MSBPRE</td><td>Most significant bit first; read data before pulsing clock</td></tr> <tr> <td>LSBPRE</td><td>Least significant bit first; read data before pulsing clock</td></tr> <tr> <td>MSBPOST</td><td>Most significant bit first; read data after pulsing clock</td></tr> <tr> <td>LSBPOST</td><td>Least significant bit first; read data after pulsing clock</td></tr> </table>	Mode	Function	MSBPRE	Most significant bit first; read data before pulsing clock	LSBPRE	Least significant bit first; read data before pulsing clock	MSBPOST	Most significant bit first; read data after pulsing clock	LSBPOST	Least significant bit first; read data after pulsing clock
Mode	Function										
MSBPRE	Most significant bit first; read data before pulsing clock										
LSBPRE	Least significant bit first; read data before pulsing clock										
MSBPOST	Most significant bit first; read data after pulsing clock										
LSBPOST	Least significant bit first; read data after pulsing clock										
SHIFTOUT DataPin, ClockPin, Mode, [Variable{\Bits} {,Variable...}]	Synchronously shift data out of the PIC MCU; the Bits parameter is used to specify how many bits are to be shifted out in each word (if not specified, the default is 8); the Mode parameter is used to specify how the data is to be shifted out, and the values are listed below: <table> <tr> <th>Mode</th><th>Function</th></tr> <tr> <td>LSBFIRST</td><td>Least significant bit first</td></tr> <tr> <td>MSBFIRST</td><td>Most significant bit first</td></tr> </table>	Mode	Function	LSBFIRST	Least significant bit first	MSBFIRST	Most significant bit first				
Mode	Function										
LSBFIRST	Least significant bit first										
MSBFIRST	Most significant bit first										
SLEEP Period	Put the PIC MCU into Sleep mode for Period seconds.										
SOUND Pin, [Note, Duration{,Note,Duration...}]	Output a string of tones and durations (which can be used to create a simple tune) on the Pin; note 0 is silence, and notes 128 to 255 are white noise; note 1 (78.5 Hz for a 4-MHz PIC MCU) is the lowest valid tone, and note 127 is the highest (10 kHz in a 4-MHz PIC MCU); duration is specified in 12-ms increments.										
STOP	Place the PIC MCU into an endless loop; the PIC MCU is not put into Sleep mode.										
SWAP Variable, Variable	Exchange the values in the two variables.										
TOGGLE Pin	Toggle the output value of the specified pin.										

WHILE Cond : WEND	Execute the code between the WHILE and the WEND statements while the Cond returns a non zero value.
WRITE Address, Value	Write the byte Value into the built-in data EEPROM; this instruction will not work with the built-in EEPROM in PIC12CExxx devices.
XIN DataPin, ZeroPin, {Timeout,Label,} [Variable {,Variable...}]	Receive data from X-10 devices; ZeroPin is used to detect the zero crossing of the input ac signal; both DataPin and ZeroPin should be pulled up with 4.7-k Ω resistors; the optional timeout (specified in 8.33-ms intervals) will cause execution to jump to Label if no data is received by the specified interval; if the first variable data destination is 16 bits, then both the house code and the key code will be saved; if the first variable is 8 bits in size, then only the key code will be saved.
XOUT DataPin, ZeroPin, [HouseCode\KeyCode {\Repeat} {,Value...}]	Send X-10 data to other devices; the ZeroPin is an input and should be pulled up with a 4.7-k Ω resistor; HouseCode is a number between 0 and 15 and corresponds to the house code set on the X-10 modules through P; the KeyCode can be either the number of a specific X-10 receiver or the function to be performed by the module.

This page intentionally left blank

C PROGRAMMING LANGUAGE

For modern systems, C is the programming language of choice because it is available for a wide range of systems and processors (including the PIC[®] microcontroller). This ubiquity requires anyone who is planning on developing applications for processing systems to have at least a passing knowledge of the language. C is often referred to as the *universal assembly language* because it is designed in such a way that it can access a system's lowest levels efficiently.

In this book I will be presenting pseudocode in C format to help explain the operation of the hardware without getting bogged down in the details of assembly language. In addition, the available third-party compilers that I present are designed primarily for the C language.

Pseudocode is a term for code that is designed to illustrate the operation of an algorithm or hardware interface. My choice of using C format means that you should have at least a passing understanding of the language. This section will quickly give you an understanding of how C is designed and how statements are defined.

Throughout this book, the code examples will be in either assembly language or C-styled assembly language.

Constant declaration:

```
const int Label = Value;
```

Variable declaration:

```
type Label [= Value];
```

Value is an optional initialization constant, where `type` can be

```
char  
int  
unsigned int  
float
```

Note that `int` is defined as the *word size* of the processor/operating system. For PCs, an `int` can be a *word* (16 bits) or a *double word* (32 bits).

There also may be other basic types defined in the language implementation. Single-dimensional arrays are declared using the form

```
type Label[ Size ] [= { Initialization Values..}];
```

Note that the array `Size` is enclosed within square brackets ([and]) and should not be confused with the optional `Initialization Values`.

Strings are defined as single-dimensional ASCIIZ arrays:

```
char String[ 17 ] = "This is a String";
```

where the last character is an ASCII NUL.

Strings also can be defined as pointers to characters:

```
char *String = "This is a String"
```

This implementation requires that the text "This is a String" be stored in two locations (in code and in data space). For the PIC microcontroller and other Harvard-architected processors, the text data could be written into data space when the application first starts up as part of the language's initialization.

Multidimensional arrays are defined with each dimension identified separately within square brackets ([and]):

```
int ThreeDSpace[ 32 ][ 32 ][ 32 ];
```

Array dimensions must be specified unless the variable is a pointer to a single-dimensional array.

Pointers are declared with the `*` character after the type:

```
char * String = "This is a String";
```

Accessing the address of the pointer in memory is accomplished using the `&` character:

```
StringAddr = &String;
```

Accessing the address of a specific element in a string is accomplished using the `&` character and a string array element:

```
StringStart = &String[ n ];
```

In the PC, it is recommended that `far` (32-bit) pointers always be used with absolute offset: segment addresses within the PC memory space to avoid problems with varying segments.

The variable's **Type** can be "overridden" by placing the new type in front of the variable in brackets:

```
[long] StringAddr = 0x0123450000;
```

Application start:

```
main(envp)
  char *envp;
{ // Application Code

    :                               // Application Code

} // End Application
```

Function format:

```
Return_Type Function( Type Parameter [, Type Parameter..])
{ // Function Start

    :                               // Function Code

    return value;

} // End Function
```

Function prototype:

```
Return_Type Function ( Type Parameter [, Type Parameter..]);
```

Expression:

```
[(..] Variable | Constant [Operator [(..] Variable |
Constant [)]..]
```

Assignment statement:

```
Variable = Expression;
```

C conditional statements consist of **if**, **?**, **while**, **do**, **for**, and **switch**. The **if** statement is defined as

```
if ( Statement )
; | { Assignment Statement | Conditional Statement.. } | Assignment
Statement | Conditional Statement
    [else ;| { Assignment Statement | Conditional Statement..} | Assignment
Statement | Conditional Statement ]
```

The `? :` statement evaluates the statement (normally a comparison) and, if not equal to zero, executes the first statement, else execute the statement after the `:`.

Statement ? Assignment Statement | Conditional Statement : Assignment Statement | Conditional Statement

The `while` statement is added to the application following the definition below:

```
while ( Statement ) ; | { Assignment Statement | Conditional Statement..
} | Assignment Statement | Conditional Statement
```

The `for` statement is defined as

```
for ( initialization (Assignment) Statement; Conditional Statement;
Loop Expression (Increment) Statement )
; | { Assignment Statement | Conditional Statement.. } | Assignment
Statement | Conditional Statement
```

To jump out of a currently executing loop, use the `break` statement:

```
break;
```

The `continue` statement skips over remaining code in a loop and jumps directly to the loop condition (for use with `while`, `for`, and `do/while` loops). The format of the statement is

```
continue;
```

For looping until a condition is true, the `do/while` statement is used:

```
do
Assignment Statement | Conditional Statement..
while ( Expression );
```

To conditionally execute according to a value, the `switch` statement is used:

```
switch( Expression ) {
case Value:           // Execute if "Statement" == "Value"
[ Assignment Statement | Conditional Statement.. ]
[break;]
default:              // If no "case" Statements are True
[ Assignment Statement | Conditional Statement.. ]
} // End switch
```

Finally, the `goto Label` statement is used to jump to a specific address:

```
goto Label;
```

Label:

To return a value from a function, the `return` statement is used:

```
return Statement;
```

Statement operators are listed in Table F.1.

Compound assignment operators are listed in Table F.2.

The order of operations for C operators is listed in Table F.3.

All directives start with `#` and are executed before the code is compiled. The common directives are listed in Table F.4.

TABLE F.1 C OPERATORS AND THEIR ARITHMETIC OPERATION

OPERATOR	ARITHMETIC OPERATION
!	Logical negation
^	Bitwise negation
&&	Logical AND
&	Bitwise AND, address of variable or structure
	Logical OR
	Bitwise OR
^	Bitwise XOR
+	Addition
++	Increment
-	Subtraction, negation
--	Decrement
*	Multiplication, indirection (pointer to variable)
/	Division
%	Modulus
==	Conditional equals
!=	Conditional not equals
<	Conditional less than
<=	Conditional less than or equals to
<<	Shift left
>	Conditional greater than
>=	Conditional greater than or equals to
>>	Shift right

TABLE F.2 C COMPOUND ASSIGNMENT STATEMENT OPERATORS

OPERATOR	OPERATION
&=	AND with the assignment variable and store the result in the assignment variable
=	OR with the assignment variable and store the result in the assignment variable
^=	XOR with the assignment variable and store the result in the assignment variable
+=	ADD to the assignment variable
-=	SUBTRACT from the assignment variable
*=	MULTIPLY the assignment variable
/=	DIVIDE the assignment variable
%=	MODULUS of the assignment variable division operation
<<=	Shift the assignment variable left
>>=	Shift the assignment variable right

TABLE F.3 C OPERATOR ORDER OF OPERATIONS

OPERATORS	PRIORITY	TYPE
() [] . ->	Highest	Expression evaluation
- ~ ! & * ++ --		Unary operators
* / %		Multiplicative operators
+ -		Additive operators
<< >>		Shifting operators
< <= > >=		Comparison operators
== !=		Comparison operators
&		Bitwise AND
^		Bitwise XOR
		Bitwise OR
&&		Logical AND
		Logical OR
?:		Conditional execution
= &= = ^= += -= *= /= %= >>= << =		Assignments
,	Lowest	Sequential evaluation

TABLE F.4 C DIRECTIVES

DIRECTIVE	FUNCTION
<code>#define Label</code> <code>[(Parameters)] Text</code>	Define a <code>Label</code> that will be replaced with <code>Text</code> when it is found in the code; if <code>Parameters</code> are specified, then replace them in the code, similar to a macro.
<code>#undefine Label</code>	Erase the defined <code>Label</code> and <code>Text</code> in memory.
<code>#include "File" <File></code>	Load the specified file in line to the text; when <code><</code> and <code>></code> encloses the filename, then the file is found using the <code>INCLUDE</code> environment path variable; if <code>"</code> and <code>"</code> enclose the filename, then the file in the current directory is searched before checking the <code>INCLUDE</code> path.
<code>#error Text</code>	Force the error listed in <code>Text</code> .
<code>#if Condition</code>	If the <code>Condition</code> is true, then compile the following code to <code>#elif</code> , <code>#else</code> , or <code>#endif</code> ; if the <code>Condition</code> is false, then ignore the following code to <code>#elif</code> , <code>#else</code> , or <code>#endif</code> .
<code>#ifdef Label</code>	If the <code>#define</code> label exists, then compile the following code; <code>#elif</code> , <code>#else</code> , and <code>#endif</code> work as expected with <code>#if</code> .
<code>#ifndef Label</code>	If the <code>#define</code> label does <i>not</i> exist, then compile the following code; <code>#elif</code> , <code>#else</code> , and <code>#endif</code> work as expected with <code>#if</code> .
<code>#elif Condition</code>	This directive works as an <code>#else #if</code> to avoid lengthy nested <code>#ifs</code> ; if the previous condition was false, checks the condition.
<code>#else</code>	Placed after <code>#if</code> or <code>#elif</code> and toggles the current compile condition; if the current compile condition was false, after <code>#else</code> , it will be true; if the current compile condition was true, after <code>#else</code> , it will be false.
<code>#endif</code>	Used to end an <code>#if</code> , <code>#elif</code> , <code>#else</code> , <code>#ifdef</code> , or <code>#ifndef</code> directive.
<code>#pragma String</code>	This is a compiler-dependent directive with different <code>Strings</code> required for different cases.

TABLE F.5 C “BACKSLASH” CHARACTERS

STRING	ASCII	CHARACTER
\r	0x0D	Carriage return (CR)
\n	0x0A	Line feed (LF)
\f	0x0C	Form feed (FF)
\b	0x08	Backspace (BS)
\t	0x09	Horizontal tab (HT)
\v	0x0B	Vertical tab (VT)
\a	0x07	Alarm/bell (BEL)
\'	0x27	Single quote (')
\"	0x22	Double quote (")
\\	0x5C	Backslash (\)
\ddd	N/A	Octal number
\xdd	0xdd	Hexadecimal character

The following words cannot be used in C applications as labels:

break
case
continue
default
do
else
for
goto
if
return
switch
while

Some constants are defined as “backslash” characters and are listed in Table F.5.

Common Library Functions

Table F.6 lists the common C functions as defined by Kernighan and Ritchie.

TABLE F.6 COMMON C FUNCTIONS

FUNCTION PROTOTYPE	DESCRIPTION
<code>int getchar(void)</code>	Get one character from “standard input” (the keyboard); if no character is available, then wait for it.
<code>int putchar(int)</code>	Output one character to the “standard output” (the screen).
<code>int printf(char *Const[, arg...])</code>	Output the Const string text; “escape sequence” characters for output are embedded in the Const string text; different data outputs are defined using the “conversion characters”: %d, %I—decimal integer %o—octal integer %x, %X—hex integer (with upper- or lower-case values); no leading 0x character string output %u—unsigned integer %c—single ASCII character %s—ASCIIZ string %f—floating point %e, %E—floating point with the precision specified by # %g, %G—floating point %p—pointer %%—print % character Different C implementations will have different printf parameters.
<code>int scanf(char *Const, arg [, *arg...])</code>	Provide formatted input from the user; the Const ASCIIZ string is used as a Prompt for the user; note that the input parameters are always pointers; conversion characters are similar to printf: %d—decimal integer %I—integer; in octal, if leading 0 or hex if leading 0x or 0X %o—octal integer (leading 0 not required) %x—hex integer (leading 0x or 0X not required) %c—single character %s—ASCIIZ string of characters; when saved, a NULL character is put at the end of the string %e, %f, %g—floating-point value with optional sign, decimal point, and exponent %%—display % character in prompt
<code>handle fopen(char *FileName, char *mode)</code>	Open file and return handle (or NULL for error); mode is a string consisting of the optional characters: r—open file for reading w—open file for writing a—open file for appending to existing files some systems handle Text and Binary files; a Text file has the CR/LF characters represented as a single CR; a Binary file does not delete any characters.

(Continued)

TABLE F.6 COMMON C FUNCTIONS (CONTINUED)	
FUNCTION PROTOTYPE	DESCRIPTION
int fclose(handle)	Close the file.
int getc(handle)	Receive data from a file one character at a time; if at the end of an input file, then EOF is returned.
int putc(handle, char)	Output data to a file one character at a time; error is indicated by EOF returned.
int fprintf(handle, char *Const[, arg...])	Output string of information to a file; the same “conversion characters” and arguments as printf are used.
int fscanf(handle, char *Const, arg[, arg...])	Input and process string of information from a file; the same “conversion characters” and arguments as scanf are used.
int fgets(char *Line, int LineLength, handle)	Get the ASCIIZ string from the file.
int fputs(char *line, handle)	Output an ASCIIZ string to a file.
strcat(Old, Append)	Put ASCIIZ Append string on the end of the Old ASCIIZ string.
strncat(Old, Append, #)	Put # of characters from Append on the end of the Old ASCIIZ string.
int strcmp(String1, String2)	Compare two ASCIIZ strings; zero is returned for match, negative for String1 < String2, and positive for String1 > String2.
int strncmp(String1, String2, #)	Compare two ASCIIZ strings for # characters; zero is returned for match, negative for String1 < String2, and positive for String1 > String2.
strcpy(String1, String2)	Copy the contents of ASCIIZ String2 into String1.
strncpy(String1, String2, #)	Copy # characters from String2 into String1.
strlen(String)	Return the length of ASCIIZ character String.
int strchr(String, char)	Return the position of the first courses in the ASCIIZ String.
int strrchr(String, char)	Return the position of the last char in the ASCIIZ String.
system(String)	Executes the system command String.
*malloc(size)	Allocate the specified number of bytes of memory; if insufficient space available, return NUL.

TABLE F.6 COMMON C FUNCTIONS (CONTINUED)

FUNCTION PROTOTYPE	DESCRIPTION
<code>*calloc(#, size)</code>	Allocate memory for the specified # of data elements of size.
<code>free(*)</code>	Free the memory.
<code>float sin(angle)</code>	Find the sine of angle (which is in radians).
<code>float cos(angle)</code>	Find the cosine of angle (which is in radians).
<code>float atan2(y, x)</code>	Find the arctangent of x and y in radians.
<code>float exp(x)</code>	Calculate the natural exponent.
<code>float log(x)</code>	Calculate the natural logarithm.
<code>float log10(x)</code>	Calculate the base 10 logarithm.
<code>float pow(x, y)</code>	Calculate x to the power y.
<code>float sqrt(x)</code>	Calculate the square root of x.
<code>float fabs(x)</code>	Calculate the absolute value of X.
<code>float frand()</code>	Get a random number.
<code>int isalpha(char)</code>	Return nonzero if character is a to z or A to Z.
<code>int isupper(char)</code>	Return nonzero if character is A to Z.
<code>int islower(char)</code>	Return nonzero if character is a to z.
<code>int isdigit(char)</code>	Return nonzero if character is 0 to 9.
<code>int isalnum(char)</code>	Return nonzero if character is a to z, A to Z, or 0 to 9.
<code>int isspace(char)</code>	Return nonzero if character is " ", HT, LF, CR, FF, or VT.
<code>int toupper(char)</code>	Convert the character to upper case.
<code>int tolower(char)</code>	Convert the character to lower case.

PICC Library Functions

Table F.7 lists the functions available in HT-Soft's PICC language. When using these libraries, make sure that the correct `include` files (listed in the PICC manual) are included. Note that only functions are listed here; macros and defines will have to be found in the PICC documentation. In addition, this list should not be considered the ultimate reference to these functions.

TABLE F.7 HI-TECH SOFTWARE'S PICC COMPILER LIBRARY FUNCTIONS

FUNCTION PROTOTYPE	DESCRIPTION
<code>int abs(int i)</code>	Return the absolute positive value of <code>i</code> .
<code>double acos(double f)</code>	Return the value in radians of the converse of <code>cos(f)</code> .
<code>char * asctime(struct tm * t)</code>	Convert time in <code>tm</code> structure into a pointer to an ASCII string in the format <code>Fri Sep 15 01:23:45 2006\n\0</code> .
<code>double asin(double f)</code>	Return the value in radians of the converse of <code>sin(f)</code> .
<code>double atan(double f)</code>	Return the value in radians of the converse of <code>tan(f)</code> .
<code>double atof(const char * s)</code>	Convert the string passed to the function to a floating-point number; if invalid number, returns 0.0.
<code>int atoi(const char * s)</code>	Convert the string passed to the function to an integer number; if invalid number, returns 0.
<code>long atoi(const char * s)</code>	Convert the string passed to the function to a long integer number; if invalid number, returns 0.
<code>void * bsearch(const void * key, void * base, size_t n_memb, size_t size, int (*compare)(const void *, const void *))</code>	Search a sorted array for an element matching key.
<code>double ceil(double f)</code>	Returns the smallest whole number not less than <code>f</code> .
<code>char * cgets(char * s)</code>	Perform repeated calls to <code>getche()</code> to read one line of input from the console and return an ASCII string.
<code>double cos(double f)</code>	Return the cosine of <code>f</code> (which is in radians).
<code>double cosh(double f)</code>	Return the hyperbolic cosine of <code>f</code> .
<code>void cputs (const char * s)</code>	Writes ASCII string <code>s</code> to the console.
<code>char * ctime(time_t *t)</code>	Converts the time in seconds to the string described in <code>asctime</code> .
<code>div_t div(int numerator, int demoninator)</code>	Return the quotient and remainder of the integer division.
<code>Double eval_poly (double x, const double * d, int n)</code>	Evaluates a polynomial with coefficients in array <code>d</code> .
<code>double exp(double f)</code>	Return the exponent value of the argument.
<code>double fabs(double f)</code>	Return the absolute value of the floating-point argument.

TABLE F.7 HI-TECH SOFTWARE'S PICC COMPILER LIBRARY FUNCTIONS (CONTINUED)

FUNCTION PROTOTYPE	DESCRIPTION
<code>double floor(double f)</code>	Return the largest whole number not greater than <code>f</code> .
<code>double frexp(double f, int * p)</code>	Convert the floating-point number <code>f</code> into a fraction and power of 2.
<code>char getch(void)</code>	Read a single character from the console and echo on back to the console.
<code>char getche(void)</code>	Read a single character from the console without echoing it back.
<code>char * gets(char * s)</code>	Read a line from the standard input into buffer <code>s</code> as an ASCII string; equivalent to <code>cgets()</code> in a PIC microcontroller.
<code>struct tm *gmtime(time_t * t)</code>	Convert number of seconds starting from 00:00:00 January 1, 1970, into <code>tm</code> structure.
<code>double ldexp(double f, int i)</code>	Integer <code>i</code> is added to the exponent of floating-point <code>f</code> and returned; this is the inverse of <code>frexp</code> .
<code>ldiv_t ldiv(long numerator, long denominator)</code>	Return the quotient and remainder from the division operation.
<code>struct tm * localtime(time_t * t)</code>	Converts time pointed to by <code>t</code> into <code>tm</code> structure; this is the same function as <code>gmtime</code> .
<code>double log(double f)</code>	Return the natural logarithm of <code>f</code> .
<code>double log10(double f)</code>	Return the base 10 logarithm of <code>f</code> .
<code>void longjmp(jmp_buf buf, int val)</code>	Jump to addresses outside of current execution page; normally used with <code>setjmp</code> .
<code>int memcmp(const void * s1, const void * s2, size_t n)</code>	Compares two blocks of memory of length <code>n</code> and returns signed value; comparison does not stop at null character.
<code>double modf(double value, double *iptr)</code>	Split value into integer and fractional parts, each having the same sign as value.
<code>int persist_check(int flag)</code>	Test hidden value and checksum in EEPROM set up by <code>persist_validate</code> .
<code>void persist_validate(void)</code>	Executed after persistent variable stored in EEPROM is altered to reset hidden value and checksum.
<code>double pow(double f, double p)</code>	Return value <code>f</code> raised to the power <code>p</code> .

(Continued)

TABLE F.7 HI-TECH SOFTWARE'S PICC COMPILER LIBRARY FUNCTIONS (CONTINUED)

FUNCTION PROTOTYPE	DESCRIPTION
<code>void putch(char c)</code>	Output character <code>c</code> to console.
<code>int putchar(int c)</code>	Not available in PICC.
<code>int puts(const char * s)</code>	Output ASCIIZ string <code>s</code> to console.
<code>void qsort(void * base, size_t nel, size_t width, int (* func)(const void *, const void *))</code>	Sort array of <code>nel</code> items, each of length <code>width</code> , starting at <code>base</code> ; function <code>func</code> compares items and returns value less than, equal to, or greater than zero.
<code>int rand(void)</code>	Produce a pseudorandom value; initialized with <code>srand</code> function.
<code>int scanf(const char * fmt, ...)</code>	Break up incoming <code>stdin</code> string into data segments and store in appropriate variables.
<code>int setjmp(jmp_buf buf)</code>	Specify addresses for <code>longjmp</code> function.
<code>double sin(double f)</code>	Return the sine of <code>f</code> (which is in radians).
<code>double sinh(double f)</code>	Return the hyperbolic sine of <code>f</code> .
<code>double sqrt(double f)</code>	Return the square root of <code>f</code> using Newton's method.
<code>void srand(unsigned int seed)</code>	Initialize the random number generator.
<code>char * strcat(char * destination, char * source)</code>	Concatenate source to the end of destination.
<code>char * strchr(const char * s, int c)</code>	Search string <code>s</code> for character <code>c</code> and return pointer to string starting at <code>c</code> .
<code>char * strchr(const char * s, int c)</code>	Case-insensitive search of string <code>s</code> for character <code>c</code> and return pointer to string starting at <code>c</code> .
<code>int strcmp(const char * s1, const char * s2)</code>	Compare two strings and return an integer to indicate less than, equal to, or greater than.
<code>int stricmp(const char * s1, const char * s2)</code>	Case-insensitive compare of two strings and return an integer to indicate less than, equal to, or greater than.
<code>char * strcpy(char * destination, char * source)</code>	Copy the source string into the memory pointed to by the destination.

TABLE F.7 HI-TECH SOFTWARE'S PICC COMPILER LIBRARY FUNCTIONS (CONTINUED)

FUNCTION PROTOTYPE	DESCRIPTION
<code>size_t strcspn(const char * s1, const char s2)</code>	Return the number of characters in s1 that do not include s2.
<code>size_t strlen(const char * s)</code>	Return the number of characters in s.
<code>char * strncat(char * destination, char * source, size_t n)</code>	Concatenate source to the end of destination for n characters.
<code>int strncmp(const char * s1, const char * s2, size_t n)</code>	Compare two strings up to a maximum of n characters and return an integer to indicate less than, equal to, or greater than.
<code>int strnicmp(const char * s1, const char * s2, size_t n)</code>	Case-insensitive compare of two strings up to a maximum of n characters and return an integer to indicate less than, equal to, or greater than.
<code>char * strncpy(char * destination, char * source, size_t n)</code>	Copy n characters from source into the destination.
<code>char * strpbrk(const char * s1, const char * s2)</code>	Return a pointer to the first occurrence of a character from s2 in s1.
<code>char * strrchr(char * s, int c)</code>	Search from the end of a string to the last occurrence of c.
<code>char * strrichr(char * s, int c)</code>	Case-insensitive search from the end of a string to the last occurrence of c.
<code>size_t strspn(const char * s1, const char * s2)</code>	Return the length of the initial segment of s1 consisting of characters found in s2.
<code>char * strstr(const char * s1, const char * s2)</code>	Return a pointer to s2 found in s1.
<code>char * stristr(const char * s1, const char * s2)</code>	Case-insensitive return to a pointer to s2 found in s1.
<code>char * strtok(char * s1, const char * s2)</code>	Break up s1 into a series of smaller strings as specified by s2.
<code>double tan(double f)</code>	Return the tangent of f (which is in radians).
<code>double tanh(double f)</code>	Return the hyperbolic tangent of f.

(Continued)

TABLE F.7 HI-TECH SOFTWARE'S PICC COMPILER LIBRARY FUNCTIONS (CONTINUED)	
FUNCTION PROTOTYPE	DESCRIPTION
<code>time_t time(time_t * t)</code>	User-written function that returns the number of seconds since 00:00:00 on January 1, 1970.
<code>char tolower(int c)</code>	Convert the character to lower case.
<code>char toupper(int c)</code>	Convert the character to upper case.
<code>void ungetch(char c)</code>	Push <code>c</code> back onto the console input stack for later processing.
<code>long xtoi(const char * s)</code>	Convert the hexadecimal number string passed to the function to an integer value.

Microchip C18 Library Functions

Table F.8 lists the functions available in Microchip's C18 compiler libraries that are beyond the standard C libraries listed in Table F.6. When using these library functions, make sure that the correct `include` files (listed in the manual) are included. Microchip's documentation for these functions is very complete and comprehensive. Along with having very complete documentation, when you look through the list of functions, you will see that they were designed for microcontroller applications specifically.

Note that only functions are listed here; macros and defines may be required to work with these functions, and their descriptions are found in the C18 compiler documentation. The list in Table F.8 should not be considered the ultimate reference to these functions. When multiple hardware devices of a specific type are available, the `#` character is used to indicate that there are multiple functions, each one devoted to a specific piece of hardware. Some functions either do not specify a specific hardware device or can be selected from multiple devices, in which case the optional device number is indicated by the `(#)` string.

TABLE F.8 MICROCHIP C18 C COMPILER LIBRARY FUNCTIONS

FUNCTION PROTOTYPE	DESCRIPTION
<code>void AckI2C(#)</code>	Generate I2C acknowledge condition for single or multiple (#) peripherals.
<code>signed char atob (const char * s)</code>	Convert a string to an 8-bit signed byte.
<code>double atof (const char * s)</code>	Convert a string to a floating-point number.
<code>int atoi(const char * s)</code>	Convert a string to a 16-bit signed integer.
<code>long atol(const char * s)</code>	Convert a string to a long integer.
<code>char * btoa(signed char value, char * string)</code>	Convert an 8-bit signed byte to a string.
<code>void baud(#)USART(unsigned char baudconfig)</code>	Set the baud rate configuration bits for the specified enhanced USART.
<code>char BusyADC(void)</code>	Poll ADC hardware and return true if already executing.
<code>char Busy(#)USART (void)</code>	Return nonzero if the USART is transmitting.
<code>unsigned char BusyXLCD(void)</code>	Return nonzero if the LCD controller is busy.
<code>void CAN2510BitModify(unsigned char addr, unsigned char mask, unsigned char data)</code>	Modify the specific bits in the MCP2510 CAN bus controller.
<code>unsigned char CAN2510ByteRead(unsigned char address)</code>	Read the specified register of the MCP2510 CAN bus controller
<code>void CAN2510ByteWrite(unsigned char address, unsigned char value)</code>	Write to the specified register of the MCP2510 CAN bus controller.
<code>unsigned char CAN2510DataRead(unsigned char buffernum, unsigned long * msgid, unsigned char * numbytes, unsigned char * data)</code>	Read a message from the specified MCP2510 receive buffer.
<code>unsigned char CAN2510DataReady(unsigned char buffernum)</code>	Return nonzero value if a message was found in the specified buffer of the MCP2510.

(Continued)

TABLE F.8 MICROCHIP C18 C COMPILER LIBRARY FUNCTIONS (CONTINUED)

FUNCTION PROTOTYPE	DESCRIPTION
<code>void CAN2510Disable(void)</code>	Disables the chip select of the MCP2510.
<code>void CAN2510Enable(void)</code>	Enables the chip select of the MCP2510.
<code>unsigned char CAN2510ErrorState(void)</code>	Return the error state of the CAN bus interfaced by the MCP2510.
<code>unsigned char CAN2510Init (unsigned char bufferconfig, unsigned short long bittimeconfig, unsigned char interruptenables, unsigned char spi_syncmode, unsigned char spi_busmode, unsigned char spi_smpphase)</code>	Initialize PIC18 SPI port to communicate with MCP2510.
<code>void CAN2510InterruptEnable (unsigned char interruptenables)</code>	Modify the CAN2510 interrupt enable bits.
<code>unsigned char CAN2510InterruptStatus(void)</code>	Return the source of the MCP2510 interrupt request.
<code>void CAN2510LoadBufferStd(unsigned char buffernum, unsigned int msgid, unsigned char numbytes, unsigned char * data)</code>	Load a standard data frame into MCP2510.
<code>void CAN2510LoadBufferXtd(unsigned char buffernum, unsigned long msgid, unsigned char numbytes, unsigned char * data)</code>	Load an extended data frame for transfer by the MCP2510.
<code>unsigned char CAN2510ReadMode(void)</code>	Returns the MPC2510's current operation mode.
<code>unsigned char CAN2510ReadStatus(void)</code>	Return the status of the MPC2510's transmit and receive buffers.
<code>void CAN2510SequentialWrite (unsigned char * dataarray, unsigned char CAN2510addr, unsigned char numbytes)</code>	Write a string of bytes to the MPC2510.

TABLE F.8 MICROCHIP C18 C COMPILER LIBRARY FUNCTIONS (CONTINUED)

FUNCTION PROTOTYPE	DESCRIPTION
void CAN2510SetBufferPriority (unsigned char buffernum, unsigned char bufferpriority)	Specify the priority for the specified MPC2510 buffer.
void CAN2510SetMode(unsigned char mode)	Configure the MCP2510's mode of operation.
unsigned char CAN2510SetMsgFilterStd (unsigned char buffernum, unsigned int mask, unsigned int * filters)	Configure all the MCP2510's filter and mask values of a specific buffer for a standard message.
unsigned char CAN2510SetMsgFilterXtd (unsigned char buffernum, unsigned long mask, unsigned long * filters)	Specifies the filter and mask values of the specific MCP2510 receive buffer.
void CAN2510SetSingleFilterStd (unsigned char filternum, unsigned int filter)	Configure the specified MCP2510 receive filter with a value for the standard message.
void CAN2510SetSingleFilterXtd (unsigned char filternum, unsigned int filter)	Configure the specified MCP2510 receive filter with a value for an extended message.
unsigned char CAN2510SetSingleMaskStd (unsigned char masknum, unsigned int mas)	Configure the specified MCP2510 receive buffer mask with a value for a standard-format message.
unsigned char CAN2510SetSingleMaskXtd (unsigned char masknum, unsigned int mas)	Configure the specified MCP2510 receive buffer mask with a value for an extended-format message.
unsigned char CAN2510WriteBuffer(unsigned char buffernum)	Initiate the MCP2510's buffer transmission.
unsigned char CAN2510WriteStd(unsigned int msgid, unsigned char msgpriority, unsigned char numbytes, unsigned char * data)	Write a standard-format message out of the MCP2510 using the first available transmit buffer.

(Continued)

TABLE F.8 MICROCHIP C18 C COMPILER LIBRARY FUNCTIONS (CONTINUED)

FUNCTION PROTOTYPE	DESCRIPTION
unsigned char CAN2510WriteXtd(unsigned int msgid, unsigned char msgpriority, unsigned char numbytes, unsigned char * data)	Write an extended-format message out of the MCP2510 using the first available transmit buffer.
void ClearCSSWSPI(void)	Clear the software SPI interface chip select pin.
char Clock_test(void)	Generate a delay for I2C slave clock stretching.
void CloseADC(void)	Disable ADC hardware and interrupt mechanism.
void CloseCapture#(void)	Disable input capture #.
void CloseECapture#(void)	Disable enhanced input capture #.
void CloseI2C(#) (void)	Disable the SSP(#) module.
void CloseMwire(#) (void)	Disable the SSP(#) module used for Microwire communications.
void ClosePORTB(void)	Disable interrupt request sources and internal pull-up resistors for PORTB.
void CloseEPWM1(void)	Disable enhanced PWM channel.
void ClosePWM#(void)	Disable PWM channel.
void CloseRB(#)INT(void)	Disable interrupt request sources for the specified PORTB pin.
void CloseSPI(#) (void)	Disable the SSP(#) module used for SPI communications.
void CloseTimer#(void)	Disable the specified timer.
void Close(#)USART(void)	Disable the specified USART.
void ConvertADC(void)	Start an ADC conversion.
unsigned char DataRdyI2C(#) (void)	Return nonzero if data in SSP(#) buffer.
unsigned char DataRdyMwire(#) (void)	Return nonzero if Microwire device is ready.
unsigned char DataRdySPI(#) (void)	Return nonzero if SPI data is available.
char DataRdy(#)USART(void)	Return nonzero if USART has received a character.
void Delay10TCYx(unsigned char units)	Delay in multiples of 10 instruction cycles.

TABLE F.8 MICROCHIP C18 C COMPILER LIBRARY FUNCTIONS (CONTINUED)

FUNCTION PROTOTYPE	DESCRIPTION
<code>void Delay100TCYx(unsigned char units)</code>	Delay in multiples of 100 instruction cycles.
<code>void Delay1KTCYx(unsigned char unit)</code>	Delay in multiples of 1,000 instruction cycles.
<code>void Delay10KTCYx(unsigned char unit)</code>	Delay in multiples of 10,000 instruction cycles.
<code>void DisablePullups(void)</code>	Disable PORTB's internal pull-up resistors.
<code>unsigned char EEAckPolling(#)</code> (unsigned char control)	Generate acknowledge for Microchip EEPROM I2C memory devices.
<code>unsigned char EEByteWrite(#)</code> (unsigned char control, unsigned char address, unsigned char data)	Write a single byte to the Microchip EEPROM I2C memory device.
<code>unsigned int EECurrentAddRead(#)</code> (unsigned char control)	Read a single byte from the Microchip EEPROM I2C memory device.
<code>unsigned char EEPageWrite(#)</code> (unsigned char control, unsigned char address, unsigned char * wrptr)	Write a string of data to the Microchip EEPROM I2C memory device.
<code>unsigned int EERandomRead(#)</code> (unsigned char control, unsigned char address)	Read a single byte from an arbitrary address from a Microchip EEPROM I2C memory device.
<code>unsigned char EESequentialRead(#)</code> (unsigned char control, unsigned char address, unsigned char * rdptr, unsigned char length)	Read a string of data length long starting at an arbitrary address in a Microchip EEPROM I2C memory device.
<code>void EnablePullups(void)</code>	Enable PORTB's internal pull-up resistors.
<code>unsigned char getsI2C(#)</code> (unsigned char * rdptr, unsigned char length)	Read a fixed-length string from the I2C(#) bus operating in master I2C mode.
<code>void getsMWire(#)</code> (unsigned char * rdptr, unsigned char length)	Read a string from the Microwire device.
<code>void getsSPI(#)</code> (unsigned char * rdptr, unsigned char length)	Read a string from the SPI bus.

(Continued)

TABLE F.8 MICROCHIP C18 C COMPILER LIBRARY FUNCTIONS (CONTINUED)

FUNCTION PROTOTYPE	DESCRIPTION
<code>void getsUART(char * buffer, unsigned char len)</code>	Read a string from the software USART.
<code>void gets(#)USART(char * buffer, unsigned char length)</code>	Return a string from the specified USART.
<code>void IdleI2C(#) (void)</code>	Return when specified I2C bus is available.
<code>char isBOR(void)</code>	Returns nonzero if reset was caused by a brown-out detect.
<code>unsigned char iscntrl (unsigned char ch)</code>	Return nonzero if the character is a control character.
<code>unsigned char isgraph (unsigned char ch)</code>	Return nonzero if the character is a graphic character.
<code>unsigned char islower (unsigned char ch)</code>	Return nonzero if the character is a lower-case alphabetic character.
<code>char isLVD(void)</code>	Returns nonzero if reset was caused by low-voltage detect.
<code>char isMCLR(void)</code>	Returns nonzero if reset was caused by MCLR pin.
<code>char isPOR(void)</code>	Returns nonzero if reset was caused by power-on reset.
<code>unsigned char isprint(unsigned char ch)</code>	Return nonzero if the character is printable.
<code>unsigned char ispunct(unsigned char ch)</code>	Return nonzero if the character is a punctuation character.
<code>unsigned char isspace(unsigned char ch)</code>	Return nonzero if the character is a white space character.
<code>unsigned char isupper(unsigned char ch)</code>	Return nonzero if the character is an upper-case alphabetic character.
<code>unsigned char isxdigit(unsigned char ch)</code>	Return nonzero if the character is a hexadecimal digit.
<code>char isWDTTO(void)</code>	Return nonzero if reset was caused by watchdog timer timeout.
<code>char isWDTWU(void)</code>	Return nonzero if reset was caused by wakeup caused by the watchdog timer.
<code>char isWU(void)</code>	Return nonzero if reset was caused by sleep wakeup through MCLR or interrupt.

TABLE F.8 MICROCHIP C18 C COMPILER LIBRARY FUNCTIONS (CONTINUED)

FUNCTION PROTOTYPE	DESCRIPTION
<code>char * itoa(int value, char * string)</code>	Convert a 16-bit signed integer to a string.
<code>char * ltoa(long value, char * string)</code>	Convert a signed long integer to a string.
<code>void NotAckI2C(#) (void)</code>	Generate I2C not acknowledge condition.
<code>void OpenADC(unsigned char config, unsigned char config2)</code>	Select the pin to connect the ADC and enable the ADC in the correct mode; see documentation for config and config2 values for different devices.
<code>void OpenCapture#(unsigned char config)</code>	Configure and enable input capture #.
<code>void OpenECapture# (unsigned char config)</code>	Configure and enable enhanced input capture #.
<code>void OpenI2C#((unsigned char sync_mode, unsigned char slew)</code>	Configure the SSP(#) module.
<code>void OpenMwire#((unsigned char sync_mode)</code>	Configure the SSP(#) module for Microwire operations.
<code>void OpenPORTB (unsigned char config)</code>	Configure PORTB's interrupts and internal pull-up resistors.
<code>void OpenPORTB#INT(unsigned char config)</code>	Enable PORTB interrupt request hardware.
<code>void OpenEPWM1(char period)</code>	Configure the enhanced PWM channel.
<code>void OpenPWM#(char period)</code>	Configure the specified PWM channel.
<code>void OpenSPI#((unsigned char sync_mode, unsigned char bus_mode)</code>	Initialize SSP(#) module for SPI communications.
<code>void OpenSWSPI(void)</code>	Configure the I/O pins of the software SPI.
<code>void OpenTimer# (unsigned char config)</code>	Configure and enable timer#.
<code>void OpenUART(void)</code>	Configure the I/O pins for the software USART.
<code>void Open#)USART (unsigned char config, unsigned int spbrg)</code>	Configure and enable the specified USART.

(Continued)

TABLE F.8 MICROCHIP C18 C COMPILER LIBRARY FUNCTIONS (CONTINUED)

FUNCTION PROTOTYPE	DESCRIPTION
<code>void OpenXLCD (unsigned char lcdtype)</code>	Configure PIC MCU I/O pins and initialize LCD controller.
<code>unsigned char putsI2C(#) (unsigned char * wrptr)</code>	Write a data string to the I2C(#) bus in either master or slave mode.
<code>void putsSPI(#) (unsigned char * wrptr)</code>	Write a string to the SPI(#) device.
<code>void putsUART(char * buffer)</code>	Write a string to the software USART.
<code>void puts(#)USART(char * data)</code>	Write ASCIIZ string of characters to the specified USART.
<code>void putrs(#)USART (const rom char * data)</code>	Write ASCIIZ string, defined in program memory, of characters to the specified USART
<code>void putsXLCD(char * buffer)</code>	Write a string to LCD controller.
<code>int rand(void)</code>	Return a pseudorandom integer.
<code>int ReadADC(void)</code>	Return the 16-bit result of the ADC operation.
<code>unsigned char ReadAddrXLCD(void)</code>	Return the address byte from the LCD controller.
<code>void ReadCapture#</code>	Read the result of a capture event from input capture #.
<code>void ReadECapture#</code>	Read the result of a capture event from enhanced input capture #.
<code>char ReadDataXLCD(void)</code>	Read a data byte from the LCD controller.
<code>unsigned char ReadI2C(#) (void)</code>	Read a single byte from the I2C(#) bus; this function also may be known as <code>getcI2C(#)</code> .
<code>unsigned char ReadSPI(#) (void)</code>	Read a single byte from the SPI(#) device; this function also may be known as <code>getcSPI(#)</code> .
<code>unsigned int ReadTimer# (void)</code>	Read the value of Timer#.
<code>char Read(#)USART(void)</code>	Read a byte from the specified USART; this function also may be known as <code>getc(#)USART</code> .
<code>char ReadUART(void)</code>	Read a byte from the software USART; this function also may be known as <code>getcUSART</code> .
<code>unsigned char ReadWire(#) (unsigned char high_byte, unsigned char low_byte)</code>	Read a single byte from the Microwire(#) device; this function also may be known as <code>getcMwire(#)</code> .
<code>void RestartI2C(#)(void)</code>	Generate restart condition for the I2C(#) bus.

TABLE F.8 MICROCHIP C18 C COMPILER LIBRARY FUNCTIONS (CONTINUED)

FUNCTION PROTOTYPE	DESCRIPTION
<code>void SetCGRamAddr (unsigned char addr)</code>	Specify the LCD character generator address.
<code>void SetChanADC (unsigned char channel)</code>	Select the channel to be used for the ADC operation.
<code>void SetCSSWSPI(void)</code>	Set the software SPI interface's chip select pin.
<code>void SetDDRamAddr (unsigned char addr)</code>	Specify the LCD display data area address.
<code>void SetOutputPWM# (unsigned char outputconfig, unsigned char outputmode)</code>	Set the PWM output configuration bits for ECCP.
<code>void SetOutputEPWM1 (unsigned char outputconfig, unsigned char outputmode)</code>	Set the enhanced PWM output configuration bits for ECCP.
<code>void srand(unsigned int seed)</code>	Specify the starting seed for the <code>rand</code> function.
<code>void StartI2C(#) (void)</code>	Generate a start condition for the I2C(#) bus.
<code>void StopI2C(#) (void)</code>	Generate stop operation for I2C(#) bus.
<code>char SWAckI2C(void)</code>	Generate an I2C bus acknowledge.
<code>char SWGetsI2C(unsigned char *rdptr, unsigned char length)</code>	Read a string from the I2C bus.
<code>char SWNotAckI2C(void)</code>	Generate an I2C bus not acknowledge.
<code>char SWPutsI2C(unsigned char *wrptr)</code>	Write a string to the I2C bus.
<code>char SWReadIC(void)</code>	Read a byte from the I2C bus; this function also may be known as <code>SWGetcI2C</code> .
<code>void SWRestartI2C(void)</code>	Generate an I2C bus restart condition.
<code>void SWStartI2C(void)</code>	Generate an I2C bus start condition.
<code>void SWStopI2C(void)</code>	Generates an I2C bus stop condition.
<code>char SWWriteI2C(unsigned char dataout)</code>	Write a byte to the I2C bus; this function also may be known as <code>SWPutcI2C</code> .
<code>char tolower(char c)</code>	Convert upper-case character to lower case or leave alone.
<code>char toupper(char c)</code>	Convert lower-case character to upper case or leave alone.

(Continued)

TABLE F.8 MICROCHIP C18 C COMPILER LIBRARY FUNCTIONS (CONTINUED)	
FUNCTION PROTOTYPE	DESCRIPTION
<code>void WriteCmdXLCD (unsigned char cmd)</code>	Write a command to the LCD controller.
<code>void WritedataXLCD (char data)</code>	Write a byte to the LCD data area.
<code>unsigned char WriteI2C(#) (unsigned char data_out)</code>	Write a single byte to the I2C(#) bus.
<code>unsigned char WriteMwire(#) (unsigned char data_out)</code>	Write a single byte to the Microwire(#) device.
<code>unsigned char WriteSPI(#) (unsigned char data_out)</code>	Write a single byte to the SPI(#) bus; this function also may be known as <code>putcSPI(#)</code> .
<code>char WriteSWSPI(char data)</code>	Write a byte to the software SPI; this function also may be known as <code>putcSWSPI</code> .
<code>void WriteTimer# (unsigned int tmr_value)</code>	Write a value to <code>timer#</code> .
<code>void WriteUART(char data)</code>	Write a byte to the software USART; this function also may be known as <code>putcUSART</code> .
<code>void Write(#)USART(char data)</code>	Write a byte to the specified USART; this function also may be known as <code>putc(#)</code> USART.

REUSE, RETURN, AND RECYCLE

The most efficient programmers I have ever met *never* write any new code. Instead, they have built up a library of code snippets that can be reused for different applications. These snippets are usually well known by the programmer in terms of operation, parameters, and peculiarities. In this appendix I have included a fair number of snippets and macros that I have found to be useful in developing my own PIC® microcontroller applications.

One of the biggest strengths of the PIC microcontroller is its ability to provide digital input/output (I/O) efficiently in a number of different ways. This flexibility can be used to implement I/O functions in the PIC microcontroller that weren't designed in originally. As part of this appendix, I present a number of simple I/O "bit banging" functions that provide commonly required functions for the PIC microcontroller that can be included into your code along with the other snippets provided.

The different features of the snippets can be modified for use in your applications with a bit of cut and pasting in an editor. You also should be able to find useful and clever code from other sources, including magazines, other books, and the Internet. This recycling of code is why I have named this appendix after the conservationist's "three R's." While I'm not a total "free software" advocate, I do believe that small pieces of useful code should be shared freely to help and encourage others. In this spirit, I ask that if you find or develop a useful function out of a few instructions, please share it with other people. If you don't know how to do this, you can share the code with a list-server (such as the PICList) community and let other people put it up on the Web for you. I'll always be happy to look at what you have come up with.

The most important piece of hardware in the PIC microcontroller used in applications is the I/O pins. When accessing the I/O pins, remember that there are three different ways of reading and writing them. The byte or port wide access method uses the `movf` and `movwf` instructions to read and write to the full I/O port. Along with these instructions, the arithmetic and bitwise instructions can be used with the port as the result's destination to simplify and optimize the operations.

The second method is to access the I/O port pins individually using the `bcf`, `bsf`, `btfsf`, and `btfs` instructions. These instructions work well, but when using `bcf`

and `bsf`, remember that they also can change the output value of a bit if the port bit is at an unexpected or unwanted value.

The last method is to use the rotate instructions `rlf` and `rrf` to pass data in and out of the I/O ports using the STATUS register's carry flag. Using this method means that either bit 0 or bit 7 of the I/O port will be accessed, but this method is very effective and efficient. To do this, the I/O situation has to be planned so that shifting data in and out does not affect the operation of the output values of the I/O port. This is not hard to do—the most obvious way of doing it is to set the rest of the bits on a port as input. With a bit of planning, shifting can be done on mixed input and output very effectively.

Useful Snippets

Here are a number of useful pieces of code that you can use in your applications. In some cases, I am the originator of the code; in others, when I know who the originator was, I have given him or her credit.

NEGATING THE CONTENTS OF A FILE REGISTER

Converting the contents of a file register to its twos complement value without affecting the w register is simply accomplished by

```
comf Reg, f      ; Invert the bits in the Register
incf Reg, f      ; Add One to them to turn into 2's
                  ; Complement
```

This code should not be used on any special hardware control registers.

NEGATING THE CONTENTS OF THE W REGISTER

If you have to negate the contents of the w register, you could use the preceding code (after saving the value in w into a register), or you could use for low-end devices (16C5x or 12C5xx)

```
addwf Reg, w     ; w = w + Reg
subwf Reg, w     ; w = Reg - w
                  ; w = Reg - ( w + Reg )
                  ; w = -w
```

Any file register can be used for this code because its contents are never changed.

In mid-range parts, the single instruction

```
sublw 0          ; w = 0 - w
```

could be used.

INCREMENTING/DECREMENTING THE W REGISTER

Here are a couple of snippets that will allow you to increment and decrement the w register without affecting any file registers if you don't have `addlw/sublw` instructions (i.e., in the case of low-end processors). `Reg` can be any register that does not change during execution of the three instructions. For low-end parts, any file register can be used because there is no danger of them being updated by an interrupt handler.

To increment:

```
xorlw 0x0FF      ; Get 1s Complement of Number
addwf Reg, w     ; w = Reg + (w^0x0FF)
subwf Reg, w     ; w = Reg + ((Reg + (w^0x0FF))^0x0FF) + 1
                ; w = w + 1
```

To decrement, the instructions are rearranged:

```
subwf Reg, w     ; w = Reg + (2^0x0FF) + 1
xorlw 0x0FF      ; Get 1s Complement of Result
addwf Reg, w     ; w = w - 1
```

ROTATING A BYTE IN PLACE

These two lines will rotate the contents of a file register in a low-end or mid-range PIC microcontroller without losing data in the carry flag. When working with the PIC18, the `rlncf` and `rrncf` instructions can be used for the same function. Rotates right and left can be implemented with this snippet. Note that the carry flag is changed.

```
rlf Register, w   ; Load Carry with the high bit
rlf Register, f   ; Shift over with high bit going low
```

COPY BITS FROM ONE REGISTER TO ANOTHER

Here is a fast way to save specific bits from one register into another:

```
movf Source, w
xorwf Destination, w
andlw B'xxxxxxxx' ; Replace "x" with "1" to Copy the Bit
xorwf Destination, f
```

CONVERTING A NYBBLE TO ASCII

This is a question that comes up all the time when the contents of a byte are to be displayed/output. The most obvious way of doing this is to use a table read:

NybbletoASCII:

```
addwf PCL, f     ; Add the Contents of the Nybble to PCL/
dt "0123456789ABCDEF" ; return the ASCII as a Table Offset
```

However, I think a much better way of doing this is

```
NybbletoASCII:      ; Convert a Nybble in "w" to ASCII
    addlw 0x036      ; Add '0' + 6 to Value
    btfsc STATUS, DC ; If Digit Carry Set, then 'A' - 'F'
    addlw 7          ; Add Difference Between '9' and 'A'
    addlw 0-6

    return           ; Return the ASCII of Digit in "w"
```

This method will take three instruction cycles longer than the previous code, but it requires 12 fewer instructions.

CONVERTING AN ASCII BYTE TO A HEX NYBBLE

The code below is really a rearrangement of the preceding snippet. Using the aspect that the high nybble of ASCII A to F is one greater than the high nybble of 0 to 9, a value is conditionally added to make the result 0x000 to 0x00F.

```
ASCIItoNybble:
    addlw 0x0C0      ; If "A" to "F", Set the Carry Flag
    btfss STATUS, C  ; If Carry Set, then 'A' - 'F'
    addlw 7          ; Add Difference Between '9' and 'A'
    addlw 9

    return           ; Return the ASCII of Digit in "w"
```

Note that ASCII characters other than 0 to 9 and A to F will result in an incorrect result.

USING TOCKI AS AN INTERRUPT SOURCE PIN

Some time ago, a question came up on the PICList asking if the timer input pin could be used as an interrupt source pin. The answer to this is yes—if the timer (and prescaler) is set up so that the next transition will increment the timer and cause an interrupt. Here's some code to do it in a mid-range PIC microcontroller:

```
movlw    B'11000000' ; First Setup with Instruction Clock
option   ; as TMR0 Source
movlw    B'11100000' ; Option Setup for TOCK1 TMR0 Source
clrf     TMR0        ; Set TMR0 to 0x0FF
decf     TMR0, f
option   ; Enable Timer on Outside Interrupt
          ; Edge
          ; NOTE - Executing this Instruction
          ; after "decf" will Load the
          ; Synchronizer with a "1"
```



```
bcf      STATUS, C
rrf      Dividend, f      ; Dividend /2 (ie "x/2" in Series)
movf     Dividend, w      ; Is it Equal to Zero?
btfsc    STATUS, Z
goto     Div3_Done        ; If it is, then Stop

addwf    Quotient         ; Add the Value to the Quotient

rrf      Dividend, f      ; Dividend /2 (ie "x/4" in Series)
movf     Dividend, w
btfsc    STATUS, Z
goto     Div3_Done

subwf    Quotient, f      ; Quotient = Quotient-(Dividend / 4)

goto     Div3_Loop

Div3_Done:

movf     Quotient, w      ; Return the Quotient

return
```

SIXTEEN-BIT COUNTER WITH A CONSTANT LOOP DELAY

When I first started working with the PIC microcontroller, I thought I was exceedingly clever when I came up with

```
movlw    HiDelay          ; Load the Delay Values
movwf    HiCount
movlw    LoDelay
movwf    LoCount
Dlay:    ; Loop Here Until HiCount/LoCount == 0
decfsz   LoCount, f
goto     Dlay
decfsz   HiCount, f
goto     Dlay
```

Then Marc Heuler showed the code

```
movlw    HiDelay          ; Load the Delay Values
movwf    HiCount
movlw    (LoDelay ^ 0xFF) + 1
Dlay:
addlw    1                ; Increment the Counter by 1
btfsc    STATUS, Z
decfsz   HiCount, f      ; Decrement the High Counter
goto     Dlay
```

This loop takes five cycles to execute regardless of whether or not `HiCount` is to be decremented (and uses one less file register than my method above). The actual time delay is calculated using the 16-bit number from

$$\text{Time Delay} = 16\text{BitDelay} * 5 \text{ ins/loop} * 4 \text{ clocks/ins} / \text{clock frequency}$$

This formula is a lot easier to use than having to (correctly) figure out the delay for the `Delay` loop code given above this. In the first edition of this book, it took three printings to get the first `Delay` loop's delay printed correctly.

The first method is useful if you require approximately a 200,000-instruction-cycle delay. To implement this, rather than loading `HiDelay` and `LoDelay` with constants, simply clear them before entering the two `decfsz` instruction loops.

SIXTEEN-BIT PULSE MEASUREMENT WITH FIVE-CYCLE DELAY

This is an improvement on the 16-bit delay code that I presented in the first edition of this book. `PulseWidth` is a 16-bit value that contains the number of times through the loop. The code that measures the pulse width for a high pulse is

```
clrf PulseWidth      ; Reset the Timer
clrf PulseWidth + 1

btfss PORTn, Bit     ; Wait for the Pulse to go high
goto $ - 1

incfsz PulseWidth, f ; Increment the Counter
decf PulseWidth + 1, f
btfsc PORTn, Bit     ; Loop while Still High
goto $ - 3

movf PulseWidth, w    ; Make 16 Bit Result Valid
addwf PulseWidth + 1, f
```

DETECT A CHANGE IN A REGISTER

Bob Fehrenbach has passed on a number of his snippets for inclusion on my Web page, and they have shown up in this book. The first detects the change in a bit and saves the new data for later execution. This code can be used to detect changes in the I/O ports, timers, and other registers that can be updated externally to the software execution.

```
movf Reg, w
andlw Mask           ; Mask out unused bits
xorwf old, w         ; Compare to previous value
btfsc STATUS, Z      ; If Zero set, bits are the Same
goto no_change
xorwf old             ; Bits are different, Store New
                    ; pattern in "old"
```

TEST A BYTE WITHIN A RANGE

This is an algorithm that I continually reinvent (although I don't think I've ever come up with anything as efficient as Bob Fehrenbach's routine):

```
movf Num, w
addlw 255 - hi_lim      ; "Num" is equal to -hi_lim
addlw hi_lim - lo_lim + 1 ; "Num" is > 255 if it is above
btfsc STATUS, C         ; the lo-lim
goto in_range
```

SWAP THE CONTENTS OF W WITH A REGISTER

I know that this one has been around forever, but it never hurts to be reminded of it and to have it written down for easy access.

```
xorwf Reg, f      ; w = w, Reg = Reg ^ w
xorwf Reg, w      ; w = w ^ (Reg ^ w), Reg = Reg ^ w
                  ; w = Reg, Reg = Reg ^ w
xorwf Reg, f      ; w = Reg, Reg = Reg ^ w ^ Reg
                  ; w = Reg, Reg = w
```

This algorithm for swapping values without a temporary value code can be implemented in C as

```
VariableA = VariableA ^ VariableB; // A = A ^ B
VariableB = VariableB ^ VariableA; // B = B ^ A
                                     // = B ^ (A ^ B)
                                     // = A
VariableA = VariableA ^ VariableB; // A = (A ^ B) ^ A
                                     // = B
```

SWAP THE CONTENTS OF TWO REGISTERS

Using the algorithm from the preceding point, here's a fast snippet to swap the contents of two file registers:

```
movf X, w
subwf Y, w      ; W = Y - X
addwf X, f      ; X = X + (Y - X)
subwf Y, f      ; Y = Y - (Y - X)
```

COMPARE AND SWAP IF Y < X

Here's a compare and swap (uses the swap presented earlier):

```

movf X, w
subwf Y, w      ; Is Y >= X?
btfsc STATUS, C ; If Carry Set, Yes
goto $ + 2      ; Don't Swap
addwf X, f      ; Else, X = X + (Y - X)
subwf Y, f      ; Y = Y - (Y - X)

```

CONVERT ASCII TO UPPER CASE

This is a practical application of the preceding snippet. I think that this subroutine demonstrates how the code works quite well.

ToUpper:

```

addlw 255 - 'z'      ; Get the High limit
addlw 'z' - 'a' + 1 ; Add Lower Limit to Set Carry
btfss STATUS, C      ; If Carry Set, then Lower Case
addlw h'20'          ; Carry NOT Set, Restore Character
addlw 'A'             ; Add 'A' to restore the Character
return

```

COUNTING THE NUMBER OF 1S IN A BYTE

If you're a regular on the PICList, you will know of Dmitry Kiryashov and his interest in providing the most efficient routines possible for carrying out operations. The code below is his optimization of the classic problem of counting the number of 1s in a byte in 12 instructions/12 cycles.

```

; (c) 1998 by Dmitry Kirashov

rrf X, w      ; "X" Contains Byte
andlw 0x55    ; -a-c-e-g
subwf X, f    ; ABCDEFGH
              ; where AB=a+b, etc.
              ; the same trick as in example_1

movwf X
andlw 0x33    ; --CD--GH
addwf X, f
rrf X, f      ; 0AB00EF0
              ; 00CD00GH
addwf X, f    ; 0AB00EF0
              ; 0CD00GH0
rrf X, f      ; 0ABCD.0EFGH

swapf X, w
addwf X, w
andlw 0x0F    ; Bit Count in "w"

```

GENERATING PARITY FOR A BYTE

The six instructions below (provided by John Payson) will calculate the even parity for a byte. At the end of the routine, bit 0 of X will have the even parity bit of the original number. Even parity means that if all the 1s in the byte are summed along with the parity bit, an even number will be produced.

```
swapf X, w
xorwf X, f

rrf X, w
xorwf X, f

btfsc X, 2
incf X, f
```

KEEPING A VARIABLE WITHIN A RANGE

Sometimes when handling data you will have to keep integers within a range. The four instructions below will make sure that the variable Temp always will be in the range of 0 to Constant.

```
movlw Constant      ; 0 <= Temp <= Constant
subwf Temp, w
btfsc STATUS, C
subwf Temp, f
```

SWAPPING BIT PAIRS

Another of Dmitry Kiryashov's routines is this sequence of instructions for swapping bit pairs in a byte in five instructions/cycles.

```
                                ; (c) 1998 by Dmitry Kirashov

movwf X                        ; Save the Incoming Byte in
                                ; a temporary register
                                ; w = X = ABCDEFGH
andlw 0x055                    ; w = 0B0D0F0H
addwf X, f                     ; X = ABCDEFGH + 0B0D0F0H

rrf X, f                       ; X = (ABCDEFGH + 0B0D0F0h) >> 1
addwf X, w                     ; w = BADCFEHG
```

BITWISE OPERATIONS

Setting and resetting bits based on the state of other bits is not something the PIC micro-controller seems to be able to do well naturally. By using multiple bit condition tests,

the actual operations are pretty easy to implement. Note that these routines should not be used for changing I/O port values because there may be an incorrect intermediate value. If you are using this code for changing an I/O port or hardware control register bit, make sure that you read the register's contents into the w register and use the `andlw` and `iorlw` instructions to change the bit before writing the new value back to the register. The intermediate value could initiate some hardware operation that is not desired.

Setting a bit by ANDing two others together is accomplished by

```
bsf Result      ; Assume the result is True
btfsc BitA      ; If BitA != 1 then result is False
btfss BitB      ; If BitB == 0 then result is False
bcf Result      ; Result is False, Reset the Bit
```

To show how this operation could be accomplished on an I/O port bit, I have included the code

```
movf PORTB, w    ; Store PORTB in "w" for "AND" Op'n
iorlw 1 << Result ; Assume the Result is True
btfsc BitA      ; If BitA != 1 then result is False
btfss BitB      ; If BitB == 0 then result is False
andlw 0x0FF ^ (1 << Result) ; Result is False, Reset the Bit
movwf PORTB     ; Save the Result
```

ORing two bits together is similar to the AND operation, except that the result is expected to be false, and when either bit is set, the result is true.

```
bcf Result      ; Assume the result is False
btfss BitA      ; If BitA != 0 then result is True
btfsc BitB      ; If BitB == 0 then result is False
bsf Result      ; Result is True, Set the Bit
```

The final operation is the NOT. There are two ways of implementing this operation based on where the input value is relative to the output value. If they are the same (i.e., the operation is to complement a specific bit), the code to be used is simply

```
movlw 1 << BitNumber ; Complement Specific Bit for "NOT"
xorwf BitRegister, f
```

If the bit is in another register, then the value stored is the complement of it:

```
bcf Result      ; Assume that the Input Bit is Set
btfss Bit       ; - If it is Set, then Result Correct
bsf Result      ; Input Bit Reset, Set the Result
```

Mykemacs.inc

I want to introduce you some basic programming and I/O functions for low-end and mid-range PIC microcontrollers. When I first started writing this, I was trying to figure out how to best present them so that they could be used easily without you having to figure out how to modify code that is cut and pasted into your application. The solution I came up with is to provide macros for these functions.

The macros listed below are examples of quite advanced macros. Along with providing different methods of interfacing, I also use the expected PIC microcontroller's clock speed to calculate delays within the macros. For the most part, they can be used without modification, but you should read through the accompanying text to make sure that you are aware of any issues with the macros on different devices.

When this code was written, it was designed for low-end and mid-range PIC microcontrollers. You will find that for initializing interfaces, I have used the mid-range TRIS registers instead of the common `tris` instructions. This was done because of the ease with which specific pins can access specific `tris` bits using the `bsf` and `bcf` instructions. For low-end PIC microcontrollers, you will have to come up with your own TRIS register values based on the I/O pins you are going to access.

All these macros, including the structured programming macros I presented earlier in this book, are bundled up together in what I have immodestly called *mykemacs.inc*. This file is located in the C:\PICDwnLd\Macros\mykemacs subdirectory of your PC's PIC Microcontroller directory. To use *mykemacs.inc* in your applications, copy the file into your PC's C:\Program Files\Microchip\MPASM Suite folder.

When *mykemacs.inc* is to be used in your application, include the file using the statement

```
include "mykemacs.inc"
```

When you want to use a function, you can select it from the invocation instructions given in the following sections.

SIMPLE DELAY

As I look through this book, I realize that I have not given you a good, simple delay that will work in the generic case. The important point of the preceding sentence is the term *generic case*—often a delay is needed that does not affect any other registers or the PIC microcontroller's STATUS bits. The `DlayMacro` that I have provided below will allow you to delay any number of cycles (up to a limit of 777 cycles) and provides the calculations to a specific instruction cycle. The macro is

```
DlayMacro Macro Cycles      ; Delay Macro for Edges
    variable i, TCycles, Value, TFlag
    TCycles = Cycles
    Value = 1 << 7
```



```

i = 7
TFlag = 0
  if (TCycles > 5)
  while (i >= 0)
    if ((TFlag == 0) && ((Value * 3) <= TCycles))
      bsf DlayCount, i
    TFlag = 1
    TCycles = TCycles - (Value * 3)
  else
    if ((TFlag != 0) && (((Value * 3) + 1) <= TCycles))
      bsf DlayCount, i
    TCycles = TCycles - ((Value * 3) + 1)
    endif
  endif
Value = Value >> 1
i = i - 1
endw
  if (TCycles > 3)
    Error "Delay Cycles too Large for Macro"
  endif
  decfsz DlayCount, f
  goto $ - 1
endif
while (TCycles > 1)
  goto $ + 1
TCycles = TCycles - 2
endw
  if (TCycles == 1)
    nop                ; Delay the Last Cycle
  endif
endm

```

This macro may seem quite complex, but the actual code it produces is very simple. The macro will work on low-end, mid-range, and PIC18 PIC microcontroller architectures.

The basis of the delay code is the three-instruction-cycle loop:

```

decfsz DlayCount, f
goto $ - 1

```

Each time this loop executes, three cycles are taken up. To set the initial `DlayCount`' value, the number of cycles to delay is compared against the maximum value possible for a bit value in `DlayCount`. For example, setting bit 2 of `DlayCount` will cause the loop code to execute four times. If bit 6 is used, `DlayCount` will loop 64 times. Instead of dividing the total number by 3 (which is the number of loops required) and then loading the `w` register with the value (which changes the `w` register) and then writing to the `DlayCount` variable, I set the appropriate bit for the count in `DlayCount`.

This is a bit hard to understand (and pretty hard to see in the macro code above). To try to make the operation more obvious, I want to show some pseudocode that better illustrates how the delay value is calculated. The output of the code is assumed to be the `DlayCount` variable with the appropriate bits set.

```
TotalDelay = RequestedDelay;
DlayCount = 0;           // Final Value to Delay
Value = 1 << 7;          // Test Value
TFlag = 0;
for (i = 0; i < 8; i++ ) { // Repeat for all 8 Bits
    if ((TFlag == 0) && (TotalDelay > (Value * 3))) {
        DlayCount = DlayCount | Value; // Mark the Correct Bit Value
        TotalDelay = TotalDelay - (Value * 3);
        TFlag = 1;                     // Mark that Value Changes
    } else if ((TFlag != 0) && (TotalDelay > ((Value * 3) + 1))) {
        DlayCount = DlayCount | Value; // Mark the Correct Bit Value
        TotalDelay = TotalDelay - ((Value * 3) + 1);
    }
    Value = Value >> 1;                // Take down the Division Value
}
```

The reason why one is added to the delay after the first time through is because of the way the code is produced. The first time a `DlayCount` bit is set, the code becomes

```
bsf DlayCount, Bit
decfsz DlayCount, f
goto $ - 1
```

and the delay is simply calculated as 2 to the power of `Bit` multiplied by 3. Thus, for bit 4 being set, the delay would be

```
Delay = (2 ** Bit) * 3
       = (1 << Bit) * 3
       = (1 << 4) * 3
       = 16 * 3
       = 48
```

When another bit is set, the code becomes

```
bsf DlayCount, Bit1
bsf DlayCount, Bit2
decfsz DlayCount, f
goto $ - 1
```

In this case, the delay is 3 times the final value in `DlayCount` plus 1. As a formula, the instruction delay is

```
DlayMacroLoopDelay = (DlayCount * 3) + (#DlayCount Bits Set - 1)
```

This probably seems a bit cumbersome to work with, but the macro takes care of the calculation for you.

After the loop has finished, the code will check for any left-over instruction cycles and insert `goto $ + 1` or `nop` instructions as required so that the `DlayMacro` executes for exactly the specified number of instruction cycles. When using this macro, you will have to declare `DlayCount` and make sure that it is accessible from within whatever register banks are active when the macro is invoked and that it is cleared before its first use. On exit from the `DlayMacro` code, `DlayCount` will always be zero. To ensure that `DlayCount` is always zero, never change its value.

You might be surprised to see that there is an initial check for the `DlayMacro` loop so that it only executes if the number of cycles to delay is greater than five. The reason for this is that a combination of `goto $ + 1` and `nop` instructions can be implemented more efficiently than the delay loop. The macro will produce the most efficient code that it can.

When you look at this code, you might be wondering just how efficient the code is. By explicitly testing the value for a specific bit value and putting it into the delay, you might feel like I'm not being that efficient, especially in the case of making the loop

```
movlw InitialValue      ; Delay = InitialValue * 3 + 1
movwf DlayCount
decfsz DlayCount, f
goto $ - 1
goto $ + 1              ; Optional
nop                     ; Optional
```

which has a worst-case size of six instructions (the `DlayMacro` has a worst-case size of 12 instructions). The only comment I can make to this is `DlayMacro` does not change the `w` register, and you will find that for many cases `DlayMacro` will end up producing a comparable number of instructions.

For example, invoking the macro

```
DlayMacro 100
```

will produce the code

```
bsf DlayCount, 5
bsf DlayCount, 0
decfsz DlayCount, f
goto $ - 1
```

which is comparable in size with what the other code will do but does not change the `w` register. This feature of not changing the `w` register will be useful in the following example code. In the worst case, the `DlayMacro` will produce 12 instructions (to the other method's six). This happens in only one instance, and you will find that for most instruction cycle delays this macro produces code that is within the four to six instructions of the other case.

LCD INTERFACES

To simplify the work required for adding LCDs to your applications, I wanted to create a series of four macros that would allow you to interface with LCDs in either low-end or mid-range PIC microcontrollers using one of four subroutines. By using a consistent interface, the same software can be used for a variety of different situations without requiring that the application code be changed—the vision was that just a new macro would need to be invoked to access the LCD.

The four LCD subroutines are

LCDPORTInit—Used in mid-range devices to define the I/O ports that interface to the LCD

LCDInit—Put the LCD into 2 line mode

LCDChar—Send an ASCII character to the LCD

LCDIns—Send an LCD instruction

Along with these subroutines, each macro creates a **Dlay5** subroutine that delays execution for 5 ms for LCD reset operations. In each LCD access type macro, I create a 160-μs delay code that is inserted in line and based on the **DlayMacro** and does not change the value of any registers.

For the most part, the interfaces are direct to the LCDs, but there are some points that you should be aware of that I point out in the text below.

The first macro is **LCD8**, which provides a basic interface to the LCD with worst-case startup delays. To invoke it, the statement

LCD8 DataPort, EPort, EPin, RSPort, RSPin, RWPort, RWPIn, Frequency

is put in where **DataPort** is the 8-bit I/O port. **EPort** and **EPin** are the E clock definition. **RSPort** and **RSPin** are the RS LCD data type input. **RWPort** and **RWPIn** are the pins used to poll the LCD for data reply (and are essentially unused). **Frequency** is the PIC microcontroller operating speed and is used to calculate the delay values. The only variable required for the **LCD8** and **LCD8Poll** macros is the 8-bit variable **Dlay**.

This macro should work with any low-end or mid-range PIC microcontroller. Note that the **LCDPORTInit** subroutine cannot be used with low-end PIC microcontrollers; to set up the I/O ports, you will have to create your own **tris** statements. The macro is

```
LCD8 Macro DataPort, EPort, EPin, RSPort, RSPin, RWPort, RWPIn,
Freq
    variable Dlay5Value, Dlay160Value, Dlay160Bit1 = -1, Dlay160Bit2 = -1
    variable BitCount = 0
    variable Value = 128, Bit = 7
    Dlay5Value = ((5007 * (Freq / 1000) / 4000) / 7) + 256
```

```

Dlay160Value = (163 * (Freq / 1000) / 4000) / 3

while (Bit >= 0)                ; Find the Number of Bits and their
                                ; positions in "Dlay160Value"
if ((Dlay160Value & Value) != 0)
if (Dlay160Bit1 == -1)          ; Set the Upper Bit
Dlay160Bit1 = Bit
else
if (Dlay160Bit2 == -1)
Dlay160Bit2 = Bit
endif
endif
BitCount = BitCount + 1
endif
Value = Value >> 1
Bit = Bit - 1
endw
if (BitCount > 2)                ; Just Want max two Bits
if ((Dlay160Bit1 - 1) == Dlay160Bit2)
Dlay160Bit1 = Dlay160Bit1 + 1 ; Shift Top up by 1
Dlay160Bit2 = -1                ; Delete Second
else
Dlay160Bit2 = Dlay160Bit2 + 1 ; Shift Bottom up by 1
endif
endif

Dlay5                          ; Delay 5 msecs
movlw (Dlay5Value & 0xFF00) >> 8
movwf Dlay
movlw Dlay5Value & 0xFF
subwf Dlay, w
xorlw 0xFF
addwf Dlay, w
btfsc STATUS, Z
decfsz Dlay, f
goto $ - 5
return

LCDPORTInit                    ; Initialize the I/O Ports
bsf STATUS, RP0                ; ONLY used by mid-range
movlw 0x000
movwf DataPort
bcf EPort, EPin
bcf RSPort, RSPin
bcf RWPort, RWPIn
bcf STATUS, RP0
bcf EPort, EPin
bcf RSPort, RSPin

```

```
    bcf  RWPort, RWPIn
    return

LCDIns                                ; Send the Instruction to the LCD
    movwf DataPort
    bcf  RSPort, RSPin
    if (Freq > 8000000)                ; Make Sure Proper Delay is In Place
    if (Freq < 16000000)
        nop
    else
        goto $ + 1
    endif
    endif
    bsf  EPort, EPin
    if (Freq > 8000000)                ; Make Sure Proper Delay is In Place
    if (Freq < 16000000)
        nop
    else
        goto $ + 1
    endif
    endif
    bcf  EPort, EPin
    bsf  Dlay, Dlay160Bit1            ; Delay 160 µsecs
    if (Dlay160Bit2 != -1)
    bsf  Dlay, Dlay160Bit2
    endif
    decfsz Dlay, f
    goto $ - 1
    andlw 0x0FC                      ; Have to Delay 5 msecs?
    btfsc STATUS, Z
    call Dlay5
    return

LCDChar                                ; Send the Character to the LCD
    movwf DataPort
    bsf  RSPort, RSPin
    if (Freq > 8000000)                ; Make Sure Proper Delay is In Place
    if (Freq < 16000000)
        nop
    else
        goto $ + 1
    endif
    endif
    bsf  EPort, EPin
    if (Freq > 8000000)                ; Make Sure Proper Delay is In Place
    if (Freq < 16000000)
        nop
```

```

else
goto $ + 1
endif
endif
bcf EPort, EPin
bsf Dlay, Dlay160Bit1           ; Delay 160 usecs
if (Dlay160Bit2 != -1)
bsf Dlay, Dlay160Bit2
endif
decfsz Dlay, f
goto $ - 1
return

LCDInit                         ; Do the 8 Bit Initialization
call Dlay5                     ; Wait 15 msecs
call Dlay5
call Dlay5
movlw 0x030
call LCDIns                    ; Send the Reset Instruction
call Dlay5
movlw 0x030
call LCDIns
movlw 0x030
call LCDIns
movlw 0x038                    ; Set Interface Length
call LCDIns
movlw 0x010                    ; Turn Off Display
call LCDIns
movlw 0x001                    ; Clear Display RAM
call LCDIns
movlw 0x006                    ; Set Cursor Movement
call LCDIns
movlw 0x00E                    ; Turn on Display/Cursor
call LCDIns
return
endm

```

Looking at the `Dlay160` code, you'll notice that I restricted the number of possible bit settings to two. If more than two are set, then I first increment the lower bit value. If the lower bit value is immediately below the upper one, then I delete the lower one and increment the upper value. This ensures that I do not delay for less than the 160 μ s required by the LCD. The reason why I held the delay down to two cycles is to keep the delay code as simple as possible and avoid the need for invoking `DlayMacro`.

The `LCD8Poll` macro produces slightly more sophisticated code than that produced by the `LCD8` macro. Instead of providing hard-coded delays in the application, the code polls the LCD to see if the operation is complete before continuing. This is done by

putting the DataPort into input mode and then strobing the E bit (with RS reset and RW set) and looking at bit 7 of the I/O port. The macro code is

```
LCD8Poll Macro DataPort, EPort, EPin, RSPort, RSPin, RWPort, RWPin,
Freq
    variable Dlay5Value, Dlay160Value, Dlay160Bit1 = -1, Dlay160Bit2
= -1
    variable BitCount = 0
    variable Value = 128, Bit = 7
    errorlevel 0,-224
Dlay5Value = ((5007 * (Freq / 1000) / 4000) / 7) + 256
Dlay160Value = (163 * (Freq / 1000) / 4000) / 3

    while (Bit >= 0)                ; Find the Number of Bits and their
                                    ; Positions in "Dlay160Value"
    if ((Dlay160Value & Value) != 0)
    if (Dlay160Bit1 == -1)          ; Set the Upper Bit
Dlay160Bit1 = Bit
    else
    if (Dlay160Bit2 == -1)
Dlay160Bit2 = Bit
    endif
    endif
    BitCount = BitCount + 1
    endif
    Value = Value >> 1
    Bit = Bit - 1
    endw
    if (BitCount > 2)                ; Just Want max two Bits
    if ((Dlay160Bit1 - 1) == Dlay160Bit2)
Dlay160Bit1 = Dlay160Bit1 + 1        ; Shift Top up by 1
Dlay160Bit2 = -1                    ; Delete Second
    else
Dlay160Bit2 = Dlay160Bit2 + 1        ; Shift Bottom up by 1
    endif
    endif

Dlay5                                ; Delay 5 msecs
    movlw (Dlay5Value & 0xFF00) >> 8
    movwf Dlay
    movlw Dlay5Value & 0xFF
    subwf Dlay, w
    xorlw 0xFF
    addwf Dlay, w
    btfsc STATUS, Z
    decfsz Dlay, f
    goto $ - 5
    return
```



```

LCDPORTInit                                ; Initialize the I/O Ports
    bsf STATUS, RP0                        ; ONLY used by mid-range
    movlw 0x000
    movwf DataPort
    bcf EPort, EPin
    bcf RSPort, RSPin
    bcf RWPort, RWPin
    bcf STATUS, RP0
    bcf EPort, EPin
    bcf RSPort, RSPin
    bcf RWPort, RWPin
    return

LCDIns                                      ; Send the Instruction to the LCD
    movwf Dlay
    movlw 0xFF                             ; Read the "BF" Flag
    tris DataPort
    bcf RSPort, RSPin                      ; Read the Instruction Register
    bsf RWPort, RWPin
    goto $ + 1
    bsf EPort, EPin
    nop
    movf DataPort, w                       ; Read the Data Port Value
    nop
    bcf EPort, EPin
    andlw 0x080                            ; Is the High Bit Set?
    btfss STATUS, Z
    goto $ - 7
    bcf RWPort, RWPin
    movlw 0                                ; Put the DataPort Back into Output Mode
    tris DataPort
    movf Dlay, w                           ; Get the Saved Character
    movwf DataPort
    if (Freq > 8000000)                     ; Make Sure Proper Delay is In Place
    if (Freq < 16000000)
    nop
    else
    goto $ + 1
    endif
    endif
    bsf EPort, EPin
    if (Freq > 8000000)                     ; Make Sure Proper Delay is In Place
    if (Freq < 16000000)
    nop
    else
    goto $ + 1
    endif
    endif

```

```
    bcf EPort, EPin
    return

LCDChar                                ; Send the Character to the LCD
    movwf Dlay
    movlw 0x0FF                        ; Read the "BF" Flag
    tris DataPort
    bcf RSPort, RSPin                  ; Read the Instruction Register
    bsf RWPort, RWPin
    goto $ + 1
    bsf EPort, EPin
    nop
    movf DataPort, w                  ; Read the Data Port Value
    nop
    bcf EPort, EPin
    andlw 0x080                        ; Is the High Bit Set?
    btfss STATUS, Z
    goto $ - 7
    bsf RSPort, RSPin
    bcf RWPort, RWPin
    movlw 0                            ; Put the DataPort Back into Output Mode
    tris DataPort
    movf Dlay, w                      ; Get the Saved Character
    movwf DataPort
    if (Freq > 8000000)                ; Make Sure Proper Delay is In Place
    if (Freq < 16000000)
    nop
    else
    goto $ + 1
    endif
    endif
    bsf EPort, EPin
    if (Freq > 8000000)                ; Make Sure Proper Delay is In Place
    if (Freq < 16000000)
    nop
    else
    goto $ + 1
    endif
    endif
    bcf EPort, EPin
    return

LCDInit                                ; Do the 8 Bit Initialization
    call Dlay5                        ; Wait 15 msecs
    call Dlay5
    call Dlay5
    movlw 0x030
    movwf DataPort
```

```

if (Freq > 8000000)          ; Make Sure Proper Delay is In Place
if (Freq < 16000000)
nop
else
goto $ + 1
endif
endif
bsf EPort, EPin
if (Freq > 8000000)          ; Make Sure Proper Delay is In Place
if (Freq < 16000000)
nop
else
goto $ + 1
endif
endif
bcf EPort, EPin             ; Send the Reset Instruction
call Dlay5
if (Freq > 8000000)          ; Make Sure Proper Delay is In Place
if (Freq < 16000000)
nop
else
goto $ + 1
endif
endif
bsf EPort, EPin
if (Freq > 8000000)          ; Make Sure Proper Delay is In Place
if (Freq < 16000000)
nop
else
goto $ + 1
endif
endif
bcf EPort, EPin             ; Send the Reset Instruction
bsf Dlay, Dlay160Bit1       ; Delay 160 usecs
if (Dlay160Bit2 != -1)
bsf Dlay, Dlay160Bit2
endif
decfsz Dlay, f
goto $ - 1
movlw 0x030
call LCDIns
movlw 0x038                  ; Set Interface Length
call LCDIns
movlw 0x010                  ; Turn Off Display
call LCDIns
movlw 0x001                  ; Clear Display RAM
call LCDIns
movlw 0x006                  ; Set Cursor Movement

```

```
call LCDIns
movlw 0x00E           ; Turn on Display/Cursor
call LCDIns
return
errorlevel 0,+224      ; Enable "TRIS" Indicators
endm
```

It is important to note that the two LCD routines work identically and require just the Delay variable. To demonstrate this, the test code I used for both macros was identical, with just the macro invocation changed:

```
title "LCD8Poll - Test out the 8 Bit LCD Interface"
;
; This Code Sends the data to the LCD in 8 Bit mode with
; Polling and sends an ASCII "A" to the LCD.
;
;
; Hardware Notes:
; The PIC is a 16C84 Running at 4 MHz.
; Reset is tied directly to Vcc and PWRT is Enabled.
; PortB is the Data Port
; RA0 is the "E" Bit
#define E    PORTA, 0
; RA1 is the "RS" Bit
#define RS   PORTA, 1
; RA2 is the "RW" Bit
#define RW   PORTA, 2
;
;
LIST R=DEC
INCLUDE "p16f84.inc"
include "mykemacs.inc"

; Register Usage
CBLOCK 0x00C           ; Start Registers at End of the Values
Delay                  ; 8 Bit Delay Variable
ENDC

PAGE
__CONFIG __CP_OFF & __XT_OSC & __PWRTE_ON & __WDT_OFF
; Note that the WatchDog Timer is OFF
; Demo Code, Loop Forever Toggling PA0 (Flashing the LED)

org 0

call LCDPORTInit
```

```

call LCDInit          ; Initialize the LCD Port

movlw "A"             ; Put the "A" on the LCD
call LCDChar

goto $

```

```

LCD8Poll PORTB, E, RS, RW, 4000000      ; This can also be "LCD8"

```

```

end

```

For 4-bit LCD interfacing (the LCD4 macro), the LCD8 macro was used and modified to allow the data transfer. To invoke the macro, the similar statement

```

LCD4 DataPort, DataBit, EPort, EPin, RSPort, RSPin, RWPort, RWPIn,
Freq

```

is used. The DataBit parameter is the lowest of the four data bits. It can only be 0 or 4. The macro requires the LCDTemp variable along with Dlay. The macro is

```

LCD4 Macro DataPort, DataBit, EPort, EPin, RSPort, RSPin, RWPort,
      RWPIn, Freq
      variable Dlay5Value, Dlay160Value, Dlay160Bit1 = -1, Dlay160Bit2
= -1
      variable BitCount = 0
      variable Value = 128, Bit = 7
Dlay5Value = ((5007 * (Freq / 1000) / 4000) / 7) + 256
Dlay160Value = (163 * (Freq / 1000) / 4000) / 3

      if ((DataBit != 0) && (DataBit != 4))
error "Invalid 'DataBit' Specification - Can only be '0' or '4'"
endif

      while (Bit >= 0)                                ; Find the Number of Bits and their
                                                         ; Positions in "Dlay160Value"
      if ((Dlay160Value & Value) != 0)
      if (Dlay160Bit1 == -1)                            ; Set the Upper Bit
Dlay160Bit1 = Bit
      else
      if (Dlay160Bit2 == -1)
Dlay160Bit2 = Bit
      endif
      endif
      BitCount = BitCount + 1
      endif

```

```
Value = Value >> 1
Bit = Bit - 1
    endw
    if (BitCount > 2)                ; Just Want max two Bits
    if ((Dlay160Bit1 - 1) == Dlay160Bit2)
Dlay160Bit1 = Dlay160Bit1 + 1        ; Shift Top up by 1
Dlay160Bit2 = -1                    ; Delete Second
    else
Dlay160Bit2 = Dlay160Bit2 + 1        ; Shift Bottom up by 1
    endif
    endif

Dlay5                                ; Delay 5 msecs
    movlw (Dlay5Value & 0xFF00) >> 8
    movwf Dlay
    movlw Dlay5Value & 0xFF
    subwf Dlay, w
    xorlw 0xFF
    addwf Dlay, w
    btfsc STATUS, Z
    decfsz Dlay, f
    goto $ - 5
    return

LCDPORTInit                          ; Initialize the I/O Ports
    bsf STATUS, RP0                  ; ONLY used by mid-range
    if (DataBit == 0)
    movlw 0xF0
    else
    movlw 0x0F
    endif
    movwf DataPort
    bcf EPort, EPin
    bcf RSPort, RSPin
    bcf RWPort, RWPIn
    bcf STATUS, RP0
    bcf EPort, EPin
    bcf RSPort, RSPin
    bcf RWPort, RWPIn
    return

LCDIns                                ; Send the Instruction to the LCD
    movwf LCDTemp                    ; Save the Value
    if (DataBit == 0)
    swapf LCDTemp, w                  ; Most Significant Nybble First
    andlw 0x0F
    else
```

```

andlw 0x0F0
endif
movwf DataPort
bcf  RSPort, RSPin
if (Freq > 8000000)           ; Make Sure Proper Delay is In Place
if (Freq < 16000000)
nop
else
goto $ + 1
endif
endif
bsf  EPort, EPin
if (Freq > 8000000)           ; Make Sure Proper Delay is In Place
if (Freq < 16000000)
nop
else
goto $ + 1
endif
endif
bcf  EPort, EPin
if (DataBit == 0)
movf LCDTemp, w
andlw 0x00F
else
swapf LCDTemp, w             ; Least Significant Nybble Second
andlw 0x0F0
endif
movwf DataPort
bcf  RSPort, RSPin
if (Freq > 8000000)           ; Make Sure Proper Delay is In Place
if (Freq < 16000000)
nop
else
goto $ + 1
endif
endif
bsf  EPort, EPin
if (Freq > 8000000)           ; Make Sure Proper Delay is In Place
if (Freq < 16000000)
nop
else
goto $ + 1
endif
endif
bcf  EPort, EPin
bsf  Dlay, Dlay160Bit1       ; Delay 160 usecs
if (Dlay160Bit2 != -1)

```

```
    bsf Dlay, Dlay160Bit2
    endif
    decfsz Dlay, f
    goto $ - 1
    movf LCDTemp, w
    andlw 0x0FC                ; Have to Delay 5 msecs?
    btfsc STATUS, Z
    call Dlay5
    return

LCDChar                ; Send the Character to the LCD
    movwf LCDTemp        ; Save the Value
    if (DataBit == 0)
        swapf LCDTemp, w    ; Most Significant Nybble First
        andlw 0x00F
    else
        andlw 0x0F0
    endif
    movwf DataPort
    bsf RSPort, RSPin
    if (Freq > 8000000)      ; Make Sure Proper Delay is In Place
    if (Freq < 16000000)
        nop
    else
        goto $ + 1
    endif
    endif
    bsf EPort, EPin
    if (Freq > 8000000)      ; Make Sure Proper Delay is In Place
    if (Freq < 16000000)
        nop
    else
        goto $ + 1
    endif
    endif
    bcf EPort, EPin
    if (DataBit == 0)
        movf LCDTemp, w
        andlw 0x00F
    else
        swapf LCDTemp, w    ; Least Significant Nybble Second
        andlw 0x0F0
    endif
    movwf DataPort
    bsf RSPort, RSPin
    if (Freq > 8000000)      ; Make Sure Proper Delay is In Place
    if (Freq < 16000000)
        nop
```



```

else
goto $ + 1
endif
endif
bsf EPort, EPin
if (Freq > 8000000) ; Make Sure Proper Delay is In Place
if (Freq < 16000000)
nop
else
goto $ + 1
endif
endif
bcf EPort, EPin
bsf Dlay, Dlay160Bit1 ; Delay 160 usecs
if (Dlay160Bit2 != -1)
bsf Dlay, Dlay160Bit2
endif
decfsz Dlay, f
goto $ - 1
return

LCDInit ; Do the 8 Bit Initialization
call Dlay5 ; Wait 15 msecs
call Dlay5
call Dlay5
if (DataBit == 0) ; Send the Reset Instruction
movlw 0x003
else
movlw 0x030
endif
movwf DataPort
if (Freq > 8000000) ; Make Sure Proper Delay is In Place
if (Freq < 16000000)
nop
else
goto $ + 1
endif
endif
bsf EPort, EPin
if (Freq > 8000000) ; Make Sure Proper Delay is In Place
if (Freq < 16000000)
nop
else
goto $ + 1
endif
endif
bcf EPort, EPin
call Dlay5

```

```
bsf EPort, EPin                ; Send Another Reset Instruction
if (Freq > 8000000)             ; Make Sure Proper Delay is In Place
if (Freq < 16000000)
nop
else
goto $ + 1
endif
endif
bcf EPort, EPin
bsf Dlay, Dlay160Bit1          ; Delay 160 usecs
if (Dlay160Bit2 != -1)
bsf Dlay, Dlay160Bit2
endif
decfsz Dlay, f
goto $ - 1
bsf EPort, EPin                ; Send the Third Reset Instruction
if (Freq > 8000000)             ; Make Sure Proper Delay is In Place
if (Freq < 16000000)
nop
else
goto $ + 1
endif
endif
bcf EPort, EPin
bsf Dlay, Dlay160Bit1          ; Delay 160 usecs
if (Dlay160Bit2 != -1)
bsf Dlay, Dlay160Bit2
endif
decfsz Dlay, f
goto $ - 1
if (DataBit == 0)              ; Send the Data Length Specification
movlw 0x002
else
movlw 0x020
endif
movwf DataPort
if (Freq > 8000000)             ; Make Sure Proper Delay is In Place
if (Freq < 16000000)
nop
else
goto $ + 1
endif
endif
bsf EPort, EPin                ; Make Sure Proper Delay is In Place
if (Freq > 8000000)
if (Freq < 16000000)
nop
else
```

```

goto $ + 1
endif
endif
bcf EPort, EPin
bsf Dlay, Dlay160Bit1           ; Delay 160 usecs
if (Dlay160Bit2 != -1)
bsf Dlay, Dlay160Bit2
endif
decfsz Dlay, f
goto $ - 1
movlw 0x028                     ; Set Interface Length
call LCDIns
movlw 0x010                     ; Turn Off Display
call LCDIns
movlw 0x001                     ; Clear Display RAM
call LCDIns
movlw 0x006                     ; Set Cursor Movement
call LCDIns
movlw 0x00E                     ; Turn on Display/Cursor
call LCDIns
return
endm

```

I recommend that you put all the I/O pins and the 4-bit `DataPort` on the same 8-bit I/O port. The reasons for doing this is that when using this code, writes to the `DataPort` will change the output values of other registers. To avoid any potential problems, put the 7 bits together in the same `PORT`, and use the other bit only for input.

The last LCD interface macro that I am providing you with is the two-wire LCD interface. This interface is quite a bit slower than the others that I have presented, but it uses the fewest PIC microcontroller I/O pins. The circuit schematic for this interface was shown earlier in the book (as are the schematics for all the LCD interfaces). The `LCD2` macro requires only the `Dlay` and `LCDTemp` variables.

```

LCD2 Macro ClockPort, ClockPin, DataPort, DataPin, Freq
    variable Dlay5Value, Dlay160Value, Dlay160Bit1 = -1, Dlay160Bit2
    = -1
    variable BitCount = 0, i
    variable Value = 128, Bit = 7
    Dlay5Value = ((5007 * (Freq / 1000) / 4000) / 7) + 256
    Dlay160Value = (163 * (Freq / 1000) / 4000) / 3

    while (Bit >= 0)                ; Find the Number of Bits and their
                                    ; Positions in "Dlay160Value"
    if ((Dlay160Value & Value) != 0)
    if (Dlay160Bit1 == -1)          ; Set the Upper Bit
    Dlay160Bit1 = Bit

```

```
    else
    if (Dlay160Bit2 == -1)
Dlay160Bit2 = Bit
    endif
    endif
    BitCount = BitCount + 1
    endif
    Value = Value >> 1
    Bit = Bit - 1
    endw
    if (BitCount > 2) ; Just Want max two Bits
    if ((Dlay160Bit1 - 1) == Dlay160Bit2)
Dlay160Bit1 = Dlay160Bit1 + 1 ; Shift Top up by 1
Dlay160Bit2 = -1 ; Delete Second
    else
Dlay160Bit2 = Dlay160Bit2 + 1 ; Shift Bottom up by 1
    endif
    endif

Dlay5 ; Delay 5 msecs
    movlw (Dlay5Value & 0xFF00) >> 8
    movwf Dlay
    movlw Dlay5Value & 0xFF
    subwf Dlay, w
    xorlw 0xFF
    addwf Dlay, w
    btfsc STATUS, Z
    decfsz Dlay, f
    goto $ - 5
    return

LCDPORTInit ; Initialize the I/O Ports
    bsf STATUS, RP0 ; ONLY used by mid-range
    bcf ClockPort, ClockPin
    bcf DataPort, DataPin
    bcf STATUS, RP0
    bcf ClockPort, ClockPin
    bcf DataPort, DataPin
    return

LCDIns ; Send the Instruction to the LCD
    movwf LCDTemp ; Save the Value
    movlw 6 ; Clear the Shift Register
    movwf Dlay
    bsf ClockPort, ClockPin
    bcf ClockPort, ClockPin
    decfsz Dlay, f
    goto $ - 3
    movwf Dlay ; w still equals 6
```

```

movf LCDTemp, w           ; Shift out the Upper 4 Bits
swapf LCDTemp, f
bsf LCDTemp, 5           ; Make LCDTemp Correct for Shifting
bcf LCDTemp, 4           ; This is "RS" Bit
bcf DataPort, DataPin    ; Shift Out Each Bit
btfsc LCDTemp, 5         ; 5 is the Current MSB
bsf DataPort, DataPin    ; Shift Out the Next Highest Bit
bsf ClockPort, ClockPin
bcf ClockPort, ClockPin
rlf LCDTemp, f
decfsz Dlay, f
goto $ - 7
bsf DataPort, DataPin    ; Latch in the Data
if (Freq > 8000000)      ; Make Sure Proper Delay is In Place
if (Freq < 16000000)
nop
else
goto $ + 1
endif
endif
bcf DataPort, DataPin
bsf Dlay, 2              ; Dlay = 6 for Shift Out
bsf Dlay, 1
bsf ClockPort, ClockPin ; Clear the Shift Register
bcf ClockPort, ClockPin
decfsz Dlay, f
goto $ - 3
movwf LCDTemp           ; Shift out the Low Nybble
bsf Dlay, 2              ; Dlay = 6 for Shift Out
bsf Dlay, 1
bsf LCDTemp, 5           ; Make LCDTemp Correct for Shifting
bcf LCDTemp, 4           ; This is "RS" Bit
bcf DataPort, DataPin    ; Shift Out Each Bit
btfsc LCDTemp, 5         ; 5 is the Current MSB
bsf DataPort, DataPin    ; Shift Out the Next Highest Bit
bsf ClockPort, ClockPin
bcf ClockPort, ClockPin
rlf LCDTemp, f
decfsz Dlay, f
goto $ - 7
bsf DataPort, DataPin    ; Latch in the Data
if (Freq > 8000000)      ; Make Sure Proper Delay is In Place
if (Freq < 16000000)
nop
else
goto $ + 1
endif
endif
bcf DataPort, DataPin

```

```
    bsf Dlay, Dlay160Bit1          ; Delay 160 usecs
    if (Dlay160Bit2 != -1)
    bsf Dlay, Dlay160Bit2
    endif
    decfsz Dlay, f
    goto $ - 1
    andlw 0x0FC                    ; Have to Delay 5 msecs?
    btfsc STATUS, Z
    call Dlay5
    return

LCDChar                            ; Send the Character to the LCD
    movwf LCDTemp                  ; Save the Value
    movlw 6                        ; Clear the Shift Register
    movwf Dlay
    bsf ClockPort, ClockPin
    bcf ClockPort, ClockPin
    decfsz Dlay, f
    goto $ - 3
    movwf Dlay                      ; w still equals 6
    movf LCDTemp, w                ; Shift out the Upper 4 Bits
    swapf LCDTemp, f
    bsf LCDTemp, 5                  ; Make LCDTemp Correct for Shifting
    bsf LCDTemp, 4                  ; This is "RS" Bit
    bcf DataPort, DataPin           ; Shift Out Each Bit
    btfsc LCDTemp, 5                ; 5 is the Current MSB
    bsf DataPort, DataPin           ; Shift Out the Next Highest Bit
    bsf ClockPort, ClockPin
    bcf ClockPort, ClockPin
    rlf LCDTemp, f
    decfsz Dlay, f
    goto $ - 7
    bsf DataPort, DataPin           ; Latch in the Data
    if (Freq > 8000000)              ; Make Sure Proper Delay is In Place
    if (Freq < 16000000)
    nop
    else
    goto $ + 1
    endif
    endif
    bcf DataPort, DataPin
    bsf Dlay, 2                      ; Dlay = 6 for Shift Out
    bsf Dlay, 1
    bsf ClockPort, ClockPin         ; Clear the Shift Register
    bcf ClockPort, ClockPin
    decfsz Dlay, f
    goto $ - 3
    movwf LCDTemp                  ; Shift out the Low Nybble
```

```

bsf   Dlay, 2                ; Dlay = 6 for Shift Out
bsf   Dlay, 1
bsf   LCDTemp, 5             ; Make LCDTemp Correct for Shifting
bsf   LCDTemp, 4             ; This is "RS" Bit
bcf   DataPort, DataPin      ; Shift Out Each Bit
btfsc LCDTemp, 5             ; 5 is the Current MSB
bsf   DataPort, DataPin      ; Shift Out the Next Highest Bit
bsf   ClockPort, ClockPin
bcf   ClockPort, ClockPin
rlf   LCDTemp, f
decfsz Dlay, f
goto  $ - 7
bsf   DataPort, DataPin      ; Latch in the Data
if (Freq > 8000000)          ; Make Sure Proper Delay is In Place
if (Freq < 16000000)
nop
else
goto  $ + 1
endif
endif
bcf   DataPort, DataPin
bsf   Dlay, Dlay160Bit1      ; Delay 160 usecs
if (Dlay160Bit2 != -1)
bsf   Dlay, Dlay160Bit2
endif
decfsz Dlay, f
goto  $ - 1
return

LCDInit                      ; Do the 8 Bit Initialization
call  Dlay5                  ; Wait 15 msecs
call  Dlay5
call  Dlay5
movlw 0x023                  ; Initialize the I/O Port
movwf LCDTemp                ; Save the Value
movlw 6                      ; Clear the Shift Register
movwf Dlay
bsf   ClockPort, ClockPin
bcf   ClockPort, ClockPin
decfsz Dlay, f
goto  $ - 3
movwf Dlay
bcf   DataPort, DataPin      ; Shift Out Each Bit
btfsc LCDTemp, 5             ; 5 is the Current MSB
bsf   DataPort, DataPin      ; Shift Out the Next Highest Bit
bsf   ClockPort, ClockPin
bcf   ClockPort, ClockPin
rlf   LCDTemp, f

```

```
decfsz Dlay, f
goto $ - 7
bsf DataPort, DataPin          ; Latch in the Data
if (Freq > 8000000)             ; Make Sure Proper Delay is In Place
if (Freq < 16000000)
nop
else
goto $ + 1
endif
endif
bsf DataPort, DataPin
call Dlay5
bsf DataPort, DataPin          ; Send another 0x03 to the LCD
if (Freq > 8000000)             ; Make Sure Proper Delay is In Place
if (Freq < 16000000)
nop
else
goto $ + 1
endif
endif
bsf DataPort, DataPin
bsf Dlay, Dlay160Bit1          ; Delay 160 usecs
if (Dlay160Bit2 != -1)
bsf Dlay, Dlay160Bit2
endif
decfsz Dlay, f
goto $ - 1
bsf DataPort, DataPin          ; Send another 0x03 to the LCD
if (Freq > 8000000)             ; Make Sure Proper Delay is In Place
if (Freq < 16000000)
nop
else
goto $ + 1
endif
endif
bsf DataPort, DataPin
bsf Dlay, Dlay160Bit1          ; Delay 160 usecs
if (Dlay160Bit2 != -1)
bsf Dlay, Dlay160Bit2
endif
decfsz Dlay, f
goto $ - 1
movlw 0x022                    ; Initialize the I/O Port
movwf LCDTemp                  ; Save the Value
movlw 6                        ; Clear the Shift Register
movwf Dlay
bsf ClockPort, ClockPin
bsf ClockPort, ClockPin
```



```

decfsz Dlay, f
goto $ - 3
movwf Dlay
bcf DataPort, DataPin          ; Shift Out Each Bit
btfsc LCDTemp, 5               ; 5 is the Current MSB
bsf DataPort, DataPin          ; Shift Out the Next Highest Bit
bsf ClockPort, ClockPin
bcf ClockPort, ClockPin
rlf LCDTemp, f
decfsz Dlay, f
goto $ - 7
bsf DataPort, DataPin          ; Latch in the Data
if (Freq > 8000000)             ; Make Sure Proper Delay is In Place
if (Freq < 16000000)
nop
else
goto $ + 1
endif
endif
bcf DataPort, DataPin
bsf Dlay, Dlay160Bit1          ; Delay 160 µsecs
if (Dlay160Bit2 != -1)
bsf Dlay, Dlay160Bit2
endif
decfsz Dlay, f
goto $ - 1
movlw 0x028                    ; Set Interface Length
call LCDIns
movlw 0x010                    ; Turn Off Display
call LCDIns
movlw 0x001                    ; Clear Display RAM
call LCDIns
movlw 0x006                    ; Set Cursor Movement
call LCDIns
movlw 0x00E                    ; Turn on Display/Cursor
call LCDIns
return
endm

```

When you execute the two-wire LCD interface, you will find that it takes up about twice the number of instructions as any of the other methods and takes six or more times to run. Its chief advantage is the requirement for only two I/O pins to operate.

I should point out that all these LCD interfaces will work properly with interrupts enabled and processing. If any operations are interrupted, then the timings will be stretched out (which is not a problem). The only issue would be if minimum processing times were interrupted.

NRZ SERIAL I/O

Throughout this book I have worked with RS-232 as a method of interfacing to the PIC microcontroller. In this section I want to give you two different ways of creating a “bit banging” interface using very easy-to-use macros. Using these macros, you should be able to implement an NRZ serial interface in literally minutes for your application.

The first method is a traditional “bit banging” interface that can be used by both low-end and mid-range PIC microcontrollers that do not have built-in UART (or USART) ports. To set up the serial interfaces, the macro

NRZSerialNI Macro TXPort, TXPin, RXPort, RXPIn, Polarity, Rate, Freq

is invoked, where TXPort and TXPin, along with RXPort and RXPIn, are used to define the transmit port and the receive port, respectively. As I will discuss in the next section, these pairs of pins can be combined into a single define to make the definition easier. The polarity of the signals is defined as Pos for positive or positive logic and Neg for negative or inverted logic (useful for interfacing to RS-232 directly through a current-limiting resistor). Rate is the data rate (in bits per second), and Freq is the speed the processor is executing at in hertz.

For the traditional “bit banging” interface, the macro can be invoked anywhere in the code, but I recommend doing so at the end of the application for mid-range PIC microcontrollers and at the start of the application, after putting in a jump around the code, in low-end microcontrollers.

When the macro is expanded, the bit delay calculations are made, and the “bit banging” serial receive and transmit subroutines are inserted into the application. The macros can be used by either low-end or mid-range PIC microcontrollers without modification. The macro code is

```
NRZSerialNI Macro TXPort, TXPin, RXPort, RXPIn, Polarity, Rate,
Frequency
    variable BitDelay
    BitDelay = Frequency / (4 * Rate)

    SerialRX                                ; Receive 8-N-1
    if (Polarity == Pos)
        btfsc RXPort, RXPIn ; Wait for a Bit to Come in
    else
        btfss RXPort, RXPIn
    endif
    goto $ - 1

    DlayMacro BitDelay / 2                ; Wait 1/2 a Bit to Confirm

    if (Polarity == Pos)
        btfsc RXPort, RXPIn ; Confirm Data is Correct
    else
        btfss RXPort, RXPIn
```

```

endif
goto SerialRX          ; If Just a "Glitch", Restart Start Bit
; Poll
movlw 8                ; Wait for 8 Bits
SRXLoop
if ((BitDlay - 10) > 770) ; Check to See if Value is Too Large
DlayMacro 770          ; Put in a "Double" Delay
DlayMacro BitDlay - (770 + 10)
else
DlayMacro BitDlay - 10  ; Wait for the Middle of the Next Bit
endif

bcf STATUS, C          ; Check the Incoming Data
if (Polarity == Pos)
btfsc RXPort, RXPin
else
btfss RXPort, RXPin
endif
bsf STATUS, C
rrf NRZTemp, f         ; Shift in the Bit

subwf NRZTemp, w       ; Decrement and End if == 0
xorlw 0xFF
addwf NRZTemp, w
btfss STATUS, Z
goto SRXLoop

if ((BitDlay - 9) > 770); Check to See if Value is Too Large
DlayMacro 770          ; Put in a "Double" Delay
DlayMacro BitDlay - (770 + 9)
else
DlayMacro BitDlay - 9  ; Wait for the Middle of the Next Bit
endif

if (Polarity == Pos)   ; Is there a Stop Bit?
btfss RXPort, RXPin
else
btfsc RXPort, RXPin
endif
goto SerialRX         ; No, Start All Over Again

movf NRZTemp, w       ; Return the Received Byte
return                ; Note - Zero Returned in Low-End
; Devices

SerialTX
movwf NRZTemp         ; Save the Byte to Output
movlw 10
bcf STATUS, C         ; Start with Sending the Start Bit

```

```
STXLoop
  if (Polarity == Pos)      ; Output the Current Bit
    btfsc STATUS, C
  else
    btfss STATUS, C
  endif
  goto $ + 4                ; 6 Cycles Required Each Time
  nop
  bcf  TXPort, TXPin        ; Output a "Low"
  goto $ + 3
  bsf  TXPort, TXPin        ; Output a "High"
  goto $ + 1
  if ((BitDelay - 15) > 770) ; Check to See if Value is Too Large
    DelayMacro 770          ; Put in a "Double" Delay
    DelayMacro BitDelay - (770 + 15)
  else
    DelayMacro BitDelay - 15 ; Wait for the Middle of the Next Bit
  endif
  subwf NRZTemp, w          ; Decrement the Bit Counter
  xorlw 0x0FF
  addwf NRZTemp, w
  btfsc STATUS, Z
  return                    ; Can Return to Caller
  bsf  STATUS, C            ; Shift Down the Next Bit
  rrf  NRZTemp, f
  goto STXLoop
endm
```

If you are working with a mid-range PIC microcontroller, you also can invoke the `NRZSerialNISetup` macro that creates the `SerialSetup` subroutine. This subroutine puts the TX pin in output mode and drives an idle output. This subroutine should be executed as early as possible after the application has started. This will ensure that the PIC microcontroller does not inadvertently cause the receiver to process invalid data by missing the first start bit.

```
NRZSerialNISetup Macro TXPort, TXPin, Polarity
SerialSetup                ; Setup the Serial I/O Bits
  bsf  STATUS, RP0
  bcf  TXPort, TXPin        ; Make TX Pin an Output
  bcf  STATUS, RP0
  if (Polarity == Pos)
    bsf  TXPort, TXPin      ; Transmit "idle"
  else
    bcf  TXPort, TXPin
  endif
  return
endm
```

Sending a byte is as simple as loading the `w` register with the ASCII byte to transmit and calling `SerialTX`. By calling a `SerialRX`, the PIC microcontroller will poll the RX pin until a valid byte has been received. In mid-range devices, on return from `SerialRX`, the `w` register will have the byte received. For low-end PIC microcontrollers, you will have to retrieve the received byte from the variable `NRZTemp` because there is no way to return data from a subroutine in the `w` register.

When using this macro, you will have to declare the `DlayCount` and `NRZTemp` variables. If you are going to use this macro, then you will have to run without interrupts and make sure that data is transmitted and received in such a way that nothing is lost. This isn't a huge hardship, but it is one that you should be aware of.

If you want to have better performance in your mid-range PIC microcontroller, you can use the `TMR0` interrupt to poll the receive line and send data at a regular byte interval. The macro presented below will enable the `TMR0` interrupt to send a request to the processor three times each potential bit. I have described this algorithm elsewhere in this book, and the base code itself is taken from the YAP-II's interrupt handler/serial interface.

Invocation of the interrupt-based serial interface is similar, with the only differences being that the macro name is slightly different and that this code should be put at the start of the application (ideally at the reset vector). The code itself will install an interrupt handler, along with hardware (I/O ports, `TMR0`, and interrupts) initialization and the `SerialRX` and `SerialTX` subroutines. These subroutines work exactly the same as the "bit banging" version of the serial I/O macro, except that while they are operating, the interrupt handler can execute simultaneous transmits and receives.

```
NRZSerialI Macro TXPort, TXPin, RXPort, RXPin, Polarity, Rate,
Frequency
    variable BitDlay, Prescaler, TMR0Reset
    BitDlay = (Frequency / (3 * 4 * Rate)) - 10
    TMR0Reset = BitDlay / 2      ; Using TMR0, Calculate the Timer Reset
    Value
    Prescaler = 0                ; And the Prescaler
    while (TMR0Reset > 0x0FF) ; Find the Proper Reset Value
    TMR0Reset = TMR0Reset / 2
    Prescaler = Prescaler + 1
    endw
    if (Prescaler > 7)           ; Can't Use TMR0
    error "Bit Delay cannot use TMR0 for Polling Clock"
    endif
    TMR0Reset = 256 - TMR0Reset ; Get the TMR0 Reset Value

    goto AfterInt                ; Jump to After Interrupt
    org 4
Int                               ; Interrupt Handler
    movwf _w
    movf STATUS, w
```

```
    bcf  STATUS, RP0           ; Make Sure in Bank 0
    movwf _status

    bcf  INTCON, T0IF          ; Reset the Timer Interrupt

    movlw TMR0Reset
    movwf TMR0

; First, Check for a Received Character
Int_RX

    movlw 0x004                ; Check for Bit?
    addwf RXCount, f
    btfss STATUS, DC           ; DC Not Affected by "clrf
    goto _RXNo                 ; Nothing to Check for (Yet)

    movf RXCount, w            ; Everything Read Through?
    xorlw 0x091
    btfsc STATUS, Z
    goto _RXAtEnd              ; Yes, Check for Stop Bit

    bcf  STATUS, C              ; Read the Current State
    if (Polarity == Pos)
    btfsc RXPort, RXPIN        ; Sample at 10 Cycles
    else
    btfss RXPort, RXPIN
    endif
    bsf  STATUS, C
    rrf  RXByte, f

    bsf  RXCount, 2             ; Start Counting from 4

_RXEnd13
    nop
    goto _RXEnd                ; End 15 Cycles From "Int_RX" - Finished
Receiving Bit

_RXEnd8                        ; Finished - 8 Cycles to Here
    goto $ + 1
    nop
    goto _RXEnd13

_RXNo                          ; 5 Cycles from "Int_RX" - No Bit to Receive

    btfsc RXCount, 0           ; Something Running?
    goto _RXEnd8              ; End 8 Cycles from "Int_RX" - Yes, Skip Over

    btfsc RXCount, 3           ; Checking Start Bits?
    goto _RXStartCheck
```

```

    if (Polarity == Pos)
    btfsc RXPort, RXPIN      ; If Line Low - "Start" Bit
    else
    btfss RXPort, RXPIN
    endif
    bcf RXCount, 2          ; Don't Have a "Start" Bit

    goto _RXEnd13           ; End 18 cycles from "Int_RX"

_RXStartCheck              ; 10 Cycles to Here

    if (Polarity == Pos)
    btfsc RXPort, RXPIN
    else
    btfss RXPort, RXPIN
    endif
    movlw 0x0FF             ; Nothing - Clear "RXCount"

    addlw 1
    movwf RXCount

    goto _RXEnd            ; 16 Cycles to End

_RXAtEnd                  ; 9 Cycles from "Int_RX" - Check Last
; Bit
    if (Polarity == Pos)
    btfsc RXPort, RXPIN
    else
    btfss RXPort, RXPIN
    endif
    bsf RXFlag

    clrf RXCount           ; Finished - Reset Check - 12 Cycles

    goto $ + 1
    goto _RXEnd

_RXEnd

; Next, Check for Transmitting a Character - Intrinsic Delay 22 Cycles
Int_TX

    movlw 0x004            ; Interrupt Transmit Increment Value
    addwf TXCount, f
    btfss STATUS, DC       ; Send the Next Byte?
    goto _TXSendDelayCheck

    bsf TXCount, 2         ; Want to Increment 3x not Four for each Bit

```

```
bsf STATUS, C
rrf TXByte, f

movf TXPort, w          ; Send Next Bit
andlw 0xFF ^ (1 << TXPin)
if (Polarity == Pos)
btfsc STATUS, C
else
btfss STATUS, C
endif
iorlw 1 << TXPin
movwf TXPort            ; Cycle 12 is the Bit Send

goto _TXCompletedGoOn   ; TX Takes 14 Cycles

_TXSendDelayCheck       ; Don't Send Bit, Check for Start Bit

btfss TXCount, 0        ; Bit Zero Set (Byte to Send)?
goto _TXNothingtoCheck

movlw 0x004             ; Setup the Timer to Increment 3x
movwf TXCount

movf TXPort, w          ; Output Start Bit
if (Polarity == Pos)
andlw 0xFF ^ (1 << TXPin)
else
iorlw 1 << TXPin
endif
movwf TXPort

goto _TXCompletedGoOn   ; TX First Bit Takes 14 Cycles

_TXNothingtoCheck       ; Nothing Being Sent?

movf TXCount, w
xorlw 0x004             ; Zero (Originally) TXCount?
btfss STATUS, Z
xorlw 0x004 ^ 0x09C
btfsc STATUS, Z
clrf TXCount

_TXCompletedGoOn        ; Finished with TX, Do RX

movf _status, w         ; Restore the Interrupts
movwf STATUS
swapf _w, f
swapf _w, w
retfie
```


SerialRX

```

    bcf  RXFlag                ; Reset the

    btfss RXFlag              ; Wait for a Character to be Received
    goto $ - 1

    movf RXByte, w            ; Return the Character Read in

    return

```

SerialTX

```

    movf TXCount, f           ; Anything Being Sent?
    btfss STATUS, Z           ; Wait for the Previous Send to End
    goto $ - 2

    movwf TXByte              ; Send out the Character

    bsf  TXCount, 0           ; Indicate to the Interrupt Handler that
    it can Send Something

    return

```

```

AfterInt                ; Can Return the Value
    bsf  STATUS, RP0         ; Setup the Interrupts/TX Output
    bcf  TXPort, TXPin
    movlw 0x0D0 + Prescaler
    movwf OPTION_REG ^ 0x080 ; User Prescaler with TMR0
    bcf  STATUS, RP0
    if (Polarity == Pos)
    bsf  TXPort, TXPin       ; Output "Idle" for Data Transmit
    else
    bcf  TXPort, TXPin
    endif
    movlw TMR0Reset          ; Reset the Timer
    movwf TMR0
    movlw (1 << GIE) + (1 << T0IE)
    movwf INTCON             ; Start up the Interrupts

    clrf RXCount             ; Make Sure No Counts are Starting
    clrf TXCount

    endm

```

Along with the macro invocation, the following variables will have to be declared for the code to work:

`_w, _status`—Interrupt handler context save registers
`RXByte, TXByte`—Data transmit and receive bytes
`RXCount, TXCount`—Serial transfer data count/status variables
`Flags`—Execution flag variable

Along with these variables, the `RXFlag` bit also will have to be defined for use by the code to indicate when a valid byte has been received.

MID-RANGE I2C INTERFACE

When I need to create my own “bit banging” inter-intercomputer master interface for the PIC microcontroller to communicate with peripheral devices, I use the macro that is listed below and call the five subroutines that are put into the application, `I2CBitSetup`, `I2CStart`, `I2CStop`, `I2CSend`, and `I2CRead`. The macro is designed for mid-range PIC microcontrollers, and I will comment on implementing I2C on low-end devices. I will not bother with the same discussion for the `PIC17Cxx` and `PIC18Cxx` because these parts with a built-in master SSP can be purchased easily.

```
I2CSetup Macro ClockPort, ClockPin, DataPort, DataPin, Rate, Frequency
    variable Dlay, Fraction    ; Delay in Instruction Cycles
    Dlay = ((Frequency * 110) / (800 * Rate)) / 1000
    Fraction = ((Frequency * 110) / (800 * Rate)) - (Dlay * 1000)
    if (Fraction > 499)
    Dlay = Dlay + 1
    endif

I2CBitSetup                                ; Setup I2C Lines for Application
    bsf STATUS, RP0
    bcf ClockPort, ClockPin    ; Driving Output
    bcf DataPort, DataPin
    bcf STATUS, RP0
    bsf ClockPort, ClockPin    ; Everything High Initially
    bsf DataPort, DataPin
    DlayMacro Dlay              ; Make Sure Lines are High for adequate
                                ; Period of Time

    return

I2CStart                                    ; Send a “Start” Pulse to the I2C Device
    bsf ClockPort, ClockPin
    bsf DataPort, DataPin
    DlayMacro Dlay - 2
    bcf DataPort, DataPin      ; Drop the Data Line
    DlayMacro Dlay
    bcf ClockPort, ClockPin    ; Drop the Clock Line
```

```

DlayMacro Dlay - 2          ; Wait for the Specified Period
return                      ; Exit with Clock = Low, Data = Low

I2CStop                    ; Pass Stop Bit to I2C Device
DlayMacro Dlay
bsf  ClockPort, ClockPin   ; Clock Bit High
DlayMacro Dlay
bsf  DataPort, DataPin
return                      ; Exit with Clock = High, Data = High

I2CRead                    ; Read 8 Bits from the Line
                          ; Reply with "ACK" in Carry Flag
                          ; Put in the Carry Flag
bsf  I2CTemp, 0
btfsc STATUS, C
bcf  I2CTemp, 0            ; If Carry Set, then Send "Ack" (-ative)
bsf  STATUS, RP0           ; Let the I2C Device Drive the Data Line
bsf  DataPort, DataPin
bcf  STATUS, RP0
movlw 0x010 - 8

I2CRLoop
bsf  ClockPort, ClockPin   ; Bring the Clock Line Up
DlayMacro (Dlay / 2) - 1
bcf  STATUS, C
btfsc DataPort, DataPin    ; Sample the Incoming Data
bsf  STATUS, C
DlayMacro (Dlay / 2) - 2
bcf  ClockPort, ClockPin
rlf  I2CTemp, f            ; Shift in the Bit
andlw 0x07F               ; Store the Ack of Bit 7 of the Data
btfsc STATUS, C
iorlw 0x080               ; If High, Set Bit 7
addlw 0x001               ; Finished, Do the Next Bit
DlayMacro Dlay - 9        ; Put in "TLow"
btfss STATUS, DC
goto I2CRLoop
bcf  DataPort, DataPin
bsf  STATUS, RP0          ; Send Ack Bit
bcf  DataPort, DataPin
bcf  STATUS, RP0
andlw 0x080              ; High or Low?
btfss STATUS, Z
bsf  DataPort, DataPin    ; Low, Send Ack
DlayMacro Dlay / 18       ; Any Reason to delay?
bsf  ClockPort, ClockPin
DlayMacro Dlay
bcf  ClockPort, ClockPin
bcf  DataPort, DataPin
movf I2CTemp, w          ; Get the Received Byte
return                  ; Return with Clock = Data = Low

```

```
I2CSend                                ; Send the 8 Bits in "w" and Return Ack
    movwf I2CTemp
    movlw 0x010 - 8
I2CSLoop
    rlf I2CTemp, f                      ; Shift up the Data into "C"
    btfsc STATUS, C
    goto $ + 4
    nop
    bcf DataPort, DataPin              ; Low Bit
    goto $ + 3
    bsf DataPort, DataPin              ; High Bit
    goto $ + 1
    bsf ClockPort, ClockPin            ; Strobe Out the Data
    DlayMacro Dlay
    bcf ClockPort, ClockPin
    DlayMacro Dlay - 12
    addlw 1
    btfss STATUS, DC
    goto I2CSLoop
    DlayMacro 6
    bsf STATUS, RP0                    ; Now, Get the Ack Bit
    bsf DataPort, DataPin
    bcf STATUS, RP0
    bsf ClockPort, ClockPin
    DlayMacro (Dlay / 2) - 1
    bcf STATUS, C
    btfss DataPort, DataPin
    bsf STATUS, C                      ; Line Low, "Ack" Received
    DlayMacro (Dlay / 2) - 2
    bsf STATUS, RP0
    bcf DataPort, DataPin
    bcf STATUS, RP0
    bcf ClockPort, ClockPin
    bcf DataPort, DataPin
    return                             ; Return with Ack in Carry,
    endm                               ; Clock = Data = Low
```

Invoking the macro may seem a bit overwhelming (but it really isn't). Normally, when I create applications that use specific pins for specific functions, I use the `#define` directive like

```
#define I2CClock PORTA, 0
```

which, when used with the `I2CSetup` macro, simplifies the number of parameters and cuts down on the thinking required. Using `I2CClock` and `I2CData` defines, the macro invocation becomes

```
I2CSetup I2CClock, I2CData, Rate, Frequency
```

The `Rate` parameter is the I2C rate specified in kilohertz. The normal two speeds are 100 and 400 kHz. The `Frequency` parameter is the execution clock speed and is used, along with the `Rate` parameter, to determine whether or not delays are required to implement the I2C code.

When the `I2CSetup` macro is invoked, one of the first things that I do is to calculate the `D1ay` value. This value is used to specify the number of instruction cycles that must be delayed before the next edge is produced. This is similar to the need for delays in the `NRZ Serial I/O` macros presented in the preceding section. The I2C timings are a lot less critical because all I am doing is trying to make sure that no “minimum” timings are violated.

After invoking the `I2CSetup` macro, the `I2CBitSetup` subroutine is called, which sets the I2C `clock` and data bits as output, and both lines are driven “high” for the I2C devices connected to the PIC microcontroller to be put into idle state. The `Clock` line will be driven at all times by the PIC microcontroller, but the `Data` line should be pulled up (with a 1- k Ω to 10-k Ω resistor for the 400- and 100-kHz data rates, respectively) to allow devices with open collector drives to work.

To send data, the `I2CStart` subroutine is called, followed by the `I2CSend`, `I2CRead`, and `I2CStop` subroutines. The `w` register is used for passing data back and forth, and the carry flag is used to indicate the state of the acknowledge bit. When carry is set, the acknowledgement (`Ack`) is received; when carry is reset, a `Nack` is received. This is also true for sending data to the I2C peripheral device.

Earlier in the book I discussed sending serial data to I2C devices using the format

```
idle  — Start  — CommandWriteA — AddressByteA — Start  —
CommandReadA —
DataA — DataN — Stop  — idle
```

to explain how a 16-bit I2C EEPROM read would be carried out. Using the subroutines in the `I2CSetup` macro, the PIC microcontroller code for carrying out this function would be

```
call I2CStart                ; Start the Transfer

movlw CommandWrite           ; Send the Address to Read the
call I2CSend                 ; Sixteen Bit Word

movlw AddressByte
call I2CSend

call I2CStart                ; Reset the I2C EEPROM to Read Back

movlw CommandRead            ; Send the Read Command
call I2CSend
```

```
bsf STATUS, C                ; Read the Byte with Ack
call I2CRead
movwf I2CData

bcf STATUS, C                ; Read the next byte and stop the
call I2Cread                  ; transfer with the Nack
movwf I2CData + 1

call I2CStop                  ; Finished with the I2C Operation
```

Note that in this code I ignore the acknowledgments coming back from the addressed devices. Normally, when I work with I2C, I know what is on the network, so I don't have to worry about whether or not the device is active unless the acknowledgment is used as a Ready flag (which is the case for many I2C EEPROMs and ADCs).

This code will not work for the low-end PIC microcontroller architecture. I have not included sample code for the low-end PIC microcontroller architecture because of the way the TRIS registers are accessed (i.e., written from the w register and cannot be read back). For this reason, I recommend that the code shown above should be used as a base for implementing a low-end PIC microcontroller I2C interface. In doing this, you should keep track of the current TRIS values, change these values, and then write them into the TRIS register. Note that these operations should be done *outside* the loops in I2CRead and I2CSend.

BUTTON DEBOUNCING

Button debouncing is not something that most people consider to be a “bit banging” routine. Normal “bit banging” routines are code that emulates a function that is available in different devices. I am not aware of any microcontrollers that have a button debounce feature, but the two macros presented here will give you in-line debounce capabilities for your applications with a very simple interface or one that is more complex. The more complex button debouncing macro will “poll” the button and jump to a label if the button is pressed and debounced.

The first button debouncing macro is inserted in the source code and waits for a port pin to reach a set state for a specific amount of time before continuing.

```
Debounce macro HiLo, Port, Bit
if HiLo == Lo
btfss Port, Bit      ; Is the Button Pressed?
else
btfsc Port, Bit
endif
goto $ - 1           ; Yes - Wait for it to be Released
movlw InitDelay      ; Wait for Release to be Debounced
movwf Delay           ; Have to Delay 20 msecs
movlw 0
if HiLo == Lo
```

```

btfss Port, Bit      ; If Button Pressed, Wait Again for it
else
btfsc Port, Bit
endif
goto $ - 6           ; to be Released
ifndef Debug         ; Skip Small Loop if "Debug" Defined
addlw 1              ; Increment the Delay Count
btfsc STATUS, Z      ; Loop If Low Byte (w) Not Equal to Zero
else
nop                  ; Match the Number of Instructions
nop
endif
decfsz Delay
goto $ - 5
endm

```

The InitDelay constant is calculated using the formula

$$\text{TimeDelay} = (((\text{InitDelay} - 1) * 256) * 7) / (\text{Frequency} / 4)$$

or

$$\text{InitDelay} = ((\text{TimeDelay} * (\text{Frequency} / 4)) / (256 * 7)) + 1$$

Using these formulas (which could be inserted into the macro), you will calculate an InitDelay value of 12, for a 20-ms debounce delay for a PIC microcontroller running at 4 MHz. If you take out the rounding error, you will find that the actual debounce delay is 19.7 ms, which is more than close enough to provide an accurate debouncing interval.

The second button debounce macro works similarly to the Parallax BASIC Stamp's Button function. For more information on the Button function, check out the PICBASIC language definition in Appendix E

```

Button macro Port, Pin, Down, Delay, Rate, Variable, Target, Address
local ButtonEnd
incf Variable, w      ; Increment the Counter Variable
if ((Down == 0) && (Target == 0)) || ((Down == 1) && (Target == 1))
btfsc Port, Pin       ; If Low, then Valid Pin
else
btfss Port, Pin       ; If High, then Valid Pin
endif
clrw                  ; Not Pressed, Clear the Counter
movwf Variable        ; Save the Counter Value
movlw Delay & 0x07F
subwf Variable, w     ; Button Debounced?
btfsc STATUS, Z
goto Address          ; If Equal, then "Yes"

```

TABLE G.1 Button MACRO PARAMETERS	
PARAMETER	FUNCTION
Port, Pin	The button pin (i.e. PORTA, 0)
Down	The state when the button is pressed
Delay	The number of iterations of the macro code before the Address is jumped to (to 127); if set to 0, then jump if Target is met without any debouncing; if bit 7 of Delay is set, then no autorepeats
Rate	After the initial jump to Address, the number of cycles (to 127) before autorepeating
Target	The state (1 or 0) to respond to.
Address	The Address to jump to when the button is pressed or autorepeats

```
if ((Delay & 0x080) != 0)    ; Is Autorepeat used?
btfsc STATUS, C
decf Variable                ; No - Decrement if > "Delay"
else
btfss STATUS, C
goto ButtonEnd              ; Less than Expected - End
xorlw Rate                  ; At the Autorepeat Point yet?
btfsc STATUS, Z
goto ButtonEnd              ; No - Keep Incrementing
movlw Delay                 ; Yes, Reset back to the Original
movwf Variable              ; Count and Repeat
goto Address
endif
ButtonEnd                   ; Macro Finished
endm
```

The macro’s parameters are defined as shown in Table G.1.

Sixteen-Bit Numbers

As you’ll probably notice in the investigations and projects, I often find 8-bit numbers insufficient for many practical counters and timers. Sixteen-bit (and larger) numbers can be handled easily in the PIC, even though it is an 8-bit processor (only able to move 8 bits at a time).

So that you don’t have to read through the text to try to find every incident of 16-bit data handling, I’ve tried to list them here for you along with a few extras. These snippets

of code have been made into macros and added to the CD-ROM. These macros are also available as an `include` file that is loaded with the MPLAB files onto your hard disk. While I haven't listed every possible operation, the various routines can be built on to create whatever functions are required.

If you compare the philosophies behind the operations to those in the first edition, you will see that this edition's 16-bit operations have not changed substantially. What has changed is how I have approached the code and use MPASM's built-in features to take care of some operations that I did manually before.

DEFINING 16-BIT NUMBERS

I define 16-bit numbers in a manner similar to that of an 8-bit number and just give them to bytes in the RAM register space. The example below shows how to define an 8-bit variable, followed by two 16-bit variables:

```
RAM equ    12                ; Start of RAM for the PIC16C71

Reg_8 equ   RAM              ; Define the 8 Bit Register
Reg_16 equ  RAM + 1          ; Define the first 16 Bit Register
Reg2_16 equ  RAM + 3         ; Define the 2nd 16 Bit Register
```

or using the `CBLOCK` command in MPASM with the number of bytes in the variable specified:

```
CBLOCK 12                ; Start of RAM for the 16C71
Reg_8                    ; Define the 8 Bit Register
Reg_16:2                 ; Define the first 16 Bit Register
Reg2_16:2                ; Define the 2nd 16 Bit Register
ENDC
```

Note that `Reg2_16` is two addresses above the start of `Reg_16`. This is to give `Reg_16` 2 bytes. I access the high byte of the 16-bit variable by using the name and adding one to it. For example, to access the high byte of `Reg_16`, I use `Reg_16 + 1`.

Personally, I use the `CBLOCK` method exclusively because it avoids the need for recalculating the addresses for the variables if anything is added or taken away.

Note that I haven't included operations on `Stack` variables. It is not very difficult to carry them out, but the resulting operations depend on the implementation used and how the FSR register is set up and handled.

The algorithms and code below are all in two's complement format. This means that you can traverse between positive and negative numbers and allow them to interact with each other easily. It should be noted that if a positive number becomes greater than $32,000 - 1$, it will become negative as far as these routines are concerned. Care must be taken to make sure that the range limits are not exceeded; otherwise, the values will change.

INCREMENTS AND DECREMENTS

Incrementing a 16-bit value is very simple:

```
incf  Reg, f           ; Increment the Low byte
btfsc STATUS, Z        ; Do we have Zero (Multiple of 256)?
    incf Reg + 1, f     ; Increment High byte (if necessary)
```

If a PIC18 is used, the `infsnz` instruction to simplify the 16-bit increment by one instruction:

```
infsnz Reg, f          ; Increment "Reg's" Low Byte and Skip
incf  Reg + 1, f        ; High Byte Increment if Result is Not
; Equal to Zero
```

The decrement of a 16-bit value isn't quite so simple:

```
movf  Reg, f           ; Set "Z" if LOW "Reg" == 0
btfsc STATUS, Z
    decf Reg + 1, f     ; If Low byte is Zero, Decrement High
decf  Reg, f
```

Because the `decf` operation only sets the zero flag on completion, there is no way of knowing whether or not a borrow of the high byte is required after decrementing the low byte. By testing the low byte against zero before decrementing, the need for decrementing the high byte can be ascertained and carried out if required.

No improvement that is similar to the 16-bit increment is available for 16-bit decrement in the PIC18 devices.

ADDITION/SUBTRACTION OF CONSTANTS

Addition and subtraction of 16-bit variables to constants can be done in a very similar manner. There is one trick, however, and this is to always do the high byte first.

The reason for doing the operation on the high byte first is to ensure the destination has the final result, and if the low byte requires a carry or a borrow, the high byte can be incremented or decremented, respectively, easily without having to add extra instructions to do the high byte.

Thus, for adding a constant to a value, that is,

`Reg = Reg + 0x1234`

the following code is used:

```
movlw HIGH 0x1234      ; Add the high byte first
addwf Reg + 1, f
movlw LOW 0x1234        ; Add the Low Byte Next
```

```

addwf Reg, f
btfsc STATUS, C           ; Don't Inc high byte if carry Reset
    incf Reg + 1, f

```

This code can be cut down according to whether or not the high or low byte of the constant is equal to zero.

Note that I have used the `LOW` and `HIGH` directives to determine the appropriate byte for the operation. These directives are used throughout this appendix as what I consider the correct way to access 16-bit constant values a byte at a time.

For the PIC18, the `addwfc` instruction can be used to simplify the operations and eliminate the need for checking the status of the carry flag. For the addition above, the PIC18 code is

```

movlw LOW 0x1234           ; Add Low Byte First
addwf Reg, f
movlw HIGH 0x1234          ; Add High Byte Next
addwfc Reg + 1, f

```

This improvement is available in all the 16-bit additions.

The corresponding subtraction, that is,

`Reg = Reg - 0x1234`

looks like

```

movlw HIGH 0x1234          ; Subtract the High Byte First
subwf Reg + 1, f
movlw LOW 0x1234           ; Subtract the Low Byte Next
subwf Reg, f
btfss STATUS, C            ; Don't Dec high byte if carry Set
    decf Reg + 1, f

```

Again, the enhanced instructions in the PIC18 can be used to simplify this 16-bit operation. Using the `subwfb` instruction, the 16-bit subtraction can be simplified to

```

movlw LOW 0x1234           ; Subtract the Low Byte First
bsf STATUS, C              ; Don't pass any "Borrow"
subwfb Reg, f               ; Reg = Reg - w - !C
movlw HIGH 0x1234
subwfb Reg + 1, f           ; Reg + 1 = Reg + 1 - w - !C

```

Using the `subwfb` instruction, not only is the 16-bit subtraction operation's size decreased by one instruction, but it also is easier to read.

The `addwfc` and `subwfb` enhancements can be used in all the 16-bit addition and subtraction operations given below. When using these instructions, follow the same format of finding the least significant byte's result, followed by the most significant byte's

result, which is opposite to how the operations are carried out in low-end and mid-range PIC microcontrollers.

If you are adding and subtracting to a 16-bit variable and storing the result in another variable, then basically the same code can be used, as I've shown below. The difference between the two methods is that rather than specifying the `Source` destination in the `addwf/subwf` instructions, the `w` register is specified, and the result is stored in the destination, that is,

Destination = Source + 0x5678

and will look like this:

```
movlw HIGH 0x5678           ; Add High Byte First
addwf Source + 1, w
movwf Destination + 1, f    ; Store Result in Destination
movlw LOW 0x5678           ; Add Low Byte Next
addwf Source, w
movwf Destination, f        ; Store Result
btfsc STATUS, C             ; Is the Carry Flag Set?
    incf Destination + 1, f ; Yes, Increment High Byte
```

Subtraction with the result being stored somewhere else is carried out exactly the same way.

ADDITION/SUBTRACTION OF OTHER VARIABLES

Addition of a 16-bit variable to another 16-bit variable is similar to that of adding a constant to a 16-bit variable. If the destination is the same as one of the values, such as

$a = a + b$

the code looks like this:

```
movf b + 1, w               ; Add the High Bytes
addwf a + 1, f
movf b, w                   ; Add the Low Bytes
addwf a, f
btfsc STATUS, C             ; Add the Carry to High Byte
    incf a + 1, f
```

If the destination is different from both values to be added, that is,

$c = a + b$

the code is changed to save the sums in the `w` register and then store them in `c` like

```
movf a + 1, w               ; Add the High Bytes
addwf b + 1, w
```

```

movwf c + 1
movf a, w                ; Add the Low Bytes
addwf b, w
movwf c
btfsc STATUS, C          ; Increment due to Carry
incf c + 1

```

Subtraction is carried out in the same way, but care must be taken to ensure that the subtracting register is kept straight (something that is less of an issue with addition). If you want to do the following statement:

$c = a - b$

you would use the code

```

movf b + 1, w            ; Get Value to be subtracted
subwf a + 1, w           ; Do the High Byte
movwf c + 1
movf b, w                ; Get the Value to be Subbed
subwf a, w
movwf c
btfss STATUS, C          ; Look for the Carry
decf c + 1

```

OTHER OPERATIONS ON CONSTANTS AND VARIABLES

Doing other operations (bitwise or whatever) on 16-bit values can use the code shown above as a base. The big difference between it and the code above is that you don't have to worry about carrying values.

For example, ANDing a 16-bit variable with 0x0A5A5 would be done like

```

movlw 0xA5                ; Get Value for ANDING
andwf Reg + 1, f          ; Do the High Byte
andwf Reg, f              ; Do the Low Byte

```

And this follows on for the other types of operations (i.e., with another 16-bit variable or with a different destination).

There is one difference, however, and that has to do with rotating 16-bit values. Rotating must be carried out in such a way that the carry flag is always correct for the shift. This means that the carry flag first must be cleared (to put a 0 in the bit getting the carry flag). Next, the first rotate should be selected in such a way that the second will have a valid carry flag (i.e., holding the value to be transferred from the first register).

For example, to shift left, use

```

bcf STATUS, C            ; Clear the Carry Flag for new bit
rlf Reg, f               ; Shift the Low Byte
rlf Reg + 1, f           ; Shift High Byte with Low Carry

```

and to shift right, use

```
bcf STATUS, C           ; Clear Carry Flag for the New bit
rrf Reg + 1, f           ; Shift down the High Byte
rrf Reg, f               ; Shift Low Byte with Valid Carry
```

COMPARISONS WITH 16 BIT VARIABLES

Comparisons involving 16-bit variables require that the comparison value (or register) be subtracted from the register to be checked. The results of this will then tell you what is going on with the condition. I use the same code as shown above, save the result in temporary values, and then look at the result. The subtraction code used for comparing a 16-bit variable with another 16-bit variable is

```
movf Reg2 + 1, w         ; Get the High Byte of the Result
subwf Reg1 + 1, w
movwf _2                 ; Store in a Temporary Register
movf Reg2, w             ; Get the Low Byte
subwf Reg1, w
btfss STATUS, C          ; Decrement High if Necessary
decf _2
```

At the end of this series of instructions, the w register contains Reg2 - Reg1, and _2 contains Reg2HI - Reg1HI with the borrow result of Reg2 - Reg1.

If the variable is to be compared against an immediate value, then the movf instruction would be replaced with movlw and the 2 bytes of the immediate value.

There are six basic conditions that you can look for: equals, not equals, greater than, greater than or equal, less than, and less than or equal. Thus, to discover whether or not I have any of these conditions, I add one of the following code examples.

For equals and not equals, the value in the w register is ORed with _2 to see if the result is equal to zero:

```
iorwf _2, w              ; Is the Result == 0?
```

For equals, add the lines

```
btfss STATUS, Z          ; Execute following Code if == 0
goto Zero_Skip           ; Else, Code != 0, Skip Over
```

For not equals, append

```
btfsc STATUS, Z          ; Execute following if != 0
goto NotZero_Skip        ; Else, Code == 0, Skip Over
```

If a greater than comparison is made (the 16-bit variable is greater than the comparison value), then the result will not be less than zero. Actually, the same code (just with a different bit skip) can be used to test.

For greater than:

```

btfsc _2, 7          ; Not Negative, 16 Bit is Greater
goto NotGreater_Skip ; Else, Skip if Not Greater than
iorwf _2, w          ; Is it Equal to Zero?
btfsc STATUS, z      ; No, It is Greater than
Goto NotGreater_Skip ; Else, if Zero, Not Greater than

```

Note that just the most significant bit of the 16-bit difference is checked. If this bit is set, then the 16-bit variable is less than the comparison. If it is reset, then it is greater than, and you should check to see if the result is not equal to zero (or else it is equal).

For less than:

```

btfss _2, 7          ; Negative, 16 Bit is Less Than
goto NotLess_Skip    ; Else, Skip because Not Less Than

```

To check for greater or equal to, the last three lines of the code checking for greater than are simply erased. To check for less or equal to, the three lines from not equal to are added before the check for less than.

Here is the complete code for compare and skip on Reg1 less than or equal to Reg2:

```

movf Reg2 + 1, w      ; Get the High Byte of the Result
subwf Reg1 + 1, w
movwf _2              ; Store in a Temporary Register
movf Reg2, w          ; Get the Low Byte
subwf Reg1, w
btfss STATUS, C       ; Decrement High if Necessary
decf _2
iorwf _2, w           ; Check for Equal to Zero
btfsc STATUS, Z       ; If Not Zero, Jump Over
goto EqualLess_Skip   ; Equals, Jump to the Code
btfsc _2, 7           ; If Number is Negative, execute
goto EqualLess_Skip   ; Else, Jump Over

```

MULTIPLICATION

For both multiplication and division, repeated addition could be used, but I find that using a scaling routine works much better and is faster. These algorithms test bits and only operate if it is appropriate to do so.

Here is multiplication that requires a separate byte for counting the iterations through Loop:

```

clrf Product
clrf Product + 1

```

```
movlw 16                ; Operating on 16 Bits
movwf BitCount

Loop:                   ; Loop Here for Each Bit

    rrf Multiplier + 1, f    ; Shift the Multiplier down
    rrf Multiplier, f      ; by one

    btfss STATUS, C        ; If the bit is set, add
    goto Skip              ; the Multiplicand to the
                           ; "Product"
    movf Multiplicand + 1, w
    addwf Product + 1, f
    movf Multiplicand, w
    addwf Product, f
    btfsc STATUS, C
    incf Product + 1, f

Skip:                   ; Shift up Multiplicand and
    bcf STATUS, C          ; Loop Around
    rlf Multiplicand, f
    rlf Multiplicand + 1, f

    decfsz BitCount
    goto Loop
```

The code given below is the most efficient way of doing a 16-bit multiply with a 32-bit result. It is not immediately obvious, but it's very clever. Rather than using a 32-bit add each time the shifted data is detected, it provides a 16-bit add (with valid carry) and then shifts the data down.

This code does not change `Multiplicand` but does change `Multiplier`. Note that in the code I use a 32-bit value for `Product` (using a `Product:5` line in the `CBLOCK` variable declare statement).

```
clrf Product + 2        ; "Product" will be the
clrf Product + 3        ; Result of the Operation

movlw 16                ; Operating on 16 Bits
movwf BitCount

Loop:                   ; Loop Here for Each Bit

    rrf Multiplier + 1, f    ; Shift the Multiplier down
    rrf Multiplier, f      ; by one

    btfss STATUS, C        ; If the bit is set, add
    goto Skip              ; the Multiplicand to the
                           ; "Product"
```



```

    clrf Product + 4
    movf Multiplicand + 1, w
    addwf Product + 3, f
    btfsc STATUS, C          ; Make Sure the Carry is Passed
    incf Product + 4, f      ; to the Next Byte
    movf Multiplicand, w
    addwf Product + 2, f
    btfsc STATUS, C
    incfsz Product + 3, f    ; Make Sure Carry is Passed with
    goto $ + 2              ; the Shift
    incf Product + 4, f

Skip:                          ; Shift "Product" Down with
    bcf STATUS, C
    rrf Product + 4, f
    rrf Product + 3, f      ; the Reset Carry from the
    rrf Product + 2, f      ; Multiplier shift down or
    rrf Product + 1, f      ; the result of the sixteen
    rrf Product, f          ; bit addition.

    decfsz BitCount
    goto Loop

```

Both of the multiplication routines shown here will work with positive and negative numbers.

For the PIC microcontrollers that have built-in 8×8 multipliers, the code for 16-bit multiplication uses the techniques taught in high school mathematics for multiplying together to variable factors. Instead of thinking of a 16-bit number as just a contiguous set of 16 bits, I can represent it as two 8-bit numbers with the high byte multiplied by 256 (0x100):

$$A = (Ah * 0x100) + Al$$

Breaking 16 numbers up into this format, I can FOIL ("First, Outside, Inside, Last") the two factors together using the equation

$$\begin{aligned}
 A \times B &= ((Ah * 0x100) + Al) \times ((Bh * 0x100) + Bl) \\
 &= ((Ah * 0x100) \times (Bh * 0x100)) + ((Ah * 0x100) \times Bl) \\
 &\quad + (Al \times (Bh * 0x100)) + (Al \times Bl) \\
 &= (Ah \times Bh * 0x10000) + (Ah \times Bl * 0x100) \\
 &\quad + (Al \times Bh * 0x100) + (Al \times Bl)
 \end{aligned}$$

Multiplying numbers by factors of 256 is accomplished simply by putting them in the next higher significant byte.

Using this formula and the built-in multiplier, 16-bit multiplication with a 32-bit result can be accomplished in the PIC18 using the code

```

    clrf Product + 2          ; Clear the High-Order Bits
    clrf Product + 3

```

```
movf  Al, w           ; Do the "L" Multiplication first
mulwf Bl
movf  PRODL, w        ; Save result
movwf Product
movf  PRODH, w
movwf Product + 1
movf  Al, w           ; Do the "I" Multiplication
mulwf Bh
movf  PRODL, w        ; Save the Most Significant Byte First
addwf Product + 1, f
movf  PRODH, w
addwfc Product + 2, f ; Add to the Last Result
movf  Ah, w           ; Do the "O" Multiplication
mulwf Bl
movf  PRODL, w        ; Add the Lower Byte Next
addwf Product + 1, f
movf  PRODH, w        ; Add the High Byte First
addwfc Product + 2, f
btfsc STATUS, C       ; Add the Carry
incf  Product + 3, f
movf  Ah, w           ; Do the "F" Multiplication
mulwf Bh
movf  PORDL, w
addwf Product + 2, f
movf  PRODH, w
addwfc Product + 3, f
```

DIVISION

The division routine provided here first finds how far the divisor can be shifted up before comparing to the quotient. The `Count` variable in this routine is a 16-bit variable that is used to both count the bits and add to the quotient. `Temp` is an 8-bit temporary storage variable. At the end of the division routine, `Dividend` will contain the remainder of the operation.

```
clrf  Quotient
clrf  Quotient + 1

movlw 1               ; Initialize Count
movwf Count
clrf  Count + 1

StartLoop:             ; Find How Large "Divisor" can
                       ; be
btfsc Divisor + 1, 7   ; If at the "top", then do
goto  Loop             ; the Division

bcf  STATUS, C         ; Shift Count and Divisor Up
rlf  Count, f
```

```

    rlf  Count + 1, f

    rlf  Divisor, f
    rlf  Divisor + 1, f

    goto StartLoop

Loop:                                ; Now, Take Away "Divisor"
                                    ; from "Dividend"
    movf Divisor + 1, w ; If Divisor < Dividend then
    subwf Dividend + 1, w ; Don't Take Away
    movwf Temp
    movf Divisor, w
    subwf Dividend, w
    btfss STATUS, C
    decf Temp, f
    btfsc Temp, 7 ; If "Temp" Negative then
    goto Skip ; Divisor < Dividend

    movwf Dividend ; Save the New Dividend
    movf Temp, w
    movwf Dividend + 1

    movf Count, w ; Add Count to the Quotient
    addwf Quotient + 1, f
    movf Count, w
    addwf Quotient + 1, f ; No Opportunity for Carry

Skip:                                ; Shift Divisor/Count Down
    bcf STATUS, C
    rrf Divisor + 1, f
    rrf Divisor, f

    rrf Count + 1, f ; If Carry Set after Count
    rrf Count, f ; Shift, Finished

    btfss STATUS, C ; If Carry NOT Set, then
    goto Loop ; Process next Bit

```

This division routine is designed to handle only positive numbers—there is not a general algorithm that handles both positive and negative numbers and passes back both the quotient and remainder with the correct polarity efficiently. Along with the problems with negative values and returning the required value, handling zero can cause problems. In many processors, division by zero causes a system fault.

In my experience, I have found that implementing a division routine is very dependent on the expected values. There are some very efficient algorithms for specific divisors, the most obvious being how to divide by multiples of 2 (simply shift the dividend value to the right an appropriate number of times).

A general form for a division routine (using the algorithm shown above) could be that division of the core of the pseudocode is a bit-shift analogous algorithm to multiplication that can handle positive and negative numbers.

```
if (Dividend < 0) {           // Change dividend to positive number
    Dividend = 0 - Dividend;
    dividendneg = 1;          // Mark we have to change it back
} else
    dividendneg = 0;
if (Divisor < 0) { // Repeat with the Divisor
    Divisor = 0 - Divisor;
    divisorneg = 1;
} else
    divisorneg = 0;

Count = 0;                    // Going to Count where division starts
Quotient = 0;                 // Store the Quotient
while (( Divisor & 0x0400 ) != 0) {
// Find the Start of the Division
    Count = Count + 1; // Increment the Number of Bits Shifted
    Divisor = Divisor << 1;
}

while (Count != 0) {          // Now, do the Division
    if (Dividend >= Divisor) { // A subtract can take place
        Quotient = Quotient + 2 ^ Count;
        Dividend = Dividend - Divisor;
    }
    Count = Count - 1;
    Divisor = Divisor >> 1;
}

if (Dividendneg == 1)        // Now, change the values
    if (Divisorneg == 1) {
        Quotient = Quotient;
        Remainder = 0 - Dividend;
    } else {
        Quotient = 0 - Quotient;
        Remainder = 0 - Dividend;
    }
else                          // The Dividend was Positive
    if (Divisorneg == 1) {
        Quotient = 0 - Quotient;
        Remainder = Dividend;
    } else {
        Quotient = Quotient;
        Remainder = Dividend;
    }
}
```

GLOSSARY

Accumulator Register used as temporary storage for an operation's data source and destination. In the PIC microcontroller, this is known as the *w* register.

Active components Generally integrated circuits and transistors. Active components require external power to operate.

ADC Acronym for *analog-to-digital converter*. Hardware devoted to converting the value of a dc voltage into a digital representation. See *DAC*.

Address The location of a register, RAM byte, or instruction word within its specific memory space.

ALU Acronym for *arithmetic logic unit*. The circuit within a computer processor that carries out mathematical operations.

Amps Measure of current. One amp is the movement of one coulomb of electrons in one second.

Analog A quantity at a fractional value rather than a binary value: a one or a zero. Analog voltages are the quantity most often measured.

AND Logic gate that outputs a 1 when all inputs are a 1.

Argument A user-specified value for a subroutine or macro. An argument can be a numeric value, a string, or a pointer depending on the application. See *Parameter*.

Array Collection of variables that can be addressed by an arbitrary index.

ASCII American Standard Character Interchange Interface. Bit-to-character representation standard most used in computer systems.

ASCIIZ A string of ASCII characters ended by a null (0x000) byte.

Assembler A computer program that converts assembly-language source to object code. See *Cross-assembler*.

Assembly language A set of word symbols used to represent the instructions of a processor. Along with a primary instruction, there are parameters that are used to specify values, registers, or addresses.

Asynchronous serial Data sent serially to a receiver without clocking information. Instead, data-synching information for the receiver is available inside the data packet or as part of each bit.

Bare board See *Raw card*.

BCD Binary code decimal. Using 4 bits to represent a decimal number (0 to 9).

BGA Acronym for *ball grid array*. A chip solder technology that provides connection from a chip to a bare board via a two-dimensional grid of solder balls.

Binary numbers Numbers represented as powers of 2. Each digit is two raised to a specific power. Binary can be represented in the forms: 0b0nnnn, B'nnnn', or %nnnn, where nnnn is a multidigit binary number consisting of 1s and 0s.

Bipolar logic Logic circuits made from bipolar transistors (either discrete devices or integrated onto a chip).

Bit banging Simulating interface functions with code.

Bit mask A bit pattern that is ANDed with a value to turn off specific bits.

Boost power supply A type of switch-mode power supply that is designed to convert electrical energy from one voltage to a higher one.

Bounce Spurious signals in a changing line. Most often found in mechanical switch closings.

Buck power supply A type of switch-mode power supply that is designed to convert electrical energy from one voltage to a lower one.

Burning See *Programming*.

Bus An electrical connection between multiple devices, each using the connection for passing data.

Capacitor Device used for storing electrical charge. Often used in microcontroller circuits for filtering signals and input power by reducing transient voltages. The different types include ceramic disk, polyester, tantalum, and electrolytic. Tantalum and electrolytic capacitors are polarized.

Ceramic resonator A device used to provide timing signals to a microcontroller. Generally, ceramic resonators are cheaper and more robust than a crystal but have poorer frequency accuracy. See *Crystal*.

Character Series of bits used to represent an alphabetic, numeric, control, or other symbol or representation. See *ASCII*.

Chip package The method by which a chip is protected from the environment (usually encased in either ceramic or plastic) with wire interconnects to external circuitry. See *PTH* and *SMT*.

CISC Acronym for *complex instruction set computer*. A computer architecture that has a large number of very complete instructions rather than a few short instructions. See *RISC*.

Clock A repeating signal used to run a processor's instruction sequence.

Clock cycle The operation of a microcontroller's primary oscillator going from a low voltage to a high voltage and back again. This is normally referenced as the speed at which the device runs. Multiple clock cycles may be used to make up one instruction cycle.

CMOS logic Logic circuits made from N-channel and P-channel MOSFET (metal oxide silicon field effect transistors) devices (either discrete devices or integrated onto a chip).

Comparator A device that compares two voltages and returns a logic 1 or 0 based on the relative values.

Compiler A program that takes a high level language source file and converts it to either assembly-language code or object code for a microcontroller.

Concatenate Joining two pieces of data together to form a single contiguous piece of data.

Contiguous When a set amount of data is placed altogether in memory and can be accessed sequentially using an index pointer, it is said to be contiguous. Data is noncontiguous if it is placed in different locations that cannot be accessed sequentially.

Control store See *Program memory* or *Program store*.

Constant Numeric value used as a parameter for an operation or instruction. This differs from a variable value that is stored in a RAM or register memory location.

CPU Central processing unit; what I refer to as the microcontroller's processor.

Cross-assembler A program written to take assembly-language code for one processor and convert it to object code while working on an unrelated processor and operating system. See *Assembler*.

Crystal Device used for precisely timing the operation of a microcontroller. See *Ceramic resonator*.

Current The measurement of the number of electrons that pass by a point in a second each second. The units are amps, which are coulombs per second.

D-shell connectors A style of connector often used for RS-232 serial communications as well as other wiring protocols. The connector is D-shaped to provide a method of polarizing the pins and ensuring that the internal pins are connected properly.

DAC Acronym for *digital-to-analog converter*. Hardware designed to convert a digital representation of an analog dc voltage into that analog voltage. See *ADC*.

DCE Acronym for *data communications equipment*, the RS-232 standard by which modems are wired. See *DTE*.

Debounce Removing spurious signals in a noisy input.

Debugger A debugger is an application used by a programmer to find the problems in an application. This program is normally run on the application's target system. The PIC microcontroller is unique because some part numbers provide built-in debugger features.

Decimal numbers Base 10 numbers used for constants. These values normally are converted into hex or binary numbers for the microcontroller.

Decoupling capacitor Capacitor placed across V_{cc} and ground of a chip to reduce the effects of increased/decreased current draws from the chip.

Demultiplexor Circuit for passing data from one source to an output specified from a number of possibilities. See *Demux*, *Multiplexor*, and *Mux*.

Demux Abbreviation for *demultiplexor*.

Digital A term used to describe a variety of logic families where values are either high (1) or low (0).

Driver Any device that can force a signal onto a net. See *Net* and *Receiver*.

DTE Acronym for *data terminal equipment*, the RS-232 standard connection on a PC's serial port. See *DCE*.

Duty cycle In a pulse wave modulated digital signal, the duty cycle is the fraction of time the signal is high over the total time of the repeating signal.

Edge-triggered Logic that changes based on the change of a digital logic level. See *Level-sensitive*.

Editor Program located on your development system that is used to modify application source code.

EEPROM Acronym for *electrically erasable programmable memory*. Nonvolatile memory that can be erased and reprogrammed electrically (i.e., it doesn't require the ultraviolet light like an EPROM). See *Flash*.

Emulator Replacement electric circuit connected to the development system that will allow the application to be executed under the developer's direct control, allowing observation of how it works and permitting some changes to the application ("what if?").

EPROM Acronym for *erasable programmable read only memory*. Nonvolatile memory that can be programmed electrically and later erased using ultraviolet light.

Event-driven programming Application code that waits for external events to process before executing.

External memory RAM or ROM memory connected through a standard bus (SPI, I2C, or the PIC18's external device bus).

FIFO Acronym for *first in, first out*. Memory that will retrieve data in the order in which it was stored. See *LIFO*.

Flash A type of EEPROM. Flash is normally erased electrically in blocks instead of as individual memory locations.

Flip-flop A basic memory cell that can be loaded with a specific logic state and read back. The logic state will be stored as long as power is applied to the cell.

Floating The term used to describe a pin that has been left unconnected and is floating relative to ground.

Floating-point numbers The term used to describe real numbers in a computer system.

Fosc PICmicro clock frequency.

Frequency The number of repetitions of a signal that can take place in a given period of time (typically 1 second). See *Period* and *Hertz*.

FTP File Transfer Protocol. A method of transferring files to/from the Internet.

Functions A subroutine that returns a parameter to the caller.

Fuzzy logic A branch of computer science in which decisions are made on partially on data rather than on or off data such as digital logic uses. These decisions are often made for controlling physical and electronic systems. See *PID*.

Ground Negative voltage to microcontroller/circuit. Also referred to as *Vss*.

GUI Acronym for *graphical user interface* (often pronounced "goeey"). A GUI is used, along with a graphical operating system (such as Microsoft Windows), to provide a simple, consistent computer interface for users that consists of a screen, a keyboard, and a mouse.

Harvard architecture Computer processor architecture that interfaces with two memory subsystems, one for instructions (program memory) memory and one for variable memory and I/O registers. See *Princeton architecture*.

Hertz A unit of measurement of frequency. One hertz (or Hz) means that an incoming signal is oscillating once per second.

Hex File type produced by assembler/compiler/linker that can be programmed into a microcontroller.

Hex numbers A value from 0 to 15 that is represented using 4 bits or the characters 0 through 9 and A through F.

High level language A set of English (or other human language) statements that have been formatted for use as instructions for a computer. Some popular high level languages used for microcontrollers include C, BASIC, Pascal, and Forth.

Horizontal synch A pulse used to indicate the start of a scan line in a video monitor or TV set.

Hysteresis Characteristic response to input that causes the output response to change based on the input. Typically used in microcontroller input pins to debounce input signals.

Hz Abbreviation for *hertz*.

ICD Acronym for *in-circuit debugger* and normally known as MPLAB ICD 2. This interface allows the built-in debugger hardware in a PIC microcontroller to be accessed.

ICSP Acronym for *in-circuit serial programming*, a connector, voltage, and signal standard used for programming PIC microcontrollers.

Index register An 8- or 16-bit register that can have its contents used to point to a location in variable storage, control store, or the microcontroller's register space. See *Stack pointer*.

Inductor Wire wrapped around some kind of form (metal or plastic) to provide a magnetic method of storing energy. Inductors are often used in oscillator and filtering circuits.

Infrared A wavelength of light (760 nm or longer) that is invisible to the human eye. Often used for short-distance communications.

Interpreter A program that reads application source code and executes it directly rather than compiling it.

Interrupt An event that requires that the microcontroller's processor stop what it is doing and respond.

Instruction A set of bits (converted from source code by an assembler or compiler) that are executed by a computer processor to perform basic functions.

Instruction cycle The minimum amount of time needed to execute a basic function in a microcontroller. One PIC microcontroller instruction cycle typically takes several clock cycles. See *Clock cycles*.

In-system programming The ability to program a microcontroller's control store while the device is in the final application's circuit without having to remove it. See *ICSP*.

I/O space An address space totally devoted to providing access to I/O device control registers.

I²C Acronym for *inter-intercomputer communication*, often pronounced as “eye-two-see” or “eye-squared-see.” A synchronous serial network protocol designed to allow microcontrollers to communicate with peripheral devices and each other.

k Ω Abbreviated term for stating a resistor value in thousands of ohms.

kHz Abbreviation for units used to measure frequency in thousands of cycles per second.

Label An identifier used within a program to denote the address location of a control store or register address. See *Variable*.

Latency The time or cycles required for hardware to respond to a change in input.

LCD Acronym for *liquid-crystal display*. A device used for outputting information from a microcontroller.

Lead-free Solder material developed to replace traditional tin-lead solder alloys.

LED Acronym for *light-emitting diode*. A diode (rectifier) device that will emit light of a specific frequency when current is passed through it. When used with microcontrollers, LEDs are usually wired with the anode (positive pin) connected to Vcc and the microcontroller I/O pin connected to the cathode and sinking current (using a series resistor) to turn on the LED. In typical LEDs in hemispherical plastic packages, the flat side (which has the shorter lead) is the cathode.

Level conversion The process of converting logic signals from one family to another.

Level-sensitive Logic that changes based on the state of a digital logic signal. See *Edge-triggered*.

LIFO Acronym for *last in, first out*. Type of memory in which the most recently stored data will be the first retrieved. See *FIFO*.

Linker A software product that combines object files into a final program file that can be loaded into a microcontroller.

Lithium ion A type of rechargeable battery type/chemistry. The typical full-voltage cell voltage for a lithium ion (Li ion) is 4.2 V.

Logic analyzer A tool that will graphically show the relationship of the waveforms of a number of different pins.

Logic gate A circuit that outputs a logic signal based on input logic conditions.

Logic probe A simple device used to test a line for either being high, low, transitioning, or in a high-impedance state.

Macro A programming construct that replaces a string of characters (and parameters) into a previously specified block of code or information.

Manchester encoding A method for serially sending data that does not require a common clock.

Mask programmable ROM A method of programming a memory that takes place at final assembly of a microcontroller. When the aluminum traces of a chip are laid down, a special photographic mask is made to create wiring that will result in a specific program being read from a microcontroller's control store.

Master In microcontroller and external device networking, a master is a device that initiates and optionally controls the transfer of data. See *Multimaster* and *Slave*.

Matrix keyboard A set of pushbutton switches wired in an *X/Y* pattern to allow button states to be read easily. Also known as a *matrix keypad*.

Matrix keypad A set of pushbutton switches wired in an *X/Y* pattern to allow button states to be read easily. Also known as a *matrix keyboard*.

MCU Abbreviation for *microcontroller*.

Memory Circuit designed to store instructions or data.

Memory array A collection of flip-flops arranged in a matrix format that allows consistent addressing.

Memory-mapped I/O A method of placing peripheral registers in the same memory space as RAM or variable registers.

MHz This is an abbreviation for the units used to measure frequency in millions of cycles per second.

Microwire A synchronous serial communications protocol.

MIPS Acronym for *millions of instructions per second*. This acronym is also known as the *misleading indicator of performance* and should not be considered when deciding which microcontroller or processor to use for an application.

Monitor A program used to control the execution of an application inside a processor.

MPU Abbreviation for *microprocessor*.

ms One-thousandth of a second (0.001 s). See *ns* and μ s.

Multimaster A microcontroller networking philosophy that allows multiple masters on a network bus, each able to initiate data transfers. See *Slave*.

Multiplexor Device for selecting and outputting a single stream of data from a number of incoming data sources. See *Demultiplexor*, *Demux*, and *Mux*.

Mux Abbreviation for *multiplexor*.

Negative active logic A type of logic where the digital signal is said to be asserted if it is at a low (0) value. See *Positive active logic*.

Nesting Placing subroutine or interrupt execution within the execution of other subroutines or interrupts.

Net A technical term for the connection of device pins in a circuit. Each net consists of all the connections to one device pin in a circuit and must have one driver and at least one receiver.

NiCad Abbreviation for *nickel-cadmium batteries*. These batteries are rechargeable, although typically they provide 1.2 V per cell output compared with 1.5 to 2.0 V for standard dry or alkaline radio batteries.

NMOS logic Digital logic where only N-channel MOSFET transistors are used.

Noise High-frequency variances in a signal line that are caused by switch bounce or electrical signals picked up from other sources.

NOT Logic gate that inverts the state of the input signal (1 NOT is 0).

ns One-billionth of a second (0.000000001 s). See μ s and ms.

NTSC Acronym for the *National Television Standards Committee*, the standards organization responsible for defining the TV signal format used in North America.

.obj File type produced after assembly or high level language compilation containing the hex values (op codes) that make up a processor's instructions. An object file either can be loaded directly into a microcontroller or multiple object files can be linked together to form an executable file that is loaded into a microcontroller's control store. See *Linker*.

Octal numbers A method of representing numbers as the digits from 0 to 7. This method of representing numbers is not used widely, although some high level languages, such as C, have made it available to programmers.

One's complement The result of XORing a byte with 0xFF, which will invert each bit of a number. See *Two's complement*.

Op codes The hex values that make up the processor instructions in an application.

Open-collector/drain output An output circuit consisting of a single transistor that can pull the net it is connected to ground.

OR Basic logic gate; when any input is set to a 1, a 1 is output.

ORT Acronym for *ongoing reliability testing*. A continuing set of tests that are run on a manufactured product during its life to ensure that it will be as reliable as originally specified.

Oscillator A circuit used to provide a constant-frequency repeating signal for a microcontroller. This circuit can consist of a crystal, ceramic resonator, or resistor-capacitor network for providing the delay between edge transitions. The term is also used for a device that can be wired to a microcontroller to provide clocking signals without having to provide a crystal, caps, and other components to the device.

Oscilloscope An instrument that is used to observe the waveform of an electrical signal. The two primary types of oscilloscopes in use today are the analog oscilloscope, which writes the current signal onto the phosphors of a CRT. The other common type of oscilloscope is the digital storage oscilloscope, which saves the analog values of an incoming signal in RAM for replaying on either a built-in CRT or a computer connected to the device.

OTP One-time programmable. This term generally refers to a device with EPROM memory encased in a plastic package that does not allow the chip to be exposed to ultraviolet light. Note that EEPROM devices in a plastic package also may be described as OTP when they can be erased electrically and reprogrammed.

Parallel Passing data between devices with all the data bits being sent at the same time on multiple lines. This is typically much faster than sending data serially.

Parameter A user-specified value for a subroutine or macro. A parameter can be a numeric value, a string, or a pointer depending on the application. See *Argument*.

Passive components Resistors, capacitors, inductors, diodes, or other components that do not require a separate power source to operate.

PCA Acronym for *printed-circuit assembly*. A bare board with components (both active and passive) soldered onto it.

PCB Acronym for *printed-circuit board*. See *Raw card*.

.pdf Files suitable for viewing with Adobe Postscript.

Period The length of time that a repeating signal takes to go through one full cycle. The reciprocal of *frequency*.

PID Acronym for *parallel integrating differential*. A classic method of controlling physical and electronic systems. See *Fuzzy logic*.

Ping The operation of sending a message to a device to see if it is operating properly.

Pod The term given for a hardware circuit that interfaces some piece of equipment to another circuit.

Poll A programming technique in which a bit (or byte) is checked repeatedly until a specific value is found.

Pop The operation of taking data off of a stack memory.

Positive active logic Logic that becomes active when a signal becomes high (1). See *Negative active logic*.

ppm Acronym for *parts per million*, used as a unit of measurement for small quantities in a large population. An easy way of calculating the ppm of a value is to divide the value by the total number of samples or opportunities and multiply by 1 million. One percent is equal to 10,000 ppm; 10 percent is equal to 100,000 ppm.

Princeton architecture Computer processor architecture that uses one memory subsystem for instructions (control store) memory, variable memory, and I/O registers. See *Harvard architecture* and *Von Neumann*.

Program counter A counter within a computer processor that keeps track of the current program execution location. This counter can be updated by the counter and have its contents saved/restored on a stack.

Programming Loading a program into a microcontroller control store; also referred to as *burning*.

Program memory Also known as *control store* or *program store*. Memory (usually nonvolatile) devoted to saving the application program when the microcontroller is powered down.

Program store Also known as *program memory* or *control store*. Memory (usually nonvolatile) devoted to saving the application program when the microcontroller is powered down.

PROM Acronym for *programmable read-only memory*. An array of fuses built into a memory device, each of which can be electrically blown to change their state.

PTH Acronym for *pin through hole*. Technology in which the pins of a chip are inserted into holes drilled into an FR4 printed-circuit card before soldering.

Pull-down A resistor (typically 100 to 500 Ω) that is wired between a microcontroller pin and ground. See *Pull-up*.

Pull-up A resistor (typically 1 to 100 k Ω) that is wired between a microcontroller pin and Vcc. A switch pulling the signal at the microprocessor pin may be used to provide user input. See *Pull-down*.

Push The operation of putting data onto a stack memory.

PWA Acronym for *printed wiring assembly*. A PCB with components soldered to it.

PWB Acronym for *printed wiring board*. See *Raw card*.

PWM Acronym for *pulse width modulation*. A digital output technique where a single line is used to output analog information by varying the length of time a pulse is active on the line.

RAM Acronym for *random access memory*. Memory that can be read from or written to at anytime. In microcontrollers, virtually all RAM is static RAM (SRAM), which means that data is stored within it as long as power is supplied to the circuit. Dynamic RAM (DRAM) is very rarely used in microcontroller applications. EEPROM may be used for nonvolatile RAM storage.

Raw card PCB material with copper traces attached to it that allow components to be interconnected. Also known as *PCB* and *bare board*.

RC Acronym for *resistor/capacitor* network that is used to provide a specific delay for a built-in oscillator or reset circuit.

Receiver A device that senses the logic level in a circuit. A receiver cannot drive a signal. See *Driver* and *Net*.

Recursion A programming technique where a subroutine calls itself with modified parameters to carry out a task. This technique is not recommended for microcontrollers that may have a limited stack.

Register A memory address devoted to saving a value (like RAM) or providing a hardware interface for a processor.

Relocatable Code written or compiled in such a way that it can be placed anywhere in the control store memory map after assembly and run without any problems.

Resistor A device used to limit current in a circuit.

Resistor ladder A circuit that is comprised of a number of resistors that can be selected to provide varying voltage divider circuits and output differing analog voltages.

Reset Placing a microcontroller in a known state before allowing it to execute.

RISC Acronym for *reduced instruction set computer*. This is a philosophy in which the operation of a computer is sped up by reducing the operations performed by a processor to the absolute minimum for application execution and making all resources accessible by a consistent interface. The advantages of RISC include faster execution time and a smaller instruction set. See *CISC*.

ROM Acronym for *read-only memory*. This type of memory typically is used for control store because it cannot be changed by a processor during the execution of an application. Mask programmable ROM is specified by the chip manufacturer to build devices with specific software as part of the device and cannot be programmed in the field.

Rotate A method of moving bits within a single or multiple registers. No matter how many times a rotate operation or instruction is carried out, the data in the registers will not be lost. See *Shift*.

RS-232 An asynchronous serial communications voltage standard with normal logic levels for a 1 of -12 V and for a 0 of $+12\text{ V}$.

RS-485 A differential pair, TTL voltage level communications system.

RTOS Acronym for *real-time operating system*, which is a program that controls the operation of an application.

Scan The act of reading through a row of matrix information for data rather than interpreting the data as a complete unit.

Serial Passing multiple bits using a serial line one at a time. See *Parallel*.

Servo A device that converts an electric signal into mechanical movement. Radio control modeler's servos have their position specified by a 1- to 2-ms pulse every 20 ms.

Sharp Alternative term for #, which may be known as the pound sign or by its character name *octothorpe*.

Shift A method of moving bits within a single or multiple registers. After a shift operation, bits are lost. See *Rotate*.

Simulator A program used to debug applications by simulating the operation of the microcontroller.

Slave In microcontroller networking, a device that does not initiate communications but does respond to the instructions of a master.

SMD Acronym for *surface-mounted devices*, which are soldered to the surface of a PCB instead of having leads that pass through the PCB and are soldered from the other side.

SMT Acronym for *surface-mount technology*. Technology in which the pins of a chip are soldered to the surface of a printed-circuit card and not through holes.

Software The term used for the application code that is stored in a microcontroller's program memory. Some references may use the term *firmware* for this code.

Source code Human-readable instructions used to develop an application. Source code is converted by a compiler or assembler into instructions the processor can execute and store into a .hex file.

SPI A synchronous serial communications protocol.

Splat Alternative term for asterisk (*). Easier to say and spell and funnier than *asterisk*.

SRAM Acronym for *static random-access memory*. A type of memory array that will not lose its contents while power is applied.

Stack LIFO memory used to store program counter and other context register information.

Stack pointer An index register available within a processor that is used for storing data and updating itself to allow the next operation to be carried out with the index pointing to a new location.

State analyzer A tool used to store and display state data on several lines. Rather than requiring a separate instrument, this is often an option available in many logic analyzers.

State machine A programming technique that uses external conditions and state variables for determining how a program is to execute.

String Series of ASCII characters saved sequentially in memory. When ended with a NUL (0x00), it is known as an *ASCIIZ string*.

Subroutines A small application program devoted to carrying out one task or operation. Usually called repeatedly by other subroutines or the application mainline.

Synchronous serial Data transmitted serially along with a clocking signal that is used by the receiver to indicate when the incoming data is valid.

Task A small, autonomous application that is similar in operation to a subroutine but can execute autonomously to other application tasks or mainline.

Timer A counter incremented by either an internal or an external source. Often used to time events rather than counting instruction cycles.

Traces Electrical signal paths etched in copper in a printed-circuit card.

Transistor An electronic device that controls current flow.

Two's complement A method for representing positive and negative numbers in a digital system. To convert a number to a two's complement negative, it is complemented (converted to one's complement) and incremented.

UART Acronym for *universal asynchronous receiver/transmitter*. Peripheral hardware inside a microcontroller used to asynchronously communicate with external devices. See *USART* and *Asynchronous serial*.

USART Acronym for *universal synchronous/asynchronous receiver/transmitter*. Peripheral hardware inside a microcontroller used to synchronously (using a clock signal either produced by the microcontroller or provided externally) or asynchronously communicate with external devices. See *UART* and *Synchronous serial*.

μs One-millionth of a second (0.000001 s). See *ns* and *ms*.

UV light Abbreviation for *ultraviolet light*. Light at shorter wavelengths than the human eye can see. UV light sources are often used with windowed microcontrollers with EPROM control store for erasing the contents of the control store.

Variable A label used in an application program that represents an address that contains the actual value to be used by the operation or instruction. Variables are normally located in RAM and can be read from or written to by a program.

Vcc Positive power voltage applied to a microcontroller/circuit. Generally 2.0 to 6.0 V depending on the application. Also known as *Vdd*.

Vdd See *Vcc*.

Vertical synch A signal used by a monitor or TV set to determine when to start displaying a new screen (field) of data.

Vias Holes in a PCB.

Volatile RAM is considered to be volatile because when power is removed, the contents are lost. EPROM, EEPROM, Flash, and PROM are considered to be non-volatile because the values stored in the memory are saved, even if power is removed.

Voltage The amount of electrical force placed on a charge.

Voltage regulators A circuit used to convert a supply voltage into a level useful for a circuit or microcontroller.

Volts Unit of voltage.

Von Neumann Chief scientist responsible for the Princeton architecture.

Vss See *Ground*.

Wait states Extra time added to an external memory read or write.

Watchdog timer Timer used to reset a microcontroller on overflow. The purpose of the watchdog timer is to return the microcontroller to a known state if the program begins to run errantly (or amok).

Wattage Measure of power consumed. If a device requires 1 A of current with a 1-V drop, 1 W of power is being consumed.

Word The basic data size used by a processor. In the PIC microcontroller families, the word size is 8 bits.

XOR A logic gate that outputs a 1 when the inputs are at different logic levels.

ZIF Acronym for *zero insertion force*. ZIF sockets will allow the plugging/unplugging of devices without placing stress on the device's pins.

.zip files Files combined together and compressed into a single file using the PKZIP program by PKWARE, Inc.

This page intentionally left blank

INDEX

This page intentionally left blank

[] (square brackets), 1109	+= (addition), 498, 1128	<<= (shift left), 498, 1128
{ } (braces), 1109	, (comma), 1091	<> (not equal to), 94, 1092,
! (not equal to), 498, 1127	– (negation), 496, 1127	1111
!= (not equal to), 498, 507,	– (subtraction), 92, 496,	> (greater than), 94, 498,
1111, 1127	1092, 1110, 1127	507, 1092, 1111, 1127
“ “ (ASCII string), 1092	— (decrement), 1127	>= (greater than or equal to),
“ (double quotes), 1104	-= (subtraction), 498, 1128	94, 498, 507, 1092,
# (pound character), 1138	-> (structure pointer), 410	1111, 1127
\$ (current address), 123, 367,	.\$\$ files, 113	>> (shift right), 92, 496,
368, 377, 379, 496, 497	/ (division), 92, 496, 1092,	1110, 1127
\$ (string variable), 1092	1110, 1127	>>= (shift right), 498, 1128
% (modulus), 496, 1127	/*...*/ (comment), 500	\ (backslash), 500
%= (modulus), 498, 1128	// (modulus), 92, 1110	\” (double quote), 1130
& (ampersand), 410, 1124	/= (division), 498, 1128	\’ (single quote), 1130
& (AND), 92, 496, 507,	: (colon), 123, 379, 1101	\\ (backslash), 1130
1110, 1127	; (semicolon), 127, 1092,	\a (alarm/bell), 1130
&& (AND), 497, 498, 1111,	1101, 1107	\b (backspace), 1130
1127	= (assignment), 497	\ddd (octal number), 1130
&/ (inverted bitwise value	= (equal to), 94, 507, 1092	\f (form feed), 1130
AND), 1110	== (equal to), 498, 1111,	\n (line feed), 1130
&= (AND), 498, 1128	1127	\r (carriage return), 1130
‘ (single quote), 1101	? (question mark), 1090,	\t (horizontal tab), 1130
* (indirection), 1127	1126	\v (vertical tab), 1130
* (multiplication), 92, 496,	? (variable display com-	\xdd (hexadecimal num-
1092, 1110, 1127	mand), 92	ber), 1130
* (splat), 410, 1225	@ (at character), 1104	~ (complement), 496, 1110
** (significant product), 1110	< (less than), 94, 498, 507,	^ (exponentiation), 1092
*/ (multiplication), 920	1111, 1127	^ (XOR), 92, 496, 1110,
*= (multiplication), 498,	<= (less than or equal to), 94,	1127
1128	498, 507, 1092, 1111,	^= (XOR), 498, 1128
+ (addition), 92, 496, 1092,	1127	^^ (XOR), 1111
1110, 1127	<< (shift left), 92, 496, 1110,	(OR), 92, 496, 507, 1110,
++ (increment), 1127	1127	1127

- / (inverted bitwise value OR), 1110
- |= (OR), 1128
- |^ (negation), 92, 1127
- |^/ (inverted bitwise value XOR), 1110
- || (OR), 497, 498, 1111, 1127
- ABS (BASIC), 1091
- ABS (PICBASIC), 1110
- Absolute offset, 1124
- Absolute time, 1116
- Absolute value, 1091, 1110, 1133–1134
- AC electronics formulas, 1064–1065
- Acceleration due to gravity, 1061
- Access bank file registers, 543, 545
- Access data EEPROM, 93
- Accumulator, 10, 247, 1213
- Active components, 1213
- ADC conversion, 1142
- ADC hardware, 1139, 1142
- ADC operation, 1146, 1147
- ADCLess.asm application, 798–804
- ADCON0 register, 1027, 1032, 1042–1043
- ADCON1 register, 1027, 1029, 1032–1034, 1042
- ADDCF, 1013
- ADDDCF, 1013, 1015
- Addendum IR robot project, 898–903
- Addendum to electronic thermometer project, 910–913
- Addition, 352, 353, 496, 1092, 1110
 - in C programming language, 1127, 1128
 - with carry, 285
 - in mid-range PIC microcontrollers, 316–318
 - with sixteen-bit values, 1202–1205
- ADDLW, 317, 986, 991
- Address(es), 376–381, 1213
- Address summary, 117
- Addressing modes, 13
- ADDWF, 316–318, 339, 981, 986, 992
- ADDWFC, 993
- ADIE, 1029
- ADRESH, 1029, 1043
- ADRESL, 1029, 1043
- Alarm, 1130
- Allocate memory, 1132, 1133
- Alpha characters, 1133
- Alphabetic character, 1144
- Alphanumeric characters, 1133
- Alternating current (ac), 1073
- ALU. *See* arithmetic/logic unit
- ALU STATUS register, 14
- Aluminum layer, 158, 159
- American Standard Character Interchange Interface. *See* ASCII
- ampasm (use MPASM assembler), 1099
- Amps, 1213
- Analog, 1213
- Analog clock project, 878–883
- Analog input, 653–657
- Analog I/O, 649–657
 - input, 653–657
 - interfacing with, 682–690
 - PWM I/O, 685–690
 - reading potentiometers, 683–685
 - voltage comparators, 650–653
- Analog I/O application code
 - resistance measurement without ADC, 798–804
 - resistor ladder output, 804–810
- Analog oscilloscopes, 1084, 1085
- Analog-to-digital converter (ADC), 649–650, 653–657, 714, 715, 1027, 1029, 1145, 1213
- AND, 1213
 - in assembly language, 389–390
 - bitwise, 496
 - in C programming language, 1127, 1128
 - logical, 497
 - in mid-range PIC microcontrollers, 324–325
 - in MS-extension BASIC, 1093
 - in PICBASIC, 1110, 1111
- AND (MS-extension BASIC), 1093
- AND (PICBASIC), 1111
- ANDLW, 325, 981, 986, 994
- ANDWF, 325, 981, 986, 994
- Animation, 144, 678–679
- API (application programming interface), 538
- Append string, 1132
- Apple II, 88, 89
- Application code, 7, 790–794
 - for analog I/O. *See* analog I/O application code
 - button press with debounce, 774–779
 - current calculation, 771–773
 - for I/O with interrupts (*see* I/O with interrupts application code)
 - jump, 755–770
 - random number generator, 790–794
 - for serial I/O. *See* serial I/O application code
 - simplest, 794–798
 - TMR0 delay, 779–781
 - TMR0 setup with prescaler, 786–790
 - watchdog timer, 781–786
- Application code version, 162
- Application design and hardware interfacing, 589–616
 - interfacing to external devices, 611–616
- power management, 603–608
- project management, 599–602
- requirements definition, 590–595
- reset, 608–610
- resource allocation, 595–597
- user interfacing, 597–599
- Application development,
 - block diagram of, 520–521
- Application development tools, 39–41
- Application map file, 522–523
- Application programming interface (API), 538
- Arccosine, 1134

- Architectures, processor. *See* processor architectures
- Arcsine, 1134
- Arctangent, 1133, 1134
- Argument, 1213
- Arithmetic instructions, 293, 316–324
- Arithmetic series, 393–398
- Arithmetic/logic unit (ALU), 244, 247, 248, 254–259, 1213
- Array addressing, 263–264
- Arrays, 1213
- in assembly language, 411–414
 - and data stack, 73, 74
 - declaring, 1103, 1124
 - dimensions of, 1124
 - erase array variable, 1093
 - optimizing, 440
 - return first index of array variable, 1093
 - return last index of array variable, 1093
 - starting array index, 1090
- ASC (MS-extension BASIC), 1095
- ASCII (American Standard Character Interchange Interface), 1067–1071, 1213
- character table, 1067
 - control characters, 1068–1069
 - convert character to, 1095
 - convert nybble to ASCII macro, 1151–1152
 - convert to character, 1095
 - convert to hex nybble macro, 1152
 - convert to upper case macro, 1157
 - IBM extended character set, 1069–1071
- ASCII control characters, 678
- ASCII data type, 1104
- ASCII string, 1092
- ASCHIZ, 1132, 1146, 1213
- ASM (PICBASIC), 1104
- ASM files, 108
- Assembler calculator, 494–499
- Assembler options, 132–133
- Assembler source files, 40, 1092
- Assemblers, 40, 68–70, 1214.
- See also* MPASM assembler
- Assembly language, 1214
- conditional assembly/compilation, 500–507
 - flashing LED application of, 152–154
 - insertion into PICBASIC applications, 1104, 1107
 - interrupt handlers in, 1107–1108
 - macros in, 489–492
- Assembly-language techniques, 373–444
- bit AND/OR, 389–390
 - constant multiplication/division, 392–400
 - delays, 400–405
 - event-driven programming, 426–429
 - guidelines for, 443–444
 - interrupts, 419–423
 - labels/addresses/flags, 376–381
 - optimizing PIC MCU applications, 438–443
 - patch space, 405–407
 - porting code between PIC MCU device architectures, 430–438
 - reentrant subroutines, 423
 - sample template, 374–376
 - simulating logic, 423–426
 - 16-bit operations, 390–392
 - sorting data, 414–419
 - state machine programming, 429–430
 - structures/pointers/arrays, 407–414
 - subroutines with parameter passing, 381–385
 - subtraction/comparing/negation, 385–389
- ASSIGNMENT (BASIC statement), 1090
- ASSIGNMENT (C), 1125
- ASSIGNMENT (PICBASIC), 1114
- Assignment indicator, 1092
- Assignment operators, 498
- Assignment statement, 71, 89–91
- Associative law (Boolean arithmetic), 1066
- Astronomical unit, 1061
- Asynchronous serial, 30, 1214
- Asynchronous serial communication, 644–649
- Asynchronous (NRZ) serial interfaces, 693–704
- RS-232, 695–702
 - RS-485/RS-422, 702–704
- Asynchronous serial I/O hardware with pin stimulus file, 832–841
- Asynchronous stimulus, 146, 147
- At character (@), 1104
- ATN (BASIC), 1091
- Atomic mass unit, 1062
- Attach technology, 37
- Audio notes, 1062
- Audio output, 690–692
- B**, 1013
- Background color, 1096
 - Backslash, 1130
 - Backspace (BS), 1068, 1130
 - Backup file (.bkx), 113
 - BADRAM (MPASM), 129
 - Ball grid array (BGA), 38, 1214
 - Bank addressing, 250–253
 - in mid-range PIC microcontrollers, 312–315
 - in PIC18 microcontrollers, 286–288
 - Bank select register (BSR), 359
 - BANKISEL (MPASM), 129
 - Banks, 247, 250
 - BANKSEL (MPASM), 129
 - .bas files, 108, 1099
 - Base, 1078
 - BASE (BASIC), 1090
 - Base 10 logarithm, 1133, 1135
 - BASIC (Beginner's All-purpose Symbolic Instruction Code) programming language, 88–90, 1089–1121
 - function statements, 1090–1098
 - Microsoft extensions to, 1093–1098
 - operators, 1092
 - order of operations, 1092
 - Basic SSP (BSSP), 633–635
 - BASIC Stamp. *See* Parallax BASIC Stamp language
 - BASIC87x, 90–95
 - arithmetic operators in, 92
 - characteristics of, 90–91
 - conditional operators in, 94
 - statements in, 93–94
 - user interface commands in, 95

- BasicRS.asm application,
 - 832–841
- Battery symbol, 1073, 1074
- Battlestar Gallactica*, 811
- Baud rate, 694, 1139
- Baudot asynchronous serial data,
 - 644
- Baudot teletype, 693, 694
- BC, 995, 1013
- Bc macro, 437
- BCA (ball grid array), 1214
- BCD (binary code decimal),
 - 1214
- BCF, 311, 312, 981, 986, 995,
 - 1149–1150
- BDC, 1013
- BEEP (MS-extension BASIC),
 - 1097
- Bell (BEL), 1068, 1130
- Best meets*, 617
- Beta, 1078
- BGA (ball grid array), 1214
- Bill of material (BOM), 601
- Binary code decimal (BCD),
 - 1214
- Binary data type, 1104
- Binary numbers, 1214
- Bipolar logic, 1214
- Bipolar NPN transistors, 1078
- Bipolar stepper motor control,
 - 730–733
- Bit, maximum array size for,
 - 1103
- BIT (PICBASIC), 1102
- Bit AND, 389–390, 496, 1127
- Bit banging, 574–576, 1214. *See also* recycled code
- Bit mask, 1214
- Bit negation, 1127
- Bit operations, 1110–1111
- Bit OR, 390, 496, 1127
- Bit ripple, 441
- Bit XOR, 496, 1127
- Bit-banging I/O, 19–21
- Bitwise operations (macro),
 - 1158–1159
- Bitwise value, 1110
- blackier than black*, 946
- Blank lines, 91, 128, 134
- Blanks, 1095
- Blinking light project, 883–891
- Block diagrams, 243
- Blocking, 540
- Bluetooth, 32
- BN, 995
- BNC, 995, 1013
- BNDc, 1013
- BNN, 995
- BNOV, 995
- BNZ, 995, 1013
- Boltzmann entropy constant,
 - 1061
- Bondout chip, 80, 81, 209
- Books, 966–970
 - about PIC microcontroller,
 - 966–967
 - for electronics/software development, 967–970
- Boolean arithmetic, 1065–1066
- Boost power supply, 1214
- Bootloaders, 165, 166, 527–536
 - mid-range, 530–535
 - need for, 527–528
 - PIC18, 535–536
 - requirements for, 528–530
 - sample code for, 532–535
- BootRAM, 535–536
- Borrow flag, 258
- Bounce, 665–668, 1214
- BOV, 995
- BRA, 367–369, 963, 996
- Braces ({}), 1109
- Branch, 14
 - in assembly language, 388
 - in PIC18 microcontrollers,
 - 367–369, 377–378, 437, 963
- BRANCH (PICBASIC), 1112
- BREAK (C), 1130
- Breakpoints, 144, 238, 239, 512,
 - 552
- Brown outs, 6
- Brown-out detect (BOD), 448,
 - 461, 1144
- Brown-out detect reset (BOR),
 - 269, 448, 609
- BSF, 312, 981, 996, 1149–1150
- BSR register, 284, 287, 358,
 - 359, 1030, 1049
- BTFSC, 312, 332–336,
 - 981–982, 986–987, 996, 1149
- BTFSS, 312, 332–336, 996,
 - 1149
- BTG, 996
- Bubble sort, 414, 419
- Buck power supply, 1214
- Build tools, application. *See* MPLAB IDE application
 - build tools
- Built-in debugger, 58–59
- Built-in EEPROM/Flash access
 - in, 618–624
- Built-in functions (PICBASIC),
 - 1112–1121
- Built-in oscillator, 6
- Bumps, 39
- Bus, 1214
- Bus protocol, 4, 5
- Bussed connections, 28, 29
- BUTTON (macro), 1199–1200
- BUTTON (PICBASIC), 1112
- Button debouncing (macro),
 - 1198–1200
- Button press with debounce,
 - 774–779
- BUTTON_PAUSE (PICBASIC),
 - 1105
- Byte, maximum array size for,
 - 1103
- Byte wide access method,
 - 1149
- BZ, 996, 1013
- C (insert comments into PICBASIC-produced assembler source file),
 - 1099
- C compiler for PIC18, 135
- .c files, 108
- C for PIC microcontroller, 92,
 - 96–101
- C programming language,
 - 1123–1148
 - backslash characters, 1130
 - common functions,
 - 1130–1133
 - compound assignment statement operators, 1128
 - directives, 1129
 - linked application in, 519–526
 - logical/arithmetic operators,
 - 1127
 - macros in, 499–500, 1130
 - Microchip C18 functions,
 - 1138–1148
 - order of operations, 1128
 - PICC functions, 1133–1138
 - reserved words, 1130
- Calculator, assembler, 443,
 - 494–499
- Calibration, 57–58
- Calibration value register, 1019,
 - 1022
- Calibration values, 477–478

- CALL, 301, 302, 382, 982, 987, 996
 - and interrupt handlers, 328
 - in low-end PIC microcontrollers, 353–355
 - in mid-range PIC microcontrollers, 328, 330–332
 - in PIC18 microcontrollers, 367
 - in PICBASIC, 1112
- Call instruction, 14
- CALLOC (C), 1133
- Calls, subroutine, 93, 268, 278, 432–433, 1090
- Capacitors, 1075–1077, 1214
 - charged, 683–684
 - decoupling, 446, 447, 603–604
 - derating, 447
 - external, 455–456
 - time required to charge, 799
 - types of, 1076
- Capture event, 1146
- Carriage return (CR), 1068, 1130
- Carry flag (C), 258–259, 318, 320, 321, 327, 386–387, 1150, 1205–1206
- CASE (C), 1130
- CASE (MS-extension BASIC), 1095
- CBLOCK, 123–125, 129, 380, 390, 408–409, 444, 1201
- CCP (compare, capture, and PWM), 1022, 1024
- CCP PWM, 721–724
- CCP1CON register, 628, 629, 1041
- CCP2CON register, 628, 629, 1040–1041
- CCPR1H, 1041
- CCPR1L, 1041
- CCPR2H, 1041
- CCPR2L, 1041
- CCPR2x registers, 1022
- CCPx register, 1022
- CCPxCON register, 1022, 1025
- CCPxIF flag, 1024
- Celestica, 597
- Central processing unit (CPU), 244–248, 1215
- Centronics, 749, 751
- Ceramic disk, 446, 447
- Ceramic encapsulants, 36
- Ceramic packaging, 36–37, 51
- Ceramic resonator, 1214
- Ceramic resonators, 456–458
- C4, 39
- CGRAM, 679
- Change execution, 93
- Change on interrupt, 476, 477
- CHAR * ASCTIME (PICC), 1134
- CHAR * BTOA (C18 C), 1139
- CHAR * CGETS (PICC), 1134
- CHAR * CTIME (PICC), 1134
- CHAR * GETS (PICC), 1135
- CHAR * ITOA (C18 C), 1145
- CHAR * LTOA (C18 C), 1145
- CHAR * STRCAT (PICC), 1136
- CHAR * STRCHR (PICC), 1136
- CHAR * STRCPY (PICC), 1136
- CHAR * STRICHR (PICC), 1136
- CHAR * STRIRCHR (PICC), 1137
- CHAR * STRISTR (PICC), 1137
- CHAR * STRNCAT (PICC), 1137
- CHAR * STRNCPY (PICC), 1137
- CHAR * STRPBK (PICC), 1137
- CHAR * STRRCHR (PICC), 1137
- CHAR * STRSTR (PICC), 1137
- CHAR * STRTOK (PICC), 1137
- CHAR BUSY (C18 C), 1139
- CHAR BUSYADC (C18 C), 1139
- CHAR DATARDY USART (C18 C), 1142
- CHAR GETCH (PICC), 1135
- CHAR GETCHE (PICC), 1135
- CHAR ISBOR (C18 C), 1144
- CHAR ISLVD (C18 C), 1144
- CHAR ISMCLR (C18 C), 1144
- CHAR ISPOR (C18 C), 1144
- CHAR ISWDTTO (C18 C), 1144
- CHAR ISWDTWU (C18 C), 1144
- CHAR ISWU (C18 C), 1144
- CHAR SWACKI2C (C18 C), 1147
- CHAR SWGETSI2C (C18 C), 1147
- CHAR SWNOTACKI2C (C18 C), 1147
- CHAR SWPUTSI2C (C18 C), 1147
- CHAR SWREADI2C (C18 C), 1147
- CHAR SWWRITEI2C (C18 C), 1147
- CHAR TOLOWER (C18 C), 1147
- CHAR TOLOWER (PICC), 1138
- CHAR TOUPPER (C18 C), 1147
- CHAR TOUPPER (PICC), 1138
- CHAR WRITESWSPI (C18 C), 1148
- Character, 678–679, 1215
- Character box, 678
- CHAR_PACING (PICBASIC), 1105
- Checksum, 157–158, 1135
- Chip package, 1215
- Chip-on-board (COB) packaging, 38–39, 52
- Chop, 1084
- CHR\$ (MS-extension BASIC), 1095
- Christmas decoration with blinking light and music project, 883–891
- Chromiance*, 946
- Circle, 1096
- CIRCLE (MS-extension BASIC), 1096
- CISC (complex instruction set computer), 1215
- Clamping diodes, 610, 862
- CLEAR, 256, 1112
- Clear command, 192–193
- Clear screen, 1095
- Clock(s), 6, 29, 30, 1215
 - analog clock project, 878–883
 - internal, 487
 - programmable, 211
 - RTOS application for digital, 553, 555–563
 - as stimulus, 147, 148
 - in synchronous serial interface, 704
 - system, 454–460
- Clock cycle, 12, 1215
- Clock speed, 137, 138, 1025
- CLOSE (MS-extension BASIC), 1098
- Close file, 1132
- Closed series, 397
- CLRC, 1014
- CLRDC, 1014
- CLRF, 982, 987, 997
- CLRW, 307, 982, 987
- CLRWD, 345, 982, 987, 997
- CLS (MS-extension BASIC), 1095
- CMCON register, 1026, 1031
- CMIE, 1027
- CMIF, 1027
- CMOS logic, 1215
- CMOS SRAM memory cell, 26
- CMOS technology, 3

- CODE (MPASM), 129
 - Code file (.cod or .cof), 118
 - Code protection, 158–159, 1100
 - CODEPAGE (MPLINK), 140
 - Collectors, 1078
 - Colon, 379
 - COLOR (MS-extension BASIC), 1096
 - Column, 1096
 - COMF, 323–324, 982, 987, 997
 - Comma, 1091
 - Command line options, 1099
 - Comments, 91, 93, 127–128, 295, 500, 1091, 1099, 1101, 1108
 - Communication,
 - microcontroller. *See* microcontroller communication
 - Commutative law (Boolean arithmetic), 1066
 - Comparators, 1215
 - and logic probes, 1082
 - voltage, 650–653
 - Compare/capture PWM (CCP) module, 628–633
 - Comparison operators, 498, 507, 514
 - Comparisons
 - in assembly language, 385–389, 497–499
 - with sixteen-bit variables, 1206–1207
 - Compilation, 500–507
 - Compilers, 40, 70–76, 488, 519, 1215
 - Complement, 323–324, 496
 - Complementary law (Boolean arithmetic), 1065
 - Complementary metal oxide silicon (CMOS), 1080
 - Complex instruction set computer (CISC), 7–8, 1215
 - Composite video, 945, 950
 - Compound assignment statement operators, 1128
 - CON (PICBASIC), 1104
 - Concatenate, 500, 1092, 1136, 1137, 1215
 - Conditional assembly
 - statements, 130, 131, 500–509
 - in compilation, 500–507
 - for debug, 507–509
 - indenting, 443
 - Conditional branch instructions, 369
 - Conditional instructions, 14
 - Conditional jumping, 293–294, 334, 760–766
 - Conditional macros, 513–518
 - Conditional operators, 94
 - Conditionally change execution, 93
 - CondJump application, 760–766
 - Conducting region, 1079
 - CONFIG (assembly language), 444
 - CONFIG (MPASM), 129
 - Configuration fuses/fuse
 - registers, 112, 158, 159, 165, 205, 451–469
 - external memory, 463–466
 - labels/values, 453, 467
 - memory protection, 463
 - in MPASM, 125–127
 - operations controlled by, 56
 - PIC MCU, 466–469
 - of PIC microcontroller, 55–57
 - in PIC18 devices, 469
 - power-up timer, 461–462
 - programming, 170
 - registers, PIC MCU configuration, 466–469
 - reset, 460, 461
 - system clock/oscillators, 454–460
 - watchdog timer, 462–463
- Connectors (J), 1074
- CONST (MS-extension BASIC), 1093
- CONSTANT (MPASM), 129
- Constants, 1215
 - in assembly language, 124–125, 390–391
 - declaring, 129, 1104
 - defining, 1093
 - multiplication/division of, 392–400
 - AND operation with sixteen-bit, 1205
 - physical, 1061–1062
- Context information, 16, 17
- Context registers
 - and interrupt handlers, 342, 343
 - in PIC18 RTOS design, 542–543
- Context saving
 - interrupt-handler code for, 419–421
 - timer/interrupt handler with, 810–815
- Contiguous, 1215
- Contingency, 601
- CONTINUE (C), 1130
- Continue statement, 1126
- Contrast voltage, LCD, 679
- Control characters, ASCII, 1068–1069, 1144
- Conversion program, 40
- Convert ASCII to hex nybble (macro), 1152
- Convert ASCII to upper case (macro), 1157
- Convert nybble to ASCII (macro), 1151–1152
- Copy bits from register to register (macro), 1151
- Corrective actions, proposing, 571–572
- cos (BASIC), 1091
- COS (PICBASIC), 1111
- Cosine, 1091, 1111, 1133, 1134
- Costing, project, 600
- Costs, 54
- Coulombs, 1075
- COUNT (PICBASIC), 1112
- Count number of 1s in byte (macro), 1157
- CPFSEQ, 997
- CPFSGT, 998
- CPFSLT, 998
- CPU. *See* central processing unit
- Critical path, 601
- Cross-assembler, 1215
- Crowbarring, 448
- Crystal, 1215
- Crystal resonators, 456–458
- Crystals, external, 456–458
- CSRLIN (MS-extension BASIC), 1095
- Ctrl codes, 40
- Current, 1215
 - amount of required, 3
 - calculating requirements for, 771–773
 - in game components, 607
 - sink vs. source, 18
- Current address, 496, 497

- Cursor, 677
 - enable, 674
 - move, 674
 - move to CGRAM/display, 675
 - return current column of, 1096
 - return current line of, 1095
 - return current row of, 1096
 - set move direction for, 673
- Cyclical redundancy check (CRC) code, 901–903
- Cywin, 740–741
- Cylon.asm application, 810–815

- DA (MPASM), 130
- DAC (digital-to-analog converter), 1216
- Dallas semiconductor, 924, 954
- Data, declaring, 134
- DATA (BASIC), 1090
- DATA (BASIC87x), 93
- DATA (MPASM), 130
- DATA (MS-extension BASIC), 1094
- DATA (PICBASIC), 1112
- Data areas, 13
- Data block header, 1090
- Data communications equipment (DCE), 695, 697, 699, 1216
- Data define, 93
- Data EEPROM size, 50
- Data movement, 260–264
 - direct addressing, 260–261
 - immediate data values, 261–262
 - indexed addressing, 262–264
 - in PIC microcontrollers, 260–264
- Data movement instructions, 12–13, 293
 - in low-end PIC MCU instruction set, 349–352
 - in mid-range PIC MCU instruction set, 304–316
 - for PIC MCU, 298–300
 - in PIC18 MCU instruction set, 357–364
- Data packets, 892
- Data processing instructions, 13–14, 293
 - in low-end PIC MCU instruction set, 352–353
 - in mid-range PIC MCU instruction set, 316–328
- Data processing instructions (*Cont.*):
 - for PIC MCU, 300–301
 - in PIC18 MCU instruction set, 364–366, 436–437
- Data rate, 694
- Data set ready (DSR), 697–699
- Data stack, 72–76, 87–88, 96–99
- Data terminal equipment (DTE), 695–697, 1216
- Data transmitter ready (DTR), 697–699
- Data type bytes, 157
- Data types
 - BASIC, 1089, 1090
 - and compilers, 84
 - PICBASIC, 1104
- Data values, 130
- Datasheets, referencing, 45–48
- DATE\$ (MS-extension BASIC), 1097
- daw, 998
- DB (MPASM), 130
- dc electronics formulas, 1063–1064
- dc motors, 711–724
 - CCP PWM for motor speed control, 721–724
 - disassembled, 713
 - electromagnetic charging fields in, 712, 713
 - H-bridge motor control, 715–717
 - interior parts of, 712
 - low-value resistor in, 714–715
 - NPN transistor in, 713, 714
 - software PWM for motor speed control, 717–721
 - types of, 712
- DCD (PICBASIC), 1110
- DCE (data communications equipment), 1216
- DCFSNZ, 999
- DE (MPASM), 130
- De Morgan's theorem (Boolean arithmetic), 1066
- debounce (application), 774–779
- debounce (macro), 1198–1199
- Debounce/debouncing, 665, 1216
 - button debouncing macro, 1198–1200
 - button press with, 774–779
 - of inputs with interrupts, 826–832
- Debug, enable/disable, 1113
- DEBUG (PICBASIC), 1112
- Debug file, 40
- Debug monitor program, 1115
- DEBUG_BAUD (PICBASIC), 1105
- DEBUG_BIT (PICBASIC), 1105
- Debugger(s), 40, 1216
 - emulators vs., 207
 - hardware, 80–81
 - MPLAB ICD 2, 214–219
- Debugging, 565–587
 - corrective action proposals, 571–572
 - defines/conditional assembly for, 507–509
 - exercise for, 577–587
 - expected state documentation, 566–567
 - hypothesis and testing, 569–570
 - of macros, 509–513
 - options for, 49
 - problem characterization, 567–569
 - simulation of application, 573–576
 - solution release, 576–577
 - test fixes, 572–576
- DEBUGIN (PICBASIC), 1112
- DEBUGIN_BIT (PICBASIC), 1105
- DEBUGIN_MODE (PICBASIC), 1105
- DEBUGIN_REG (PICBASIC), 1105
- DEBUG_MODE (PICBASIC), 1105
- DEBUG_PACING (PICBASIC), 1105
- DEBUG_REG (PICBASIC), 1105
- DECF, 322–324, 982, 987, 998–999
- DECFSZ, 337–338, 983, 987, 999
- Decimal data type, 1104
- Decimal numbers, 1216
- Declare, 93
- DECLARE (MS-extension BASIC), 1093
- Declare subroutine or function, 1093
- Decoupling, 446–451
- Decoupling capacitors, 6, 446, 447, 603–604, 1216
- Decrement, 322–324, 337–338, 1127, 1151, 1202

- DEF (MS-extension BASIC), 1094
- DEFAULT (C), 1130
- Default configuration fuse
 - register settings, 1100
- Default numbering system, 122, 127
- DEFDBL (MS-extension BASIC), 1093
- Define, 492–494, 507–509
 - of internal data, 93
 - of subroutine, 1094
 - of variable, 1093
- DEFINE (assembly language), 375–376, 380–381, 492–494, 507–509
- DEFINE (C), 499–500, 1129
- DEFINE (MPASM), 130
- DEFINE (PICBASIC), 1101
- DEFINT (MS-extension BASIC), 1093
- DEFLNG (MS-extension BASIC), 1093
- DEFSNG (MS-extension BASIC), 1093
- DEFSTR (MS-extension BASIC), 1093
- Delay macro(s)
 - simple, 1160–1163
 - sixteen-bit counter with
 - constant loop delay, 1154–1155
 - sixteen-bit pulse measurement with five-cycle delay, 1155
- Delays
 - in assembly language, 400–405
 - TMR0, 480–482
- Delete label, 134
- Demultiplexor, 1216
- Depletion region, 1080
- Derating, 447, 603, 604
- Desktop version, MPLAB IDE, 118–120
- Destination (PIC MCU instruction), 329
- Detect change in register (macro), 1155
- Device driver, 538
- Device I/O, 6
- Device operating parameters, 376
- Device packaging, 35–39
- Device pinouts, 1030–1060
 - low-end, 1031
 - mid-range, 1032–1034, 1053–1058
 - PIC18, 1034, 1058–1060
- Dielectric, 1075–1077
- Different logical values, 1111
- Differential-pair serial communications, 701–703
- DIG (PICBASIC), 1110
- Digit carry (DC) flag, 259, 318, 320, 386–387
- Digit number, 1110
- Digital, 1216
- Digital clock and thermometer
 - RTOS application, 553, 555–563
- Digital I/O, 486
- Digital multimeters (DMMs), 803, 810, 1081
- Digital oscilloscopes, 1085
- Digital signal processors (DSPs), 2, 650, 953
- Digital storage oscilloscopes (DSOs), 1085
- Digital-to-analog converter (DAC), 1216
- Digitizers, 1085
- DIM (BASIC), 1089, 1090
- DIM (BASIC87x), 93
- Diodes, 1076, 1077
- Direct addressing, 246, 260–261, 298, 299
- Direct bank addressing, 252
- Direct current (dc), 1073
- Directives
 - in C programming language, 1129
 - in MPASM assembler, 121, 128–135
- Directory path, 1099
- DISABLE (PICBASIC), 1113
- DISABLE INTERRUPT (PICBASIC), 1113
- Disable interrupts, 1113
- Disassembler, 239–240
- Disc drives, 2
- Display
 - enable, 674
 - move, 676
 - read/write ASCII to, 675
 - shift, 674
- Distributive law (Boolean arithmetic), 1066
- Divide assignment variable, 1128
- Divide by three (macro), 1153–1154
- Divide macro, 399–400
- Division, 496, 1092, 1110, 1134, 1135
 - in assembly language, 397–400
 - in C programming language, 1127
 - in mid-range PIC microcontrollers, 327
 - with sixteen-bit numbers, 1210–1212
- DIV_T DIV (PICC), 1134
- DLAYMACRO, 1160–1163
- DO... LOOP WHILE (MS-extension BASIC), 1094
- DO... WHILE (C), 1126
- DO (C), 1130
- DO macro, 514
- Do not assemble compiled code, 1099
- DO WHILE... LOOP (MS-extension BASIC), 1094
- Documentation, 595
- DOUBLE ACOS (PICC), 1134
- DOUBLE ASIN (PICC), 1134
- DOUBLE ATAN (PICC), 1134
- DOUBLE ATOF (C18 C), 1139
- DOUBLE ATOF (PICC), 1134
- DOUBLE CEIL (PICC), 1134
- DOUBLE COS (PICC), 1134
- DOUBLE COSH (PICC), 1134
- DOUBLE EVAL_POLY (PICC), 1134
- DOUBLE EXP (PICC), 1134
- DOUBLE FABS (PICC), 1134
- DOUBLE FLOOR (PICC), 1135
- DOUBLE FREXP (PICC), 1135
- DOUBLE LDEXP (PICC), 1135
- DOUBLE LOG (PICC), 1135
- DOUBLE LOG10 (PICC), 1135
- DOUBLE MODF (PICC), 1135
- Double negation law (Boolean arithmetic), 1065
- DOUBLE POW (PICC), 1135
- Double precision floating-point numbers, 1090, 1093
- Double quotes ("), 1104, 1130
- DOUBLE SIN (PICC), 1136
- DOUBLE SINH (PICC), 1136

- DOUBLE SQRT (PICC), 1136
- DOUBLE TAN (PICC), 1137
- DOUBLE TANH (PICC), 1137
- Double word, 1124
- DRAM (dynamic RAM), 1223
- Draw, 1096–1097
- DRAW (MS-extension BASIC), 1096–1097
- Driver, 1216
- DS275, 954, 955
- DS1820 temperature sensor, 924, 926
- D-shell connectors, 1216
- DSREAD subroutine, 561–562
- DSRESET subroutine, 560–561
- DSEND subroutine, 561
- DT (MPASM), 130
- DT Compress application, 232–234
- DTE (data terminal equipment), 1216
- DTMFOUT (PICBASIC), 1113
- Dual in-line package (DIP), 51, 1034, 1058
- Dump operation, 193
- Duty cycle, 685, 1216
- DW (MPASM), 130
- Dynamic RAM (DRAM), 1223
- ECCP, 1147
- ECCP PWM, 722
- ECIO mode, 459
- Edge-triggered, 1216
- Editors, 39–40, 65–68, 1216
- EEADR register, 1021
- EECON1 register, 618–621, 1021, 1023
- EECON2 pseudoregister, 618, 619, 621, 1021
- EEDATA register, 1021
- EEIE interrupt request bit, 1021
- EEPROMReadMacro, 235
- EEPROM (electrically erasable programmable memory), 25, 1216
 - access data in, 93
 - access to built-in, 618–624
 - code protection in, 159
 - early use of, 155
 - generate acknowledge for I2C memory devices, 1143
 - read byte from I2C memory devices, 1143
 - test hidden value and checksum in, 1135
 - write byte to I2C memory devices, 1143
 - write string to I2C memory devices, 1143
- EEPROM (PICBASIC), 1113
- 8 × 8 multiplier, 285–286
- 8-N-1, 695
- 18-pin PIC microcontrollers, 1055
- 8250 UART, 743
- El Cheapo programmer, 196–203
 - bill of materials for, 199
 - bottom layer design for, 201
 - circuitry of, 198
 - photograph of, 197
 - process of building, 202–203
 - PTH PIC MCU installation options for, 202
 - silkscreen overlay for, 200
 - trouble shooting for, 203
- Electric motors, 712
- Electrical engineering formulas, 1063–1065
 - AC electronics, 1064–1065
 - DC electronics, 1063–1064
- Electrically erasable programmable memory. *See* EEPROM
- Electrolytic capacitors, 446
- Electromagnetic charging fields, 712, 713
- Electromagnetic waves
 - frequency formula, 1065
- Electron rest mass, 1061
- Electronic components and symbols, basic, 1073–1080
- Electronic reference information, 1073–1087
 - basic components/symbols, 1073–1080
 - test equipment, 1080–1087
- Electronic thermometer
 - project(s)
 - addendum to, 910–913
 - with seven segment LED displays, 903–910
- Electronics Now*, 891
- Electronvolt value, 1061
- Electrostatic discharges (ESDs), 462
- ELEF (C), 1129
- else, 1129
- else (C), 1129, 1130
- else (MPASM), 130, 501, 514
- ELSEIF (MS-extension BASIC), 1094
- Embedded microcontrollers
 - applications for, 5–7
 - basic features of, 3
 - pins available in, 28
 - primary role of, 1
 - for simple applications, 2
- Emitters, 1078
- EmuCall macro, 236–237
- Emu-II emulator, 208, 219–241
 - and application debugging rules, 228
 - bill of material for, 223
 - built-in interface functions for, 227
 - command set for, 225
 - execution options for, 220
 - registers supported by, 226–227
 - reserved address instructions for, 229
 - schematic of, 222
 - unassembled display, 239
- Emulator(s), 40, 79–80, 207–214, 1216
 - debuggers vs., 207
 - Emu-II, 219–241
 - full/partial, 241
 - MPLAB ICE-2000, 210–213
 - MPLAB REAL ICE, 213–214
 - types of, 207–210
- EmuReturn macro, 237
- ENABLE (PICBASIC), 1113
- ENABLE DEBUG (PICBASIC), 1113
- ENABLE INTERRUPT (PICBASIC), 1113
- Enable register, 249
- Encapsulant, 2
- Encapsulation, 35
- Encyclopedia of Electronic Circuits* (Rudolf Van Graf), 590
- END (BASIC), 1091
- END (BASIC87x), 93
- END (MPASM), 130, 514
- END (PICBASIC), 1113
- END DEF (MS-extension BASIC), 1094
- End function, 1094
- END FUNCTION (MS-extension BASIC), 1094

- END SELECT (MS-extension BASIC), 1095
- END SUB (MS-extension BASIC), 1094
- End subroutine, 1094
- ENDASM (PICBASIC), 1104, 1107
- ENDC (MPASM), 129, 130
- ENDIF (C), 1129
- ENDIF (MPASM), 131, 501, 502
- ENDIF (MS-extension BASIC), 1094
- ENDIF (PICBASIC), 1114
- ENDM (MPASM), 131
- ENDW (MPASM), 131, 501
- Enhanced input capture, 1142
- Enhanced PWM channel, 1142
- Environmental qualification, 594–595
- Environmental stress screening (ESS), 597
- EOF (MS-extension BASIC), 1098
- EPROM (erasable programmable read only memory), 23–25, 36, 158, 159, 1217
- EQU (MPASM), 129
- Equal, 1111, 1127
- Equivalence, 1093
- Equivalent series resistance (ESR), 446
- EQV (MS-extension BASIC), 1093
- Erasable programmable read only memory. *See* EPROM
- Erase, 1129
- ERASE (MS-extension BASIC), 1093
- Erase array variable, 1093
- Erase operation, 173, 176–177
- ERL (MS-extension BASIC), 1094
- ERR (MS-extension BASIC), 1094
- Error(s)
 - assembler, 70, 120
 - return line of, 1094
- ERROR (C), 1129
- ERROR (MPASM), 131
- ERROR (MS-extension BASIC), 1094
- Error characterization, 567
- Error file (.err), 117
- Error handler, 1094
- Error number, 1094
- ERRORLEVEL (MPASM), 131
- Ethernet, 32
- European Union, 595
- Event sensors, 50
- Event-driven operating system, 537
- Event-driven programming, 426–429, 1217
- Exchange variables, 1120
- Exclusive or, 326, 1111, 1227
 - in C programming language, 1128
 - in MS-extension BASIC, 1093
 - in PICBASIC, 1110, 1111
- Execute
 - after persistent variable is altered, 1135
 - based on value, 1095
 - system command, 1132
- Execution change instructions, 14–15, 293–294
 - in low-end PIC MCU instruction set, 353–356
 - in mid-range PIC MCU instruction set, 328–344
 - for PIC MCU, 301–303
 - in PIC18 MCU instruction set, 366–371
- Execution control state machines, 766–770
- Execution status flags, 256, 259
- EXIT (MS-extension BASIC), 1094
- EXTM (MPASM), 131
- EXP (BASIC), 1091
- EXPAND (MPASM), 131
- Expected state, documentation of, 566–567
- Exponent, 1133, 1134
- Exponentiation, 1091, 1092
- Expressions, 1125
- EXTERN (MPASM), 131
- External capacitors, 455–456
- External crystals, 456–458
- External devices, interfacing to, 611–616
 - combining input and output, 613–615
 - memory-mapped I/O, 611
 - parallel-bus devices, 612–613
 - simulated open-collector/open-drain I/O, 615–616
- External interfaces, 2
- External memory, 4, 463–466, 1217
- External oscillators, 456–458
- External reset circuitry, 269
- External resistors, 455–456
- External resonators, 456–458
- External ROM, 22
- F flag, 271–273
- FAEs (field application engineers), 966
- Failure analysis, 565
- Families, microcontroller, 59–62
- Fan governor project, 953–963
- Farads (F), 1075
- Fast stack, 288–289, 543
- Feature-to-part-number table, 977–979
- Feedback, 598
- Fehrenbach, Bob, 1155
- Field application engineers (FAEs), 966
- FIFO (first in, first out), 1217
- File(s)
 - change name of, 1097
 - close, 1098, 1132
 - delete, 1097
 - end of, 1098
 - input from, 1132
 - list, 1098
 - move file pointer, 1098
 - open, 1098, 1131
 - output string to, 1132
 - read formatted data from, 1098
 - write to, 1098
- File name, 1097
- File register size, 50
- File registers, 13, 245, 248–254, 373
 - bank addressing, 250–253
 - optimizing, 439–440
 - zero registers, 253–254
- File Transfer Protocol (FTP), 1217
- File types, 107–118
- FILES (MS-extension BASIC), 1098
- FILL (MPASM), 131
- Fine pitch parts, 38
- First in, first out (FIFO), 1217
- First-pass assembly, 68, 69
- 5-pin ICSP cable adapter, 206
- Five-cycle delay (macro), 1155

- FIX (MS-extension BASIC), 1093
- Flag bits, 257, 335
- Flags, 376–381
- Flash, 465, 1217
- Flash program memory, 25, 36
 - built-in access to, 618–624
 - code protection in, 159
 - early use of, 155
 - patch space in, 406
 - programming of, 171–173
 - and simplified code, 487
- Flip-flop, 26, 1217
- FLOAT ATAN2 (C), 1133
- FLOAT COS (C), 1133
- FLOAT EXP (C), 1133
- FLOAT FABS (C), 1133
- FLOAT FRAND (C), 1133
- FLOAT LOG (C), 1133
- FLOAT LOG10 (C), 1133
- FLOAT POW (C), 1133
- FLOAT SIN (C), 1133
- FLOAT SQRT (C), 1133
- Floating, 616, 1217
- Floating gate, 23–24
- Floating the pin, 568
- Floating-point number, 1090, 1093, 1217
 - add integer to exponent of, 1135
 - convert into fraction and power of 2, 1135
 - convert string to, 1134, 1139
 - define variable as, 1093
- FOR... TO... NEXT (BASIC), 1091
- FOR (BASIC87x), 93
- FOR (C), 1130
- FOR (PICBASIC), 1113
- For statement, 1126
- Force error, 1129
- Form feed (FF), 1068, 1130
- Format, 1096
- Form_Load subroutine, 747
- Formulas
 - for electrical engineering, 1063–1065
 - mathematical, 1065–1066
- 40-pin PIC microcontrollers, 1056
- 40-pin PIC18 PTH microcontrollers, 1059
- 44-pin PLCC SMT package PIC microcontrollers, 1056, 1059
- 44-pin QFP SMT package
 - mid-range PIC microcontrollers, 1057, 1060
- Fosc, 1217
- Fractions, 397–400, 1135
- Frame, of video, 946, 950
- Framing error, 647
- FREE (C), 1133
- Free memory, 1133
- Free software tools, 740–742
- FREQOUT (PICBASIC), 1113
- Frequency(-ies), 690, 1217
 - of audio notes, 1063
 - formula for, 1065
 - output, 1113
 - for touch tone telephones, 1062
- FSR register(s), 249, 250, 252, 253, 350–352, 358–360, 1017, 1030
 - in C for PIC microcontrollers, 99, 100
 - in low-end devices, 434, 435
 - minimum values for, 276
 - in PIC18 devices, 285, 287
 - as temporary storage, 440
- FSR0H, 1050
- FSR0L, 1050
- FSR1H, 1049
- FSR1L, 1049
- FSR2H, 1049
- FSR2L, 1049
- FTP (File Transfer Protocol), 1217
- Full emulators, 207, 241
- Full-stepping, 730
- FUNCTION (MS-extension BASIC), 1094
- Functions, 1217
 - in BASIC, 1090–1098
 - common functions in C programming language, 1130–1133
 - current-requirements calculation/check, 771–773
 - debounce, 774–779
 - define, 1093, 1094
 - format of, 1125
 - in Microchip C18 programming language, 1138–1148
 - in PICC programming language, 1130–1133
 - prototype for, 1125
- Fuse register settings, 1100
- Fuseable link, 23
- Fuses, 1075, 1077
- Fuzzy logic, 1217
- Fuzzy logic fan governor project, 953–963
- Gandt chart, 600–601
- Gas constant, 1061
- Gates, 1078, 1079
- General Instruments, 60, 159
- Generate listing, symbol table, map files, 1099
- Generate listing file, 1099
- Generate parity for byte (macro), 1158
- GET (MS-extension BASIC), 1098
- Get byte, 94
- Get data, 1090
- Get formatted input, 1095
- GETC12C (C18 C), 1146
- GETCMWIRE (C18 C), 1146
- GETCSPI (C18 C), 1146
- GETCUSART (C18 C), 1146
- GETC#USART (C18 C), 1146
- GLINTD bit, 280
- GLOBAL (MPASM), 132
- Global variables, 132, 382, 523
 - and high level languages, 86–88
 - in linking, 523
- Glob-top packaging, 38
- GOSUB (BASIC), 1090
- GOSUB (BASIC87x), 93
- GOSUB (PICBASIC), 1113
- GOTO, 983, 988, 1000
- GOTO (BASIC), 1090
- GOTO (BASIC87x), 93
- GOTO (C), 1126, 1130
- GOTO (PIC MCU instruction), 301, 302
 - and interrupt handlers, 343
 - in low-end PIC microcontrollers, 353, 354
 - in mid-range PIC microcontrollers, 328–330, 334–336
 - in PIC18 microcontrollers, 366–371
- GOTO (PICBASIC), 1107, 1113
- Gotos, 14
- Gradicules, 1084
- Graphic character, 1144

- Graphical user interface (GUI), 1217
- Graphics figure, draw, 1096–1097
- Graphics window, 1096
- Greater than, 1092, 1111, 1127
- Greater than or equal to, 1092, 1111, 1127
- Ground (GND), 697, 1073, 1217
- Guardbanding, 593
- GUI (graphical user interface), 1217
- Gull-wing leaded devices, 37
- H/-? (display help screen), 1099
- Half-stepping, 730
- Hall-effect sensor project, 854–859
- Hall-effect switch, 858
- Handbook of Microcontrollers* (Predko), 881, 949
- HANDLE_FOPEN(C), 1131
- Handshaking, 697, 698, 700–701
- Hardware, 617–659
 - analog I/O. *See* Analog I/O
 - CCP module, 628–633
 - and compilers, 84
 - debugging problems with, 109
 - ICSP, 659
 - internal, 2–5
 - mid-range built-in EEPROM/Flash access, 618–624
 - of PIC microcontrollers, 248–254
 - PSP, 657–659
 - serial I/O. *See* Serial I/O
 - TMR1, 624–626
 - TMR2, 626–628
- Hardware debuggers, 80–81
- Hardware interrupts, 15–17
- Hardware operating parameters, 375
- Hardware registers, 112, 245, 249, 250, 287, 439–440
- Hardwired processors, 11–12
- Harmony remotes, 449, 603
- Harvard architecture, 8, 9, 11, 245, 1217
- H-bridge motor control, 715–717
- Help screen, display, 1099
- Henry (unit), 1076
- Hertz (Hz), 1217
- Hex, 1217
- Hex data type, 1104
- Hex file format (.hex), 40, 107, 117, 118, 156–158, 1092
- Hex numbers, 1218
- Hexadecimal character, 1130, 1138, 1144
- Hidden value, 1135
- HIGH (high 8 bits), 390–392, 496
- HIGH (MPASM), 124
- HIGH (PICBASIC), 1113
- High level language compilers, 8, 17, 71
- High level languages, 40, 83–103, 1218
 - BASIC and PICBASIC, 88–90
 - BASIC87x, 90–95
 - C for PIC microcontroller, 92, 96–101
 - global/local variables, 86–88
- High logic level, 1113
- Higher parameter, 1110
- Highest bit, 1111
- High-voltage (HV) devices, 446–451
- Hilgreave, 533, 739, 745
- Hitachi 44780, 672–674, 676, 677
- HI-TECH Software, 136
- Hobbyist programmers, 181–182
- Horizontal synch, 1218
- Horizontal tab (HT), 1068, 1130
- HSER_BAUD (PICBASIC), 1105
- HSER_EVEN (PICBASIC), 1105
- HSERIN (PICBASIC), 1113
- HSER_ODD (PICBASIC), 1105
- HSEROUT (PICBASIC), 1113
- HSER_RCSTA (PICBASIC), 1105
- HSER_SPBRG (PICBASIC), 1105
- HSER_TXSTA (PICBASIC), 1105
- HSPLL mode, 459–460
- HT-Soft, 1133
- Human-readable format, 118, 739, 745
- Hybrid networks, 34, 35
- Hyperbolic cosine, 1134
- Hyperbolic sine, 1136
- Hyperbolic tangent, 1137
- HyperTerminal, 191, 192, 533, 739, 745–746
- Hypothesis, debugging, 569–570
- Hysteresis, 1218
- Hz (hertz), 1218
- I2C (inter-intercomputer communication), 1025, 1219
 - generate not acknowledge condition, 1145
 - interfacing with, 707–710 and MSSP, 638–643
 - read byte from, 1146
 - write string to, 1146
- I2C bus, 1144, 1146–1148
- I2C interface, 1194–1198
- I2C memory devices, 1143
- I2C slave clock, 1142
- I2C_HOLD (PICBASIC), 1105
- I2C_INTERNAL (PICBASIC), 1105
- I2CREAD (PICBASIC), 1114
- I2C_SCLOUT (PICBASIC), 1106
- I2CSetup (macro), 1194–1198
- I2C_SLOW (PICBASIC), 1106
- I2CWRITE (PICBASIC), 1114
- IBM, 39
- IBM extended ASCII characters, 11–12, 1069–1071
- ICD (in-circuit debugger), 217–218, 1218. *See also* MPLAB ICD
 - 2 debugger
- ICD 2 debugger, 214–219
- ICE 2000 emulator, 210–213
- ICSP (in-circuit serial programming), 58–59, 659, 1218
 - hardware design for, 218, 219
 - ICD vs., 217–218
 - programmer interface for, 166–177
- ICSP cable adapter, 206
- ID location (IDLOCS), 162, 170
- IDATA (MPASM), 132
- Idempotent law (Boolean arithmetic), 1066
- Identify functions (Boolean arithmetic), 1065
- Identity law (Boolean arithmetic), 1065
- IDLOCS (MPASM), 132
- IF... THEN... ELSE (MS-extension BASIC), 1094
- IF... THEN... ELSE (PICBASIC), 1114
- IF... THEN (BASIC), 1090
- IF... THEN (BASIC87x), 93
- IF... THEN (PICBASIC), 1114
- IF (C), 1129, 1130

- IF (MPASM), 132, 501, 503–506
- IF (PICBASIC), 1109
- If condition definitions, 387
- If statement, 72, 1125
- IFC macro, 514
- IFDEF (C), 1129
- IFDEF (MPASM), 132, 501, 502, 508
- If-less-than compare/swap (macro), 1156–1157
- IFNDEF (C), 1129
- IFNDEF (MPASM), 132, 501, 502
- IFV macro, 514
- Immediate data, 246, 261–262, 298
- IMP (MS-extension BASIC), 1093
- Implication, 1093
- .inc files, 109–112
- INCF, 322, 324, 983, 988, 1000–1001
- INCFSZ, 337, 338, 983, 988, 1001
- In-circuit debugger. *See* ICD
- In-circuit serial programming. *See* ICSP
- INCLUDE (C), 1129
- INCLUDE (MPASM), 132
- INCLUDE (PICBASIC), 1101, 1114
- Include files, 112, 375
- Include files (.inc), 122
- Inclusive ORing, 325
- Increment, 322, 323, 337–338
- Incrementing
 - of sixteen-bit numbers, 1202
 - of w register, 1151
- Index register, 1218
- Indexed addressing, 246, 262–264, 299–300
 - in low-end PIC microcontrollers, 351–352
 - in mid-range PIC microcontrollers, 315–316
 - in PIC18 microcontrollers, 362–364
- INDF register, 250, 252, 315
- INDF registers, 362, 363
- INDF0 register, 1050
- INDF1 register, 1050
- INDF2 register, 1049
- INDIF register, 249
- Indirect addressing, 253
- Indirection (pointer to variable), 1127
- Inductors, 1076, 1218
- Infrared (IR), 603, 1218
- Infsnz, 1001
- Initializing variables, 130, 443
- INKEY\$ (MS-extension BASIC), 1095
- Input
 - analog, 653–657
 - combining output and, 613–615
 - configure/enable capture, 1145
 - get formatted, 1095, 1131
 - get from console, 1090
 - get one character, 1131
 - get specified number of characters, 1095
 - line into buffer as ASCII string, 1135
 - in low-end devices, 434
 - one character from console and echo on console, 1135
 - one character from console without echo, 1135
 - one line at time, 1134
 - return number of pending characters from keyboard buffer, 1095
 - string from file, 1132. *See also* I/O
- INPUT (BASIC), 1090
- INPUT (BASIC87x), 94
- INPUT (MS-extension BASIC), 1098
- INPUT\$ (MS-extension BASIC), 1095
- INPUT (PICBASIC), 1114
- Input capture, 1142
- Input mode, 251, 473
- Input pin, 1114
- Input pin interrupt with timer pin input, 815–819
- InsTemplate.asm, 294
- INSTR (MS-extension BASIC), 1095
- Instruction, 1218
- Instruction bytes, 157
- Instruction cycle, 12, 246, 400–405, 1142, 1143, 1218
- Instruction cycle period, 400
- Instruction decode and control circuitry, 247
- Instruction page, 133
- Instruction register, 247
- Instruction set(s)
 - legend, 979, 980
 - low-end, 980–985
 - Microchip special instruction mnemonics, 980, 1013–1016
 - mid-range, 980, 986–990
 - for PIC MCU. *See* PIC MCU instruction set
 - PIC18, 980, 991–1012
 - table of, 977, 979–1016
- Instruction symbol definitions, 980
- Instructions
 - data-movement, 12–13
 - data-processing, 13–14
 - execution-change, 14–15
 - PIC microcontroller, 121
 - processor-control, 15
 - types of, 12
- In-system programming, 1218
- INT (BASIC), 1091
- INT ABS (PICC), 1134
- INT ATOF (PICC), 1134
- INT ATOI (C18 C), 1139
- INT FCLOSE (C), 1132
- INT FGSETS (C), 1132
- INT FPRINTF (C), 1132
- INT FPUTS (C), 1132
- INT FSCANF (C), 1132
- INT GETC (C), 1132
- INT GETCHAR (C), 1131
- INT ISALNUM (C), 1133
- INT ISALPHA (C), 1133
- INT ISDIGIT (C), 1133
- INT ISLOWER (C), 1133
- INT ISSPACE (C), 1133
- INT ISUPPER (C), 1133
- INT MEMCMP (PICC), 1135
- INT PERSIST_CHECK (PICC), 1135
- INT PRINTF (C), 1131
- INT PUTC (C), 1132
- INT PUTCHAR (C), 1131
- INT PUTCHAR (PICC), 1136
- INT PUTS (PICC), 1136
- INT RAND (PICC), 1136
- INT SCANF (C), 1131
- INT SCANF (PICC), 1136
- INT SETJMP (PICC), 1136
- INT STRCHR (C), 1132

- INT STRCMP (C), 1132
- INT STRCMP (PICC), 1136
- INT STRICMP (PICC), 1136
- INT STRNCMP (C), 1132
- INT STRNCMP (PICC), 1137
- INT STRNICMP (PICC), 1137
- INT STRRCHR (C), 1132
- INT TOLOWER (C), 1133
- INT TOUPPER (C), 1133
- INTCON register, 249, 250,
252, 483–485, 1021, 1022,
1051
- INTCON2 register, 1050–1051
- INTCON3 register, 1050
- IntDeb.asm application,
826–832
- Integer
 - convert ASCII code integer to
character, 1095
 - convert character to ASCII,
1095
 - convert floating-point number
to, 1093
 - convert hexadecimal string to,
1138
 - convert real number to, 1091
 - convert string to, 1095, 1134
 - convert string to signed, 1139
 - convert to string, 1145
 - define variable as, 1093
 - split value into fraction and,
1135
- Integer data, 1090
- Integrated development environ-
ment (IDE), 39, 41, 81–83
- Intel, 25
- Intel 8051, 4, 12, 18
- Intel format, 391
- Interface commands,
BASIC87x, 95
- Interface length, 674
- Interfaces, 50
- Interfacing PIC MCU with
three-wire RS, 841–851
- Interfacing to external devices.
See external devices,
interfacing to
- Inter-intercomputer
communication (I2C), 4, 5,
1219
- Interlaced video, 946
- Internal data define, 93
- Internal hardware, 2–5, 487
- Internal operating parameters,
1105–1106
- Internal oscillators, 57–58,
454–455
- Internal reset capabilities, 269,
487, 609–610
- Internet, 34
- Interpreter, 1218
- Interrupt(s), 1218
 - and breakpoints, 238, 239
 - change on, 476, 477
 - configure, 1145
 - debouncing inputs with,
826–832
 - enable, 1113
 - operation of, 483–485
 - in PIC18 RTOS design, 548
 - return nonzero if reset caused
by wakeup through, 1144
 - simulating input pin interrupt
with timer pin input,
815–819
 - TMR0 interrupt for LED
PWM dimming, 819–826
- INTERRUPT (BASIC87x), 94
- Interrupt enable flags, 483
- Interrupt handler vector, 17
- Interrupt handlers, 15–17, 272,
1107–1108
 - in assembly language,
419–423
 - context-saving, 419–421
 - disable, 1113
 - enable request hardware, 1145
 - guidelines for writing, 419
 - hardware, 15–17
 - jump to, 1115
 - in mid-range PIC
microcontrollers,
341–344
 - with PIC microcontrollers,
271–273
 - in PIC18 microcontrollers,
292, 370–371
 - software, 15
 - without context saving,
421–423
- Interrupt instructions, 14–17
- Interrupt latency, 484
- Interrupt mask, 15
- Interrupt mechanism, 1142
- Interrupt request enable bit (E
bit), 271
- Interrupt request flag, 271
- Interrupt request hardware, 1145
- Interrupt request sources, 1142
- Interrupt service routine, 16, 272
- Interrupt template code,
1107–1108
- Interrupt vector address, 281
- Intrinsic current, 604
- Inverted bitwise value, 1110
- I/O (input/output)
 - Analog. *See* analog I/O
 - device interfacing for. *See* I/O
device interfacing
 - digital, 486
 - memory-mapped, 611
 - serial. *See* serial I/O
 - simulated open-collector/
open-drain, 615–616
- I/O bit banging, 1149
- I/O device interfacing, 661–710
 - analog I/O, 682–690
 - asynchronous NRZ serial
interfaces, 693–704
 - audio output, 690–692
 - LCDs, 672–682
 - LEDs, 661–665
 - matrix keypads, 668–672
 - relays/solenoids, 692–693
 - switch bounce, 665–668
 - synchronous serial interfaces,
704–710
- I/O function registers, 13
- I/O hardware registers, 249,
250
- I/O pins, 6
 - functions of, 17–21
 - optimizing, 441
 - options and function of, 3
 - of PIC microcontrollers, 50
 - reading from/writing to,
1149–1150
 - simulating microprocessor bus
with, 4
- I/O PINS register, 249
- I/O port pins, 1103, 1145
- I/O register addresses,
1016–1052
 - low-end PIC microcontrollers,
1016–1019
 - mid-range PIC microcon-
troller registers, 1017,
1019–1034
 - PIC18, 438, 1030, 1035–1052
- I/O registers, 245
- I/O space, 1218
- I/O with interrupts application
code, 810–832
 - debouncing inputs with
interrupts, 826–832

- I/O with interrupts application
 - code (*Cont.*):
 - simulating input pin interrupt with timer pin input, 815–819
 - timer/interrupt handler with context saving, 810–815
 - TMR0 interrupt for LED PWM dimming, 819–826
- IORLW, 325, 983, 988, 1001
- IORWF, 325, 983, 988, 1002
- iPath (specify new directory path), 1099
- IPR1 register, 1038
- IPR2 register, 1039
- IR robot project, addendum, 898–903
- IRBetter project, 898–903
- IRP bit, 129
- ISA card, 924, 926
- ISO9000 certification, 595
- J-led devices, 37
- JTAG interface, 80–81
- Jump on status flag condition, 335
- Jump to functions, 14, 755–770
 - in BASIC, 1090, 1091
 - in C programming language, 1126
 - conditional, 760–766
 - execution control state machines, 766–770
 - in MS-extended BASIC, 1094
 - outside of current execution page, 1135
 - in PICBASIC, 1112, 1113, 1115
 - between program memory pages, 756–760
- Keep variable within range (macro), 1158
- Kernel, 538
- Key matrix project, 872–877
- Key project, 872–877
- Keypads, 668–672, 1220
- KHz, 1219
- Kickback diode, 713
- KILL (MS-extension BASIC), 1097
- Kiryashov, Dmitry, 1157, 1158
- K[omega], 1219
- L293D chip, 896
- Labels, 93, 130, 1219
 - in assembly language, 117, 122–123, 378–379, 497
 - in C programming language, 1127, 1129
 - external, 131
 - global, 132
 - PICBASIC, 1100–1101
- Last file register address, 133
- Last in, first out (LIFO), 27, 1219
- LATA, 1036
- LATB, 1036
- LATC, 1036
- LATCH register, 474
- LATD, 1036
- LATE, 1036
- Latency, 1219
- LBOUND (MS-extension BASIC), 1093
- LCALL, 758, 980, 1013, 1014
- LCASE\$ (MS-extension BASIC), 1095
- LCD(s) (liquid-crystal displays), 1219
 - interfacing with, 672–682
 - send to, 1114
 - specify data area address, 1147
 - two-wire control for, 680–682
 - types/character locations for, 676
 - ultrasonic ranging project, 866–872
 - as user interfaces, 598, 599
 - write byte to, 1148
- LCD character, 1147
- LCD controller
 - initialize, 1146
 - read data byte from, 1146
 - return address byte from, 1146
 - return nonzero if busy, 1139
 - write command to, 1148
 - write string to, 1146
- LCD interface macro(s), 1164–1185
 - LCD2, 1179–1185
 - LCD4, 1173–1179
 - LCD8, 1164–1167
 - LCD8Po11, 1167–1173
- LCD interface project, 859–866
- LCD ranging project, ultrasonic, 866–872
- LCD_BITS (PICBASIC), 1106
- LCDCHAR (subroutine), 1166–1167, 1170, 1176–1177, 1182–1183
- LCD_DBIT (PICBASIC), 1106
- LCD_DREG (PICBASIC), 1106
- LCD_EBIT (PICBASIC), 1106
- LCD_EREG (PICBASIC), 1106
- LCDINIT (subroutine), 1167, 1170–1172, 1177–1179, 1183–1185
- LCDINS (subroutine), 1166, 1169–1170, 1174–1176, 1180–1182
- LCD_LINES (PICBASIC), 1106
- LCDNYBBLE (subroutine), 681–682
- LCDOUT (PICBASIC), 1114
- LCDPORTINIT (subroutine), 1164–1166, 1169, 1174, 1180
- LCD_RSBIT (PICBASIC), 1106
- LCD_RSREG (PICBASIC), 1106
- LDIV_T LDIV (PICC), 1135
- Lead centers, 37, 38
- Lead frame, 36
- Lead frame pins, 37
- Lead-free, 1219
- Leading blanks, 1095
- Least macro, 417
- LED(s) (light-emitting diodes), 1219
 - assembler-flashing, 152–154
 - flashing, 149–154
 - interfacing with, 661–665
 - multisegment, 663–665
 - PICC-Lite flashing, 151–152
 - as user interfaces, 598
- LED displays project(s)
 - electronic thermometer with seven segment, 903–910
 - multisegment, 663–665
- LED PWM dimming, 819–826
- LED series increment application, 544–554
- LedPWM.asm application, 819–826
- LEFT\$ (MS-extension BASIC), 1095
- LEN (MS-extension BASIC), 1095
- Length of string, 1095
- Less than, 1092, 1111, 1127
- Less than or equal to, 1092, 1111, 1127
- LET (BASIC), 1090
- LCDCHAR (subroutine), 1166–1167, 1170, 1176–1177, 1182–1183
- LCD_DBIT (PICBASIC), 1106
- LCD_DREG (PICBASIC), 1106
- LCD_EBIT (PICBASIC), 1106
- LCD_EREG (PICBASIC), 1106
- LCDINIT (subroutine), 1167, 1170–1172, 1177–1179, 1183–1185
- LCDINS (subroutine), 1166, 1169–1170, 1174–1176, 1180–1182
- LCD_LINES (PICBASIC), 1106
- LCDNYBBLE (subroutine), 681–682
- LCDOUT (PICBASIC), 1114
- LCDPORTINIT (subroutine), 1164–1166, 1169, 1174, 1180
- LCD_RSBIT (PICBASIC), 1106
- LCD_RSREG (PICBASIC), 1106
- LDIV_T LDIV (PICC), 1135
- Lead centers, 37, 38
- Lead frame, 36
- Lead frame pins, 37
- Lead-free, 1219
- Leading blanks, 1095
- Least macro, 417
- LED(s) (light-emitting diodes), 1219
 - assembler-flashing, 152–154
 - flashing, 149–154
 - interfacing with, 661–665
 - multisegment, 663–665
 - PICC-Lite flashing, 151–152
 - as user interfaces, 598
- LED displays project(s)
 - electronic thermometer with seven segment, 903–910
 - multisegment, 663–665
- LED PWM dimming, 819–826
- LED series increment application, 544–554
- LedPWM.asm application, 819–826
- LEFT\$ (MS-extension BASIC), 1095
- LEN (MS-extension BASIC), 1095
- Length of string, 1095
- Less than, 1092, 1111, 1127
- Less than or equal to, 1092, 1111, 1127
- LET (BASIC), 1090

- LET (PICBASIC), 1114
- Letter suffixes (for PIC microcontrollers), 53, 54
- Level conversion, 1219
- Level-sensitive, 1219
- LFSR, 1002
- LGOTO, 758, 1014
- LGOTO macro, 505
- LGOTO macro, 509–510
- Library(-ies), 526
 - and C for PIC microcontrollers, 101
 - converting object files into, 77
 - specify different, 1099
- lLibrary (specify different library), 1099
- Library file (.lib), 117–118
- Library routines, 520, 1109
- Life expectancy, 607–608
- LIFO (last in, first out), 1219
- Light, speed of, 1061
- Line, draw, 1096
- LINE (MS-extension BASIC), 1096
- Line continuation, 93
- Line feed (LF), 1068, 1130
- LINE INPUT (MS-extension BASIC), 1095
- Linker, 1219
- Linker file (.lkr), 113, 117, 522–526
- Linkers, 76–77
- Linking, 519–526
 - defined, 520
 - files used in, 522–523
 - and global variables, 523
 - and linker file, 522–526
 - in modern applications, 520–521
 - and MPLIB librarian, 526
 - sample, 521–522
- Liquid-crystal display (LCD), 1219
- LIST (MPASM), 132–133
- LIST R-DEC (MPASM), 125
- Listing file (.lst), 40, 113–117, 523–526, 529–530, 1099
- Lists
 - search, 1114
 - store value at offset, 1114
- Literal data, 298
- Lithium ion, 1219
- Load file, 1114, 1129
- Loading file, 40
- LOCAL (MPASM), 133
- Local variables, 86–88, 133
- LOCATE (MS-extension BASIC), 1096
- LOG (BASIC), 1091
- Logarithm, 1091, 1133, 1135
- Logic analyzers, 1086–1087, 1219
- Logic gate, 1219
- Logic instructions, 324–328
- Logic probe, 1219
- Logic probes, 1081–1083
- Logic simulation, 423–426
- Logical AND, 497
- Logical negation, 1127
- Logical operations/operators
 - in C programming language, 1127
 - in PICBASIC, 1108, 1111
- Logical OR, 497
- Logitech, 445, 449, 603
- LONG ATOI (PICC), 1134
- LONG ATOL (C18 C), 1139
- Long integer, 1090
 - convert string to, 1134, 1139
 - convert to string, 1145
 - define variable as, 1093
- LONG XTOI (PICC), 1138
- LOOKDOWN (PICBASIC), 1114
- LOOKDOWN2 (PICBASIC), 1114
- LOOKUP2 (PICBASIC), 1115
- Loop, 1091, 1094, 1113
- Loop control, 337–338
- Loop delay (macro), 1154–1155
- Looping, 402–405, 1126
- Loose standard, 709
- LOW (low 8 bits), 390–392, 496
- LOW (MPASM), 124
- LOW (PICBASIC), 1115
- Low logic level, 1115
- Low-end PIC MCU instruction set, 348–356
 - data movement, 349–352
 - data processing, 352–353
 - execution change, 353–356
 - processor control, 356
 - size of, 349
- Low-end PIC microcontrollers, 60–61
 - device pinouts for, 1031
 - instruction set for, 980–985
 - I/O register addresses for, 1016–1019
 - OPTION and TRIS registers, 278–279
 - OPTION register in, 350–351
- Low-end PIC microcontrollers (Cont.):
 - parallel programming of, 160–163
 - porting assembler applications from mid-range PIC MCUs, 430–435
 - processor architecture of, 274–279
 - program counter, 277–278
 - program memory size/reset vector address, 274
 - projects for, 853–877
 - register access, 275–276
 - RTOS, 541–542
- Lower case, 1133
 - convert from, 1147
 - convert to, 1095, 1133, 1138, 1147
 - return nonzero if, 1144
- Lower parameter, 1110
- LOWGOTO application, 756–758
- Low-value resistor, 714–715
- Low-voltage detect, 1144
- LPRINT (MS-extension BASIC), 1097
- .lst file, 113
- LTRIM\$ (MS-extension BASIC), 1095
- Luminance, 945
- LVDCON register, 1047–1048
- Macro(s), 1160–1200, 1219
 - advantages of using, 490–492
 - button debouncing, 1198–1200
 - declaring, 133
 - enable printing expansions in, 131
 - end statement of, 131
 - errors in, 117
 - forced stop in, 131
 - LCD interfaces, 1164–1185
 - mid-range I2C interface, 1194–1198
 - NRZ serial I/O, 1186–1194
 - simple delay, 1160–1163
 - for sixteen-bit numbers. *See* sixteen-bit numbers
 - subroutines vs., 489, 490
- MACRO (MPASM), 133

- Macro development, 489–518
 - in application debug, 507–509
 - and assembler calculator, 494–499
 - conditional assembly/compilation, 500–509
 - debugging, 509–513
 - and defines, 492–494, 507–509
 - multiline C, 499–500
 - in PIC MCU assembly-language, 489–492
 - structured programming, 513–518
- Macro expansion, 133
- Magnitude of number, 1110
- Make files, 76, 77, 117
- MALLOC (C), 1132
- Manchester encoding, 31, 558, 1219
- Map file, generate, 1099
- Map file (.map), 118
- MaryaBas project, 913–922
- Marya's toy project, 913–922
- MaryaToy project, 910–913
- Mask programmable ROM, 1220
- Mask ROM, 23
- Master, 29, 635, 636, 1220
- Master clear (MCLR) pin, 58, 460–461, 1144
- Master SSP (MSSP), 638–643, 1025, 1027, 1028
- Master/slave remote thermometer interface project, 922–931
- Mathematical conversions, 1066–1067
- Mathematical formulas, 1065–1066
- Mathematical operations/operators
 - in C programming language, 1127
 - in PICBASIC, 1108–1111
- Matrix keyboard, 1220
- Matrix keypads, 668–672, 1220
- MAX (PICBASIC), 1110
- Maxim MAX232, 699, 700
- Maximum array sizes, 1103
- MAXRAM (MPASM), 133
- MCP2510 CAN bus controller, 1139–1142
- Memory, 1220
 - allocate, 1132, 1133
 - compare two blocks of, 1135
 - and compilers, 84
 - electrically erasable PROM, 25
 - erasable PROM, 23–25
 - external, 463–466
 - free, 1133
 - LIFO, 27
 - in low-end PIC microcontrollers, 349, 350
 - nonvolatile program, 21–22
 - program, 21–22
 - programmable read-only, 23
 - read-only, 22, 23
 - stack, 27
 - static random-access, 25–26
 - types of, 21–27
 - variable, 22
 - volatile, 26
 - volatile variable, 22
- Memory areas, 13
- Memory array, 1220
- Memory management unit (MMU), 543
- Memory protection, 463
- Memory-mapped I/O, 611, 1220
- Mesh networks, 34
- Message byte, 547
- Message string, 133
- Messages, 70, 117
- MESSG, 512–513
- MESSG (MPASM), 133
- Metal oxid silicon field effect transistor (MOFSET), 1078–1079
- MHz, 1220
- MIC-II project, 934–943
- Microchip, 41, 43–48, 54, 104–105, 155, 211, 229, 659
- Microchip C18 functions, 1138–1148
- Microchip Direct, 965–966
- “Microchip Masters” seminars, 966
- Microchip programmers, 178–181
 - MPLAB PM3 Universal Device Programmer, 180–181
 - PICSTART Plus, 178–180
- Microchip special instruction mnemonics, 980, 1013–1016
- Microchip Technology, 965–966
- Microcoded processors, 10–11
- Microcontroller application software development, 7
- Microcontroller communication, 28–35
 - network, 32–35
 - point-to-point, 28–32
- Microcontroller project, 5–6
- Microcontrollers (MCUs), 1–41, 1220
 - and application development tools, 39–41
 - applications for, 5–7
 - architectures of, 7–12
 - built-in features of, 3
 - communication with, 28–35
 - and device packaging, 35–39
 - with external memory, 4
 - with external support, 2
 - instructions/software for, 12–17
 - internal hardware of, 2–5
 - memory types for, 21–27
 - microprocessors vs., 2
 - peripheral functions of, 17–21
 - power levels for, 6
 - primary role of, 1
 - types of, 1
- MicroEngineering Labs (meLabs), 137, 204, 1091, 1099
- Microprocessor (MPU), 2, 1220
- Microsoft, 740, 746, 1093–1098
- Microsoft's Visual Studio, 41
- Microsoft Windows protect mode, 1099
- Microwire, 1220
- Microwire devices, 1142, 1143, 1146, 1148
- Microwire protocol, 705, 706
- MID\$ (MS-extension BASIC), 1095
- MidGoto application, 759
- Mid-range I2C interface (macro), 1194–1198
- Mid-range PIC MCU instruction set, 303–348
 - data movement, 304–316
 - data processing, 316–328
 - execution change, 328–344
 - processor control, 345–348

- Mid-range PIC microcontrollers, 61
 - bootloaders for, 530–535
 - built-in EEPROM/Flash
 - access in, 618–624
 - device pinouts for, 1032–1034, 1053–1058
 - instruction set for, 980, 986–990
 - I/O register addresses for, 1017, 1019–1034
 - OPTION register in, 471
 - part number to feature table for, 979
 - porting assembler applications
 - low-end PIC MCUs, 430–435
 - projects for, 878–953
 - RTOS, 541–542
- Millions of instructions per second (MIPS), 1220
- MIN (PICBASIC), 1110
- Minimum.asm, 121
- MIPS (millions of instructions per second), 1220
- Mnemonics, Microchip special instruction, 980, 1013–1016
- MOD (MS-extension BASIC), 1093
- Model train light control with hall-effect sensor project, 854–859
- Modulus, 496, 1093, 1110, 1127, 1128
- MOFSET driver transistor, 718, 719
- Momentary-off switches, 1074, 1076
- Momentary-on switches, 1074, 1076
- Monitor, 1220
- Motor control, 711–738
 - dc motors, 711–724
 - R/C servo control, 733–738
 - stepper motors, 724–733
- Motorola 68000, 11
- Motorola format, 391
- Mouse, 2
- MOVF, 305–308, 357–361, 984, 988, 1002, 1149
- MOVFF, 358–360, 1003
- MOVFW, 1014, 1015
- MOVLB, 358, 1003
- MOVLW, 304–306, 988, 1003
- MOVWF, 307, 308, 984, 988, 1003, 1149
- MPASM assembler, 69, 70, 120–135, 501, 980, 1099
 - comments in, 127–128
 - configuration fuse registers in, 125–127
 - constant declarations in, 124–125
 - labels in, 122–123
 - numbering system in, 122, 127
 - readability of, 122
 - required statements in, 122
 - variable declarations in, 123–124
- MPLAB, 1100
- MPLAB ICD 2 debugger, 214–219
 - hardware design for, 218, 219
- ICSP vs., 217–218
- programming, 217
- MPLAB ICE-2000 emulator, 210–213
- MPLAB IDE, 41, 63, 64, 103–154
 - desktop of, 118–120
 - file types, 107–118
 - installing, 103–108
- MPLAB IDE application build tools, 120–141
 - C compiler for PIC18, 135
 - MPASM assembler, 120–135
 - MPLINK, 139–141
 - PICBASIC, 137–139
 - PICC compiler, 136–139
- MPLAB IDE simulator, 78–79, 294–297
- MPLAB PM3 Universal Device Programmer, 180–181
- MPLAB REAL ICE emulator, 213–214
- MPLAB SIM, 142
- MPLIB librarian, 526
- MPLINK linker, 139–141
- MPSIM, 63, 64
- MPU. *See* Microprocessor
- Ms, 1220
- MSComm serial control, 746–749
- MS-DOS, 1099
- MULFW, 365
- MULLWF, 1003
- Multidimensional arrays, 1124
- Multiline C macros, 499–500
- Multimaster, 1220
- Multimastering, 707, 710
- Multiplexor (mux), 1220
- Multiplication, 365, 366, 496, 1092, 1110
 - in assembly language, 392–397
 - in C programming language, 1127, 1128
 - in mid-range PIC microcontrollers, 327
 - in PIC18 devices, 285–286
 - with sixteen-bit numbers, 1207–1210
- Multiplier, 285–286
- Multiply macro, 396–397
- Multipole switches, 1074, 1077
- Multitasking operating system, 537
- Multi-threaded operating system, 537
- MULWF, 365, 366
- [mu]s, 1226
- Music project, 883–891
- Mykemacs.inc, 1160
- MyLGoto macro, 757, 759
- NAME (MS-extension BASIC), 1097
- Naming conventions
 - BASIC, 1089, 1090
 - PICBASIC, 1101
- Nanowatt devices, 606
- NAP, 346
- NAP (PICBASIC), 1115
- National Television Standards Committee (NTSC), 944, 1221
- Native interface, 85
- Natural exponent, 1133
- Natural logarithm, 1091, 1133, 1135
- NCD (PICBASIC), 1111
- N-channel devices, 1078, 1079
- N-channel MOFSET transistors, 1079
- N-doped silicon, 1078
- Negate contents of file register (macro), 1150
- Negate contents of w register (macro), 1150
- Negation, 365, 366, 496
 - in assembly language, 388–389

- Negation (*Cont.*):
 in C programming language, 1127
 in mid-range PIC microcontrollers, 324
- Negative active logic, 1220
- Negative borrow flag, 259
- Negative power (V_{ss}), 3
- Negative terminal (GND or V_{ss}), 1073
- Negative voltage, 700
- Negatively active, 268
- NEGF, 365, 366, 1014–1016
- NEGW, 1003–1004
- Nesting, 444, 503, 1220
- Nesting interrupts, 17
- Net, 1221
- Network communication, 28, 32–35
- Neutron rest mass, 1061
- Never return zero (NRZ), 695
- NEXT (BASIC87x), 93
- NEXT (PICBASIC), 1113
- NiCad (nickel-cadmium) batteries, 1221
- Nickel-cadmium (NiCad) batteries, 1221
- NMOS logic, 1221
- Nodes, 32
- NOEXPAND (MPASM), 133
- Noise, 1221
- NOLIST (MPASM), 133
- Nonconditional instructions, 14
- Nondisclosure agreement (NDA), 155
- Nonlinear programming, 768
- Nonrecurring expense (NRE), 23
- Nonreturn to zero (NRZ), 1025
- Non-return to zero (NRZ) data packet, 30, 31
- Nonvolatile program memory, 21–22
- NOF, 347–348, 370–371, 407, 512, 984, 989, 1004
- NOT, 1127, 1221
- NOT AND (PICBASIC), 1111
- Not equal to, 1092, 1111, 1127
- NOT OR (PICBASIC), 1111
- NOT XOR (PICBASIC), 1111
- Notes, audio, 1062, 1097
- NPN transistor, 713–714
- NRZ asynchronous serial communications, 1025
- NRZ serial I/O macro(s), 1186–1194
- NRZ serial I/O macro(s) (*Cont.*):
 NRZSerialI, 1189–1194
 NRZSerialNI, 1186–1188
 NRZSerialNISetup, 1188–1189
- Ns, 1221
- NTSC (National Television Standards Committee), 1221
- NTSC video output project, 944–953
- NUL character, 339
- Null (NUL), 1068
- Numbering system, 122, 127
- Numeric characters, 1133
- Nybble, 259, 1151–1152
- Object code, 40
- Object file (.obj), 40, 76–77, 113, 520–522, 1221
- Octal numbers, 1130, 1221
- od (generate listing, symbol table, map files), 1099
- Offset
 hex, 156–158
 return value at, 1114
- Ohm's law, 1063
- o1 (generate listing file), 1099
- ON... GOTO (BASIC), 1091
- ON DEBUG GOTO (PICBASIC), 1115
- ON ERROR GOTO (MS-extension BASIC), 1094
- ON INTERRUPT (PICBASIC), 1107
- ON INTERRUPT GOTO (PICBASIC), 1115
- OnComm events, 749
- 123 *PIC Microcontroller Experiments for the Evil Genius* (Predko), 903
- One's complement, 1221
- One-time programmable (OTP), 24–25, 36, 50, 51, 1222
- Ongoing reliability testing (ORT), 1221
- Op codes, 1221
- OPB804 optoisolator, 955–957
- OPEN... FOR... AS (MS-extension BASIC), 1098
- OPEN (MS-extension BASIC), 1098
- Open file, 1131
- Open-collector/drain output, 615–616, 1221
- Open-source tools, 742
- Operating capabilities, 50
- Operating features, 445–488
 configuration fuses, 451–469
 interrupt operation, 483–485
 OPTION register, 470–478
 power input/decoupling, 446–451
 and selecting PIC MCU, 485–488
 TMR0, 478–483
- Operating specification, 1090
- Operating systems, 537–539
- Operating voltage, 49
- Operators
 assembler-calculator, 496, 498
 BASIC, 1092
 BASIC87x, 92, 94
 C programming language, 1127
 PICBASIC, 1109–1111
- Optimization
 in assembly language, 438–443
 and compilers, 85
- Optimized linking, 85
- OPTION, 308–310, 984, 989, 1016
- OPTION (BASIC), 1090
- OPTION register, 249, 275, 470–478, 1016–1020
 low-end, 470
 in low-end PIC MCUs, 278–279, 350–351, 470
 in mid-range PIC MCUs, 471
 parallel I/O, 472–477
 PIC12C50x/PIC16C505 pin accessing, 477–478
- Optoisolator, 956
- OR, 1111, 1221
 in assembly language, 390
 bitwise, 496
 in C programming language, 1127, 1128
 logical, 497
 in mid-range PIC microcontrollers, 325–326
 in MS-extension BASIC, 1093
 in PICBASIC, 1110, 1111
- OR (MS-extension BASIC), 1093
- OR (PICBASIC), 1111

- Order of operations, 71, 72
 - BASIC, 1092
 - C programming language, 1128
 - PICBASIC, 1109
- ORG (MPASM), 133
- ORG (PIC MCU instruction), 330
- ORG 0 (MPASM), 125
- ORT (ongoing reliability testing), 1221
- Orthogonality, 8
- osc (PICBASIC), 1106
- OSCCAL register, 1017, 1019, 1022
- OSCCAL_1K (PICBASIC), 1106
- OSCCAL_2K (PICBASIC), 1106
- OSCCON, 1048
- Oscillator type, 49, 457
- Oscillators, 6, 454–460, 1100, 1221
 - external crystals/resonators and, 456–458
 - and external resistors/capacitors, 455–456
 - internal, 454–455
 - PIC18 circuitry, 458–460
- Oscilloscopes, 774, 775, 1084–1085, 1222
- OSCON register, 460
- OTP. *See* one-time programmable
- Output
 - ASCIIZ string to console, 1136
 - audio, 690–692
 - character to console, 1136
 - combining input and, 613–615
 - to console, 1090
 - data to file one character at a time, 1132
 - of one character, 1131
 - of string text, 1131
 - string to console, 1134
 - string to file, 1132. *See also* I/O
- OUTPUT (PICBASIC), 1115
- Output buffer full (OBF), 658
- Output data, 94
- Output frequency, 1113
- Output mode, 251, 473
- Output pin, 1113
- Output set/reset (Boolean arithmetic), 1065
- Output string of tones, 1120
- Output touch tone sequence, 1113
- Overrun error, 647
- P18InsTemplt.asm, 356–357
- PA bits, 278
- Packaging, 50–53
- Packet, 694
- Packet of bits, 693
- Page, 266, 377
- PAGE (MPASM), 133
- Page boundaries, 444
- Page break, 133
- Page size, 266
- PAGESEL (MPASM), 133
- Parallax BASIC Stamp (PBASIC) language, 89, 137, 934–935, 1091, 1109
- Parallel, 28, 1222
- Parallel capacitance formula, 1064
- Parallel integrating differential (PID), 1222
- Parallel I/O, 472–477
- Parallel port adapter, 206
- Parallel port for PC interfacing, 749–754
- Parallel programming, 159–166
 - early, 159–160
 - low-end, 160–163
 - PIC17, 163–166
- Parallel resistance formula, 1063
- Parallel slave port (PSP), 657–659, 922–924, 1021, 1023
- Parallel-bus devices, 612–613
- Parallelism, 9
- Parameter variables, 382
- Parameters, 381–385, 1222
- Parentheses, 497
- Parity bit, 30, 31, 695
- PARITYLABEL, 1113
- Parsec, 1061
- Part numbers
 - of PIC microcontrollers, 53
 - tables of feature and, 977–979
- Part summary, 47
- Part suppliers, 973–975
- Partial emulators, 241
- Parts per million (ppm), 1222
- Passive components, 1222
- Patch code, 347–348
- Patch space, 372, 405–407
- PAUSE (PICBASIC), 1115
- PAUSEUS (PICBASIC), 1115
- Payson, John, 1158
- PBASIC, 89, 90, 137, 1091
- PC interfacing, 739–754
 - parallel port for, 749–754
 - serial port for, 742–749
 - software application development tools for, 740–742
- PC latch register, 302, 303
- PC software application development tools, 740–742
- PCA (printed-circuit assembly), 1222
- PCB (printed-circuit board), 1222
- P-channel MOFSET, 1080
- P-channel MOFSET transistors, 1079, 1080
- PCKATHU, 1051
- PCL register, 249, 250, 252
- PCLATH register, 249, 250, 252, 336, 980, 1013, 1014, 1016, 1019, 1021, 1051
- PCON register, 270, 1019, 1021
- PCs
 - microcontroller applications within, 2
 - as RTOSs, 538–541
- PCTherm project, 922–931
- .pdf, 1222
- P-doped silicon, 1078
- PEEK (PICBASIC), 1115
- Pentium processor, 9
- People costs, 600
- Perfect gas law formula, 1065
- Period, 685, 1222
- Periodicals, 971–972
- Peripheral functions, 17–21
- Peripherals, acknowledging, 1139
- Permeability of vacuum, 1062
- Permittivity of free space, 1061
- Persistent variable, 1135
- Phantom register, 263
- Phase errors, 503–504
- Phased-locked loop circuitry, 246
- Physical constants, 1061–1062
- Pi, 1061
- PIC (peripheral interface controllers), 159
- PIC 16C505, 860

- PIC MCU instruction set, 293–372
 - data movement, 298–300
 - data processing, 300–301
 - execution change, 301–303
 - functional groups of, 293–294
 - instruction types, 297–298
 - low-end. *See* low-end PIC MCU instruction set
 - mid-range. *See* mid-range PIC MCU instruction set
- PIC18. *See* PIC18 MCU instruction set
- setting up MPLAB IDE simulator with test template, 294–297
- PIC MCU I/O pins, 1146
- PIC MCU processor
 - architecture, 49, 243–292
 - ALU, 254–259
 - CPU, 244–248
 - data movement, 260–264
 - differences in, 273–292
 - hardware and file registers, 248–254
 - interrupts, 271–273
 - low-end, 274–279
 - PIC17, 279–282
 - PIC18, 282–292
 - program counter/stack, 264–268
 - reset, 268–271
- PIC microcontrollers, 43–62
 - configuration fuse register of, 55–57
 - families of, 59–62
 - features/peripherals of, 48–50
 - in-circuit serial programming/built-in debugger of, 58–59
 - internal oscillator calibration of, 57–58
 - letter suffixes for, 53, 54
 - low-end, 60–61
 - and Microchip Web site, 43–48
 - mid-range, 61
 - packaging options with, 50–53
 - part number conventions/ordering for, 53
 - PIC17 devices, 61–62
 - PIC18 devices, 62
 - pin selections for, 167
 - selection of, 485–488
 - unique features of, 54–59
- PIC12C6xx microcontrollers, 1054
- PIC12C50x microcontrollers, 477–478, 1031, 1053
- PIC12F6xx microcontrollers, 1054
- PIC12F50x microcontrollers, 1053
- PIC16C54 microcontrollers, 349, 1053
- PIC16C55 microcontrollers, 1054
- PIC16C57 microcontrollers, 1054
- PIC16C61 microcontroller, 244
- PIC16C84 microcontroller, 251
- PIC16C92x LCD driver microcontrollers, 1034, 1058
- PIC16C505 microcontrollers, 477–478, 1031, 1053
- PIC16C554 microcontrollers, 349
- PIC16F54 microcontrollers, 1053
- PIC16F69x microcontrollers, 1054
- PIC16F84 microcontrollers, 251, 254, 1099
- PIC16F505 microcontrollers, 1053
- PIC16F684, 149, 150
- PIC16F690, 487
- PIC16F877, 466–468
- PIC16HV540, 449–451
- PIC17 microcontrollers, 61–62, 273, 279–282
 - external memory in, 464
 - in-circuit serial programming schematic, 166
 - processor architecture of, 279–282
 - programming of, 163–166
- PIC17C44, 180
- PIC18 MCU instruction set, 356–372
 - data movement, 357–364
 - data processing, 364–366
 - execution change, 366–371
 - processor control, 371–372
- PIC18 microcontrollers, 62
 - addressing in, 377–378
 - architecture of, 282–292
 - block diagram of, 283
 - bootloaders for, 535–536
- PIC18 microcontrollers (*Cont.*):
 - configuration fuses in, 469
 - device pinouts for, 1034, 1058–1060
 - external memory in, 464
 - instruction set for, 980, 991–1012
 - interrupts in, 292
 - I/O register addresses for, 1030, 1035–1052
 - porting assembler applications to, 435–438
 - program counter in, 288–292
 - programming of, 173–177
 - projects for, 953–963
 - referencing instruction operation in, 284
 - register access/bank addressing in, 286–288
 - RTOS design in. *See* PIC18 RTOS design
- PIC18 oscillator circuitry, 458–460
- PIC18 RTOS design, 542–563
 - access bank file registers, 543
 - bill of materials, 550
 - context register saves in, 542–543
 - digital clock/thermometer bill of materials, 555
 - fast stack in, 543
 - interrupts supported in, 548
 - MMU in, 543
 - multitasking application, 553, 555–563
 - PCs vs., 538–541
 - simple application, 544–554
 - status and message byte, 547
 - task context registers, 546
 - task request definitions, 549
- PIC18F877, 453, 454
- PIC18F1320, 487
- PIC18F2550, 45, 46
- PIC14000 microcontrollers, 1034, 1057
- PICBASIC, 88–90, 137–139, 1091, 1092, 1099–1121
 - advantages/disadvantages of, 922
 - built-in functions, 1112–1121
 - command line options, 1099
 - data types, 1104
 - default configuration fuse register settings, 1100

- PICBASIC (*Cont.*):
 - internal operating parameters, 1105–1106
 - logical operations, 1111
 - Marya's toy project, 913–922
 - mathematical/bit operations, 1110–1111
 - maximum array sizes, 1103
 - pin specifications, 1103
- PICBASIC PRO, 1092, 1099
- PICBASIC PROK, 137, 138
- PICC compiler, 136–139
- PICC functions, 1133–1138
- PICC language, 1133–1138
- PICC-Lite compiler. *See* PICC compiler
- PICC-Lite flashing LED, 151–152
- PICList, 458
- PICMASTER emulator, 213
- PICmicro, 1099
- PICSTART Plus (PSP or PS+) programmer, 178–180
- PID (parallel integrating differential), 1222
- PIE register, 1021
- PIE1 register, 1037
- PIE2 register, 1038
- Pin input, 1114
- Pin output, 1113, 1115
- Pin stimulus file, 832–841
- Pin through hole (PTH), 37, 38, 1031, 1223
- Ping, 1222
- Ping character, 700–701
- Ping command, 192
- Pins
 - accessing, 477–478
 - for PIC MCU devices, 167
 - PICBASIC specifications for, 1103
- PIR register, 1021
- PIR1 register, 1037
- PIR2 register, 1038
- Planck constant, 1061
- Plastic encapsulants, 35, 36
- PLAY (MS-extension BASIC), 1097
- PLUSW register, 362–364
- PLUSW0, 1050
- PLUSW1, 1049
- PLUSW2, 1049
- PNP bipolar transistor, 1078, 1079
- Pod, 1222
- Point on display, 1096
- Pointers, 99, 409–411
- Point-to-point communication, 28–32
- Poisoned work environment, 602
- POKE (PICBASIC), 1115
- Polaroid 6500, 866–870
- Poll, 1222
- Polling, 30, 666–667, 674
- Polyester caps, 446, 447
- Polynomial, 1134
- Pop, 27, 72–75, 291, 383, 1222
- POP, 1004
- Pop macro, 96
- PORT register, 249
- PORTA, 1027
- PORTA register, 1035
- PORTB internal pull-up resistors, 1143, 1145
- PORTB interrupts, 1145
- PORTB pin, 1142
- PORTB register, 1035, 1142
- PORTC register, 1035
- PORTD, 1021
- PORTD register, 1036
- PORTE, 1021
- PORTE register, 1036
- Porting assembler code between
 - architectures, 430–438
 - from mid-range to low-end, 430–435
 - to PIC18, 435–438
- POS (MS-extension BASIC), 1096
- Positive active logic, 1222
- Positive power (Vdd), 3
- Positive terminal (Vcc or Vdd), 1073
- POSTDEC register, 362, 363
- POSTDEC0, 1050
- POSTDEC1, 1050
- POSTDEC2, 1049
- POSTINC register, 362
- POSTINC0, 1050
- POSTINC2, 1049
- pot, 933
- POT (PICBASIC), 1116
- Potentiometers, 683–685, 719, 721, 722, 734, 1074, 1075, 1116
- Power
 - for dc motors, 714
 - formula for, 1063
 - reduction in, 54
- “Power, clocks, reset,” 445
- Power (exponentiation), 1133, 1135
- Power input, 446–451
- Power levels, 6
- Power load, 1073, 1074
- Power management, 603–608
 - decoupling capacitors, 603–604
 - estimating power requirements, 604–605
 - power-down modes, 606–608
- Power supply, 1073, 1074
- Power up, 269, 270
- Power-down modes, 606–608
- power-on reset, 1144
- power-up timer, 461–462
- PPICMICRO (specify PICMICRO processor), 1099
- PR2 register, 1046
- PRAGMA (C), 1129
- PREINC register, 362, 363
- PREINC0, 1050
- PREINC1, 1050
- PREINC2, 1049
- Prescaler, 482–483, 786–790, 814–815
- PRESET (MS-extension BASIC), 1096
- Primary register set, 282
- Princeton architecture, 8–9, 11, 22, 1223
- PRINT (BASIC), 1090
- PRINT (BASIC87x), 94
- PRINT (MS-extension BASIC), 1098
- Print concatenation, 1092
- PRINT USING (MS-extension BASIC), 1096
- Printable character, 1144
- Printed circuit board (PCB), 35
- Printed wiring assembly (PWA), 1223
- Printed wiring board (PWB), 1223
- Printed-circuit assembly (PCA), 1222
- Printed-circuit board (PCB), 1222
- Printers, 749–751, 1097
- Priority, task, 539
- Priority mode, 547
- Problem characterization, 567–569
- Processes, 537–538

- PROCESSOR (MPASM), 134
- Processor architectures, 7–12
 - CISC vs. RISC, 7–8
 - Harvard vs. Princeton, 8–9
 - microcoded vs. hardwired, 10–12
 - of PIC microcontrollers. *See* PIC MCU processor architecture
- Processor control instructions, 15, 294, 298
 - in low-end PIC MCU instruction set, 356
 - in mid-range PIC MCU instruction set, 345–348
 - in PIC18 MCU instruction set, 371–372
- Processor registers, 13, 249
- PRODH, 1051
- PRODL, 1051
- Product, 1110
- Production programmed, 171
- Program, 12
- Program counter (PC), 15, 247, 1223
 - incrementing, 170
 - in low-end PIC microcontrollers, 277–278, 433
 - in mid-range PIC microcontrollers, 264–268
 - in PIC18 microcontrollers, 288–292
 - and RTOS, 546
- Program counter (PCL) register, 302, 303
- Program memory, 13, 21–22, 247, 1223
 - checking, 170, 171
 - maximum, 376
 - programming, 170
 - reserve, 130
 - reserve block of, 134
- Program memory size, 49, 274
- Program memory type, 49
- Program store, 1223
- Programmable clock, 211
- Programmable read-only memory (PROM), 23, 1223
- Programmers, PIC microcontroller, 155–206
 - and code protection, 158–159
 - El Cheapo, 196–203
 - and hex file format, 156–158
- Programmers, PIC microcontroller (*Cont.*):
 - hobbyist, 181–182
 - and ICSP programmer interface, 166–177
 - microchip, 178–181
 - parallel, 159–166
 - third-party, 204–206
 - YAP-II, 182–196
- Programming, 1223
 - MPLAB ICD 2, 217
 - parallel, 159–166
 - simplifying, 487–488
- Programming method, 49
- Project management, 599–602
- Projects, 853–963
 - addendum IR robot, 898–903
 - addendum to electronic thermometer, 910–913
 - analog clock, 878–883
 - Christmas decoration with blinking light/music, 883–891
 - electronic thermometer with seven segment LED displays, 903–910
 - fuzzy logic fan governor, 953–963
 - for low-end devices, 853–877
 - MIC-II, 934–943
 - for mid-range devices, 878–953
 - model train light control with hall-effect sensor, 854–859
 - NTSC video output, 944–953
 - for PIC18 devices, 953–963
 - PICBASIC port of Marya's toy, 913–922
 - RS-485 master/slave remote thermometer interface, 922–931
 - serial LCD interface, 859–866
 - servo controller, 931–934
 - switch matrix key matrix, 872–877
 - TV IR remote-control robot, 891–898
 - ultrasonic LCD ranging, 866–872
- PROM (programmable read-only memory), 1223
- Protect mode, 1099
- Proton rest mass, 1061
- Prototype programmed, 171
- PSET (MS-extension BASIC), 1096
- Pseudocode, 1123
- Pseudorandom integer, 1146
- Pseudorandom variable, 1116, 1136
- PSP. *See* parallel slave port
- PSPIE flag, 1021
- PSPIF flag, 1021
- PSPMODE bit, 1021, 1023
- PTH (pin through hole), 1223
- Pull-down, 1223
- Pull-down transistors, 873
- Pull-up, 475, 476, 1223
- Pull-up resistors, 6, 1142–1143, 1145
- Pulse, 1116
- Pulse width, 685
- Pulse width modulation (PWM), 1223
 - motor speed control with, 717–721
 - operation of, 631–633
 - set output configuration bits for ECCP, 1147
 - TMR0 interrupt for dimming LED, 819–826
- Pulse-coded data format, 31–32
- Pulse-width modulated signal, 1116
- PULSIN (PICBASIC), 1116
- PULSOUT (PICBASIC), 1116
- Punctuation character, 1144
- push, 27, 72–75, 291, 383, 1223
- Push, 1004
- Push Macro, 96
- Pushbuttons, 1074
- PUT (MS-extension BASIC), 1098
- PUTCSPI (C18 C), 1148
- PUTCSWSPi (C18 C), 1148
- PUTCUSART (C18 C), 1148
- PUTC#USART (C18 C), 1148
- PWA (printed wiring assembly), 1223
- PWB (printed wiring board), 1223
- PWM. *See* pulse width modulation
- PWM (PICBASIC), 1116
- PWM channel, 1142, 1145
- PWM I/O, interfacing with, 685–690
- PWM signal, 685
- PWRTE, 1100

- Q cycles, 246
- Qualification plan, 592–595
- Question mark (?), 1090
- QuickSort routine, 417–419
- Quotient, 1134, 1135

- RA4 pin, 474, 475
- Radian angle, 1091
- Radiofrequency (RF), 603
- RadioShack, 903–905
- RADIX (MPASM), 134
- RAM (random access memory), 1223
- RANDOM (PICBASIC), 1116
- Random access memory (RAM), 1223
- Random function, 1147
- Random number, 1090, 1133, 1136
- Random number generator, 790–794
- Random.asm (application), 790–794
- RANDOMIZE (BASIC), 1090
- Range, 1156, 1158
- Raster, 945
- Ratio of circumference to diameter of circle, 1061
- Raw card, 35, 37, 39, 1223
- RC (resistor/capacitor), 1224
- RC charging formula, 1064
- RC discharging formula, 1064
- .rc files, 741
- RC oscillator, 6
- R/C servo control, 733–738
- RC time constant formula, 1064
- RCALL, 367, 368, 1004
- RCIF interrupt request bit, 1026
- RCIO mode, 459
- RCON register, 1046–1047
- RCREG register, 1025, 1026, 1039
- RCSTA register, 1026, 1028, 1039
- RCTIME (PICBASIC), 1116
- Read
 - result of capture event from input capture, 1146
- READ (BASIC), 1090
- READ (BASIC87x), 94
- READ (MS-extension BASIC), 1094, 1098
- READ (PICBASIC), 1116
- Read from DATA, 1094
- Read string, 1114

- Reading
 - of data, 675
- Read-only memory (ROM), 22, 23, 1224
- REAL ICE emulator, 213–214
- Real-time operating system (RTOS), 537–563, 1224
 - defined, 537
 - low-end/mid-range, 541–542
 - PIC18 design, 542–563
 - PIC18 vs. PC, 538–541
- Receive
 - asynchronous data, 1116–1118
 - data from file one character at a time, 1132
 - from USART, 1113
 - from X-10 devices, 1121
- Receiver, 1224
- Recursion, 1224
- Recycled code, 1149–1212
 - bitwise operations, 1158–1159
 - convert ASCII to hex nybble, 1152
 - convert ASCII to upper case, 1157
 - convert nybble to ASCII, 1151–1152
 - copy bits from register to register, 1151
 - count number of 1s in byte, 1157
 - detect change in register, 1155
 - divide by three, 1153–1154
 - generate parity for byte, 1158
 - if-less-than compare/swap, 1156–1157
 - increment/decrement w register, 1151
 - keep variable within range, 1158
 - macros. *See* macros
 - negate contents of file register, 1150
 - negate contents of w register, 1150
 - rotate byte in place, 1151
 - sixteen-bit counter with constant loop delay, 1154–1155
 - sixteen-bit numbers. *See* sixteen-bit numbers
 - sixteen-bit pulse measurement with five-cycle delay, 1155
- Recycled code (*Cont.*):
 - swap bit pairs, 1158
 - swap contents of two registers, 1156
 - swap contents of w with register, 1156
 - test byte within range, 1156
 - T0CK1 as interrupt source pin, 1152–1153
- Redefine variable, 1093
- REDIM (MS-extension BASIC), 1093
- Reduced instruction set computer (RISC), 7–8, 1224
- Reentrant subroutines, 423
- Reference signal, 1085
- Referencing instruction operation, 284
- Register(s), 1224
 - clearing, 252
 - declare, 93
 - flag range of unimplemented, 129
 - in low-end devices, 434, 435
 - as memory areas, 13
 - PIC MCU configuration, 466–469
 - types of, 13
- Register access, 275–276, 286–288
- Register addresses, 246–247. *See also* I/O register addresses
- Register arithmetic, 293
- Register definition files, 443
- Register map, 286
- Regression testing, 573
- Relative jumps, 378
- Relaxation oscillator, 57
- Relays, 692–693, 713
- Release of solution, 576–577
- Reliability, 594
- Relocatable, 1224
- REM (BASIC), 1091
- Remainder, 1093, 1110, 1134, 1135
- Remark, 1091
- Remote thermometer interface project, 922–931
- Remote-control robot project, 891–898
- Repeat execution, 93
- Requirements definition, 590–595
- RES (MPASM), 134

- Reserve block of program memory, 134
- Reserved words, 94, 1130
- Reset, 49, 1224
 - in application design, 608–610
 - internal, 609–610
 - in low-end PIC microcontrollers, 757–758
 - of PIC microcontrollers, 268–271, 460, 461
- Reset, 1004
- RESET (MS-extension BASIC), 1098
- reset circuit, 6
- Reset next data value, 1091
- Reset operation
 - of PIC microcontrollers, 49
- Reset vector, 269, 271
- Reset vector address, 274
- Resistance, 798–804
- Resistor color code chart, 1075
- Resistor ladder, 1224
- Resistor ladder output, 804–810
- Resistor ladder voltage output, 804–810
- Resistor/capacitor (RC), 1224
- Resistors, 455–456, 714–715, 1073–1075, 1224
- Resonance, 1064
- Resonators, 456–458
- Resource allocation, 595–597
- Resources, 965–975
 - books, 966–970
 - Microchip Technology, 965–966
 - part suppliers, 973–975
 - periodicals, 971–972
 - websites, 970–973
- Respin, 54
- Restart condition, 1146
- RESTORE (BASIC), 1091
- Restore execution, 1116
- RESUME (MS-extension BASIC), 1094
- RESUME (PICBASIC), 1116
- RETFIE, 332, 989, 1004
- RETLW, 338, 339, 984, 989, 1005
- return, 984, 989, 1005
- RETURN (BASIC), 1090
- RETURN (BASIC87x), 94
- RETURN (C), 1130
- RETURN (PIC microcontroller instruction), 330, 331, 355, 433–434
- RETURN (PICBASIC), 1116
- Return execution, 1090
- Return first index of array variable, 1093
- Return last index of array variable, 1093
- Return length of string, 1132
- Return position of first character, 1132
- Return position of last character, 1132
- Return statement, 330–332, 1127
- Return value in register address, 1115
- Return values, 507
- REV (PICBASIC), 1111
- REVERSE (PICBASIC), 1116
- Reverse bits, 1111
- Reverse polish notation (RPN), 72
- RIGHT\$ (MS-extension BASIC), 1095
- Ring indicator (RI), 697
- Ring networks, 34
- RISC (reduced instruction set computer), 1224
- RJ-45 connector, 216, 217
- RL time constant, 1064
- RLCF, 1005
- RLCNF, 1005
- RLF, 326–328, 984, 989
- RND (BASIC), 1090
- Robot project(s)
 - addendum IR, 898–903
 - TV IR remote-control, 891–898
- ROM (read-only memory), 1224
- Rotate, 326–328, 1224
 - of byte in place macro, 1151
 - of sixteen-bit values, 1205–1206
- Rotating instructions, 1150
- Rounding, 400
- Row, 1096
- RPx bit settings, 129, 253
- RRCF, 1006
- RRCNF, 1006
- RRF, 327, 984–985, 989
- RS-232, 30, 188, 206, 695–702, 841–851, 954, 1224
- RS-232 serial ports, 739, 742–744
- RS-422, 703–704
- RS-485, 702–704, 1224
- RS-485 master/slave remote thermometer interface project, 922–931
- RTLW (PIC microcontroller instruction), 330–332
- RTOS (real-time operating system), 1224
- RTOS requests, 549
- RTRIM\$ (MS-extension BASIC), 1095
- s (do not assemble compiled code), 1099
- Same logical state, 1111
- Scan, 1224
- Scan code, 671
- Scheduling, 600–601
- Schmidt trigger inputs, 455, 475, 667–668
- SCREEN (MS-extension BASIC), 1096
- Screen mode, 1096
- Search, 1134, 1136–1137
- Search list, 1114
- Second-pass assembly, 68–70
- Seconds, 1135, 1138
- Seconds since midnight, 1097
- SEEK (MS-extension BASIC), 1098
- SELECT (MS-extension BASIC), 1095
- Semaphore, 541
- Semicolon, 295, 1101, 1107
- Semiconductors, 1076, 1077
- Send asynchronous data, 1118–1119
- Send string, 1114
- Send to LCD, 1114
- Send X-10 data, 1121
- SendCharMacro, 235, 236
- SendMsg subroutine, 234–235
- Separator, 1092
- SER2_BITS (PICBASIC), 1106
- Serial, 1224
- Serial data transmission, 19
- Serial I/O, 633–649
 - master SSP and I2C operation, 638–643
 - SSP communications module, 633–638

- Serial I/O (*Cont.*):
 - USART asynchronous serial communications, 643–649
- Serial I/O application code, 832–851
 - interfacing PIC MCU with three-wire RS, 841–851
 - simulated asynchronous serial I/O hardware with pin stimulus file, 832–841
- Serial LCD interface (SLI) project, 859–866
- Serial number, 162
- Serial port interface, 739
- Serial ports, 742–749
 - HyperTerminal, 745–746
 - Visual Basic MSComm serial control, 746–749
- Series capacitance formula, 1063
- Series resistance formula, 1063
- SERIN (PICBASIC), 1116–1117
- SERIN2 (PICBASIC), 1117–1118
- SEROUT (PICBASIC), 1118–1119
- SEROUT2 (PICBASIC), 1119
- Server farms, 34
- Servo, 1225
- Servo control, R/C, 733–738
- Servo-controller project, 931–934
- SET, 256
- SET (MPASM), 134
- SETC, 1015
- SETDC, 1015
- SETF, 1006
- SetPointer macro, 410–411
- SETZ, 1015
- Seven-segment LED displays, 663–665
- SFRs. *See* special function registers
- SGN (BASIC), 1091
- Shaded addresses, 254
- Shadowed, 249, 253
- Shared memory, 249
- Sharp, 1225
- Shift, 1110, 1225
- Shift data synchronously, 1120
- Shift left, 496, 1127, 1128, 1205–1206
- Shift register, 680–681
- Shift right, 496, 1127, 1128
- SHIFTLN (PICBASIC), 1120
- SHIFTOUT (PICBASIC), 1120
- Short.asm application, 794–798
- Sign, 1091
- Signed byte, 1139
- SIGNED CHAR ATOB (C18 C), 1139
- Signed integer, 1139, 1145
- SimmStick, 949
- Simple delay (macro), 1160–1163
- Simulated asynchronous serial I/O hardware with pin stimulus file, 832–841
- Simulated open-collector/open-drain I/O, 615–616
- Simulating applications, 141–154
 - assembler flashing LED, 152–154
 - and debugging, 565–566, 573–576
 - PICC-Lite flashing LED, 151–152
 - simple application, 149–151
 - stimulus, 146–149
 - Watch window files, 145–146
- Simulating input pin interrupt with timer pin input, 815–819
- Simulating logic, 423–426
- Simulating microprocessor bus, 4
- Simulator, 1225
- Simulator program, 40
- Simulators, 77–79
- SIN (BASIC), 1091
- SIN (PICBASIC), 1111
- SINE, 1091, 1111, 1133, 1136
- Single data line communications, 30
- Single media networks, 32–33
- Single precision floating-point numbers, 1090, 1093
- Single quote ('), 1101, 1130
- Single shot, 1083
- Single-dimensional arrays, 1124
- Single-stepping, 144
- Single-throw switches (SPST), 1074, 1077
- Sink current, 18
- 16-bit access, 465–466
- 16-bit operations, 390–392
- Sixteen-bit counter with constant loop delay (macro), 1154–1155
- Sixteen-bit numbers, 91, 1200–1212
 - addition/subtraction of constants, 1202–1204
 - addition/subtraction of variables, 1204–1205
 - comparisons with variables, 1206–1207
 - constants operations, 1202–1206
 - defining, 1201
 - division, 1210–1212
 - increments/decrements, 1202
 - multiplication, 1207–1210
 - variables operations, 1204–1206
- Sixteen-bit pulse measurement with five-cycle delay (macro), 1155
- 64-pin DIP PIC microcontroller package, 1058
- SIZE_T STRCSN (PICC), 1137
- SIZE_T STRLEN (PICC), 1137
- SIZE_T STRSN (PICC), 1137
- SKIP (PIC MCU instruction), 302, 303, 758, 759
- Skip on bit condition, 332–336
- Skips, 14
- SKPC, 1015
- SKPDC, 1015
- SKPNC, 1015
- SKPND, 1015
- SKPNZ, 1015
- SKPZ, 1015
- Slave, 29, 635, 636, 638, 1225
- Sleep, 270, 449, 450, 606, 1115, 1120
- SLEEP (PICBASIC), 1120
- Sleep instruction, 15, 345–346, 985, 989, 1007
- Sleep wakeup, 1144
- SLI project. *See* serial LCD interface project
- Slide switches, 1074
- SMD (surface-mounted devices), 1225
- SMT. *See* surface-mount technology
- Snubber resistor, 716
- Software, 1225
- Software development tools, 63–154
 - assemblers, 68–70
 - compilers, 70–76
 - editors, 65–68

- Software development tools
 - (*Cont.*):
 - emulators, 79–80
 - free, 740–742
 - hardware debuggers, 80–81
 - high level languages. *See* high level language
 - integrated development environments, 81–83
 - linkers, 76–77
 - for MPLAB IDE. *See* MPLAB IDE application build tools
 - for PC applications, 740–742
 - simulating applications with. *See* simulating applications
 - simulators, 77–79
 - version support, 101–103
- Software interrupts, 15
- Software PWM, 717–721
- Solder paste, 38
- Solenoids, 692–693
- Solution naïve, 569
- Solution release, 576–577
- Sony, 893
- Sort, 1136
- Sorting data, 414–419
- Sound
 - input/output of, 2
 - speed of, 1062
- SOUND (MS-extension BASIC), 1097
- SOUND (PICBASIC), 1120
- Source code, 39, 116–117, 519–522, 1225
- Source code listing, 133
- Source current, 18
- Source file (.\$\$\$), 113
- Source files, 1092
- Sp, 205
- SPACE (MPASM), 134
- SPACE\$ (MS-extension BASIC), 1095
- Spaces, 1096
- SPBRG register, 1025, 1039
- SPC (MS-extension BASIC), 1096
- Special characters, 1133
- Special function registers (SFRs), 249, 282, 295
 - in PIC18 microcontrollers, 286, 287, 356
 - as stimulus, 147, 149
- Specify file data, 1094
- Speed
 - control of motor speed, 717–724
 - of data transmission, 698
 - of light in vacuum, 1061
 - maximum, 3–4
 - of operating system, 205–206
 - of processor, 445
 - of sound at sea level in dry air at 20°C, 1062
 - of sound in water at 20°C, 1062
- SPI. *See* synchronous serial communications protocol
- SPI bus, 1143, 1148
- SPI data, 1142
- SPI interface chip select pin, 1142, 1147
- SPI protocol, 705, 706
- Splat (*), 1225
- SQR (BASIC), 1091
- SQR (PICBASIC), 1111
- Square brackets ([]), 1109
- Square root, 1091, 1111, 1133, 1136
- SRAM. *See* static random-access memory
- SSP buffer, 1142
- SSP modules, 1024, 1025, 1027
 - configure, 1145
 - configure for Microwire operation, 1145
 - disable, 1142
 - disable modules for Microwire communications, 1142
 - disable modules for SPI communications, 1142
 - initialize for SPI communications, 1145
- SSPADD buffers, 1025
- SSPADD register, 1045
- SSPBUF register, 1024, 1025, 1045
- SSPCON register, 1025, 1026
- SSPCON1 register, 1025, 1027, 1044
- SSPCON2 register, 1025, 1028, 1043–1044
- SSPIE bit, 1025
- SSPIF bit, 1025
- SSPSTAT register, 1025, 1027, 1028, 1044–1045
- Stack memory, 27
- Stack operations, 71–76
- Stack pointer, 1225
- Stacks, 1225
 - of PIC microcontrollers, 264–268
 - in PIC18 microcontrollers, 288–292
 - push character back onto input stack for later processing, 1138
- Standby, 606
- Star networks, 29, 33
- Start bit, 30, 694
- Starting address, 133
- State analyzer, 1225
- State analyzers, 1086–1087
- State machine, 1225
- State machine programming, 429–430
- STATEMC application, 768–770
- Static random-access memory (SRAM), 25–26, 535, 1225
- Status byte, 547
- Status condition, 14
- Status flags, 338, 387
- STATUS register, 249–253, 756–758, 1017–1020, 1049
 - instructions not affecting, 421
 - PA bits in, 278
 - of PIC microcontrollers, 257–259
 - reset event bit values in, 270
 - update IRP bit of, 129
 - update RPx bit settings of, 129
- Stefan-Boltzmann constant, 1061
- STEP (BASIC87x), 93
- STEP (PICBASIC), 1113
- Stepper motors, 724–733
 - bipolar motor control, 730–733
 - unipolar motor control, 725–729
- Stimulus (for simulating applications), 146–149
- Stimulus files, 63, 78–79
- STKPTR register, 290–291, 1052
- Stop, 1115, 1120
- STOP (BASIC), 1091
- STOP (PICBASIC), 1120
- Stop bit, 30, 31, 694–695
- Stop execution, 93
- Stop processing, 1113
- Store constants, 1112

- Store new values, 1113
- STR\$ (MS-extension BASIC), 1095
- Straight-through cable, 751
- STRCAT (C), 1132
- STRCPY (C), 1132
- String(s), 1104, 1226
 - append, 1132
 - break up incoming string into data segments and store in variables, 1136
 - break up string into smaller, 1137
 - case-insensitive compare, 1137
 - case-insensitive return pointer to string 2 found in string 1, 1137
 - compare, 1114, 1115, 1137
 - concatenate, 1136, 1137
 - convert ASCII code to character, 1095
 - convert integer number to, 1095
 - convert lower case to upper case, 1095
 - convert signed byte to, 1139
 - convert signed integer to, 1145
 - convert time to, 1134
 - convert to floating-point number, 1134, 1139
 - convert to integer, 1134
 - convert to integer ASCII code, 1095
 - convert to long integer, 1134, 1139
 - convert to signed byte, 1139
 - convert to signed integer, 1139
 - convert upper case to lower case, 1095
 - copy, 1132, 1136, 1137
 - declaring, 1124
 - define variable as, 1093
 - directive for different cases, 1129
 - get specified number of characters, 1095
 - input, 1132
 - in MPASM, 134
 - output, 1132
 - receive, 1114
 - remove leading blanks, 1095
 - remove trailing blanks, 1095
- String(s) (*Cont.*):
 - return blanks, 1095
 - return leftmost characters, 1095
 - return length of, 1095, 1132
 - return length of segment with characters, 1137
 - return number of characters in, 1137
 - return number of pending characters from keyboard buffer, 1095
 - return pointer to first occurrence of character, 1137
 - return pointer to string 2 found in string 1, 1137
 - return position of first character, 1132
 - return position of last character, 1132
 - return position of substring, 1095
 - return rightmost characters, 1095
 - return/overwrite characters at position, 1095
 - search, 1136, 1137
 - send, 1114
 - wait for characters, 1119
 - write, 1146
- String data, 1090
- String variable, 1092
- STRLEN (C), 1132
- STRNCAT (C), 1132
- STRNCPY (C), 1132
- STRUCT TM * GMTIME (PICC), 1135
- STRUCT TM * LOCALTIME (PICC), 1135
- Structre.inc file, 513
- Structured programming macros, 513–518
- Structured programming statements, 137
- Structures, assembly-language, 407–409
- SUB (MS-extension BASIC), 1094
- SUBCF, 1015
- SUBFWB, 364–365
- SUBLW, 321–322, 989, 1008–1009
- Subroutines, 268, 1226
 - call, 1090, 1113
 - and data stack, 73, 74
 - define, 1093, 1094
 - in low-end devices, 432–433
 - macros vs., 489, 490
 - with parameter passing, 381–385
 - reentrant, 423
- Substring position, 1095
- SUBTITLE (MPASM), 134
- Subtraction, 352, 353, 496, 1092, 1110
 - in assembly language, 385–389
 - from assignment variable, 1128
 - in C programming language, 1127, 1128
 - and data stack, 76
 - in mid-range PIC microcontrollers, 318–322
 - in PIC18 microcontrollers, 364–366
 - problems with, 256
 - with sixteen-bit values, 1202–1205
 - true, 285
- SUBWF, 318–322, 985, 990, 1009
- SUBWFB, 1007–1008, 1010
- Sum, 1110
- Supercomputers, 34
- Surface-mount technology (SMT), 37–38, 52, 1031, 1225
- Surface-mounted devices (SMD), 1225
- Suspicious activity, 567–568
- SWAP (PICBASIC), 1120
- Swap macro(s)
 - for bit pairs, 1158
 - for contents of two registers, 1156
 - for contents of w with register, 1156
 - if-less-than compare and swap, 1156–1157
- SWAPF, 310–311, 343–344, 985, 990, 1011
- SWGETC12C (C18 C), 1147
- Switch, 1126
- SWITCH (C), 1130
- Switch bounce, 665–668
- Switch matrix, 668

- Switch matrix key matrix
 - project, 872–877
- SWPUTCI2C (C18 C), 1147
- Symbol (PICBASIC), 1102, 1103
- Symbol table, generate, 1099
- Synchronous serial, 29, 30, 1226
- Synchronous serial bus, 441
- Synchronous serial
 - communications, 1025
- Synchronous serial
 - communications protocol (SPI), 1225
 - configure I/O pins for, 1145
 - initialize SSP module for, 1145
 - read byte from, 1146
 - write byte to, 1148
 - write string to, 1146
- Synchronous serial interfaces, 704–710
- Synchronous serial port (SSP)
 - communications, 633–638
- Synchronous stimulus, 146–148
- SYSTEM (C), 1132
- System clock, 454–460
- System command, 1132

- T0CON register, 1048
- T1CON register, 625, 1022, 1023, 1046
- T1SYNCH, 626
- T2CON register, 627, 1022, 1024, 1045–1046
- T3CON register, 1040
- TAB (BASIC), 1091
- Tab columns, printer, 1091
- TAB *Electronics Microcontroller Handbook* (Predko), 553, 563
- TABLAT, 1051
- Table jumps, 278
- Table latch registers, 165
- Table pointer registers, 464
- Table read, 360–361, 370
- Table write, 361
- Table write instruction sequence, 173–176
- Tables, 265
 - in low-end PIC microcontrollers, 355–356, 433
 - in mid-range PIC microcontrollers, 338–341
 - in PIC18 microcontrollers, 369–370
- TABLRD, 1011
- TABLWT, 1011
- Tabs, 1092
- Tamiya, 897
- TAN (BASIC), 1091
- Tangent, 1091, 1137
- Tantalum capacitors, 446, 447, 603
- Task byte, 547
- Task context registers, 546
- Task information block (TIB), 545–547
- Tasks, 538, 1226
 - and hardware devices, 540–541
 - priority of, 539
- TBLPTRH, 1051
- TBLPTRL, 1051
- TBLPTRU, 1051
- TBLRD, 360–361
- TBLWT, 361
- Teams, project, 601–602
- Teletype, 693, 694
- Template
 - for MPLAB IDE simulator, 295
 - for PIC18 instructions, 356–357
 - sample assembly-language, 374–376
- Template.asm, 374
- Temporary variables, 73
- Terminal emulators, 699
- Test byte within range (macro), 1156
- Test equipment, 1080–1087
 - digital multimeters, 1081
 - logic probes, 1081–1083
 - logic/state analyzers, 1086–1087
 - oscilloscopes, 1084–1085
- Testing
 - application, 594
 - of code, 444
 - of corrective code, 572–576
 - of debugging hypothesis, 569–570
- Thermistor, 903–905
- Thermo project, 903–910
- Thermometer project(s)
 - addendum to, 910–913
 - electronic thermometer with seven segment LED displays, 903–910
- Thermometer project(s) (*Cont.*):
 - RS-485 master/slave remote thermometer interface, 922–931
 - RTOS application for digital, 553, 555–563
- Third-party programmers, 204–206
- Thread, 537, 538
- 3RS.asm application, 841–851
- 370 assembly language, 11
- Three-wire RS, 841–851
- Threshold voltage, 611
- Time
 - for capacitor charge, 799
 - convert into tm structure, 1135
 - convert number of seconds, 1135
 - convert to string, 1134
 - return number of seconds, 1138
- TIME\$ (MS-extension BASIC), 1097
- Timed operations, 1108
- TIMEEND (application), 779–781
- Timer(s), 1226
 - configure/enable, 1145
 - disable, 1142
 - of PIC microcontrollers, 50
 - power-up, 461–462
 - read value of, 1146
 - watchdog. *See* Watchdog timer
 - write value to, 1148
- TIMER (MS-extension BASIC), 1097
- Timer event handler, 851
- Timer pin input, 815–819
- Timer/interrupt handler with context saving, 810–815
- Time-sharing operating system, 537
- TIME_T TIME (PICC), 1138
- Timing, optimizing, 442
- TITLE (MPASM), 134
- Title line, 375
- TMR0, 478–483
 - delays, 480–482
 - watchdog timer prescaler, 482–483
- TMR0 delays, 480–482, 779–781
- TMR0 interrupt for LED PWM dimming, 819–826

- TMR0 register, 249
- TMR0 setup with prescaler, 786–790
- TMR0H, 1049
- TMR0Int.asm application, 815–819
- TMR0L, 1048
- TMR1, 624–626, 628
- TMR1 register, 1022
- TMR1H, 625, 1046
- TMR1L, 625, 1046
- TMR2, 624, 626–628
- TMR2 register, 1022, 1046
- TMR3H, 1040
- TMR3L, 1040
- TOCK1 as interrupt source pin (macro), 1152–1153
- TOGGLE (PICBASIC), 1120
- Tools. *See* MPLAB IDE
 - application build tools;
 - Software development tools
- Topology, 32
- TOSL, 1052
- TOSU, 1052
- Touch tone sequence, 1113
- Touch tone telephone
 - frequencies, 1063
- Traces, 1226
- Traffic light controller, 766–768
- Trailing blanks, 1095
- Train light control with
 - hall-effect sensor project, 854–859
- TrainCtl project, 854–859
- Transformer current/voltage
 - formula, 1064
- Transistors, 714, 1226
- Transmission-line characteristic
 - impedance formula, 1065
- Transmit from USART, 1113
- TRIACs, 857
- TRIS, 309, 310, 350–351, 985, 990, 1016
- TRIS register, 251, 275, 278–279, 350–351, 473–476, 1016
- TRISA register, 1036
- TRISB register, 252, 1036
- TRISC register, 1036
- TRISD register, 1036
- TRISE register, 1021, 1036–1037
- Tristatable output, 18
- Trows, 1074, 1076
- True subtract with borrow, 285
- TSTFSZ, 1011
- Turns ratio formula, 1064
- TV IR remote-control robot
 - project, 891–898
- 28-pin PIC microcontrollers, 1055, 1057
- 28-pin PIC18 PTH
 - microcontrollers, 1058
- Twisted-pair wiring, 703
- Two resistors in parallel
 - formula, 1063
- 293D quad half H-bridge, 715–716
- Two-pass assemblers, 40
- Two's complement, 319, 323–324, 1201, 1226
- Two-wire LCD control, 680–682
- TXIE, 1026
- TXIF, 1026
- TXREG register, 1025, 1026, 1039
- TXSTA register, 1026, 1029, 1039
- UART (universal asynchronous receiver/transmitter), 1226
- UBOUND (MS-extension BASIC), 1093
- UCASE\$ (MS-extension BASIC), 1095
- UDATA (MPASM), 134
- UDATA_ACS (MPASM), 134
- UDATA_OVR (MPASM), 134
- UDATA_SHR (MPASM), 134
- ULN2003A, 693
- Ultrasonic LCD ranging project, 866–872
- Ultraviolet (UV) light, 24, 25, 1226
- UNDEFINE (C), 1129
- UNDEFINE (MPASM), 134
- Unipolar stepper motor control, 725–729
- Universal assembly language, 1123
- Universal asynchronous receiver/transmitter (UART), 1226
- Universal Device Programmer, MPLAB PM3, 180–181
- Universal hobbyist programmer, 181–182
- Universal serial bus (USB), 34, 35, 739
- Universal synchronous receiver/transmitter. *See* USART
- UNSIGNED CHAR (C18 C), 1139
- UNSIGNED CHAR
 - CAN2510BYTEREAD (C18 C), 1139
- UNSIGNED CHAR
 - CAN2510DATAREAD (C18 C), 1139
- UNSIGNED CHAR
 - CAN2510DATAREADY (C18 C), 1139
- UNSIGNED CHAR
 - CAN2510ERRORSTATE (C18 C), 1140
- UNSIGNED CHAR
 - CAN2510INIT (C18 C), 1140
- UNSIGNED CHAR
 - CAN2510INTERRUPTSTATUS (C18 C), 1140
- UNSIGNED CHAR
 - CAN2510READMODE (C18 C), 1140
- UNSIGNED CHAR
 - CAN2510READSTATUS (C18 C), 1140
- UNSIGNED CHAR
 - CAN2510SETMSGFILTERSTD (C18 C), 1141
- UNSIGNED CHAR
 - CAN2510SETMSGFILTERXTD (C18 C), 1141
- UNSIGNED CHAR
 - CAN2510SETSINGLEMASKSTD (C18 C), 1141
- UNSIGNED CHAR
 - CAN2510SETSINGLEMASKXTD (C18 C), 1141
- UNSIGNED CHAR
 - CAN2510WRITEBUFFER (C18 C), 1141
- UNSIGNED CHAR
 - CAN2510WRITESTD (C18 C), 1141
- UNSIGNED CHAR
 - CAN2510WRITEXTD (C18 C), 1142
- UNSIGNED CHAR DATARDYI2C (C18 C), 1142

- UNSIGNED CHAR
 - DATARDYMWIRE (C18 C), 1142
- UNSIGNED CHAR DATARDYSPI (C18 C), 1142
- UNSIGNED CHAR
 - EEACKPOLLING (C18 C), 1143
- UNSIGNED CHAR EEBYTEWRITE (C18 C), 1143
- UNSIGNED CHAR EEP-AGEWRITE (C18 C), 1143
- UNSIGNED CHAR
 - EESEQUENTIALREAD (C18 C), 1143
- UNSIGNED CHAR GETSI2C (C18 C), 1143
- UNSIGNED CHAR ISCNTL (C18 C), 1144
- UNSIGNED CHAR ISGRAPH (C18 C), 1144
- UNSIGNED CHAR ISLOWER (C18 C), 1144
- UNSIGNED CHAR ISPRINT (C18 C), 1144
- UNSIGNED CHAR ISPUNCT (C18 C), 1144
- UNSIGNED CHAR ISSPACE (C18 C), 1144
- UNSIGNED CHAR ISUPPER (C18 C), 1144
- UNSIGNED CHAR ISXDIGIT (C18 C), 1144
- UNSIGNED CHAR WRITEI2C (C18 C), 1148
- UNSIGNED CHAR WRITEMWIRE (C18 C), 1148
- UNSIGNED CHAR WRITESPI (C18 C), 1148
- UNSIGNED INT
 - EECURRENTADDRESS (C18 C), 1143
- UNSIGNED INT EERANDOMREAD (C18 C), 1143
- UNTILC macro, 514
- UNTILV macro, 514
- UPPER (MPASM), 124
- UPPER (upper 8 bits), 390–391
- Upper case
 - convert ASCII to upper case macro, 1157
 - convert from, 1147
 - convert to, 1095, 1133, 1138, 1147
 - return nonzero if, 1144
- USART (universal synchronous receiver/transmitter), 1025, 1026, 1226
 - configure I/O pins for, 1145
 - configure/enable, 1145
 - disable, 1142
 - read byte from, 1146
 - read string from, 1144
 - receive bytes from, 1113
 - return nonzero if character received, 1142
 - return nonzero if transmitting, 1139
 - transmit from, 1113
 - write byte to, 1148
 - write string to, 1146
- USART asynchronous serial communications, 643–649
- USB. *See* universal serial bus
- User interface, 6, 595
 - in application design, 597–599
 - in BASIC87x, 95
 - in simulators, 79
- UV light. *See* ultraviolet light
- v (turn on verbose mode), 1099
- Van Graf, Rudolf, 590
- VAR (MS-extension BASIC), 1095
- VAR (PICBASIC), 1101–1103
- Variable(s), 1226
 - in assembly language, 379–381, 505–506
 - in BASIC, 89–90, 1089, 1090
 - comparisons with sixteen-bit, 1206–1207
 - data-type naming conventions, 1089, 1090
 - declare, 93
 - declaring, 129, 135
 - get from file, 1098
- Variable(s) (*Cont*):
 - global, 86–88, 382, 523
 - initializing, 443
 - local, 86–88, 133
 - AND operation with sixteen-bit, 1205
 - parameter, 382
 - PICBASIC, 1101–1103
 - 16-bit, 390
 - type of, 1125
- VARIABLE (MPASM), 135
- Variable declaration, 1090
- Variable display command, 92
- Variable memory, 249
- Variable resistance voltage divider, 804–810
- Variable resistors, 1074, 1075
- Vcc, 1226
- Vdd, 167–168, 171, 1026, 1030, 1226
- Verbose mode, turn on, 1099
- Version number, 375, 493–494
- Version support, 101–103
- Vertical blanking, 946, 947, 950
- Vertical synch, 945, 1226
- Vertical tab (VT), 1130
- Vias, 1226
- Video field, 946
- Video output project, 944–953
- VIEW (MS-extension BASIC), 1096
- VIEW SCREEN (MS-extension BASIC), 1096
- Viewport, 1096
- Virtual memory, 543
- Visual Basic 6.0, 740, 746
- Visual Basic MSComm serial control, 746–749
- Visual Studio.Net, 740
- VLadder.asm application, 804–810
- VOID * BSEARCH (PICC), 1134
- VOID ACKI2C (C18 C), 1139
- VOID BAUD (C18 C), 1139
- VOID CAN2510BITMODIFY (C18 C), 1139
- VOID CAN2510BYTEREAD (C18 C), 1139
- VOID CAN2510BYTEWRITE (C18 C), 1139
- VOID CAN2510DISABLE (C18 C), 1140
- VOID CAN2510ENABLE (C18 C), 1140
- VOID CAN2510INTERRUPTEN-ABLE (C18 C), 1140
- VOID CAN2510LOADBUFFERSTD (C18 C), 1140
- VOID CAN2510LOADBUFFERXTD (C18 C), 1140
- VOID CAN2510SEQUENTIAL-WRITE (C18 C), 1140
- VOID CAN2510SETBUFFER-PRIORITY (C18 C), 1141
- VOID CAN2510SETMODE (C18 C), 1141

- VOID CAN2510SETSINGLE-
FILTERSTD (C18 C), 1141
- VOID CAN2510SETSINGLE-
FILTERXTD (C18 C), 1141
- VOID CLEARCSSWSPI (C18 C),
1142
- VOID CLOCK_TEST (C18 C),
1142
- VOID CLOSEADC (C18 C), 1142
- VOID CLOSECAPTURE (C18 C),
1142
- VOID CLOSEECAPTURE (C18
C), 1142
- VOID CLOSEPWM1 (C18 C),
1142
- VOID CLOSEI2C (C18 C), 1142
- VOID CLOSMWIRE (C18 C),
1142
- VOID CLOSEPORTB (C18 C),
1142
- VOID CLOSEPWM (C18 C), 1142
- VOID CLOSERB (C18 C), 1142
- VOID CLOSESPI (C18 C), 1142
- VOID CLOSETIMER (C18 C),
1142
- VOID CONVERTADC (C18 C),
1142
- VOID CPUTS (PICC), 1134
- VOID DELAY1KTCYX (C18 C),
1143
- VOID DELAY10KTCYX (C18 C),
1143
- VOID DELAY10TCYX (C18 C),
1142
- VOID DELAY100TCYX (C18 C),
1143
- VOID DISABLEPULLUPS
(C18 C), 1143
- VOID ENABLEPULLUPS (C18
C), 1143
- VOID GETSMWIRE (C18 C),
1143
- VOID GETSSPI (C18 C), 1143
- VOID GETSUART (C18 C), 1144
- VOID IDLEI2C (C18 C), 1144
- VOID LONGJMP (PICC), 1135
- VOID NOTACKI2C (C18 C),
1145
- VOID OPEN USART (C18 C),
1145
- VOID OPENADC (C18 C), 1145
- VOID OPENCAPTURE# (C18 C),
1145
- VOID OPENECAPTURE# (C18
C), 1145
- VOID OPENEPWM1 (C18 C),
1145
- VOID OPENI2C (C18 C), 1145
- VOID OPENMWIRE (C18 C),
1145
- VOID OPENPORTB (C18 C),
1145
- VOID OPENPORTB#INT (C18
C), 1145
- VOID OPENPWM# (C18 C), 1145
- VOID OPENSPI (C18 C), 1145
- VOID OPENSWSPI (C18 C),
1145
- VOID OPENTIMER# (C18 C),
1145
- VOID OPENUART (C18 C), 1145
- VOID PERSIST_VALIDATE
(PICC), 1135
- VOID PUTCH (PICC), 1136
- VOID QSORT (PICC), 1136
- VOID SETCGRAMADDR (C18 C),
1147
- VOID SETCHANADC (C18 C),
1147
- VOID SETCSSWSPI (C18 C),
1147
- VOID SETDDRAMADDR (C18 C),
1147
- VOID SETOUTPUTPWM1
(C18 C), 1147
- VOID SETOUTPUTPWM# (C18
C), 1147
- VOID SRAND (C18 C), 1147
- VOID SRAND (PICC), 1136
- VOID STARTI2C (C18 C), 1147
- VOID STOPI2C (C18 C), 1147
- VOID SWRESTARTI2C (C18 C),
1147
- VOID SWSTARTI2C (C18 C),
1147
- VOID SWSTOPI2C (C18 C),
1147
- VOID UNGETCH (PICC), 1138
- VOID WRITECMDXLCD (C18 C),
1148
- VOID WRITEDATAXLCD (C18
C), 1148
- VOID WRITETIMER# (C18 C),
1148
- VOID WRITEUART (C18 C),
1148
- VOID WRITE#USART (C18 C),
1148
- Volatile, 1227
- Volatile memory, 26
- Volatile variable memory, 22
- Voltage, 171, 172, 1073, 1227
- negative, 700
- operating, 49
- output, 804
- of PIC microcontrollers, 49
- and selection of PIC MCUs,
486–487
- threshold, 611
- Voltage comparators, 650–653
- Voltage levels, 6
- Voltage margins, 171
- Voltage reference output (Vref),
1026, 1030
- Voltage regulators, 449, 450, 1227
- Voltage supply, 167, 168
- Volts, 1227
- Von Neumann, John, 8, 1227
- Von Neumann architecture, 8,
9, 60
- Vpp, 167, 168, 171
- VRCON register, 1026, 1029,
1030
- Vss (voltage), 1227
- VT (vertical tab), 1130
- W (working) register, 247, 248,
421
- WAIT (PICBASIC), 1119
- Wait states, 1227
- WAITSTR (PICBASIC), 1119
- Wakeup, 1144
- Warnings, 70, 117
- Warren, Andy, 1153
- Watch window, 145–146,
295–297
- Watchdog timer (WDT), 270,
462–463, 781–786, 1100,
1227
- clearing of, 345
- default configuration fuse
register setting for, 1100
- prescaler, 482–483
- providing support for, 463
- Watchdog timer timeout, 1144
- Watchdog timer wakeup, 1144
- Wattage, 1227
- WDT. *See* watchdog timer
- WDT.asm application, 782
- WDTCON register, 1047
- Websites, 970–973
- Microchip Technology,
965–966
- for PIC microcontroller,
970–971

- Websites (*Cont.*):
 - for projects/information, 972–973
- WEND (PICBASIC), 1121
- Wheatstone bridge formula, 1064
- WHILE (C), 1130
- WHILE (MPASM), 135, 501, 506, 512
- WHILE (PICBASIC), 1121
- While statement, 1126
- WHILEC macro, 514
- WHILEV macro, 514
- White space character, 1144
- Whole number, 1134, 1135
- WiFi, 32
- WINDOW (MS-extension BASIC), 1096
- WINDOW SCREEN (MS-extension BASIC), 1096
- windowed ceramic packaging, 36, 51
- windows, 295–297
- Wirz Electronics, 183
- Word, 1103, 1124, 1227
- Word size, 14, 1124
- WordPad, 65–66
- Work environment, 602
- Working register. *See* W register
- Wozniak, Steve, 88
- WREG register, 249, 284, 293, 300, 301, 1050
- WRITE (MS-extension BASIC), 1098
- WRITE (PICBASIC), 1121
- Write data value, 1121
- Write value to register address, 1115
- XIN (PICBASIC), 1121
- Xmas project, 883–891
- XON/XOFF protocol, 698
- XOR, 1227
 - bitwise, 496
 - in C programming language, 1127, 1128
 - in mid-range PIC microcontrollers, 326
 - in MS-extension BASIC, 1093
 - in PICBASIC, 1110, 1111
- XOR (MS-extension BASIC), 1093
- XOR (PICBASIC), 1111
- XORLW, 326, 985, 990, 1012
- XORWF, 326, 985, 990, 1012
- XOUT (PICBASIC), 1121
- YAP-II programmer, 182–196
 - 19-pin interface for, 189, 190
 - accessory connector pin definition for, 191
 - bill of materials for, 186
 - bottom layer design of, 187
 - commands for, 191–193
 - enhancements of, 183
 - ICSP connector pin definition for, 189, 190
 - photograph of, 182
 - purpose of, 182
 - schematic of, 184, 185
 - silkscreen overlay of, 188
 - specifications for, 183
 - top layer design of, 187
 - Visual Basic interface with, 196
- zero flag (Z), 257–258, 325, 326, 386, 387
- zero insertion force (ZIF), 179, 1227
- zero registers, 253–254
- ZIF. *See* zero insertion force
- ZigBee, 32
- .zip files, 1227

This page intentionally left blank

Discount Coupon - Limited Time Offer

20% Off Retail Price



Save 20% on DV164007 - MPLAB® ICD 2 MODULE
(Kit contains the MPLAB ICD 2 module, ICD interface cable,
USB cable, RS-232 serial cable and a power supply)

Microchip Technology is offering a 20% discount off the retail price of the MPLAB ICD 2 In-Circuit Debugger (DV164007). The MPLAB ICD 2 module is a low-cost development tool that connects between a PC and the designers target board, allowing direct in-circuit debugging of a target PIC® microcontroller or dsPIC® digital signal controller. Programs can be executed in real time or single step, watch variables established, break points set, memory read/writes accomplished and more. The MPLAB ICD 2 module can also be used as a development programmer.

This discount can only be redeemed at www.microchipdirect.com and this offer is not valid after December 31, 2008.

Terms and Conditions

This offer applies only to the MPLAB ICD 2 In-Circuit Debugger. Orders must be placed at www.microchipdirect.com. Select the MPLAB ICD 2 development tool, add item to cart and proceed to the checkout process. Enter the coupon code below and select the green button "apply coupon". The discount will be automatically applied to your purchase. This voucher may not be used in conjunction with any other offer and has no cash value. Only one voucher may be used per transaction. This offer may be withdrawn without prior notice. Coupon must be used conjunction with any other terms and conditions specified by Microchip Technology Inc.



Discount Coupon Code: PRED008

Offer expires December 31, 2008.