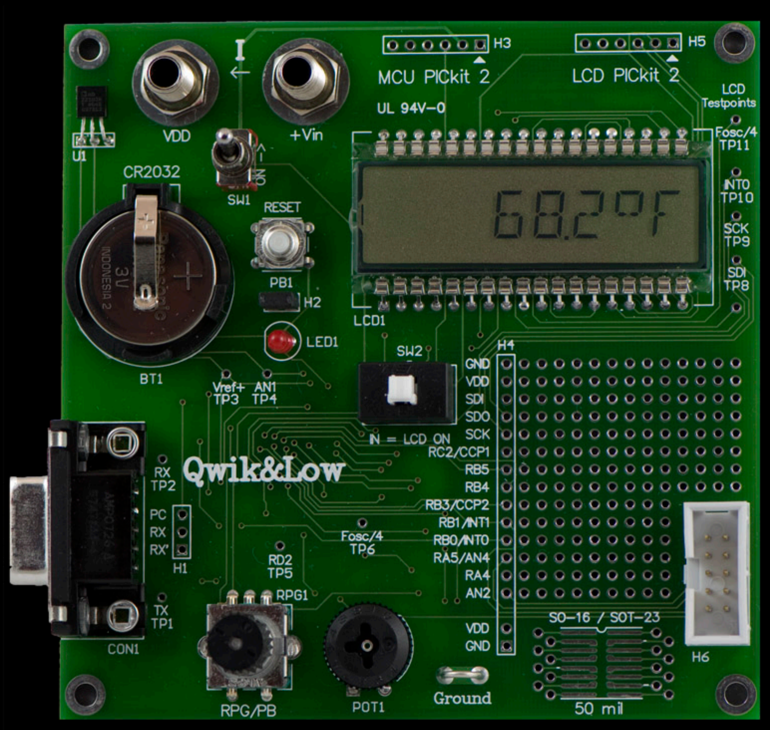
Coin-Cell-Powered Embedded Design



John B. Peatman

Coin-Cell-Powered Embedded Design

John B. Peatman

Professor of Electrical and Computer Engineering
Georgia Institute of Technology

Qwik&Low Books
1418 Iroquois Path, NE
Atlanta, GA 30319
(770) 457-6133
jpeatman@comcast.net

ISBN 978-0-9799770-0-8
9 8 7 6 5 4 3 2 1

This book is available as a $15.50
print-on-demand paperback book from
www.lulu.com

or as a free download along
with supporting material from
www.qwikandlow.com

To six former students who changed the direction of my professional life:

Jim Carreker and Neal Williams
1968–1969

Joe Bazzell and Larry Madar
1989–1990

Rawin Rojvanit and Chris Twigg
2001–2002

CONTENTS

PREFACE

The premise of this book rests on the reality that as embedded microcontrollers reach into virtually all corners of modern life, many of these applications can take advantage of the benefits accruing when powered from a coin cell battery. Some of these are:

- small product size,
- reduced product cost,
- enhanced design simplicity,
- portability,
- electrical isolation.

Manufacturers have met this reality with low-power, extended-supply-voltage versions of their microcontroller chips. For example, Microchip Technology, the number one supplier of 8-bit microcontrollers in the world, is using what they call *nanoWatt Technology*™ features to upgrade their entire microcontroller product line by:

- enhancing the feature set of an older part with new power-controlling modules and new power-sensitive operating modes,
- reducing leakage current,
- extending the supply voltage range from 5.5 V down to 2.0 V,

while, *at the same time*, reducing chip cost. The new parts will handle the old applications. In addition, they open the door to a wide range of new applications.

The intent of this book is to explore how these features impact the design process and the new opportunities these features make possible. This book employs the Qwik&Low board, shown on the cover of this book, as the vehicle for the reader to carry out these explorations. The board uses two Microchip PIC™ microcontrollers mounted on the under side of the board, one as a general purpose microcontroller (PIC18LF4321) and one as the controller (PIC18LF6390) for the board's liquid crystal display. The board was designed by the author and is available from MICRODESIGNS (www.microdesignsinc.com), a local Atlanta company formed by Bill Kaduck and Dave Cornish, two former students and astute designers.

Another goal of this book has been to provide readers with a low-cost tool for their learning. It introduces the reader to code writing for a microcontroller using the C programming language. Microchip's C18 compiler, available to anyone in the form of a free student version, is used throughout. In fact, with just the few constraints introduced, and used, throughout the book to minimize the execution time of algorithms, the student version of the compiler produces machine code that is virtually identical to that produced by the commercial version.

Free supporting tools are available at the author's website, **www.qwikandlow.com.** The centerpiece of this support is *QwikBug*, a debugging user interface. QwikBug supports:

- downloading of an application program to the Qwik&Low board,
- running the program,
- stopping at a breakpoint,
- stepping from one line to the next of the C source file,
- monitoring or changing the content of selected variables and registers.

QwikBug employs the same Background Debug Mode used by Microchip's debugging tools but does so using nothing more than a serial connection (via either a serial cable or a USB-to-serial adapter) to a PC.

This is the author's seventh textbook, with four earlier books published by McGraw-Hill and two by Prentice Hall. In gratitude to the many students who have supported the author's activities over many years, the book is being made available at no cost. However, this decision has its downside. By foregoing the fine and generous support of a commercial publisher, the author will miss their astute help in having it reach its intended audience of professors, students, and professionals. The free download of the book is available from the print-on-demand book printer, **www.lulu.com,** by searching their website for Peatman. The site also lists a printed version of the book carrying the intended first printing price of $15.50, the same price as the author's first textbook, published by McGraw-Hill thirty-five years ago. While the reader awaits the one- to two-week delivery of the printed edition, the free downloaded version can be used.

ABOUT THE BOOK

This book will typically be used in a one-semester course at the senior level. Alternatively, it might be used at the junior level if it is deemed worthwhile to trade the increased engineering sophistication of seniors for the opportunity to follow this course with other design-oriented courses and individual project activities.

The book takes the route of writing application code from the outset in C. To support readers having no experience with C as well as those having extensive experience, the code writing is introduced via a series of template programs. It is the author's experience that this approach brings everyone along, with those new to C able to "do more of the same" as they modify template programs to develop code for new lab projects. More experienced C code writers find plenty to hold their interest as they explore how alternative codings of an algorithm impact the execution time of the algorithm. With an environment in which the microcontroller sleeps (and draws only a few microamperes of current from a coin cell when it is not otherwise doing useful work), a shortened execution time of an algorithm leads directly to a lowering of the average coin cell current.

Code written in C does not translate in an obvious manner into the machine code executed by the microcontroller. That is, shortening the C code to implement an algorithm does not necessarily translate into faster execution of the algorithm. The issues that arise in this translation are discussed throughout the book. Tools are included for monitoring code size and code execution time.

An appendix is included for those who are interested in studying how a C algorithm is implemented in the assembly language of the PIC18LF4321 microcontroller. This is especially useful when code is being debugged that does not seem to do what it is intended to do. The Microchip compiler is "wrapped" in a c18.exe utility that not only compiles the C code into machine code but also generates a qwik.lst file. The appendix explains how this file can be used to understand program code execution.

The book begins with the perspective of Steve Sanghi, CEO and President of Microchip Technology, on low-power designs. The first three chapters present an overview of the environment of the book, low-power operation, and the Qwik&Low board used as a vehicle for all that follows. Chapter 4 introduces the first template program and lays the groundwork for understanding and using it. Chapter 5 extends the first template to the use of the Qwik&Low board's liquid-crystal display. Chapter 6 opens the door for an application program to be tested, with measurement results displayed on the PC monitor. Chapter 7 introduces the use of interrupts to control the timing of events with counters controlled by a crystal oscillator.

Chapter 8 presents an interesting digression. In this chapter a stepper motor and its controller circuit draw power from their own power supply and take control inputs from the Qwik&Low board, serving as a model for more general expansion.

For a course that is driven by a sequence of lab projects, Chapters 9 and 10 are likely to be studied and used, at least in rudimentary form, earlier than the consecutive sequence of chapters would dictate. Chapter 9 explains the use of the microcontroller's analog-to-digital converter. Early in the course, a project might use it to introduce a variable into an algorithm with the ADC output from a one-turn potentiometer input. The chapter also deals with the scaling and display of a ratiometric temperature sensor's output. Chapter 10 discusses the use of a rotary pulse generator (a.k.a., a rotary encoder) to vary an input parameter to an application.

Chapters 11 and 13 provide two takes on timing measurements. Chapter 11 deals with the monitoring of the execution time of an algorithm, a critical issue for average current draw and an elusive issue when coding is done in C. Interrupt timing measurements are also studied. Chapter 13 looks at how the microcontroller's ±2% accurate internal oscillator can be calibrated against the 32768 Hz, ±50-ppm (parts per million) watch crystal oscillator used by two of the microcontroller's timers. It then goes on to convert accurate cycle count measurements into accurate microsecond measurements.

Chapter 12 explores in depth the chip's full interrupt capabilities that were first introduced in Chapter 7. It discusses the drastic effect that alternative ways of expressing interrupt service routines in C can have on execution time.

Chapter 14, like Chapters 9 and 10 before it, might well be introduced before its sequential order in the book. It describes how the nonvolatile EEPROM in the chip can be used to save and restore data through power disruptions.

Chapter 15 explains the 1-wire interface protocol used by Dallas/Maxim for a family of parts. These parts include the silicon serial number chip that gives each Qwik&Low board a unique serial number.

Chapter 16 describes the operation of the PIC18LF6390 LCD controller chip and its firmware. This illustrates the operation of a low-power chip dedicated to a specific, limited task. It carries out the task while drawing an average current of only 6 μA, orders of magnitude less than the current drawn by popular dot-matrix, multiple-character, 5 V displays.

Chapter 17 considers another role for the serial peripheral interface used by the microcontroller to send data to the LCD controller. This interface can be used for attaching chips with additional I/O features to the microcontroller. Two chips are considered: a digital-to-analog converter, and a temperature sensor with a high-resolution digital output.

The book concludes with four appendices. The first describes the installation and use of the QwikBug debugger. The second describes the CPU structure, instruction set, and addressing modes of the PIC18 microcontroller family. The intent is to gain an understanding of the assembly code produced by the compiler of an application program written in C. The last two appendices describe the circuitry of the Qwik&Low board and of the stepper motor control board.

ACKNOWLEDGMENTS

It has been my good fortune over the last several years to be guided in my academic endeavors by Gary May, Chair of the School of Electrical and Computer Engineering at the Georgia Institute of Technology, and by Roger Webb, Gary's predecessor.

This book would not have been written but for the assistance of some able students. Every line of C code in the book was either written by Alex Singh or written by the author and vetted by Alex. He suggested numerous alternatives for expressing a given algorithm in C, all in the interest of examining alternatives produced by the C compiler. He wrote the code for the EEPROM of Chapter 14, for the silicon serial number chip of Chapter 15 and the elegant display controller code of Chapter 16. Alex developed the c18.exe compiler wrapper that invokes the Microchip C18 compiler with carefully selected parameters chosen to optimize code efficiency and execution time and to produce the same resulting machine code for both the commercial version of the compiler and the student version. Alex's wrapper displays the number of bytes of machine code produced by the application program as well as the percentage of available program memory used by the program. Alex also developed the first pass of the www.qwikandlow.com website, posting everything needed by the author's students as they used the book in manuscript form during the 2007 fall semester.

Ryan Hutchinson developed the QwikBug user interface and Kenneth Kinion the QwikBug executive for use with assembly language source files during the 2007 spring semester. Ryan graduated from Georgia Tech in May 2007 and generously agreed to spend nights and weekends during the summer reworking the user interface for source files written in C. The result is a rock-solid tool validated by a semester of student use and fully described by Ryan in Appendix A1.

Louis Howe developed QwikProgram 2, a utility needed to program Kenneth Kinion's QwikBug executive into the PIC18LF4321 chip. This is a task that is thwarted by Microchip's programming utilities. QwikBug requires a programmed "Background Debug Mode" vector because Microchip's utilities reserve the BDM vector use for their own debugger. A purchased Qwik&Low board includes this QwikBug executive programmed into its PIC18LF4321 chip.

David Bauer developed QwikIndent, a utility that reformats the indenting of a source file and aligns the comments appended to the lines of C code. He also developed QwikPH, a utility to update the "Program Hierarchy" table showing the relationships between all the functions in an application program. Both of these utilities employ the comforting feature of renaming the source file before producing a new modified source file and thus providing the ability to revert to the original source file, if desired.

Cody Planteen developed the QwikLst utility. As described in Appendix A2, it is included in the invocation of the c18.exe utility to enhance the resulting assembly language list file into a more readable qwik.lst file. It thereby helps to clarify the C compiler's implementation of an application program.

I have long been indebted to Rick Farmer for his help with the physical design and PC board implementation of assorted projects over the years. Rick's extensive design experience once again has proven crucial to the development of the Qwik&Low board and also of the stepper motor controller board.

At Microchip Technology, application engineers (and former students) Rawin Rojvanit and David Flowers have been quick with answers to elusive questions or guides to their appropriate colleagues. Thus Brett Duane brought clarity to a start-up problem with the Timer1 oscillator. Naga Maddipati helped to track down and fix an elusive leakage current problem in the PIC18LF6390 LCD controller. Steve Sanghi and Eric Sells provided valuable material and insights for Chapter 1.

Bill Kaduck and Dave Cornish of MICRODESIGNS, Inc. have supported my PC board plus book efforts for years. I am grateful for their continued support with boards for this book.

Mark Schutte, my son-in-law and master photographer, has produced all of the photographs in this book. He also designed the book's cover. Leland Strange, President and CEO of Intelligent Systems Corporation, has long provided professional insights as well as fostering my continually evolving class and laboratory activities at Georgia Tech.

This low cost but quality book would not have happened without the gracious and accommodating help of Saradha, Project Manager, and Erin Connaughton, Full Service Manager, with Laserwords of Chennai, India. They have managed the rendering of my penciled line drawings into finished artwork and have produced the finished book pages.

Finally, with the publication of my seventh textbook, I am once again indebted to my wife, Marilyn. From the outset she has supported the development of a text that could serve the reader with the gift of a free downloadable text. Marilyn has translated my handwritten words into readable, understandable text and has been involved with every decision in the production of the book. She brightens the days of my life.

John B. Peatman
Georgia Institute of Technology
jpeatman@comcast.net

Chapter

1

INTRODUCTION

1.1 LOW-POWER DESIGNS: THE WAY FORWARD IN EMBEDDED APPLICATIONS[1]

The need to reduce overall power consumption plays a crucial role in many embedded applications. This intention could be twofold—to extend battery life or meet regulations like Energy Star. Low power and dependable operation are important in embedded applications. As microcontroller designs continue to move into smaller applications with limited power resources, the availability of economical small-pin-count devices with complex peripherals and power-saving features has become increasingly desirable.

Batteries are the power source in low-power designs. Since the advances in battery technology are incremental in nature, it is left to the ingenuity of the embedded designer to get the most out of the power source using suitable microcontrollers and related devices. This can be seen in examples such as PDAs, mobile phones, media players, laptops and other devices. System designers face many challenges posed by compact and portable device electronics.

Chip designers have incorporated several power-saving features into their devices that give designers control over power consumption. The main focus of a successful

1 This section was written by Steve Sanghi, CEO and President, Microchip Technology Inc., Chandler, Arizona.

low-power design is a microcontroller that features a variety of sleep modes and clock modes. The idle modes of the microcontroller power down the CPU while allowing peripherals such as an ADC to continue to operate. To conserve power, in most applications, system controllers need to remain in a low-power state most of the time, waking up periodically under a timer's interrupt to run program code.

Several techniques exist that allow designers to save power. The most obvious one is being able to turn off the peripherals when they are not needed. For example, the Brown-out Reset (BOR) feature is not necessary in battery-powered applications. On the other hand, designers can turn off the CPU using an idle instruction and keep the peripherals running. By invoking the sleep state, the power consumption can be reduced by as much as 96%.

Power saving can be optimally achieved in low-power designs by having the microcontroller control power used by both internal and external peripherals. This requires the partitioning of the design based on power consumption during its operation. When designing a low-power product, determine the required operation states and plan to shutdown unwanted circuitry. As a rule of thumb, if a single peripheral in the device consumes most of the power, worrying about reducing the microcontroller's power will have no impact on the overall system power consumption.

Safety is a high priority in some applications such as medical and mission-critical applications. In these applications, system designers need to provide for emergency situations where an appliance can suffer from loss of power or program control. There could be instances where the loss of a clock source can trigger an erroneous execution of a product's control program. In certain microcontrollers, designers can take advantage of a fail-safe clock monitor feature to detect the loss of a clock source—thus helping the system toward either a gentle shutdown or a "stay-alive" mode, if shutdown is not desired.

By using the latest microcontrollers, designers can implement power-management techniques and build cost-effective low-power devices. Minimizing power consumption in embedded systems enables the use of smaller batteries in portable systems. The combination of lower-power peripherals and microcontroller sleep modes improves the design of a low-power solution. This opens up new product design opportunities in space-constrained applications that could not afford the cost of a microcontroller. The impact of low-power designs can have significant implications from disposable medical devices to consumer electronics and beyond.

1.2 THE LEARNING CURVE

Microcontroller manufacturers develop their products for a highly fractionated market. With approximately two dozen manufacturers of microcontrollers worldwide, no one company dominates the marketplace. The competition is especially fierce among manufacturers of *8-bit* microcontrollers, which pervade low-cost high-volume applications. These microcontrollers, with instructions that operate on 8 bits at a time, afford a manufacturer the best tradeoff for developing families of parts having a rich assortment of features reaching across the family. For example, the Qwik&Low board used in conjunction with this book employs two Microchip Technology microcontrollers. They share the same CPU architecture and instruction set. One is a general-featured, multipurpose microcontroller. The other concentrates its feature set around the ability to serve as a versatile LCD controller that supports a variety of LCD multiplexing modes and a variety of serial and parallel data input formats.

Microchip's 8-bit microcontrollers span from 6-pin to 100-pin parts, each with its own set of on-chip peripheral modules (e.g., an analog-to-digital converter). Microchip has pioneered the use of low-cost reprogrammable flash memory, accounting for over 2 billion of the 5 billion microcontrollers it has sold to date.

For years Motorola dominated the 8-bit microcontroller market with one out of every three microcontrollers sold being a Motorola product. In 2002, a marketplace increasingly supplied by many competitors had reduced Motorola's market share and led to Microchip gaining the number one spot in the number of 8-bit microcontrollers shipped. Microchip has continued to hold this number one spot since then and has, this year, also gained the number one spot in 8-bit microcontroller revenue.

Given this fractionated market and given the benefits that follow market share, Microchip has pursued a careful *learning curve* strategy. The principle of the learning curve states that each doubling of the quantity of parts produced results in a fixed percentage decrease in the unit cost of a part. By passing these reduced costs along to customers in the form of reduced prices, Microchip has gained market share. As a consequence, even sophisticated parts such as those used on the Qwik&Low board have a unit price of about $3.

The payoff for users of this book is that their learning is directed toward a family of parts that finds wide use and is competitively priced. Just as important, readers will learn techniques for using a microcontroller in an energy-efficient manner that will translate to another manufacturer's microcontroller. Even as students graduate and find themselves immersed in a company environment with tools and facilities organized to use another manufacturer's microcontroller, the skills and perspectives gained here will reap dividends there.

1.3 THE PIC18LF4321 MICROCONTROLLER

The general-purpose microcontroller employed in this book is shown in block diagram form in Figure 1-1. It has the diversity of resources that make it a good match for a wealth of applications. Its features also make it a fine vehicle for exploring applications where power is supplied by a wafer-thin coin cell. Alternative approaches to hardware and program code organization will be compared for their resulting average current draw from the coin cell.

1.4 THE PIC18LF6390 LCD CONTROLLER

The Qwik&Low board includes a second microcontroller with internal resources dedicated to the ongoing task of updating and refreshing a liquid-crystal display (LCD). This second microcontroller illustrates one of the trends in the world of low-power applications. This microcontroller includes an LCD module that autonomously refreshes the LCD while the rest of the chip sleeps and draws virtually no power. Only

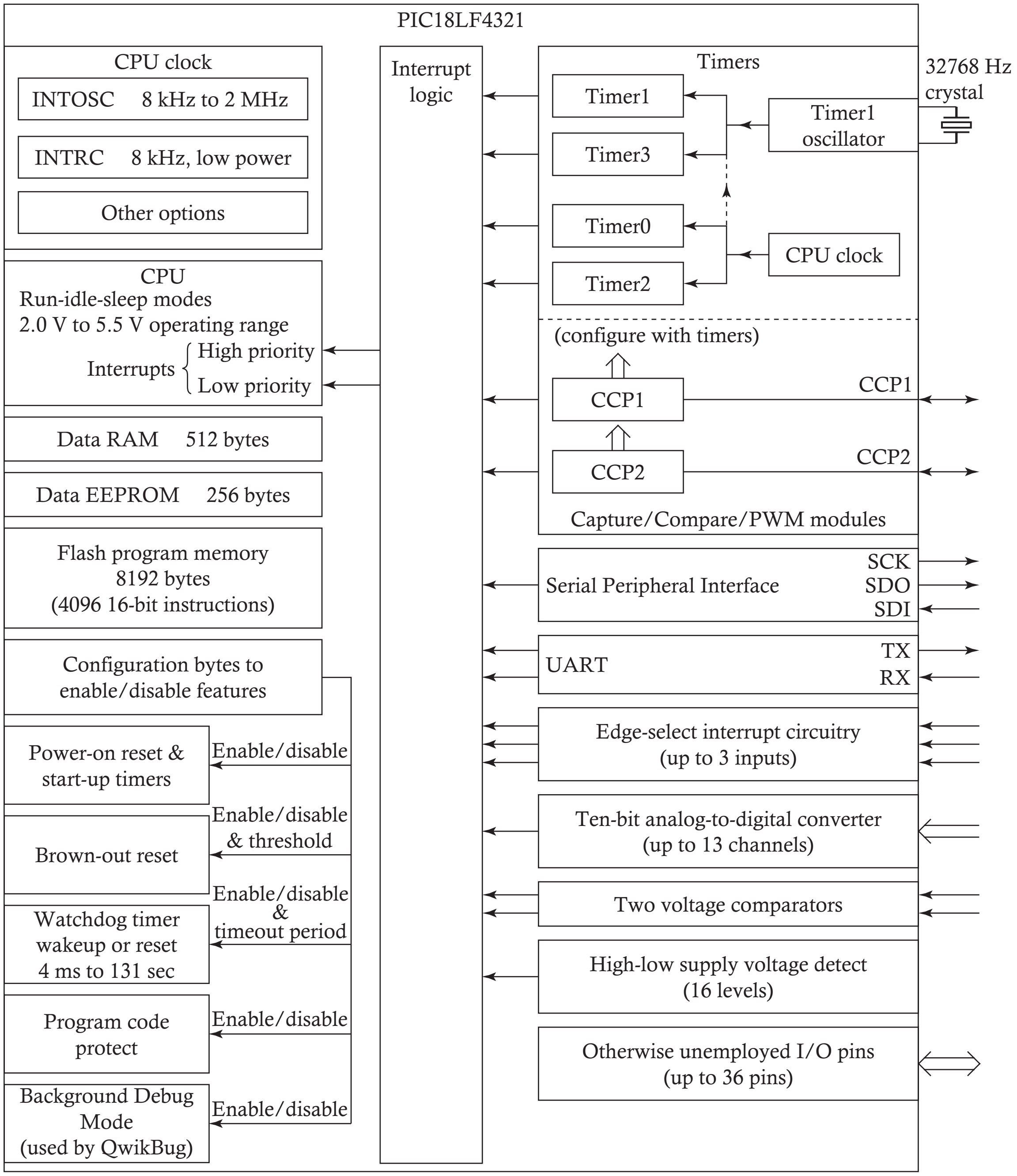

PIC18LF4321
CPU clock
INTOSC 8 kHz to 2 MHz
INTRC 8 kHz, low power
Other options
CPU
Run-idle-sleep modes
2.0 V to 5.5 V operating range
Interrupts
High priority
Low priority
Data RAM 512 bytes
Data EEPROM 256 bytes
Flash program memory
8192 bytes
(4096 16-bit instructions)
Configuration bytes to enable/disable features
Power-on reset & start-up timers
Enable/disable
Brown-out reset
Enable/disable & threshold
Watchdog timer wakeup or reset 4 ms to 131 sec
Enable/disable & timeout period
Program code protect
Enable/disable
Background Debug Mode (used by QwikBug)
Enable/disable
Interrupt logic
Timers
Timer1
Timer3
Timer0
Timer2
Timer1 oscillator
32768 Hz crystal
CPU clock
(configure with timers)
CCP1
CCP2
CCP1
CCP2
Capture/Compare/PWM modules
Serial Peripheral Interface
SCK
SDO
SDI
UART
TX
RX
Edge-select interrupt circuitry (up to 3 inputs)
Ten-bit analog-to-digital converter (up to 13 channels)
Two voltage comparators
High-low supply voltage detect (16 levels)
Otherwise unemployed I/O pins (up to 36 pins)

FIGURE 1-1 Block diagram of PIC18LF4321 microcontroller

when a string of characters making up a new message is received by the chip does the CPU awaken and process the characters, load the registers used autonomously by the LCD module, and then return to sleep.

1.5 QWIKBUG DEVELOPMENT ENVIRONMENT

The PIC18LF4321 microcontroller on the Qwik&Low board has been programmed with an executive *kernel* that works with its counterpart, QwikBug, running on a PC. QwikBug supports application programs written in C and compiled using Microchip's C18 compiler. QwikBug can be used to download a program, run it, stop at a breakpoint, "single step" successive lines of C source code, examine and modify registers and RAM variables, and serve as a display of character strings for an application program (in addition to the Qwik&Low board's LCD).

All of these tools are free, beginning with the Student Edition of Microchip's C18 compiler. While "Student Edition" may sound worrisome, it is available to anyone and is identical to the commercial version for 60 days, after which some of its code optimization is withdrawn. The author has found that the relatively small program code examples used in this book compiled to virtually identical "hex" files whether using the "expired" student version of the compiler or the "site license" commercial version.

The QwikBug "install" program can be downloaded and installed from the author's www.qwikandlow.com website to a PC. If the PC has a serial port, a standard "straight-through" serial cable is used for the connection to the Qwik&Low board. Otherwise, a USB-to-serial adapter cable is used. Once installed, QwikBug presents the easy-to-use, uncluttered user interface described in Appendix A1.

The source code for QwikBug is freely available from the www.qwikandlow.com website for anyone interested in making modifications to it (e.g., to modify its choice of configuration options, to change the watchdog timer timeout period from 16 ms to 8 ms). Reprogramming the modified QwikBug into the chip requires a PICkit™2 programmer and a special programming utility, QwikProgram 2, as described in Appendix A1. The need for a special utility arises because of QwikBug's use of the chip's Background Debug Mode vector. This vector is located within the PIC18LF4321 chip at an address that Microchip's PICkit 2 programming utility prohibits accessing, reserving its use for their in-circuit debugging tool.

1.6 PROGRAMMING WITH THE PICKIT 2 PROGRAMMER

Microchip's low-cost ($35) PICkit 2 programmer can be used to program either the PIC18LF4321 MCU or the PIC18LF6390 LCD controller. If application code is programmed into the MCU directly (i.e., without QwikBug), the program can be run directly from reset, complete with the user program's "configuration byte" choices (which are normally ignored, deferring to QwikBug's choices). The LCD controller code can be modified to enhance its user interface.

Figure 2-11 in the next chapter illustrates the programming connection. It requires the addition of a 6-pin male-to-male header inserted into the 6-pin female socket of the PICkit 2. The resulting male probe of the PICkit 2 is inserted (but not soldered) into the 6-pin unpopulated header pattern labeled "MCU PICkit 2" on the Qwik&Low board. For programming the LCD controller, the header pattern labeled "LCD PICkit 2" is used.

Chapter

2

LOW-POWER OPERATION

2.1 OVERVIEW

This chapter considers the opportunities and challenges of using a popular MCU (microcontroller unit) under the constrained design goal of powering it with a low-cost 3-V coin cell. The specifications of the coin cell will lead to a consideration of the microcontroller and its power-saving operating modes. A vehicle for testing operating alternatives will take the form of a manufactured *Qwik&Low* board. A case will be presented for writing code in C rather than in assembly language.

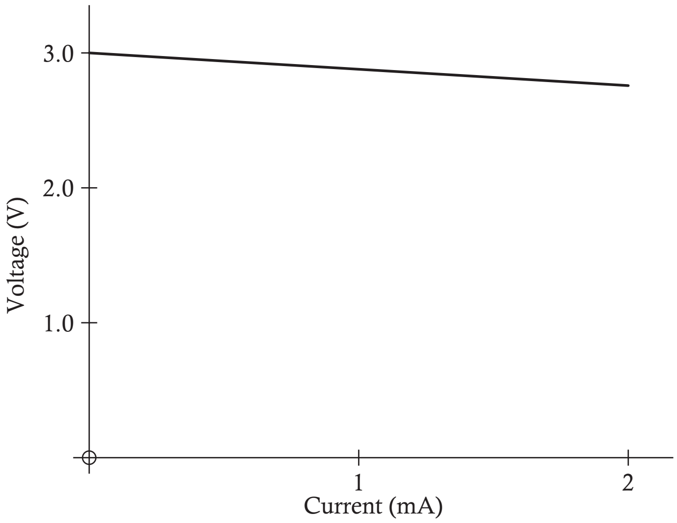
2.2 CR2032 COIN CELL

The low-cost ($0.25), popular CR2032 coin cell is shown in Figure 2-1. Note that while this coin cell is specified for a standard discharge current of 0.4 mA, a larger current can be drawn from it subject to degradation from its specified life of 220 mAh. The challenge will be to explore ways to organize an application so as to draw only a few tens of microamperes from the coin cell, perhaps with brief excursions to currents in the milliampere range. By keeping such excursions short, the average current can be minimized.

(a) Coin cell and a dime

Nominal voltage = 3 V
Rated capacity = 220 mAh (milliampere-hours)
Standard discharge current = 0.4 mA

(b) Specifications



(c) Voltage vs. Current

FIGURE 2-1 CR2032 lithium coin cell

2.3 A PIC18LFxxxx FAMILY OF MICROCONTROLLERS

Microchip Technology's microcontrollers employing nanoWatt Technology™ offer a user a variety of ways to operate the chip to minimize the current draw on the coin cell. Other than the amount of program memory and RAM, each chip listed in Figure 2-2a includes essentially the same feature set, listed in Figures 2-2b and c. This book will focus on the low-power features of Figure 2-2b while using the I/O features of Figure 2-2c.

44-pin TQFP package	28-pin DIP package	Flash program memory (kilobytes)	RAM (bytes)	44/28-pin price (1-25)
PIC18LF4221-I/PT	PIC18LF2221-I/SP	4	512	\$3.24/\$2.96
PIC18LF4321-I/PT	PIC18LF2321-I/SP	8	512	\$3.41/\$3.11
PIC18LF4420-I/PT	PIC18LF2420-I/SP	16	768	\$6.71/\$5.25
PIC18LF4520-I/PT	PIC18LF2520-I/SP	32	1536	\$7.38/\$5.83

(a) Price versus program memory and RAM size (prices from www.microchipdirect.com).

Eight internally generated, 2% accurate CPU clock frequencies (2 MHz down to 8 kHz).
Separate very low power but 10% accurate internal 8 kHz CPU clock.
Another very low power oscillator using external 32768 Hz watch crystal for 50 ppm accuracy.

Power-managed modes of operation:
Run mode: CPU on; peripherals on
Idle mode: CPU off; peripherals on
Sleep mode: CPU off; peripherals off

Operation down to 2.4V for CPU clock of 2 MHz or less.

Watchdog timer for periodic wakeup from lowest-power sleep mode.
Timeout period of 4 ms, 8 ms, 16 ms, 32 ms, . . . , up to over 2 minutes.

(b) Low-power-achieving features.

Up to 32/25 I/O pins on 44-pin/28-pin parts.
Four sixteen-bit timers plus associated circuitry for versatile timing measurement and control.
13/10 analog-to-ten-bit-digital converter inputs (44-pin/28-pin parts).
Two voltage comparators.
SPI and I^2C serial peripheral expansion support.
USART asynchronous and synchronous serial communication support.
Twenty interrupt sources.

(c) Input/output functions

FIGURE 2-2 Family features of four PIC nanoWatt Technology™ microcontrollers

The table of Figure 2-2a lists the distinguishing amounts of program memory and RAM data memory plus the 2007 price for four otherwise virtually identical pairs of chips. Each pair includes a 44-pin surface-mount part and the same microcontroller in a 28-pin DIP package (with 11 of its I/O pads not brought out to DIP pins). Although even the 4221/2221 pair with 4,096 bytes of program memory should suffice for the tasks of this text, the "sweet spot" 4321/2321 pair, with double the program memory for essentially the same price, will be used throughout.

2.4 INTOSC AND INTRC, THE INTERNAL OSCILLATORS

Figure 2-3a illustrates the circuitry that selects one of the eight internally generated CPU (central processing unit) clock frequencies, from 2 MHz down to 7.8 kHz. The highest seven of these frequencies are derived from a relatively accurate (±2%) 8-MHz oscillator, a programmable divider, and a final divide-by-four circuit. The lowest frequency of about 7.8 kHz can be derived in either of two ways. The lowest-power choice uses the relatively inaccurate INTRC oscillator. The divided-down INTOSC oscillator provides a more accurate clock frequency, but draws roughly an extra 150 µA of current from the 3-V coin cell as will be seen in Figure 2-4.

The internal oscillator frequency choice is under the control of a user program. By writing any of seven choices to the chip's **OSCCON** register, any of the upper seven frequencies can be selected. If the binary value 00000010 is written to **OSCCON**, then the lowest frequency, 7.8 kHz, will be selected. In this case, the most-significant bit of the chip's **OSCTUNE** register can be set to select the higher-power, more accurate INTOSC source. Clearing this bit will select the low-power, less accurate INTRC source.

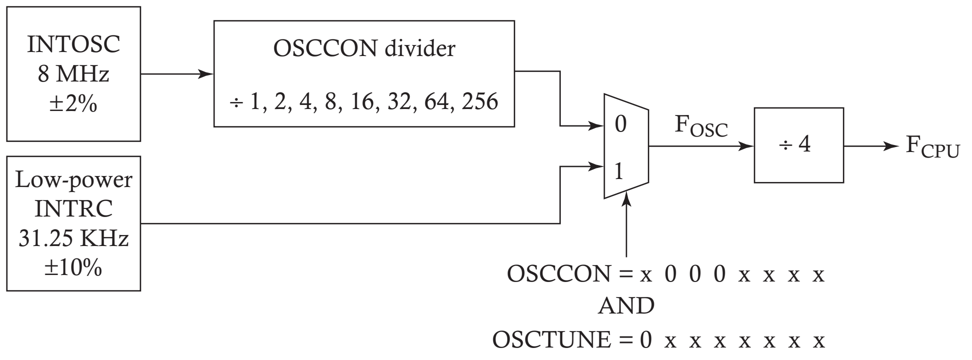
The effect of the choice of CPU clock upon the current drawn from a 3-V coin cell powering the chip is illustrated in Figure 2-4. These are typical values taken from actual measurements. Given that Microchip's data sheet does not address all the variations of interest here, the table values provide design guidance by way of their comparative values.

If the CPU were continuously clocked, the data of Figure 2-4 would tell the whole story of the CPU's relatively high current versus clock rate. Added to this current would be the current drawn by peripheral functions, both in the microcontroller and external to it.

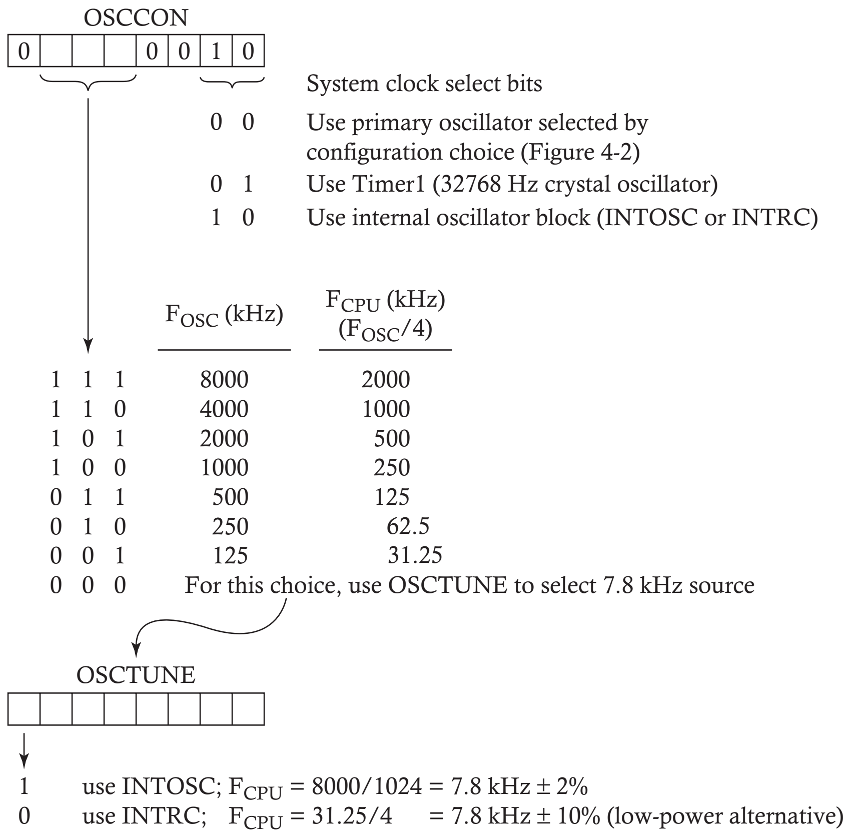
2.5 INTERMITTENT SLEEP MODE OPERATION

The PIC18LFxxxx microcontrollers include a "sleep" instruction that, when executed, causes the CPU clock to stop. With nothing being clocked, the chip exhibits a measured value of leakage current of 0.1 µA. By awakening the chip from its sleep mode periodically, it can respond to inputs and control outputs and then go back to sleep. This is illustrated in Figure 2-5.

To execute instructions in periodic bursts in this way, the microcontroller needs a mechanism that can awaken it periodically. A built-in mechanism that is satisfactory for many applications is the PIC18LFxxxx watchdog timer, illustrated in Figure 2-6. Its programmable scaler can be used to create watchdog timeout periods of 4 ms, 8 ms, 16 ms, 32 ms, . . ., up to something over 2 min. The choice of divider is selected when the chip is programmed with what are called its *configuration bytes*. These bytes select features of the chip that cannot be altered during the execution of a user program.



(a) CPU clock derivation from INTOSC or INTRC oscillators.



	System clock select bits
0 0	Use primary oscillator selected by configuration choice (Figure 4-2)
0 1	Use Timer1 (32768 Hz crystal oscillator)
1 0	Use internal oscillator block (INTOSC or INTRC)

	F_{OSC} (kHz)	F_{CPU} (kHz) ($F_{OSC}/4$)
1 1 1	8000	2000
1 1 0	4000	1000
1 0 1	2000	500
1 0 0	1000	250
0 1 1	500	125
0 1 0	250	62.5
0 0 1	125	31.25
0 0 0	For this choice, use OSCTUNE to select 7.8 kHz source	

1	use INTOSC; F_{CPU} = 8000/1024 = 7.8 kHz ± 2%
0	use INTRC; F_{CPU} = 31.25/4 = 7.8 kHz ± 10% (low-power alternative)

(b) Clock selection registers

FIGURE 2-3 INTOSC and INTRC clock sources

The watchdog timer can be enabled with a bit of a configuration byte, once and for all, independent of what a user program might choose to do. Alternatively, as shown in Figure 2-6, setting or clearing the **SWDTEN** bit of the **WDTCON** register can leave the enabling/disabling under program control. Whenever the watchdog timer is

Clock source	Nominal Fosc	Actual Fcpu (Fosc/4)	CPU clock period	Current draw
INTOSC	8 MHz	1.966 MHz	0.508 µs	1.750 mA
INTOSC	4 MHz	983 kHz	1.02 µs	1.036 mA
INTOSC	2 MHz	492 kHz	2.03 µs	674 µA
INTOSC	1 MHz	246 kHz	4.06 µs	486 µA
INTOSC	500 kHz	123.5 kHz	8.10 µs	390 µA
INTOSC	250 kHz	61.7 kHz	16.2 µs	343 µA
INTOSC	125 kHz	30.8 kHz	32.5 µs	207 µA
INTOSC	31.25 kHz	7.69 kHz	130 µs	206 µA
INTRC	31.25 kHz	8.19 kHz	122 µs	64 µA

FIGURE 2-4 Current draw of PIC18LF4321 when operating continuously from a 3V coin cell versus CPU clock source and frequency. (All internal and external functions other than CPU instruction execution are inactive.)

enabled, the low-power INTRC oscillator is enabled for use with the watchdog timer, independent of whether the INTRC oscillator is also selected as the CPU clock. The running of INTRC plus the watchdog timer together draw a measured current of 2.2 µA, a miniscule current for implementing this intermittent sleep mode of operation.

The choice of the watchdog timer's divider can be used to shorten response time to input/output demands. Alternatively, it can be used to reduce average current by lengthening the "T_{period}" interval of Figure 2-5.

Another issue bearing on this choice is the effect of the contact bounce exhibited by pushbutton switches and keypad switches employed in a user interface. If T_{period} is selected to be larger than the maximum contact bounce time of any switches in an application, then sensing the state of a switch during successive intervals effectively debounces the switch, as shown in Figure 2-7. Figure 2-7a illustrates a circuit for sensing the press of a pushbutton. During successive awakenings of the chip, the input is

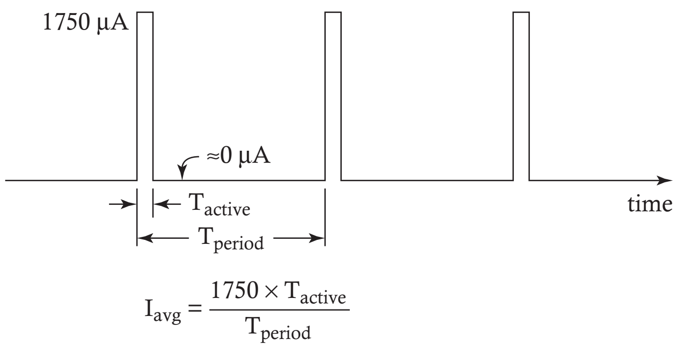
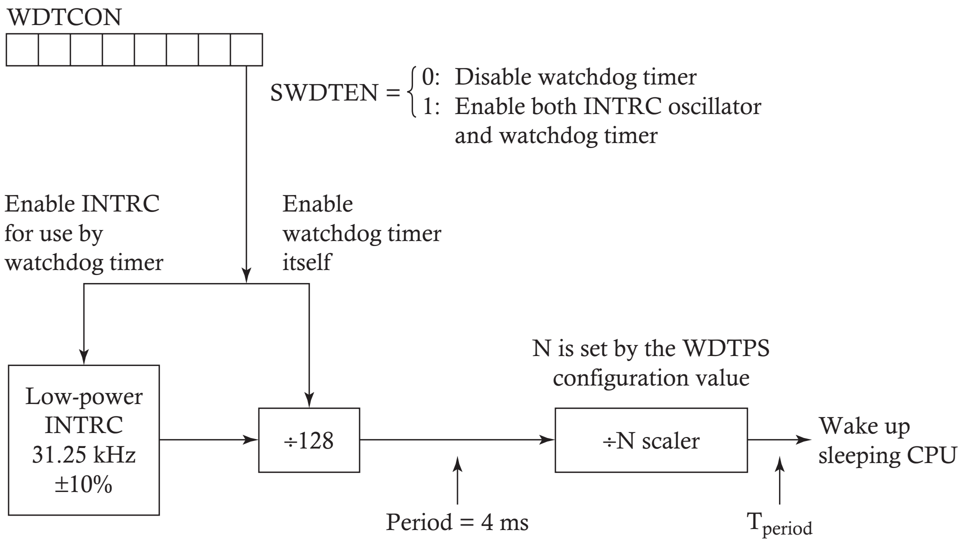


FIGURE 2-5 Effect of intermittent sleep mode upon average current. (shown for F_{OSC} = 8 MHz, F_{CPU} = 2 MHz)


WDTCON
SWDTEN = { 0: Disable watchdog timer; 1: Enable both INTRC oscillator and watchdog timer
Enable INTRC for use by watchdog timer
Enable watchdog timer itself
Low-power INTRC 31.25 kHz ±10%
÷128
N is set by the WDTPS configuration value
÷N scaler
Wake up sleeping CPU
Period = 4 ms
T_period

(a) Circuit

WDT	
OFF :	Watchdog timer disabled; control is passed to the SWDTEN bit of the WDTCON register
ON :	Watchdog timer is enabled, regardless of the state of the SWDTEN bit

WDTPS	T_{period}
1	4 ms
2	8 ms
4	16 ms
8	32 ms
16	64 ms
•	
•	
•	
32768	≈2 minutes

(b) Configuration options programmed into the chip (unavailable for change by the user program)

$$I_{(INTRC+WDT)} = 2.2\ \mu A$$

(c) Measured value of current

FIGURE 2-6 Watchdog Timer

read. Action is taken when the pin is read as a 0 and the previous reading was a 1. Before the keypress, a succession of 1s was read. While the key is held down, a succession of 0s is read. If the key is bouncing at the moment the key state is read and if it is read as a 1, then a single change from 1 to 0 will be detected at the time of the following

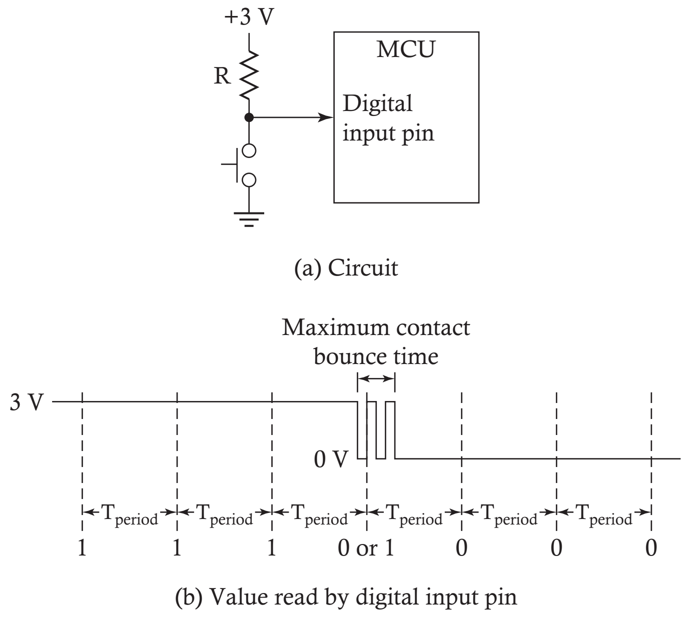

+3 V
R
MCU
Digital input pin
(a) Circuit
Maximum contact bounce time
3 V
0 V
T period
1
1
1
0 or 1
0
0
0
(b) Value read by digital input pin

FIGURE 2-7 Debouncing an electromechanical switch

sample. If the bouncing key is read as a 0, a single change from 1 to 0 will be noted at the time of this sample. In neither case does the

$$\cdots 1 \to 1 \to 1 \to 0 \to 1 \to 0 \to 1 \to 0 \to 0 \to 0 \cdots$$

sequence register as anything other than a single $1 \to 0$ transition.

The maximum contact bounce time of small electromechanical switches is commonly specified as less than 10 ms. If $T_{period} = 4$ ms, the user program just needs to check the state of a switch every third awake time to debounce it. For $T_{period} = 8$ ms, the switch can be checked every other awake time. For $T_{period} = 16$ ms, the switch is checked every awake time, with no keybounce ever seen.

2.6 EFFECT OF CLOCK FREQUENCY

A final consideration when operating the MCU in this intermittent sleep mode is the effect of the choice of F_{OSC}, the frequency of the clock source. First consider the commonly occurring case where the awake time is determined solely by the time it takes to execute a user program that is not delayed by pauses while waiting for a peripheral function (e.g., a timer or a serial data transfer module). In this case, the awake time (and hence the average current draw on the 3-V coin cell) is proportional to the CPU clock period times the instantaneous current draw. Figure 2-8 illustrates the average current draw if a user program takes 100 CPU clock cycles to execute every 16 ms. The data for this table are drawn from Figure 2-4. The average current is calculated as

$$I_{avg} = \left\langle \frac{T_{exec}}{16\ \text{ms}} \right\rangle \times I_{awake}$$

where T_{exec} is the time to execute code for 100 CPU clock cycles.

The conclusion to be drawn from Figure 2-8 is dramatic. The choice of a high value of F_{OSC} minimizes current draw. At the other extreme, the choice of the low-current INTRC oscillator produces a higher average current draw than all but the lowest F_{OSC} derived from the INTOSC oscillator. It is important to remember that these conclusions rest on having an application that is not delayed by waiting for the completion of a task by a peripheral module.

2.7 USER PROGRAM STEPS TO REDUCE CURRENT DRAW

The microcontroller used with this text has two ways to reduce the average current draw when user code includes externally produced delays. The simplest approach is to slow the CPU clock, perhaps by switching to the INTRC oscillator, and repeatedly polling the state of an input pin. When the pin changes state, indicating the completion of the external event, the CPU clock is switched back to the normal clock and code execution continues using fast clock cycles.

A second way to reduce the current draw while waiting for the completion of an external event makes use of the chip's interrupt circuitry. Rather than repeatedly polling the state of an input pin and waiting for it to indicate that an external event has been completed, the chip can first be put to sleep. The change in state of the input pin can awaken the chip and resume execution at the same point simply by using what would normally be an external interrupt pin for this purpose.

If code execution includes a delay caused by an *internal* module (e.g., waiting for the completion of a serial transfer), then the clock for the module is usually the same as that used to clock the CPU. In this case, changing the module's frequency or stopping it is usually not an option. However, a variation of the interrupt approach just

Clock source	Nominal Fosc	Pcpu Actual CPU clock period	Texec Time to execute 100 clock periods)	Iawake	Iavg
INTOSC	8 MHz	0.508 µs	50.8 µs	1750 µA	5.6 µA
INTOSC	4 MHz	1.02 µs	102 µs	1036 µA	6.6 µA
INTOSC	2 MHz	2.03 µs	203 µs	674 µA	8.6 µA
INTOSC	1 MHz	4.06 µs	406 µs	486 µA	12 µA
INTOSC	500 kHz	8.10 µs	810 µs	390 µA	20 µA
INTOSC	250 kHz	16.2 µs	1,62 ms	343 µA	35 µA
INTOSC	125 kHz	32.5 µs	3.25 ms	207 µA	42 µA
INTOSC	31.25 kHz	130 µs	13.0 ms	206 µA	167 µA
INTRC	31.25 kHz	122 µs	12.2 ms	64 µA	49 µA

FIGURE 2-8 Average current calculations

Clock source	Nominal Fosc	Irun Run mode current (CPU on; peripheral on)	Iidle Idle mode current (CPU off: peripherals on)	Iidle / Irun
INTOSC	8 MHz	1750 µA	834 µA	.48
INTOSC	4 MHz	1036 µA	503 µA	.49
INTOSC	2 MHz	674 µA	325 µA	.48
INTOSC	1 MHz	486 µA	235 µA	.48
INTOSC	500 kHz	390 µA	179 µA	.46
INTOSC	250 kHz	343 µA	158 µA	.46
INTOSC	125 kHz	207 µA	148 µA	.71
INTOSC	31.25 kHz	206 µA	140 µA	.68
INTRC	31.25 kHz	64 µA	5 µA	.08

FIGURE 2-9 Idle mode current compared with run mode current

discussed can be used. These microcontrollers include an *idle* mode. When the **IDLEN** bit in the **OSCCON** register is set, the subsequent execution of a "sleep" instruction will stop the clock to the CPU, but maintain it to the peripheral modules. Virtually all of the internal peripheral modules include an interrupt mechanism that can be set up to awaken the CPU when its task is complete. As shown in Figure 2-9, stopping the clock to the CPU while maintaining it to the chip's peripheral modules cuts the current draw by one half for any but the lowest frequencies of the INTOSC oscillator. The 92% reduction in current when clocking only the chip's peripheral modules with the INTRC oscillator may be a reflection of how little current the low-power oscillator itself draws. In contrast, the leveling off of both the run-mode current and the idle-mode current of the chip when clocked by the INTOSC oscillator and its scaler for its lower frequencies may be a reflection of the substantial current drawn by the 8-MHz oscillator itself.

2.8 THE QWIK&LOW BOARD

This book is intended to be supported by the Qwik&Low board shown in Figure 2-10. It will be described in more detail in Chapter Three. Here are its salient features:

- It uses peripherals whose current draw can be reduced to zero (either under control of the PIC18LF4321 MPU or by opening a switch or jumper) to monitor the MPU current.
- It uses an eight-character LCD having a PIC18LF6390 chip as its controller, to reduce the LCD-plus-LCD-controller current draw to 5 µA.
- It includes a prototype area to allow the use of additional peripheral devices.

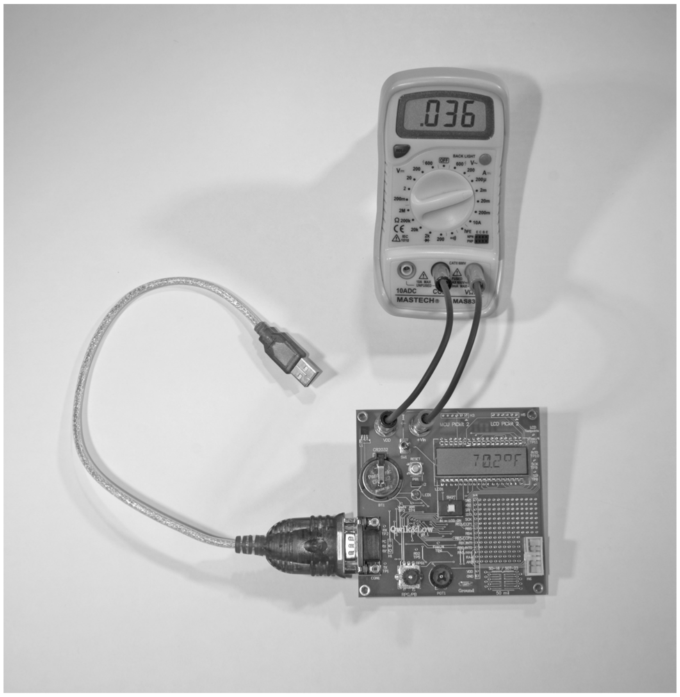


FIGURE 2-10 Qwik&Low board

- The PIC18LF4321 can be programmed with Microchip's low-cost ($35) PICkit 2 programmer shown probing the Qwik&Low board in Figure 2-11.
- Alternatively, the PIC18LF4321 can be linked to a PC via either a $5 serial cable (for PCs having a DB-9 serial port) or a $10 USB-to-serial adapter. Then a free QwikBug utility (discussed in Chapter Three) running on the PC can download and run a C-compiled "hex" file on the PIC18LF4321. QwikBug supports debugging with breakpoint capability, single stepping, and variable watching and modifying. A window in QwikBug's display can be written to by a user program to extend the display capability of the board beyond that of the eight-character LCD.

2.9 CODING IN C

Most microcontroller applications developed professionally have had their code written in C rather than in the microcontroller's assembly language. There are several reasons for C to be preferred:

- C code, even without comments, is close to being self-documenting. Thus it is easier for others to understand and augment the original developer's code.
- The C compiler, rather than the code developer, handles functions such as multiplication, division, and table lookup that are built into standard C.

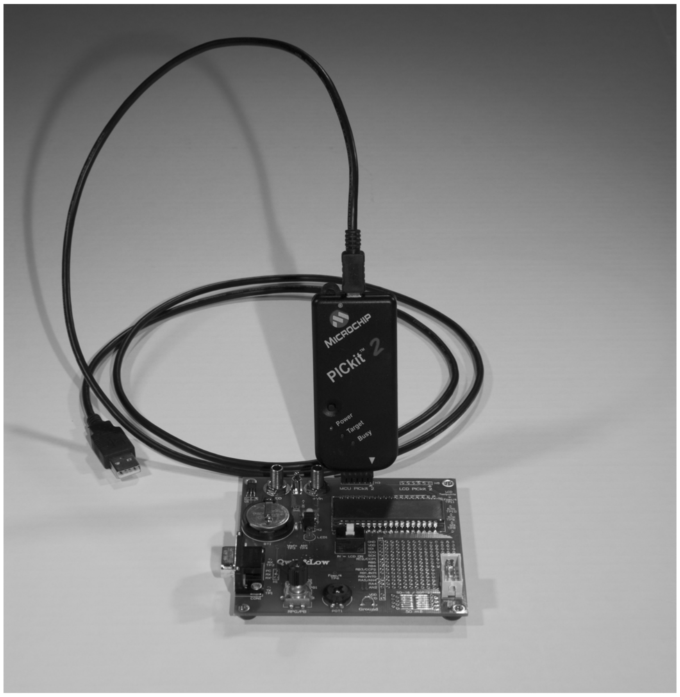


FIGURE 2-11 PICkit 2 programmer

- The need to understand the role of the microcontroller's CPU structure (i.e., its registers, addressing modes, and instruction set) is passed to the C compiler and, thereby, bypassed by the code developer. While there is some justification for lamenting the resulting loss of control over how algorithms are carried out by the microcontroller, there is the compensating accuracy of the resulting C implementation.

Writing code in C is not without its downsides:

- The resulting machine code will be larger than if it had been written in the microcontroller's assembly language. This is generally not a problem, as long as the microcontroller has sufficient program memory to hold the machine code.
- The code developed in C may not execute as fast as if it had been developed in assembly language. However, for issues that really matter, such as measuring a pulse width precisely, the microcontroller includes resources to take over this role, independent of the speed of execution of the program code.
- C compliers are generally expensive. However, whereas the commercial version of Microchip's C18 compiler is pricey, their student version is available free to anybody and provides exactly the same features and optimized compilation for 60 days. After that, only the optimization is reduced. Even so, the resulting machine code is generally satisfactory.
- The code writer must be, or become, familiar with writing code in C. Although this is becoming a common skill for electrical and computer engineering students,

> it is certainly not universally so. However, it is the author's experience that template programs can be used for this purpose. These template programs can progress through a sequence of increasingly complex tasks. Along the way, student projects can build on a given template by "doing more of the same" with an added peripheral device.

Ultimately, the role of the developer of a microcontroller application is to understand the functioning of peripheral devices external to the chip (e.g., a temperature sensor) as well as peripheral modules within the chip. Program control of these peripherals will reduce to testing status bits in registers, setting or clearing control bits in registers, and reading from and writing to registers. These steps are virtually the same, whether implemented in C code or assembly language code. The understanding of how to deal with peripheral devices will be a central theme of this book.

Chapter

3

QWIK&LOW BOARD

3.1 OVERVIEW

The chapter begins with a brief list of items needed to support the book using the Qwik&Low board. The I/O and support circuitry surrounding the PIC18LF4321 on the board are described, as is the board's LCD circuitry.

3.2 EQUIPMENT SETUP

The Qwik&Low Board is built and tested by MICRODESIGNS, Inc. (www.microdesignsinc.com). MICRODESIGNS, Inc. is a 30-year-old company founded by Bill Kaduck and Dave Cornish, two colleagues and former students of the author.

Also needed are the supplies listed in Figure 3-1. Any digital multimeter (DMM) with a microammeter scale having a resolution of 1 µA will serve, but the normal test probes need to be replaced with test leads having banana jacks on each end. An excellent, low-cost DMM that has been found by the author to be sturdy and reliable in his Georgia Tech instructional laboratory is available from www.elexp.com (Part No. F01DMMAS830). They also have banana plug test leads (Part No. F05ALS4).

The Qwik&Low board comes with the PIC18LF4321 programmed with QwikBug. Consequently, the reader does not need to purchase a programmer. Rather, access the

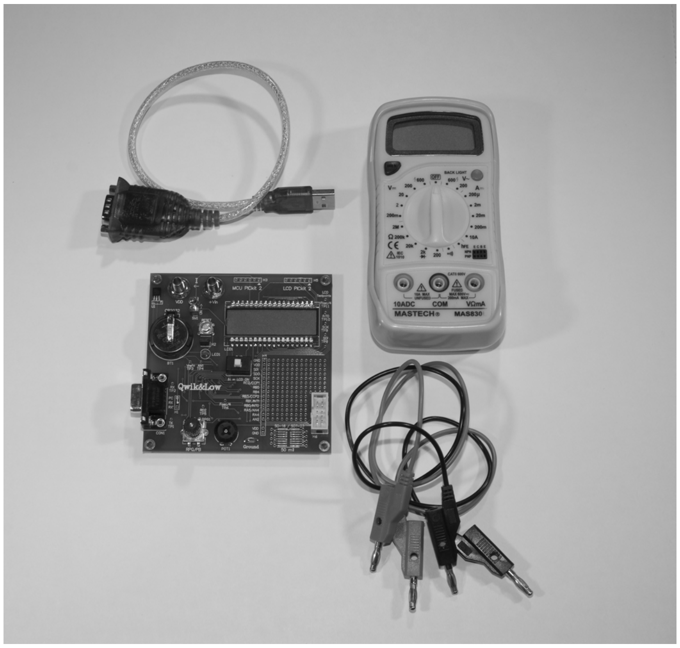


FIGURE 3-1 Required Qwik&Low supplies

author's website, www.qwikandlow.com, to obtain and install the QwikBug utility to run on a PC. This free utility, prepared by Ryan Hutchinson and Kenneth Kinion not only allows the user to download and run a compiled C file on the PIC18LF4321, but also to stop at a breakpoint and single step line by line through the C source file while monitoring and optionally modifying selected watch variables.

If the user chooses to overwrite the QwikBug utility in the chip, Microchip's PICkit 2 programmer ($35) will be needed. If the user later decides to reinstall QwikBug into the board's MCU (microcontroller unit, the PIC18LF4321), a QwikProgram 2 utility developed by Louis Howe is available to download from the www.qwikandlow.com website and then install and run on a PC. QwikProgram 2 programs not only the QwikBug utility itself but also the normally inaccessible *Background Debug Mode* vector located at address 0x200028.

QwikBug employs a serial connection to a PC. To meet the requirements of the RS-232 standard, such a connection requires the transmit and receive lines of the MCU to be inverted and voltage-translated so that the MCU's 0 V and 3 V levels communicate appropriately with the PC's +15 V and –15 V levels. However, for this nonstandard *test* serial port, the Qwik&Low board employs transmit and receive signal inversion built into the MCU itself. The MCU's receive input is clamped to the 0 V and 3 V levels by protection diodes built into the chip, with current limited by a 1 MΩ series resistor. The MCU's transmit output voltage swing of 0 V to 3 V does not meet the RS-232 standard, but is sufficient to be interpreted satisfactorily by every PC and every serial-to-USB adapter the author and his students have tried. This simplified connection causes minimal current draw on the coin cell with and without the serial cable connected.

If the reader's PC includes a serial port with its 9-pin male DB-9 connector, all that is needed is the normal *straight-through* DB-9M to DB-9F serial cable. "Straight through" means that the pins of the DB-9M connector are connected to the corresponding pins of the DB-9F connector. If the reader has an up-to-date notebook computer, it probably does not include a serial port. In this case, the USB-to-serial adapter will be needed at a cost of \$10–\$15, available by Googling *USB to serial adapter* to find any one of many sources.

One last, but more expensive, piece of test equipment that finds repeated utility with the Qwik&Low board is an oscilloscope. Both the PIC18LF4321 MCU and the PIC18LF6390 LCD controller have their internal CPU (central processing unit) clock ($F_{OSC}/4$) brought out to a test point on the board. By probing this point for the MCU, a user can see when the chip is awake and when it is asleep, and thereby discern the MCU's duty cycle and the effect that a low duty cycle has on current draw. The LCD controller only awakens when a new display string is sent to it by the MCU. The scope can monitor the duration of the serial transfer. It can also monitor the LCD controller's CPU clock ($F_{OSC}/4$), to discover how long the LCD controller takes to process a received display string from the MCU before returning to sleep.

3.3 INPUT/OUTPUT PERIPHERAL POWER

As shown in Figure 3-2, each of the peripheral components on the Qwik&Low board is connected to the MCU in a way that permits power to be removed from the component. In some cases, an MCU output pin provides the peripheral power. In other cases, the power is applied or removed with a jumper or a switch.

To understand how this control of power to a peripheral translates into average current draw, consider the 20-kΩ one-turn potentiometer. If it were powered directly from the 3-V coin cell supply, it would draw a constant 150 μA, completely overriding those things that can be done to otherwise reduce the average current draw on the coin cell to a few microamperes. By setting bit 7 of **PORTA** and thereby driving the **RA7** pin to 3 V, then converting the analog input to **AN0**, and finally clearing bit 7 of **PORTA**, the average current draw is reduced by an amount proportional to the duty cycle of this operation. For example, assume it takes 30 μs to power-up the potentiometer, enable the analog-to-digital converter (ADC) module, carry out the conversion, power down the potentiometer, and disable the ADC module. If a conversion is carried out every 200 ms (i.e., five times a second), the average current draw due to the potentiometer will be reduced by a factor of

$$\frac{30\ \mu\text{s}}{200\ \text{ms}} = \frac{30\ \mu\text{s}}{200000\ \mu\text{s}} = 0.00015$$

The 150 μA instantaneous current becomes a negligible 0.023 μA average current.

Each of the peripheral devices will be discussed as it is used in subsequent chapters. For now, the role of Figure 3-2 is to illustrate this control of current draw by each device. In the case of the LED, its current draw when turned on is on the order of 1 mA (given a voltage drop of about 2 V in the LED). Although this is higher than desirable as a constant load, if the LED is blinked for just 16 ms every 4 s for a duty cycle of 0.004, a visible blink of light will occur while contributing about 4 μA to the average current draw. Such blinking will be helpful for telling when an undebugged user program is running and not stuck somewhere due to a bug. The jumper allows the relatively heavy LED current to be shut down until such a bug is removed.

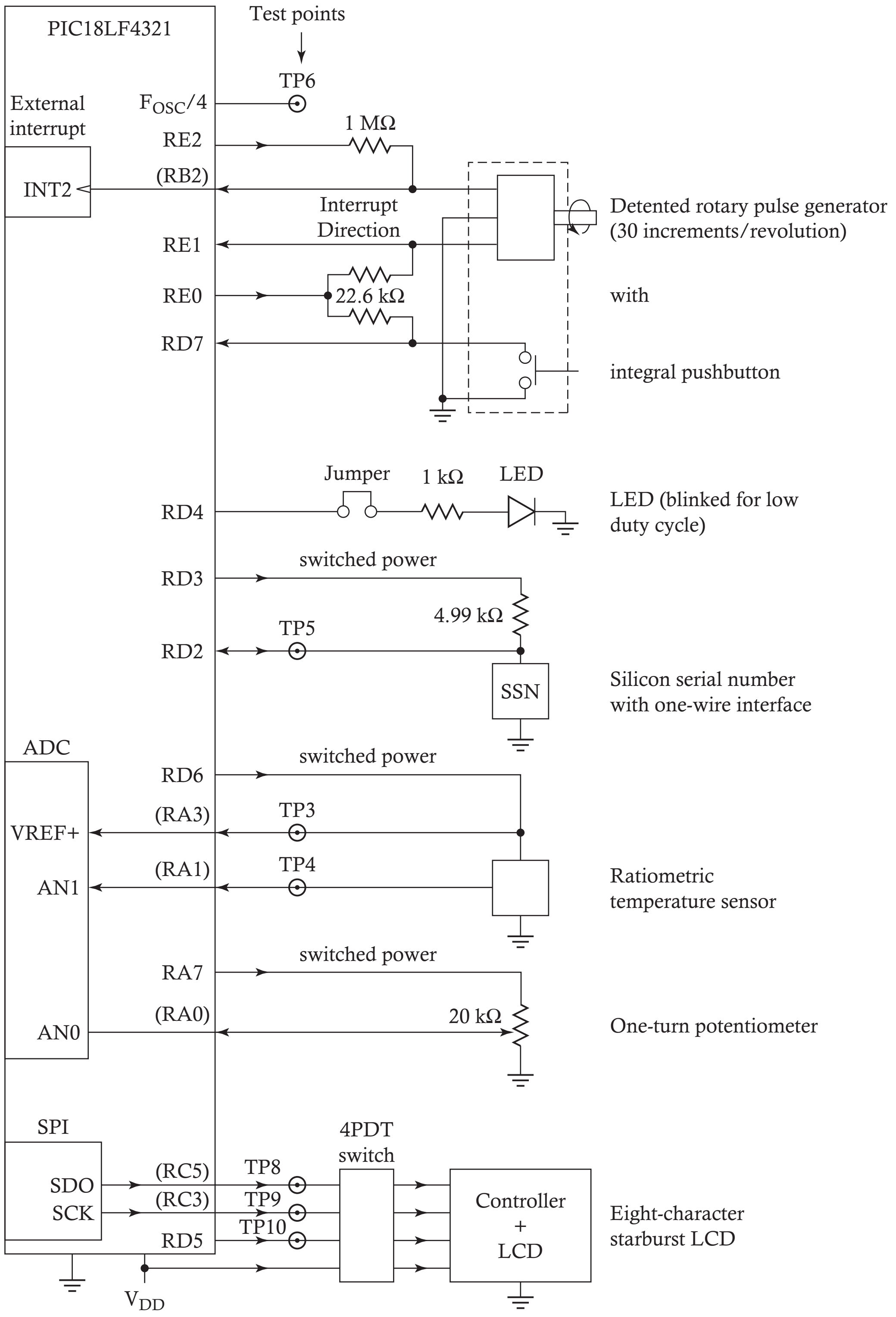


FIGURE 3-2 Qwik&Low MCU control of peripheral power

3.4 POWER SWITCHING AND CURRENT MONITORING

Figure 3-3 illustrates the circuit used to switch on and measure the coin cell current. If the digital multimeter (DMM) is not connected to the board, the power switch works alone to control the supply current. With the DMM connected to the board and set to its "off" position, the board can be powered in either of two ways:

- Flick on the power toggle switch for simplicity, without monitoring the current.
- With the power switch off, rotate the DMM's control knob to its "2 mA" (i.e., 2,000 μA full scale) position to monitor the supply current.

Depending on what scales are traversed between the DMM's "off" setting and its "2 mA" setting, the MCU may experience one or more voltage steps as the control knob is turned. For example, the DMM of Figure 2-10 has an internal resistance of 100 Ω on its "2 mA" scale. But it traverses a "200 μA" scale along the way having an internal resistance of 800 Ω. With its default startup F_{OSC} = 1 MHz, the MCU draws about 400 μA initially, so with a coin cell voltage of 3.00 V, the MCU's V_{DD} steps through the sequence of 0 V to 2.67 V to 2.96 V (even with the LCD's controller switched off). This sequence does not seem to produce a faulty startup. However, if a different DMM is used that causes an unreliable startup of the MCU when powered up in this way, the power-up sequence can be altered to:

1. Turn on the power switch.
2. Turn the DMM's control knob from its "off" position through its various scales to its "2 mA" position.
3. Turn off the power switch.

This discussion raises another point worth noting. Most low-cost DMMs feature a 2,000-count scale for all their measurements. That means that a "2 mA" (or a "2,000 μA") scale has a resolution of 1 μA. If there is also a "200 μA" scale, it will have a resolution of 0.1 μA (good) but also an internal resistance that is, perhaps, eight times higher than that of the "2 mA" scale (bad). With the MCU being operated in an intermittently awake mode with a current in the range of a milliampere, V_{DD} may exhibit a negative blip of tens or hundreds of millivolts each time the MCU awakens to do its useful work. This will not make a difference for digital transducers but may affect the behavior of transducers with analog voltage outputs as well as the MCU's analog-to-digital converter (ADC). Given this situation, a user can simply turn on the power switch for best results from an analog measurement, and turn off the power switch while making the corresponding coin cell current measurement.

3.5 PICkit 2 PROGRAMMER CONNECTION

A new Qwik&Low board comes with its MCU already programmed with QwikBug. Consequently, it does not need the PICkit 2 programmer to load a user program into the chip. Instead, QwikBug uses the serial port connection for this purpose. QwikBug's executive program, residing in the high addresses of the MCU's program memory,

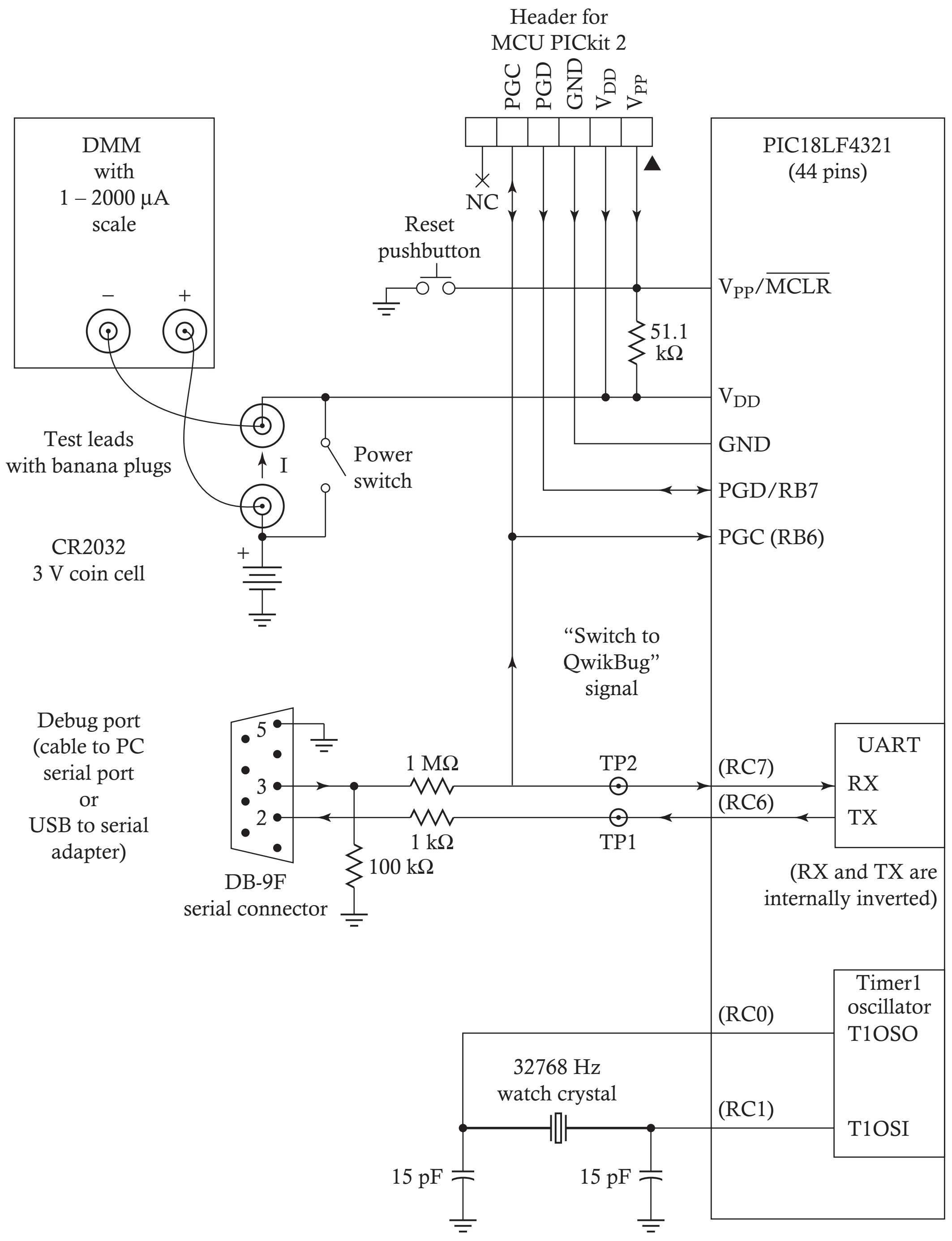


FIGURE 3-3 Qwik&Low MCU support circuitry

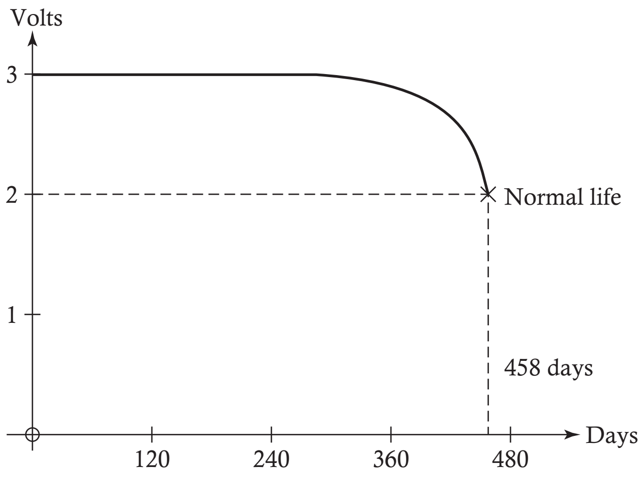
reads in a user program downloaded from a PC and writes it into the low addresses of the MCU's program memory.

QwikBug takes advantage of the same *debug* mode employed by the PICkit 2 programmer. It uses the PGC (RB6) pin of Figure 3-3 to let the PC's QwikBug utility get the attention of the MCU when it is running a user program. By sending a serial character to the Qwik&Low board, the resulting wiggling of the MCU's RX UART pin also wiggles the PGC pin, as shown in Figure 3-3. This will awaken the MCU if it is asleep. Whether or not it is asleep, the user program will be interrupted and will vector to the QwikBug executive program in the MCU. The user program will be paused and control will return to QwikBug.

3.6 EFFECT OF COIN CELL AGING

Over time, the CR2032 lithium coin cell will exhibit a decrease in its loaded output voltage. An especially useful feature of a lithium cell is the flatness of its discharge characteristic, as shown in Figure 3-4. With a fixed load of 20 µA, the characteristic shows no appreciable droop in the voltage over the first half of its rated life. Even after three-quarters of its rated life, the voltage is only down to 2.9 V.

Both the MCU and the LCD controller are specified to operate down to 2.42 V even with F_{OSC} as high as 8 MHz, the clock rate used by the QwikBug executive and by the LCD controller. All but two of the peripheral parts on the Qwik&Low board will operate down to 2.42 V. The two exceptions are the temperature sensor (down to 2.7 V) and the silicon serial number part (down to 2.8 V). But, again, these are both good for more than three-quarters of the coin cell's life.



Nominal capacity = 220 mAh

$$\text{Life with 20 µA load} = \frac{220\text{ mAh}}{20\text{ µA}} = 458\text{ days}$$

FIGURE 3-4 CR2032 discharge characteristic at room temperature with a 20 µA load current

3.7 WATCH CRYSTAL CIRCUITRY

The Qwik&Low board's MCU includes an optional low-power oscillator that can employ an external 32,768-Hz watch crystal to provide 50 parts per million (ppm) frequency accuracy. This oscillator (shown in Figure 3-3) can be used to clock either of two internal 16-bit counters, even as the rest of the chip sleeps. As such, it provides an alternative to the use of the INTRC oscillator and the watchdog timer mechanism of Figure 2-6 and its low-current draw of 2.2 μA. Considering Figure 2-5, the watch crystal oscillator plus Timer1 combination can provide any interval, T_{period}, up to 2 s.

The Timer1 oscillator has two alternative configurations. The configuration selection (discussed in Section 4.3) is made by turning on or off a low-power Timer1 oscillator configuration bit. The low-power option is intended for use with V_{DD} above 4 V. It includes a 3-V regulator to maintain the accuracy of the crystal oscillator even as V_{DD} varies. For operation of the chip with power from a 3-V coin cell, the Microchip application engineers do not recommend using the low-power option. The author's experience is that with the configuration selection

```
LPT1OSC = OFF
```

the Timer1 oscillator will reliably start up and run in less than 0.2 s after it is enabled and will draw, together with the Timer1 counter, about 6.5 μA. In contrast, with the configuration selection

```
LPT1OSC = ON
```

the Timer1 oscillator may not start up and run at all. For those boards that do start up, the startup time can be measured in seconds. For such, the current draw, together with the Timer1 counter, drops to about 1.5 μA.

3.8 QWIK&LOW LCD

The PIC18LF6390 LCD controller, the LCD, and the surrounding circuitry and connections are illustrated in Figure 3-5. With its inputs seeing very slow (37 Hz) changes to what is essentially a very low capacitive load, the LCD has little impact on current draw. The LCD controller's CPU sleeps constantly until the MCU awakens it with a falling edge on its **INT0** interrupt input. The PIC18LF6390's CPU then receives a serially sent string of ASCII-coded characters over an interval of 100 μs–200 μs, translates them, stores the translated data into LCD registers in the chip, and goes back to sleep. Then the LCD module within the chip refreshes the display, drawing just 5 μA to do so.

The 4PDT push-to-make, push-to-break switch of Figure 3-5 permits power and input connections to be disconnected from the MCU and grounded. When the switch goes from off to on, the display treats that operation as a reliable power-on reset. Thus, the switch serves two purposes:

- It powers the LCD down to remove its current draw from a measurement of the board's total current draw.
- It powers the LCD up and initializes it for reliable operation.

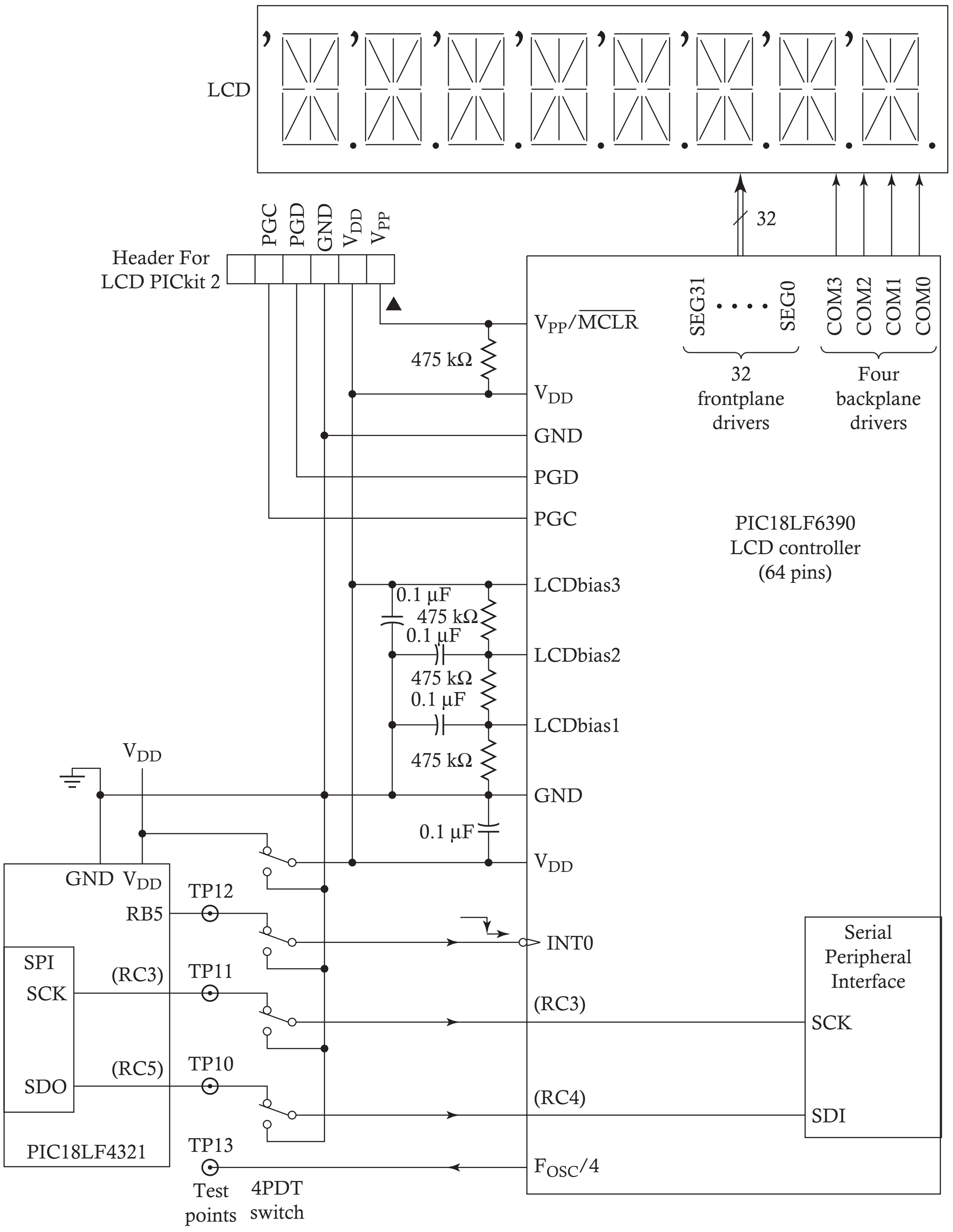


FIGURE 3-5 Qwik&Low LCD

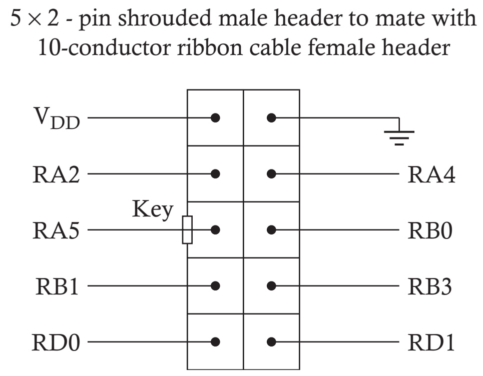

5 × 2 - pin shrouded male header to mate with 10-conductor ribbon cable female header
VDD
RA2
RA4
Key
RA5
RB0
RB1
RB3
RD0
RD1

FIGURE 3-6 Qwik&Low expansion header

The LCD PICkit 2 connection is required during the manufacture of the board and probably not thereafter. If there is a feature to be added to the PIC18LF6390, the LCD source code developed by Alex Singh is available at www.qwikandlow.com.

3.9 EXPANSION HEADER

Referring again to Figure 2-10, note the shrouded 10-pin header located in the lower right-hand corner of the Qwik&Low board. It is designed to provide a 10-conductor ribbon cable connection for power, ground, and the eight other MCU pins shown in Figure 3-6.

Later in the book the connection of the Qwik&Low board to a stepper motor driver board will be considered. That board and its associated stepper motor employ their own wall transformer power supply. The connection draws essentially no current from the Qwik&Low coin cell.

3.10 SUMMARY OF MCU PIN USE

Each of the pins of the PIC18LF4321 chip may be used for its dedicated internal function (e.g., an analog-to-digital input), or as a general-purpose I/O pin, or in a supportive role (e.g., the reset input pin). External to the chip, a pin takes on a role determined by its connection to other devices.

When looking for an otherwise unused pin, the chart of Figure 3-7 will be of assistance. For example, the occurrence of some event can be sensed by the CPU and signaled to a user by setting an otherwise unused pin high. Figure 3-7 indicates that RB0, for example, is available for this purpose. While it could be probed as pin 8 of the MCU chip, it is much more convenient to probe it at the test point labeled RB0/INT0 on the H4 header pattern, residing just to the left of the proto area.

Pin Number	Pin Function	Qwik&Low Function	Test Point	H3 PICkit 2	H4 Proto Area	H6 Connector
1	RC7/RX	RX input from PC or from RX' proto pin	TP2			
2	RD4	Red LED				
3	RD5	Chip select for starburst LCD	TP10			
4	RD6	Power for temperature sensor				
5	RD7	Pushbutton input				
6	GND			x	x	x
7	VDD			x	x	x
8	RB0/INT0				x	x
9	RB1/INT1				x	x
10	RB2/INT2	RPG interrupt input				
11	RB3/CCP2				x	x
12	ICPGC	(Unused debugging feature)				
13	ICPGD	(Unused debugging feature)				
14	RB4				x	
15	RB5				x	
16	RB6/PGC	(Reserved by background debug mode)		x		
17	RB7/PGD	(Reserved by background debug mode)		x		
18	MCLR/VPP	Reset pushbutton		x		
19	RA0/AN0	AN0 input from potentiometer				
20	RA1/AN1	Input from temperature sensor	TP4			
21	RA2/AN2				x	x
22	RA3/AN3/VREF+	Connected to RD6 for VREF+ input	TP3			
23	RA4				x	x
24	RA5				x	x
25	RE0	Power for pushbutton and RPG direction				
26	RE1	RPG direction input				
27	RE2	Power for RPG interrupt input				
28	VDD				x	
29	GND				x	
30	RA7	Power for potentiometer				
31	Fosc/4	Output of CPU clock	TP6			
32	T1OSO	32768 Hz watch crystal for Timer1 osc.				
33	ICRST	(Unused debugging feature)				
34	ICPORTS	Tied to VDD to avoid 28-pin emulation				
35	T1OSI	32768 Hz watch crystal for Timer1 osc.				
36	RC2/CCP1				x	
37	RC3/SCK	SPI clock for LCD	TP9		x	
38	RD0	Optional stepper DIR (pin 9 of H6)				x
39	RD1	Optional stepper STEP (pin 10 of H6)				x
40	RD2	I/O for DS2401	TP5			
41	RD3	Power for DS2401 silicon serial number				
42	RC4/SDI	SPI input			x	
43	RC5/SDO	SPI output to LCD	TP8		x	
44	RC6/TX	TX output to PC	TP1			

FIGURE 3-7 MCU pins and their uses.

If a new peripheral chip is added to the board in the proto area, Figure 3-7 will help to select MCU pins that will not conflict with an already dedicated external connection. MCU pins that are brought out to the H4 header pattern simplify the connections between the MCU and the peripheral chip.

Chapter

4

A FIRST TEMPLATE PROGRAM (T1.c)

4.1 OVERVIEW

This chapter introduces the first template program. It explains the code and its implications. It conclude with information on how to obtain, install, and run Microchip's free C18 compiler.

4.2 A T1.c TEMPLATE PROGRAM

The first program to be considered is listed in Figure 4-1. If the reader is not familiar with coding in C, it should be pointed out that multiple lines of comments can be inserted into a source file by bracketing the comments between

```
/*
```

and

```
*/
```

Alternatively, a double slash, //, tells the C compiler to ignore the remainder of a line. The first 10 lines of T1.c indicate what the program does. The Program Hierarchy consisting of the next few lines lists the functions making up the program, with indenting used to indicate that three functions are called from the main program. The function

```
/******* T1.c *****************
 *
 * Use Fosc = 4 MHz for Fcpu = Fosc/4 = 1 MHz.
 * Sleep for 16 ms (nominal), using watchdog timeout for wakeup.
 * Toggle RC2 output every 16 milliseconds for measuring looptime with scope.
 * Blink LED on RD4 for 16 ms every four seconds.
 * Check pushbutton and turn on LED continuously while it is pressed.
 *
 *       Current draw = 4 uA (with LED and LCD switched off)
 *
 ******* Program hierarchy *****
 *
 * main
 *  Initial
 *  BlinkAlive
 *  Pushbutton
 *
 ******************************
 */

#include <p18f4321.h>           // Define PIC18LF4321 registers and bits

/******************************
 * Configuration selections
 ******************************
 */

#pragma config OSC = INTIO1     // Use internal osc, RA6=Fosc/4, RA7=I/O
#pragma config PWRT = ON        // Enable power-up delay
#pragma config LVP = OFF        // Disable low-voltage programming
#pragma config WDT = OFF        // Disable watchdog timer initially
#pragma config WDTPS = 4        // 16 millisecond WDT timeout period, nominal
#pragma config MCLRE = ON       // Enable master clear pin
#pragma config PBADEN = DIG     // PORTB<4:0> = digital
#pragma config CCP2MX = RB3     // Connect CCP2 internally to RB3 pin
#pragma config BOR = SOFT       // Brown-out reset controlled by software
#pragma config BORV = 3         // Brown-out voltage set for 2.1V, nominal
#pragma config LPT1OSC = OFF    // Deselect low-power Timer1 oscillator

/******************************
 * Global variables
 ******************************
 */
unsigned int DELAY;             // Counter for obtaining a delay
unsigned char ALIVECNT;         // Counter for blinking "Alive" LED

/******************************
 * Constant strings
 ******************************
 */
```

FIGURE 4-1 T1.c template

```
/*******************************
 * Variable strings
 *******************************
 */

/*******************************
 * Function prototypes
 *******************************
 */

void Initial(void);
void BlinkAlive(void);
void Pushbutton(void);

/*******************************
 * Macros
 *******************************
 */
#define Delay(x) DELAY = x; while(--DELAY){ Nop(); Nop(); }

/////// Main program //////////////////////////////////////////////////////////

/*******************************
 * main
 *******************************
 */

void main()
{
   Initial();                    // Initialize everything
   while (1)
   {
      PORTCbits.RC2 = !PORTCbits.RC2;  // Toggle pin, for measuring loop time
      BlinkAlive();              // Blink "Alive" LED
      Pushbutton();              // Turn on LED while pushbutton is pressed
      Sleep();                   // Sleep, letting watchdog timer wake up chip
      Nop();
   }
}

/*******************************
 * Initial
 *
 * This function performs all initializations of variables and registers.
 *******************************
 */

void Initial()
{
   OSCCON = 0b01100010;          // Use Fosc = 4 MHz (Fcpu = 1 MHz)
```

FIGURE 4-1 *(continued)*

```
   ADCON1 = 0b00001011;          // RA0,RA1,RA2,RA3 pins analog; others digital
   TRISA = 0b00001111;           // Set I/O for PORTA
   TRISB = 0b01000100;           // Set I/O for PORTB
   TRISC = 0b10000000;           // Set I/O for PORTC
   TRISD = 0b10000100;           // Set I/O for PORTD
   TRISE = 0b00000010;           // Set I/O for PORTE
   PORTA = 0;                    // Set initial state for all outputs low
   PORTB = 0;
   PORTC = 0;
   PORTD = 0b00100000;           // except RD5 that drives LCD interrupt
   PORTE = 0;
   Delay(50000);                 // Pause for half a second
   RCONbits.SBOREN = 0;          // Now disable brown-out reset
   ALIVECNT = 247;               // Blink immediately
   WDTCONbits.SWDTEN = 1;        // Enable watchdog timer
}

/*******************************
 * BlinkAlive
 *
 * This function briefly blinks the LED every four seconds.
 * With a looptime of about 16 ms, count 250 looptimes
 *******************************
 */

void BlinkAlive()
{
   PORTDbits.RD4 = 0;            // Turn off LED
   if (++ALIVECNT == 250)        // Increment counter and return if not 250
   {
      ALIVECNT = 0;              // Reset ALIVECNT
      PORTDbits.RD4 = 1;         // Turn on LED for 16 ms every 4 secs
   }
}

/*******************************
 * Pushbutton
 *
 * This function overrides the role of the BlinkAlive function and turns on
 * the LED for the duration of a pushbutton press.
 *******************************
 */

void Pushbutton()
{
   PORTEbits.RE0 = 1;            // Power up the pushbutton
   Nop();                        // Delay one microsecond before checking it
   if (!PORTDbits.RD7)           // If pressed
   {
      PORTDbits.RD4 = 1;         // turn on LED
   }
   PORTEbits.RE0 = 0;            // Power down the pushbutton
}
```

FIGURE 4-1 *(continued)*

name *main* is used by C programs to indicate to the C compiler where program execution is to begin. The line

```
    #include <p18f4321.h>
```

is needed by the C compiler to assign addresses to register names like **OSCCON**, **TRISA**, and **WDTCON**. Also listed in the p18f4321.h file are the bit numbers associated with bit names like **RC2**, **SBOREN**, and **SWDTEN**. For example, **SWDTEN** is the name of bit 0, the least-significant bit of the **WDTCON** register. Because the compiler knows **SWDTEN** is the name of bit 0, it is not necessary for the user to know it. Knowing the names of bits and the registers in which they reside is sufficient.

4.3 CONFIGURATION SELECTIONS

Listed next are the configuration choices for the MCU. Actually, these choices have already been made when QwikBug was programmed into the chip. When T1.hex (the compiled version of T1.c) is downloaded to the Qwik&Low board, QwikBug ignores the configuration selections. They are shown here to indicate the configuration options under which the T1.c template program will run. And they indicate an essential part of the T1.c file were it to be programmed into the MCU with the PICkit 2 programmer instead of being downloaded by QwikBug.

Many of these configuration choices are described in Figure 4-2, which shows the choice selected in boldface. The two watchdog timer choices for **WDT** and **WDTPS** were described in Figure 2-6.

OSC	= **INTIO1**	Selects internal oscillator block; uses RA6 for Fosc/4 output; uses RA7 for I/O.
	= INTIO2	Selects internal oscillator block; RA6 and RA7 both available for I/O.
	= RCIO	Selects external RC oscillator on RA7; RA6 available for I/O.
	= LP	Uses RA6 and RA7 for 32768 Hz crystal oscillator.
	= XT	Uses RA6 and RA7 for 1-4 MHz crystal oscillator.
	= HS	Uses RA6 and RA7 for 4-25 MHz crystal oscillator.
	= HSPLL	Uses RA6 and RA7 with 10 MHz crystal and phase-locked loop for 40 MHz oscillator.
	= EC	Uses an external oscillator into RA7; uses RA6 for Fosc/4 output.
	= ECIO	Uses an external oscillator into RA7; uses RA6 for I/O.

(a) Oscillator power-up configuration

PWRT	= **ON**	Introduces a delay of about 66 ms after the chip detects that power has been turned on and before CPU clocking begins.
	= OFF	No 66 ms delay.

(b) Power-up timer

FIGURE 4-2 Configuration selections

CCP2MX	= **RB3**	CCP2 input/output is multiplexed with the RB3 pin.
	= RC1	CCP2 input/output is multiplexed with the RC1 pin.

(c) CCP2 configuration

LPT1OSC	= ON	Timer1 oscillator is configured for low-power, 32768 Hz operation.
	= **OFF**	Timer1 oscillator is configured for higher-power, higher frequency operation.

(d) Timer1's oscillator configuration

DEBUG	= **ON**	Background debugger is enabled - needed by QwikBug.
	= OFF	Background debugger is disabled; RB6 and RB7 available for I/O.

(e) Background debug mode use

LVP	= **OFF**	This choice is needed for normal, fast start from reset.
	= ON	This choice can cause a delay of several seconds coming out of reset.

(f) In-circuit serial programming (ICSP) option that permits programming voltage = V_{DD}

MCLRE	= **ON**	RE3/MCLR pin is an active dedicated active-low master reset input.
	= OFF	RE3/MCLR pin is a general purpose RE3 I/O pin.

(g) Optional reset pin

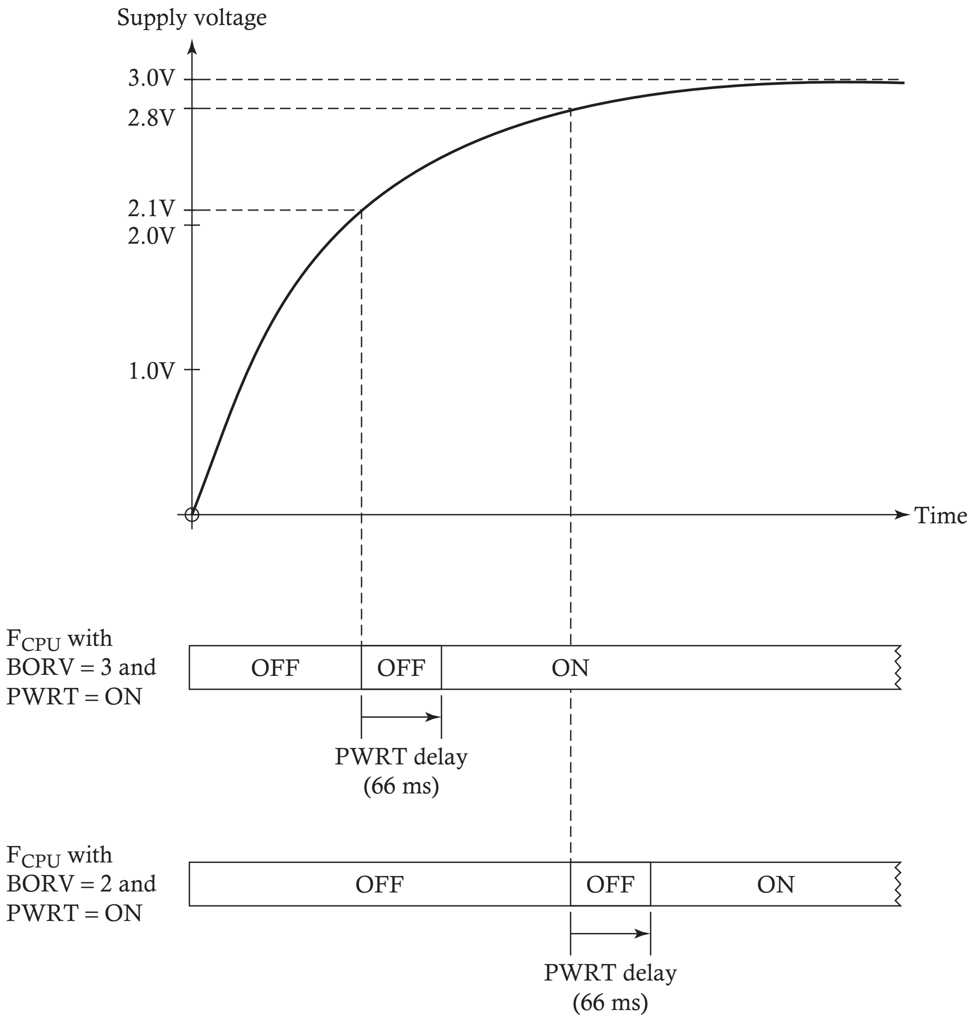
PBADEN	= **DIG**	PORTB bits 4,3,2,1,0 are configured as digital I/O pins at reset.
	= ANA	PORTB bits 4,3,2,1,0 are configured as analog input pins at reset.

(h) PORTB reset configuration

FIGURE 4-2 *(continued)*

The **OSC** choice of **INTIO1** selects the primary, power-on reset, oscillator for the chip. This choice can be overridden at any time by the user program by changing the content of the **OSCCON** register (see Figure 2-3).

The brownout-reset options are described in Figure 4-3. The original intent of a brownout reset was to stop the clocking of the CPU when V_{DD} drops below a specified threshold level, as when a power switch is opened. Here, the brownout-reset mechanism is used at startup, to hold the chip in the reset state until sometime after the power switch connects the coin cell to the V_{DD} line supplying the MCU. With **BORV** = 3 and with the power-up timer enabled with **PWRT** = ON, clocking of the CPU begins about 66 ms after V_{DD} rises above about 2.1 V. Should the power



(a) Power-on behavior of brownout reset

BOR	= **SOFT**	Brownout reset controlled by **SBOREN** bit in **RCON** register and enabled at reset
	= OFF	Brownout reset disabled (0 μA current draw)
	= ON	Brownout reset enabled (≈34 μA current draw)
	= NOSLP	Brownout reset in run and idle modes; disabled in sleep mode

(b) Brownout-reset configuration

SBOREN	= **1**	Brownout feature enabled
	= 0	Brownout feature disabled

(c) Software control of the brownout-reset feature when BOR = SOFT (**SBOREN** is bit 6 of the **RCON** register)

FIGURE 4-3 Brownout-reset options

switch exhibit contact bounce, a reliable startup will ensue, even after one or more false starts. With **BOR** = SOFT, the brownout feature can be disabled after startup by clearing the **SBOREN** bit in the **RCON** register to eliminate its current draw of about 34 μA.

4.4 GLOBAL VARIABLES

The *Global variables* section of T1.c assigns two variables to the program, both as unsigned numbers. The unsigned *int* variable, **DELAY**, ranges from 0 to 65,535. The unsigned *char* variable, **ALIVECNT**, ranges from 0 to 255.

Sophisticated C code writers may note that the **DELAY** variable is used only within the **Initial** function. Once initialized, **ALIVECNT** is used only within the **BlinkAlive** function. In both cases, the variable could have been defined to be *local* to the function within which it is used. However, because the definition of local variables produces extra machine code and extra execution time by Microchip's C18 compiler, only global variables will be used throughout this book.

4.5 BIT MANIPULATIONS

As a programming language, C offers no direct support for defining a *bit* type or for testing or modifying 1 bit of a register or variable. Microchip's C18 compiler alleviates this deficiency in the case of registers. Thus

```
WDTCONbits.SWDTEN = 1;
```

will set the **SWDTEN** bit in the **WDTCON** register. For testing or manipulating a bit of a variable, the C18 compiler does not provide the same support. Thus

```
ALIVECNTbits.7 = 0;
```

will generate a compiler error rather than generating code that will clear bit 7 of the RAM variable, **ALIVECNT**.

When writing code for a microcontroller, a commonly recurring need arises for *flag* bits that can be set, cleared, and tested. Because the PIC18LF4321 has 512 bytes of RAM available, dedicating some of these to serve as two-valued flags is not unreasonable. Thus, in the template program of the next chapter, a *char* (8-bit) variable named **PBFLAG** is introduced to distinguish between operation *before* the pushbutton is first pressed and subsequently. Before the pushbutton is first pressed, **PBFLAG** is cleared to zero with the line

```
PBFLAG = 0;
```

and the display shows the message

PRESS PB

After the pushbutton is pressed, **PBFLAG** is set to one with

```
PBFLAG = 1;
```

and the display switches to its ongoing program use. The flag is tested with

```
if (!PBFLAG)
{
   <do these tasks before pushbutton is first pressed>
}
```

or with

```
if (PBFLAG)
{
   <do these tasks if pushbutton has already been pressed>
}
```

The pushbutton can be powered up, as shown in Figure 3-2, by setting **RE0**. Then **RD7** can be read to set **NEWPB** if **RD7** is low (i.e., if the pushbutton is pressed)

```
NEWPB = !PORTDbits.RD7;
```

This flag is compared with the value of **NEWPB** found some time earlier and saved in **OLDPB** with

```
if (!OLDPB && NEWPB)// Look for last time = 0, now = 1
{
   <do these tasks if pushbutton is newly pressed>
}
```

4.6 FUNCTION PROTOTYPES

Each function, other than **main**, must be listed in the *Function prototypes* section, to indicate the type (e.g., char) of any parameters to be passed to the function other than the global variables, and the type of any parameter to be returned by the function. Throughout this book all parameters will be passed to a function as global variables. Furthermore, within a function, *local* variables will be avoided. The reasons for these decisions are:

- Parameters passed in the call of the function add significantly to both the resulting function code and its execution time. The latter issue is a major theme of this book because an increase in execution time translates into a proportional increase in average current draw.
- Local variables do the same, increasing both the amount of code and the execution time. Furthermore, only global variables can serve as watch variables for QwikBug. Thus, this decision fosters the debugging of new program code.

4.7 A CALIBRATED DELAY MACRO

The *Macros* section of T1.c includes a single macro definition:

```
#define Delay(x) DELAY = x; while(--DELAY){ Nop(); Nop(); }
```

This tells the C18 compiler that, when it subsequently sees the character sequence:

```
Delay(50000)
```

it should make the substitution:

```
DELAY = 50000; while(--DELAY){ Nop(); Nop(); }
```

The compiler will generate code that, when executed, will load 50,000 into the *unsigned int* variable, **DELAY**, then decrement **DELAY**. If the decremented value of **DELAY** equals zero, then the execution of the macro is done. Otherwise two Nop() macros are executed before **DELAY** is decremented again. Each Nop() macro is compiled to a "no operation" assembly language instruction.

What is interesting here is the relationship between the parameter value and the duration of the resulting delay. The insertion of the line

`Delay(`*`value`*`);` **(4-1a)**

will create a delay in the program execution of *exactly*

$$\text{delay} = (10 \times \textit{value}) \text{ clock periods for } \textit{value} < 256$$

or

$$\text{delay} = (10 \times \textit{value}) + 1 \text{ clock periods for } \textit{value} \geq 256$$

Ignoring the additional clock period for *value* $\geq$ 256 and the 2% accuracy of the internal clock, this macro can be used to generate a calibrated delay. With F_{OSC} = 4 MHz, the CPU clock period equals one microsecond and the delay will equal

$$\text{delay} = 10 \times \textit{value} \text{ microseconds} \qquad \textbf{(4-1b)}$$

given the global variable declaration

`unsigned int DELAY;` **(4-1c)**

In general, it is difficult to predict how the C18 compiler will optimize the code of a program. Alex Singh discovered that without the inclusion of any assembly language code within the Delay() macro, it would be compiled in three different ways in different source files (optimized for speed of execution, optimized for minimal code generation, or not optimized at all). However, with the inclusion of any assembly code in a macro definition, the macro is always compiled to the same machine code.

4.8 MAIN FUNCTION

The **main** function begins with a call of **Initial**. Then the **main** function enters an infinite loop in which it toggles a pin that can be probed on one of the Qwik&Low board's H4 header pins. With the help of a scope, the time the CPU takes to traverse the loop can be measured as the time from a rising edge of the **RC2** pin to the next falling edge. The main program then calls **BlinkAlive** and **Pushbutton** in succession before executing the **Sleep** macro. Note that the C compiler identifies the **RC2** bit within the **PORTC** register as **PORTCbits.RC2** and toggles it with

```
PORTCbits.RC2 ^= 1;
```

4.9 8-BIT AND 16-BIT REGISTERS

The role of the **Initial** function is to initialize registers, control and status bits, and variables. Most of the PIC18LF4321 registers are 8 bits long. The few that are 16 bits long generally carry two names. For example, the 10-bit output of the analog-to-digital converter can be *right justified* into the 16-bit register, **ADRES**, and treated as an unsigned int variable ranging from 0 to 1,023. On the other hand, it is sometimes useful to use the analog-to-digital converter as an 8-bit converter. Its output can be *left justified* into **ADRES**. The upper 8 bits, accessed as **ADRESH**, range from 0 to 255. The least-significant 2 bits of the 10-bit conversion reside in the upper 2 bits of **ADRESL** and are ignored.

Throughout this book, as a multiple-function hardware module of the PIC18LF4321 chip is discussed, it will be dealt with one function at a time. All of the registers, control bits, and status bits associated with that function will be described. Then the C code to make use of that function will reduce to interactions with those registers and bits.

4.10 CLOCK RATE CHOICE

Referring back to Figure 2-3, it can be seen that the first line of the **Initial** function of Figure 4-1

```
OSCCON = 0b01100010;
```

selects F_{OSC} = 4 MHz and a CPU clock rate of F_{CPU} = 1 MHz. Since most instructions are executed in one CPU clock period, this means that a sequence like

```
PORTBbits.RB0 = 1;
PORTBbits.RB0 = 0;
```

will generate a 1-μs positive pulse on the **RB0** pin.

Sometimes a short pause is required between the activation of a process and the reading of the output of the process. Inserting the macro

```
Nop();
```

can be used to insert a pause of 1 μs in the execution of the code as the CPU executes a single-cycle "no operation" machine instruction.

The decision to select F_{OSC} = 4 MHz rather than the higher value of 8 MHz is driven largely by the data of Figure 2-4. This illustrates that while the MCU is awake and running with F_{OSC} = 8 MHz, it draws 1.750 mA, a heavy current for the coin cell while undebugged code leaves the chip constantly awake. The choice of F_{OSC} = 4 MHz drops this steady current draw of a malfunctioning program to a milliampere, considerably better. For intermittent sleep mode operation with F_{OSC} = 4 MHz, an application suffers an average current draw penalty of

$$\frac{6.6 - 5.6}{5.6} \times 100 = 18\%$$

relative to the average current draw with F_{OSC} = 8 MHz.

The choice of F_{OSC} = 4 MHz versus an even slower clock rate is driven by Figure 2-8, given that the applications discussed in this book will usually operate in the intermittent sleep mode. When such is not the case, the slow INTRC internal oscillator of Figure 2-3 or the slow Timer1 crystal oscillator of Figure 3-3 will present an excellent alternative for applications that can deal with the slow execution of a CPU clock that executes only about eight machine instructions every millisecond.

4.11 ANALOG PINS VERSUS DIGITAL I/O PINS

The initialization of **ADCON1** in general selects which of the chip's possible 13 inputs to the analog-to-digital converter will be used as analog inputs and which will be used as digital I/O pins. Given the Qwik&Low I/O connections of Figure 3-2, the choice used in the T1.c code is to select the four ADC pins

AN0, AN1, AN2, and VREF+/AN3

Adding one or two additional analog input channels will be discussed in Chapter Nine.

4.12 DIGITAL INPUTS VERSUS OUTPUTS

Each digital I/O pin used by the Qwik&Low board must be properly configured as either an input or an output, whether or not it is used by the code of T1.c. These pins are shown in Figure 3-2. Input/output configuring is carried out by setting (input) or clearing (output) the **TRIS** register bits. MCU pins not connected to anything on the Qwik&Low board should be made outputs. Thus the initialization

```
TRISD = 0b10000100;
```

sets up bits 7 and 2 of **PORTD** as inputs and bits 6, 5, 4, 3, 1, and 0 as outputs. The pins that are unused by the board are indicated as such in Figure 3-7. These are **RD0** and **RD1**, also set up as outputs.

All of the **PORTD** output port pins are initialized to zero except for **RD5** that is set to one via the line

```
PORTD = 0b00100000;
```

A $1 \rightarrow 0$ transition from this output pin will be used to wake up the LCD controller. For now it is left to idle high.

4.13 BROWNOUT MODULE DISABLING

After initializing the oscillator and the states of the I/O pins, the **Initial** subroutine uses the **Delay** macro to wait half a second before continuing. During this time, the brownout reset mechanism will have resolved any powering-up issues and the LCD controller will have had time to initialize itself. At the completion of the delay, the brownout reset module is shut down, to eliminate a current draw of about 34 µA on the coin cell. The user program variables (**ALIVECNT** in this case) are initialized, and the watchdog timer of Figure 2-6 is started counting (from zero).

4.14 MAIN LOOP

Upon returning from the **Initial** function, the **main** function toggles the **RC2** output pin. As shown in Figure 4-4, a scope can probe this pin (labeled **RC2/CCP1** on the H4 strip) to verify that the watchdog timer's timeout period is close to 16 ms, the time selected by the **WDTPS** = 4 configuration choice (see Figure 2-6). For slow events, the resulting *loop times* can be counted to derive the event timing. Thus the **BlinkAlive**

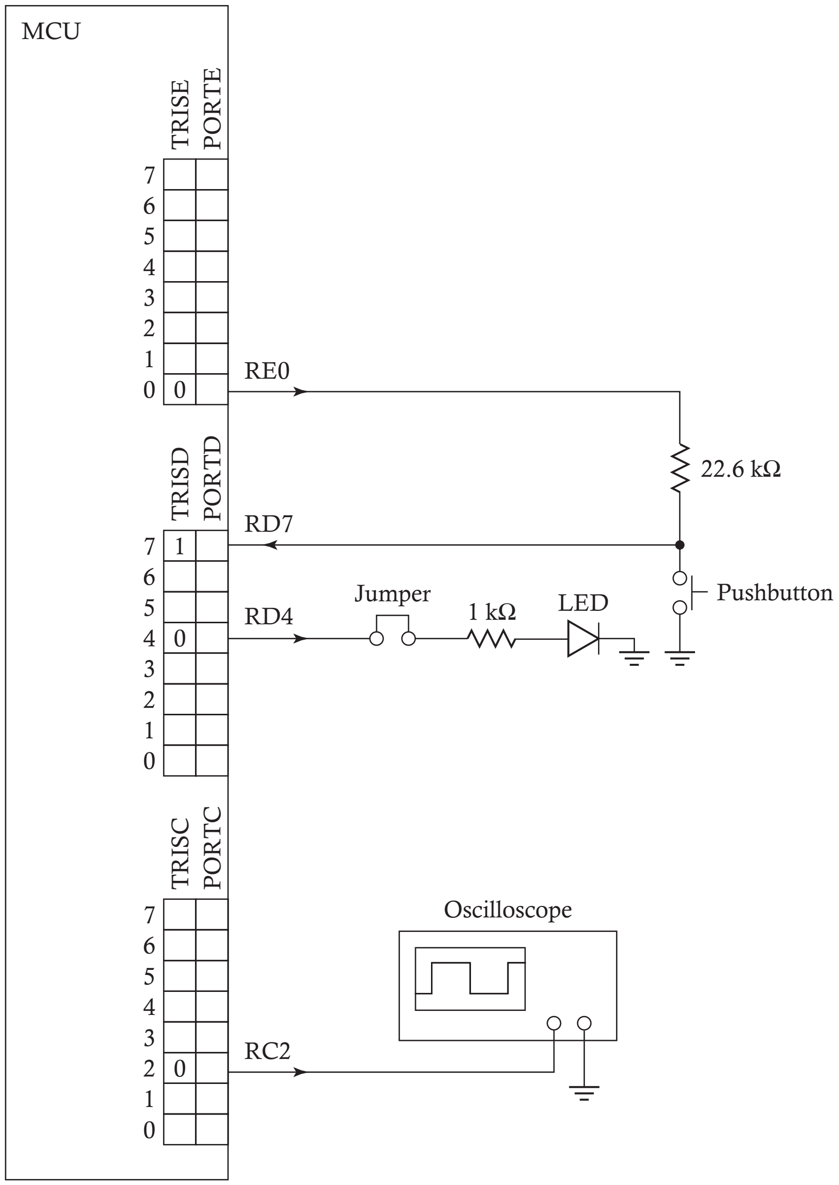


FIGURE 4-4 T1.c input/output connections

function is called during each pass around the main loop. Each call occurs approximately 16 ms after the previous one, so every

$$250 \times 16 \text{ ms} = 4{,}000 \text{ ms} = 4 \text{ s}$$

the **ALIVECNT** variable will have been incremented to 250, reset to zero, and the LED driven from **RD4** will be turned on. It remains on until 16 ms later when **BlinkAlive** is again called and **RD4** is cleared with the opening statement of **BlinkAlive**,

```
PORTDbits.RD4 = 0;
```

Note this use by the C compiler of the term **PORTDbits** rather than the term **PORTD** when accessing a specific bit (**RD4**) in a register (**PORTD**). The **main** function closes with a **Sleep** macro that is translated by the C compiler into the PIC18LF4321's "sleep" instruction. Upon awakening from sleep, the CPU may not carry out the operation of the next instruction correctly. By having that next instruction be a "nop" instruction, no *intended* operation is passed over. Consequently, a **Sleep** macro should always be followed by a **Nop** macro (translated to the chip's "no operation" one cycle instruction).

4.15 COMPILATION

Set up a folder

C:\WORK

to hold source files (e.g., T1.c) as well as the files generated as a result of the compilation of source files. In addition, it is useful to have a desktop icon that opens into this folder and another desktop icon that opens a DOS window into this folder.

For Windows XP, the www.qwikandlow.com website has a batch file

MakeWork.bat

and two desktop short cuts

Work
DOS for work

that can be downloaded to the reader's desktop. The batch file creates a new folder

C:\Work

Clicking on the **Work** desktop icon opens the C:\Work folder. Clicking on the **DOS for Work** desktop icon opens a DOS window with a

C:\Work>

prompt.

Download from www.qwikandlow.com into the new C:\Work folder the batch file

C18.exe

and the source file

T1.c

Finally, install the student version of Microchip's C18 compiler including their path-list settings. This can be found on the Microchip website by Googling

+"MPLAB C18 compiler" +"Student Edition"

To try compilation, click on the **DOS for Work** icon. Then after the C:\Work> prompt, type

```
C18  T1
```

To edit the T1.c file, any text editor can be used. The Crimson editor is a popular and free one, available from www.crimsoneditor.com. It understands C and it flags syntax errors.

PROBLEMS

4-1 Faster blinking Modify the T1.c file into T1faster.c so as to blink the LED every second. Recompile, download, and run the result.

4-2 Pushbutton modification Form a T1pb.c file in which the **BlinkAlive** function and its call are removed from the file. Modify the **Pushbutton** function so that it blinks on for only 16 ms in response to each pushbutton press.

4-3 Another pushbutton modification Form a T1pb2.c file. In response to each pushbutton press, blink the LED twice. Each blink should last for one loop time (i.e., about 16 ms). The duration between blinks should be 32 loop times (i.e., about 0.5 second). Make sure that the MCU sleeps between loop times.

4-4 Measurements For each of the above programs, make two measurements.

a) Measure the current draw with the LED jumper removed. Is there any measurable difference between the current draw for these programs?

b) Probe the MCU's CPU clock, $F_{OSC}/4$ at test point TP6. Referring to Figure 2-5, measure both T_{period} and the maximum value in each case of T_{active}. How do these compare between the programs?

4-5 Oscillator Control For this project, you will carry out the eight **INTOSC** clock source tests of Figure 2-8. However, instead of executing the 100 clock periods called for there, just execute enough code to switch the oscillator frequency to the next value in response to a pushbutton press. Initialize **OSCCON** to 0b01110010 and **OSCTUNE** to 0b10000000. This will produce the conditions for measuring P_{CPU}, T_{exec}, and I_{avg} for the first row of the table of Figure 2-8. Each of the seven pushes of the pushbutton will yield the conditions for the remaining rows (ignoring the row for the **INTRC** clock source). To change **OSCCON** just once for each pushbutton press, define and use the two flag variables **NEWPB** and **OLDPB** discussed at the end of Section 4.5.

Chapter

5

SPI BUS AND THE LCD (T2.c)

5.1 OVERVIEW

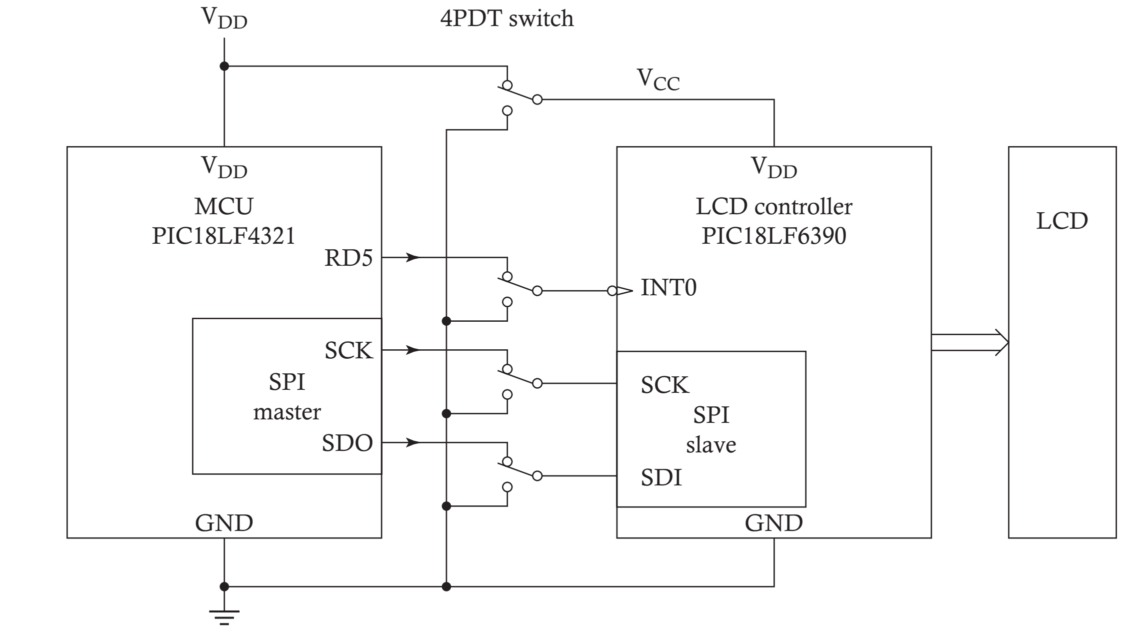
This chapter is based on the development of the firmware for the LCD controller by Alex Singh. He has developed an elegant implementation of the LCD controller specification of this chapter.

The chapter begins with an explanation of the MCU's and the LCD controller's *Serial Peripheral Interface* (SPI) and how it is used for the fast serial transfer of *display strings* to update the LCD. A template program, T2.c, introduces a **Display** function for sending a variable string to the display.

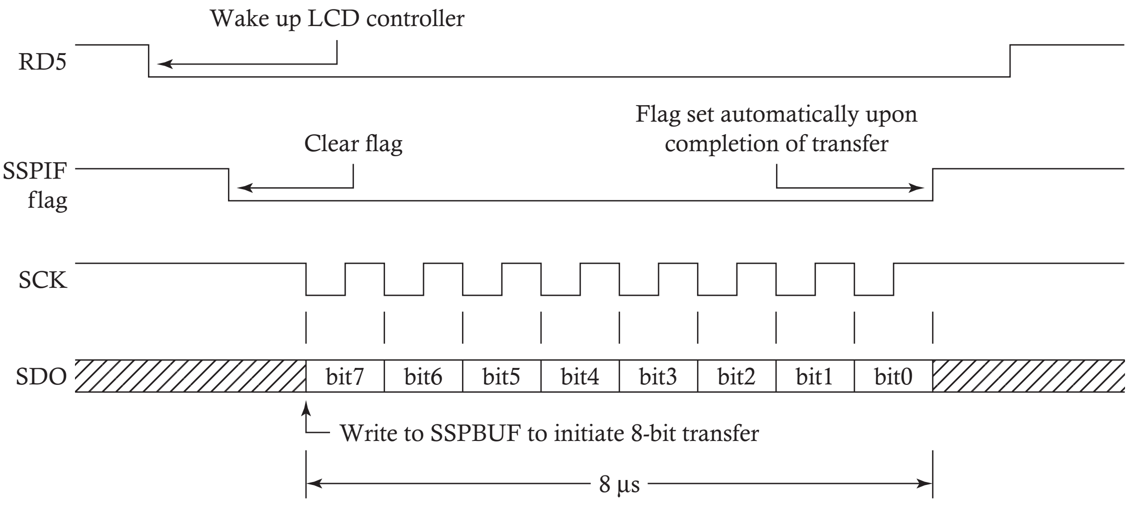
5.2 SERIAL PERIPHERAL INTERFACE

The PIC18LF4321 MCU and the PIC18LF6390 LCD controller each use their SPI for the communication of display messages from the MCU to the LCD controller, as shown in Figure 5-1a. The SPI bus is a fast serial interface. In response to writing a byte to the MCU's **SSPBUF** register, the 8 bits are shifted out of its **SDO** (serial data out) pin, synchronized to eight clock pulses on its **SCK** (serial clock) pin, as shown in Figure 5-1b.

The MCU signals the LCD controller to wake up with a one to zero falling edge from its **RD5** output pin to the LCD controller's **INT0** interrupt input pin. Upon reception of this falling edge, the LCD controller's CPU awakens to receive a string of characters from the MCU, interpret them into their 14-segment "starburst" representation, and load the results into LCD data registers before returning to sleep. While the LCD controller's CPU sleeps, its LCD module refreshes the LCD display at a 37 Hz refresh

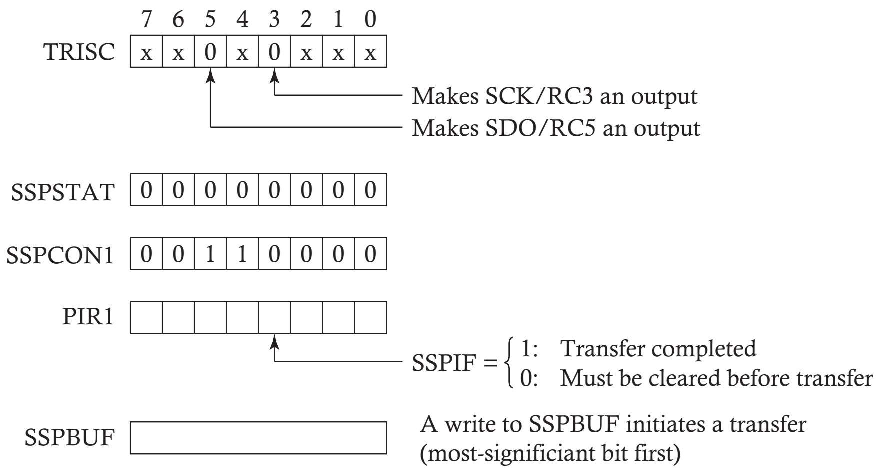


(a) Connections between MCU and LCD controller


Wake up LCD controller
RD5
Clear flag
Flag set automatically upon completion of transfer
SSPIF flag
SCK
SDO
bit7
bit6
bit5
bit4
bit3
bit2
bit1
bit0
Write to SSPBUF to initiate 8-bit transfer
8 µs

(b) Waveforms

FIGURE 5-1 MCU's SPI use for LCD display



(c) MCU's SPI registers

FIGURE 5-1 *(continued)*

rate. It is this combination of sleeping CPU, very slow refresh rate, and low capacitance loading by the LCD pins that produces the small 5-μA current draw of the LCD and its controller when it is not being updated by a display string from the MCU.

Figure 5-1b also illustrates the role of the MCU's **SSPIF** flag that is set upon the completion of the 1-byte transfer. To send a sequence of bytes, the **SSPIF** flag is cleared, the first byte of the sequence is written to the **SSPBUF** register, and program execution waits until the **SSPIF** flag is set before clearing **SSPIF** and writing the next byte to **SSPBUF**.

With each 1-byte transfer taking just the 8 μs dictated by an **SCK** clock output that consists of eight pulses of the MCU's $F_{OSC}/4$ CPU clock, this interface helps to minimize the awake time of both the MCU when it deals with the display and the LCD controller when it awakens to receive an update.

The MCU's SPI registers and their initialization to produce the waveform of Figure 5-1b are illustrated in Figure 5-1c. The SPI has many options:

- Whether the SPI module drives **SCK** (master mode) or uses **SCK** as a clock input (slave mode).
- Whether the **SCK** pin idles high (as in Figure 5-1b) or low.
- Whether it uses its fastest clock rate of $F_{osc}/4$ or a slower rate.

For the connection of Figure 5-1a to function properly, it is important for the MCU's SPI to be set up as master and the LCD controller's SPI to be set up as slave. It is also important for the MCU and the LCD controller to agree on the polarity of the **SCK** pulses. With its use of F_{OSC} = 8 MHz, the LCD controller can accept SPI inputs at any of the MCU's SPI clock rates. The initialization of **SSPSTAT** and **SSPCON1** shown in Figure 5-1c produces the waveforms of Figure 5-1b and produces the idle-high **SCK** that the LCD controller expects.

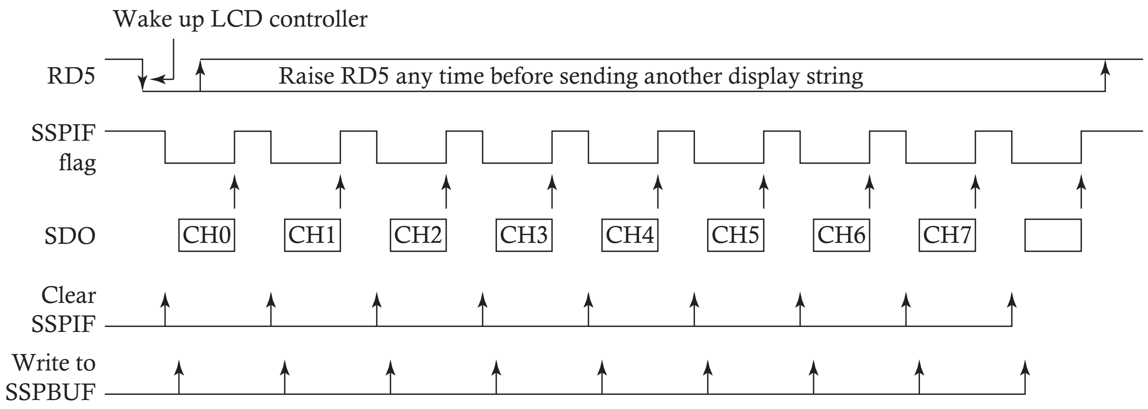
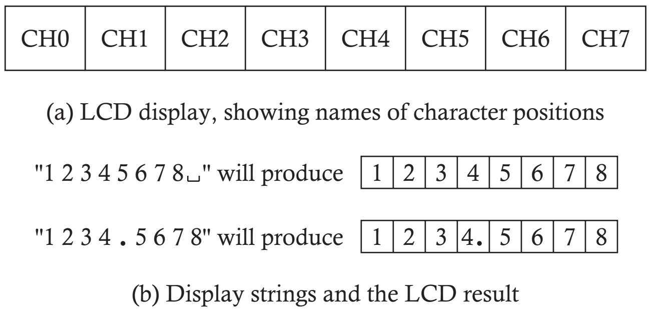


FIGURE 5-2 Display string format consisting of nine bytes (eight ASCII-coded characters plus an optional ASCII-coded decimal point)

5.3 DISPLAY STRINGS

The sequence of operations needed to update the entire display's eight characters plus an optional decimal point via a new message string is shown in Figure 5-2. The sequence begins with the dropping of the **RD5** pin from one to zero. Since the LCD controller only reacts to this falling edge, it is unimportant when this pin is raised again. It is only necessary that it be high again when a subsequent message string is ready to be sent.

Before the first byte is written to **SSPBUF**, the **SSPIF** flag is cleared. Before each subsequent byte is written to **SSPBUF**, the CPU waits for the automatic setting of the **SSPIF** flag at the completion of the 1-byte transfer before clearing the flag and writing the next byte. After receiving the 9 bytes, the LCD controller interprets and displays the bytes, and then returns to sleep. With the character positions named as in Figure 5-3a, the characters in a display string are arranged in the same order as shown in Figure 5-3b. Thus the first character sent will appear in the leftmost character position, with subsequent characters appearing in order to the right of this position. If the 9 bytes include a decimal point, the decimal point is displayed with the character that precedes it. If no decimal point is included in the string, the ninth byte received is ignored.


CH0
CH1
CH2
CH3
CH4
CH5
CH6
CH7
"1 2 3 4 5 6 7 8␣" will produce
1 2 3 4 5 6 7 8
"1 2 3 4 . 5 6 7 8" will produce
1 2 3 4. 5 6 7 8

(a) LCD display, showing names of character positions

(b) Display strings and the LCD result

FIGURE 5-3 Display string positioning of characters

5.4 DISPLAYABLE CHARACTERS

Any byte that is received by the LCD controller will be interpreted as:

- A displayable character (see Figure 5-4a)
- A reinterpreted character (see Figure 5-4b)

ASCII code does not include the degree symbol for units of temperature. The ASCII code for a question mark (0x3F), if received by the LCD controller, will be reinterpreted to display the degree symbol.

If the LCD controller receives any lower-case letter, it is reinterpreted as the corresponding upper-case character. If any unrecognized codes are received, the LCD controller will turn on all segments of that character position to alert the user of a faulty choice, given the limitations of a starburst character representation.

5.5 DECIMAL POINT

Because the LCD includes an optional decimal point with each character position, the LCD controller treats the reception of the ASCII code for a decimal point as a special case. For example, the nine-character display string

```
"1234.5678"
```

will show up as

1234.5678

Numbers:	0 1 2 3 4 5 6 7 8 9
Upper-case letters:	A B C ... X Y Z
Recognized symbols:	( ) ' . + – * / < > ^

(a) Characters displayed in response to their ASCII codes.

Character sent	**is reinterpreted as**
?	° (degree symbol for temperature)
a b c ... x y z	A B C ... X Y Z

(b) Reinterpreted characters

Turn on all segments for that character position

(c) Unaccounted-for 8-bit codes.

FIGURE 5-4 Displayable characters

with both the 4 and the decimal point in the CHAR3 position.

The LCD display on the Qwik&Low board has eight decimal points, one on the right side of each character position. Consequently the display string

```
".12345678"
```

will be displayed as

.1234567

with the last character (8) ignored. A more readable result will occur by sending

```
"0.1234567"
```

to produce the following display

0.1234567

5.6 T2.C, A DISPLAY TEMPLATE

The template program of Figure 5-5 illustrates how to deal with the LCD display.

The template program also illustrates several new considerations arising because of interactions with a second microcontroller.

```
/******* T2.c ******************
 *
 * Use Fosc = 4 MHz for Fcpu = Fosc/4 = 1 MHz.
 * Sleep for 16 ms (nominal), using watchdog timeout for wakeup.
 * Toggle RC2 output every 16 milliseconds for measuring looptime with scope.
 * Blink LED on RD4 for 16 ms every four seconds.
 * Post PRESS PB message on LCD until first pushbutton push.
 * Increment LCD's CHAR0:CHAR1 every second.
 * Increment LCD's CHAR3:CHAR4 for each pushbutton press.
 *
 *       Current draw = 7 uA (with LED and LCD switched off)
 *
 ******* Program hierarchy *****
 *
 * main
 *  Initial
 *    Display
 *  BlinkAlive
 *  Time
 *  Pushbutton
 *  UpdateLCD
 *    Display
 *
 *******************************
 */
```

FIGURE 5-5 T2.c template

```
#include <p18f4321.h>           // Define PIC18LF4321 registers and bits
#include <string.h>             // Used by the LoadLCDSTRING macro

/*******************************
 * Configuration selections
 *******************************
 */

#pragma config OSC = INTIO1     // Use internal osc, RA6=Fosc/4, RA7=I/O
#pragma config PWRT = ON        // Enable power-up delay
#pragma config LVP = OFF        // Disable low-voltage programming
#pragma config WDT = OFF        // Disable watchdog timer initially
#pragma config WDTPS = 4        // 16 millisecond WDT timeout period, nominal
#pragma config MCLRE = ON       // Enable master clear pin
#pragma config PBADEN = DIG     // PORTB<4:0> = digital
#pragma config CCP2MX = RB3     // Connect CCP2 internally to RB3 pin
#pragma config BOR = SOFT       // Brown-out reset controlled by software
#pragma config BORV = 3         // Brown-out voltage set for 2.1V, nominal
#pragma config LPT1OSC = OFF    // Deselect low-power Timer1 oscillator

/*******************************
 * Global variables
 *******************************
 */
char PBFLAG;                    // Flag, set after first press of pushbutton
char LCDFLAG;                   // Flag, set to send string to display
char NEWPB;                     // Flag, set if pushbutton is now pressed
char OLDPB;                     // Flag, set if pushbutton was pressed last loop
unsigned char ALIVECNT;         // Scale-of-248 counter for blinking "Alive" LED
unsigned char TIMECNT;          // Scale-of-62 counter of loop times = 1 second
unsigned char ONES;             // For display of seconds
unsigned char TENS;
unsigned char PBONES;           // For display of pushbutton count
unsigned char PBTENS;
unsigned char i;                // Index into strings
unsigned int DELAY;             // Sixteen-bit counter for obtaining a delay
char LCDSTRING[] = "PRESS PB "; // LCD display string

/*******************************
 * Function prototypes
 *******************************
 */

void Initial(void);
void BlinkAlive(void);
void Pushbutton(void);
void Time(void);
void UpdateLCD(void);
void Display(void);

/*******************************
 * Macros
 *******************************
 */
```

FIGURE 5-5 *(continued)*

```
#define Delay(x) DELAY = x; while(--DELAY){ Nop(); Nop(); }
#define LoadLCDSTRING(lit)  strcpypgm2ram(LCDSTRING,(const far rom char*)lit)
/////// Main program /////////////////////////////////////////////////////////
/*******************************
 * main
 *******************************
 */
void main()
{
  Initial();                    // Initialize everything
  while (1)
  {
    PORTCbits.RC2 ^= 1;         // Toggle pin, for measuring loop time
    BlinkAlive();               // Blink "Alive" LED
    Time();                     // Display seconds
    Pushbutton();               // Display pushbutton count
    UpdateLCD();                // Update LCD
    Sleep();                    // Sleep, letting watchdog timer wake up chip
    Nop();
  }
}
/*******************************
 * Initial
 *
 * This function performs all initializations of variables and registers.
 *******************************
 */
void Initial()
{
  OSCCON = 0b01100010;          // Use Fosc = 4 MHz (Fcpu = 1 MHz)
  SSPSTAT = 0b00000000;         // Set up SPI for output to LCD
  SSPCON1 = 0b00110000;
  ADCON1 = 0b00001011;          // RA0,RA1,RA2,RA3 pins analog; others digital
  TRISA = 0b00001111;           // Set I/O for PORTA
  TRISB = 0b01000100;           // Set I/O for PORTB
  TRISC = 0b10000000;           // Set I/O for PORTC
  TRISD = 0b10000000;           // Set I/O for PORTD
  TRISE = 0b00000010;           // Set I/O for PORTE
  PORTA = 0;                    // Set initial state for all outputs low
  PORTB = 0;
  PORTC = 0;
  PORTD = 0b00100000;           // except RD5 that drives LCD interrupt
  PORTE = 0;
  SSPBUF = ' ';                 // Send a blank to initialize state of UART
  Delay(50000);                 // Pause for half a second
  RCONbits.SBOREN = 0;          // Now disable brown-out reset
  PBFLAG = 0;                   // Clear flag until pushbutton is first pressed
  LCDFLAG = 0;                  // Flag to signal LCD update is initially off
  TIMECNT = 0;                  // Reset TIMECNT
  TENS = '5';                   // Initialize to 59 so first display = 00
  ONES = '9';
```

FIGURE 5-5 *(continued)*

```
  PBTENS = '0';                    // Initialize count of pushbutton presses
  PBONES = '1';
  OLDPB = 0;                       // Initialize to unpressed pushbutton state
  ALIVECNT = 247;                  // Blink immediately
  WDTCONbits.SWDTEN = 1;           // Enable watchdog timer
  Display();                       // Display initial "PRESS PB" message
  LoadLCDSTRING("00 01    ");      // Reinitialize LCDSTRING
}

/*******************************
 * BlinkAlive
 *
 * This function briefly blinks the LED every four seconds.
 * With a looptime of about 16 ms, count 4x62 = 248 looptimes
 *******************************
 */
void BlinkAlive()
{
  PORTDbits.RD4 = 0;               // Turn off LED
  if (++ALIVECNT == 248)           // Increment counter and return if not 248
  {
    ALIVECNT = 0;                  // Reset ALIVECNT
    PORTDbits.RD4 = 1;             // Turn on LED for 16 ms every 4 secs
  }
}
/*******************************
 * Time
 *
 * After pushbutton is first pushed, display seconds.
 *******************************
 */
void Time()
{
  if (PBFLAG)                      // After pushbutton is first pushed,
  {
    if (++TIMECNT == 62)           // count TIMECNT to 1 second
    {
      TIMECNT = 0;                 // Reset TIMECNT for next second
      if (++ONES > '9')            // and increment time
      {
        ONES = '0';
        if (++TENS > '5')
        {
          TENS = '0';
        }
      }
      LCDSTRING[0] = TENS;         // Update display string
      LCDSTRING[1] = ONES;
      LCDFLAG = 1;                 // Set flag to display
    }
  }
}
```

FIGURE 5-5 *(continued)*

```
/*******************************
 * Pushbutton
 *
 * After pushbutton is first pressed, display pushbutton count.
 *******************************
 */
void Pushbutton()
{
  PORTEbits.RE0 = 1;            // Power up the pushbutton
  Nop();                        // Delay one microsecond before checking it
  NEWPB = !PORTDbits.RD7;       // Set flag if pushbutton is pressed
  PORTEbits.RE0 = 0;            // Power down the pushbutton
  if (!OLDPB && NEWPB)          // Look for last time = 0, now = 1
  {
    if (!PBFLAG)                // Take action for very first PB press
    {
      PBFLAG = 1;
      ALIVECNT = 0;             // Synchronize LED blinking to counting
      TIMECNT = 61;             // Update display immediately
    }
    else                        // Take action for subsequent PB presses
    {
      if (++PBONES > '9')       // and increment count of PB presses
      {
        PBONES = '0';
        if (++PBTENS > '9')
        {
          PBTENS = '0';
        }
      }
    }
    LCDSTRING[3] = PBTENS;      // Update display string for simulated LCD
    LCDSTRING[4] = PBONES;
    LCDFLAG = 1;                // Set flag to display
  }
  OLDPB = NEWPB;                // Save present pushbutton state
}
/*******************************
 * UpdateLCD
 *
 * This function updates the 8-character LCD once the pushbutton has
 * first been pressed, if Time or Pushbutton has set LCDFLAG.
 *******************************
 */
void UpdateLCD()
{
  if(PBFLAG && LCDFLAG)
  {
    Display();
    LCDFLAG = 0;
  }
}
```

FIGURE 5-5 *(continued)*

```
/******************************
 * Display()
 *
 * This function sends LCDSTRING to the LCD.
 ******************************
 */
void Display()
{
  PORTDbits.RD5 = 0;            // Wake up LCD display
  for (i = 0; i <= 8; i++)
  {
    PIR1bits.SSPIF = 0;         // Clear SPI flag
    SSPBUF = LCDSTRING[i];      // Send byte
    while (!PIR1bits.SSPIF);    // Wait for transmission to complete
  }
  PORTDbits.RD5 = 1;            // Return RB5 high, ready for next string
}
```

FIGURE 5-5 *(continued)*

5.7 INITIALIZATION OF TWO MICROCONTROLLERS

The PIC18LF4321 MCU and the PIC18LF6390 LCD controller each has its own power-on reset circuit. Each one has to deal with contact bounce in the power switch. Also it is important for the LCD controller to have initialized itself before the first display string is sent to it by the MCU. Accordingly, the MCU's **Initial** function includes a half-second delay followed by a disabling of the brownout-reset circuit. Any power switch contact bounce occurring during this half second will reset the MCU and start up again with the same, proper initialization sequence of instructions, followed by the shutting down of the brownout-reset module to eliminate its constant 34 μA current draw on the coin cell.

The LCD controller powers up in the same way, to keep from being corrupted by power switch contact bounce. It uses a shorter delay, but with enough leeway to account for any difference in the brownout modules' threshold voltages and for the more extensive initialization required by the LCD controller.

5.8 SPI INITIALIZATION

The initialization of the serial peripheral interface consists of the initialization of the **SSPSTAT** and **SSPCON1** registers with the values shown in Figure 5-1c. Also, the **RD5** pin of **PORTD** is set up as an output, driven high. Thus, when the MCU is ready to send its first display string to the LCD controller, all three output lines, **RD5**, **SCK**, and **SDO** will have been correctly initialized.

5.9 THE DISPLAY FUNCTION AND LCDSTRING

The **Display** function is called twice within the T2.c template program. First, it is called in the **Initial** function when it displays "PRESS PB". Subsequently, it is called by the **UpdateLCD** function after the pushbutton has first been pressed. Thereafter, it is called by the **UpdateLCD** function to display the elapsed seconds and to update the number of pushbutton presses.

Before the **Display** function is called, **LCDSTRING** must be loaded with the nine ASCII-coded characters to be sent to the display. The T2.c template program does this in several ways. In the *Global variables* section, the line

```
char LCDSTRING[] = "PRESS PB "; // LCD display string
```

both defines **LCDSTRING** as a *char* array and initializes it to contain the nine characters between the quotes. At the end of the **Initial** function, the **Display** function sends this message string to the display and then uses a **LoadLCDSTRING** macro to reinitialize **LCDSTRING** with the initial values of the numbers that will be displayed subsequently, when the **Time** function and the **Pushbutton** function update individual characters in **LCDSTRING**. When the string of characters is sent to the display, the sequence of events shown in Figure 5-2 occur.

The **Display** function begins by dropping **RD5** to awaken the LCD controller. It sends the nine characters to the display via the SPI bus. After clearing the SPI's **SSPIF** flag, each character is written to the SPI's **SSPBUF** register. The function then waits for the completion of the transfer (signaled by the setting of the **SSPIF** flag) before sending the next character. After all characters have been sent, **RD5** is raised, ready for the next call of **Display**.

5.10 THE TIME FUNCTION

Within the **Time** function, the lines

```
LCDSTRING[0] = TENS;
```

and

```
LCDSTRING[1] = UNITS;
```

insert the ASCII characters stored in the **TENS** and **ONES** variables into the first 2 bytes of **LCDSTRING**. The lines leading up to these lines increment the two-digit ASCII-coded number in **TENS:ONES** from 00 to 59 and back to 00. By counting 62 loop times in **TIMECNT** between each increment of **TENS:ONES**, the displayed time is incremented every second (within the accuracy of the watchdog timer's nominal 16 ms timeout period). The test of **PBFLAG** maintains the initial startup message on the display

PRESS PB

until the first press of the pushbutton. The setting of the **LCDFLAG** at the end of the **Time** function is used to signal the **UpdateLCD** function that **LCDSTRING** has been changed and that the LCD should be updated accordingly.

5.11 THE PUSHBUTTON FUNCTION

Like the **Time** function, the **Pushbutton** function increments a counter, **PBTENS: PBONES**. When the **Pushbutton** function has updated **LCDSTRING** in response to a pushbutton press, it sets **LCDFLAG**, just as was done by the **Time** function. Consequently, whenever either a 1-s tick or a pushbutton press occurs, the **UpdateLCD** function will update the display during that same pass around the main loop.

To understand how the MCU detects a pushbutton press, refer back to the circuit of Figure 3-2. **RE0** is first raised. A 1-µs pause is introduced by the **Nop()** macro to allow time to change whatever capacitance is associated with the **RE0** trace on the Qwik&Low board. If the pushbutton is pressed, the **RD7** input will be read as a zero and the line

```
NEWPB = !PORTDbits.RD7;
```

will put a nonzero value into the **NEWPB** byte serving as a flag. If the pushbutton is not pressed, **RD7** will be read as a one and **NEWPB** will be zero.

The state of the pushbutton one loop time (i.e., 16 ms) ago is held in **OLDPB**. The combined condition

```
!OLDPB && NEWPB
```

detects the beginning of a keypress. Keybounce has been suppressed by the loop time sampling of the keyswitch state, as per Figure 2-7.

Before the first press after reset, **PBFLAG** will equal zero. Accordingly, when the first press occurs, **PBFLAG** will be set and the initialization of **ALIVECNT** and **TIMECNT** will occur just for this initial press. The ASCII values for zero and one initialized into **PBTENS** and **PBONES** are copied into **LCDSTRING[3]** and **LCDSRING[4]** and **LCDFLAG** is set, to signal the **UpdateLCD** function to overwrite the

PRESS PB

message with a time of 00 and a number of keypresses of 01. On subsequent keypresses, the ASCII values held in **PBTENS** and **PBONES** are incremented and then displayed by the **UpdateLCD** function.

PROBLEMS

5-1 Initial message Change the initial message from “PRESS PB” to “WELCOME”.

5-2 Relocation of display elements

a) Move the elapsed time to the CHAR1:CHAR2 position on the display.

b) Move the count of pushbutton presses to the CHAR5:CHAR6 position on the display.

5-3 Blast off counter Change the display of elapsed time to count down from an initial value of 10. When zero is reached:

a) Fill the screen with eight asterisks and stop further updating of the display.

b) Blank the screen. Twelve loop times later write asterisks to the middle two character positions. After another 12 loop times, write asterisks to the middle four character positions. After another 12 loop times, write asterisks to the middle six character positions. Finally, after another 12 loop times, fill the screen with eight asterisks and stop any further updating of the display.

Chapter

6

PC MONITOR USE (MEASURE.c)

6.1 OVERVIEW

Because a user program has access to the same UART module in the MCU that is used by QwikBug, the *Console* window within QwikBug can provide the Qwik&Low board with two distinct opportunities:

- It can be used by a user's application program to supplement the eight-character LCD. For example, it can display the 16-hex-digit serial number read from the DS2401 silicon serial number IC of Chapter Fifteen.
- It can be used by a user's test program that exercises a user algorithm or function as a way to report measurement results. For example, this chapter will end with a Measure.c program that compares the execution time of four functions that convert a variable into a decimal display.

The chapter begins with an examination of the MCU's UART module, its setup, and its ability to transmit data reliably to the PC using the MCU's internal oscillator having a ±2% frequency accuracy. The chapter ends with the Measure.c template program. When compared with the use of a scope to measure execution times in units of microseconds, the counting of CPU cycles explored here produces exact CPU cycle counts. Consequently, its measurement results of cycle counts do not vary as the same code is run on multiple Qwik&Low boards. In contrast, scope measurements of execution times in units of microseconds will vary from board to board because of variations in the 2% accurate internal clock frequency.

6.2 WAVEFORMS AND BAUD RATE ACCURACY

The *UART*, universal asynchronous receiver transmitter, is a module in the MCU that is used by the QwikBug utility to send and receive information between the PC and the MCU. For this transmission, the PC employs a *baud* rate of 19,200 baud; that is, a transmission rate in which the duration of each bit is

$$1 \text{ bit time} = \frac{1}{19{,}200}\text{ s} \approx 50\ \mu\text{s}$$

The protocol employed for the asynchronous serial data transmission from the MCU to the PC is illustrated in Figure 6-1 for a 3-byte transfer. Each byte is *framed* between a high start bit and a low stop bit, producing a 10-bit frame having a duration of about half a millisecond. As pointed out in conjunction with Figure 3-3, the MCU is able to implement the signal level inversion for its output to the PC that is normally implemented with an external chip. Thus, the TX signal idles low to drive the RS-232 cable going to the PC, just the opposite of what would be expected from a UART whose output is inverted externally.

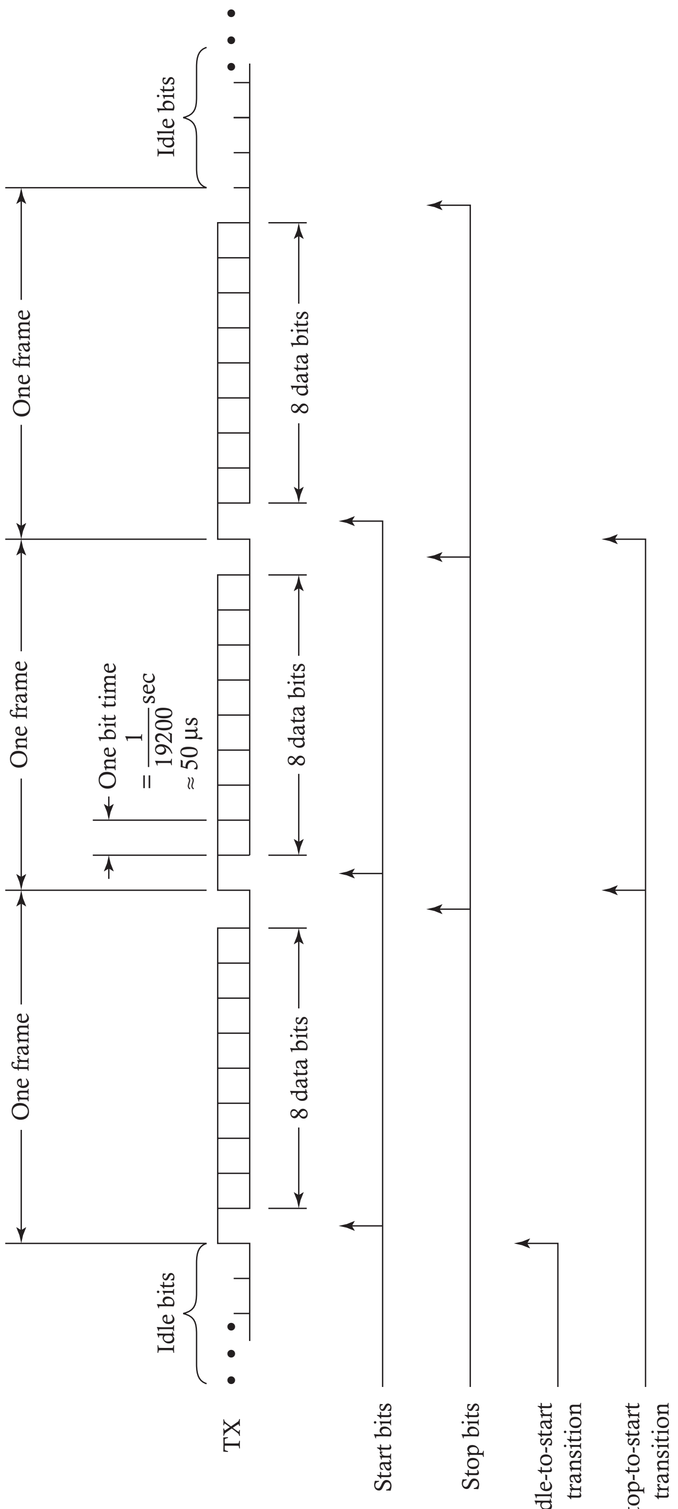
Because both clock and data are combined in the single TX output from the MCU, the PC must synchronize on the serial data stream in order to read the data bits reliably. The PC knows that each byte of data is framed between low idle bits or between the low trailing stop bit of a frame and the high leading start bit of the following frame. This low-to-high transition triggers a counter in the PC's UART that divides each bit time into 16 "ticks".

The PC's crystal baud rate oscillator with a frequency accuracy of better than 100 parts per million will introduce an error, relative to the nominal 19,200 baud rate, of no more than 0.01%. Because of its sampling of the received waveform, the PC's UART can miss the time of the rising edge of the start bit by up to one tick. The PC's UART actually reads each bit in the middle of each bit time as measured by counting ticks. Thus, as shown in Figure 6-2, each frame consisting of 160 ticks is sampled at the 24th, . . . , 136th ticks to read the 8 data bits. It finally samples the input at the 152nd tick and expects to read the low stop bit.

If the input is high at the 152nd tick because of the MCU's baud rate clock frequency being off from the nominal 19,200 baud by a sufficient amount, the PC's UART registers a *framing error*. The effect of a slow MCU baud rate clock is to stretch the waveform of Figure 6-2 relative to the PC's tick clock. If this stretching is as much as 160 – 152 – 1 = 7 ticks relative to the 152 ticks when the stop bit is read, a framing error will occur, signaling the reception of possibly erroneous data. A maximum deviation of the MCU baud rate from the nominal baud rate of 19,200 baud follows from this of

$$\text{Baud rate} = 19{,}200 \text{ baud} \pm \frac{7}{152} \times 100\% = 19{,}200 \text{ baud} \pm 4.60\%$$

The generation of a baud rate approximating 19,200 baud by the MCU is illustrated by the circuit of Figure 6-3a, with the chip's internal oscillator being divided down by either 16 or 64 followed by a divide-by-(N + 1) counter. The resulting relationship



Receiver synchronizes on Idle-to-start transition and resynchronizes on each subsequent stop-to-start transition.

FIGURE 6-1 Transmission of a three-byte string

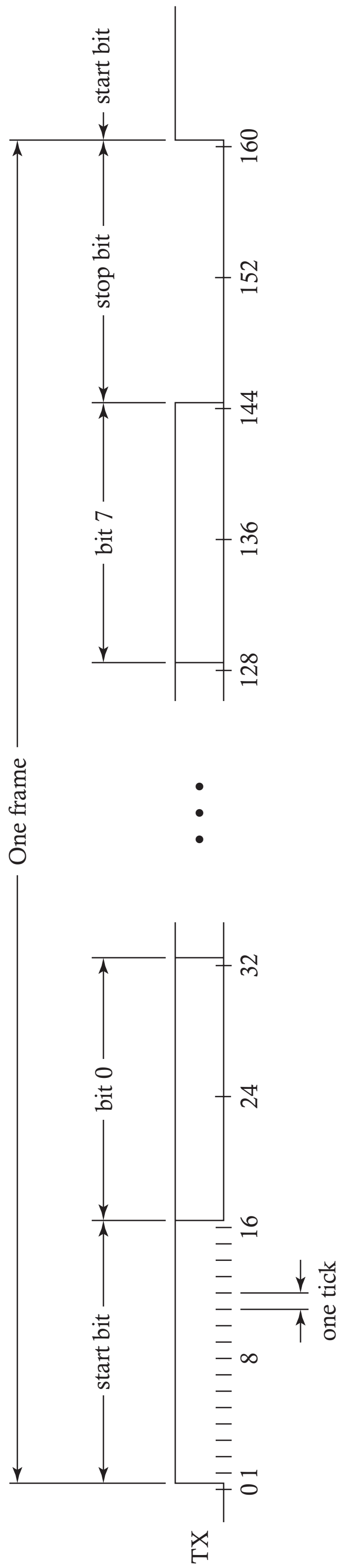
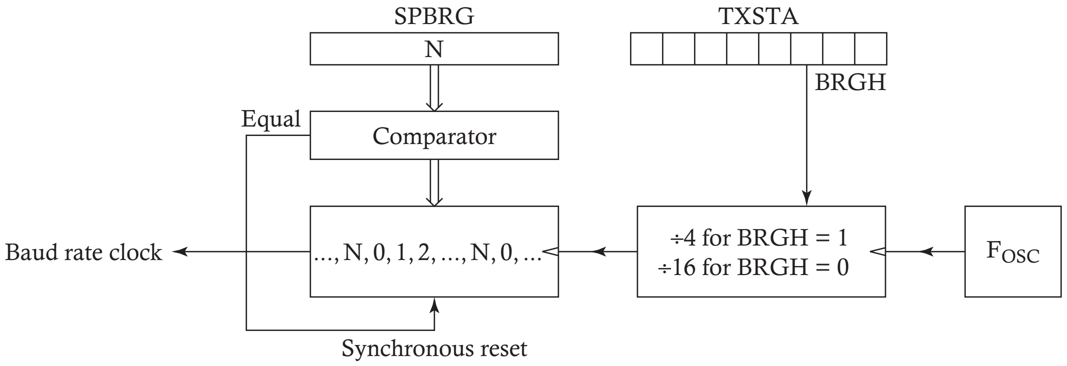


FIGURE 6-2 Division of one frame into 160 ticks

between F_{OSC}, baud rate, **BRGH**, and **SPBRG** is shown in Figure 6-3b. Using the MCU's INTOSC internal oscillator, the 19,200-baud rate of the PC can be approximated by any of the four settings of Figure 6-3c, depending upon the F_{osc} value selected.

The frequency error of the INTOSC oscillator is specified to be less than ±2% at 25°C (i.e., 77°F) over the full supply voltage range of 2.0 V to 5.5 V. This error plus the 0.16% baud-rate error of Figure 6-3c are comfortably less than the ±4.60% accuracy required by the PC's UART.



(a) MCU'S UART baud-rate generator circuit

BRGH = 1	$\frac{F_{OSC}}{\text{Baud rate}} = 4\,(\text{SPBRG} + 1)$
BRGH = 0	$\frac{F_{OSC}}{\text{Baud rate}} = 16\,(\text{SPBRG} + 1)$

(b) Baud rate derivation from F_{OSC}

F_{OSC}	BRGH	SPBRG	Baud rate error
8 MHz	0	25	–0.16%
4 MHz	0	12	–0.16%
2 MHz	1	25	–0.16%
1 MHz	1	12	–0.16%

(c) BRGH and SPBRG settings for 19200 baud

FIGURE 6-3 Baud rate generation by the MCU

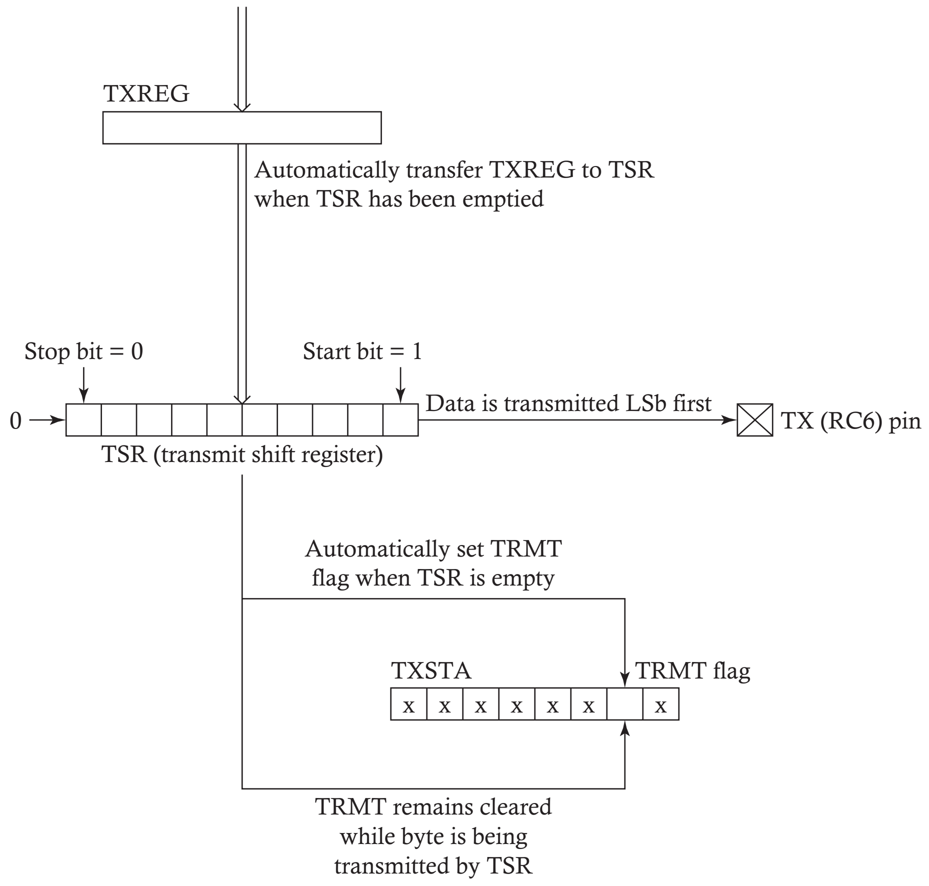

TXREG
Automatically transfer TXREG to TSR when TSR has been emptied
Stop bit = 0
Start bit = 1
0
Data is transmitted LSb first
TX (RC6) pin
TSR (transmit shift register)
Automatically set TRMT flag when TSR is empty
TXSTA
TRMT flag
x x x x x x x
TRMT remains cleared while byte is being transmitted by TSR

FIGURE 6-4 TX circuitry

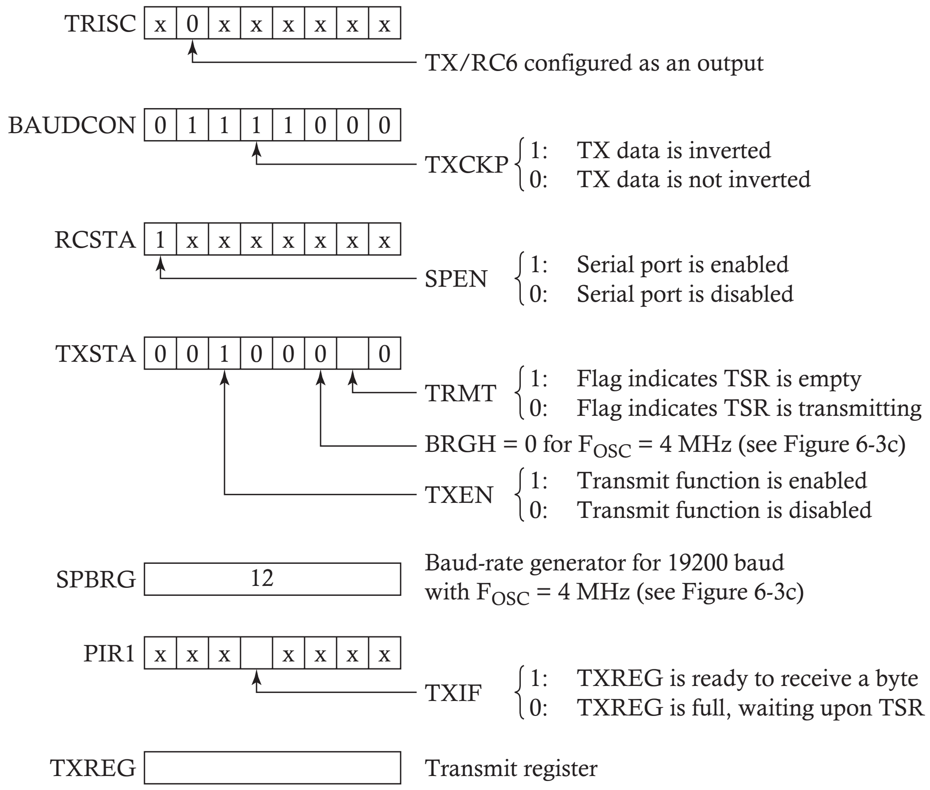
6.3 UART'S TX CIRCUITRY AND USE

The circuitry of Figure 6-4 implements the TX (transmit) portion of the UART module in the MCU. It consists of two registers plus a **TRMT** (transmit) flag that can be used for flow control. When a string of bytes is sent to the display, before a new byte is written to **TXREG**, a pause until the present byte in the UART has been completely transferred can be implemented by pausing while **TRMT** = 0.

An alternative flag (**TXIF**) could have been used that signals when **TXREG** is ready for a new byte. This flag provides the benefit of allowing 2 bytes to be written to the UART before the first half-millisecond pause occurs. However, before the chip is put to sleep, it is necessary to pause while **TRMT** = 0 so that no intended byte being sent to the PC is aborted when F_{OSC} is stopped.

6.4 UART INITIALIZATION

The UART module in the MCU must be initialized before it can be used. The baud rate settings of Figure 6-3c for F_{OSC} = 4 MHz are reflected in the register contents of Figure 6-5.


TRISC x 0 x x x x x x
TX/RC6 configured as an output
BAUDCON 0 1 1 1 1 0 0 0
TXCKP 1: TX data is inverted 0: TX data is not inverted
RCSTA 1 x x x x x x x
SPEN 1: Serial port is enabled 0: Serial port is disabled
TXSTA 0 0 1 0 0 0 0
TRMT 1: Flag indicates TSR is empty 0: Flag indicates TSR is transmitting
BRGH = 0 for F_OSC = 4 MHz (see Figure 6-3c)
TXEN 1: Transmit function is enabled 0: Transmit function is disabled
SPBRG 12
Baud-rate generator for 19200 baud with F_OSC = 4 MHz (see Figure 6-3c)
PIR1 x x x x x x x
TXIF 1: TXREG is ready to receive a byte 0: TXREG is full, waiting upon TSR
TXREG
Transmit register

(a) Registers

```
/*******************************
 * InitTX
 *
 * This function initializes the UART for its TX output function. It assumes
 * Fosc = 4 MHz. For a different oscillator frequency, use Figure 6-3c to
 * change BRGH and SPBRG appropriately.
 *******************************
 */

void InitTX()
{
    RCSTA = 0b10010000;         // Enable UART
    TXSTA = 0b00100000;         // Enable TX
    SPBRG = 12;                 // Set baud rate
    BAUDCON = 0b00111000;       // Invert TX output
}
```

(b) Initialization

FIGURE 6-5 UART registers and initialization for TX output

Even though QwikBug has already initialized the UART in order to download a user program, run it, and aid in debugging it, QwikBug has done so with F_{OSC} = 8 MHz. For a user program operating with F_{OSC} = 4 MHz, the baud rate settings must be reinitialized to the settings shown in Figure 6-5 in order to have the PC accept the MCU output correctly at 19,200 baud. QwikBug handles these shared registers with care, saving user contents on entering QwikBug at a breakpoint or after a single step, and restoring the user contents on exiting back to the user program.

6.5 TXASCII MACRO

The fundamental building block for sending an ASCII-coded character to the PC is a **TXascii** macro. The macro does two things:

- It sends its ASCII-coded parameter, whether a constant or a *char* variable, to **TXREG** for transmission to the PC.
- It waits for the completion of the transfer by testing the **TRMT** bit of Figure 6-5a and pausing until it becomes set.

The macro definition and examples of its use are shown in Figure 6-6.

6.6 NUMBER-TO-ASCII CONVERSION

In the last chapter, ASCII-coded characters were formed in the **Time** function (and similarly in the **Pushbutton** function) by starting with

```
ONES = '0'              and          TENS = '0'
```

That is, each of these variables began with the ASCII code for zero. Thereafter, these values were updated by incrementing to the next ASCII code or by resetting to `'0'`.

```
#define TXascii(in)   TXREG = in; while(!TXSTAbits.TRMT)
```

(a) TXascii macro definition

```
TXascii(HUNDREDS);          // Display ASCII-coded content of HUNDREDS

TXascii(0x41);              // Display the letter A
TXascii(0x0D);              // Carriage return
TXascii(0x0A);              // Line feed

TXascii('A');               // Display the letter A
TXascii('\r');              // Carriage return
TXascii('\n');              // Line feed
```

(b) Useful invocations

FIGURE 6-6 TXascii macro definition and several useful invocations

More generally, a number will be obtained as a result of a measurement and will need to be converted to ASCII-coded *char* variables:

ONES **TENS** **HUNDREDS** **THOUSANDS** etc.

ready for display. In this section, two algorithms will be considered. The first breaks out the digits, most-significant-digit first, by successive subtractions. The second breaks out the digits, least-significant-digit first, by successive divisions. For each of these algorithms, two versions will be developed.

ASCII and **ASCIID** convert **NUMBER**, a value ranging from 0 to 255, with the functions shown in Figure 6-7. The first line of **ASCII** initializes the three output variables to '0'. The second line forms **HUNDREDS** by repeatedly subtracting 100 from **NUMBER** until **NUMBER** is less than 100. The third line forms **TENS** by repeatedly subtracting 10 from what remains in **NUMBER**. The fourth line is reached with **NUMBER** having a value ranging between zero and nine. This value is added to the ASCII-coded zero initialized into **ONES**.

In Figure 6-8, **ASCII4** and **ASCII4D** operate on **BIGNUM**, the *int* version of **NUMBER** by adding the extra lines of code needed to generate one more digit. Although numbers up to 65,535 can be held in **BIGNUM**, restricting the conversion to any four-digit number up to 9,999 will serve the needs that arise in this book.

```
Global variables:

unsigned char NUMBER;             // Eight-bit number to be converted
unsigned char HUNDREDS,TENS,ONES;  // ASCII coding of digits

Function prototypes:

void ASCII(void);
void ASCIID(void);
```

(a) Definitions

```
/*******************************
 * ASCII
 *
 * This function converts the unsigned char parameter passed to it
 * in NUMBER, ranging from 0 to 255, to three ASCII-coded digits
 * by performing successive subtractions.
 * Simplified by Chad Kersey.            Takes up to 98 cycles.
 *******************************
 */
void ASCII()
{
   ONES = TENS = HUNDREDS ='0';  //Initialize to ASCII zeroes
   while (NUMBER >= 100) { HUNDREDS++; NUMBER -= 100; } // Form HUNDREDS
```

FIGURE 6-7 Conversion of the *char* variable NUMBER ranging from 0 to 255

```
   while (NUMBER >= 10) { TENS++; NUMBER -= 10; }    // Form TENS
   ONES += NUMBER;                                   // Form ONES
}
```

(b) ASCII for conversion by successive subtractions

```
/******************************
 * ASCIID
 *
 * This function converts the unsigned char parameter passed to it
 * in NUMBER, ranging from 0 to 255, to three ASCII-coded digits
 * by performing successive divisions.          Takes up to 357 cycles.
 ******************************
 */
void ASCIID()
{
   ONES = '0' + (NUMBER % 10);        // Form ONES
   NUMBER = NUMBER / 10;
   TENS = '0' + (NUMBER % 10);        // Form TENS
   HUNDREDS = '0' + (NUMBER / 10);    // Form HUNDREDS
}
```

(c) ASCIID for conversion by successive divisions

FIGURE 6-7 *(continued)*

6.7 MEASURE.c, A CYCLE COUNTING PROGRAM

With the four functions of the last section and the **TXascii** macro of Section 6.6 ready for use, this section introduces in Figure 6-9 a Measure.c template program that will evaluate the number of cycles needed to execute each function. Using **Start**, **Stop**, and **Send** functions that will be developed in Chapter 13, Measure.c starts a counter of CPU clock cycles (Timer0) immediately before the call of each ASCII conversion function, stops the counter immediately after the conversion, and sends the resulting number of cycles to the QwikBug Console. The numbers used for the conversions

NUMBER = 199 and **BIGNUM** = 9,999

represent worst-case values (i.e., values that produce the most cycles) for **ASCII** and **ASCII4**, the successive-subtraction algorithms. They are reasonable values for estimating worst-case cycle counts for the successive-division algorithms. Determining the actual worst-case cycle count for each of the two successive-division algorithms is left as end-of-chapter problems.

The resulting numbers of cycle counts are listed in the header of the Measure.c template. The successive-subtraction algorithms require, in the worst case, about a quarter of the number of cycles of the successive-division algorithms. Furthermore, the successive-subtraction algorithms produce cycle counts that are proportional to the sum of the digits in the result, and can thus produce a significantly reduced cycle count in a specific case. For these reasons, ASCII and ASCII4 are used throughout the rest of the book whenever a conversion is needed.

```
Global variables:

unsigned int BIGNUM;        // Ranges from 0 to 9999
unsigned char THOUSANDS,HUNDREDS,TENS,ONES; // ASCII coding of digits

Function prototypes:

void ASCII4(void);
void ASCII4D(void);
```

(a) Definitions

```
/*******************************
 * ASCII4
 *
 * This function converts the unsigned int parameter passed to it
 * in BIGNUM, ranging from 0 to 9999, to four ASCII-coded digits
 * by performing successive subtractions.
 * Simplified by Chad Kersey.                    Takes up to 353 cycles.
 *******************************
 */
void ASCII4()
{
   ONES = TENS = HUNDREDS = THOUSANDS ='0';  //Initialize to ASCII zeroes
   while (BIGNUM >= 1000) { THOUSANDS++; BIGNUM -= 1000; } // Form THOUSANDS
   while (BIGNUM >= 100)  { HUNDREDS++; BIGNUM -= 100; }   // Form HUNDREDS
   while (BIGNUM >= 10)  { TENS++; BIGNUM -= 10; }         // Form TENS
   ONES += BIGNUM;                                         // Form ONES
}
```

(b) ASCII4 for conversion by successive subtractions

```
/*******************************
 * ASCII4D
 *
 * This function converts the unsigned int parameter passed to it
 * in BIGNUM, ranging from 0 to 9999, to four ASCII-coded digits
 * by performing successive divisions.           Takes up to 1498 cycles.
 *******************************
 */
void ASCII4D()
{
   ONES = '0' + (BIGNUM % 10);       // Form ONES
   BIGNUM = BIGNUM / 10;
   TENS = '0' + (BIGNUM % 10);       // Form TENS
   BIGNUM = BIGNUM /10;
   HUNDREDS = '0' + (BIGNUM % 10);   // Form HUNDREDS
   THOUSANDS = '0' + (BIGNUM / 10);  // Form THOUSANDS
}
```

(c) ASCII4D for conversion by successive divisions

FIGURE 6-8 Conversion of the *int* variable BIGNUM ranging from 0 to 9999

```
/******* Measure.c *************
 *
 * A number between 0 and 9999 is converted to ASCII-coded digits two ways:
 *   ASCII4 forms each digit by successive subtractions (up to 353 cycles).
 *   ASCII4D forms each digit via two divisions (up to 1498 cycles).
 * Then a number between 0 and 255 is converted to ASCII-coded digits two ways:
 *   ASCII forms each digit by successive subtractions (up to 98 cycles).
 *   ASCIID forms each digit via two divisions (up to 357 cycles).
 * For each one, the result is displayed on the LCD.
 * The execution time (cycles)is displayed on the PC
 * Execution stops with a sleep command.
 *
 * Start and Stop functions are added to measure the execution time of the
 * code between them.  The Send function sends the time to the PC monitor.
 *
 * Use Fosc = 4 MHz for Fcpu = Fosc/4 = 1 MHz.
 *
 ******* Program hierarchy *****
 *
 * main
 *  Initial
 *  InitTX
 *  Start
 *  Stop
 *  Send
 *    TXascii
 *  ASCII4
 *  ASCII4D
 *  ASCII
 *  ASCIID
 *  Display
 *
 *******************************
 */

#include <p18f4321.h>           // Define PIC18LF4321 registers and bits

/*******************************
 * Configuration selections
 *******************************
 */
#pragma config OSC = INTIO1     // Use internal osc, RA6=Fosc/4, RA7=I/O
#pragma config PWRT = ON        // Enable power-up delay
#pragma config LVP = OFF        // Disable low-voltage programming
#pragma config WDT = OFF        // Disable watchdog timer initially
#pragma config WDTPS = 4        // 16 millisecond WDT timeout period, nominal
#pragma config MCLRE = ON       // Enable master clear pin
#pragma config PBADEN = DIG     // PORTB<4:0> = digital
#pragma config CCP2MX = RB3     // Connect CCP2 internally to RB3 pin
#pragma config BOR = SOFT       // Brown-out reset controlled by software
#pragma config BORV = 3         // Brown-out voltage set for 2.1V, nominal
#pragma config LPT1OSC = OFF    // Deselect low-power Timer1 oscillator
```

FIGURE 6-9 Measure.c

```
/*******************************
 * Global variables
 *******************************
 */
unsigned int DELAY;             // Sixteen-bit counter for obtaining a delay
unsigned char NUMBER;           // Eight-bit number to be converted
unsigned int BIGNUM;            // Sixteen-bit number to be converted
unsigned char THOUSANDS,HUNDREDS,TENS,ONES; // ASCII coding of digits
unsigned char i;                // Index into strings
unsigned int CYCLES;            // Result of Timer0 counting cycles
char LCDSTRING[] = "         "; // Nine-character display string

/*******************************
 * Function prototypes
 *******************************
 */
void Initial(void);
void InitTX(void);
void Start(void);
void Stop(void);
void Send(void);
void ASCII4(void);
void ASCII4D(void);
void ASCII(void);
void ASCIID(void);
void Display(void);

/*******************************
 * Macros
 *******************************
 */
#define Delay(x) DELAY = x; while(--DELAY){ Nop(); Nop(); }
#define TXascii(in)  TXREG = in; while(!TXSTAbits.TRMT)

/////// Main program //////////////////////////////////////////////////////

/*******************************
 * main
 *******************************
 */
void main()
{
   Initial();                   // Initialize everything
   InitTX();                    // and the UART as well

   BIGNUM = 9999;
   Start();
   ASCII4();                    // Convert BIGNUM           Takes 353 cycles
   Stop();
   LCDSTRING[0] = THOUSANDS;
   LCDSTRING[1] = HUNDREDS;
   LCDSTRING[2] = TENS;
```

FIGURE 6-9 *(continued)*

```
   LCDSTRING[3] = ONES;
   Send();                     // Send cycle count to PC for display

   LCDSTRING[4] = '.';         // Use decimal point as separator

   BIGNUM = 9999;
   Start();
   ASCII4D();                  // Convert BIGNUM          Takes 1498 cycles
   Stop();
   LCDSTRING[5] = THOUSANDS;
   LCDSTRING[6] = HUNDREDS;
   LCDSTRING[7] = TENS;
   LCDSTRING[8] = ONES;
   Send();                     // Send this cycle count to PC for display
   Display();                  // Verify correct conversions on LCD

   Delay(50000); Delay(50000); Delay(50000); Delay(50000) // Two-second pause

   NUMBER = 199;
   Start();
   ASCII();                    // Convert NUMBER           Takes 98 cycles
   Stop();
   LCDSTRING[0] = ' ';
   LCDSTRING[1] = HUNDREDS;
   LCDSTRING[2] = TENS;
   LCDSTRING[3] = ONES;
   Send();                     // Send cycle count to PC for display

   LCDSTRING[4] = '.';         // Use decimal point as separator

   NUMBER = 199;
   Start();
   ASCIID();                   // Convert NUMBER           Takes 357 cycles
   Stop();
   LCDSTRING[5] = ' ';
   LCDSTRING[6] = HUNDREDS;
   LCDSTRING[7] = TENS;
   LCDSTRING[8] = ONES;
   Send();                     // Send this cycle count to PC for display
   Display();                  // Verify correct conversions on LCD

   Sleep();                    // Sleep forever
}

/******************************
 * Initial
 *
 * This function performs all initializations of variables and registers.
 ******************************
 */
```

FIGURE 6-9 *(continued)*

```
void Initial()
{
   OSCCON = 0b01100010;         // Use Fosc = 4 MHz (Fcpu = 1 MHz)
   SSPSTAT = 0b00000000;        // Set up SPI for output to LCD
   SSPCON1 = 0b00110000;
   ADCON1 = 0b00001011;         // RA0,RA1,RA2,RA3 pins analog; others
                                // digital
   TRISA = 0b00001111;          // Set I/O for PORTA
   TRISB = 0b01000100;          // Set I/O for PORTB
   TRISC = 0b10000000;          // Set I/O for PORTC
   TRISD = 0b10000000;          // Set I/O for PORTD
   TRISE = 0b00000010;          // Set I/O for PORTE
   PORTA = 0;                   // Set initial state for all outputs low
   PORTB = 0;
   PORTC = 0;
   PORTD = 0b00100000;          // except RD5 that drives LCD interrupt
   PORTE = 0;
   SSPBUF = ' ';                // Send a blank to initialize state of UART
   Delay(50000);                // Pause for half a second
   RCONbits.SBOREN = 0;         // Now disable brown-out reset
}

/*******************************
 * InitTX
 *
 * This function initializes the UART for its TX output function.  It assumes
 * Fosc = 4 MHz.  For a different oscillator frequency, use Figure 6-3c to
 * change BRGH and SPBRG appropriately.
 *******************************
 */
void InitTX()
{
   RCSTA = 0b10010000;          // Enable UART
   TXSTA = 0b00100000;          // Enable TX
   SPBRG = 12;                  // Set baud rate
   BAUDCON = 0b00111000;        // Invert TX output
}

/*******************************
 * Start
 *
 * This function clears Timer0 and then starts it counting.
 *******************************
 */
void Start()
{
   T0CON = 0b00001000;          // Set up Timer0 to count CPU clock cycles
   TMR0H = 0;                   // Clear  Timer0
   TMR0L = 0;
   T0CONbits.TMR0ON = 1;        // Start counting
}
```

FIGURE 6-9 *(continued)*

```
/*******************************
 * Stop
 *
 * This function stops counting Timer0, and reads the result into CYCLES.
 *******************************
 */
void Stop()
{
   T0CONbits.TMR0ON = 0;         // Stop counting
   CYCLES = TMR0L;               // Form CYCLES from TMR0H:TMR0L
   CYCLES += (TMR0H * 256);
   CYCLES -= 3;                  // Remove 3 counts so back-to-back Start-Stop
}                                // functions produce CYCLES = 0

/*******************************
 * Send
 *
 * This function converts CYCLES to four ASCII-coded digits and sends
 * the result to the PC for display.
 *******************************
 */
void Send()
{
   BIGNUM = CYCLES;              // Load ASCII4's input parameter
   ASCII4();                     // Convert
   TXascii('\r');                // Send carriage return
   TXascii('\n');                // Send line feed
   TXascii(THOUSANDS);           // Send four-digit number
   TXascii(HUNDREDS);
   TXascii(TENS);
   TXascii(ONES);
}

/*******************************
 * Display()
 *
 * This function sends LCDSTRING to the LCD.
 *******************************
 */
void Display()
{
   PORTDbits.RD5 = 0;            // Wake up LCD display
   for (i = 0; i <= 8; i++)
   {
      PIR1bits.SSPIF = 0;        // Clear SPI flag
      SSPBUF = LCDSTRING[i];     // Send byte
      while (!PIR1bits.SSPIF);  // Wait for transmission to complete
   }
   PORTDbits.RD5 = 1;            // Return RB5 high, ready for next string
}
```

FIGURE 6-9 *(continued)*

```
/******************************
 * ASCII
 *
 * This function converts the unsigned char parameter passed to it
 * in NUMBER, that ranges between 0 and 255, to three ASCII-coded digits
 * by performing successive subtractions.
 * Simplified by Chad Kersey.                Takes a maximum of 98 cycles.
 ******************************
 */
void ASCII()
{
   ONES = TENS = HUNDREDS ='0';  //Initialize to ASCII zeroes
   while (NUMBER >= 100)  { HUNDREDS++; NUMBER -= 100; }  // Form HUNDREDS
   while (NUMBER >= 10)  { TENS++; NUMBER -= 10; }        // Form TENS
   ONES += NUMBER;                                        // Form ONES
}

/******************************
 * ASCIID
 *
 * This function converts the unsigned char parameter passed to it
 * in NUMBER, that ranges between 0 and 255, to three ASCII-coded digits
 * by performing successive divisions.             Takes up to 357 cycles.
 ******************************
 */
void ASCIID()
{
   ONES = '0' + (NUMBER % 10);       // Form ONES
   NUMBER = NUMBER / 10;
   TENS = '0' + (NUMBER % 10);       // Form TENS
   HUNDREDS = '0' + (NUMBER / 10);   // Form HUNDREDS
}

/******************************
 * ASCII4
 *
 * This function converts the unsigned int parameter passed to it
 * in BIGNUM, that ranges between 0 and 9999, to four ASCII-coded digits
 * by performing successive subtractions.
 * Simplified by Chad Kersey.                Takes a maximum of 353 cycles.
 ******************************
 */
void ASCII4()
{
   ONES = TENS = HUNDREDS = THOUSANDS ='0';  //Initialize to ASCII zeroes
   while (BIGNUM >= 1000)  { THOUSANDS++; BIGNUM -= 1000; } // Form THOUSANDS
   while (BIGNUM >= 100)  { HUNDREDS++; BIGNUM -= 100; }    // Form HUNDREDS
   while (BIGNUM >= 10)  { TENS++; BIGNUM -= 10; }          // Form TENS
   ONES += BIGNUM;                                          // Form ONES
}
```

FIGURE 6-9 *(continued)*

```
/******************************
 * ASCII4D
 *
 * This function converts the unsigned int parameter passed to it
 * in BIGNUM, that ranges between 0 and 9999, to four ASCII-coded digits
 * by performing successive divisions.             Takes up to 1498 cycles.
 ******************************
 */
void ASCII4D()
{
   ONES = '0' + (BIGNUM % 10);       // Form ONES
   BIGNUM = BIGNUM / 10;
   TENS = '0' + (BIGNUM % 10);       // Form TENS
   BIGNUM = BIGNUM /10;
   HUNDREDS = '0' + (BIGNUM % 10);   // Form HUNDREDS
   THOUSANDS = '0' + (BIGNUM / 10);  // Form THOUSANDS
}
```

FIGURE 6-9 *(continued)*

PROBLEMS

6-1 ASCIID worst case Modify the Measure.c template to form a Measure-ASCIID.c. This program is to run the successive-division algorithm, ASCIID, 256 times with each possible value of **NUMBER.** Each run is to (possibly) update two *int* values **MIN** and **MAX** and to update a *short long* (i.e., 24-bit) value **SUM**. At the conclusion, send **MIN** and **MAX** to the QwikBug Console for display. Then form

```
AVG = (int)(SUM >> 8)
```

to divide **SUM** by the 256 trials to get the average number of cycles. Send **AVG** out for display.

6-2 ASCII worst case Repeat the last problem for the ASCII successive-subtraction algorithm.

6-3 ASCII4D worst case Repeat for the 0–9,999 cases of the four-digit successive-division algorithm.

6-4 ASCII4 worst case Repeat for the 0–9,999 cases of the four-digit successive-subtraction algorithm.

6-5 Display execution time Measure the number of cycles taken to execute the **Display** function.

Chapter

7

REORGANIZATION OF TIMING VIA INTERRUPTS (T3.c)

7.1 OVERVIEW

The code of the template programs to this point has put the MCU to sleep after carrying out the main loop tasks. Then the low-power watchdog timer has awakened the chip every 16 ms to repeat the mainline tasks.

In this chapter, an alternative approach for awakening the chip is employed. Now the watchdog timer remains disabled. The MCU's low-power Timer1 oscillator runs continuously and is able to clock its Timer1 counter regardless of whether the remainder of the MCU is asleep or awake. The Timer1 counter is used to control the loop time by awakening the chip periodically with an interrupt. The MCU also has a Timer3 counter, also clocked by the Timer1 oscillator. Tasks requiring faster periodic "ticks" can easily do so by using Timer3 to produce high-priority interrupts.

7.2 LOW- AND HIGH-PRIORITY INTERRUPTS

The PIC18LF4321 supports two levels of interrupts. It also supports a score of interrupt sources, any one of which can be used to suspend the CPU's execution of the main program and divert the CPU to either a *high-priority interrupt service routine* (HPISR) or a *low-priority interrupt service routine* (LPISR). Furthermore, if a low-priority interrupt has interrupted the main program and the CPU has begun executing the LPISR when a high-priority interrupt source requests service, the CPU automatically suspends the LPISR and executes the HPISR before returning to where it left off in the LPISR.

A new template program, T3.c, is introduced that still carries out the tasks of the T2.c template, but with the code reorganized as shown in Figure 7-1. The new **main** function calls the **Initial** function where it carries out all of the initialization tasks of T2.c plus the initialization of Timer1 as a low-priority interrupt source and Timer3 as a high-priority interrupt source. Then the CPU puts itself to sleep. Only the Timer1 oscillator with its external 32,768-Hz watch crystal and the Timer1 and Timer3 counters continue to run. The current draw of the MCU drops to 6.5 μA.

Timer1 is used to produce a 10-ms loop time for all of the main loop tasks. Timer1 is a 16-bit timer that counts from 0 up to 65,535 and then rolls over, back to 0 to

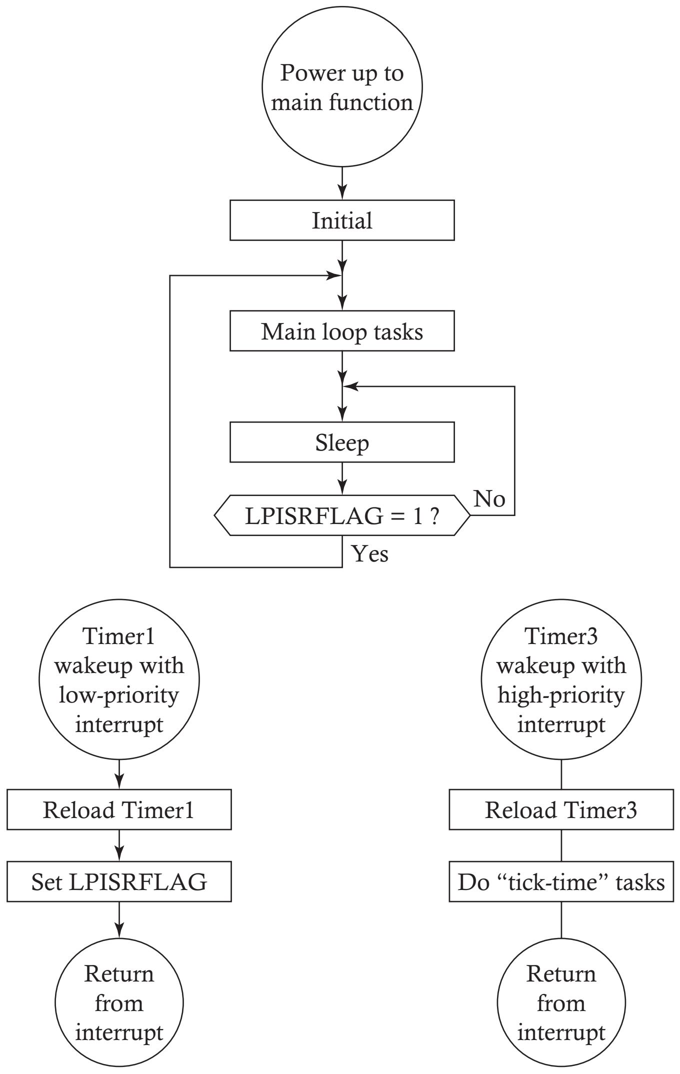


FIGURE 7-1 Reorganization to use Timer1 for controlling loop time and Timer3 for controlling faster recurring tasks

continue counting. The rollover produces a low-priority interrupt. Within the LPISR, Timer1 is reloaded with a number that cuts out all but 328 counts so that the next rollover will occur in

$$328\text{counts} \times \frac{1}{32{,}768\text{counts/s}} \approx \frac{1}{100}\,\text{s} = 10\text{ms}$$

Timer3 is set up to produce faster "tick time" interrupts in the same manner. For now, the task is to produce a 1-μs-positive pulse every 4 ms on an output pin. The pulse can be monitored with a scope or used to step the stepper motor of Chapter Eight at a rate of 250 steps/s.

The sequencing of the two interrupt service routines is illustrated in Figure 7-2. Note how the LPISR is executed every 10 ms, perhaps being briefly extended by a concurrent HPISR.

7.3 INTERRUPTS AND THE C18 COMPILER

When a low-priority interrupt occurs, the CPU sets aside its present state and loads its program counter with the low-priority interrupt vector address, 0x0018. The C18 compiler generates the code to begin execution of the **LoPriISR** shown in Figure 7-3. The code to set aside CPU registers upon entry to an interrupt service routine and to later restore CPU registers back to their state at the time of the interrupt is handled transparently by the compiler. To the writer of user code, each interrupt service routine is written with the same form as any other function. The difference lies in its being called by a hardware-initiated event (e.g., the setting of an interrupt flag by a timer).

7.4 TIMER1 OSCILLATOR

The Timer1 oscillator is a module that can function independently of the Timer1 counter. Its external circuitry consists of the 32,768-Hz crystal and two 15-pF capacitors of Figure 3-3. As mentioned in Section 3.7, this oscillator can be configured to operate reliably with V_{DD} = 3 V using the configuration selection

```
LPT1OSC = OFF
```

This choice uses a somewhat higher power driver in its oscillator circuit than is used by the circuit selected with the configuration choice

```
LPT1OSC = ON
```

The "OFF" choice also removes a 3-V regulator from the circuit, intended for use with a higher V_{DD} value. The "OFF" choice is needed for reliable operation with the Qwik&Low board's 3-V V_{DD} supply. The oscillator, together with its clocking of both Timer1 and Timer3 while the CPU sleeps, draws about 6.5 μA.

The Qwik&Low board can use the Timer1 oscillator for loop-time control. The crystal oscillator runs even as the rest of the chip sleeps, providing the timing accuracy associated with a 50 parts per million (i.e., ±0.005%) crystal. In contrast, use of the

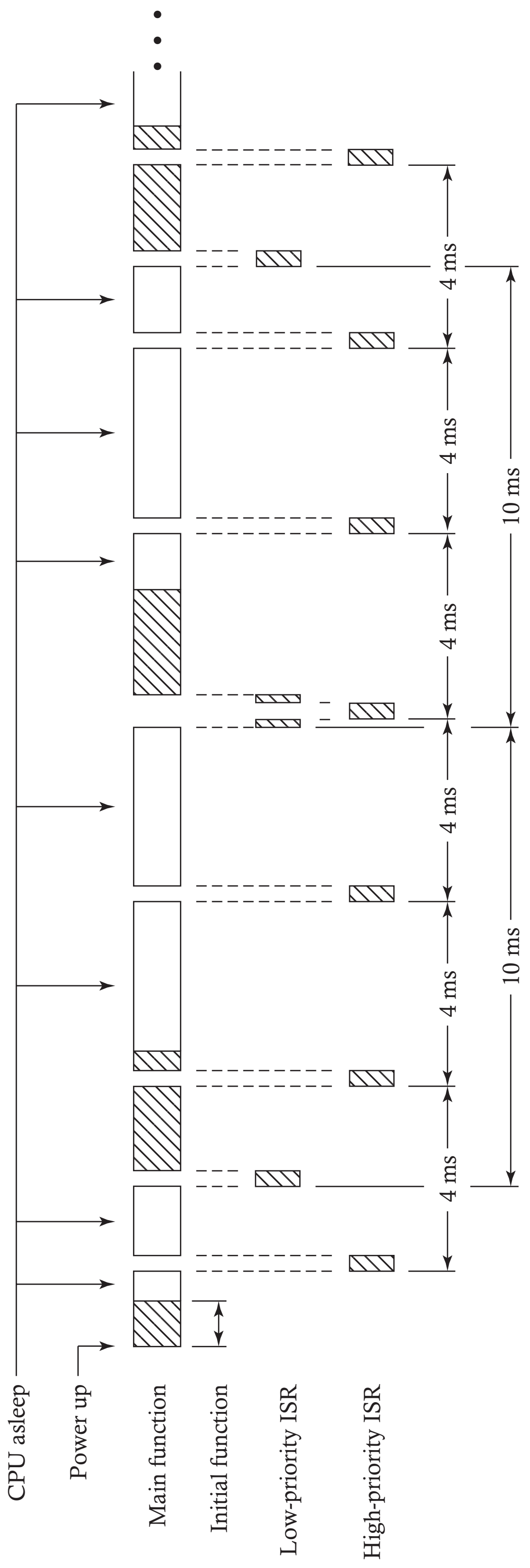


FIGURE 7-2 Sequencing of the two interrupt service routines

```
/*******************************
 * Interrupt vectors
 *******************************
 */
// For high priority interrupts:
#pragma code high_vector=0x08
void interrupt_at_high_vector(void)
{
  _asm GOTO HiPriISR _endasm
}
#pragma code
#pragma interrupt HiPriISR

// For low priority interrupts:
#pragma code low_vector=0x18
void interrupt_at_low_vector(void)
{
  _asm GOTO LoPriISR _endasm
}
#pragma code
#pragma interruptlow LoPriISR
```

(a) Handling of vectoring to HiPriISR and LoPriISR

```
/*******************************
 * HiPriISR
 *******************************
 */
void HiPriISR()
{

   <Tasks to be done>

}

/*******************************
 * LoPriISR
 *******************************
 */
void LoPriISR()
{

   <Tasks to be done>

}
```

(b) Interrupt service routines themselves

FIGURE 7-3 C18 compiler's handling of interrupt vectoring

watchdog timer for loop-time control saves \$1.35 in parts cost (the cost of the crystal and its two capacitors) and draws only 2.2 μA. However, using the watchdog timer

- Provides limited alternatives for a loop time (i.e., 4-ms minimum or some power of two greater than this).
- Does so with the much larger error specification of ±14%.
- Does not support fast counting as a side benefit.

7.5 TIMER1 COUNTER

The circuit of Figure 7-4 illustrates the use of the Timer1 counter in conjunction with the Timer1 oscillator. Each time the 16-bit **TMR1H:TMR1L** counter overflows, the **TMR1IF** flag in the **PIR1** register is set and, with the initialization shown, a low-priority interrupt occurs. The chip, which had been asleep, awakens and the low-priority interrupt service routine is executed. Two Timer1 housekeeping tasks must be carried out:

- **TMR1H:TMR1L** must be reinitialized to eliminate all but 328 counts until the next interrupt.
- The **TMR1IF** flag in the **PIR1** register must be cleared.

Because the low-priority interrupt automatically clears the low-priority global interrupt enable bit, **GIEL,** but not the **GIEH** bit, a high-priority interrupt can be accepted and acted on while the LPISR is being executed. When the CPU completes the execution of the LPISR, it automatically reenables low-priority interrupts by setting the **GIEL** bit and returns to the main loop and to the sleep instruction of Figure 7-1.

With its clock input from the 32,768-Hz Timer1 oscillator, Timer1 will not be incremented for 30.5 μs after the interrupt occurs. The CPU awakens immediately and begins executing machine instructions at a rate of 1 (or, at most, 2) μs per instruction. Unless a high-priority interrupt intervenes, Timer1 will be reinitialized before the counter is clocked again and any counts are missed. The loop time will be

$$\frac{328}{32{,}768} = 0.010009765 \text{ s} = 10 \text{ ms} + 0.1\%$$

If an intervening high-priority interrupt does occur in the few microseconds after Timer1 rolls over and interrupts the LPISR, it is unlikely to cause even one count of Timer1 to be lost. Even one lost count produces a loop time of

$$0.010009765 + 0.000030518 = 0.010040283 = 10 \text{ ms} + 0.4\%$$

Given the normal circumstance of the HPISR infrequently overlapping with the LPISR as in Figure 7-1, the accuracy of this loop-time mechanism should be close to 0.1%.

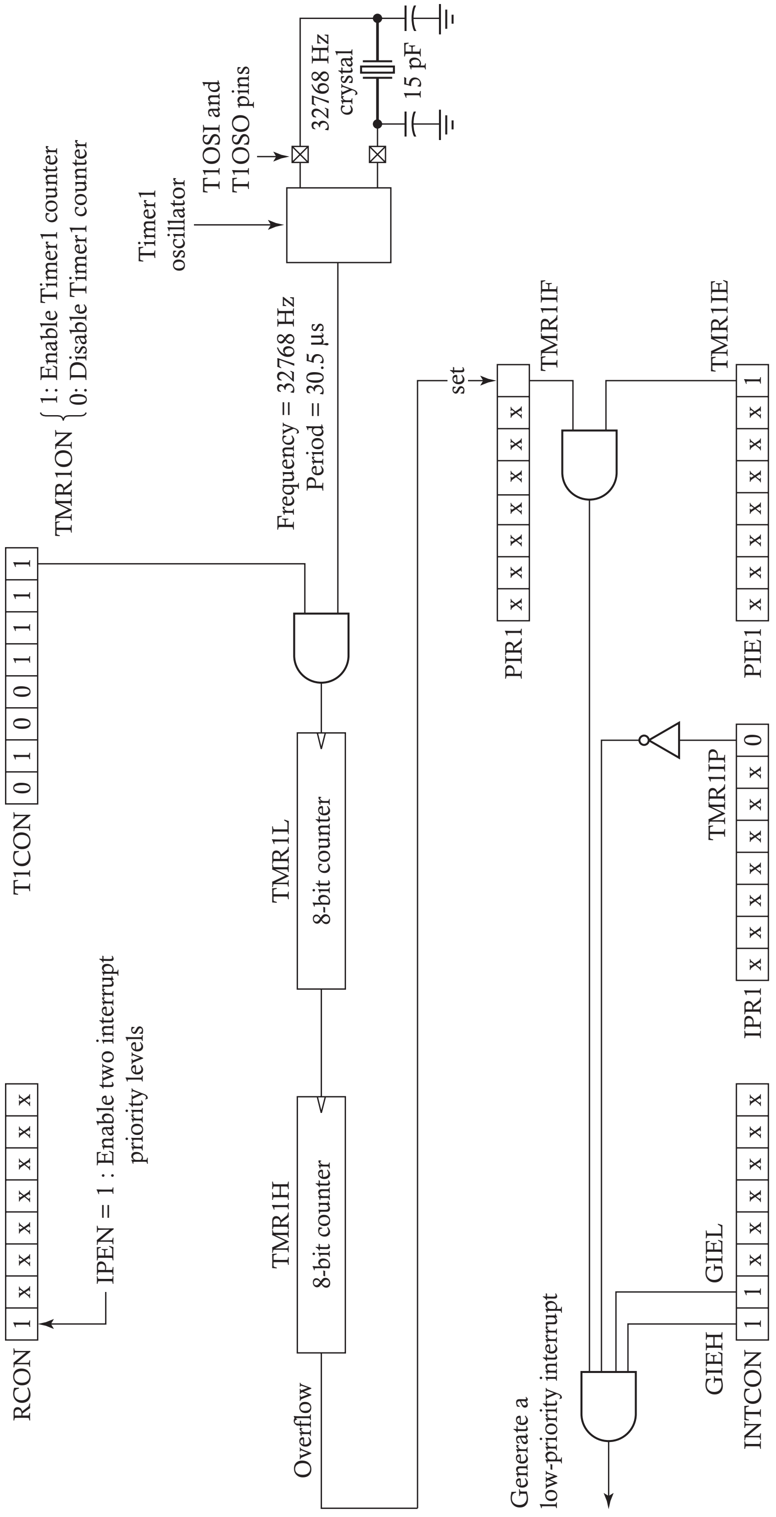


FIGURE 7-4 Timer1 use for controlling loop time

7.6 TIMER3 COUNTER

The Timer3 counter is identical in performance to Timer1. Its registers are illustrated in Figure 7-5. To have it generate a high-priority interrupt rather than Timer1's low-priority interrupt, it is only necessary to set its interrupt priority bit, **TMR3IP,** in the **IPR2** register.

Timer3 (as well as Timer1) has a **TMR3ON** bit in its **T3CON** control register that can be used to stop the counter for its reinitialization. It (as well as Timer1) also has the option of synchronizing the edges of the slow Timer1 oscillator to the edges of the faster $F_{OSC}/4$ CPU clock. The latter option is helpful if the counter is to be read or written to as it is being clocked without occasionally obtaining a garbled result. However, synchronization causes the counter to stop being clocked when the chip is put to sleep, even though the Timer1 oscillator is still running. Switching synchronization on and off also does not provide satisfactory operation because it results in lost counts. Using the option of stopping the counter, reinitializing it, and then starting it again is done instead.

7.7 THE T3.c TEMPLATE PROGRAM

The T3.c template program is listed in Figure 7-6. It differs from T2.c in the following ways:

- The addition of **LoopTime, HiPriISR** and **LoPriISR** function prototypes.
- The addition of the Interrupt vectors section.
- The creation of the **LoPriISR** and **HiPriISR** functions.
- The initialization of the Timer1 oscillator, Timer1, and Timer3 to run and to produce interrupts.
- The replacement of the **Sleep** macro in the **main** loop with the call of a **LoopTime** function.
- The addition of a **LoopTime** function that handles the wakeup from the **Sleep** instruction in one way for Timer1 interrupts and in another way for Timer3 interrupts.

PROBLEMS

7-1 LoopTime function The testing of the **LPISRFLAG** in the **LoopTime** function located at the end of the T3.c program of Figure 7-6 is used to exit from the *while* loop. Upon entry to the **LoopTime** function, the CPU immediately goes to sleep. It will remain asleep until either Timer1 or Timer3 causes an interrupt. As part of its execution, Timer1's LPISR will set the **LPISRFLAG** bit.

a) Describe the program flow when the chip is asleep and a Timer3 interrupt occurs.

b) Describe the program flow when the chip is asleep and a Timer1 interrupt occurs.

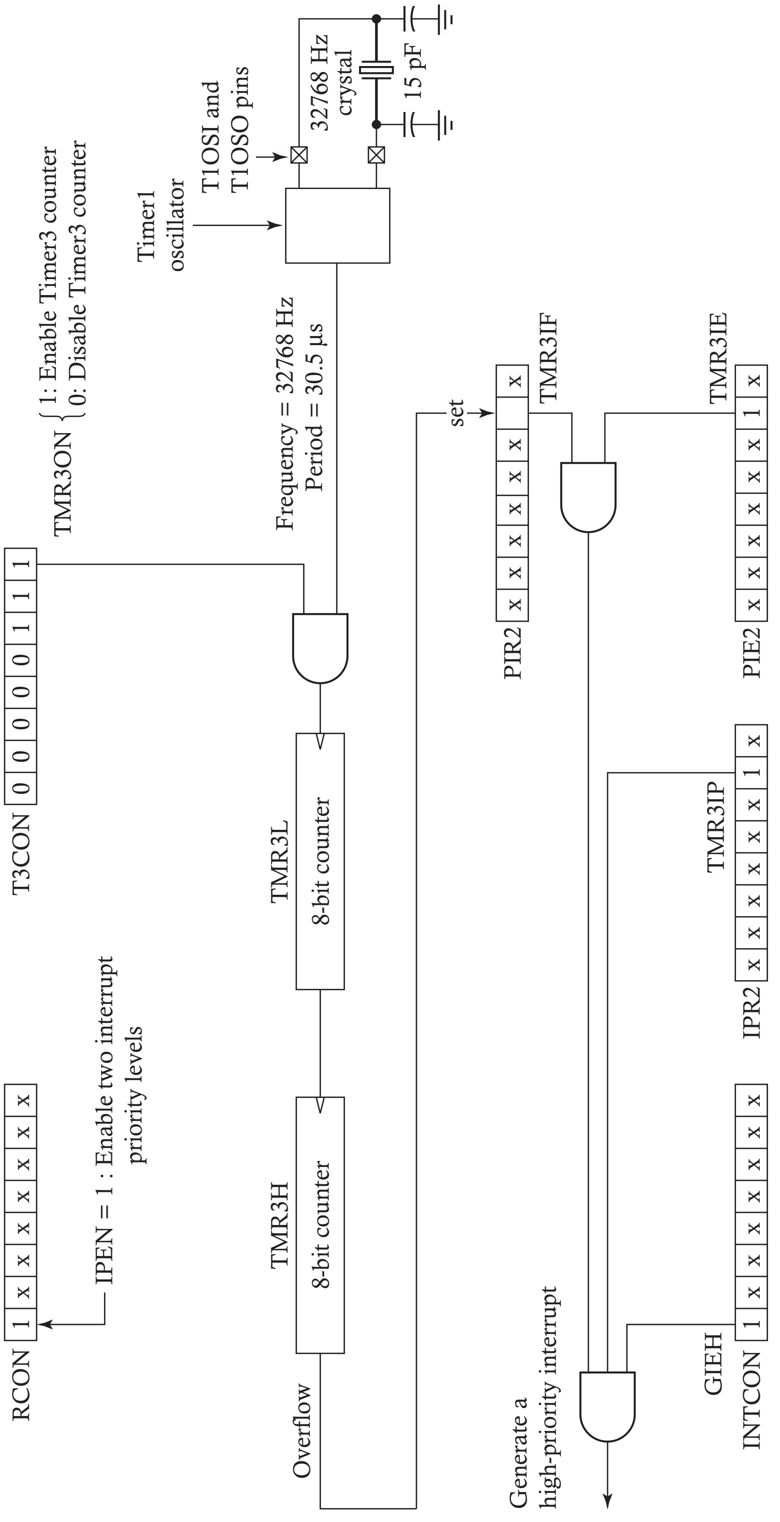


FIGURE 7-5 Timer3 use for generating fast ticks

```
/******* T3.c ******************
 *
 * Use Fosc = 4 MHz for Fcpu = Fosc/4 = 1 MHz.
 * Timer1 and Timer3 are both clocked by the Timer1 crystal oscillator.
 * Same mainline code as for T2.c.
 * LoopTime function puts chip to sleep.  Timer1 awakens chip every 10 ms
 * within LoopTime function.  CPU adjusts Timer1 content for it to timeout
 * after another ten milliseconds.
 * Timer3 generates high-priority interrupts every 4 ms to step motor.
 * Toggle RC2 output every 10 milliseconds for measuring looptime with scope.
 * Blink LED on RD4 for 10 ms every four seconds.
 * Post PRESS PB message on LCD until first pushbutton push.
 * Thereafter, increment and display LCD's CHAR0:CHAR1 every second
 * and increment and display LCD's CHAR3:CHAR4 for each pushbutton press.
 *
 *          Current draw = 31 uA (with LED and LCD switched off but whether
 *                               or not the stepper motor is connected.)
 *
 ******* Program hierarchy *****
 *
 * main
 *    Initial
 *       Display
 *    BlinkAlive
 *    Time
 *    Pushbutton
 *    UpdateLCD
 *       Display
 *    LoopTime
 *
 * LoPriISR
 *
 * HiPriISR
 *
 *******************************
 */

#include <p18f4321.h>           // Define PIC18LF4321 registers and bits
#include <string.h>             // Used by the LoadLCDSTRING macro

/*******************************
 * Configuration selections
 *******************************
 */
#pragma config OSC = INTIO1     // Use internal osc, RA6=Fosc/4, RA7=I/O
#pragma config PWRT = ON        // Enable power-up delay
#pragma config LVP = OFF        // Disable low-voltage programming
#pragma config WDT = OFF        // Disable watchdog timer initially
#pragma config WDTPS = 4        // 16 millisecond WDT timeout period, nominal
#pragma config MCLRE = ON       // Enable master clear pin
#pragma config PBADEN = DIG     // PORTB<4:0> = digital
```

FIGURE 7-6 T3.c template.

```
#pragma config CCP2MX = RB3     // Connect CCP2 internally to RB3 pin
#pragma config BOR = SOFT       // Brown-out reset controlled by software
#pragma config BORV = 3         // Brown-out voltage set for 2.0V, nominal
#pragma config LPT1OSC = OFF    // Deselect low-power Timer1 oscillator

/*******************************
 * Global variables
 *******************************
 */
char PBFLAG;                    // Flag, set after first press of pushbutton
char LCDFLAG;                   // Flag, set to send string to display
char NEWPB;                     // Flag, set if pushbutton is now pressed
char OLDPB;                     // Flag, set if pushbutton was pressed last loop
char LPISRFLAG;                 // Flag, set when LP interrupt has been handled
unsigned int ALIVECNT;          // Scale-of-400 counter for blinking "Alive" LED
unsigned int STEPCNT;           // 65536 - number of counts between steps
unsigned char TIMECNT;          // Scale-of-100 counter of loop times = 1 second
unsigned char UNITS;            // For display of seconds
unsigned char TENS;
unsigned char PBUNITS;          // For display of pushbutton count
unsigned char PBTENS;
unsigned char i;                // Index into strings
unsigned int DELAY;             // Sixteen-bit counter for obtaining a delay
char LCDSTRING[] = "PRESS PB "; // LCD display string

/*******************************
 * Function prototypes
 *******************************
 */
void Initial(void);
void BlinkAlive(void);
void Time(void);
void Pushbutton(void);
void UpdateLCD(void);
void Display(void);
void LoopTime(void);
void HiPriISR(void);
void LoPriISR(void);

/*******************************
 * Macros
 *******************************
 */
#define Delay(x) DELAY = x; while(--DELAY){ Nop(); Nop(); }
#define LoadLCDSTRING(lit)  strcpypgm2ram(LCDSTRING,(const far rom char*)lit)

/*******************************
 * Interrupt vectors
 *******************************
 */
```

FIGURE 7-6 *(continued)*

```
// For high priority interrupts:
#pragma code high_vector=0x08
void interrupt_at_high_vector(void)
{
  _asm GOTO HiPriISR _endasm
}
#pragma code
#pragma interrupt HiPriISR

// For low priority interrupts:
#pragma code low_vector=0x18
void interrupt_at_low_vector(void)
{
  _asm GOTO LoPriISR _endasm
}
#pragma code
#pragma interruptlow LoPriISR

/////// Main program //////////////////////////////////////////////////////////

/******************************
 * main
 ******************************
 */
void main()
{
   Initial();                    // Initialize everything
   while (1)
   {
      PORTCbits.RC2 ^= 1;        // Toggle pin, for measuring loop time
      BlinkAlive();              // Blink "Alive" LED
      Time();                    // Display seconds
      Pushbutton();              // Display pushbutton count
      UpdateLCD();               // Update LCD
      LoopTime();                // Use Timer1 to wakeup and loop again
   }
}

/******************************
 * HiPriISR
 *
 * This high-priority interrupt service routine creates a positive pulse on
 * RD1 every 4 ms.   Four milliseconds = 4000 * 0.032768 = 131 Timer3 counts.
 * Add STEPCNT = 65536+1-131 = 65406 counts to Timer3.
 * The +1 in the above equation results because the 0-to-1 transition when
 * Timer3 is reenabled increments Timer3.
 * RB0 is toggled to measure step period.
 ******************************
 */
void HiPriISR()
{
   T3CONbits.TMR3ON = 0;         // Disable clock input to Timer3
   TMR3L += STEPCNT;             // Add into lower byte
```

FIGURE 7-6 *(continued)*

```
   TMR3H = (TMR3H + STATUSbits.C) + (STEPCNT >> 8);  // and into upper byte
   T3CONbits.TMR3ON = 1;        // Reenable Timer3
   PIR2bits.TMR3IF = 0;         // Clear interrupt flag
   PORTDbits.RD1 = 1;           // Create 1 us wide positive pulse
   PORTDbits.RD1 = 0;           // to step motor
   PORTBbits.RB0 ^= 1;          // Toggle pin to measure step period
}

/*******************************
 * LoPriISR
 *
 * This low-priority interrupt service routine updates Timer1 to interrupt
 * every 10 ms.  Ten ms = 10000 * 0.032768 = 328 to cut out all
 * but 328 counts.  Add 65536+1-328 = 65209 = 0xFEB9 to Timer1 = 0x0000 (or
 * at most a count or two higher if the HPISR intervenes).
 *******************************
 */
void LoPriISR()
{
   T1CONbits.TMR1ON = 0;        // Pause Timer1 counter
   TMR1L += 0xB9;               // Cut out all but 328 counts of Timer1
   T1CONbits.TMR1ON = 1;        // Resume Timer1 counter
   TMR1H = 0xFE;                // Upper byte of Timer1 will be 0xFE
   PIR1bits.TMR1IF = 0;         // Clear interrupt flag
   LPISRFLAG = 1;               // Set a flag for LoopTime
}

/*******************************
 * Initial
 *
 * This function performs all initializations of variables and registers.
 *******************************
 */
void Initial()
{
   OSCCON = 0b01100010;         // Use Fosc = 4 MHz (Fcpu = 1 MHz)
   SSPSTAT = 0b00000000;        // Set up SPI for output to LCD
   SSPCON1 = 0b00110000;
   ADCON1 = 0b00001011;         // RA0,RA1,RA2,RA3 pins analog; others digital
   TRISA = 0b00001111;          // Set I/O for PORTA
   TRISB = 0b01000100;          // Set I/O for PORTB
   TRISC = 0b10000000;          // Set I/O for PORTC
   TRISD = 0b10000000;          // Set I/O for PORTD
   TRISE = 0b00000010;          // Set I/O for PORTE
   PORTA = 0;                   // Set initial state for all outputs low
   PORTB = 0;
   PORTC = 0;
   PORTD = 0b00100000;          // except RD5 that drives LCD interrupt
   PORTE = 0;
   SSPBUF = ' ';                // Send a blank to initialize state of UART
   Delay(50000);                // Pause for half a second
   RCONbits.SBOREN = 0;         // Now disable brown-out reset
   PBFLAG = 0;                  // Clear flag until pushbutton is first pressed
```

FIGURE 7-6 *(continued)*

```
   LCDFLAG = 0;                 // Flag to signal LCD update is initially off
   LPISRFLAG = 0;               // Flag to signal that LPISR has been executed
   TIMECNT = 0;                 // Reset TIMECNT
   TENS = '5';                  // Initialize to 59 so first display = 00
   UNITS = '9';
   PBTENS = '0';                // Initialize count of pushbutton presses
   PBUNITS = '1';
   ALIVECNT = 300;              // Blink immediately
   STEPCNT = 65406;             // Cut out all but 131 counts of Timer3
   OLDPB = 0;                   // Initialize to unpressed pushbutton state
   T1CON = 0b01001111;          // Timer1 - loop time via low-pri interrupts
   T3CON = 0b00000111;          // Timer3 - step motor via hi-pri interrupts
   TMR1H = 0xFE;                // Set Timer1 to be 10 ms away from
   TMR1L = 0xB9;                // next roll over (65536 + 1 - 328 = 0xFEB9)
   TMR3H = 0xFF;                // Set Timer3 to be 4 ms away from
   TMR3L = 0xAE;                // next roll over (65536 + 1 - 131 = 0xFF7E)
   PIE1bits.TMR1IE = 1;         // Enable local interrupt source
   PIE2bits.TMR3IE = 1;         // Enable local interrupt source
   IPR1bits.TMR1IP = 0;         // Use Timer1 for low-priority interrupts
   IPR2bits.TMR3IP = 1;         // Use Timer3 for hi-priority interrupts
   RCONbits.IPEN = 1;           // Enable high/low priority interrupt feature
   INTCONbits.GIEL = 1;         // Global low-priority interrupt enable
   INTCONbits.GIEH = 1;         // Enable both high and low interrupts
   Display();                   // Display initial "PRESS PB" message
   LoadLCDSTRING("00 01    ");  // Reinitialize LCDSTRING
}

/*******************************
 * BlinkAlive
 *
 * This function briefly blinks the LED every four seconds.
 * With a looptime of about 10 ms, count 400 looptimes.
 ******************************
 */
void BlinkAlive()
{
   PORTDbits.RD4 = 0;           // Turn off LED
   if (++ALIVECNT == 400)       // Increment counter and return if not 400
   {
      ALIVECNT = 0;             // Reset ALIVECNT
      PORTDbits.RD4 = 1;        // Turn on LED for one looptime
   }
}

/*******************************
 * Time
 *
 * After pushbutton is first pushed, display seconds.
 ******************************
 */
```

FIGURE 7-6 *(continued)*

```
void Time()
{
   if (PBFLAG)                  // After pushbutton is first pushed,
   {
      if (++TIMECNT == 100)     //  count TIMECNT to 1 second
      {
         TIMECNT = 0;           // Reset TIMECNT for next second
         if (++UNITS > '9')     // and increment time
         {
            UNITS = '0';
            if (++TENS > '5')
            {
               TENS = '0';
            }
         }
         LCDSTRING[0] = TENS;   // Update display string
         LCDSTRING[1] = UNITS;
         LCDFLAG = 1;           // Set flag to display
      }
   }
}

/*******************************
 * Pushbutton
 *
 * After pushbutton is first pressed, display pushbutton count.
 *******************************
 */
void Pushbutton()
{
   PORTEbits.RE0 = 1;           // Power up the pushbutton
   Nop();                       // Delay one microsecond before checking it
   NEWPB = !PORTDbits.RD7;      // Set flag if pushbutton is pressed
   PORTEbits.RE0 = 0;           // Power down the pushbutton
   if (!OLDPB && NEWPB)         // Look for last time = 0, now = 1
   {
      if (!PBFLAG)              // Take action for very first PB press
      {
         PBFLAG = 1;
         ALIVECNT = 399;        // Synchronize LED blinking to counting
         TIMECNT = 99;          // Update display immediately
      }
      else                      // Take action for subsequent PB presses
      {
         if (++PBUNITS > '9')   // and increment count of PB presses
         {
            PBUNITS = '0';
            if (++PBTENS > '9')
            {
               PBTENS = '0';
            }
```

FIGURE 7-6 *(continued)*

```
         }
      }
      LCDSTRING[3] = PBTENS;    // Update display string for simulated LCD
      LCDSTRING[4] = PBUNITS;
      LCDFLAG = 1;              // Set flag to display
   }
   OLDPB = NEWPB;               // Save present pushbutton state
}

/******************************
 * UpdateLCD
 *
 * This function updates the 8-character LCD if Time
 * or Pushbutton has set LCDFLAG.
 ******************************
 */
void UpdateLCD()
{
   if(PBFLAG && LCDFLAG)
   {
      Display();
      LCDFLAG = 0;
   }
}

/******************************
 * Display
 *
 * This function sends LCDSTRING to the LCD.
 ******************************
 */
void Display()
{
   PORTDbits.RD5 = 0;           // Wake up LCD display
   for (i = 0; i < 9; i++)
   {
      PIR1bits.SSPIF = 0;       // Clear SPI flag
      SSPBUF = LCDSTRING[i];    // Send byte
      while (!PIR1bits.SSPIF);  // Wait for transmission to complete
   }
   PORTDbits.RD5 = 1;           // Return RB5 high, ready for next string
}

/******************************
 * LoopTime
 *
 * This function puts the chip to sleep upon entry.
 * For a Timer3 interrupt, it executes the HPISR and then returns to sleep.
 * For a Timer1 interrupt, it executes the LPISR and then exits.
 ******************************
 */
```

FIGURE 7-6 *(continued)*

```
void LoopTime()
{
   while (!LPISRFLAG)          // Sleep upon entry and upon exit from HPISR
   {                           // Return only if LPISR has been executed,
       Sleep();                //  which sets LPISRFLAG
       Nop();
   }
   LPISRFLAG = 0;              // Sleep upon next entry to LoopTime
}
```

FIGURE 7-6 *(continued)*

c) Describe the program flow when the chip is awakened by a Timer1 interrupt but a Timer3 interrupt intervenes just before the LPISR sets the **LPISRFLAG.**

7-2 Timer1 interrupt interval When the low-priority interrupt service routine cuts out all but 328 counts in its count sequence, it rolls over approximately every 10 ms. As pointed out in Section 7.5, the exact time is 10,009.765 µs.

a) How many parts per million is this off from the nominal 10 ms?

b) How does this compare with the 50-ppm accuracy of the crystal oscillator?

c) Repeat parts (a) and (b) if the number of counts of the Timer1 oscillator between Timer1 rollovers were reduced to 327.

7-3 Effect of stopping-starting Timer1 Consider that the output of the Timer1 oscillator is a squarewave, and that the Timer1 counter is clocked on the rising edge of this squarewave. With an oscillator period of about 30.5 µs, the oscillator output will be high for about 15 µs after the rising edge that produced the low-priority interrupt. During that time, the CPU wakes up and switches on its F_{CPU} = 1 MHz clock. It takes several microseconds to set aside the program counter and a few other CPU registers before clearing the **TMR1ON** bit to block clock edges from reaching the Timer1 counter. It takes 2 µs to update **TMR1L** before setting the **TMR1ON** bit again. As shown in Figure 7-4, both the **TMR1ON** bit and the Timer1 oscillator's squarewave output are inputs to an AND gate whose output clocks the Timer1 counter.

a) Draw a timing diagram showing the inputs and output of the AND gate assuming the clearing and setting of **TMR1ON** takes place during the first 15 µs after the Timer1 rollover that caused the interrupt.

b) Now repeat this, assuming a high-priority interrupt intervenes and delays the entry into the LPISR from the time of the Timer1 rollover by

$$30.5 \times n + 15\ \mu s, \text{ where n is 0 or 1 or 2.}$$

c) If the LPISR adds 65,536 + 1 – 328 = 65,211 to whatever number is in the Timer1 counter, how many Timer1 oscillator periods will occur between the last Timer1 rollover and the next one? Answer this for both parts (a) and (b).

7-4 Reinitializing Timer1 versus adding into it The **LoPriISR** of Figure 7-6, the T3.c program, adds 0xFEB9 to Timer1. A slightly simpler procedure would be to load 0xFEB9 into Timer1. In both cases, assume **TMR3ON** = 0 when the low byte, **TMR1L,** of Timer1 is updated.

a) If this adding or loading takes place within 15 μs of when Timer1 rolled over, what will be the time until the next rollover in each case?

b) Now assume that the adding or loading takes place

$$31.5 \times 2 + 5 = 68\ \mu s$$

after the Timer1 rollover, delayed by an exceptionally long intervening high-priority interrupt service routine. What will be the time until the next rollover in each case?

7-5 Effect of 8-bit add Instead of adding 0xFEB9 to **TMR1H:TMR1L,** the **LoPriISR** of the T3.c template program simply *adds* 0xB9 to **TMR1L** and *loads* 0xFE into **TMR1H.** The possible clocking of **TMR1L** is prevented by making **TMRON** = 0 during the addition on the chance that the CPU and the Timer1 oscillator might both try to change **TMR1L** at the same time. No such issue arises for **TMR1H,** which will contain 0x00 after the rollover and until 0x100 – 0xB9 counts of the Timer1 oscillator's clock periods have occurred.

a) How long is this?

b) Using the simplified update scheme for TMR1H:TMR1L of the LoPriISR at any time short of this will produce what interval between the last interrupt and the next one? Explain.

7-6 Worst-case overlapping of interrupts The machine code generated by the line

```
TMR1L += 0xB9
```

consists of a 1-μs-long instruction to load 0xB9 into a CPU register followed by a second 1-μs-long instruction to add the CPU register to **TMR1L.** An interrupt cannot break into the middle of the execution of an instruction. Hence, were the HPISR to interrupt during the execution of the LPISR's execution of the *add* instruction,

a) What would be the effect on the addition?

b) What would be the effect on the counting of Timer1 oscillator clock edges if the HPISR that causes the CPU to digress from the LPISR at this precise point in its execution takes 20 μs to execute?

c) What is the percent chance of a randomly occurring high-priority interrupt producing the effect of (b), given that the LPISR is executed every 10 ms = 10,000 μs?

Chapter

8

STEPPER MOTOR CONTROL

8.1 OVERVIEW

A stepper motor is a low-cost, high-resolution positioning actuator. Though it would not seem to fit in an environment powered solely by the coin cell of the Qwik&Low board, a $5-motor can be powered by a $5-wall transformer as shown in Figure 8-1 and driven with signals from the Qwik&Low board. The role of the driver chip is filled by a $3-part manufactured by Allegro Microsystems, a company with a long history of building sophisticated stepper-motor logic and current-switching technology into a chip. The resulting combination produces an actuator with a resolution of 200 steps per revolution that can be stepped at rates up to 800 steps/s or so.

8.2 STEPPER-MOTOR OPERATION

Taking apart a 200 steps per revolution stepper motor illuminates its operation. Figure 8-2a shows that its rotor consists of two sets of laminations separated by a thin doughnut-shaped permanent magnet. Each set of laminations has 50 teeth that are offset by one-half of a tooth from the other set. The net result is to produce a low-cost rotor consisting of 50 N-S pole pairs. The two sets of laminations and their permanent magnet are mounted on a motor shaft with ball bearings on each end for support within the stepper-motor case.

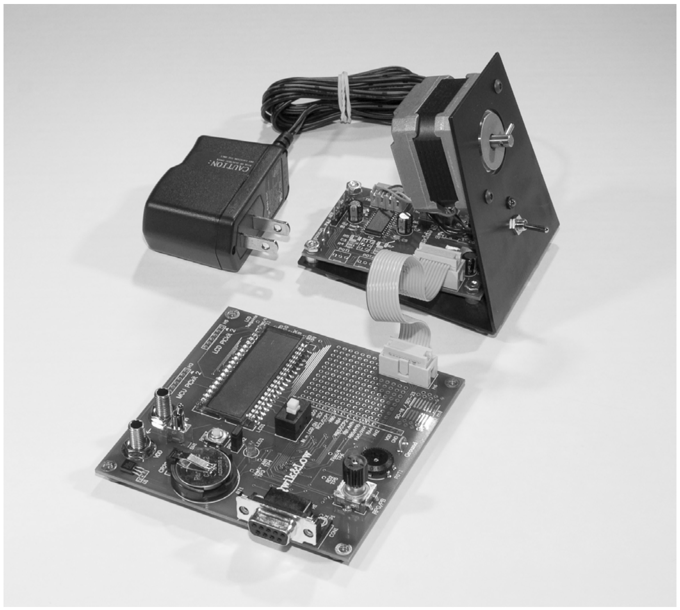


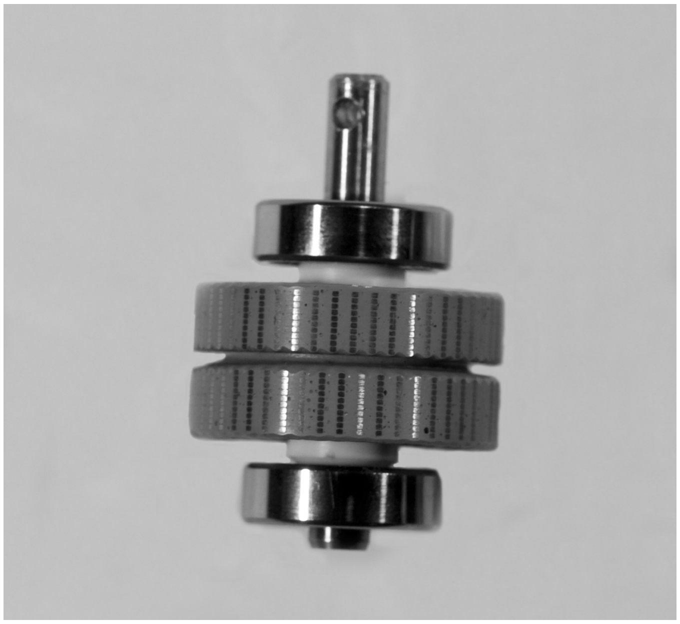
FIGURE 8-1 Stepper motor addition to the Qwik&Low board

The stator of the stepper motor is shown in Figure 8-2b. It consists of eight windings on eight poles, with every other winding connected together but wound in the opposite direction. Thus, if the windings are numbered in order, 1 – 2 – 3 – 4 – 5 – 6 – 7 – 8, then a current into 1 – 3 – 5 – 7 that makes winding number 1 a north pole will make 3 a south pole, 5 a north pole, and 7 a south pole. This is illustrated in Figure 8-3a. Backtracking to the photo of Figure 8-2b, note that each pole is broken into six teeth. These teeth have the same pitch as the 50 teeth of a set of rotor laminations. Furthermore, the 6 teeth on one pole are offset by 1.25 teeth from the 6 teeth on an adjacent pole. It is this 1.25-tooth offset that produces the offset shown in Figure 8-3a, where stator poles 1 and 5 and stator poles 3 and 7 are shown aligned with the rotor while stator poles 2 and 4 and stator poles 6 and 8 are shown offset by half a tooth.

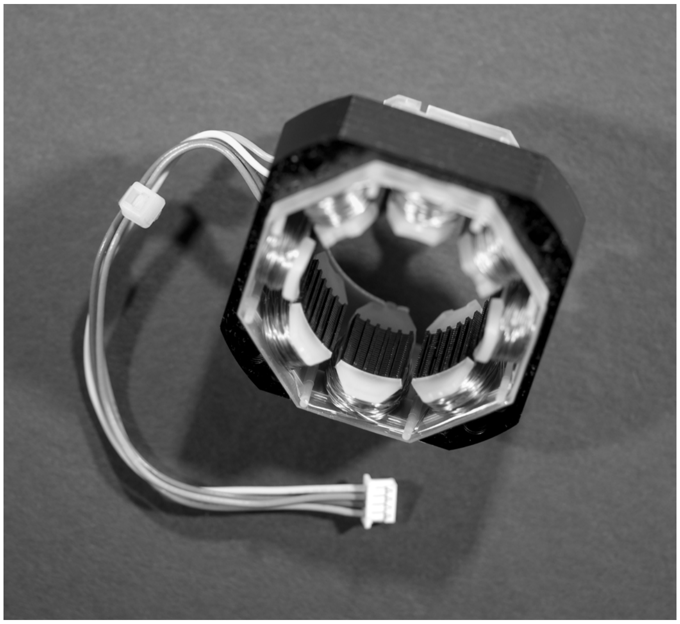
A *full step* is illustrated by the change between the winding energization of Figure 8-3a and that of Figure 8-3b. For this step, the winding current in 1 – 3 – 5 – 7 is turned off while the winding current in 2 – 4 – 6 – 8 is turned on. In response, the rotor rotates one full step. Figures 8-3b, c, d, and e illustrate the sequence of four steps that return to the winding excitation of Figure 8-3a. Fifty of these four-step sequences will result in the stepper motor turning one revolution. This accounts for the motor being designated a 200 steps per revolution motor.

A 200 steps per revolution motor can be made to step 400 steps per revolution by means of half-step sequencing of the winding currents. Let “A” represent a current into the 1 – 3 – 5 – 7 windings while “ $\overline{A}$” represents the reverse current into the same windings. Let “B” and “ $\overline{B}$” do the same for current into the 2 – 4 – 6 – 8 windings. Full stepping consists of sequencing the windings as follows:

… A B $\overline{A}$ $\overline{B}$ A B $\overline{A}$ $\overline{B}$ A …

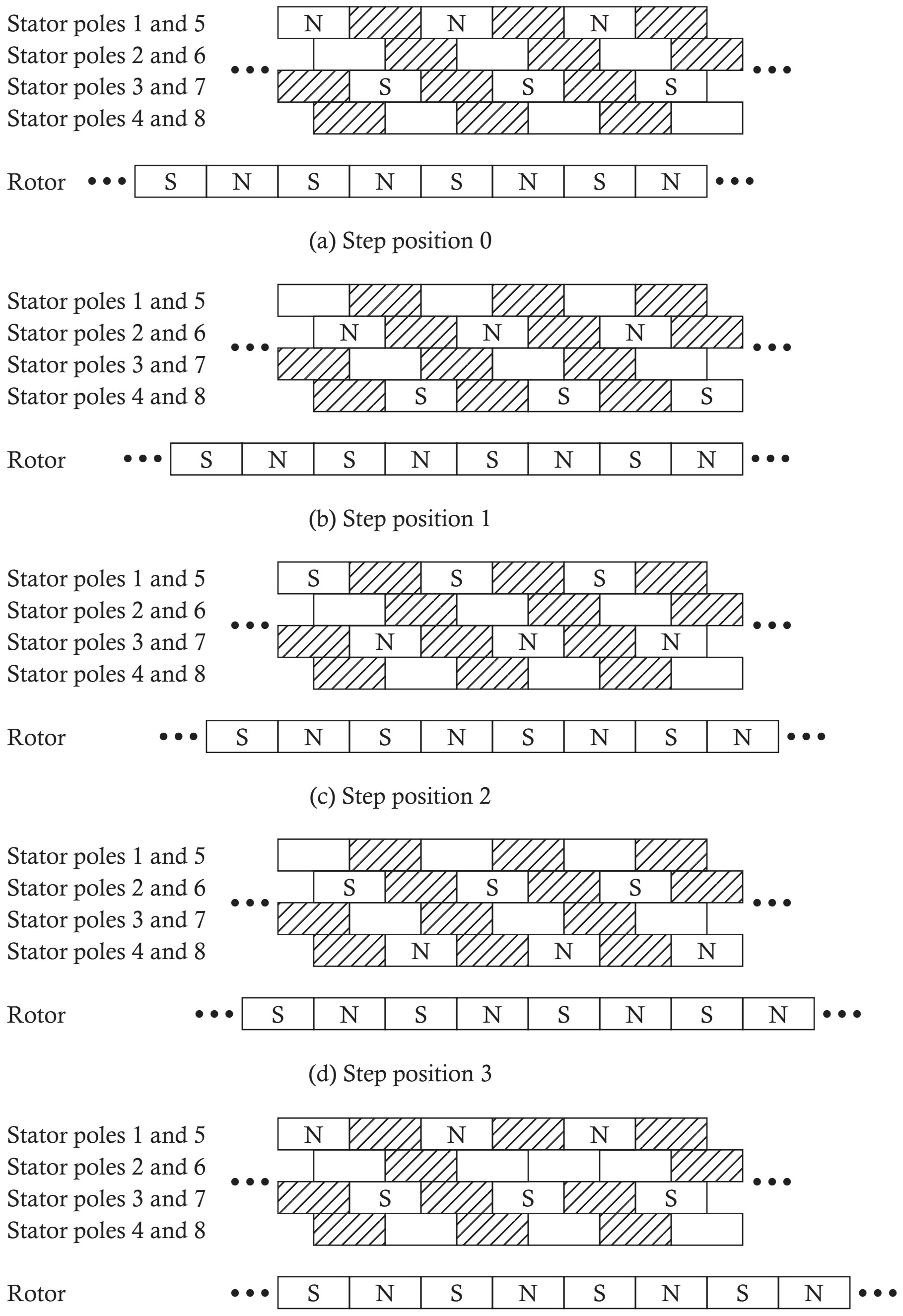


(a) Rotor



(b) Stator

FIGURE 8-2 Stepper motor



(a) Step position 0

(b) Step position 1

(c) Step position 2

(d) Step position 3

(e) Step position 4 (same as step position 0)

FIGURE 8-3 Stepper motor operation

where the winding excitation repeats every four steps. Half stepping is produced by the sequence:

$$\ldots\ A\quad AB\quad B\quad \overline{A}B\quad \overline{A}\quad \overline{A}\,\overline{B}\quad \overline{B}\quad A\overline{B}\quad A\ \ldots$$

Here, the winding excitation does not repeat before eight half steps have been taken. An alternate, popular mode of full stepping makes use of the sequence

$$\ldots\ AB\quad \overline{A}B\quad \overline{A}\,\overline{B}\quad A\overline{B}\quad AB\ \ldots$$

The sophisticated Allegro Microsystems A3967 stepper-motor driver chip used in Figure 8-1 has two logic inputs that allow a user to select any of four stepping modes: full stepping, half stepping, quarter stepping, and eighth stepping. It does so by controlling not only the direction of the A and B currents but also their magnitude. When half stepping is selected, a step position with only one winding energized uses 100.0% of the current established by the current-sensing resistor while a step position with two windings energized uses 70.7% of that current for each winding. When full stepping is selected, each winding is energized with the 70.7% of the current. Because the magnetic flux produced is the same under all circumstances for all four stepping modes, the torque produced at each step is the same. The effect is to help reduce vibration and noise.

8.3 BIPOLAR VERSUS UNIPOLAR STEPPER MOTORS

Figure 8-4a illustrates in crude form the excitation of a winding with current in either direction. Figure 8-4b shows one split winding of a *unipolar* stepper motor. It is driven by applying the motor supply voltage to the center tap and using either of two transistor switches to ground one side or the other. Six leads are brought out of the motor, three for each winding.

Contrast this with a modern *bipolar* stepper motor like that of Figures 8-1 and 8-2 for which each winding must be driven with current in either direction. An *H-bridge* driver such as that of Figure 8-4c provides the solution for reversing the current for each winding. If the upper left and bottom right transistor switches are both turned on, the A current will flow. In like manner, if the upper right and lower left transistor switches are both tuned on, the $\overline{A}$ current will flow. If all transistor switches are turned off, the current is cut off.

All high-performance stepper motors are built as bipolar stepper motors for several reasons:

- For the same number of turns of wire on each pole and the same current in each winding, a bipolar motor produces twice the magnetic flux of that produced by a unipolar motor, where only half of the winding is used at any one time. The larger flux means more torque is produced for each step, thus producing a higher maximum stepping rate.
- Reversing the current in a winding is more aggressively addressed with an H-bridge driver than with a unipolar driver. In both cases, the inductance of a

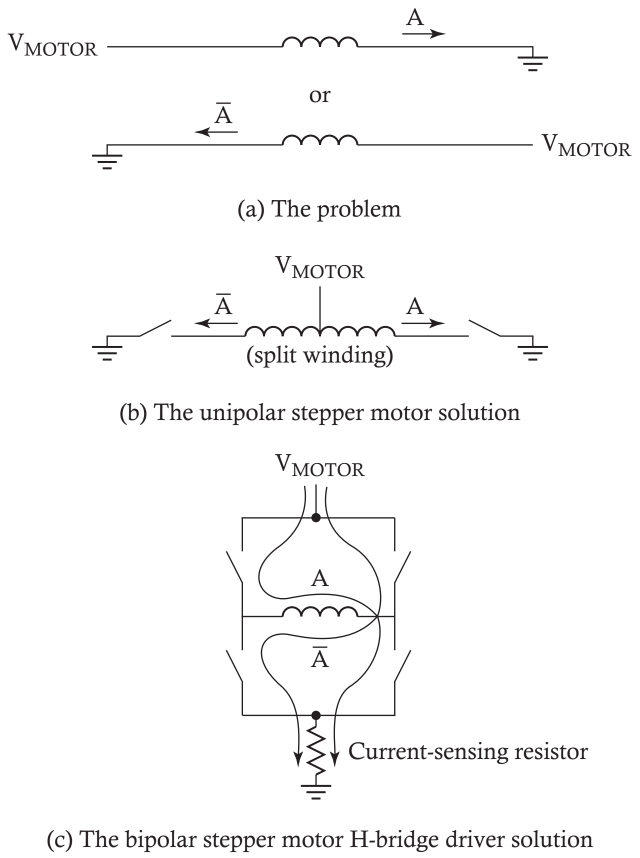


FIGURE 8-4 Two solutions to reversing the current in a stepper motor winding

winding impedes the change. A unipolar driver typically allows the current in a turned-off winding to decay through a diode-resistor circuit. A tradeoff is made between speed of decay and maximum breakdown voltage of the opened transistor switch that confronts the back emf of the collapsing magnetic field in the winding. In contrast, a bipolar driver chip such as Allegro's A3967 deals with current control and the breakdown voltage limits of its output transistors as an integrated whole. With all four leads of the bipolar stepper motor connected only to the A3967, the controller chip has full control of the winding voltages and currents.

8.4 STEPPER-MOTOR DRIVER

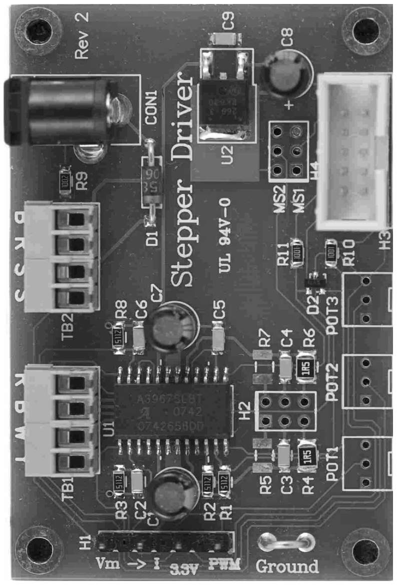
The stepper-motor driver board (designed for use in conjunction with the Qwik&Low board and available as an option) is shown in Figure 8-5a and its circuit in Figure 8-5b. The board includes (unpopulated) options for varying the drive parameters that are controlled by the $3 Allegro A3967 driver chip. The chip itself can handle a motor supply voltage as high as 30 V and a maximum current per winding of ±750 mA.

The \$5 regulated 12-V DC wall transformer shown in Figure 8-1 will supply a maximum current of 250 mA per winding. This supply can be replaced by a supply of up to 20 V @ 1.5 A (the 1.5-A limit is imposed by the A3967; the 20-V limit is imposed by the MC33269 voltage regulator). The provided supply is fine for the NEMA Size 17 (≈ 1.7 sq. in.) stepper motor shown in Figure 8-1.

The circuit shown in Figure 8-5b derives a 3.3-V supply from the 12-V input. This supplies the 50-mA current draw of the logic circuitry of the A3967. A 3.3-V regulator ensures that the logic inputs from the Qwik&Low board will never exceed this supply voltage and yet will exceed the $0.7 \times 3.3\ \text{V} = 2.3\ \text{V}$ minimum logic 1 voltage. The resulting current draw on the Qwik&Low coin cell is essentially zero when the two inputs, "Step" and "Dir", are low. When the inputs are both high, the current draw by the two 51.1-kΩ pull-down resistors is 120 μA. Consequently, the outputs from the Qwik&Low board that drive these two inputs should be kept low except for the few microseconds needed to control them for each step.

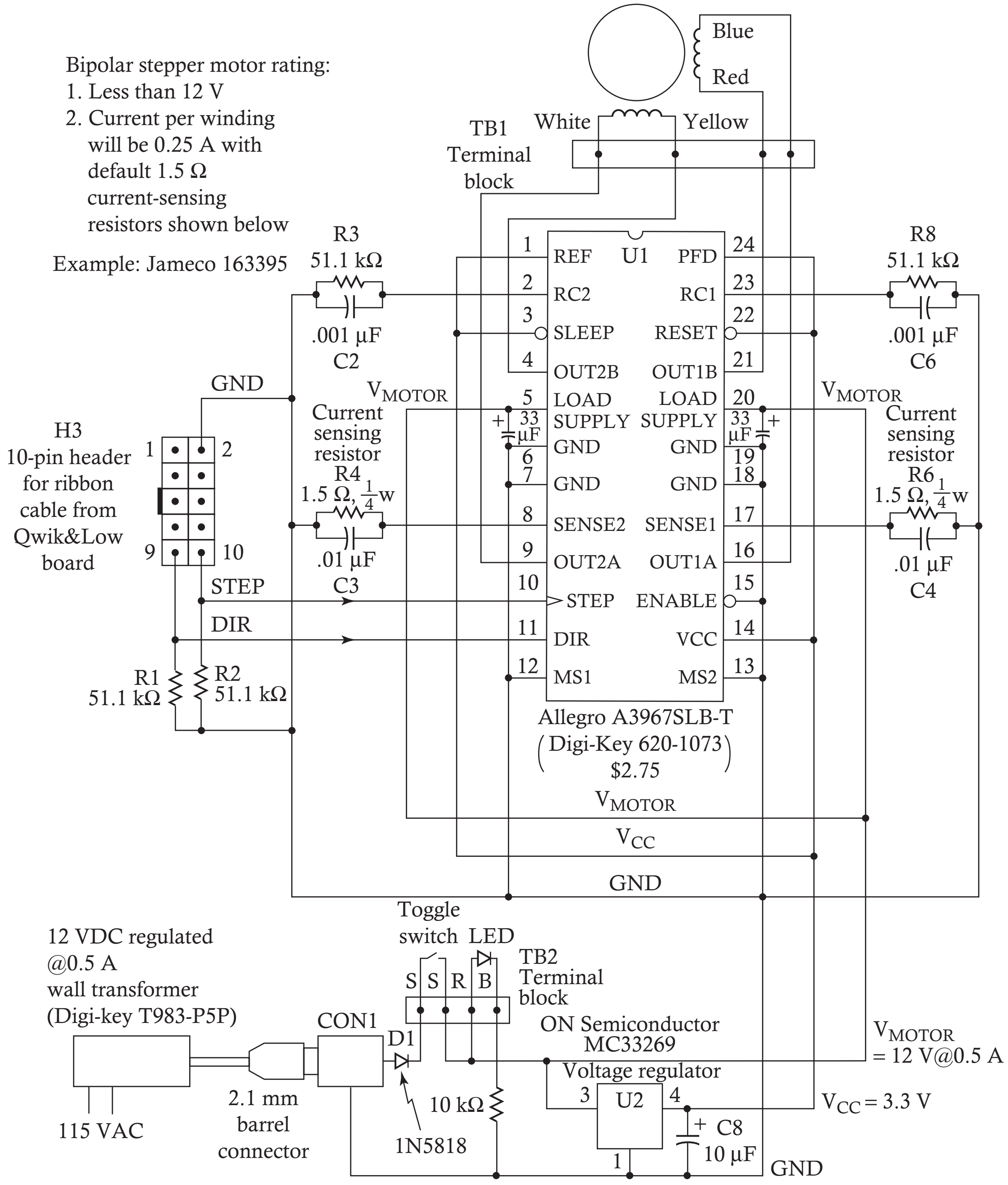
The 1.5-Ω resistors shown in Figure 8-5b are used to set the "100%" current level in each of the stepper motor's two windings; that is, the current level set when one winding is being driven while the other winding is turned off, as occurs with every other half step described in Section 8.2. This current is specified by Allegro to be

$$100\%\ \text{current} = \frac{V_{ref}}{8\ Rs} \qquad \textbf{(8-1)}$$



(a) Board

FIGURE 8-5 Stepper motor driver board



(b) Circuit (not shown are unpopulated options)

FIGURE 8-5 *(continued)*

where V_{ref} is the voltage connected to pin 1 of the chip, namely the regulated 3.3-V supply voltage. With the 1.5-Ω (1/4-W) resistor on the board, the 100% current has been set to a quarter of an ampere, in deference to the 0.5-A power supply.

The Allegro chip pulse-width-modulates the current to each motor winding, turning on the two diagonal transistor switches of the H-bridge driver when the current is low and opening the upper transistor switch when the current has risen to the 100% level. The RC circuits connected to RC1 and RC2 control the PWM frequency.

The driver board includes the provision for replacing the resistor in each RC circuit with a one-turn trimpot, should a user wish to experiment with the PWM frequency. The board also includes the option of installing alternate current-setting resistors (alternate to the 1.5-Ω resistors) plus two jumpers to select one pair or the other. Another optional potentiometer can be connected to the PFD pin to control how fast the current in a winding decays for a half step from 100% current to 70.7% current or from 70.7% to 0% current. A third option is to add jumpers to the MS1 and MS2 inputs that allow the default full-stepping mode to be replaced by half stepping, quarter stepping, or one-eighth stepping. Or the MS1 and MS2 pins can be connected to pins on the 10-pin header connected to the Qwik&Low board to allow this choice to be made by the MCU. These options provide refinements to what is, by default, an excellent and clean application of Allegro's superb driver chip. For more information on the stepper-motor driver board, refer to Appendix A4.

8.5 STEPPING

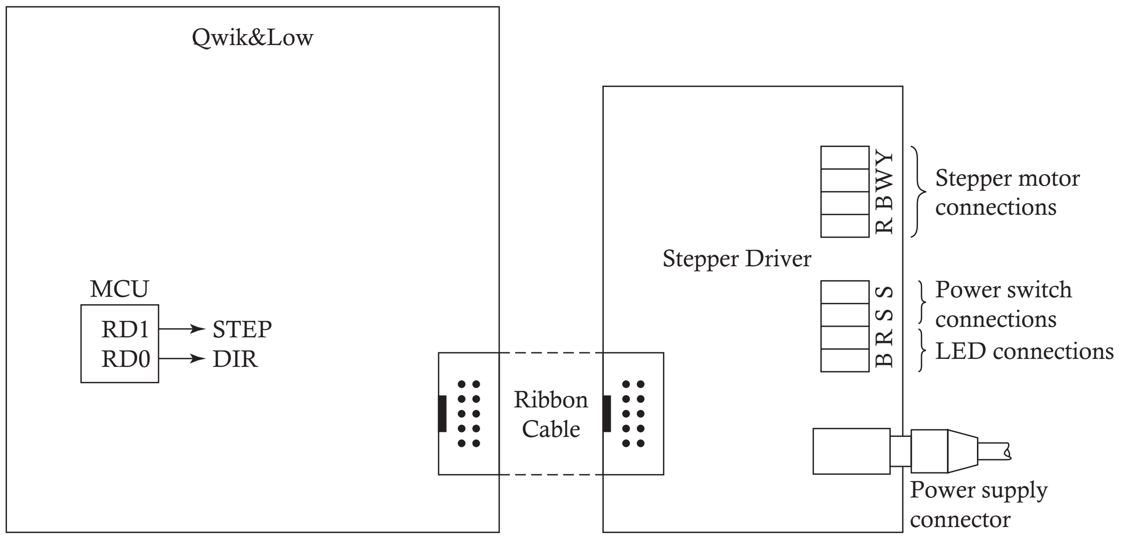
Stepper motors are widely used as open-loop positioning actuators because of being able to reach a given position by counting steps from the present position. Furthermore, the error in step position never accumulates; that is, 563 steps CW followed by 563 steps CCW will return the motor output to the same position from which it started, as long as no steps are lost by stepping too fast.

With the 10-pin ribbon cable connected between the Qwik&Low board and the stepper-motor controller board as shown in Figure 8-6a, the two MCU pins, **RD0** and **RD1**, control stepping, as indicated in Figure 8-6b. Thus to take a CW step, the following sequence is executed:

```
PORTDbits.RD0 = 1;          //Clockwise
Nop();                      //Pause 1µs
PORTDbits.RD1 = 1;          //Take step
PORTDbits.RD0 = 0;          //Zero the pulldown resistor currents
PORTDbits.RD1 = 0;
```

A CCW step is simpler:

```
PORTDbits.RD1 = 1;          //Take step
PORTDbits.RD1 = 0;          //Zero the pulldown resistor current
```



(a) Connections

$$\text{PORTDbits.RD1} = 0 \rightarrow 1: \quad \text{Take a step}$$

$$\text{PORTDbits.RD0} = \begin{cases} 1: & \text{CW} \\ 0: & \text{CCW} \end{cases}$$

(b) Control by MCU

FIGURE 8-6 Stepper motor connections and control

PROBLEMS

8-1 Stepper-motor operation Consider Figure 8-3a and also the photograph of Figure 8-2b.

a) On the photograph, label the poles that are north poles and those that are south poles.
b) Now do the same with a different color for the excitation of Figure 8-3b.

This illustrates how the stator north and south poles move. Given the 1.25-tooth offset between adjacent stator poles, it explains the movement of the rotor by one-quarter of the distance from one of its north poles to an adjacent north pole.

8-2 Quarter stepping To get to the data sheet for Allegro Microsystem's A3967 stepper-motor driver chip, Google "A3967". Within the data sheet, find its specification for the two winding currents at each quarter-step position.

a) What is maximum current in a winding relative to the "100% value" of Equation 8-1?

b) If used with the 12 V @ 0.5 A regulated supply of Figure 8-5b, what current-sensing resistors should be used to limit the supply current to the 0.5 A of the power supply?

8-3 Current-sensing resistor The stepper-motor driver board of Figure 8-5 uses, by default, the full-stepping mode described at the close of Section 8.2. As pointed out there, each stepper winding is energized with a current that is 70.7% of the "100%" current set by the current-sensing resistor via Equation 8-1 of Section 8.4. Consequently, another resistor might be used to control the current that drives the stepper motor harder.

a) Given the 12 V @ 0.5 A regulated supply that must supply current for two windings, what resistance value is optimum?

b) The driver board has provision for adding a second pair of current-sensing resistors plus two 3-pin headers and two jumpers so that the switch from one pair to the other involves moving two jumpers. The board uses Type 1210, 1/4-W surface-mount parts for these resistors. The sizes supplied by Digi-Key are:

…, 1.0 Ω, 1.2 Ω, 1.5 Ω, 1.8 Ω, 2.2 Ω, …

Which resistors are a good choice for the second pair, and what is the resulting 70.7% current that will excite each winding?

8-4 Alternate power supply Along with the 12-V DC @ 0.5 A regulated supply discussed here (Digi-Key T983-P5P), Digi-Key also stocks T986-P5P, an 18 V @ 0.33 A regulated supply with the same 2.1-mm barrel connector that can be used with the stepper-motor driver board. Look in the on-line catalog of surplus parts of Herbach and Radman (www.herbach.com) or of Marlin P. Jones & Assoc. (www.mpja.com) for any bipolar (or "four wire") stepper motors that specify a voltage and either current per phase or resistance per phase that can be used with the stepper-motor driver board with the 18-V supply.

a) What is the part number and what are its specifications?

b) Using Equation 8-1 in the text, what should be the resistance of the current-sensing resistor to produce a winding current of 70.7% of the value set by Equation 8-1?

8-5 Clockwise stepping The code of T3.c steps the stepper-motor CCW.

a) Modify it to step CW.

b) Check the current draw on the coin cell with and without the stepper-motor ribbon cable attached to the Qwik&Low board.

8-6 Variable stepping rate The code of T3.c uses an int variable, **STEPCNT,** to control the stepping rate. To step every 4 ms (i.e., at a rate of 250 steps/s), Timer3 must roll over every

$$4{,}000 \times 0.032768 = 131 \text{ Timer3 counts}$$

This leads to

$$\text{STEPCNT} = 65{,}536 + 1 - 131 = 65{,}406$$

Modify the **Pushbutton** function of T3.c so that each press of the pushbutton, in addition to incrementing the displayed pushbutton count, also switches the stepping rate between 250 steps/s and 500 steps/s.

Chapter

9

ANALOG-TO-DIGITAL CONVERTER

9.1 OVERVIEW

This chapter discusses the features and use of the *analog-to-digital converter* (ADC) module in the PIC18LF4321. This module greatly extends the reach of the MCU into applications that employ any of the wealth of transducers (of temperature, pressure, acceleration, weight, etc.) that produce a voltage output. The ADC is a versatile module in the MCU with the following features:

- A resolution of 1 part in 1,024.
- The inclusion of a multiplexer that allows up to 13 pins to be used as analog inputs.
- A choice of reference voltages.
- A conversion time of 24 µs.

9.2 QWIK&LOW ANALOG VERSUS DIGITAL PINS

The PIC18LF4321's ADC has 13 possible analog inputs, any one of which can be multiplexed into its 10-bit converter, as shown in Figure 9-1. The figure illustrates that each analog input is shared with a digital input or output on a port pin. For example, the

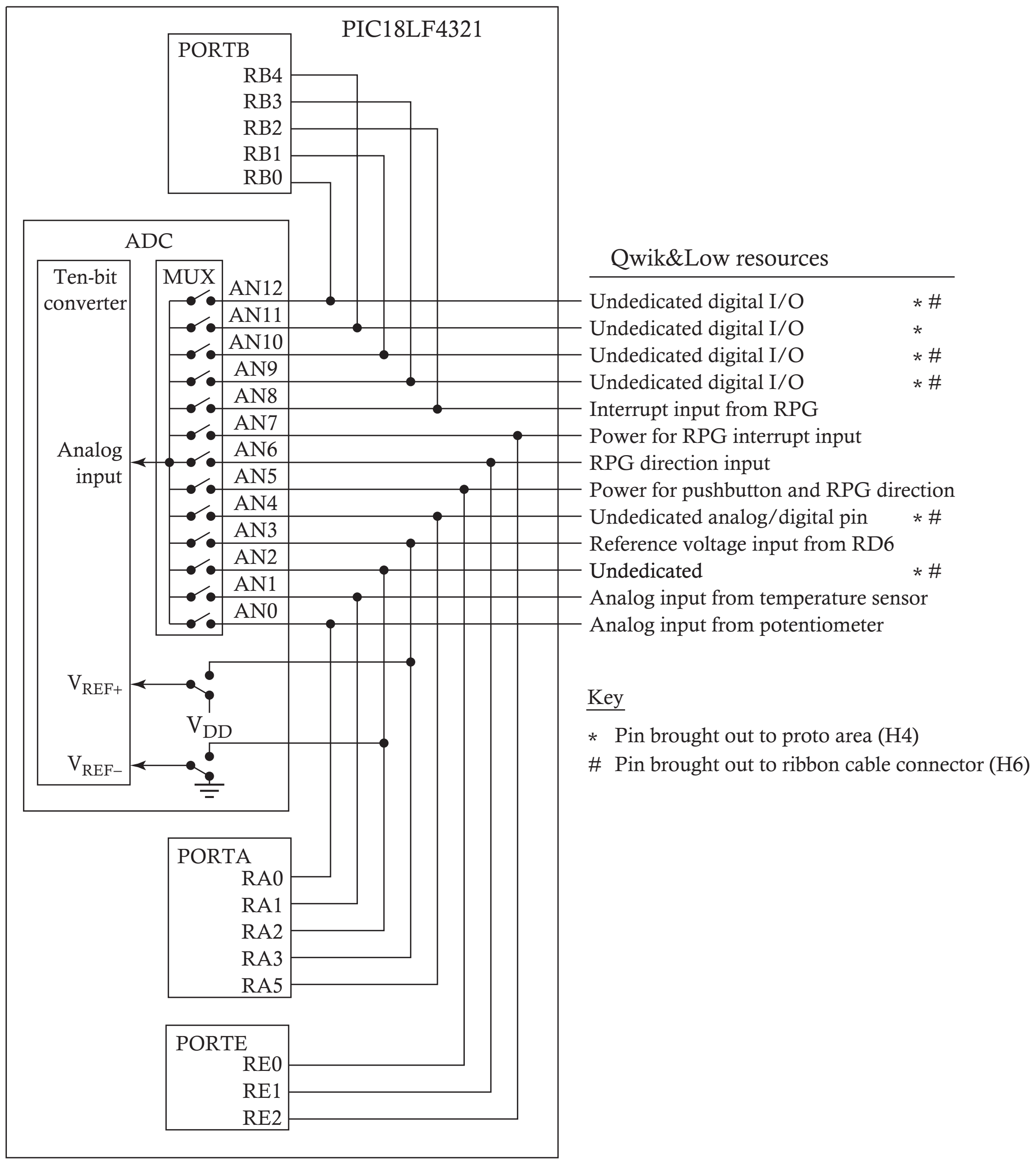
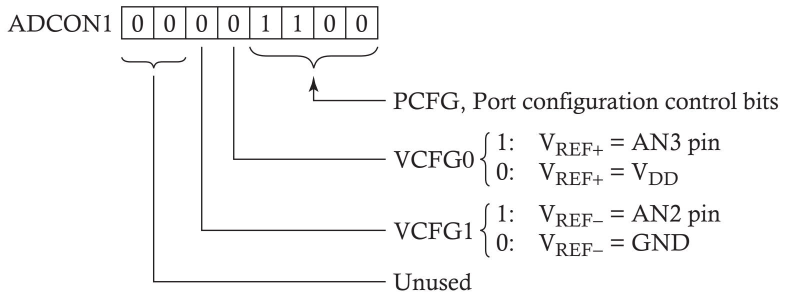

PIC18LF4321
PORTB
RB4
RB3
RB2
RB1
RB0
ADC
Ten-bit converter
MUX
AN12
AN11
AN10
AN9
AN8
AN7
AN6
AN5
AN4
AN3
AN2
AN1
AN0
Analog input
V_{REF+}
V_{DD}
V_{REF-}
PORTA
RA0
RA1
RA2
RA3
RA5
PORTE
RE0
RE1
RE2
Qwik&Low resources
Undedicated digital I/O * #
Undedicated digital I/O *
Undedicated digital I/O * #
Undedicated digital I/O * #
Interrupt input from RPG
Power for RPG interrupt input
RPG direction input
Power for pushbutton and RPG direction
Undedicated analog/digital pin * #
Reference voltage input from RD6
Undedicated * #
Analog input from temperature sensor
Analog input from potentiometer
Key
* Pin brought out to proto area (H4)
# Pin brought out to ribbon cable connector (H6)

FIGURE 9-1 Analog-to-digital converter pin connections both within the PIC18LF4321 and to Qwik&Low resources

Qwik&Low board's potentiometer is connected to the ADC's **AN0** input. Because this input shares a pin with the **RA0** bit of **PORTA,** a choice between these two uses of the pin must be made. Figure 9-2 shows how the choice is made. The 4-bit number loaded into the port configuration control bits of **ADCON1** can select any number of the 13 possible analog inputs to actually serve in this capacity, with the remaining inputs serving as digital I/O pins. Unlike some other PIC microcontrollers, the specification of


ADCON1
0 0 0 0 1 1 0 0
PCFG, Port configuration control bits
VCFG0 { 1: V_{REF+} = AN3 pin
0: V_{REF+} = V_{DD}
VCFG1 { 1: V_{REF-} = AN2 pin
0: V_{REF-} = GND
Unused

Number of analog channels	PCFG control bits				AN12	AN11	AN10	AN9	AN8	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0
0	1	1	1	1	D	D	D	D	D	D	D	D	D	D	D	D	D
1	1	1	1	0	D	D	D	D	D	D	D	D	D	D	D	D	A
2	1	1	0	1	D	D	D	D	D	D	D	D	D	D	D	A	A
3	1	1	0	0	D	D	D	D	D	D	D	D	D	D	A	A	A
4	1	0	1	1	D	D	D	D	D	D	D	D	D	A	A	A	A
5	1	0	1	0	D	D	D	D	D	D	D	D	A	A	A	A	A
6	1	0	0	1	D	D	D	D	D	D	D	A	A	A	A	A	A
7	1	0	0	0	D	D	D	D	D	D	A	A	A	A	A	A	A
•	•										•						
•	•										•						
•	•										•						

FIGURE 9-2 Selection of I/O pins to be analog inputs

the number of analog inputs selected also specifies *which* pins will be the analog pins. Furthermore, the choice is critical in the following sense:

- Any pin that is selected to be an analog pin has its digital I/O circuitry disabled.
- Any pin that is selected to be a digital pin can then be set up to be a digital (high-impedance) input and also used as an analog input. However, for an analog input in the middle voltage range (well above 0 V and well below 3 V), the digital circuitry will exhibit excessive leakage current.

The two devices designed into the Qwik&Low board having analog inputs are:

- The potentiometer.
- A temperature sensor.

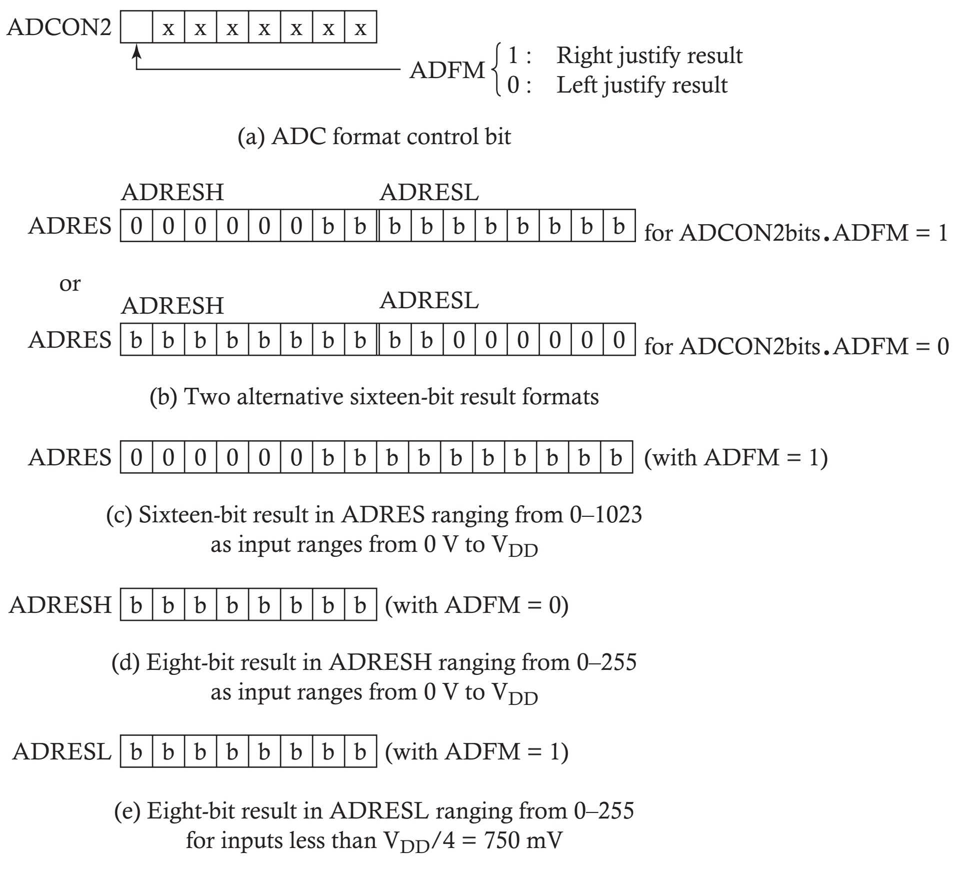
Also, the output pin of the MCU that powers the temperature sensor is used as the **VREF+/AN3/RA3** pin, taking advantage of a sensor whose voltage output is proportional to its supply voltage as well as its temperature. These have been assigned to inputs **AN0, AN1,** and **AN3** respectively so that a choice of 1011 for the Port Configuration control bits of Figure 9-2 will designate all but one (**AN2**) of the remaining pins to be available as possible digital I/O pins. Of these digital pins, any that are unused will be set up as outputs. Whether initialized high or low, the output voltage will not subject the also enabled digital input circuitry to a leakage-current-inducing voltage in the middle range.

Consider again Figure 9-1. One more analog device can easily be added to the proto area of the Qwik&Low board by connecting the analog device's output to the MCU's **AN2** input. Another analog input can be connected to the **AN4/RA5** input pin if the port configuration control bits of Figure 9-2 are changed to 1010.

9.3 ADC RESULT ALTERNATIVES

Depending on the state of the **ADFM** (ADC format) bit of Figure 9-3a at the time of an ADC conversion, the 10-bit result will be returned either right justified or left justified in the 16-bit register, **ADRES**, the 2 bytes of which can be identified as **ADRESH**

FIGURE 9-3 ADC result alternatives


ADCON2 | x | x | x | x | x | x | x
ADFM { 1 : Right justify result; 0 : Left justify result
(a) ADC format control bit
ADRESH ADRESL
ADRES 0 0 0 0 0 0 b b b b b b b b b b for ADCON2bits.ADFM = 1
or
ADRESH ADRESL
ADRES b b b b b b b b b b 0 0 0 0 0 0 for ADCON2bits.ADFM = 0
(b) Two alternative sixteen-bit result formats
ADRES 0 0 0 0 0 0 b b b b b b b b b b (with ADFM = 1)
(c) Sixteen-bit result in ADRES ranging from 0–1023 as input ranges from 0 V to V_DD
ADRESH b b b b b b b b (with ADFM = 0)
(d) Eight-bit result in ADRESH ranging from 0–255 as input ranges from 0 V to V_DD
ADRESL b b b b b b b b (with ADFM = 1)
(e) Eight-bit result in ADRESL ranging from 0–255 for inputs less than V_DD/4 = 750 mV

and **ADRESL,** as shown in Figure 9-3b. To use the full 0–1,023 range of the ADC output, **ADFM** should be set prior to the conversion. On its completion, **ADRES** can be read as the *unsigned int* variable of Figure 9-3c.

Obtaining the output of the single-turn potentiometer on the Qwik&Low board as a number having 256 values can be achieved by first clearing **ADFM** to zero, to left justify the result. When the conversion is complete, the upper 8 bits of the conversion will reside in **ADRESH,** as illustrated in Figure 9-3d. By reading only this *unsigned char* variable, the least significant 2 bits of the 10-bit conversion are ignored, throwing away the extra resolution provided by those 2 bits. What remains in **ADRESH** is a value of 0 if the potentiometer is turned fully CCW and a value of 255 if the potentiometer is turned fully CW.

Another option arises for an analog device whose output has a range reaching up to less than 750 mV (i.e., less than $V_{DD}/4$). The choice of Figure 9-3e provides the same resolution as using the 16-bit **ADRES** output of Figure 9-3b but with the computational advantage, for subsequent scaling of the output, of an 8-bit output.

9.4 REFERENCE VOLTAGE CHOICE

The **ADCON1** register of Figure 9-2 contains 2 control bits that allow an analog input to be scaled by a reference voltage input. Normally both **VCFG0** and **VCFG1** will be initialized to zero. Then the analog input will be compared against a voltage range extending between GND and V_{DD}. The PIC18LF4321 specification indicates that the chip will produce a scaled 10-bit output even with (**VREF+** – **VREF-**) down to 1.8 V.

The temperature sensor on the Qwik&Low board generates an output voltage proportional to temperature *and* to the supply voltage. This means that whether the supply voltage is 3 V early in the coin cell's life or 2.8 V some time later, the temperature reading will be the same. However, as shown in Figure 3-2, the sensor is powered from **RD6.** With an output impedance of the **RD6** pin of perhaps 200 Ω driving the 400-μA current drawn by the temperature sensor, the temperature sensor may see a supply voltage that is 80 mV less than V_{DD}. This is equivalent to 26 counts of the ADC (for which each count represents $3{,}000/1{,}024 \approx 3$ mV). One way to handle this offset is as an added, but somewhat uncertain, term in the temperature calculation. Another way is to load

```
ADCON1 = 0b00011011
```

This value will make **AN3** an analog input and will use the input on **AN3** from the **RD6** output as the **VREF+** reference voltage pin for the ADC for this measurement.

9.5 ADC TIMING

The ADCON2 register is described in Figure 9-4. In addition to controlling the left justify/right justify **ADFM** feature, this register controls the timing aspects of the converter. Figure 9-4a defines terms used in the specifications of Figure 9-4b. The clock period, T_{AD}, of the ADC should be no shorter than 1.4 μs. Given F_{OSC} = 4 MHz,

T_{AD}	is the ADC clock period.
T_{ACQ}	is the ADC acquisition time required between when an ADC channel has been selected and when a conversion can be initiated.
R_{SOURCE}	is the source (i.e., Thevenin) resistance of the device whose voltage output is being measured.
V_{REF+}	is the optional external high reference voltage (if used)

(a) ADC definitions

$$2.5\ \mu s > T_{AD} > 1.4\ \mu s$$

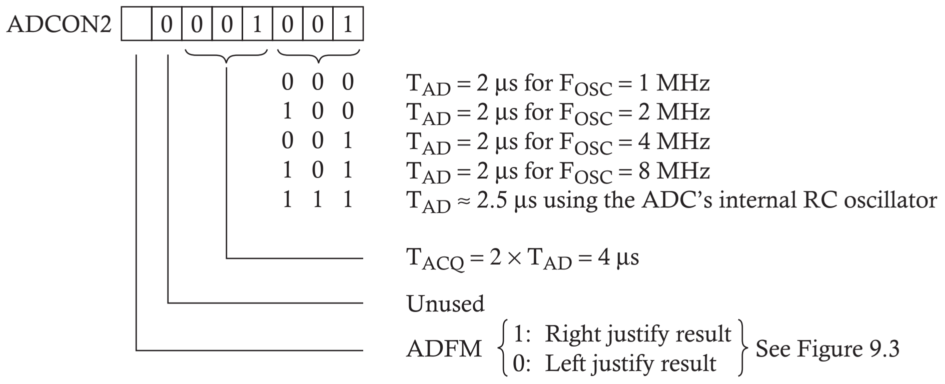
$$T_{ACQ} > 3.7\ \mu s \text{ for } R_{SOURCE} \le 5\ k\Omega$$

$$> 6.9\ \mu s \text{ for } R_{SOURCE} \le 20\ k\Omega$$

$$V_{DD} + 3.0\ V > V_{REF+} > 1.8\ V$$

$$\text{Conversion time} = 12\ T_{AD}$$

(b) PIC18LF4321 specifications


ADCON2
0 0 0 1 0 0 1
0 0 0 $T_{AD} = 2\ \mu s$ for $F_{OSC} = 1$ MHz
1 0 0 $T_{AD} = 2\ \mu s$ for $F_{OSC} = 2$ MHz
0 0 1 $T_{AD} = 2\ \mu s$ for $F_{OSC} = 4$ MHz
1 0 1 $T_{AD} = 2\ \mu s$ for $F_{OSC} = 8$ MHz
1 1 1 $T_{AD} \approx 2.5\ \mu s$ using the ADC's internal RC oscillator
$T_{ACQ} = 2 \times T_{AD} = 4\ \mu s$
Unused
ADFM 1: Right justify result
0: Left justify result
See Figure 9.3

(c) Initialization options for ADCON2

FIGURE 9-4 ADCON2 initialization

Figure 9-4c indicates that the least significant 3 bits of **ADCON2** should be initialized to 001, making $T_{AD} = 2\ \mu s$. A value of TAD of 1 µs or less is too fast for accurate conversion while a T_{AD} of 4 µs or greater will produce conversions that take 48 µs or 96 µs. With $T_{AD} = 2\ \mu s$, the 10-bit conversion will take $12 \times T_{AD} = 24\ \mu s$.

The time between selecting which input pin is to be multiplexed into the ADC and when the conversion is begun is automatically controlled by the T_{ACQ} parameter. The 001 setting of Figure 9-4c selects an acquisition time of 4 µs, sufficient for the two analog devices on the Qwik&Low board. Among the factors entering into the acquisition time determination, the two that a user has some control over are the source impedance of each analog source and the needed resolution of the result.

Example 9-1 Determine the maximum source resistance of the 20-kΩ potentiometer.

Solution

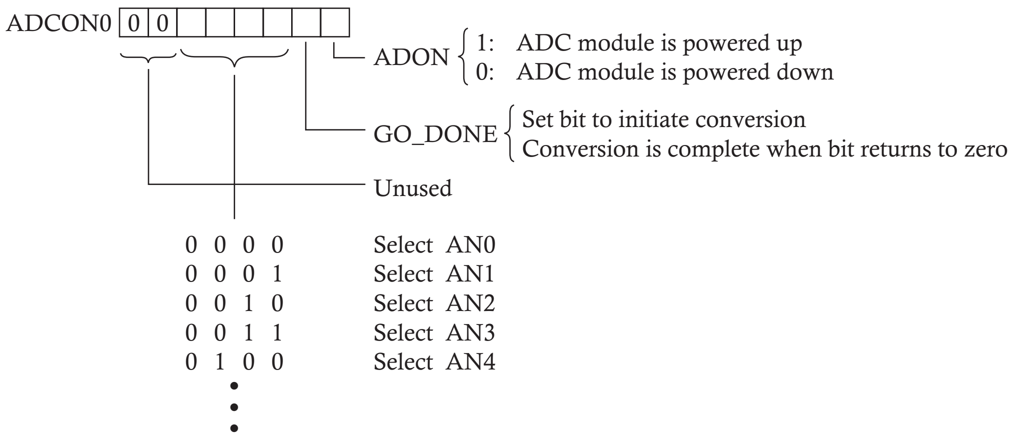
The source resistance is really the Thevenin resistance of a source. When the potentiometer is turned fully CCW, the source resistance is 0 Ω, the short circuit to ground. When the potentiometer is turned fully CW, the source resistance is again 0 Ω, the short circuit to V_{DD} (which is the same as a short circuit to ground, assuming the coin cell has essentially zero internal resistance). When the potentiometer wiper is at its midpoint, the source resistance is at its maximum of 10 kΩ to V_{DD} in parallel with the 10 kΩ to ground. That is, the maximum source resistance is 5 kΩ.

The source resistance affects the maximum acquisition time. A small internal sampling capacitor in the ADC charges up to match the (Thevenin) voltage of the analog source. The RC time constant of this charging depends on internal resistances in the ADC in series with the source resistance.

9.6 ADC INPUT SELECTION AND CONVERSION

The **ADCON0** register of Figure 9-5 contains bits to select the analog input. It also contains an **ADON** bit that selects whether the ADC module in the MCU is powered up or not. Finally, it contains a **GO_DONE** bit that is set to initiate a conversion. Then the **GO_DONE** bit becomes a flag that can be tested to determine whether the conversion has been completed.

FIGURE 9-5 ADCON0 use


ADCON0
0 0
ADON
1: ADC module is powered up
0: ADC module is powered down
GO_DONE
Set bit to initiate conversion
Conversion is complete when bit returns to zero
Unused
0 0 0 0 Select AN0
0 0 0 1 Select AN1
0 0 1 0 Select AN2
0 0 1 1 Select AN3
0 1 0 0 Select AN4

```
ADCON1 = 0x0b;            // Initialize ADC for Q&L's four analog inputs
ADCON2 = 0x09;            // ADC clock with Fosc = 4 MHz; left justify result
PORTAbits.RA7 = 1;        // Power up potentiometer (see Figure 3-2)
ADCON0 = 0x03;            // Power up ADC; select pot input; start conversion
while (ADCON0bits.GO_DONE);  // Wait for completion of conversion
PORTAbits.RA7 = 0;        // Power down potentiometer
ADCON0bits.ADON = 0;      // Power down ADC
<read result from ADRESH>
```

FIGURE 9-6 Use of ADC to read potentiometer

Because the acquisition time setting in the **ADCON2** register inserts a delay after the **GO_DONE** bit has been set and because the ADC module powers up essentially instantaneously relative to the 8 μs acquisition time selected, **ADCON0** can be loaded with the one value that will:

- Power up the ADC.
- Select an analog input.
- Initiate the conversion.

Example 9-2 Show the code to convert the output of the potentiometer into an 8-bit value in **ADRESH.**

Solution

Figure 9-6 shows the code lines to use V_{DD} as the reference voltage, to use the timing associated with F_{OSC} = 4 MHz, and to form **ADRESH.** Before the conversion is initiated, the output on **RA7** that powers the potentiometer must be set, as shown back in Figure 3-2. Then the **ADFM** bit is cleared, to have the upper 8 bits of the 10-bit result put into **ADRESH**. The conversion of the potentiometer input to **AN0** is then initiated. The result will be found in

$$T_{ACQ} + 12\ T_{AD} = 4 + (12 \times 2) = 28\ \mu s$$

with the automatic clearing of the **GO_DONE** bit signaling that **ADRESH** is ready with the result. The remaining lines power down the potentiometer and the ADC module as soon as possible before the result in **ADRESH** is read and used.

9.7 ADC CONVERSION DURING SLEEP

When the ADC is on but not converting, it typically draws 1.0 μA. Such a low current draw suggests another way that conversions can be carried out. As indicated in Figure 9-4c, the **ADCON2** register's bits for setting T_{AD} offer the option of using the ADC's own RC oscillator as its clock. Using this RC oscillator allows a conversion to be carried out while the chip is asleep. If the setting of the **GO_DONE** bit is immediately followed by the **Sleep** instruction, the current draw will be reduced.

Example 9-3 If the potentiometer output is read every tenth of a second, what is the effect on average current draw of having the conversion take place during sleep versus while the MCU is awake?

Solution

The CPU will spend less time awake in the main loop if the conversion is put off until the chip sleeps. But between the setting of the **GO_DONE** bit and its automatic clearing at the completion of the conversion, the awake CPU runs for the 28 µs pointed out at the end of the last section. Referring back to Figure 2-4, a current draw of 1.036 mA was measured when the CPU was running with F_{OSC} = 4 MHz but otherwise doing nothing to draw extra current. If the CPU is awake for an extra 28 µs every tenth of a second, this represents an extra average current draw by the CPU of

$$\frac{28}{100{,}000} \times 1{,}036\ \mu\text{A} = 0.29\ \mu\text{A}$$

regardless of how much current the ADC module draws, since that is essentially the same whether carried out while asleep or awake. On the other hand, if the conversion is carried out while asleep, its quiescent current of 1.0 µA while not converting will last for the duration of sleep within one pass through the main loop. This will contribute some fraction of

$$\frac{10\text{ms}}{100\text{ms}} \times 1.0\ \mu\text{A} = 0.10\ \mu\text{A}$$

If the CPU spends 2 ms out of every 10 ms loop time doing useful work before going to sleep, the net benefit of doing 10 conversions/s while asleep is about 0.29 – 0.08 = 0.21 µA of average current.

9.8 AD22103 TEMPERATURE SENSOR

The Analog Devices AD22103 temperature sensor is the small three-terminal device housed in the TO-92 transistor-like package located in the upper left corner of the Qwik&Low board. It is connected to the MCU as shown in Figure 3-2. As mentioned in Section 9.4, this is a *ratiometric* temperature sensor whose output voltage is not only proportional to temperature but also to its supply voltage.

Because the quiescent current is specified to be in the 350 µA–600 µA range over a supply voltage range of 2.7 V–3.6 V, the sensor needs to have its power switched off when it is not in use. Thus, the circuit of Figure 3-2 shows the sensor being powered from **RD6.** It also shows this same supply voltage serving as the reference voltage for the conversion. By taking advantage of the ratiometric feature of the sensor, the converted output value is insensitive to whether the coin cell is new, with an output voltage of 3.0 V, or run down, with an output voltage of (say) 2.7 V. Furthermore, by using this same supply voltage from **RD6** (rather than V_{DD}) as the reference voltage for the conversion, the measurement is made insensitive to the output pin's voltage drop due to its output impedance and the ≈400 µA current drawn by the sensor.

Although Analog Devices does not specify a settling time for this temperature sensor, clearly this becomes an important consideration in the following sequence:

- Set **RD6** = 1 to power up the sensor.
- Pause to allow the sensor to settle.
- Initiate the measurement.
- Wait for the completion of the conversion.
- Clear RD6 = 0 to power down the sensor.

The settling time of the sensor has two measurable components:

- The settling out of its underdamped step response (≈10 μs).
- Any longer asymptotic tail of this step response.

By waiting perhaps 20 μs after raising **RD6** and then making *repeated* full 10-bit ADC measurements spaced apart by the 24-μs conversion time of the ADC, the fast settling of the 10-bit readings can be verified. With essentially no change between the first and the second reading, evidently a 20-μs pause is sufficient.

The transfer function of the AD22103 is illustrated in graphical form in Figure 9-7a when the sensor's supply voltage, V_S, is 3.3 V. More generally, for a sensor supply voltage in the specified range of +2.7 V to +3.6 V, the output voltage, V_O, is given by the equation of Figure 9-7b. Analog Devices provides a somewhat ambiguous accuracy specification for the AD22103. The typical error of a room temperature measurement should be within half a degree Centigrade. Over the full range for the part, namely 0°C–100°C, the typical error should be within ±0.75°C.

Often it is useful to obtain temperature measurements with a *resolution* that is finer than the absolute accuracy that is warranted for a sensor. For temperature *changes*, the resolution produces temperature-difference measurements that are essentially correct. For example, if the sensor output voltage presently translates to a temperature of 71.4°F, whereas 10 min ago it translated to a temperature of 70.6°F, then the increase over the last 10 min is actually quite close to

$$71.4°\text{F} - 70.6°\text{F} = 0.8°\text{F}$$

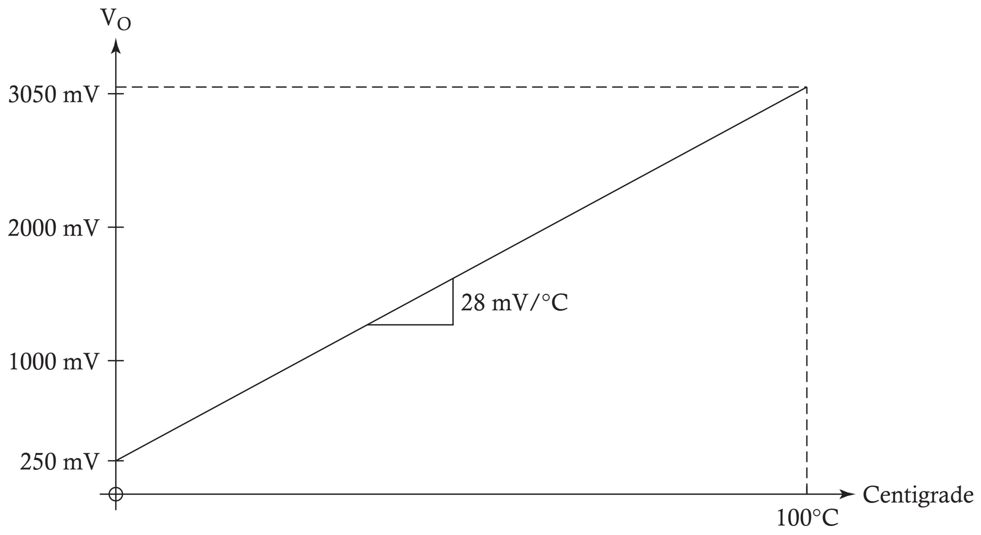
even though the absolute value of 71.4°F is probably less accurate.

Obtaining a temperature reading from an ADC measurement is expedited by translating the expression of Figure 9-7b into an expression of temperature in terms not of voltage but of **ADRES,** the 16-bit register that holds the 10-bit ADC output. If the output voltage equals 3,050 mV at 100°C when the supply voltage is 3,300 mV, then the ADC converter would translate this to

$$\frac{3{,}050}{3{,}300} \times 1{,}024 = 946.424 \text{ counts}$$

where, for the sake of the calculations that follow, the three digits to the right of the decimal point are warranted as an intermediate result, even though the ADC itself would show this result as 946. With the same 3,300 mV supply voltage, 0°C would translate to

$$\frac{250}{3{,}300} \times 1{,}024 = 77.576 \text{ counts}$$



(a) Transfer function for V_S (the supply voltage) = 3.3 V = 3300 mV

$$V_O = \frac{V_S\,(\text{volts})}{3.3\,(\text{volts})} \times \left[250\text{ mV} + (28\text{ mV/°C}) \times \text{Centigrade} \right] \text{millivolts}$$

(b) General transfer finction

$$\text{ADRES} = \left(\frac{946.424 - 77.576}{100 - 0} \times \text{Centigrade} \right) + 77.575$$

$$= (8.68848 \times \text{Centigrade}) + 77.575$$

(c) Translation of (b) to counts of ADRES

$$\text{Centigrade} = \frac{\text{ADRES}}{8.68848} - 8.9285 \qquad \text{°C}$$

(d) Reexpressing centigrade temperature as a function of ADRES

$$\text{Fahrenheit} = \left[\frac{9}{5} \times \text{Centigrade} \right] + 32 = \frac{\text{ADRES}}{4.82693} + 15.9287$$

(e) Expressing Fahrenheit temperature as a function of ADRES

FIGURE 9-7 AD22103 transfer function

Because of the ratiometric feature of the sensor, these are the same readings that would be obtained with the measurements using the Qwik&Low circuit and its lower V_{DD} and its even slightly lower **RD6** supply voltage to both the temperature sensor and the **VREF+** pin used by the ADC.

The equation of Figure 9-7c expresses the resulting value of **ADRES** as a function of the Centigrade temperature, again carrying extra (intermediate) digits of resolution.

This is inverted in Figure 9-7d to express the Centigrade temperature as a function of the value of **ADRES.** The conversion to Fahrenheit is shown in Figure 9-7e.

Carrying out the calculation of Figure 9-7d using integer arithmetic (for speed) requires that numerator and denominator be multiplied by the same integer constant before the division takes place. Recall from Figure 9-3b that clearing the **ADFM** bit of the **ADCON2** register before the ADC conversion takes place will produce a left-justified 10-bit result. In effect, this multiplies **ADRES** by 64. If the denominator of the quotient in Figure 9-7d is also multiplied by 64, the resulting expression for the Centigrade temperature becomes that shown in Figure 9-8a. If the terms in this equation are multiplied by 10, the integer result will be expressed in tenths of a degree Centigrade, as shown in Figure 9-8b. Note that a resulting integer of 253 represents a temperature of 25.3°C. If the Centigrade to Fahrenheit conversion of

$$F = \frac{9}{5}C + 32$$

is applied to these equations, the equations of Figure 9-8c and d result.

A calculation can be massaged to shorten its execution time. Note that multiply operations are significantly faster than divide operations. Also, a multiply or a divide of a multiple-byte number by $2^8 = 256 = 0x100$ amounts to moving the *bytes* left or right by 1 byte. For example,

$$2^8 \times 0x00001234 = 0x00123400$$

Applying these ideas to the equation of Figure 9-8d to obtain the temperature in tenths of a degree Fahrenheit, the equation is reexpressed in Figure 9-9a. Instead of

FIGURE 9-8 Temperature equations after converting voltage with ADCON2bits.ADFM = 0 (to left justify the output and, in effect, multiply ADRES by 64).

Centigrade = (ADRES / 556.063) – 8.9285

(a) For Centigrade measurement with 1˚C resolution

TenthC = (ADRES / 55.6063) – 89.285

(b) For Centigrade measurement with 0.1˚C resolution

Fahrenheit = (ADRES / 308.924) + 15.9287

(c) For Fahrenheit measurement with 1˚F resolution

TenthF = (ADRES / 30.8924) + 159.287

(d) For Fahrenheit measurement with 0.1˚F resolution dividing by the denominator of 30.8924, both the numerator and denominator can be first multiplied by

$$\frac{2^{16}}{30.8924} = 2{,}121.43$$

so that the denominator becomes 2^{16} and also the division by 2^{16} becomes two 1-byte shifts of the numerator. The resulting equation is shown in Figure 9-9c. To avoid overflow when the *int* register **ADRES** is multiplied by the *int* constant 2,121, **ADRES** can first be copied into the 4-byte-long variable, **VALUE.** The computation of Figure 9-9c is shown broken down into six steps for the benefit of optimization by the C18 compiler. The calculation in this form takes just 111 CPU clock periods.

FIGURE 9-9 Speeding up the calculation of temperature with 0.1˚F resolution

```
TenthF = (ADRES / 30.8924) + 159.287
```

(a) For Fahrenheit measurement with 0.1˚F resolution

```
2^16 / 30.8924 = 65536 / 30.8924 = 2121.43
```

(b) Forming a multiplier for numerator and denominator that will make the denominator equal to 2^16

```
TenthF = ((2121 * ADRES ) >> 16 ) + 159
```

(c) Resulting equation

```
unsigned long VALUE;          // 32-bit repesentation
unsigned int BIGNUM;          // 16-bit representation
```

(d) Definition of global variables

```
VALUE = ADRES;                // 32 bits to avoid overflow
VALUE *= 2121;
VALUE >>= 8;
VALUE >>= 8;
VALUE += 159;
BIGNUM = VALUE;               // 16-bit representation
```

(e) Calculation, broken down into separate steps that the C18 compiler handles better than a single-line calculation. Execution takes 111 μs.

PROBLEMS

9-1 Running average Write a Filter function that maintains an array of eight *int* values:

```
RESULT[0], RESULT[1], ..., RESULT[7]
```

a) In the **Initial** function, read a 10-bit value using the potentiometer output (as an easily available vehicle for averaging). Load this value into all eight **RESULT[*i*]** variables and form **SUM** equal to one of these, shifted left three places to multiply it by eight.

b) Every tenth of a second, subtract **RESULT[7]** from **SUM,** copy each **RESULT** value to the next location using a *for* loop, collect a new sample from the potentiometer, add it to **SUM,** and copy the sample value into **RESULT[0].** Finally, shift **SUM** right three places into the *int* variable, **BIGNUM** (to divide it by eight, the number of samples in the running average) and then use the **ASCII4** function from Figure 6-8b to break out the four digits and display them on the LCD.

9-2 ADC timing

a) Modify the **Temperature** function in the TenthsFahr.c file on the www.qwikandlow.com web site so that after collecting a sample, but before powering down the sensor, immediately collect a second sample. Display these two samples side by side on the LCD using the format exemplified by

```
73.2 73.4
```

Update the values every half second.

b) Does the extra delay produce a difference between the two values? If so, collect three samples (for timing), throw away the first and display the second and third. Does the extra delay help?

c) Set **RB0** before the initial setting of **RD6** that powers up the temperature sensor. Clear it before the first loading of **ADCON0** that produces a settled value of temperature. What is the resulting sensor setup time?

9-3 TenthsCent.c file

a) Modify the TenthsFahr.c file to use the equation of Figure 9-8b and a scheme analogous to that of Figure 9-9 to calculate the temperature with 0.1°C resolution.

b) Set and clear **RB0** around the calculation of Part (a), analogous to Figure 9-9e, and measure its execution time.

c) Compare the temperature you get with that of a mercury (or liquid) thermometer.

Chapter

10

ROTARY PULSE GENERATOR (RPG.c)

10.1 OVERVIEW

The rotary pulse generator (RPG), also known as a rotary encoder, is a widely used device for parameter entry in an application. If used as an audio system volume control, the sound level provides the feedback of its setting. If used to tune a radio, a display provides frequency feedback. Because of its versatility, ruggedness, and ease of use with a microcontroller, the RPG is used in many applications involving the entry of a multiple-valued parameter, particularly if the parameter must be incremented or decremented to be useful. For example, virtually every function generator uses an RPG rather than a keypad for controlling the frequency output.

In this chapter, the use of the detented RPG found on the Qwik&Low board will be considered. This low-cost RPG shares the same features found on the more robust RPGs of commercial products and instrument designs.

10.2 RPG RESOLUTION

Two RPGs are shown in Figure 10-1. The little Bourns RPG on the left will be used to explain the operation of an RPG when polled from the **main** loop. It uses three internal electrical contacts to three tracks of a coded wheel, as illustrated in Figure 10-2a.

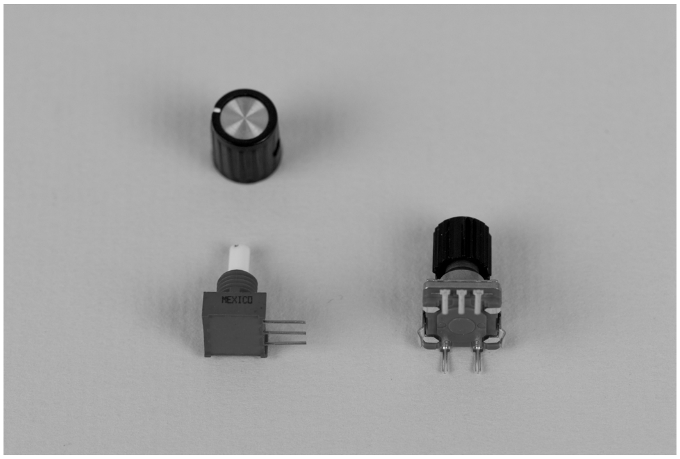


FIGURE 10-1 Two rotary pulse generators (Bourns 3315-C-001-006L; ALPS EC11G1524402)

The pull-up resistors are powered up by raising **RB3** when the RPG output is to be queried and powered down immediately afterwards, thus once again reducing the average current draw to a negligible amount. The inputs to **RB1** and **RB0** traverse through six cycles per revolution. Each cycle produces four output states, giving 24 states per revolution, as shown in Figure 10-2b.

The ALPS RPG, also pictured in Figure 10-1 and actually used on the Qwik&Low board, has a resolution of 30 *detented* increments per revolution. *Detented* means that the unit includes a mechanism that causes the RPG shaft to "click" from one of its 30 increments to the next, and that holds the position when released.

The 24 or 30 increments per revolution resolution of these two RPGs is typical of the resolution found in many commercial products and instruments. Although it might seem that "finer is better", a user will find it frustrating to have an undetented resolution so fine that one or more increments or decrements can occur when the knob is released.

In the sections that follow, the use of the Bourns RPG will be described first. It provides a good example of *polled* operation. It also provides insight that will help clarify why an interrupt handler is used with the detented ALPS unit.

10.3 RPG FUNCTIONALITY

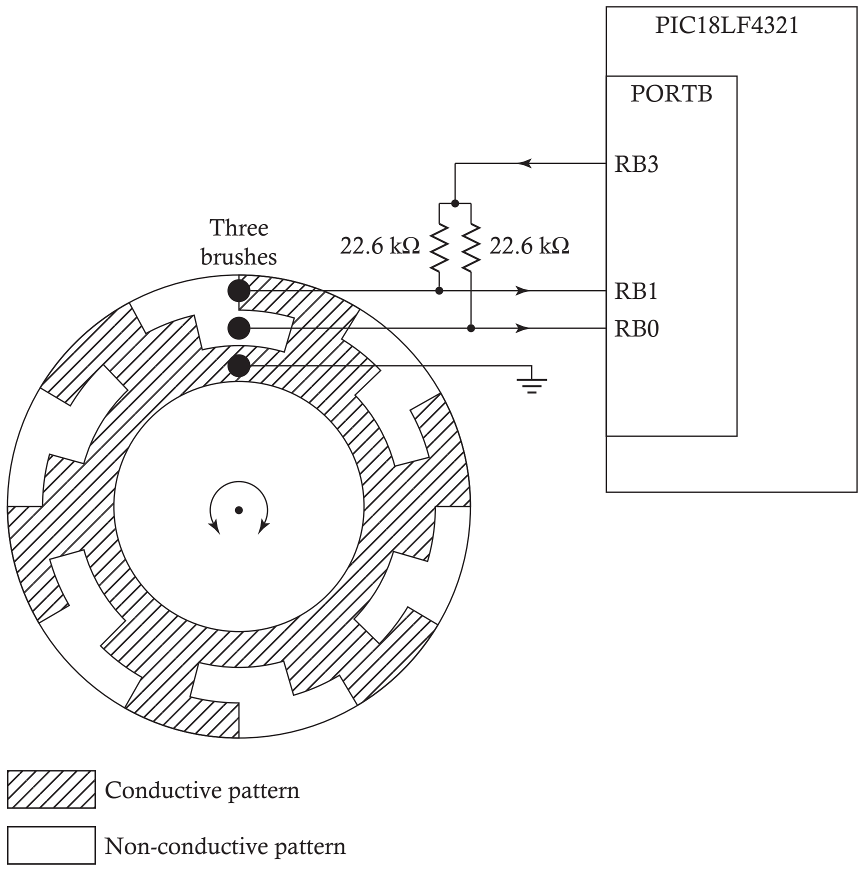
As an RPG is turned, its output pins traverse the 2-bit Gray code sequence

$$\ldots \rightarrow 00 \rightarrow 01 \rightarrow 11 \rightarrow 10 \rightarrow 00 \rightarrow \ldots$$

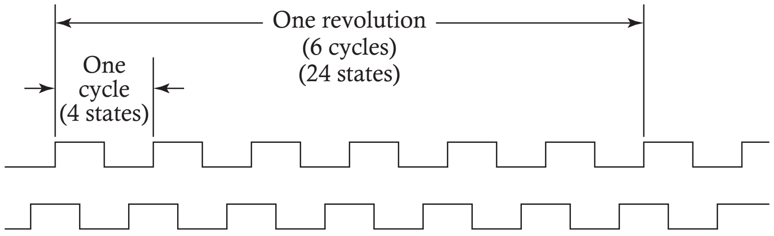
when turned in one direction and

$$\ldots \rightarrow 00 \rightarrow 10 \rightarrow 11 \rightarrow 01 \rightarrow 00 \rightarrow \ldots$$

when turned in the other direction. By reading the RPG outputs at a faster rate than they change, the resulting changes can be used to increment or decrement the parameter being controlled.



(a) Brush/wheel configuration



(b) Output

FIGURE 10-2 RPG's encoding wheel and brushes

The Bourns unit has a specified maximum turning rate of 120 rpm, or 2 revolutions/s. At this maximum rate, the minimum time between increments is about 20 ms. By reading the RPG every time around the main loop (i.e., every 10 ms), every change of state will be detected. Furthermore, because the application code integrates the effect of many rapidly occurring increments or decrements when the RPG is turned fast, missing any counts because of too-fast turning goes unnoticed. The Bourns RPG carries a maximum contact-bounce time specification of 5 ms. Consequently, reading its output every 10 ms effectively debounces it.

The ALPS unit with its detented output is used somewhat differently. Its two output pins traverse the same 2-bit Gray code sequence

$$\ldots \rightarrow 00 \rightarrow 01 \rightarrow 11 \rightarrow 10 \rightarrow 00 \rightarrow \ldots$$

when turned in one direction and

$$\ldots \rightarrow 00 \rightarrow 10 \rightarrow 11 \rightarrow 01 \rightarrow 00 \rightarrow \ldots$$

when turned in the other direction. However, only the 00 and 11 states represent *detented* positions.

10.4 THE RPG FUNCTION

If the Bourns RPG were added to the Qwik&Low board as shown in Figure 10-2a, its outputs would be sensed by raising **RB3**, waiting for 1 μs (with a **Nop()** macro) to allow for the 22.6-kΩ resistors and the slight capacitive loading of the circuitry to charge up, reading **PORTB,** and lowering **RB3** again. Each time around the main loop, **RB1** and **RB0** are compared with their state 10 ms earlier in the **RPG** function of Figure 10-3. The subroutine returns with

```
DELRPG = 0x00  if no change
       =   +1  if CW change
       =   -1  if CCW change
```

Other functions called within the main loop can use the value of **DELRPG** to increment or decrement a parameter value in response to a change in the RPG position.

When reading external pins from within a function, care should be taken to read the pins just once. In the case of the **RPG** function, the state of **PORTB** is read and masked, to form **NEWRPG**. It is needed at the beginning of the function to compare with the state 10 ms ago. At the conclusion of the function, the present state of **NEWRPG** is used to update **OLDRPG**, saving this value for use 10 ms later, next time around the main loop. For bug-free code, **OLDRPG** should be updated with the value of **NEWRPG** read at the beginning of the function rather than by reading and masking **PORTB** again.

The **RPG** function compares the value of **NEWRPG** (0, 1, 2, or 3) with the value found 10 ms ago. If these values differ, a change has taken place in the RPG position. If the bits have changed in a CW direction, one of the following four cases will have occurred:

$$0\underline{0} \rightarrow \underline{0}1$$

$$0\underline{1} \rightarrow \underline{1}1$$

$$1\underline{1} \rightarrow \underline{1}0$$

$$1\underline{0} \rightarrow \underline{0}0$$

```
PORTBbits.RB3 = 1;              // Power up RPG pullup resistors
Nop();                          // Wait a microsecond
OLDRPG = PORTB & 0b00000011;    // Load OLDRPG for RPG
PORTBbits.RB3 = 0;              // Power down RPG
```

(a) Initial function additions

```
/******************************
 * RPG
 *
 * This function generates DELRPG = +1 for a CW increment of the RPG,
 * -1 for a CCW increment, and 0 for no change.
 ******************************
 */
void RPG()
{
   DELRPG = 0;                    // Clear for "no change" return value
   PORTBbits.RB3 = 1;             // Power up RPG pullup resistors
   Nop();                         // Wait a microsecond
   NEWRPG = PORTB & 0b00000011;   // Read PORTB
   PORTBbits.RB3 = 0;             // Power down RPG
   if (NEWRPG != OLDRPG)          // Any change?
   {
      if (0b00000010 & (NEWRPG ^ (OLDRPG << 1))) // CW or CCW?
      {
         DELRPG = -1;             // CCW change
      }
      else
      {
         DELRPG = +1;             // CW change
      }
   }
   OLDRPG = NEWRPG;               // Save present RPG state for next loop
}
```

(b) The function.

FIGURE 10-3 RPG function

Note that in every case the MSb (most-significant bit) of the new number is the same as the LSb of the old number. If the RPG is changed in a CCW direction, then:

$$0\underline{0} \rightarrow \underline{1}0$$

$$1\underline{0} \rightarrow \underline{1}1$$

$$1\underline{1} \rightarrow \underline{0}1$$

$$0\underline{1} \rightarrow \underline{0}0$$

In every case the MSb of the new number is the complement of the LSb of the old number. This information forms the basis for the test, namely:

- Shift **OLDRPG** left so that its bit 0 is moved to the bit 1 position.
- Exclusive-OR the resulting shifted value in **OLDRPG** with **NEWRPG**.
- Mask off all but bit 1 to obtain a nonzero value for a CCW change and a zero value for a CW change.

The value of **DELRPG** is set accordingly.

The **RPGcounter** function of Figure 10-4 uses the **RPG** output, **DELRPG**, to increment or decrement **RPGNUM** and to display it on the two rightmost characters of the LCD.

10.5 THE DETENTED RPG

When the same technique used for the Bourns RPG is applied to the detented ALPS RPG of Figure 10-1, two issues arise:

```
   RPGNUM = 0;                  // Initialize to 0
```

(a) Initial function addition

```
/********************************
 * RPGcounter
 *
 * This function uses DELRPG to increment/decrement the signed char variable,
 * RPGNUM.  RPGNUM is then displayed in LCDSTRING[6] and LCDSTRING[7].
 ********************************
 */
void RPGcounter()
{
   RPGNUM += DELRPG;            // Update counter if RPG has been turned
   if (RPGNUM > 99)
   {
      RPGNUM = 0;
   }
   else if (RPGNUM < 0)
   {
      RPGNUM = 99;
   }
   LCDSTRING[6] = '0' + (RPGNUM / 10);  // Tens digit
   LCDSTRING[7] = '0' + (RPGNUM % 10);  // Units digit
   LCDFLAG = 1;                 // Set flag to display
}
```

(b) The function.

FIGURE 10-4 RPGcounter function

- The output representing a change occurs only at each detented position when the two pins are either in state 00 or state 11. But the transition state of 01 or 10 must also be read, to determine the direction of change.
- Because the detent mechanism inserts a *snap* into the movement, the two pins must use a shorter interval between samples (e.g., 3.333 ms) to read each state reliably during the transition from one detented position to the next.

The interrupt approach of Figure 3-2, repeated in Figure 10-5a, resolves these two issues. For this approach, the pull-up resistor on the output that goes to the MCU's interrupt pin must be continuously pulled high, resulting in a continuous current draw when the RPG is left standing in state 00. Using a 1-MΩ pull-up resistor has two beneficial effects:

- For an application that uses the RPG, the **RE2** pin that powers the pull-up resistor for the interrupt input produces a quiescent current draw of only 3 μA when the RPG outputs are at the "00" position (and 0 μA otherwise).
- The 1-MΩ pull-up resistor on the **INT2** interrupt input coupled with the associated low stray capacitance filters out the effect of contact bounce.

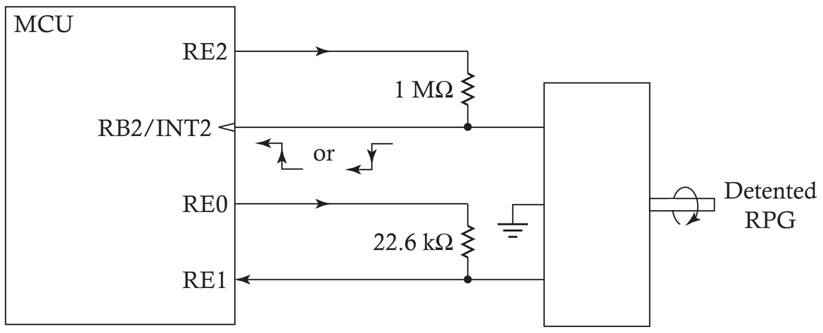
To determine the direction of rotation of the RPG, use is made of an **INTEDG2** control bit that determines whether the **INT2** input will be sensitive to a rising or a falling edge:

- If **INTEDG2** = 1, then an interrupt will occur in response to a rising edge into **INT2**.
- If **INTEDG2** = 0, then an interrupt will occur in response to a falling edge into **INT2**.

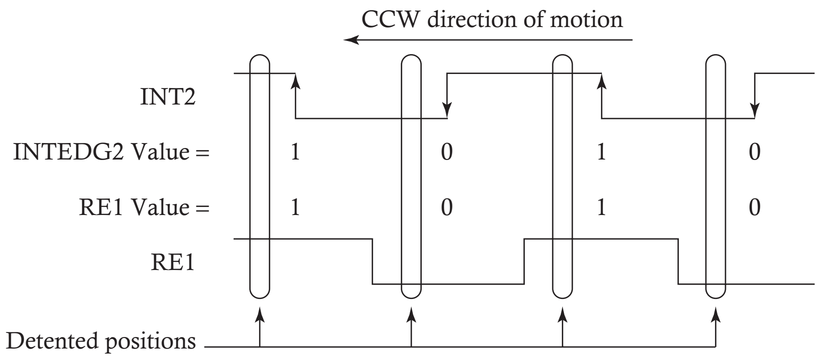
In response to each interrupt, **INTEDG2** is toggled. Consequently, *every* edge into **INT2** will produce an interrupt.

Consider the CCW rotation shown in Figure 10-5b. Note that in going from one detented position to the next, either a rising edge or a falling edge always occurs on the **INT2** input. Furthermore, for this CCW direction of motion, the state of **INTEDG2** at the time of the INT2 interrupt always matches the state of the **RE1** input read from the other RPG output. On the other hand, CW rotation produces a mismatch between **INTEDG2** and **RE1**, as shown in Figure 10-5c. This match or mismatch condition is used by the high-priority interrupt service routine of Figure 10-6b. Because more than one increment or decrement of **DELRPG** may take place during a 10-ms looptime, the HPISR may increment or decrement **DELRPG** more than once in that interval. The **RPGcounter** function of Figure 10-6c decrements a positive value of **DELRPG** toward zero while at the same time incrementing **RPGNUM**. It increments a negative value of **DELRPG** toward zero while at the same time decrementing **RPGNUM**. Thus **RPGNUM** simply accumulates all the increments and decrements of **DELRPG** into a value that is converted to a two-digit ASCII representation. The LCD is updated by setting **LCDFLAG** only when the RPG is turned.

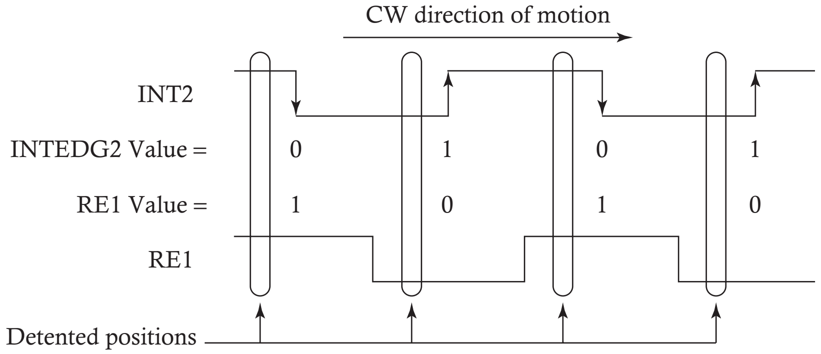
A complete, testable source file, RPG.c, is listed in Figure 10-7. It includes the modification of the **Initial** function to handle the RPG shown in Figure 10-6a. Note



(a) RPG connections



(b) CCW motion for which INTEDG2 = = RE1 at time of interrupt



(c) CW motion for which INTEDG2 != RE1 at time of interrupt

FIGURE 10-5 Determining the direction of rotation at the time of each interrupt

that after **RE2** of **PORTE** is set to power the 1-MΩ pull-up resistor for the interrupt input from the RPG, a 100-µs delay is inserted before **RB2/INT2** is read and used to initialize **INTEDG2** appropriately. Unlike other inputs that can be read with only a microsecond pause after the associated pull-up resistor has been powered up, the

```
   PORTEbits.RE2 = 1;           // Power up the pullup for the RPG's interrupt
   Delay(10);                   // Pause for 100 us
   INTCON2bits.INTEDG2 = !PORTBbits.RB2; // Select initial interrupt edge
   INTCON3bits.INT2IE = 1;      // Enable INT2 interrupt source
   INTCON3bits.INT2IP = 1;      // Use INT2 for high-priority interrrupts
   INTCON3bits.INT2IF = 0;      // Clear interrupt flag
   DELRPG = 0;                  // Indicate no initial edge
   RPGNUM = 0;                  // and initial RPG position of 00
   LCDFLAG = 1;                 // Display initial 00 value
```

(a) Initial function additions

```
/******************************
 * HiPriISR
 *
 * Respond to rising and falling edges on INT2 input from RPG.
 ******************************
 */
void HiPriISR()
{
   PORTEbits.RE0 = 1;           // Power pullup resistor to read direction
   INTCON3bits.INT2IF = 0;      // Reset interrupt flag
   if (PORTEbits.RE1 == INTCON2bits.INTEDG2)  // Direction?
   {
      ++DELRPG;
   }
   else
   {
      --DELRPG;
   }
   PORTEbits.RE0 = 0;           // Power down pullup resistor
   INTCON2bits.INTEDG2 ^= 1;    // Toggle edge sensitivity
}
```

(b) High-priority interrupt service routine

```
/******************************
 * RPGcounter
 *
 * This function uses DELRPG to update the signed char variable, RPGNUM.
 * DELRPG is returned to zero.
 * LCDSTRING[6] and LCDSTRING[7] display RPGNUM.
 ******************************
 */
void RPGcounter()
{
   while (DELRPG > 0)
   {
      --DELRPG;
```

FIGURE 10-6 Program code for detented RPG

```
      if (++RPGNUM > 99)        // Increment RPGNUM, modulo 100
      {
         RPGNUM -= 100;
      }
      TEMPCHAR = RPGNUM;
      LCDSTRING[6] = '0' + TEMPCHAR / 10;
      LCDSTRING[7] = '0' + TEMPCHAR % 10;
      LCDFLAG = 1;              // Set flag to display
   }
   while  (DELRPG < 0)
   {
      ++DELRPG;
      if (--RPGNUM < 0)         // Decrement RPGNUM, modulo 100
      {
         RPGNUM += 100;
      }
      TEMPCHAR = RPGNUM;
      LCDSTRING[6] = '0' + TEMPCHAR / 10;
      LCDSTRING[7] = '0' + TEMPCHAR % 10;
      LCDFLAG = 1;              // Set flag to display
   }
}
```

(c) RPGcounter function called from **main**.

FIGURE 10-6 *(continued)*

```
/******* RPG.c *************
 *
 * Fosc = 4 MHz for Fcpu = Fosc/4 = 1 MHz.
 * Timer1 is clocked by the Timer1 crystal oscillator.
 * LoopTime function puts chip to sleep.  Timer1 awakens chip every 10 ms
 * within LoopTime function.  CPU adjusts Timer1 content for it to timeout
 * after another 10 milliseconds.
 * INT2 edges from RPG produce high-priority interrupts and inc/dec DELRPG.
 * RPGcounter increments/decrements a two-digit number from DELRPG.
 * RC2 output is toggled every 10 milliseconds for measuring looptime.
 * LED on RD4 is blinked for 10 ms every four seconds.
 *
 * Current draw =  16 or 19 uA (depending on RPG position,
 *                             with LED and LCD switched off)
 *
 ******* Program hierarchy *****
 *
 * main
 *    Initial
 *    BlinkAlive
 *    RPGcounter
 *    UpdateLCD
 *       Display
```

FIGURE 10-7 RPG.c

```
 *    LoopTime
 *
 * HiPriISR
 *
 * LoPriISR
 *
 *******************************
 */

#include <p18f4321.h>           // Define PIC18LF4321 registers and bits

/*******************************
 * Configuration selections
 *******************************
 */
#pragma config OSC = INTIO1     // Use internal osc, RA6=Fosc/4, RA7=I/O
#pragma config PWRT = ON        // Enable power-up delay
#pragma config LVP = OFF        // Disable low-voltage programming
#pragma config WDT = OFF        // Disable watchdog timer initially
#pragma config MCLRE = ON       // Enable master clear pin
#pragma config PBADEN = DIG     // PORTB<4:0> = digital
#pragma config CCP2MX = RB3     // Connect CCP2 internally to RB3 pin
#pragma config BOR = SOFT       // Brown-out reset controlled by software
#pragma config BORV = 3         // Brown-out voltage set for 2.0V, nominal
#pragma config LPT1OSC = OFF    // Deselect low-power Timer1 oscillator

/*******************************
 * Global variables
 *******************************
 */
char LCDFLAG;                   // Flag, set to send string to display
char LPISRFLAG;                 // Flag, set when LP interrupt has been handled
unsigned char i;                // Index into strings
unsigned int DELAY;             // Sixteen-bit counter for obtaining a delay
unsigned int ALIVECNT;          // Scale-of-400 counter for blinking "Alive" LED
signed char DELRPG;             // RPG output
signed char RPGNUM;             // For display of RPG position
signed char TEMPCHAR;           // Temporary signed character

/*******************************
 * Variable strings
 *******************************
 */

char LCDSTRING[] = "      00 "; // Ongoing display string (9 characters)

/*******************************
 * Function prototypes
 *******************************
 */
void Initial(void);
```

FIGURE 10-7 *(continued)*

```
void BlinkAlive(void);
void RPGcounter(void);
void UpdateLCD(void);
void Display(void);
void LoopTime(void);
void HiPriISR(void);
void LoPriISR(void);

/******************************
 * Macros
 ******************************
 */
#define Delay(x) DELAY = x; while(--DELAY){ Nop(); Nop(); }

/******************************
 * Interrupt vectors
 ******************************
 */
// For high priority interrupts:
#pragma code high_vector=0x08
void interrupt_at_high_vector(void)
{
  _asm GOTO HiPriISR _endasm
}
#pragma code
#pragma interrupt HiPriISR

// For low priority interrupts:
#pragma code low_vector=0x18
void interrupt_at_low_vector(void)
{
  _asm GOTO LoPriISR _endasm
}
#pragma code
#pragma interruptlow LoPriISR

/******************************
 * HiPriISR
 *
 * Respond to rising and falling edges on INT2 input from RPG.
 ******************************
 */
void HiPriISR()
{
   PORTEbits.RE0 = 1;           // Power pullup resistor to read direction
   INTCON3bits.INT2IF = 0;      // Reset interrupt flag
   if (PORTEbits.RE1 == INTCON2bits.INTEDG2)  // Direction?
   {
      ++DELRPG;
   }
```

FIGURE 10-7 *(continued)*

```
   else
   {
      --DELRPG;
   }
   PORTEbits.RE0 = 0;           // Power down pullup resistor
   INTCON2bits.INTEDG2 ^= 1;    // Toggle edge sensitivity
}

/*******************************
 * LoPriISR
 *
 * Control 10 ms looptime
 *******************************
 */
void LoPriISR()
{
   T1CONbits.TMR1ON = 0;        // Pause Timer1 counter
   TMR1L += 0xB9;               // Cut out all but 328 counts of Timer1
   T1CONbits.TMR1ON = 1;        // Resume Timer1 counter
   TMR1H = 0xFE;                // Upper byte of Timer1 will be 0xFE
   PIR1bits.TMR1IF = 0;         // Clear interrupt flag
   LPISRFLAG = 1;               // Set a flag for LoopTime
}

/////// Main program /////////////////////////////////////////////////////////

/*******************************
 * main
 *******************************
 */
void main()
{
   Initial();                   // Initialize everything
   while (1)
   {
      PORTCbits.RC2 ^= 1;       // Toggle pin, for measuring loop time
      BlinkAlive();             // Blink "Alive" LED
      RPGcounter();
      UpdateLCD();              // Update LCD
      LoopTime();               // Use Timer1 to wakeup and loop again
   }
}

/*******************************
 * Initial
 *
 * This function performs all initializations of variables and registers.
 *******************************
 */
```

FIGURE 10-7 *(continued)*

```
void Initial()
{
   OSCCON = 0b01100010;          // Use Fosc = 4 MHz (Fcpu = 1 MHz)
   SSPSTAT = 0b00000000;         // Set up SPI for output to LCD
   SSPCON1 = 0b00110000;
   ADCON1 = 0b00001011;          // RA0,RA1,RA2,RA3 pins analog; others digital
   TRISA = 0b00001111;           // Set I/O for PORTA
   TRISB = 0b01000100;           // Set I/O for PORTB
   TRISC = 0b10000000;           // Set I/O for PORTC
   TRISD = 0b10000000;           // Set I/O for PORTD
   TRISE = 0b00000010;           // Set I/O for PORTE
   PORTA = 0;                    // Set initial state for all outputs low
   PORTB = 0;
   PORTC = 0;
   PORTD = 0b00100000;           // except RD5 that drives LCD interrupt
   PORTE = 0;
   LCDFLAG = 0;                  // Flag to signal LCD update is initially off
   ALIVECNT = 300;               // Blink immediately
   SSPBUF = ' ';                 // Send a blank to initialize state of UART
   Delay(50000);                 // Pause for half a second
   RCONbits.SBOREN = 0;          // Now disable brown-out reset

   T1CON = 0b01001111;           // Timer1 - loop time via low-pri interrupts
   TMR1H = 0xFE;                 // Set Timer1 to be 10 ms away from
   TMR1L = 0xB9;                 // next roll over (65536 + 1 - 328 = 0xFEB9)
   PIE1bits.TMR1IE = 1;          // Enable local interrupt source
   IPR1bits.TMR1IP = 0;          // Use Timer1 for low-priority interrupts
   LPISRFLAG = 0;                // Flag to signal that LPISR has been executed

   PORTEbits.RE2 = 1;            // Power up the pullup for the RPG's interrupt
   Delay(10);                    // Pause for 100 us
   INTCON2bits.INTEDG2 = !PORTBbits.RB2; // Select initial interrupt edge
   INTCON3bits.INT2IE = 1;       // Enable INT2 interrupt source
   INTCON3bits.INT2IP = 1;       // Use INT2 for high-priority interrrupts
   INTCON3bits.INT2IF = 0;       // Clear interrupt flag
   DELRPG = 0;                   // Indicate no initial edge
   RPGNUM = 0;                   //  and initial RPG position of 00
   LCDFLAG = 1;                  // Display initial 00 value

   RCONbits.IPEN = 1;            // Enable high/low priority interrupt feature
   INTCONbits.GIEL = 1;          // Global low-priority interrupt enable
   INTCONbits.GIEH = 1;          // Enable both high and low interrupts
}

/******************************
 * BlinkAlive
 *
 * This function briefly blinks the LED every four seconds.
 * With a looptime of about 10 ms, count 400 looptimes.
 ******************************
 */
```

FIGURE 10-7 *(continued)*

```
void BlinkAlive()
{
   PORTDbits.RD4 = 0;           // Turn off LED
   if (++ALIVECNT == 400)       // Increment counter and return if not 400
   {
      ALIVECNT = 0;             // Reset ALIVECNT
      PORTDbits.RD4 = 1;        // Turn on LED for one looptime
   }
}

/******************************
 * RPGcounter
 *
 * This function uses DELRPG to update the signed char variable, RPGNUM.
 * DELRPG is returned to zero.
 * LCDSTRING[6] and LCDSTRING[7] display RPGNUM.
 ******************************
 */
void RPGcounter()
{
   while (DELRPG > 0)
   {
      --DELRPG;
      if (++RPGNUM > 99)        // Increment RPGNUM, modulo 100
      {
         RPGNUM -= 100;
      }
      TEMPCHAR = RPGNUM;
      LCDSTRING[6] = '0' + TEMPCHAR / 10;
      LCDSTRING[7] = '0' + TEMPCHAR % 10;
      LCDFLAG = 1;              // Set flag to display
   }

   while  (DELRPG < 0)
   {
      ++DELRPG;
      if (--RPGNUM < 0)         // Decrement RPGNUM, modulo 100
      {
         RPGNUM += 100;
      }
      TEMPCHAR = RPGNUM;
      LCDSTRING[6] = '0' + TEMPCHAR / 10;
      LCDSTRING[7] = '0' + TEMPCHAR % 10;
      LCDFLAG = 1;              // Set flag to display
   }
}

/******************************
 * UpdateLCD
 *
 * This function updates the 8-character LCD if the LCDFLAG is set.
 ******************************
 */
```

FIGURE 10-7 *(continued)*

```
void UpdateLCD()
{
   if (LCDFLAG)
   {
      Display();
      LCDFLAG = 0;
   }
}

/******************************
 * Display
 *
 * This function sends LCDSTRING to the LCD.
 ******************************
 */
void Display()
{
   PORTDbits.RD5 = 0;           // Wake up LCD display
   for (i = 0; i < 9; i++)
   {
      PIR1bits.SSPIF = 0;       // Clear SPI flag
      SSPBUF = LCDSTRING[i];    // Send byte
      while (!PIR1bits.SSPIF);  // Wait for transmission to complete
   }
   PORTDbits.RD5 = 1;           // Return RB5 high, ready for next string
}

/******************************
 * LoopTime
 *
 * This function puts the chip to sleep upon entry.
 * It wakes up and executes a HPISR and then returns to sleep.
 * It wakes up and executes a LPISR and then exits.
 ******************************
 */
void LoopTime()
{
   while (!LPISRFLAG)           // Sleep upon entry and upon exit from HPISR
   {                            // Return only if LPISR has been executed,
      Sleep();                  //  which sets LPISRFLAG
      Nop();
   }
   LPISRFLAG = 0;               // Sleep upon next entry to LoopTime
}
```

FIGURE 10-7 *(continued)*

RB2/INT2 pin exhibits a 75-µs rise time after **RE2** is raised to 3 V when the RPG is in the "11" position.

Since **RE2** will be left high thereafter, this 75-µs rise time occurs every time the RPG is changed from its "00" position to its "11" position. The long rise time is what damps out any contact-bounce "chatter" when the RPG is changed from its "11" position to its "00" position. The **INT2** input is not affected by the slow rise time, inasmuch as the signal on the input pin is subjected to a *Schmitt trigger* before being passed along to the interrupt circuitry. The Schmitt trigger converts its slowly rising input to a fast snap of a change on its output. It also subjects the input to *hysteresis*, having a slightly higher rising-voltage threshold at which the output snaps than its falling-voltage threshold at which the output snaps back. This hysteresis also mitigates the effect of any contact-bounce chatter. The result is an RPG whose performance matches the ideal, with one increment of the displayed value for each detented click of the knob.

PROBLEMS

10-1 RPGcounter modification

a) Modify the **RPGcounter** function of Figure 10-4 and its initialization to use two variables

```
RPGTENS  and   RPGONES
```

in place of **RPGNUM**. Each time **DELRPG** causes an increment or decrement of **RPGNUM** in the old **RPGcounter** function, now increment or decrement the ASCII-coded value in **RPGTENS** and **RPGONES** appropriately. Refer to T2.c's Time function in Figure 5-5 for ideas.

b) Repeat for the **RPGcounter** function of Figure 10-6.

10-2 Polled RPG.c

a) Modify the **RPG** function of Figure 10-3 to poll the two outputs of the detented RPG of Figure 10-5a. Because the 1-MΩ pull-up resistor on the **RB2** input produces such a long response time, initialize **RE2** high and leave it high.

b) Write a PolledRPG.c file derived from the RPG.c file of Figure 10-7 by removing the high-priority ISR and adding the **RPG** function of Part (a), called from the main loop. Also replace the **RPGcounter** function with that of Figure 10-3.

c) Read an 8-bit value from the pot every 20th time around the main loop. Use this to modify the loop time proportional to the pot value and from (about) 1 ms if the pot is fully CCW and (about) 10 ms if the pot is fully CW.

d) As the RPG is turned, either slowly or at a faster (but "normal") rate, determine the maximum loop time for which RPG clicks (almost always) produce increments or decrements on the LCD. Measure the time between edges of **RC2** to determine this loop time.

Chapter

11

MEASUREMENTS

11.1 OVERVIEW

Coding a microcontroller application in C has benefits and drawbacks, listed in Section 2.9. As mentioned there, the size of the resulting program is not a dominating issue, as long as it fits in the available program memory within the target microcontroller. The speed of execution of algorithms is another story, given the weight placed on coin cell current draw and the resulting effect on coin cell life. The faster execution of an algorithm leads directly to more sleep time (with the MCU drawing less than 2 μA) and less awake time (with the MCU drawing more than 1 mA).

In this chapter, the monitoring of both code size and execution time will be discussed. C coding provides little insight into either issue. Being able to monitor code size and execution time allows the comparison of alternative codings of an algorithm. From such a comparison, coding choices can be made intelligently.

11.2 CODE SIZE

Alex Singh's C18 utility:

- Compiles and links a source file.
- Lists the number of program bytes.
- Shows the percentage of available program memory used.

In an environment where the C compiler may need to insert complex subroutines to implement a few lines of seemingly simple source code, this gross view of the code generated provides helpful insight.

Example 11-1 Compare the amount of code generated by Figure 11-1a with the amount of code generated by the alternative implementation of the same function shown in Figure 11-1b. This latter implementation uses the same technique of incrementing or decrementing an ASCII-coded two-digit number as was used in the Pushbutton function of the T2.c template program (Figure 5-5).

Solution

Compiling the two source files containing the two versions of the **RPGcounter** produces program byte counts for each file. The program containing the code of Figure 11-1a generates *22* more bytes of code than the identical program with only the **RPGcounter** function replaced by the code of Figure 11-1b.
This example illustrates several points:

- While the implementation of Figure 11-1b uses slightly less memory than the implementation of Figure 11-1a, it is more complicated to write and more difficult for someone else to understand. These are significant considerations.
- The amount of code generated does not correlate well with the number of lines in the source file. In the implementation of Figure 11-1a, the formation of **LCDSTRING[6]** incurs the call of a divide subroutine. The formation of **LCDSTRING[7]** has the compiler calling the same subroutine with the same divisor and dividend to form the (previously discarded) remainder. The algorithm of Figure 11-1b replaces these divides with increments, decrements, compares, and reinitializations.

```
/*******************************
 * RPGcounter
 *
 * This function uses DELRPG to update the signed char variable, RPGNUM.
 * DELRPG is returned to zero.
 * LCDSTRING[6] and LCDSTRING[7] display count.
 *******************************
 */
void RPGcounter()
{
   while (DELRPG > 0)
   {
      --DELRPG;
```

FIGURE 11-1 Two implementations of RPGcounter

```
      if (++RPGNUM > 99)
      {
         RPGNUM -= 100;
      }
      LCDFLAG = 1;              // Set flag to display
   }

   while  (DELRPG < 0)
   {
      ++DELRPG;
      if (--RPGNUM < 0)
      {
         RPGNUM += 100;
      }
      LCDFLAG = 1;              // Set flag to display
   }
   TEMPCHAR = RPGNUM;
   LCDSTRING[6] = '0' + (TEMPCHAR / 10);
   LCDSTRING[7] = '0' + (TEMPCHAR % 10);
}
```

(a) Implementation of RPGcounter using two divide operations

```
/********************************
 * RPGcounter
 *
 * This function uses DELRPG to update ASCII-coded RPGTENS:RPGUNITS
 * DELRPG is returned to zero.
 * LCDSTRING[6] and LCDSTRING[7] display count.
 *******************************
 */
void RPGcounter()
{
   while (DELRPG > 0)
   {
      --DELRPG;
      if (++RPGUNITS > '9')
      {
         RPGUNITS = '0';
         if (++RPGTENS > '9')
         {
            RPGTENS = '0';
         }
      }
      LCDFLAG = 1;              // Set flag to display
   }

   while  (DELRPG < 0)
   {
      ++DELRPG;
```

FIGURE 11-1 *(continued)*

```
      if (--RPGUNITS < '0')
      {
         RPGUNITS = '9';
         if (--RPGTENS < '0')
         {
            RPGTENS = '9';
         }
      }
      LCDFLAG = 1;           // Set flag to display
   }
   LCDSTRING[6] = RPGTENS;   // Update display string
   LCDSTRING[7] = RPGUNITS;
}
```

(b) Implementation of RPGcounter incrementing ASCII-coded RPGTENS:RPGUNITS.

FIGURE 11-1 *(continued)*

11.3 CODE EXECUTION

Using an oscilloscope to measure time intervals, the user program can be modified to set a pin prior to the execution of the code segment whose duration is to be measured and to clear the pin on its completion. If the **RB0** pin is used for this purpose, its use will be simplified by the addition of the macro definitions shown in Figure 11-2a.

Example 11-2 Assume that **TEMPCHAR** contains a number ranging between 0 and 99 and is to be displayed as a two-digit number on the LCD display. Determine the time taken to form **LCDSTRING[6]** and **LCDSTRING[7]** in Figure 11-2b.

Solution

The two lines are preceded by a **Disable()** macro to postpone interrupts and a **Start()** macro to set the **RB0** pin. The lines are followed by a **Stop()** macro and an **Enable()** macro to produce a pulse on the **RB0** pin of 210 µs. Subtracting the 1-µs interval that is produced if **Start()** is immediately followed by **Stop()** yields an execution time of 209 µs for the two lines that form the two **LCDSTRING** bytes.

If the code for which a time interval is to be measured resides in the main code, it will occasionally be interrupted if one or more high-priority interrupt sources are being employed. By disabling interrupts before the pulse on **RB0** is initiated and reenabling interrupts after the pulse has been completed, any intervening interrupts will not be lost, only delayed until the completion of the **RB0** pulse.

```
/******************************
 * Macro definitions
 ******************************
 */
#define Disable() INTCONbits.GIE = 0
#define Start()   PORTBbits.RB0 = 1
#define Stop()    PORTBbits.RB0 = 0
#define Enable()  INTCONbits.GIE = 1
```

(a) Definition of Start() and Stop() macros.

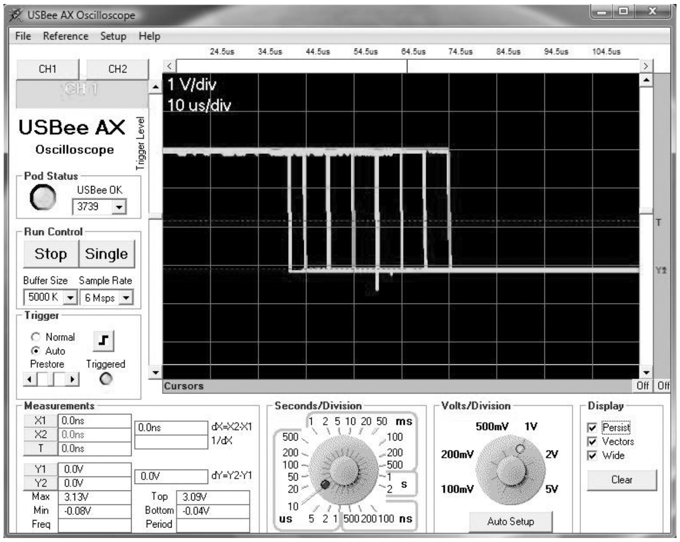
```
   Disable();
   Start();
   LCDSTRING[0] = '0' + (TEMPCHAR / 10);
   LCDSTRING[1] = '0' + (TEMPCHAR % 10);
   Stop();
   Enable();
```

(b) Formation of two ASCII-coded digits.
Execution time = 210 – 1 = 209 μs.

FIGURE 11-2 Execution time determination

11.4 VARIABLE CODE EXECUTION

If the code segment whose execution time is being measured varies, an oscilloscope's "persist" or "autostore" mode will overwrite the display with each successive trace, as illustrated in Figure 11-3. Using the scope's time cursors, both the minimum pulse width and maximum pulse width are easily ascertained. Taking advantage of the brightness of the range of falling edges, a casual estimate of the mean pulse width is also easily obtained.



1 V/div
10 us/div

USBee OK 3739

Buffer Size 5000 K
Sample Rate 6 Msps

Normal
[✓ Auto]
Prestore

Measurements

X1	0.0ns	0.0ns	dX=X2-X1
X2	0.0ns		1/dX
T	0.0ns		
Y1	0.0V	0.0V	dY=Y2-Y1
Y2	0.0V		
Max	3.13V	Top	3.09V
Min	-0.08V	Bottom	-0.04V
Freq		Period	

Display
☑ Persist
☑ Vectors
☑ Wide

FIGURE 11-3 Use of oscilloscope's "persist" mode

11.5 INTERRUPT TIMING MEASUREMENT

Measuring how long the CPU digresses from its **main** function in response to an interrupt request can take advantage of the **Delay** macro of Section 4.7. First determine the minimum time between successive interrupts. Then set the delay parameter to something less than one-tenth of this number of microseconds. This will produce a pulse that will be interrupted zero or one time, but no more than one.

Example 11-3 Determine the minimum time overhead of a high-priority interrupt.

Solution

Add the code of Figure 11-4a to the **main** function. When an interrupt occurs, the pulse width will be extended by the exact time of digression for the CPU to deal with the interrupt. The HPISR of Figure 7-6 has been reduced, in Figure 11-4b, to do nothing more than clear the interrupt flag. When the program is executed and the pulse displayed in the "persist" mode, the display of Figure 11-4c is produced. After sufficient time to have at least one interrupt occur concurrently with a pulse, the display of Figure 11-4c will result, where the trigger point on the rising edge of the pulse has been moved about 1 ms off to the left of the screen. The difference between the two falling edges of 15 μs indicates that the CPU digressed for 15 μs as it set aside the program counter and other selected CPU registers, vectored to the HPISR, executed the one flag-clearing instruction (taking 1 μs to do so), restored the selected CPU registers, and returned to where it left off in the main code.

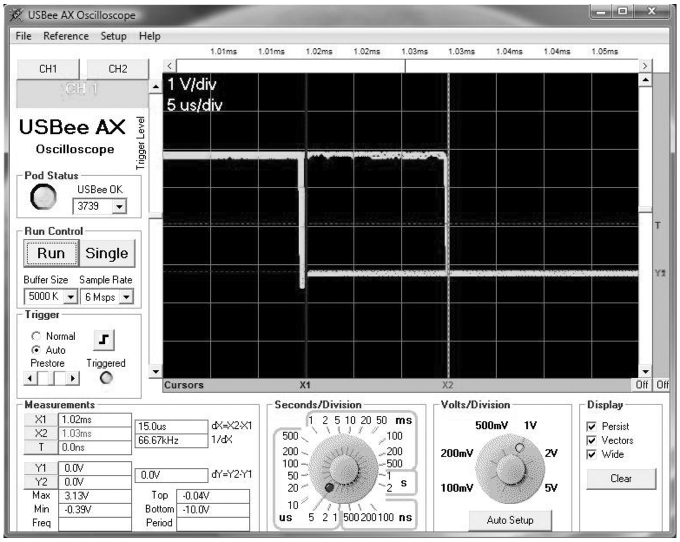
```
   Start();
   Delay(100);                // Pause for 1 ms
   Stop();
```

(a) One-millisecond pause in main loop.

```
/*******************************
 * HiPriISR
 *
 * Measure time CPU digresses from main to execute this minimal HPISR.
 *******************************
 */
void HiPriISR()
{
   PIR2bits.TMR3IF = 0;         // Clear interrupt flag
}
```

(b) A minimal high-priority interrupt service routine.

FIGURE 11-4 Determining the minimum time overhead of a high-priority interrupt service routine.



1 V/div
5 us/div

USBee OK: 3739

Buffer Size: 5000 K
Sample Rate: 6 Msps

Trigger: Auto

Measurements:

X1 1.02ms
X2 1.03ms
T 0.0ns
dX=X2-X1 15.0us
1/dX 66.67kHz
Y1 0.0V
Y2 0.0V
dY=Y2-Y1 0.0V
Max 3.13V
Min -0.39V
Freq
Top -0.04V
Bottom -10.0V
Period

Display: [✓ Persist] [✓ Vectors] [✓ Wide]

(c) Scope's screen dump

FIGURE 11-4 *(continued)*

11.6 TIME SPENT DOING USEFUL WORK IN MAIN LOOP

By inserting the **Start()** macro at the beginning of the **main** loop and the **Stop()** macro just before the call of the **LoopTime** function, the time taken to do useful work can be monitored. Each time around the main loop, some time will be spent in functions called from **main** and some time within intervening interrupts.

Generally, it is valuable to know the average of this time since it leads to the average current draw from the coin cell:

$$\text{Average current draw} = \frac{T_{\text{USEFUL WORK}}\,(\mu\text{S})}{10{,}000\,(\mu\text{S})} \times 1\text{ mA}$$

where 10,000 μs is the loop time, and (from Figure 2-4) 1 mA represents the current draw when the chip is clocked continuously with F_{OSC} = 4 MHz. The oscilloscope's "persist" mode again helps to identify a reasonable average for $T_{\text{USEFUL WORK}}$. It also permits the measurement of the worst-case value, even when that value rarely occurs.

11.7 CLEANING UP SURPRISES

When the average current draw is higher than expected, the use of the **Start()** and **Stop()** macros provides an easy way to close in on the source of the problem. The execution time of questionable functions in the **main** loop becomes a good place to start. A long-duration function at that level leads to the functions it calls. In this way, one or more offending functions can be identified. Testing makes all the difference, whereas *looking* at a long source file is a difficult method of debugging. As once pointed out to the author, “paper never protests”.

Some of the issues that can arise are:

- A function may calculate a value using long-duration multiplies and divides each time around the **main** loop while the value is used to update the display only once every 10 loop times.
- An interrupt service routine that occurs many times per 10-ms loop time also carries out an extensive routine to handle each occurrence. Instead, it might accumulate successive changes in a variable and then let a main function handle the accumulated change each loop time before resetting the variable.
- Alternative implementations of a function may yield quite different execution times. It is difficult to discern this from the source file. It is easy to discern it by the measurement of the alternatives. Time is not spent looking for alternatives to code that only *looked* like it might produce a long execution time, when, in fact, it does not.

PROBLEMS

11-1 Function size Consider the T3.c template of Figure 7-6. Determine the number of program bytes generated by the **Pushbutton** function. First, compile the code as is and note the number of program bytes generated. Then form a T3A.c template with /* inserted before the line

```
void Pushbutton()
```

and a */ after the function’s final }. Then recompile and note the reduced number of program bytes generated. The difference between these two numbers is the number of bytes sought.

11-2 Function size Form a T3B.c template program by rewriting the **Pushbutton** function of Figure 7-6 using a new *signed char* variable, **PBNUM,** to represent the number of pushbutton pushes. Use the code of Figure 11-1a as a model for how to use **PBNUM,** including the final three lines of code

```
TEMPCHAR = PBNUM;
LCDSTRING[0] = '0' + (TEMPCHAR / 10);
LCDSTRING[1] = '0' + (TEMPCHAR % 10);
```

a) Compare the number of program bytes generated by T3B.c with the number found for T3A.c of Problem 11-1, for which the **Pushbutton** code was commented out.

11-3 Execution time

a) In the main function of T3.c, insert the **Disable** and **Start** macros before

```
Pushbutton();
```

and the **Stop** and **Enable** macros after it. Recompile and measure the execution time of the **Pushbutton** function. What is the value when the pushbutton is not pressed? What is the value when the number changes from 2 to 3? From 9 to 10?

b) Repeat for T3B.c.

Chapter

12

INTERRUPTS

12.1 OVERVIEW

Interrupts provide a way for the CPU to break out of the lockstep sequence of tasks dictated by the function calls in the main loop. Stepping the stepper motor with a step period other than some multiple of the loop time represents a common interrupt application. Sensing RPG motion only when the RPG generates edges represents another.

How interrupt service routines are written in C has a profound effect on both execution time and program size. The first part of this chapter pinpoints this issue. The remainder of the chapter identifies the control and status bits associated with *every* interrupt source in the PIC18LF4321.

12.2 MCU INTERRUPT RESPONSE

Chapter Seven introduced the use of both high- and low-priority interrupts. First, the code was inserted into the source file to have the compiler handle the two *interrupt vectors* to **HiPriISR** and **LoPriISR**. Then, the interrupt service routines, **HiPriISR** and

LoPriISR, were written and added to the source file just like any other function. When an interrupt of main program execution occurs, the CPU:

1. Suspends its main program execution.
2. For a low-priority interrupt, further low-priority interrupts are disabled, but any high-priority interrupts that have been enabled are left enabled. For a high-priority interrupt, all interrupts are disabled.
3. Vectors to the interrupt service routine.
4. Copies any CPU registers that, if changed by the execution of the interrupt service routine, might corrupt the execution of the interrupted main program.
5. Executes the interrupt service routine.
6. Restores the CPU registers set aside in Step 4.
7. Reenables the interrupt level (high or low) that was disabled in Step 2.
8. Returns to the main program at the point where execution was suspended in Step 1.

12.3 COMPILER HANDLING OF INTERRUPTS

In this section, the extra code introduced by the compiler for Steps 4 and 6 will be examined. The intent is to discern *how* the writing of the code of Step 5 impacts *what the compiler does* with Steps 4 and 6. Snippets from several source file examples will be examined.

The code of Figure 12-1a is taken from a file called IntProg1.c (available from the www.qwikandlow.com web site). It shows a **main** function that uses the **Delay** macro to generate a 140-µs pulse on the **RC2** output pin every 140 µs + 200 µs = 340 µs. Timer1 and Timer3 have been set up to be clocked by the CPU's 1-MHz clock (rather than the Timer1 oscillator) so that interrupts will be synchronized to the CPU clock. For testing purposes, the **Initial** function includes the two lines shown in Figure 12-1b that control whether or not either timer runs, and therefore whether or not it generates interrupts.

Within each interrupt service routine of Figure 12-1c, the two lines of code to reload a timer and clear an interrupt flag compile to three machine code instructions that execute in 3 µs. This knowledge will be used to translate from the total time that the CPU takes to deal with an interrupt to the portion of that time that represents *overhead*. It is this overhead that can be drastically influenced by *how* the code is written. The reorganization of each interrupt service routine in Figure 12-1d breaks out these same two lines to reload a timer and clear an interrupt flag into two separate "handler" functions.

The two ways of expressing the interrupt service routines (Figure 12-1c on the one hand and Figure 12-1d on the other) plus the disabling of one interrupt source or the other via the lines of Figure 12-1b allow four experiments to be run. As each is compiled, downloaded, and run, the 140-µs pulse generated by the main loop of Figure 12-1a is monitored. The trailing edge of this pulse is extended whenever an interrupt strikes during the pulse. The amount of the extension equals the ISR overhead plus

```
void main()
{
   Initial();                  // Initialize everything
   while (1)
   {
      PORTCbits.RC2 = 1;       // Set pin high
      Delay(14);               // Pause for 140 microseconds
      PORTCbits.RC2 = 0;       // Set pin low
      Delay(20);               // Pause for 200 microseconds
   }
}
```

(a) Setup to measure the digression from a constant duration pulse by an interrupt.

```
//   PIE1bits.TMR1IE = 1;         // Enable local interrupt source
   PIE2bits.TMR3IE = 1;         // Enable local interrupt source
```

(b) Lines from Initial function, for enabling only one interrupt source at a time.

```
/*******************************
 * HiPriISR
 *
 * Reload TMR3H, clear flag, and return.
 *******************************
 */
void HiPriISR()
{
   TMR3H = 0xFF;
   PIR2bits.TMR3IF = 0;         // Clear interrupt flag
}

/*******************************
 * LoPriISR
 *
 * Reload TMR1H, clear flag, and return.
 *******************************
 */
void LoPriISR()
{
   TMR1H = 0xFF;
   PIR1bits.TMR1IF = 0;         // Clear interrupt flag
}
```

(c) One way of writing ISRs (from IntProg1.c in www.qwikandlow.com)

FIGURE 12-1 Snippets from IntProg1.c and IntProg2.c

```
/******************************
 * HiPriISR
 ******************************
 */
void HiPriISR()
{
   Timer3Handler();
}

/******************************
 * LoPriISR
 ******************************
 */
void LoPriISR()
{
   Timer1Handler();
}

/******************************
 * Timer3Handler
 ******************************
 */
void Timer3Handler()
{
   TMR3H = 0xFE;                 // Reload Timer3
   PIR2bits.TMR3IF = 0;          // Clear interrupt flag
}

/******************************
 * Timer1Handler
 ******************************
 */
void Timer1Handler()
{
   TMR1H = 0xFE;                 // Reload Timer1
   PIR1bits.TMR1IF = 0;          // Clear interrupt flag
}
```

(d) Another way of writing ISRs (from IntProg2.c in www.qwikandlow.com).

FIGURE 12-1 *(continued)*

the 3 μs to reload the timer and clear the flag. Thus, when the code of Figure 12-1c is executed with only Timer3 producing high-priority interrupts, the 140-μs pulse is extended 14 μs by an intervening interrupt. The ISR overhead of 11 μs (i.e., 14 μs – 3 μs) is listed in the top line of Figure 12-2.

The results of all four experiments are summarized in this table of Figure 12-2. The second line shows an ISR overhead of 21 μs. The 10 μs less taken by the HiPriISR to do the same thing occurs because the high-priority ISR automatically and quickly copies three CPU registers into three *shadow registers* on entry and quickly restores them on exit. The low-priority ISR must also set aside and later restore these same

	ISR overhead	Compiled bytes in entire file
HiPriISR of Figure 12-1c	11	431
LoPriISR of Figure 12-1c	21 = (11 + 10)	
HiPriISR of Figure 12-1d	99 = (11 + **88**)	707 = (431 + **276**)
LoPriISR of Figure 12-1d	109 = (21 + **88**)	

FIGURE 12-2 Effect of interrupt service routine code writing alternatives on performance

three CPU registers, but there is no automatic, fast mechanism to do so. Rather, the compiler adds instructions to do the setting aside and restoring for the low-priority interrupt service routine.

When the two source code lines in an interrupt service routine, to reload the timer and clear the flag, are broken out to a separate "handler" function as in Figure 12-1d, evidently the C compiler loses track of what registers the service routine will be using. As shown in the third and fourth lines of Figure 12-2, the compiler adds 276 bytes of code. This extra code evidently sets aside and restores every conceivable CPU register and compiler-generated variable that might ever be corrupted by an interrupt service routine. When this code is executed, it also adds 88 µs to the interrupt service routine's execution time.

These four tests provide the insight that a large performance penalty is incurred for both the program code size and execution time by delegating interrupt service routine code to a separate "handler" function.

12.4 USING ONE PRIORITY LEVEL ONLY

Given the performance perspective of Figure 12-2, it may make sense for many applications to use just one interrupt priority level. Such an application assumes that no interrupt service routine excessively delays the handling of other interrupt sources. This delay requirement is referred to as the *acceptable worst-case latency* of an interrupt source. It is the time beyond which the interrupt source may experience performance degradation. For example, if the stepper motor has a maximum stepping rate of about 1,000 steps/s, then at this maximum rate, Timer3 is interrupting every millisecond. Even if two other interrupts line up to be serviced first, their presumably short handlers are not going to insert enough of an irregularity in the timing of the steps to throw the motor out of its fast coasting rhythm.

A snippet of code from an IntProg3.c file is shown in Figure 12-3. Each of three interrupt sources makes use of the same high-priority interrupt service routine. Within **HiPriISR**, each interrupt flag is polled to determine whether it is requesting service. If so, its handler is executed in line (*without* breaking out to a separate handler function and invoking the performance hit discussed in the last section).

The **Initial** function includes the three lines shown in Figure 12-3b so that two of the three interrupts can be disabled for each of three experiments. Upon compiling and running and monitoring the **main** function's pulse on **RC2**, each interrupt causes the CPU to digress from its execution of the **main** function with an ISR overhead of 19 µs.

```
void main()
{
   Initial();                    // Initialize everything
   while (1)
   {
      PORTCbits.RC2 = 1;         // Set pin high
      Delay(14);                 // Pause for 140 microseconds
      PORTCbits.RC2 = 0;         // Set pin low
      Delay(20);                 // Pause for 200 μs
   }
}
```

(a) Setup to measure the digression from a constant duration pulse by an interrupt.

```
   T1CONbits.TMR1ON = 1;         // Enable Timer1
//   T3CONbits.TMR3ON = 1;         // Enable Timer3
//   INTCON3bits.INT2IE = 1;       // Enable INT2 interrupt source
```

(b) Lines from Initial function, for disabling all but one interrupt source.

```
/*******************************
 * HiPriISR
 *******************************
 */
void HiPriISR()
{
   if (PIR2bits.TMR3IF)          // Timer3 interrupt?
   {
      TMR3H = 0xFE;
      PIR2bits.TMR3IF = 0;
   }
   if (PIR1bits.TMR1IF)          // Timer1 interrupt?
   {
      TMR1H = 0xFE;
      PIR1bits.TMR1IF = 0;
   }
   if (INTCON3bits.INT2IF)       // INT2 interrupt?
   {
      INTCON3bits.INT2IF = 0;
   }
}
```

(c) A third way of writing ISRs (from IntProg3.c in www.qwikandlow.com).

```
Compiles to 423 bytes.
CPU digresses for 22-3 = 19 us to handle HiPriISR for Timer3.
CPU digresses for 22-3 = 19 us to handle HiPriISR for Timer1.
CPU digresses for 20-1 = 19 us to handle HiPriISR for INT2.
```

(d) Results

FIGURE 12-3 Snippets from IntProg3.c

12.5 PIC18LF4321 INTERRUPT SOURCES

The following three lines of initialization are required to enable interrupts globally:

```
RCONbits.IPEN = 1;     // Enable high/low priority interrupt feature
INTCONbits.GIEL = 1;   // Globally enable low-priority interrupts
INTCONbits.GIEH = 1;   // Globally enable high/low interrupts
```

In addition, each interrupt source has its own local interrupt control bits:

- A priority control bit
- A local enable bit
- A local flag bit

To have any interrupt source be given high-priority interrupt servicing, its priority control bit must first be set. Alternatively, the priority control bit must be cleared to have it responded to as a low-priority interrupt source.

Each interrupt source has a local enable bit that is disabled at reset. Consequently, nothing need be done to initialize the many possible interrupt sources that go unused in any given application.

Figure 12-4 lists each possible interrupt source and identifies the register and bit names for the priority control bit, the local enable bit, and the local flag bit.

12.6 USE OF THE INTERRUPT MECHANISM + IDLE MODE

In Section 2.7, the MCU's *idle* mode was discussed. Setting the **IDLEN** bit in the **OSCCON** register enables this mode. Then, when a **Sleep** macro is executed, the clocking of the CPU is halted, stopping the execution of further instructions. Meanwhile, all internal peripheral modules that had been running continue to be clocked by F_{OSC}. The result is the halving of the current draw by the MCU, as indicated in Figure 2-9.

To make use of this feature of the MCU, all local interrupt enable bits other than the one that will awaken the chip must be cleared, the **IDLEN** bit of **OSCCON** must be set, and the **GIEH** bit of **INTCON** must be *cleared*. An internal peripheral module event can be initiated and the **Sleep** mode executed. On the completion of the event, as signaled by the setting of the module's interrupt flag, the CPU will awaken and continue code execution with the code that follows the **Sleep** macro.

12.7 EXTERNAL INTERRUPTS

The detented RPG discussed in Chapter Ten made use of one of the MCU's three external interrupt pins. The bits associated with all three of these pins are described in Figure 12-5.

Name	Priority Bit	Local Enable Bit	Local Flag Bit
INT0 external interrupt	*	INTCONbits.INT0IE	INTCONbits.INT0IF
INT1 external interrupt	INTCON3bits.INT1IP	INTCON3bits.INT1IE	INTCON3bits.INT1IF
INT2 external interrupt	INTCON3bits.INT2IP	INTCON3bits.INT2IE	INTCON3bits.INT2IF
RB port change interrupt	INTCON2bits.RBIP	INTCONbits.RBIE	INTCONbits.RBIF
TMR0 overflow interrupt	INTCON2bits.TMR0IP	INTCONbits.TMR0IE	INTCONbits.TMR0IF
TMR1 overflow interrupt	IPIR1bits.TMR1IP	PIE1bits.TMR1IE	PIR1bits.TMR1IF
TMR3 overflow interrupt	IPR2bits.TMR3IP	PIE2bits.TMR3IE	PIR2bits.TMR3IF
TMR2 matches PR2 interrupt	IPR1bits.TMR2IP	PIE1bits.TMR2IE	PIR1bits.TMR2IF
CCP1 interrupt	IPR1bits.CCP1IP	PIE1bits.CCP1IE	PIR1bits.CCP1IF
CCP2 interrupt	IPR2bits.CCP2IP	PIE2bits.CCP2IE	PIR2bits.CCP2IF
A/D converter interrupt	IPR1bits.ADIP	PIE1bits.ADIE	PIR1bits.ADIF
USART receive interrupt	IPR1bits.RCIP	PIE1bits.RCIE	PIR1bits.RCIF
USART transmit interrupt	IPR1bits.TXIP	PIE1bits.TXIE	PIR1bits.TXIF
Synchronous serial port int.	IPR1bits.SSPIP	PIE1bits.SSPIE	PIR1bits.SSPIF
I^{2}C bus collision interrupt	IPR2bits.BCLIP	PIE2bits.BCLIE	PIR2bits.BCLIF
Parallel slave port int.	IPR1bits.PSPIP	PIE1bits.PSPIE	PIR1bits.PSPIF
High/low-voltage detect int.	IPR2bits.HLVDIP	PIE2bits.HLVDIE	PIR2bits.HLVDIF
Oscillator failure int.	IPR2bits.OSCFIP	PIE2bits.OSCFIE	PIR2bits.OSCFIF
Comparator interrupt	IPR2bits.CMIP	PIE2bits.CMIE	PIR2bits.CMIF
Data EEPROM write done int.	IPR2bits.EEIF	PIE2bits.EEIE	PIR2bits.EEIF

* INT0 can only be used as a high-priority interrupt

FIGURE 12-4 PIC18LF4321 interrupt sources

```
INT0 input is shared with RB0 pin
INT1 input is shared with RB1 pin
INT2 input is shared with RB2 pin

Initialize TRISB bits with -  1s for interrupt or digital inputs
                           -  0s for digital outputs or unused pins
```

(a) Initialization of PORTB pins as inputs or outputs

```
RCONbits.IPEN = 1;          // Enable high/low priority mechanism
INTCONbits.GIEL = 1;        // Globally enable low-priority interrupts
INTCONbits.GIEH = 1;        // Globally enable high-priority interrupts
```

(b) Global interrupt initialization

FIGURE 12-5 External interrupts

```
INTCON2bits.INTEDG0 = 1;     // Sense rising (1) or falling (0) edge
INTCONbits.INT0IE = 1;       // Local enable for INT0 interrupts
INTCONbits.INT0IF = 0;       // Clear flag initially, test flag subsequently
```

(c) INT0 interrupt initialization (high priority only)

```
INTCON2bits.INTEDG1 = 1;     // Sense rising (1) or falling (0) edge
INTCON3bits.INT1IP = 1;      // High (1) or low (0) priority selection
INTCON3bits.INT1IE = 1;      // Local enable for INT0 interrupts
INTCON3bits.INT1IF = 0;      // Clear flag initially, test flag subsequently
```

(d) INT1 interrupt initialization

```
INTCON2bits.INTEDG2 = 1;     // Sense rising (1) or falling (0) edge
INTCON3bits.INT2IP = 1;      // High (1) or low (0) priority selection
INTCON3bits.INT2IE = 1;      // Local enable for INT0 interrupts
INTCON3bits.INT21IF = 0;     // Clear flag initially, test flag subsequently
```

(e) INT2 interrupt initialization

FIGURE 12-5 *(continued)*

12.8 PORTB-CHANGE INTERRUPTS

If either **RB4** or **RB5** is set up as an input, a low-to-high change or a high-to-low change on that input will set the **RBIF** flag of Figure 12-4. This flag setting can be used to generate an interrupt to the CPU if the chip is awake at the time of the change. Alternatively, it can be used to awaken the chip and begin execution with the next instruction after the **Sleep** macro.

When **PORTB** is read, the value read for either of these two pins set up as an input will be copied into an internal (i.e., inaccessible) flip-flop. It is the subsequent mismatch between an input pin and the internal copy that sets **RBIF**. Once set, **RBIF** remains set, even if the input pin changes back to match the internal flip-flop. A two-step process is used to reset the **RBIF** flag:

1. Read **PORTB** so that the internal flip-flop matches the state of the input pin. A write to **PORTB** for which the bit "written" to the input pin on **RB4** or **RB5** matches the internal flip-flop will also satisfy this first step.
2. Once the state of the input pin matches the state of the internal flip-flop, then

   ```
   INTCONbits.RBIF = 0;
   ```

   will clear the flag.

If both **RB4** and **RB5** are set up as inputs, there may be an ambiguity concerning which input changed, setting **RBIF**. A narrow $0 \rightarrow 1 \rightarrow 0$ or $1 \rightarrow 0 \rightarrow 1$ pulse that is shorter than the response time to a resulting interrupt will have trouble identifying which pin caused the interrupt (if that matters).

Normally, **RB6** and **RB7** can be set up as inputs to serve as two more interrupt-on-change pins. However, with QwikBug installed, **RB6** and **RB7** are taken over by the chip's debug mode, features of which are used by QwikBug.

PROBLEMS

12-1 Measure HiPriISR of Figure 12-1c Modify the Measure.c program of Chapter Six to determine how long the CPU digresses from its execution of the main loop code to deal with the **HiPriISR** of Figure 12-1c. Your program should never stop. Rather, it should loop with timing such that one (but not more than one) interrupt is as likely as not to occur during each pass around the loop. Your main loop code should look like that of Figure 12-1a, but with the setting of **RC2** replaced by the **Start** Function and the clearing of RC2 replaced by the **Stop** and **Send** functions. Modify the duration of the **Delay** macros to obtain an interrupt between **Start** and **Stop** as often as not.

12-2 Measure HiPriISR of Figure 12-1d Repeat the procedure of Problem 12-1 for the **HiPriISR** of Figure 12-1d.

12-3 Determining what affects execution time of HiPriISR Modify the code and measurement of Problem 12-2 by introducing a new *char* variable, **NUMBER,** that is multiplied by five:

a) Only in the **Initial** function.
b) Only in the main loop, but not between **Start** and **Stop**.
c) Only in the **HiPriISR** before it calls **Timer3Handler**.
d) Only in **Timer3Handler**.

Consider the measurement result in each case, compared with the measurement result for Problem 12-2 and explain any difference and what must be happening. Note that the multiply instruction employs the CPU registers **PRODH** and **PRODL**.

Chapter

13

TIMING MEASUREMENTS REVISITED (CALIBRATE.c)

13.1 OVERVIEW

Chapter Eleven explored various timing measurements with the help of an oscilloscope. This chapter will use Timer0 and Timer1 to calibrate F_{CPU}, the nominal 1-MHz clock, against the 50 parts per million, 32,768-Hz Timer1 oscillator. Incrementing/decrementing the **OSCTUNE** register can then bring the actual value of F_{CPU} close to 1 MHz. Alternatively, any time interval measurement can be scaled from timer counts to microseconds using a previously determined calibration factor.

Either of the MCU's two CCP (capture, compare, PWM) modules can be used in conjunction with Timer3 to measure the duration of a pulse input to a CCP pin. Because the duration is measured as the number of CPU cycles occurring between the leading and trailing edges of the pulse, the time of each edge has the 1-µs resolution of the CPU clock. The duration can be calibrated as it is converted from CPU cycles to microseconds.

For measuring the execution time of an algorithm, it is the cycle count that provides the definitive measure, not the microseconds. The **Start**, **Stop**, and **Send** functions first used in Chapter Six are developed at the end of this chapter.

13.2 TIMER0 OPERATION

Timer0 is a 16-bit counter. Its use raises several issues that will be addressed in this section. Then it will be used in the next section to help calibrate F_{OSC}.

Timer0's operation as a 16-bit counter of F_{CPU} cycles is shown in Figure 13-1. The upper byte of the counter is buffered with the **TMR0H** register shown. Thus, a read of **TMR0H** does not read the upper byte of the counter itself, but rather the content of the upper byte of the counter at the instant that **TMR0L** was last read. This feature simplifies the reliable reading of Timer0 without first stopping its clocking by clearing the **TMR0ON** bit. Even if **TMR0L** rolls over from 255 to 0 between the read of **TMR0L** and a subsequent read of **TMR0H**, the 16-bit value read will be the value that was in the counter at the moment **TMR0L** was read.

In like manner, the content of Timer0 can be updated reliably even as the counter continues to count. First, **TMR0H** is loaded. Then, when **TMR0L** is written to, the 2 bytes of the counter itself are loaded simultaneously.

While buffering of the upper byte is mandatory for Timer0, it is an option for Timer1 and Timer3. If the timer is first stopped and then either read or loaded, the buffering does not help, and can actually get in the way. (For this reason, buffering for Timer1 and Timer3 was not used in Chapter Seven.)

Consider the addition of a 16-bit number, **NUMH:NUML**, to Timer0. The following code might be used:

```
TEMPL = TMR0L + NUML;         // Read 16-bit counter and add

TEMPH = TMR0H + (NUMH + STATUSbits.C);  // Add with carry

TMR0H = TEMPH;                // Load TMR0H buffer

TMR0L = TEMPL;                // Update 16-bit counter
```

Note that the read of the 16-bit counter takes place during the read of **TMR0L** in the first line. The write back into the 16-bit counter takes place during the write to **TMR0L** in the fourth line. Thus, the addition of **NUMH:NUML** into **TMR0H:TMR0L** while it continues to run, will result in somewhat less of a change in **TMR0H:TMR0L** than the number of counts in **NUMH:NUML**. For example, adding 0x0000 sets the counter back to what was in it when it was read with the first line of the code sequence listed above.

The use of Timer0 in the next section will actually start and stop the counter. Nevertheless, the buffering with **TMR0H** makes it imperative that the bytes be read (or written to) in the order dictated by Figure 13-1.

Before leaving this section, consider the Timer0 use shown in Figure 13-2. With **T0CON** initialized as shown, Timer0 is incremented by each rising edge into the Qwik&Low board's uncommitted **RA4/T0CKI** pin. The synchronizer assumes the edges of the input waveform occur at a slower rate than F_{OSC}. The synchronizer delays each input edge until the time in each CPU cycle when the circuit of Figure 13-1 increments the counter. Because of this, **TMR0L** can be read, or written to, by the CPU reliably.

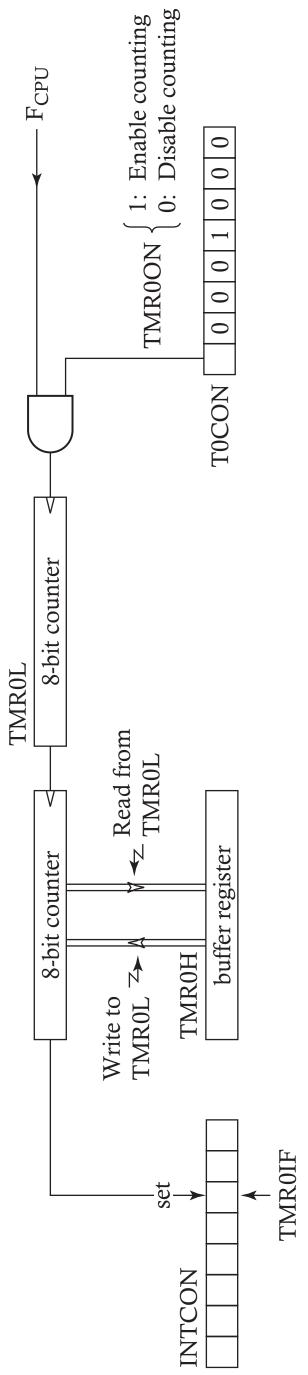


FIGURE 13-1 Timer0 use for counting F_{CPU} (1 MHz) clock periods

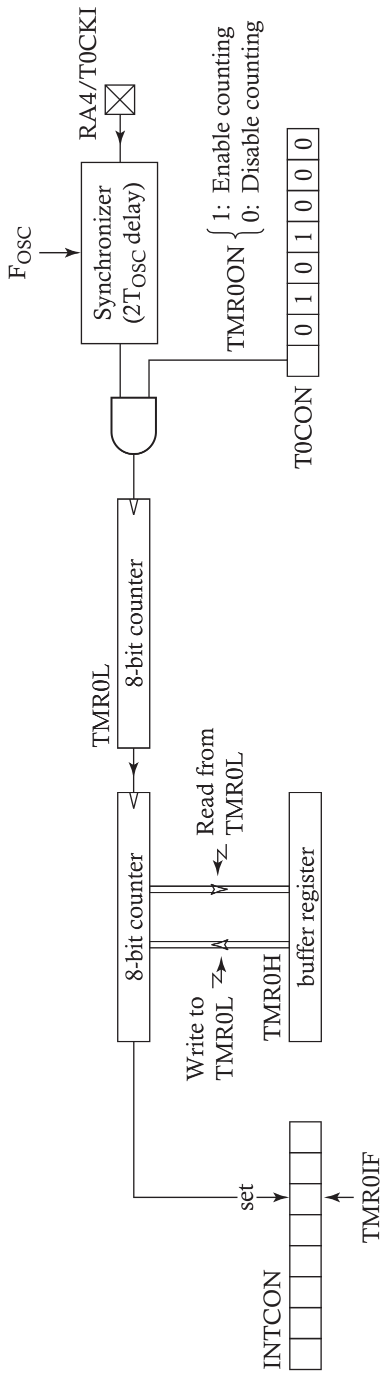


FIGURE 13-2 Timer0 use for counting input events (i.e, edges)

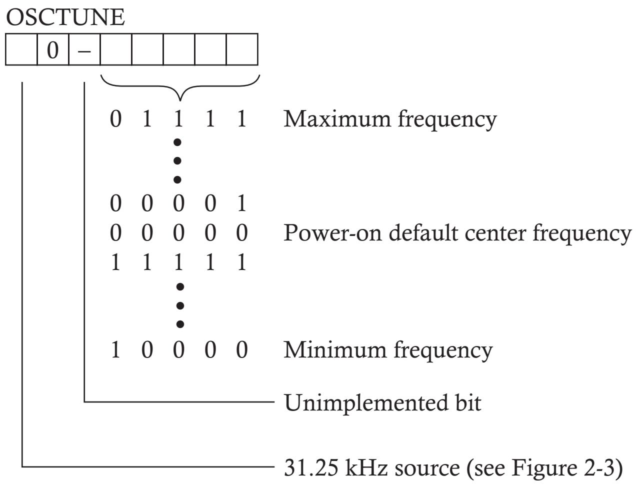

OSCTUNE
0
–
0 1 1 1 1 Maximum frequency
0 0 0 0 1
0 0 0 0 0 Power-on default center frequency
1 1 1 1 1
1 0 0 0 0 Minimum frequency
Unimplemented bit
31.25 kHz source (see Figure 2-3)

FIGURE 13-3 Tuning the internal oscillator

Timer0 also includes a prescaler option (not shown in either figure) for dividing the counter input edges by any multiple of 2 from 2 to 256. For information on this (or any) option, refer to the data manual on the www.microchip.com web site.

13.3 INTERNAL OSCILLATOR CALIBRATION

To make accurate timing measurements, the CPU clock can be calibrated against the 50-ppm-accurate Timer1 oscillator. This calibration can be done at startup. If it is done again, either periodically or in response to temperature change, this calibration can compensate for any subsequent drift in the internal oscillator frequency.

The calibration procedure counts 256 increments of the Timer1 oscillator while at the same time counting Timer0 from zero as it is clocked by the CPU's (nominal) 1-MHz clock. This time interval is

$$256 \times (1{,}000{,}000/32{,}768) = 7{,}812.5\ \mu\text{s}$$

The count will be formed in an *unsigned int* variable, **CALIB**. A subsequent time interval measurement formed in the *unsigned int* variable, **TIME**, can be converted from counts to microseconds with the calculation

$$\text{TIME} = (\text{TIME} \times 7{,}812)/\text{CALIB} \qquad \textbf{(13-1)}$$

Alternatively, if **CALIB** is less than 7,812, the internal oscillator is slow. It can be speeded up by incrementing the **OSCTUNE** register, as indicated in Figure 13-3. Successive increments or decrements of **OSCTUNE** to minimize the deviation from 7,812 may make the value of a **TIME** measurement sufficiently accurate for a specific purpose without resorting to Equation 13-1.

The calibration strategy suggested in this section makes use of the revised organization of program code shown in Figure 13-4. A *calibration supervisor* function, **CalSup1**, is inserted at the beginning of the main loop and another, **CalSup2**, is inserted at the end, just before the call of **LoopTime**. A *state variable*, **CAL**, lets each of these

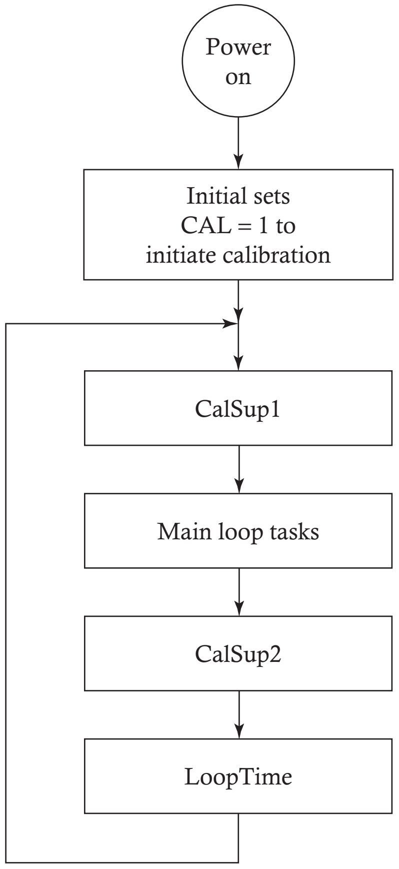


FIGURE 13-4 Reorganizing main loop for calibrating Fosc

two functions know where they are in the calibration process, from one loop to the next. Normally, **CAL** = 0 and neither function does anything more than just return. When **Initial**, or subsequently a main loop task, wants to initiate a calibration, it sets **CAL** = 1 ("Ready"). Figure 13-5 shows how the subsequent calibration sequence plays out. When **CalSup1** is next called (with **CAL** = 1), it stops and resets Timer0 to zero. It stops Timer1 and reloads it with 0xFFFE, clears its overflow flag, and enables its counting of the slow 32,768-Hz Timer1 clock again (having not missed a clock edge). When Timer1 has progressed two more counts and rolls over, Timer0 is started ("Go"). **TMR1H** is reloaded with 0xFF and its overflow flag is cleared. Then **CAL** is set to 2 so that the **CalSup2** function will be there waiting for the rollover of Timer1 after exactly 256 clock edges from the Timer1 oscillator.

After the main loop tasks have been completed for this pass around the loop, **CalSup2** is called. It sees **CAL** = 2 and takes charge of the calibration process. Instead of the chip returning to sleep immediately, **CalSup2** waits for **TMR1IF** to be set, at which time, Timer0 is stopped. Meanwhile, Timer1 is 2 + 256 counts on its way to the 328 counts that define the 10-ms loop time. So Timer1 is stopped, 0xBA is added to **TMR1L**, Timer1 is started again, and **TMR1H** is reloaded with 0xFF so that the next Timer1 overflow will occur 10 ms from the beginning of the loop, analogous to

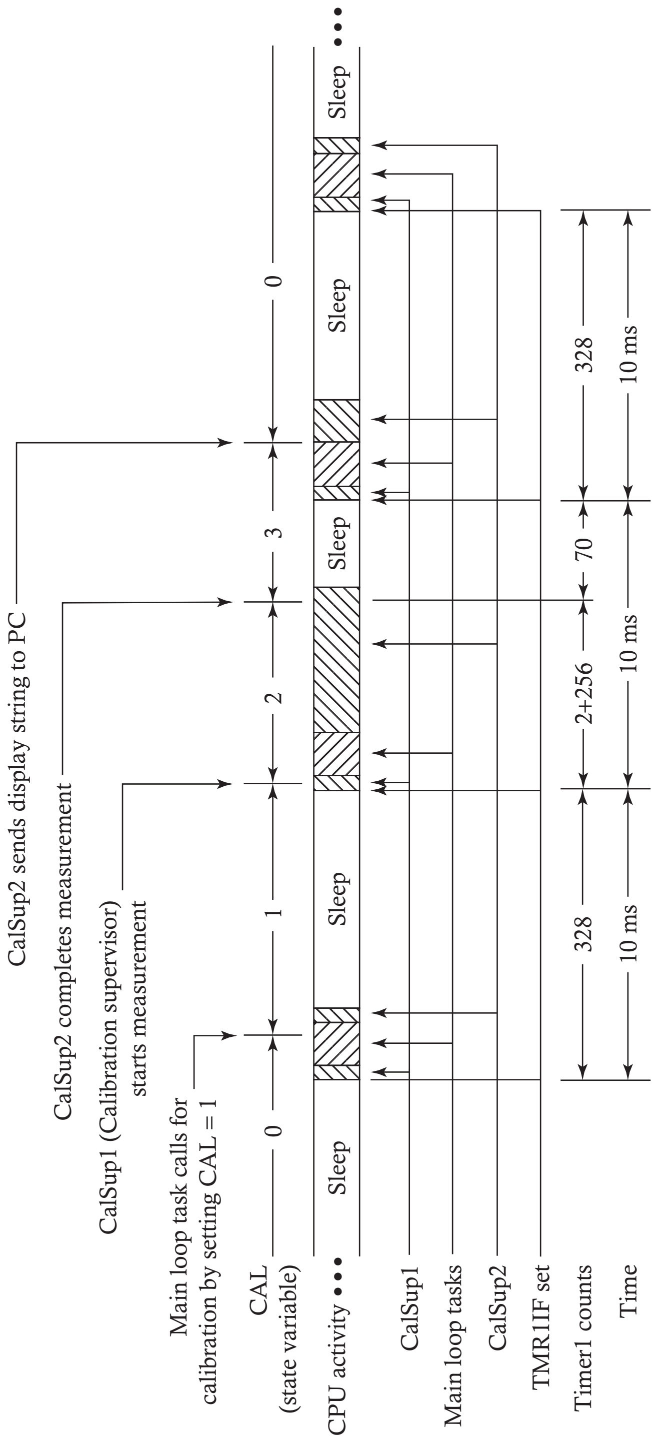

CalSup2 sends display string to PC
CalSup2 completes measurement
CalSup1 (Calibration supervisor) starts measurement
Main loop task calls for calibration by setting CAL = 1
CAL (state variable)
0
1
2
3
0
CPU activity •••
Sleep
Sleep
Sleep
Sleep
Sleep
•••
CalSup1
Main loop tasks
CalSup2
TMR1IF set
Timer1 counts
328
2+256
70
328
Time
10 ms
10 ms
10 ms

FIGURE 13-5 Calibration sequence

what is done by the LoPriISR of T3.c in Figure 7-6. Timer0 is read and used to form an *unsigned int* variable, **CALIB**, with

```
CAL1BL = TMR0L;              // Read in correct sequence

CALIBH = TMR0H;              // into unsigned int variables

CALIB = (CALIBH << 8) + CALIBL;  // Form CALIB = CALIBH:CALIBL
```

Finally, **CAL** is set to 3 and the **TMR1IF** flag is reset. On returning from the **CalSup2** function, the CPU calls the **LoopTime** function and returns to sleep for the remainder of the 10-ms loop time, as shown in Figure 13-5. On the next pass around the main loop, **CalSup2** converts **CALIB** to its four-digit ASCII representation and sends it to the PC for display.

The Calibrate.c file of Figure 13-6 illustrates the entire process. A **PBcalibrate** function is included that increments Figure 13-3's **OSCTUNE** each time the pushbutton is pushed. With **OSCTUNE** initialized to 0b00011111, the first press will produce the power-on default center frequency.

```
/******* Calibrate.c ***********
 *
 * Use Fosc = 4 MHz for Fcpu = Fosc/4 = 1 MHz.
 * Calibrate Fosc each time that the pushbutton is pressed, starting with
 * the nominal value and incrementing OSCTUNE with each subsequent press.
 * Display CALIB on PC.  CALIB = 7812 if Fosc/4 = 1 MHz exactly.
 * Use Timer1 to set loop time to 10 ms.
 * Blink LED on RD4 for 10 ms every 2.5 seconds.
 *
 ******* Program hierarchy *****
 *
 * main
 *    Initial
 *    InitTX
 *    CalSup1
 *    CalSup2
 *    BlinkAlive
 *    PBcalibrate
 *    LoopTime
 *
 *******************************
 */

#include <p18f4321.h>           // Define PIC18LF4321 registers and bits

/*******************************
 * Configuration selections
 *******************************
 */

#pragma config OSC = INTIO1     // Use internal osc, RA6=Fosc/4, RA7=I/O
#pragma config PWRT = ON        // Enable power-up delay
```

FIGURE 13-6 Calibrate.c

```
#pragma config LVP = OFF        // Disable low-voltage programming
#pragma config WDT = OFF        // Disable watchdog timer initially
#pragma config WDTPS = 4        // 16 millisecond WDT timeout period, nominal
#pragma config MCLRE = ON       // Enable master clear pin
#pragma config PBADEN = DIG     // PORTB<4:0> = digital
#pragma config CCP2MX = RB3     // Connect CCP2 internally to RB3 pin
#pragma config BOR = SOFT       // Brown-out reset controlled by software
#pragma config BORV = 3         // Brown-out voltage set for 2.0V, nominal
#pragma config LPT1OSC = OFF    // Deselect low-power Timer1 oscillator

/*******************************
 * Global variables
 *******************************
 */
unsigned char ALIVECNT;         // Scale-of-400 counter for blinking "Alive" LED
signed char i;                  // Index into strings
unsigned int DELAY;             // Sixteen-bit counter used by Delay macro
unsigned int TIMEL;             // Int version of TMR0L
unsigned int TIMEH;             // Int version of TMR0H
unsigned int CALIB;             // Calibration constant
unsigned char CAL;              // Calibration state variable
char OLDPB;                     // Old pushbutton state
char NEWPB;                     // New pushbutton state
char PCSTRING[] = "xxxx";       // String to represent 0 - 9999 value

/*******************************
 * Function prototypes
 *******************************
 */

void Initial(void);
void InitTX(void);
void BlinkAlive(void);
void CalSup1(void);
void CalSup2(void);
void PBcalibrate(void);
void LoopTime(void);

/*******************************
 * Macros
 *******************************
 */
#define Delay(x) DELAY = x; while(--DELAY){ Nop(); Nop(); }
#define TXascii(in)  TXREG = in; while(!TXSTAbits.TRMT)

/////// Main program //////////////////////////////////////////////////////////

/*******************************
 * main
 *******************************
 */
```

FIGURE 13-6 *(continued)*

```
void main()
{
   Initial();                    // Initialize everything
   InitTX();                     // Initialize UART's TX output
   while (1)
   {
      CalSup1();                 // First part of calibration supervisor
      PORTCbits.RC2 ^= 1;        // Toggle for looptime
      BlinkAlive();              // Blink "Alive" LED
      PBcalibrate();             // Calibrate each time Pushbutton is pressed
      Delay(600);                // Pause
      CalSup2();
      LoopTime();                // Sleep, letting watchdog timer wake up chip
   }
}
/*******************************
 * Initial
 *
 * This function performs all initializations of variables and registers.
 *******************************
 */

void Initial()
{
   OSCCON = 0b01100010;          // Use Fosc = 4 MHz (Fcpu = 1 MHz)
   ADCON1 = 0b00001011;          // RA0,RA1,RA2,RA3 pins analog; others digital
   TRISA = 0b00001111;           // Set I/O for PORTA
   TRISB = 0b01000100;           // Set I/O for PORTB
   TRISC = 0b10000000;           // Set I/O for PORTC
   TRISD = 0b10000000;           // Set I/O for PORTD
   TRISE = 0b00000010;           // Set I/O for PORTE
   PORTA = 0;                    // Set initial state for all outputs low
   PORTB = 0;
   PORTC = 0;
   PORTD = 0b00100000;           // except RD5 that drives LCD interrupt
   PORTE = 0;
   Delay(50000);                 // Pause for half a second
   RCONbits.SBOREN = 0;          // Now disable brown-out reset
   ALIVECNT = 247;               // Blink immediately
   OLDPB = 0;                    // Initialize pushbutton flags
   NEWPB = 0;
   CAL =0;                       // Calibration state variable - do nothing yet
   OSCTUNE = 0b00011111;         // CALIB = default value after first PB push
   PIE1bits.TMR1IE = 1;          // Enable local interrupt source
   TMR1H = 0xFF;                 // Initial value
   TMR1L = 0x00;                 //
   T1CON = 0b01001111;           // Timer1 runs from 32768 Hz oscillator
   PIR1bits.TMR1IF = 0;          // Clear Timer1 flag
   INTCONbits.GIEL = 1;          // Enable wake-up from sleep
}
```

FIGURE 13-6 *(continued)*

```
/******************************
 * InitTX
 *
 * This function initializes the UART for its TX output function.  It assumes
 * Fosc = 4 MHz.  For a different oscillator frequency, use Figure 6-3c to
 * change BRGH and SPBRG appropriately.
 ******************************
 */

void InitTX()
{
    RCSTA = 0b10010000;         // Enable UART
    TXSTA = 0b00100000;         // Enable TX
    SPBRG = 12;                 // Set baud rate
    BAUDCON = 0b00111000;       // Invert TX output
}

/******************************
 * BlinkAlive
 *
 * This function briefly blinks the LED every four seconds.
 * With a looptime of about 10 ms, count 250 looptimes.
 ******************************
 */

void BlinkAlive()
{
   PORTDbits.RD4 = 0;           // Turn off LED
   if (++ALIVECNT == 250)       // Increment counter and return if not 250
   {
      ALIVECNT = 0;             // Reset ALIVECNT
      PORTDbits.RD4 = 1;        // Turn on LED for one looptime
   }
}

/******************************
 * PBcalibrate
 *
 * Calibrate each time pushbutton is pressed with an increase calibration value
 ******************************
 */
void PBcalibrate()
{
   PORTEbits.RE0 = 1;           // Power up the pushbutton
   Nop();                       // Delay one microsecond before checking it
   NEWPB = !PORTDbits.RD7;      // Set flag if pushbutton is pressed
   PORTEbits.RE0 = 0;           // Power down the pushbutton
   if (!OLDPB && NEWPB)         // Look for last time = 0, now = 1
   {
      CAL = 1;                  // Initiate a calibration sequence
```

FIGURE 13-6 *(continued)*

```
      ++OSCTUNE;                 // Increment calibration value
   }
   OLDPB = NEWPB;                // Save present pushbutton state
}

/******************************
 * CalSup1
 *
 * First part of calibration supervisor.
 *
 ******************************
 */
void CalSup1()
{
   if (CAL == 1)
   {
      T0CON = 0b00001000;        // Timer0 stopped
      TMR0H = 0;                 // Timer0 = 0
      TMR0L = 0;
      T1CON = 0b01001110;        // Stop Timer1
      TMR1H = 0xFF;              // Wait for exact rollover
      TMR1L = 0xFE;
      PIR1bits.TMR1IF = 0;       // Clear flag
      T1CONbits.TMR1ON = 1;      // Start Timer1 again
      while (!PIR1bits.TMR1IF); // and wait for it to set again
      T0CONbits.TMR0ON = 1;      // Start Timer0
      TMR1H = 0xFF;              // Next overflow in 256 counts of Timer1
      PIR1bits.TMR1IF = 0;       // Clear flag (if not already cleared)
      CAL = 2;
   }
}

 /******************************
 * CalSup2
 *
 * Second part of calibration supervisor.
 *
 ******************************
 */
void CalSup2()
{
   if (CAL == 2)                 // Collect measurement
   {
      while (!PIR1bits.TMR1IF); // Wait for Timer1 to roll over and set flag
      T0CONbits.TMR0ON = 0;      // Stop Timer0
      T1CONbits.TMR1ON = 0;      // Pause Timer1 counter
      TMR1L += 0xBB;             // Cut out remaining counts of Timer1
      T1CONbits.TMR1ON = 1;      // Resume Timer1 counter
      TMR1H = 0xFF;              // Upper byte of Timer1 will be 0xFF
      PIR1bits.TMR1IF = 0;       // Clear interrupt flag
      TIMEL = TMR0L;             // Read Timer0 in correct sequence
```

FIGURE 13-6 *(continued)*

```
      TIMEH = TMR0H;
      CALIB = (TIMEH << 8) +TIMEL;  // Form calibration factor
      CAL = 3;                    // Done with calibration sequence
   }
   else if (CAL == 3)             // Display result
   {
      for (i = 3; i >= 0; --i)  // Form digits and display on PC
      {
         PCSTRING[i] = (CALIB % 10) + 0x30;
         CALIB = CALIB / 10;
      }
      for (i = 0; i <= 3; ++i)
      {
         TXascii(PCSTRING[i]);
      }
      TXascii(0x0D);              // Carriage return
      TXascii(0x0A);              // Line feed
      CAL = 0;
   }
}

/*******************************
 * LoopTime
 *
 * This function puts the chip to sleep, to be awakened by Timer1 rollover.
 *******************************
 */
void LoopTime()
{
   Sleep();
   Nop();
   T1CONbits.TMR1ON = 0;         // Pause Timer1 counter
   TMR1L += 0xB9;                // Cut out all but 328 counts of Timer1
   T1CONbits.TMR1ON = 1;         // Resume Timer1 counter
   TMR1H = 0xFE;                 // Upper byte of Timer1 will be 0xFE
   PIR1bits.TMR1IF = 0;          // Clear interrupt flag
}
```

FIGURE 13-6 *(continued)*

13.4 EXTERNAL TIME MEASUREMENT

The PIC18LF4321 has two CCP (capture, compare, PWM) modules that can be set up in a variety of ways to enhance the functionality of the chip's timers. Consider the circuit of Figure 13-7. Either the **CCP1** input or the **CCP2** input can be used to measure the duration of an input pulse. For example, if the **CCP1** input is used, then **CCP1CON** is initialized to 0b00000101, as shown, and the **CCP1IF** flag is cleared. When the flag is set by the reception of a rising edge, **CCPR1** is copied to an *unsigned int* **LEADINGEDGE** variable, the **CCP1M0** edge-select bit is cleared, and the **CCP1IF** flag is cleared. When this flag is set again (while the chip remains awake,

```
TRISCbits.TRISC2 = for CCP1 input          or          TRISBbits.TRISB3 = 1 for CCP2 input
T3CON     = 0b01001001  to clock Timer3 from F_CPU (1 MHz)
CCP1CON = 0b00000100  for falling edge      or     0b00000101 for rising edge on CCP1 pin
or
CCP2CON = 0b00000100  for falling edge      or     0b00000101 for rising edge on CCP2 pin
```

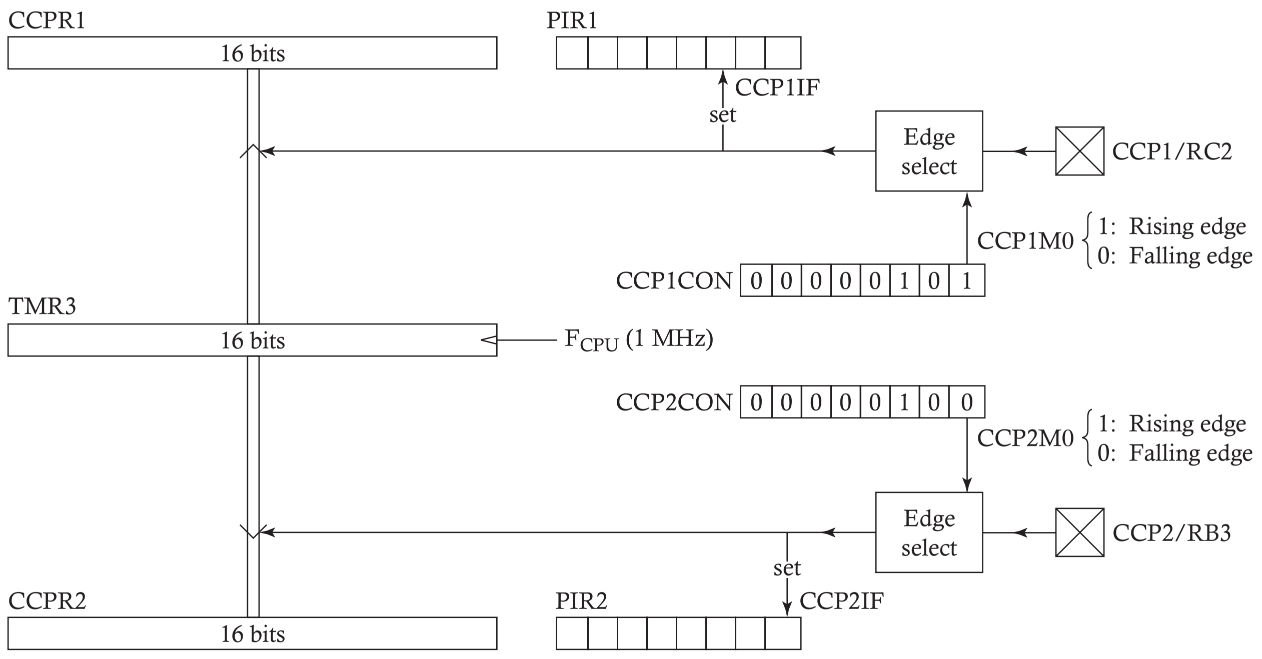


FIGURE 13-7 Capture mode operation for CCP1 or CCP2 inputs

to continue clocking **TMR3**), **CCPR1** is copied to an *unsigned int* **TRAILINGEDGE** variable. Then

```
    TIME = TRAILINGEDGE – LEADINGEDGE;
```

Note that with all three variables being declared to be unsigned integers, the difference will be correct even if Timer3 rolls over between the leading edge capture and the trailing edge capture. For example, the hex subtraction

0002 – FFFF

produces a difference of 0003 (plus a borrow that is lost to the 16-bit unsigned result).

This capture mode determination of an input pulse width produces a timing resolution of 1 μs for each edge. However, it is important to read the captured time before the same edge occurs again and overwrites the previously captured time. Because the flag bits can be set up to interrupt the CPU, the CPU can actually be undertaking other tasks while the measurement is under way. However, the CPU cannot be allowed to sleep during the measurement because that will stop the 1-MHz clocking of Timer3.

13.5 START, STOP, AND SEND FUNCTIONS

An opportunity afforded by the serial test port and its connection to the PC is its availability for displaying the number of CPU clock cycles required to execute a function or code segment. The tools needed are the **Start**, **Stop** and **Send** functions, already used by the Measure.c program of Figure 6-9. These functions are being explained here based on the present understanding of Time0's use in the configuration of Figure 13-1.

- **Start** – This function initializes Timer0 for counting CPU cycles, clears Timer0, and then starts the clocking of Timer0 with the 1-MHz CPU clock.
- **Stop** – This function stops the counting of Timer0 and forms the number of CPU cycles occurring between **Start** and **Stop** in the unsigned *int* variable, **CYCLES**.
- **Send** – This function converts **CYCLES** to its decimal representation as a number ranging from 0 to 9,999 CPU cycles, expressed as four ASCII-coded digits and sends the resulting number to QwikBug's Console display on the PC.

The three functions are listed in Figure 13-8.

PROBLEMS

13-1 Adding into Timer0 A user, unaware of the buffering of **TMR0H** shown in Figure 13-1, might make the mistake of adding **NUMH:NUML** to **TMR0H:TMR0L** with

```
TMR0L += NUML;

TMR0H += (NUMH + STATUSbits.C);
```

What would be the result?

```
/*******************************
 * Start
 *
 * This function clears Timer0 and then starts it counting.
 *******************************
 */
void Start()
{
   T0CON = 0b00001000;      // Set up Timer0 to count CPU clock cycles
   TMR0H = 0;               // Clear Timer0
   TMR0L = 0;
   T0CONbits.TMR0ON = 1;    // Start counting
}
```

(a) Start function.

```
/*******************************
 * Stop
 *
 * This function stops counting Timer0, and reads the result into CYCLES.
 *******************************
 */
void Stop()
{
   T0CONbits.TMR0ON = 0;    // Stop counting
   CYCLES = TMR0L;          // Form CYCLES from TMR0H:TMR0L
   CYCLES += (TMR0H * 256);
   CYCLES -= 3;             // Remove 3 counts so back-to-back Start-Stop
}                           // functions produce CYCLES = 0
```

(b) Stop function.

```
/*******************************
 * Send
 *
 * This function converts CYCLES to four ASCII-coded digits and sends
 * the result to the PC for display.
 *******************************
 */
void Send()
{
   BIGNUM = CYCLES;         // Load ASCII4's input parameter
   ASCII4();                // Convert
   TXascii('\r');           // Send carriage return
   TXascii('\n');           // Send line feed
   TXascii(THOUSANDS);      // Send four-digit number
   TXascii(HUNDREDS);
   TXascii(TENS);
   TXascii(ONES);
}
```

(c) Send function.

FIGURE 13-8 Start, Stop, and Send functions

13-2 Clock-edge synchronization A look at the .lst file of some code that employs the **Start** and **Stop** functions of Figure 13-8 shows that a two-cycle subroutine return instruction follows the setting of the **TMR0ON** bit at the end of the **Start** function. A two-cycle subroutine call instruction is executed before the **TMR0ON** bit is cleared at the beginning of the **Stop** function. Noting that the setting of a port pin immediately followed by the clearing of the pin produces a 1-μs pulse, so the setting of **TMR0ON** immediately followed by the clearing of **TMR0ON** ought to produce a count of one in the previously cleared Timer0 counter. Consequently, the back-to-back calls of the Start and Stop functions of Figure 13-8 ought to produce a count of 2 + 1 + 2 = 5 and require a correction of

```
CYCLES -= 5;
```

not the

```
CYCLES -= 3;
```

shown in the **Stop** function. The two-cycle difference occurs because of a clock synchronizer circuit. This circuit is shown in Figure 13-2 because of its important role in dealing with an unsynchronized clock input, **T0CKI**. Because it is a distraction, it has been left out of the simplified schematic of Figure 13-1. However, unlike the clock synchronizer associated with Timer1, it cannot be bypassed via the setting or clearing of a bit in a control register.

To verify the loss of the first two counts after **TMR0ON** is set, write a little test program that clears **TRM0H**, clears **TMR0L**, sets **TMR0ON**, clears **TMR0ON**, and then displays **TMR0L**. Repeat this by inserting one **Nop** macro, then two **Nop** macros, and finally three **Nop** macros between the setting and clearing of the **TMR0ON** bit. What is the result in each case, and how do you (now) explain it?

13-3 Oscillator calibration Run the Calibrate.c program.

a) What is the frequency increment corresponding to a single increment of the **OSCTUNE** register?

b) What value of **OSCTUNE** will produce the integer value of **CALIB** that is closest to 7,812.5, the value corresponding exactly to $F_{CPU} = 1.0000$ MHz?

c) Form a new CalibrateX.c program that uses the RPG to increment or decrement the five frequency controlling bits of **OSCTUNE**. Use the pushbutton to determine a new value of **CALIB**. Display on the LCD both of these values, using the format

```
±D CCCC
```

where ±D represents the signed frequency-setting number in **OSCTUNE** (i.e., −1, 0, +1, etc.) and
CCCC represents the value of **CALIB**. Whenever the RPG is turned, blank CCCC until the pushbutton is next pressed.

d) Use CalibrateX.c to tune the oscillator as close to 7,812 as possible. Then, without changing the tuning, record **CALIB** immediately and 1 and 2 min later.

e) Create a CalibrateY.c program that, in contrast to CalibrateX.c, never sleeps. That is, modify the **LoopTime** function so that it just waits for Timer1 to roll over before reloading the value in Timer1. The intent is to then repeat Part (d) with the chip drawing an average current of about a milliampere. See whether any attendant heating within the chip affects the calibration.

13-4 External Time Measurement The **Display** function that is used throughout this book clears **RD5**, sends the nine ASCII-coded characters in **LCDSTRING** to the LCD, and then sets **RD5**. Connect the TP10 test point **(RD5)** to the **RC2/CCP1** pin on the H4 proto area header. Then write a little test program that uses the CCP1 circuit of Figure 13-7 to capture the number of CPU clock cycles between the falling edge and the rising edge. Send the resulting number to QwikBug for display in its Console window.

13-5 Start, **Stop**, **and Send functions** Repeat Problem 13-4 by modifying the **Display** function with the **Start**, **Stop**, and **Send** functions to make the same measurement by counting cycles from the instruction that clears **RD5** at the beginning of **Display** to the instruction that sets **RD5** at the end. Then display the result on the PC.

Chapter

14

EEPROM (EETEST.c)

14.1 OVERVIEW

The PIC18LF4321 contains 256 bytes of *data EEPROM*, hereafter referred to as EEPROM. This is *nonvolatile* memory that can be written to a byte at a time, the same as a RAM location. Reads from the EEPROM take just one CPU cycle, the same as RAM. In contrast to RAM, EEPROM's nonvolatility means that its data will be retained even when power is removed from the chip and then subsequently restored. Also, unlike RAM, for which a write operation takes just one cycle, a write to a byte of EEPROM typically takes 4 ms, with the write operation being self-timed by the EEPROM module. This chapter describes EEPROM use.

14.2 EEPROM USE

Having nonvolatile memory available is an invaluable resource across the full spectrum of technical activity:

- Automotive: security system, digital radio, seat control, . . .
- Communication: cell phone personalization, answering machines, . . .
- Consumer: digital TV, monitors, home appliances, games, . . .

- Industrial: instruments, access cards, motor control, alarms, . . .
- Computer: motherboards, wireless optical mice, hard drives, . . .

Many applications require much more that the 256 bytes of EEPROM available in the PIC18LF4321, and instead typically use 1 Kbit–1 Mbit EEPROM parts having a serial interface. Many other applications are served well by a small number of bytes. One example is the exposure calibration adjustment built into a digital camera. Another is channel selection information built into a TV remote. When an oscilloscope has the ability to retain and restore a complicated setup state, this indicates the presence of a small EEPROM. When a lawn sprinkler controller remembers its settings through a power glitch or a power outage, this points to a small EEPROM. When the Qwik&Low board powers up and retrieves the most recently determined clock calibration byte from EEPROM and loads it in the **OSCTUNE** register, this represents a use of the PIC18LF4321's built-in EEPROM.

14.3 EEPROM REGISTERS AND FUNCTIONS

The 256 bytes of EEPROM on the PIC18LF4321 are accessed indirectly, by way of the registers listed in Figure 14-1. Reading a byte is carried out by writing the address to **EEADR**, setting the **RD** bit in the **EECON1** register, and then reading the result from **EEDATA**. In contrast, writing is a safeguarded process, to reduce the chance of a runaway CPU overwriting important information. For example, as the chip is powering down with the brown-out reset module enabled, the CPU will go directly from executing code properly to being reset. On the other hand, if the brown-out reset module is not enabled, then as the chip is powering down, the CPU can find its program counter bits being corrupted, leading to a jump to anywhere in the program memory and executing unintended code, including an EEPROM-write instruction sequence. Protecting this EEPROM-write instruction sequence will be addressed in Section 14.5.

For an application using the EEPROM memory, Figure 14-2a lists a line to be added to the **Initial** function. By clearing the upper 4 bits of **EECON1**, this line sets the destination of subsequent reads and writes to be the EEPROM memory module. Figure 14-2b lists a line to be added to the beginning of the main loop. Because each write lasts about 4 ms and because most of this time will usually be spent while the CPU has completed its other useful tasks and returned to sleep for the last of the 10-ms loop time, this line is executed well after the write of a byte has been completed. It disables any further write operations until a subsequent write sequence reenables it.

The **EEread** function of Figure 14-2c begins by testing the **WR** bit in the **EECON1** register to determine whether a write operation has already been initiated during this loop time. If so, it waits for that write operation to be completed before initiating the read operation. Of course, to minimize coin cell current, it is better to organize the program code within the main loop so that all EEPROM reads occur before any EEPROM writes take place. Given this, the chip can be asleep during most of the write operation.

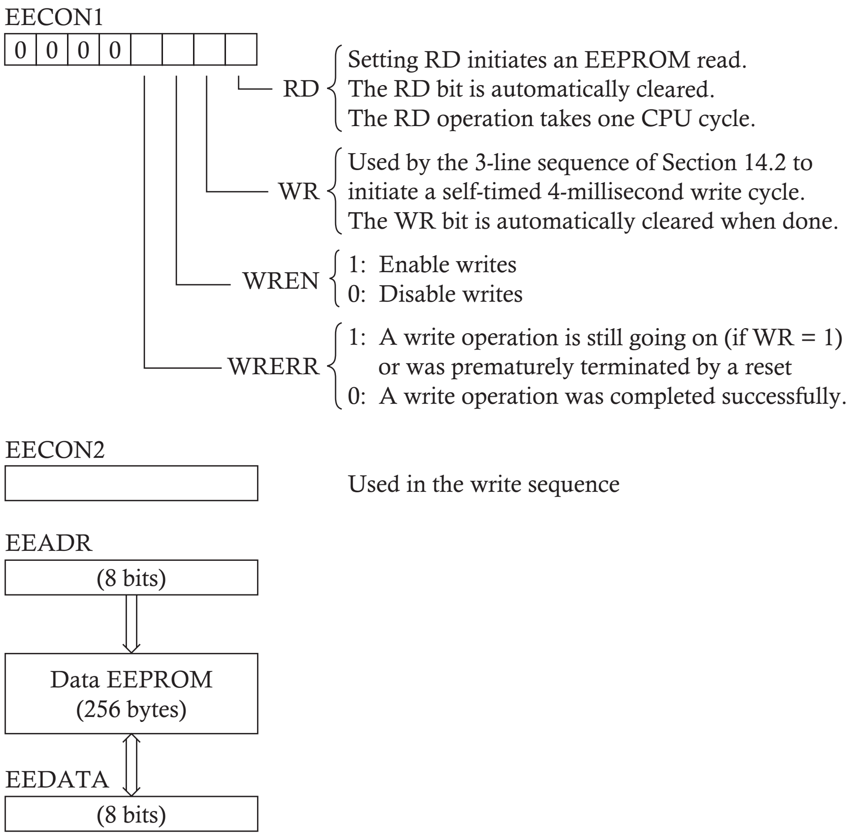

EECON1
0 0 0 0
RD
Setting RD initiates an EEPROM read.
The RD bit is automatically cleared.
The RD operation takes one CPU cycle.
WR
Used by the 3-line sequence of Section 14.2 to
initiate a self-timed 4-millisecond write cycle.
The WR bit is automatically cleared when done.
WREN
1: Enable writes
0: Disable writes
WRERR
1: A write operation is still going on (if WR = 1)
or was prematurely terminated by a reset
0: A write operation was completed successfully.
EECON2
Used in the write sequence
EEADR
(8 bits)
Data EEPROM
(256 bytes)
EEDATA
(8 bits)

FIGURE 14-1 EEPROM registers

The **EEwrite** function of Figure 14-2d likewise begins with a test to see if a write is taking place. If so, it waits for the completion of the write before proceeding. Similar to the case of the **EEread** function, it is better to organize program code so that only one EEPROM write occurs during each pass around the main loop. The **EEwrite** function then enables the brown-out reset module. The module's 34-μA current draw will continue for about 10 μs. The brown-out reset module will ensure that a powering-down CPU that may have inadvertently been led into code that called the **EEwrite** function will proceed no further.

Because the three-line sequence:

```
EECON2 = 0x55;              // Write first key
EECON2 = 0xAA;              // Write second key
EECON1bits.WR = 1;          // Set WR bit to initiate write
```

must be executed consecutively with no gaps in between, interrupts are suspended (if they were enabled) during the write operation. At the end of the function, both **GIE** and **SBOREN** are restored to their former state.

```
EECON1 = 0;                       // Initialize module state
```

(a) Addition to the Initial subroutine.

```
EECON1bits.WREN = 0;              // Disable further EEPROM writes
```

(b) Instruction to add to the beginning of the main loop, to disable WREN after any EEwrite during the previous loop.

```
/****************************
* EEread
*
* This function reads from the EEPROM address identified by EEADR
* into EEDATA.
****************************
*/
void EEread()
{
   while (EECON1bits.WR);         // Wait on the completion of any ongoing write
   EECON1bits.RD = 1;
}
```

(c) EEread function.

```
/****************************
* EEwrite
*
* This function writes the data contained in EEDATA into the EEPROM
* address identified by EEADR.
* The write is self-timed and takes about 4 milliseconds.
****************************
*/
void EEwrite()
{
   while (EECON1bits.WR);               // Complete any ongoing write
   SBORENCOPY = RCONbits.SBOREN = 1;    // Copy SBOREN for subsequent restore
   RCONbits.SBOREN = 1;                 // Enable brown-out reset
   GIEHCOPY = INTCONbits.GIEH;          // Copy GIEH for subsequent restore
   INTCONbits.GIEH = 0;                 // Disable all interrupts
   EECON1bits.WREN = 1;                 // Enable write operation
   EECON2 = 0x55;                       // Write first key
   EECON2 = 0xAA;                       // Write second key
   EECON1bits.WR = 1;                   // Set WR bit to initiate write
   INTCONbits.GIEH = GIEHCOPY;          // Restore global interrupt state
   RCONbits.SBOREN = SBORENCOPY;        // Restore brown-out reset state
}
```

(d) EEwrite function.

FIGURE 14-2 EEPROM functions

14.4 MULTIPLE WRITE SEQUENCES

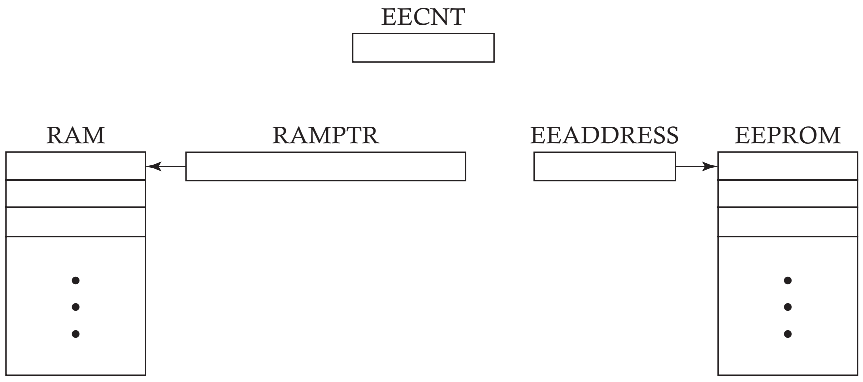
Quite often an update of data stored in EEPROM involves a sequence of writes. If the writes are destined for consecutive EEPROM addresses, the **EEArrayWrite** function of Figure 14-3 will serve. Before the function is called, the bytes are first written into consecutive RAM addresses and **RAMPTR** is loaded with the first of these addresses. Then **EEADDRESS** is loaded with the first EEPROM destination address, and **EECNT** is loaded with the number of bytes to be copied. The **EEArrayWrite** function can be called every time around the main loop. It acts only if **EECNT** is nonzero. Its action is to copy 1 byte, decrement **EECNT** and return. When all bytes have been copied, **EECNT** will have been decremented to zero so that calls of **EEArrayWrite** during subsequent passes around the main loop do nothing until the RAM array, the two pointers and **EECNT** have been reloaded for a new multiple-byte write sequence.

The example code of Figure 14-4a implements the example listed in the header of Figure 14-3e to execute two writes to EEPROM addresses 0x20 and 0x21. It precedes these two writes with a write to EEPROM address 0x10. The logic analyzer capture of these three writes is shown in Figure 14-4b and expanded for the second write in Figure 14-4c. To help identify when each write takes place, **RC2** is toggled and a 1-ms delay is inserted. This delay keeps the chip awake and is reflected in the 1 ms of $F_{OSC}/4$ clocking after each write. Figure 14-4d displays the resulting content of EEPROM as read by the PICkit 2 programmer.

14.5 PROTECTING THE WRITE SEQUENCE

The discussion associated with the **EEwrite** function of Figure 14-2 has already addressed the recurring problem of applications for which power is turned off as a matter of regular practice. By enabling the brown-out reset feature of the chip for

FIGURE 14-3 EEArrayWrite functions



(a) Registers

```
char GIEHCOPY;                    // Copy of GIEH bit for EEPROM writes
char SBORENCOPY;                  // Copy of SBOREN bit for EEPROM writes
char* RAMPTR;                     // Pointer to RAM array to be copied to EEPROM
char TEMPARRAY[] = {'a','b','c','d'};  // Characters to copy
unsigned char EEADDRESS;          // Starting address in EEPROM
unsigned char EECNT;              // Number of bytes to copy
```

(b) Global variable additions.

```
void EEread(void);
void EEwrite(void);
void EEArrayWrite(void);
```

(c) Function prototype additions.

```
EECON1 = 0;                       // Initialize module state
RAMPTR = TEMPARRAY + 1;           // Experiment with EEPROM
EEADDRESS = 0x20;
EECNT = 2;                        // Set up for writes 2 and 3
```

(d) Additions for Initial function.

```
/***************************
 * EEArrayWrite
 *
 * Each time this function is called with EECNT != 0, it writes one byte
 * from the RAM char[] pointed by RAMPTR, into the EEPROM location whose
 * address is in EEADDRESS. Then it increments RAMPTR and EEADDRESS and
 * decrements EECNT. RAMPTR, EEADDRESS and EECNT should not be modified
 * by other code until EECNT is 0.
 * Variable declaration:
 * char* RAMPTR;      unsigned char EEADDRESS;      unsigned char EECNT;
 * Example usage:
 *  char TEMPARRAY[] = {'a','b','c','d'};
 *  RAMPTR = TEMPARRAY + 1;
 *  EEADDRESS = 0x20;
 *  EECNT = 2;
 *  After 2 calls of EEArrayWrite, EECNT = 0;
 *  and EEPROM has 'b' and 'c' in address 0x20 and 0x21 respectively.
 ***************************
 */
void EEArrayWrite()
{
   if(EECNT && !EECON1bits.WR)
   {
      EEDATA = *(RAMPTR++);
      EEADR = EEADDRESS++;
      EEwrite();
      EECNT--;
   }
}
```

(e) Function.

FIGURE 14-3 *(continued)*

```
/******* EEtest.c *************
 *
 *  Test Alex's example.
 *  Show EEwrites by awake times on scope (using Delay(100)).
 *
 ******* Program hierarchy *****
 *
 * main
 *  Initial
 *
 ******************************
 */

#include <p18f4321.h>            // Define PIC18LF4321 registers and bits

/******************************
 * Configuration selections
 ******************************
 */

#pragma config OSC = INTIO1      // Use internal osc, RA6=Fosc/4, RA7=I/O
#pragma config PWRT = ON         // Enable power-up delay
#pragma config LVP = OFF         // Disable low-voltage programming
#pragma config WDT = OFF         // Disable watchdog timer initially
#pragma config WDTPS = 4         // 16 millisecond WDT timeout period, nominal
#pragma config MCLRE = ON        // Enable master clear pin
#pragma config PBADEN = DIG      // PORTB<4:0> = digital
#pragma config CCP2MX = RB3      // Connect CCP2 internally to RB3 pin
#pragma config BOR = SOFT        // Brown-out reset controlled by software
#pragma config BORV = 3          // Brown-out voltage set for 2.1V, nominal
#pragma config LPT1OSC = OFF     // Deselect low-power Timer1 oscillator

/******************************
 * Global variables
 ******************************
 */
char GIEHCOPY;                   // Copy of GIEH bit for EEPROM writes
char SBORENCOPY;                 // Copy of SBOREN bit for EEPROM writes
char* RAMPTR;                    // Pointer to RAM array to be copied to EEPROM
char TEMPARRAY[] = {'a','b','c','d'};  // Characters to copy
unsigned char EEADDRESS;         // Starting address in EEPROM
unsigned char EECNT;             // Number of bytes to copy
unsigned int DELAY;              // Sixteen-bit counter for obtaining a delay

/******************************
 * Function prototypes
 ******************************
 */

void Initial(void);
void EEread(void);
```

FIGURE 14-4 Multiple EEPROM writes

```
void EEwrite(void);
void EEArrayWrite(void);

/******************************
 * Macros
 ******************************
 */
#define Delay(x) DELAY = x; while(--DELAY){ Nop(); Nop(); }

/////// Main program //////////////////////////////////////////////////////////////

/******************************
 * main
 ******************************
 */

void main()
{
   Initial();              // Initialize everything


   EEADR = 0x10;
   EEDATA = 0xAA;
   EEwrite();              // Do the first write              // Write 1

   while (1)
   {
      EECON1bits.WREN = 0; // Disable further EEPROM writes
      EEArrayWrite();      // Deal with EEPROM writes         // Writes 2 and 3
      Sleep();             // Sleep through writes to EEPROM  // Check this
      Nop();
   }
}

/******************************
 * Initial
 *
 * This function performs all initializations of variables and registers.
 ******************************
 */
void Initial()
{
  OSCCON = 0b01100010;     // Use Fosc = 4 MHz (Fcpu = 1 MHz)
  SSPSTAT = 0b00000000;    // Set up SPI for output to LCD
  SSPCON1 = 0b00110000;
  ADCON1 = 0b00001011;     // RA0,RA1,RA2,RA3 pins analog; others digital
  TRISA = 0b00001111;      // Set I/O for PORTA
  TRISB = 0b01000100;      // Set I/O for PORTB
  TRISC = 0b10000000;      // Set I/O for PORTC
  TRISD = 0b10000000;      // Set I/O for PORTD
  TRISE = 0b00000010;      // Set I/O for PORTE
  PORTA = 0;               // Set initial state for all outputs low
```

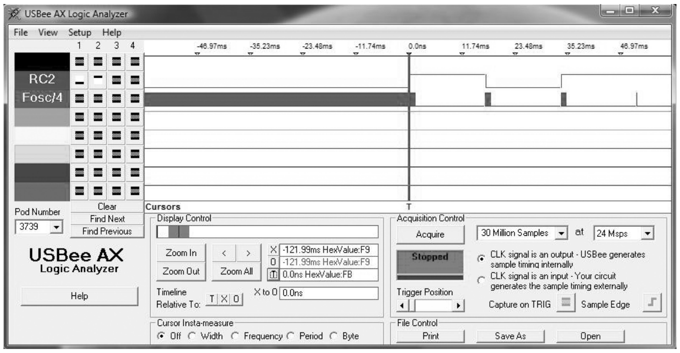
FIGURE 14-4 *(continued)*

```
  PORTB = 0;
  PORTC = 0;
  PORTD = 0b00100000;           // except RD5 that drives LCD interrupt
  PORTE = 0;
  Delay(50000);                 // Pause for half a second
  RCONbits.SBOREN = 0;          // Now disable brown-out reset
  WDTCONbits.SWDTEN = 1;        // Enable watchdog timer


  EECON1 = 0;                   // Initialize module state
  RAMPTR = TEMPARRAY + 1;       // Experiment with EEPROM
  EEADDRESS = 0x20;
  EECNT = 2;                    // Set up for writes 2 and 3
}

/*****************************
 * EEArrayWrite
 *
 * Each time this function is called with EECNT != 0, it writes one byte
 * from the RAM char[] pointed by RAMPTR, into the EEPROM location whose
 * address is in EEADDRESS. Then it increments RAMPTR and EEADDRESS and
 * decrements EECNT. RAMPTR, EEADDRESS and EECNT should not be modified
 * by other code until EECNT is 0.
 * Variable declaration:
 * char* RAMPTR;      unsigned char EEADDRESS;  unsigned char EECNT;
 * Example usage:
 *  char TEMPARRAY[] = {'a','b','c','d'};
 *  RAMPTR = TEMPARRAY + 1;
 *  EEADDRESS = 0x20;
 *  EECNT = 2;
 *  After 2 calls of EEArrayWrite, EECNT = 0;
 *  and EEPROM has 'b' and 'c' in address 0x20 and 0x21 respectively.
 *****************************
 */
void EEArrayWrite()
{
   if(EECNT && !EECON1bits.WR)   // Skip to next loop time for second write
   {
      EEDATA = *(RAMPTR++);
      EEADR = EEADDRESS++;
      EEwrite();
      EECNT--;
   }
}

/*****************************
 * EEread
 *
 * This function reads from the EEPROM address identified by EEADR
 * into EEDATA.
 *****************************
 */
```

FIGURE 14-4 *(continued)*

```
void EEread()
{
   while (EECON1bits.WR);               // Wait on the completion of any write
   EECON1bits.RD = 1;                   // Read from EEADR into EEDATA
}

/***************************
 * EEwrite
 *
 * This function writes the data contained in EEDATA into the EEPROM
 * address identified by EEADR.
 * The write is self-timed and takes about 4 milliseconds.
 ***************************
 */
void EEwrite()
{
   while (EECON1bits.WR);               // Wait on the completion of any write
   SBORENCOPY = RCONbits.SBOREN;        // Copy SBOREN for subsequent restore
   RCONbits.SBOREN = 1;                 // Enable brown-out reset
   GIEHCOPY = INTCONbits.GIEH;          // Copy GIEH for subsequent restore
   INTCONbits.GIEH = 0;                 // Disable all interrupts
   EECON1bits.WREN = 1;                 // Enable write operation
   EECON2 = 0x55;                       // Write first key
   EECON2 = 0xAA;                       // Write second key
   EECON1bits.WR = 1;                   // Set WR bit to initiate write
   INTCONbits.GIEH = GIEHCOPY;          // Restore global interrupt state
   RCONbits.SBOREN = SBORENCOPY;        // Restore brown-out reset state

   PORTCbits.RC2 ^= 1;                  // Toggle pin to flag where write occurs
   Delay(100);                          // Add a delay to show sleep after 1 ms
}
```

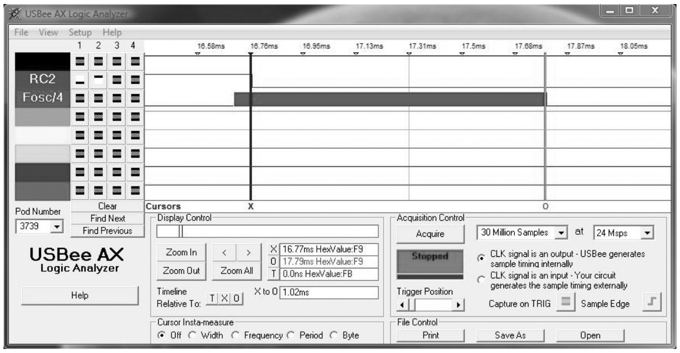
(a) EEtest.c



Pod Number 3739
Display Control
X -121.99ms HexValue:F9
O -121.99ms HexValue:F9
T 0.0ns HexValue:FB
Timeline Relative To: T X O
X to O 0.0ns
Acquisition Control
30 Million Samples at 24 Msps
[✓ CLK signal is an output - USBee generates sample timing internally]
CLK signal is an input - Your circuit generates the sample timing externally
Trigger Position
Capture on TRIG
Sample Edge
Cursor Insta-measure
[✓ Off] Width Frequency Period Byte
File Control

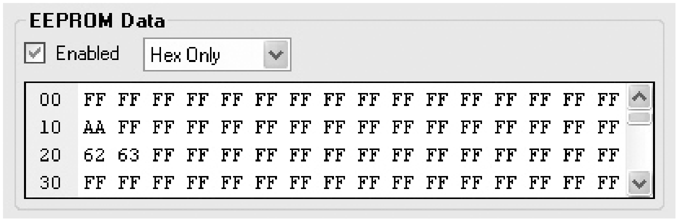
(b) Three writes + 1 ms markers

FIGURE 14-4 *(continued)*



```
16.58ms 16.76ms 16.95ms 17.13ms 17.31ms 17.5ms 17.68ms 17.87ms 18.05ms
Pod Number 3739
X  16.77ms HexValue:F9
O  17.79ms HexValue:F9
T  0.0ns HexValue:FB
X to O  1.02ms
30 Million Samples  at  24 Msps
(•) CLK signal is an output - USBee generates sample timing internally
( ) CLK signal is an input - Your circuit generates the sample timing externally
Cursor Insta-measure: (•) Off ( ) Width ( ) Frequency ( ) Period ( ) Byte
```

(c) Second write and its 1 ms marker



```
EEPROM Data
[✓] Enabled   Hex Only

00  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
10  AA FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
20  62 63 FF FF FF FF FF FF FF FF FF FF FF FF FF FF
30  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

(d) PICkit2 verification of the three writes

FIGURE 14-4 *(continued)*

the duration of the write sequence, a powering-down of the chip that is already in progress and has reached the point where V_{DD} is below 2 V will reset the chip. The out-of-range value of V_{DD} may have produced a transmutation of a bit in the program counter that was following a path to the call and execution of the **EEwrite** function. However, when the CPU executed the brown-out reset enable instruction, the CPU's execution of further instructions was stopped dead in its tracks.

Another type of concern arises when an application is subjected to a perturbation (e.g., the RF interference resulting from a lightning strike) that reaches all the way into the chip and toggles a bit of the program counter. This produces the same threat as that of the last section without any help from the solution offered there.

As an example of a different approach, consider how QwikBug writes downloaded user code into the flash program memory using the **WriteInitiateSequence** function of Figure 14-5. The first line of the function tests a bit in an undocumented **DEBUG** register. When the chip is in the Background Debug Mode used by Microchip's programming/debugging tools such as the PICkit 2 and also used by QwikBug, the **INBUG** bit is set. A perturbation of the program counter that produces an inadvertent write sequence can only be to the program memory address produced by the

```
    EECON2 = 0x55;             // Write first key
```

line. None other of the 8,192 addresses making up program memory will lead into the undesirable sequence. Even at that, the address pointer at that moment is pointing

```
WriteInitiateSequence()
{
   if (DEBUGbits.INBUG)
   {
      EECON2 = 0x55;              // Write first key
      EECON2 = 0xAA;              // Write second key
      EECON1bits.WR = 1;          // Set WR bit to initiate write
   }
}
```

FIGURE 14-5 QwikBug's WriteInitiateSequence function.

into the user area of program memory, not into the area that holds QwikBug. In this way, the QwikBug code is protected against corruption by the inadvertent execution of its own code by a user program.

A similar tack can be taken when an EEPROM write is initiated by an external event, the press of a pushbutton, for example. If the press is used to set a **PRESS** flag while the release is used to set a **RELEASE** flag, then a test of **PRESS** can be used to load up the registers used by the **EEwrite** function, and the following sequence used to initiate the write:

```
if (PRESS && RELEASE)
{
   EEwrite();
   PRESS = 0;
   RELEASE = 0;
}
```

14.6 EEPROM LIFE

The finite life of EEPROM bits is a factor of little consequence for most applications. Each write to an EEPROM byte is translated by the EEPROM module into an erase operation followed by the actual write operation. Each of the 256 addresses of EEPROM memory in the PIC18LF4321 is specified to be able to sustain a minimum of 100,000 erase per write cycles before an error may occur. This can be extended to 1,000,000 erase per write cycles for any addresses that have not been written to in the last 100,000 writes to the module by refreshing them. Knowing the rate at which EEPROM writes are being carried out by a heavy duty EEPROM application, and knowing how long it has been since the last refresh cycle, provides the information needed to copy the content of each of the 256 addresses back to itself after every 100,000 writes.

Another refresh scheme for an application that uses only *n* bytes of EEPROM is to use one of those bytes as a counter and increment it each time any of the other bytes is written to. When the counter reaches 100,000/2 = 50,000 (because each write is accompanied by a write to the counter), the next *n* bytes are used.

For example, if only 50 bytes are used, the EEPROM memory can be divided into 256/50 blocks:

```
  0-49
 50-99
100-149
150-199
200-249
```

with six addresses left over. One of these last six (e.g., address 255) can serve as a block pointer to indicate which of the five blocks is the active block. Within each block, one address (e.g., 0 or 50 or 100 or 150 or 200) can serve as the counter of writes to the block. When the counter reaches 100,000, the 50 bytes are copied to the next block, the counter in the new block zeroed and the block pointer incremented. In this case, this scheme extends the life of the EEPROM module to

$$(100{,}000/2) \times 10 \times 5 = 2{,}500{,}000$$

erase per write cycles.

This section has been a detailed look at handling the finite EEPROM life issue that never even arises for most applications. For example, the EEPROM within the microcontroller within a camera may be written to a few times during its exposure calibration at the completion of its assembly and then never written to again. In this case, the EEPROM is invaluable to the application, but its erase per write endurance is not an issue. If the MCU in the camera were a PIC18LF4321, its more relevant spec is its minimum data retention of 40 years.

PROBLEMS

14-1 SendMicroSec function

a) Modify the code developed for the CalibrateX.c program of Problem 13-3 so that it stores the content of **OSCTUNE** into address 0xFD of EEPROM and the content of **CALIB** into 0xFE and 0xFF after each calibration carried out in response to a pushbutton press.

b) Create a new **InitOSCTUNE** function. This function first looks at the *char* value located at 0xFF. This should hold the upper byte of **CALIB** if the CalibrateY.c program has been run. If **CALIB** = 7,812 = 0x1E84, then this upper byte will be 0x1E. Test to see if this upper byte stored in EEPROM address 0xFF falls within the range 0x1C to 0x21. If not, send the word “Recalibrate” to the Qwik&Low Console. If so, copy the EEPROM value from 0xFD into **OSCTUNE**. If this **InitOSCTUNE** function is called toward the end of the **Initial** function of a user program, that user program will be armed to produce calibrated microsecond measurements in addition to cycle count measurements.

c) Create a **SendMicroSec** function analogous to the **Send** function of Figure 13-8c. Use Equation 13-1 from Section 13.3 to convert **CYCLES** to **BIGNUM**.

d) Use the **Delay** macro to generate a 2,000 cycle pulse with **Delay**(200) that is immediately preceded by the call of the **Start** function and then the setting of **RB0**, and immediately followed by the clearing of **RB0** and then the call of the **Stop** function and the **SendMicroSec** function. Compare the pulse width on **RB0** with the result that is displayed on the PC. Now comment out just the **Delay** macro and do this again. The difference between these two measurements is the result of exactly 2,000 cycles of execution. How close is the resulting number of microseconds for the pulse width to the number calculated from the PC values?

e) If a "universal counter" is available to you, use its extraordinarily accurate time interval and frequency measurement capability to check both the pulse width of **RB0** and also the accuracy of $F_{OSC}/4$, the CPU clock frequency, on test point TP6. Use this information to comment on the results found in Part (d).

Chapter 15

1-WIRE INTERFACE (SSN.c)

15.1 OVERVIEW

Dallas Semiconductor, now a part of Maxim Integrated Circuits, has a family of parts that employ a 1-wire® interface. These parts fit well in the low-current environment of the Qwik&Low board, as exemplified by the Silicon Serial Number part of Figure 3-2. As shown there, one MCU pin is used to power the part. Another pin is used for bidirectional communication between the MCU and the part, with the MCU controlling the timing of each bit, each byte, and each multiple-byte message transfer.

The chapter begins with the 1-wire protocol and its implementation by Alex Singh in the SSN.c template file to read back the unique 8-byte serial number from a DS2401 Silicon Serial Number part. Then the broader issue of dealing with other 1-wire devices is discussed.

15.2 INTERFACE CIRCUITRY

The interface circuit of both a 1-wire part and the MCU is shown in Figure 15-1. The 1-wire part has a true *open-drain* output that can pull the I/O line low by overriding the pull-up resistor. If its internal line labeled "Output" in Figure 15-1 is low, the MOSFET

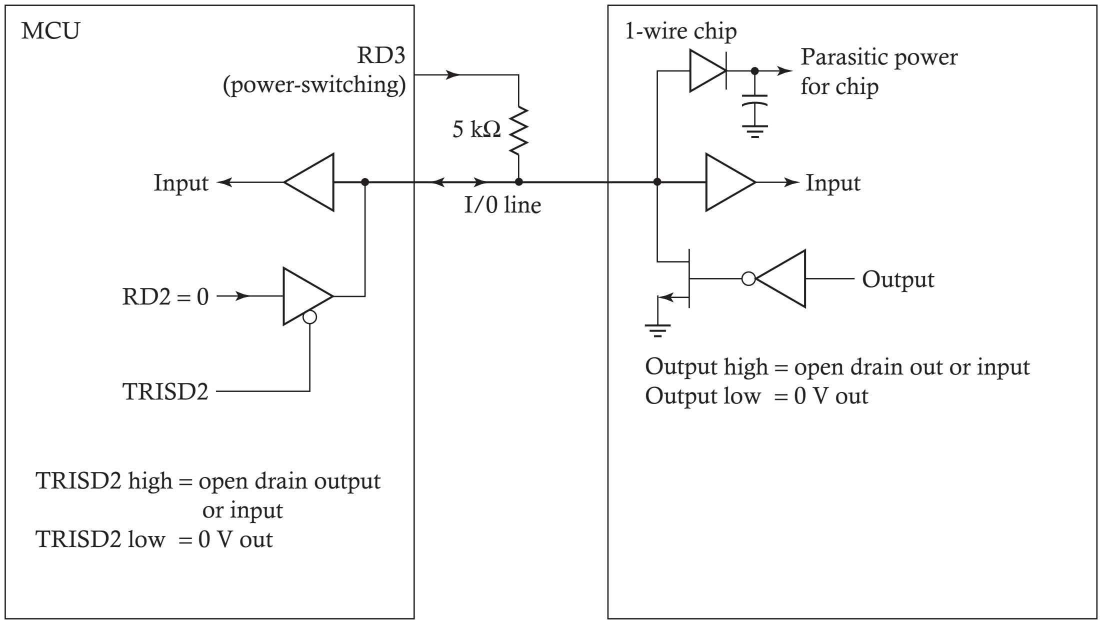


FIGURE 15-1 1-wire interface circuit

will pull the I/O line low. If "Output" is high, the MOSFET is turned off and the I/O line will be pulled high by the pull-up resistor unless some other device (the MCU in this circuit) pulls the line low. Thus, the 1-wire chip:

- Drives its "Output" low to force a "0" on the I/O line.
- Drives its "Output" high to generate an (open-drain) output of "1".
- Drives its "Output" high whenever it is expecting input data so that the MCU can control the I/O line.
- Draws "parasitic" power from the I/O line. The 1-wire protocol supports this by including at least a 180-µs "high" interval on the I/O line before ones and zeros are transferred. This interval is sufficient to charge the internal capacitor that keeps the internal logic supplied during a message transfer.

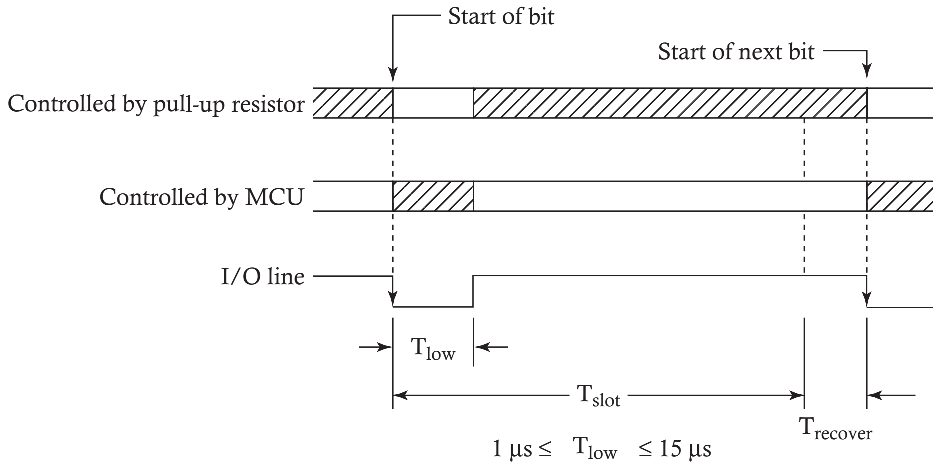
For its part, the MCU must simulate having an open-drain output with what is actually a *totem-pole* output; that is, an output driver with one MOSFET to pull the I/O line down to ground and another MOSFET to pull the I/O line up to V_{DD}. It is this latter MOSFET that must never be turned on. This is achieved by treating the **TRISD2** pin as the open-drain output. Thus, the MCU:

- Drives **TRISD2** low (with **RD2** low) to force a "0" on the I/O line.
- Drives **TRISD2** high to generate an "open-drain" output of "1".
- Drives **TRISD2** high whenever it is expecting input data so that the 1-wire chip can control the I/O line.

15.3 WRITING ONES AND ZEROS

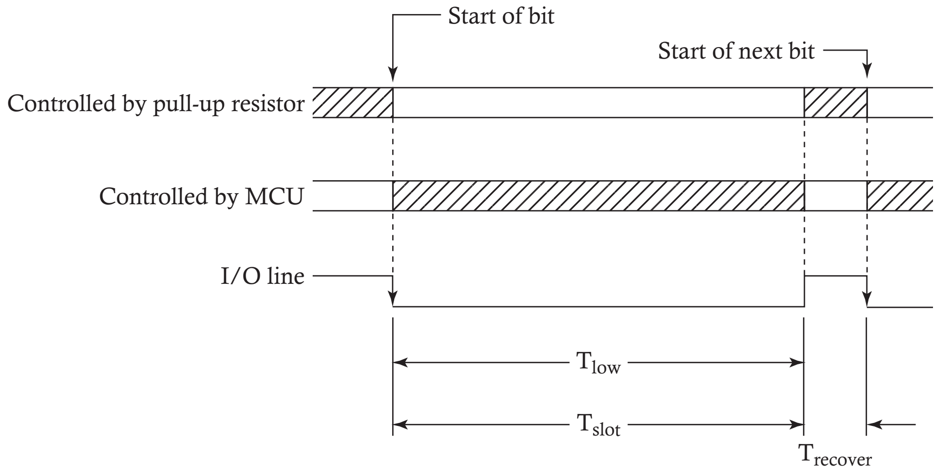
The MCU controls the timing for each bit that it sends or receives. Consequently, input and output are treated somewhat differently. Figure 15-2a illustrates the timing to send a one by pulling the line low with

```
PORTDbits.RD2 = 0;

TRISDbits.TRISD2 = 0;
```



$1\ \mu s \le T_{low} \le 15\ \mu s$
$60\ \mu s \le T_{slot} < 120\ \mu s$
$1\ \mu s \le T_{recover} < \infty$

(a) Writing a one.



$60\ \mu s \le T_{low} \le T_{slot} < 120\ \mu s$
$1\ \mu s \le T_{recover} < \infty$

(b) Writing a zero.

FIGURE 15-2 Writing ones and zeros

and releasing it immediately with

```
TRISDbits.TRISD2 = 1;
```

and then waiting out the T_{slot} time with

```
Delay(6);
```

For clarity in his SSN.c source file, Alex Singh represents the first two of these steps with macros:

```
OpenDrainLow;
```

and

```
OpenDrainHigh;
```

The code to send a one becomes

```
OpenDrainLow;
OpenDrainHigh:         // Send ONE
Delay(6);
```

and the code to send the zero shown in Figure 15-2b becomes

```
OpenDrainLow;
Delay(6);             // Send ZERO
OpenDrainHigh;
```

15.4 MESSAGE PROTOCOL

The protocol for a complete message between the MCU and the DS2401 Silicon Serial Number part is illustrated in Figure 15-3. If it begins with the I/O line powered down for longer than half a millisecond, the DS2401 will begin operation in its reset state, as required. More generally, for multiple 1-wire devices sharing a single 1-wire I/O bus, pulling the I/O line low for the half-millisecond *Reset pulse* shown in Figure 15-3 will force their interface circuitry to the reset state.

When the I/O line is released by the MCU, the DS2401 and any other 1-wire devices sharing the I/O line will pull the line low to signal their presence on the bus. The MPU can look for the line to be low 60 μs after it was released at the end of the reset pulse. This *Presence pulse* may last for up to 240 μs, but the MCU can continue with the next step of the protocol after a high *recovery* time of at least 180 μs.

All 1-wire devices expect the Presence pulse time slot to be followed by the reception from the MCU of a 1-byte command, sent LSb (least-significant-bit) first. The DS2401 responds to the *Read ROM* command, coded as 0x33. It is sent out by the MCU with

```
BYTETOSEND = 0x33;
SendByte();
```

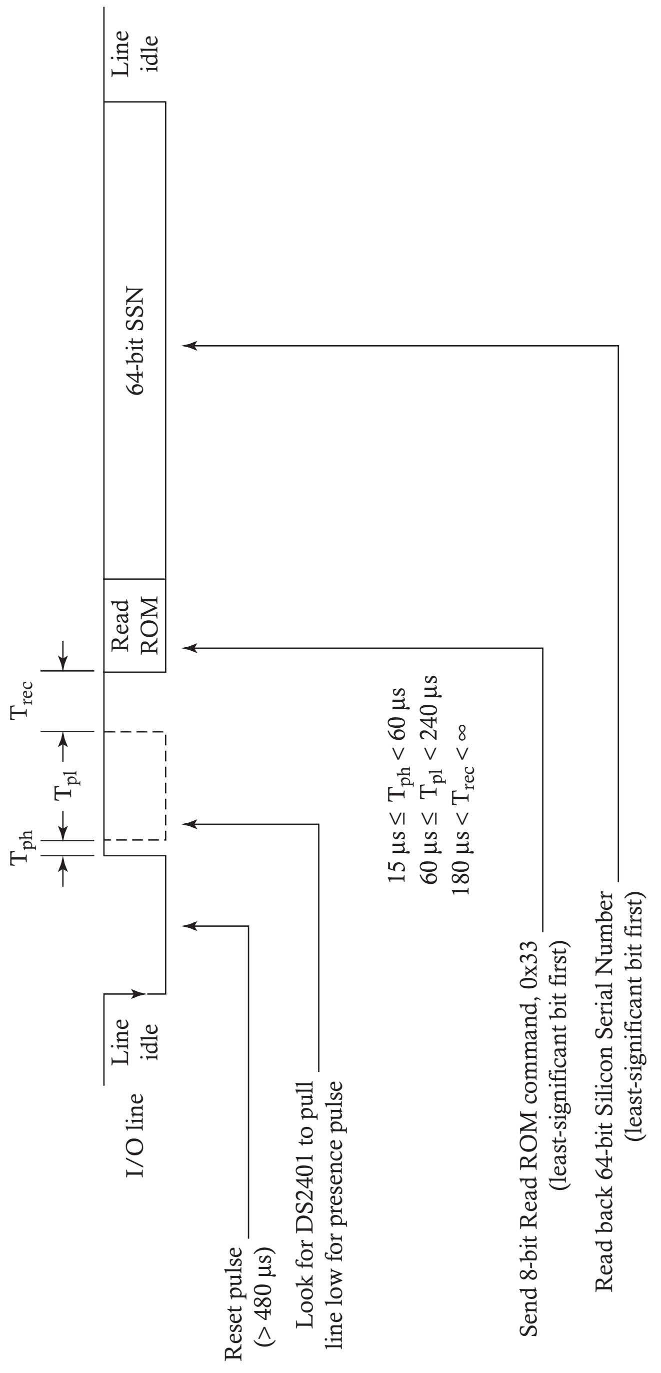

I/O line
Line idle
Reset pulse (> 480 µs)
Look for DS2401 to pull line low for presence pulse
T_{ph}
T_{pl}
T_{rec}
Read ROM
64-bit SSN
Line idle
$15\ \mu s \le T_{ph} < 60\ \mu s$
$60\ \mu s \le T_{pl} < 240\ \mu s$
$180\ \mu s < T_{rec} < \infty$
Send 8-bit Read ROM command, 0x33 (least-significant bit first)
Read back 64-bit Silicon Serial Number (least-significant bit first)

FIGURE 15-3 SSN protocol

in the **DS2401** function of the SSN.c code of Figure 15-4. Each of the 8 bits is sent as discussed in the last section. In response, the DS2401 expects to be queried for its 64-bit SSN, received a bit at a time (LSb first) as discussed in the next section.

```
/******* SSN.c *****************
 *
 * Read the DS2401 serial number and then sleep with RD3 high and RD2 = input
 * Use Fosc = 4 MHz for Fcpu = Fosc/4 = 1 MHz.
 * Developed by Alex Singh.
 *
 ******* Program hierarchy *****
 *
 * main
 *    Initial
 *    InitTX
 *    DS2401
 *       PresenceTest
 *       SendByte
 *       ReadByte
 *       SaveByte
 *       SendSSNSTRING
 *
 *******************************
 */

#include <p18f4321.h>           // Define PIC18LF4321 registers and bits

/*******************************
 * Configuration selections
 *******************************
 */
#pragma config OSC = INTIO1     // Use internal osc, RA6=Fosc/4, RA7=I/O
#pragma config PWRT = ON        // Enable power-up delay
#pragma config LVP = OFF        // Disable low-voltage programming
#pragma config WDT = OFF        // Disable watchdog timer initially
#pragma config MCLRE = ON       // Enable master clear pin
#pragma config PBADEN = DIG     // PORTB<4:0> = digital
#pragma config CCP2MX = RB3     // Connect CCP2 internally to RB3 pin
#pragma config BOR = SOFT       // Brown-out reset controlled by software
#pragma config BORV = 3         // Brown-out voltage set for 2.0V, nominal
#pragma config LPT1OSC = OFF    // Deselect low-power Timer1 oscillator

/*******************************
 * Global variables
 *******************************
 */
unsigned char i, j;             // Loop counters
unsigned int DELAY;             // Sixteen-bit counter for obtaining a delay
unsigned char BYTETOSEND;       // Used by SendByte
unsigned char SSNBYTE;          // Used to store byte read from DS2401
```

FIGURE 15-4 SSN.c

```
char SSNSTRING[20] = "xxxxxxxxxxxxxxxx\n\r\0"; // To hold DS2401's serial number

/*******************************
 * Constant array in program memory
 *******************************
 */

const char rom FormHex[] = "0123456789ABCDEF";

/*******************************
 * Function prototypes
 *******************************
 */
void Initial(void);
void InitTX(void);
void DS2401(void);
char PresenceTest(void);
void SendByte(void);
void ReadByte(void);
void SaveByte(void);
void SendSSNSTRING(void);

/*******************************
 * Macros
 *******************************
 */
#define Delay(x) DELAY = x; while(--DELAY){ Nop(); Nop(); }
#define TXascii(in)  TXREG = in; while(!TXSTAbits.TRMT)
#define OpenDrainHigh TRISDbits.TRISD2 = 1
#define OpenDrainLow  PORTDbits.RD2 = 0; TRISDbits.TRISD2 = 0

/////// Main program /////////////////////////////////////////////////////////

/*******************************
 * main
 *******************************
 */
void main()
{
   Initial();                    // Initialize everything
   InitTX();                     // Initialize UART
   DS2401();                     // Read and display SSN on PC
   Sleep();                      // Done
   Nop();
}

/*******************************
 * Initial
 *
 * This function performs all initializations of variables and registers.
 *******************************
 */
```

FIGURE 15-4 *(continued)*

```
void Initial()
{
   OSCCON = 0b01100010;          // Use Fosc = 4 MHz (Fcpu = 1 MHz)
   SSPSTAT = 0b00000000;         // Set up SPI for output to LCD
   SSPCON1 = 0b00110000;
   ADCON1 = 0b00001011;          // RA0,RA1,RA2,RA3 pins analog; others digital
   TRISA = 0b00001111;           // Set I/O for PORTA
   TRISB = 0b01000100;           // Set I/O for PORTB
   TRISC = 0b10000000;           // Set I/O for PORTC
   TRISD = 0b10000100;           // Set I/O for PORTD
   TRISE = 0b00000010;           // Set I/O for PORTE
   PORTA = 0;                    // Set initial state for all outputs low
   PORTB = 0;
   PORTC = 0;
   PORTD = 0b00100000;           // except RD5 that drives LCD interrupt
   PORTE = 0;
   Delay(50000);                 // Pause for 0.5 second for contact bounce
   RCONbits.SBOREN = 0;          // After this delay, disable brown-out reset
}

/*******************************
 * InitTX
 *
 * This function initializes the UART for its TX output function.
 * It assumes Fosc = 4 MHz.
 *******************************
 */
void InitTX()
{
   RCSTA = 0b10010000;           // Enable UART
   TXSTA = 0b00100000;           // Enable TX
   SPBRG = 12;                   // Set baud rate
   BAUDCON = 0b00111000;         // Invert TX output
}

/*******************************
 * DS2401
 *
 * This function displays on the PC the serial number given
 * by the DS2401. If it is not present, turn on red LED and halt.
 *******************************
 */
void DS2401()
{
   PORTDbits.RD3 = 1;            // Apply power to SSN part
   OpenDrainLow;
   Delay(50);                    // Master Reset with 500 us negative pulse
   OpenDrainHigh;

   if(!PresenceTest())           // Presence Test
   {
      PORTDbits.RD4 = 1;         // Turn on LED
```

FIGURE 15-4 *(continued)*

```
      Sleep();                    //  and be done
      Nop();
   }
   else
   {
      BYTETOSEND = 0x33;          // Send "Read ROM" command to get back
      SendByte();                 //  serial number
      for (j = 16; j > 0; j-=2) // Read and save 8 bytes
      {
         ReadByte();
         SaveByte();
      }
      SendSSNSTRING();
   }
   OpenDrainHigh;                 // Leave I/O pin as an input
   PORTDbits.RD3 = 1;             // Leave SSN part empowered
}

/*******************************
 * PresenceTest
 *
 * This function returns a 1 if DS2401 is present and a 0 otherwise.
 *******************************
 */
char PresenceTest()
{
   Delay(10);                     // After 100 us
   if (!PORTDbits.RD2)            //  check for Presence
   {
      while (!PORTDbits.RD2);    // Wait for line to be released
      return 1;
   }
   return 0;
}

/*******************************
 * SendByte
 *
 * This function sends BYTETOSEND to DS2401. Tslot = 85 us for ONE or for ZERO.
 *******************************
 */
void SendByte()
{
   for (i = 0; i <= 7; i++)       // Send 8 bits of command
   {
      if (BYTETOSEND & 1)        // Test bit 0
      {
         OpenDrainLow;
         OpenDrainHigh;           // Send ONE
         Delay(6);
      }
```

FIGURE 15-4 *(continued)*

```
      else
      {
         OpenDrainLow;
         Delay(6);               // Send ZERO
         OpenDrainHigh;
      }
      BYTETOSEND >>= 1;          // Move on to next bit
   }
}

/******************************
 * ReadByte
 *
 * This function reads in the 64-bit serial number a byte at a time.
 * Tslot = 70 us for ONE or for ZERO.
 * DS2401 holds line low for zero for 35 us.
 ******************************
 */
void ReadByte()
{
   SSNBYTE = 0;                  // Initialize SSNBYTE
   for (i = 0; i <= 7; i++)      // Read 8 bits from DS2401
   {
      SSNBYTE >>= 1;             // Move on to next bit
      OpenDrainLow;
      OpenDrainHigh;
      Delay(1);                  // Sample 13 us after 'clocking'
      if (PORTDbits.RD2)
      {
         SSNBYTE += 0b10000000; // Copy bit into SSNBYTE
      }
      Delay(3);
   }
}

/******************************
 * SaveByte
 *
 * This function converts the binary value in SSNBYTE into two
 * ASCII-coded digits in SSNSTRING.
 ******************************
 */
void SaveByte()
{
       SSNSTRING[j - 2] = FormHex[SSNBYTE >> 4];   // Save upper nibble
       SSNSTRING[j - 1] = FormHex[SSNBYTE & 0x0F]; // Save lower nibble
}
```

FIGURE 15-4 *(continued)*

```
/******************************
 * SendSSNSTRING
 *
 * This function sends SSNSTRING to the PC.
 ******************************
 */

void SendSSNSTRING()
{
   for (i=0; SSNSTRING[i]; ++i)
   {
      TXascii(SSNSTRING[i]);  // Send all bytes until null terminator
   }
}
```

FIGURE 15-4 *(continued)*

15.5 READING ONES AND ZEROS

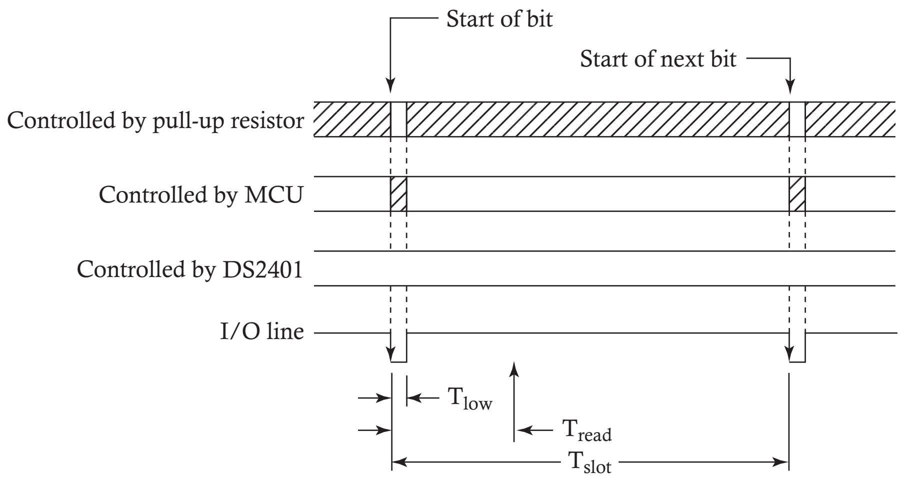
Figure 15-5 illustrates the reading of ones and zeros from the DS2401. The MCU clocks each bit time by pulling the I/O line low and then immediately releasing it. The DS2401 responds to the falling edge and leaves the line released for a "1" or pulls the line low for a "0". The MCU samples the line after about 15 µs. It waits for something over 60 µs (the T_{slot} time of Figure 15-5) before initiating the falling edge to clock the next bit response.

The **DS2401** function listed in Figure 15-4 calls its **ReadByte** function eight times to acquire the 64-bit SSN. Each call of **ReadByte** reads the I/O line eight times, shifting the bits into **SSNBYTE**. Each **ReadByte** is followed by a **SaveByte** that breaks **SSNBYTE** into two ASCII-coded hexadecimal characters and inserts these into the long **SSNSTRING** string. After **SSNSTRING** has been loaded with the 16 hex characters, the entire string is sent to the PC for display. Then the I/O pin is left in the **Open-DrainHigh** (i.e., input) state while **RD3** is powered down before the MCU is put to sleep. The board at that point draws just 0.1 µA from the coin cell. Even if the DS2401 chip is left empowered, it adds just 0.2 µA to this shutdown current.

15.6 GENERALIZING FROM THE SSN.c TEMPLATE

- The intent of the code of Figure 15-4 is to illustrate the general principles of 1-wire messaging including:
- Reset pulse generation
- Presence pulse detection
- Sending a 1-byte command
- Sending or receiving bytes to or from a 1-wire device

The SSN.c file does nothing more than to initialize the chip, read the SSN, and then go to sleep.

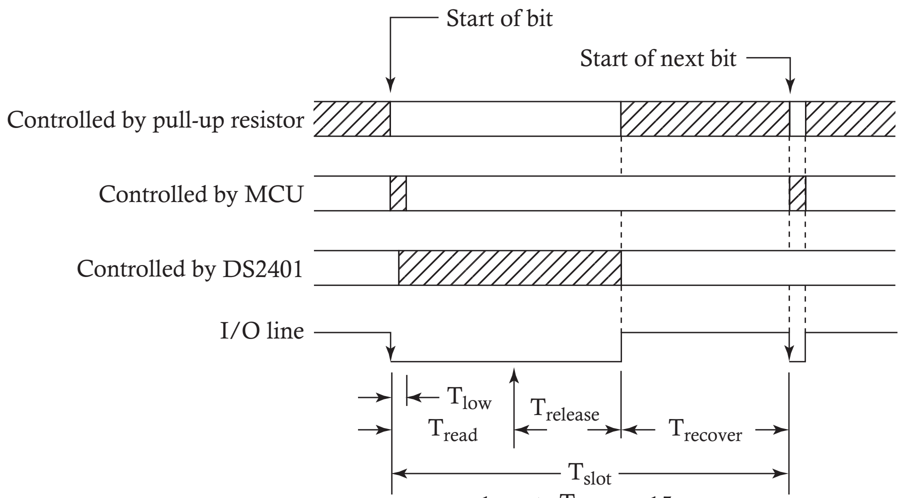


$1\ \mu s \le T_{low} < 15\ \mu s$

$T_{read} = 15\ \mu s$

$60\ \mu s \le T_{slot} < 120\ \mu s$

(a) Reading a one



$1\ \mu s \le T_{low} < 15\ \mu s$

$T_{read} = 15\ \mu s$

$0\ \mu s < T_{release} \le 45\ \mu s$

$60\ \mu s \le T_{slot} < 120\ \mu s$

$1\ \mu s \le T_{recover} < \infty$

(b) Reading a zero

FIGURE 15-5 Reading ones and zeros

More generally, 1-wire device interactions can be employed during the on-going operation of a larger program, to take advantage of the versatility of 1-wire devices. During such interactions, it is important to disable interrupts before each bit transfer is initiated. Nothing is lost if an interrupt service routine intervenes after the MCU releases the I/O line and has, in addition, completed the reading or the writing of a bit. While the I/O line is high, an I/O message string can be paused indefinitely. However, if an interrupt occurs while the MCU is holding the line low and the resulting pause causes a 1-wire device to be held low for more than 120 μs, the serial I/O messaging protocol may be reset.

15.7 MULTIPLE 1-WIRE DEVICES ON A SINGLE BUS

The Dallas/Maxim repertoire of 1-wire integrated circuits includes real-time clocks, temperature sensors, analog-to-digital converters, battery usage (current × time) monitors, and nonvolatile memory with security key access. As an example of the application of multiple devices, several DS1825 digital thermometer chips may be used to monitor potential hot spots of a PC board. Each chip, housed in a tiny 8-pin surface-mount package, has four of its pins hard-wired as a 4-bit address. Thus, not only is temperature measured, but also the location of a hot spot can be identified.

Dallas/Maxim's significantly pricier iButton® parts are housed in a rugged coin-cell-like package and can employ extended self-powered capabilities afforded by an internal lithium battery. For example, one of their Thermochron® parts can be included in a shipment of perishable food to log temperature samples, to verify whether the shipment was handled as intended.

In this section, the emphasis will be on identifying and selecting a specific 1-wire device from among several sharing a single bus. The two 1-wire commands that are used to identify and select any one device are

Search ROM	(0xF0)
Match ROM	(0x55)

The *ROM* that is being referred to here is the unique 64-bit serial number included in every 1-wire device, not just the DS2401 Silicon Serial Number part discussed earlier.

The *Search ROM* command is used iteratively to determine the ROM content of each device on the 1-wire bus. Once found, each of these can be stored in the MCU's EEPROM, and used thereafter to select a device. This Search ROM command will be discussed shortly.

Following the Reset pulse/Presence pulse sequence, a specific device is selected by sending the *Match ROM* command followed by the 64-bit ROM contents in the 64 time slots (LSb first) that follow the eight time slots of the Match ROM command. The selected device will then respond to all subsequent commands while the unselected devices will stay idle until they receive another Reset pulse.

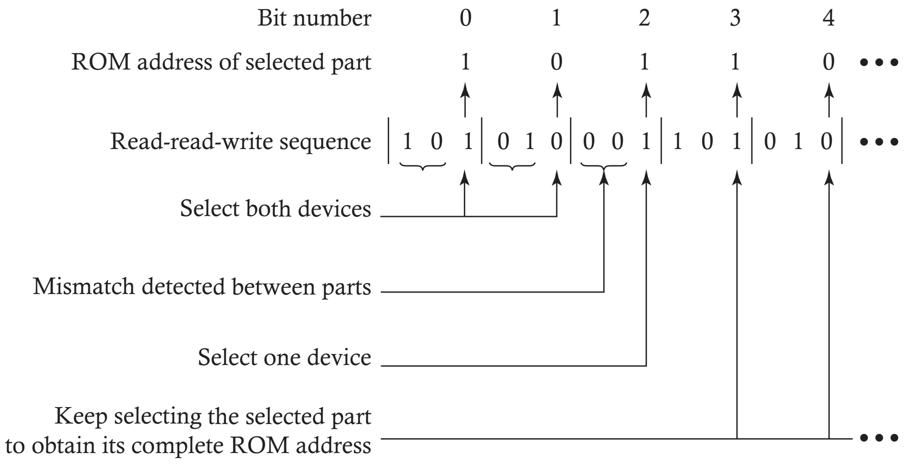
The *Search ROM* command is followed by two 1-bit reads and a 1-bit write. In response to the first read, all devices send the state of the LSb of their ROM address. During the second read, all devices send the complement of the LSb of their ROM address. The interpretation of the response to these two reads is listed in Figure 15-6a. The next

First read	Second read	Interpretation
1	0	All devices have a 1 in this ROM address position
0	1	All devices have a 0 in this ROM address position
0	0	Devices differ in this ROM address position

(a) Interpretation of a pair of reads.

Write	Interpretation
1	Only devices with a 1 in this ROM address position remain enabled
0	Only devices with a 0 in this ROM address position remain enabled

(b) Meaning of the write during the next Time slot.


Bit number 0 1 2 3 4
ROM address of selected part 1 0 1 1 0 •••
Read-read-write sequence | 1 0 1 | 0 1 0 | 0 0 1 | 1 0 1 | 0 1 0 | •••
Select both devices
Mismatch detected between parts
Select one device
Keep selecting the selected part to obtain its complete ROM address
•••

(c) Example, obtaining the ROM address of one of a pair of 1-wire parts.

FIGURE 15-6 Search ROM procedure

time slot (Figure 15-6b) is used as a write, to select all devices that match the bit written to them. Figure 15-6c illustrates the Search ROM process of successive read-read-write bits to sort out the ROM address of one of the devices. If a “1” is written during the third of the three time slots each time a bit mismatch is detected (by a read-read of 0-0), the result will be to select the part with the highest ROM address.

If only two 1-wire parts are connected to the 1-wire bus, the first bit mismatch found will be the only bit mismatch needed to obtain the second part’s ROM address. Instead of writing a one back for this bit, a zero is written, deselecting the chip whose

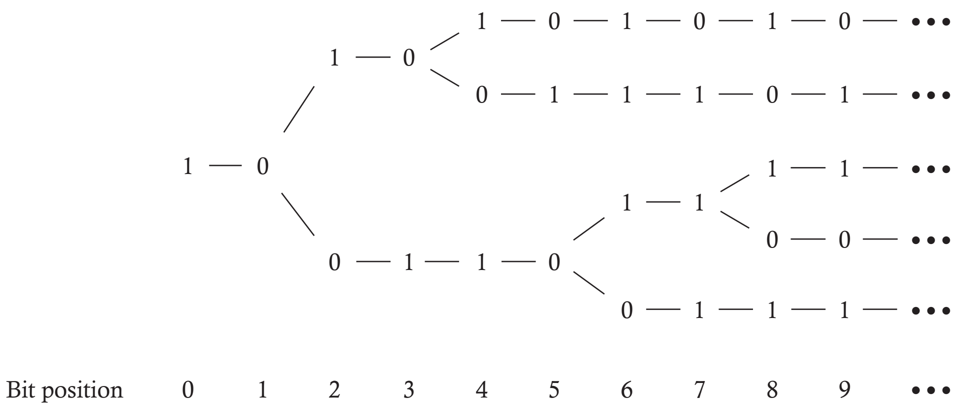


FIGURE 15-7 Tree of ROM addresses for five 1-wire parts

ROM address has already been found in order to continue to determine the ROM address for the second part.

For three 1-wire parts connected to the 1-wire bus, the path to select one of the first two chips will include a second mismatch. If the part having a one in this second mismatch position is taken, then subsequently the path having a zero in this second mismatch position must be followed.

For more than three 1-wire parts connected to the 1-wire bus, the determination of each ROM address amounts to tracing out each branch of a tree having 64 bits in it, as illustrated in Figure 15-7. Dallas/Maxim addresses this determination systematically in an application note that can be found by Googling "Book of iButton Standards" and then finding on PDF page 58, Section C.3 Search ROM Command:

"The general principle of this search process is to deselect one device after another at every conflicting bit position. At the end of each ROM Search process, the master has learned another ROM's contents. The next pass is the same as the previous pass up to the point of the last decision. At this point, the master goes in the opposite direction and continues. If another conflict is found, again zero is written, and so on. After both paths at the highest conflicting bit position are followed to the end, the master goes the same way as before but deciding oppositely at a lower conflicting position, and so on, until all ROM data are identified."

15.8 DS2415 1-WIRE TIME CHIP

The Dallas/Maxim DS2415 1-wire time chip can be added to the Qwik&Low board using the surface-mount pattern in the prototyping area of the board. The additional circuitry is shown in Figure 15-8 with the time chip drawing power directly from the Qwik&Low coin cell, retaining power even when the power switch is turned off. Were the circuit of Figure 15-8 to use the same I/O bus as the SSN part, the Search ROM procedure would have to be executed and the ROM address of each part stored in EEPROM memory. Alternatively, two otherwise unused pins of the MCU (e.g., **RA5**

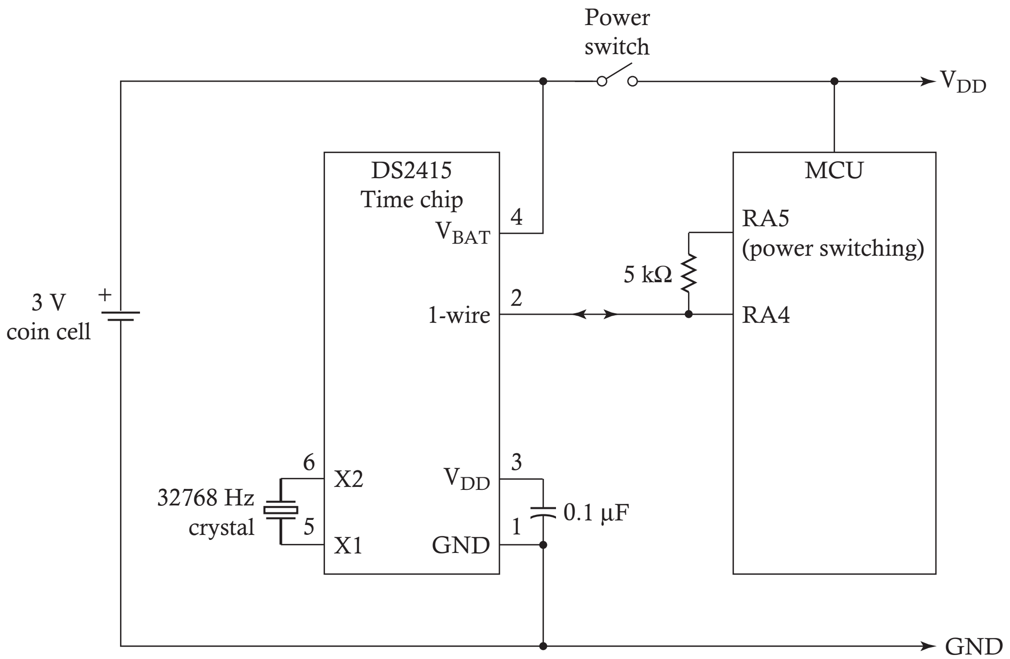


FIGURE 15-8 DS2415 1-wire time chip connections

and **RA4** selected from Figure 3-7) can be used. Although it would seem that the pull-up resistor could be powered directly from the coin cell, unfortunately when the power switch is opened, the pull-up resistor voltage will exceed the MCU's V_{DD} voltage of 0 V and will lead to a constant current drain of 3 V/5 kΩ = 600 μA.

The DS2415 time chip has the following features:

- It draws a maximum of 0.25 μA with its oscillator incrementing an internal settable and readable 32-bit counter every second.
- By being continuously powered, its counter can be employed as a real-time clock, accumulating 136 years of seconds before rolling over.

With just one 1-wire device on the bus of Figure 15-8, the message protocol of Figure 15-9 begins with the Reset pulse/Presence pulse sequence. This is followed by the

Skip ROM (0xCC)

command so that all subsequent I/O interactions will be responded to by that chip.

Once selected by the *Skip ROM* command, the DS2415 responds to just the two commands shown in Figure 15-9:

Write Clock (0x99)
Read Clock (0x66)

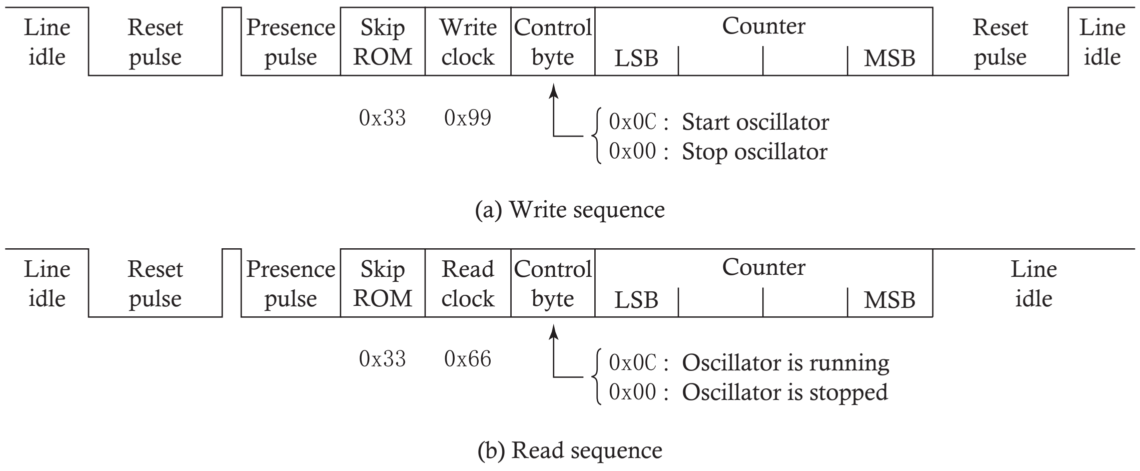


FIGURE 15-9 DS2415 write and read message sequences. (Each byte is sent, or received, least significant bit first)

Thus, to control the oscillator and set the counter, the write sequence of Figure 15-9a produces the following actions:

- The reception of the control byte immediately starts (0x0C) or stops (0x00) the DS2415 oscillator.
- The 4 bytes destined for the counter are received least-significant-byte first and are loaded into a buffer, not the counter itself.
- The final reset pulse transfers the buffer to the counter.

The read sequence of Figure 15-9b is responded to as follows:

- When the *Read Clock* command has been received, the 32-bit counter is copied into the buffer.
- The next 5 bytes received are the control byte followed by the 4-byte counter content.

PROBLEMS

15-1 Interrupts and 1-wire devices As pointed out in Section 15.6, interrupts can occur during a 1-wire transfer without corrupting the transfer if interrupts are disabled *during* each bit transfer and enabled momentarily *between* successive bit transfers. Add repeated reads of the DS2401 every second to the T3.c template file that uses both Timer1 and Timer3 for interrupts.

a) Execute your resulting code. Does the serial number, repeatedly read, ever get corrupted? Explain.

b) Add **Delay** (50) to your **HighPriorityISR** so that you are introducing a 500-μs pause every 4 ms. Does the serial number, repeatedly read, ever get corrupted? Explain.

c) Modify the **SendByte** and **ReadByte** functions in the code for Part (b) with

```
PIE2bits.TMR3IE = 0;       // Disable Timer3 interrupts
```

and

```
PIE2bits.TMR3IE = 1;       // and quickly reenable
```

lines appropriately placed, as suggested in Section 15.6. Now does the serial number, repeatedly read, ever get corrupted? If so, add one or two **Nop** macros between an enable and a following disable, to ensure the momentary enabling of the Timer3 interrupts. Explain what you find including any need for N intervening **Nop** macros.

15-2 Search ROM command Use the *Search ROM* command of Section 15.7 to keep the DS2401 chip enabled for the first 55 bits of its serial number. As each bit is determined, send it back to the PC for display, as an ASCII-coded one or zero. Send a space character (0x20) after every group of eight characters sent to the PC. For the 56th bit, send the complement of the found bit both to the PC and the DS2401. Thereafter, again send each of the remaining 8 bits read via the Search ROM 3-bit-time procedure back to the PC for display. Compare what you get relative to what the SSN.c program produces. Explain the result.

15-3 DS2415 time chip Assume that the DS2415 was cleared to zero and started counting many days ago.

a) Write a function, **Days**, that will take the number read back into the *unsigned long* variable, **TIME**, and convert it to an LCD display of days, with a resolution of 0.01 day and a range up to 9,999.99 days.

```
TIME = (60×60×24×51) + (60×60×24×.16)
     = 4406400 + 13824
     = 4420224 seconds
```

then the LCD display should show

```
0051.16
```

b) Write a second function, **DeltaDays**, that displays, in the same format, the difference

```
TIMEEND – TIMESTART
```

Chapter

16

STARBURST DISPLAY (LCD.c)

16.1 OVERVIEW

The PIC18LF6390 is the second microcontroller on the Qwik&Low board, dedicated to the very specific role of operating the eight-alphanumeric-character LCD display. This chapter will begin with a discussion of how a multiplexed LCD display works, and the waveforms that its LCD controller must generate. Then, Alex Singh's LCD.c code will be examined and his use of data structures to sort out the complexities of the chip will be explained.

16.2 STARBURST DISPLAY CONFIGURATION

The eight-alphanumeric-character *starburst* display is shown in Figure 16-1. It has 36 pins to control $32 \times 4 = 128$ segments. A 14-segment coding represents each of the 8 characters. Also associated with each character are an apostrophe segment and a decimal point segment, as shown in Figure 16-2a. Each of the 16 segments associated with a character position is controlled by a conductive segment on the front of the display and an identically shaped conductive segment on the back of the display, with liquid-crystal material sandwiched in between. To turn on a segment, a low-frequency (37 Hz) complex waveform is imposed between the *frontplane* segment and

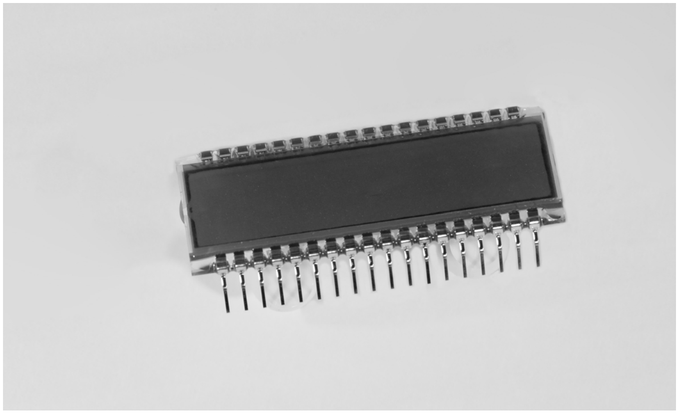


FIGURE 16-1 Starburst display (Varitronix)

the *backplane* segment. Its RMS value must be above a threshold specific to the liquid-crystal medium. Correspondingly, to turn off a segment, the RMS value between the frontplane segment and the backplane segment must be below a specified threshold.

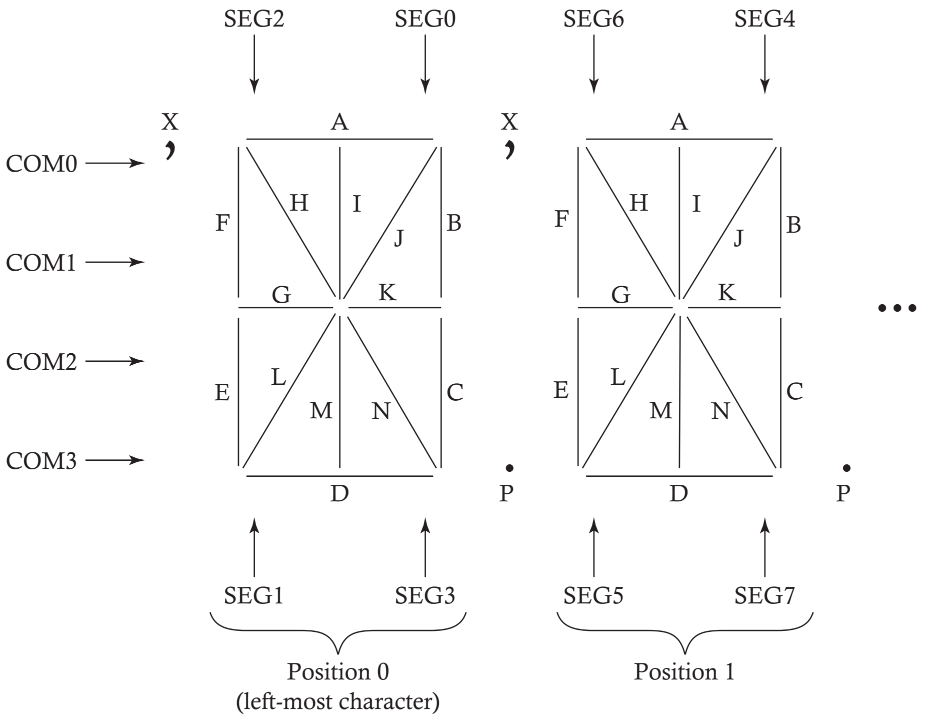
The job of controlling LCD segments is simplified for a nonmultiplexed display such as a three-digit, seven-segment numeric display. Its $7 \times 3 = 21$ digit segments, 2 decimal points, and a single backplane driver are brought out to 24 pins. A low-frequency squarewave between V_{DD} and GND is applied to the entire backplane. An "off" segment has the same waveform applied to its frontplane (for an RMS value of 0 V across the liquid-crystal medium for that segment). An "on" segment has the inverted squarewave applied to its front plane (for an RMS value of V_{DD} volts). Thus, for $V_{DD} = 3$ V, the liquid-crystal medium must only distinguish between 0 V and 3 V. The result is a display with a sharp distinction between on and off segments, regardless of the viewing angle.

In contrast, as will be discussed shortly, the *¼ multiplexing* used by the Qwik&Low LCD is supported by complex waveforms for which each period is made up of stair-step values of 0 V, 1 V, 2 V, and 3 V. The liquid-crystal medium must draw a distinction between an RMS "on" voltage of 1.73 V and an RMS "off" voltage of 1.00 V. The result is a display with a strong sensitivity to viewing angle. The Qwik&Low board's LCD is designed to be viewed from the *6:00 o'clock* position that results when the board sits flat on the workstation in front of a user. The characters fade as the viewing angle approaches the straight on position or is moved to either side. This is a drawback of using a display with only 36 pins and an LCD controller chip designed to drive it while drawing a miniscule current of just a few microamperes.

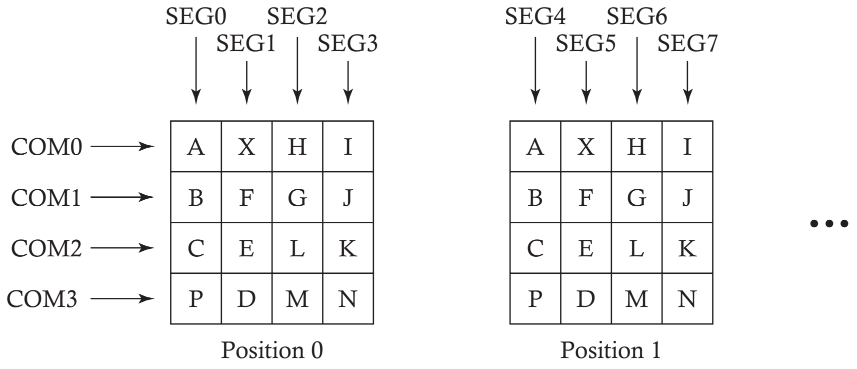
Returning to Figure 16-2b, each frontplane **SEGi** pin is connected to four segments. For example, the **SEG0** pin is connected to segments A, B, C, and P (the decimal point) for the character in POSITION 0 (the left-most character position of the display). **SEG4** is connected to segments A, B, C and P for the character in POSITION 1.

Each backplane **COMj** pin is connected, on the back of the display, to just one of the four segments associated with each frontplane **SEGi** pin. Thus, it is a controlling factor for

$$8 \text{ (characters)} \times 4 \text{ (segment pins per character)} = 32 \text{ segments}$$



(a) Segment coding



(b) Connections for frontplane SEG drivers and backplane COM drivers

FIGURE 16-2 Starburst display configuration

16.3 LCD CONTROLLER CIRCUIT

The LCD controller circuit is shown, in simplified form, in Figure 16-3. Omitted in this figure are the LCD PICkit 2 connections for programming the chip as well as seven 0.1 μF bypass capacitors. The circuit employs a *four-pole-double-throw* (4PDT) push-to-make, push-to-break switch. It connects power and SPI inputs from the MCU

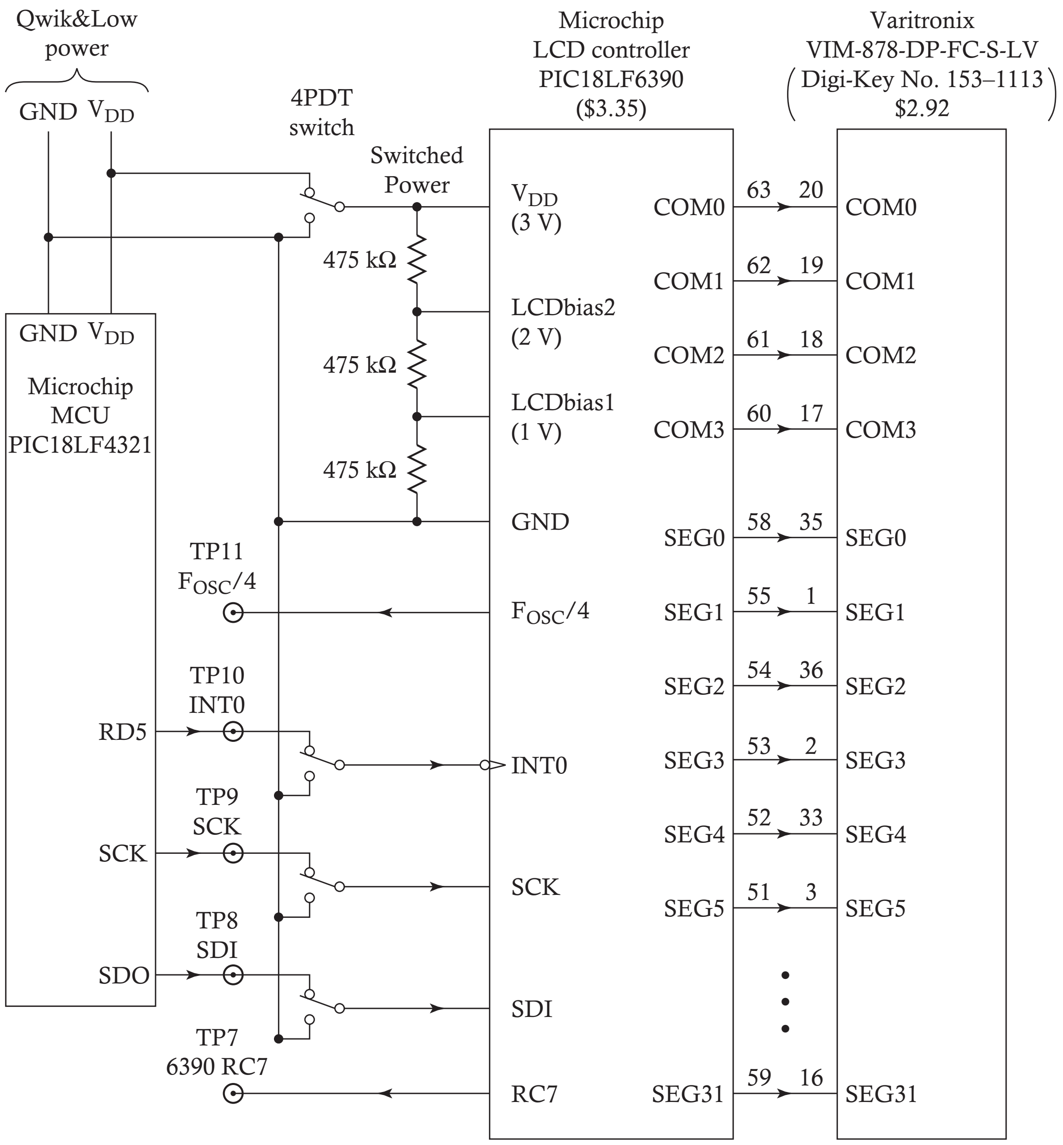


FIGURE 16-3 LCD controller circuit (simplified)

to the LCD controller or, alternatively, it breaks these connections to power down the LCD. The latter condition is used to clarify the current draw by Qwik&Low circuitry independent of the LCD current draw. It is also used to measure the small increase in current draw when the LCD current is added into that drawn for a Qwik&Low application.

The three pins that control the use of the Serial Peripheral Interface (SPI) for updating the display from the MCU are also disconnected from the MCU when the LCD controller is powered down. If this were not done, any MCU operation that left any of these outputs high would produce a leakage current, measured in milliamperes, through the overvoltage protection diodes on the LCD controller inputs to the grounded V_{DD} pin on the LCD controller.

In addition to the three test points for monitoring Serial Peripheral Interface transfers, the LCD controller offers two other test points. **TP11** ($F_{OSC}/4$) is used to monitor how long it takes the LCD controller to respond to new SPI data and to format the characters received for display into the chip's LCD registers before going back to sleep, the normal state of the LCD controller's CPU. Another pin, labeled on the back of the Qwik&Low board as **TP7** (6390 RC7), provides a user-defined flag. It can be used to monitor conditions or measure time intervals introduced if modifications are made to the LCD.c file employed by the LCD controller.

16.4 MULTIPLEXED LCD VOLTAGE WAVEFORMS

The PIC18LF6390 is designed for versatility in its role as an LCD controller. It supports four distinct serial input modes for accepting new display data. It can control nonmultiplexed LCDs as well as 1/2-, 1/3-, and 1/4-multiplexed LCDs. In addition to the 64-pin PIC18LF6390 part, there is an otherwise identical 80-pin PIC18LF8390 that can control up to $4 \times 48 = 192$ LCD segments (half again as many as the 64-pin part).

With its LCD control registers initialized for 1/4-multiplexed operation, the **COM0**, **COM1**, **COM2**, and **COM3** stairstepped waveforms are shown in Figure 16-4. With the refresh rate initialized to 37 Hz, each *frame* is repeated every 1/37 s. Note how each waveform selects a quarter frame by first dropping to 0 V and then rising to 3 V during the second half of the selected quarter frame. During the remaining three-quarter frames, each waveform rises to 2 V during the first half of the quarter frame and drops to 1 V during the second half.

Now consider Figure 16-5a that shows the waveform that the LCD controller will use to drive one of the **SEGj** pins for which all four segments are turned off. Figure 16-5b shows the waveform that the LCD controller uses to drive the backplane **COM0** pin. Referring to Figure 16-2b, this voltage is seen by the "A", "X", "H", and "I" segments of the eight starburst characters. Figure 16-5c shows the voltage waveform responded to by the liquid-crystal medium. Parts d, e, and f of Figure 16-5 form the RMS value of the waveform by first squaring each section (part d) and finding the mean value of the squared waveform (part e). The RMS value of the waveform (part f) is the square root of this mean value, 1.00 V in this case.

Figure 16-6a, b, and c show the waveforms seen when the same "A" or "X" or "H" or "I" segment of one of the eight starburst characters is turned on. The ±3 V difference across the liquid-crystal medium for the first two-eighths of the frame produces a large change in the resulting RMS voltage to 1.73 V, as calculated in parts d, e, and f.

Note that neither voltage waveform across the liquid-crystal medium (Figures 16-5c and 16-6c) has a DC component. This is important to the longevity of the medium. An LCD can tolerate brief moments of DC without apparent ill effect. However, a long-term exposure to DC will turn the medium dark and no longer useful as a display.

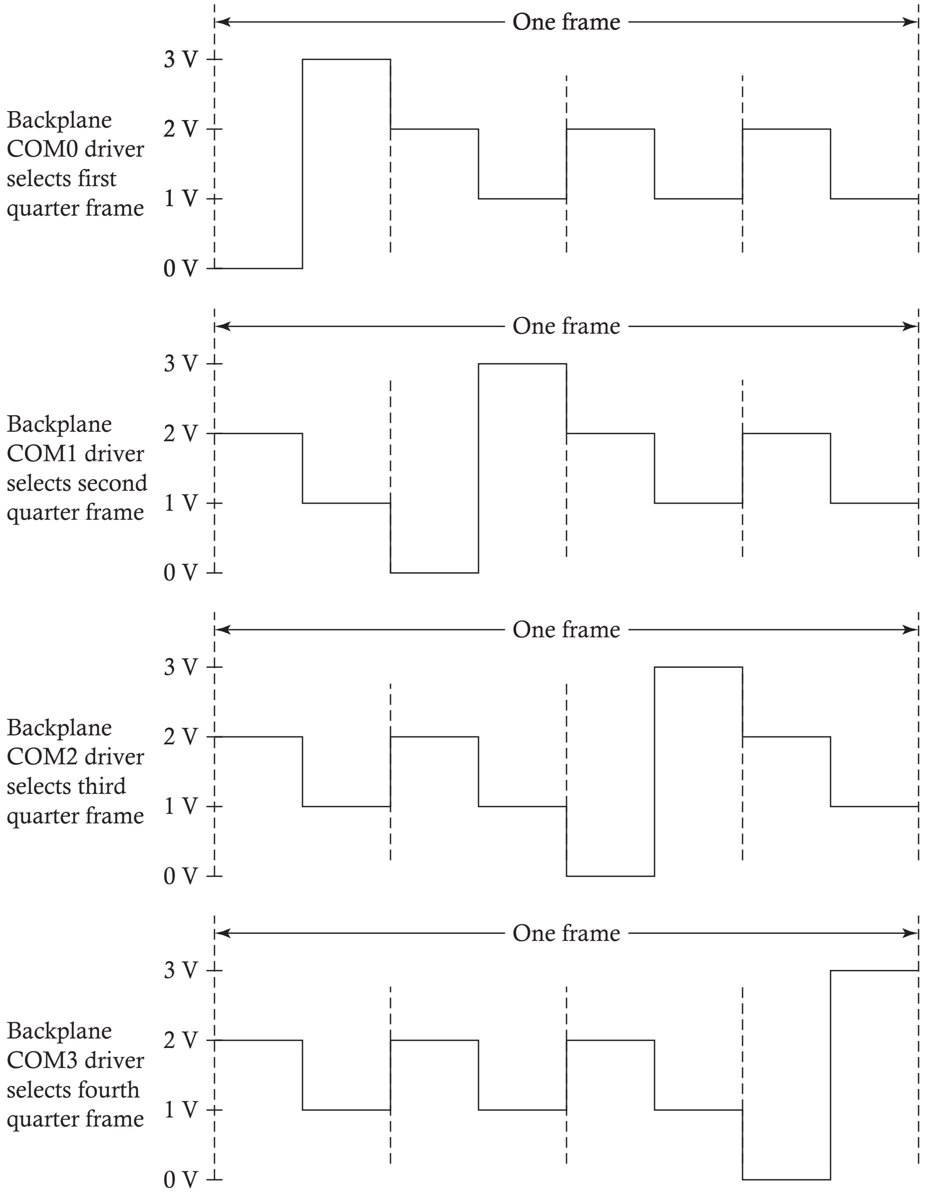
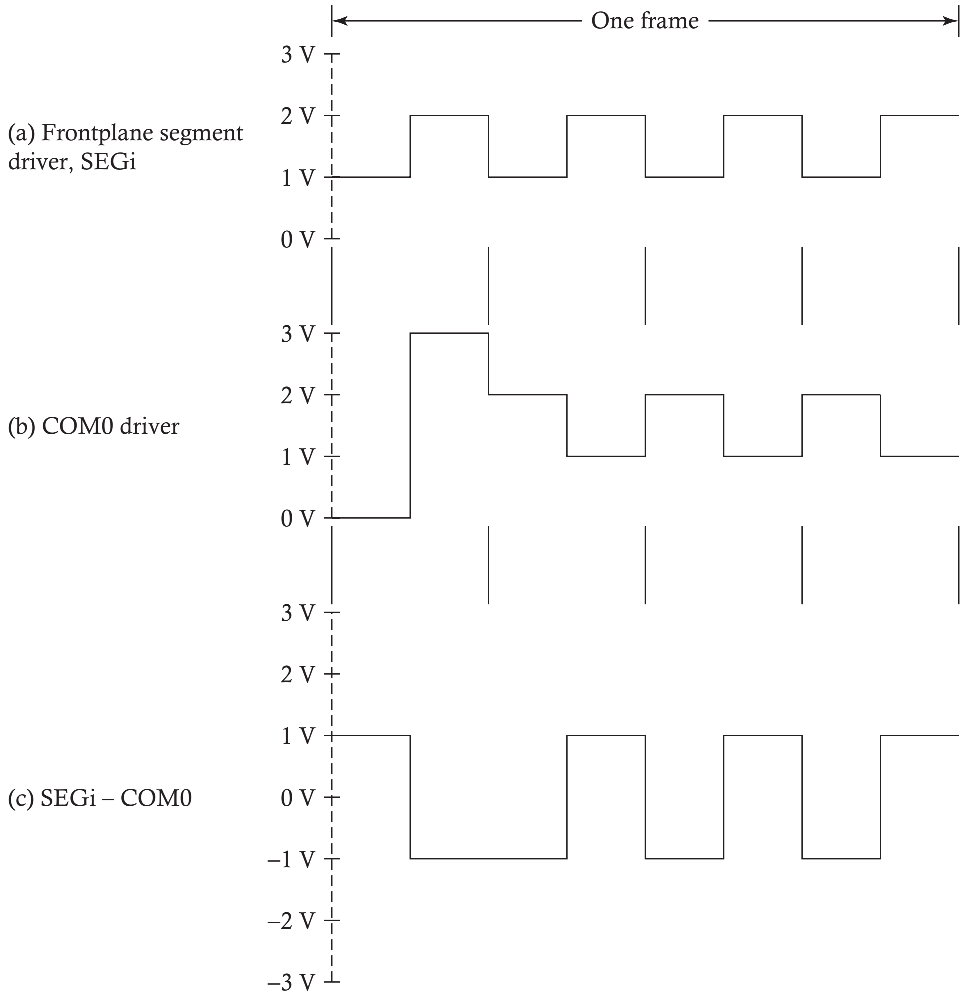


FIGURE 16-4 Backplane waveforms

16.5 LCDDATAi REGISTER USE

The PIC18LF6390 refreshes the display automatically while its CPU sleeps. To do this, each ASCII-coded character sent to it must be converted to its 14-segment representation. The result must be broken into four 4-bit parts and stored in half of four 8-bit registers. For example, the segments for POSITION 0 (the left-most character) must be stored in the lower *nibble* (i.e., the lower 4 bits) of



(d) $(\text{SEGi} - \text{COM0})^2$ = 1 V^2 1 V^2 1 V^2 1 V^2 1 V^2 1 V^2 1 V^2 1 V^2

(e) Mean of $(\text{SEGi} - \text{COM0})^2$ = $(1 + 1 + 1 + 1 + 1 + 1 + 1 + 1)/8 = 1\ V^2$

(f) RMS value = $\sqrt{1\ V^2} = 1.00\ V$

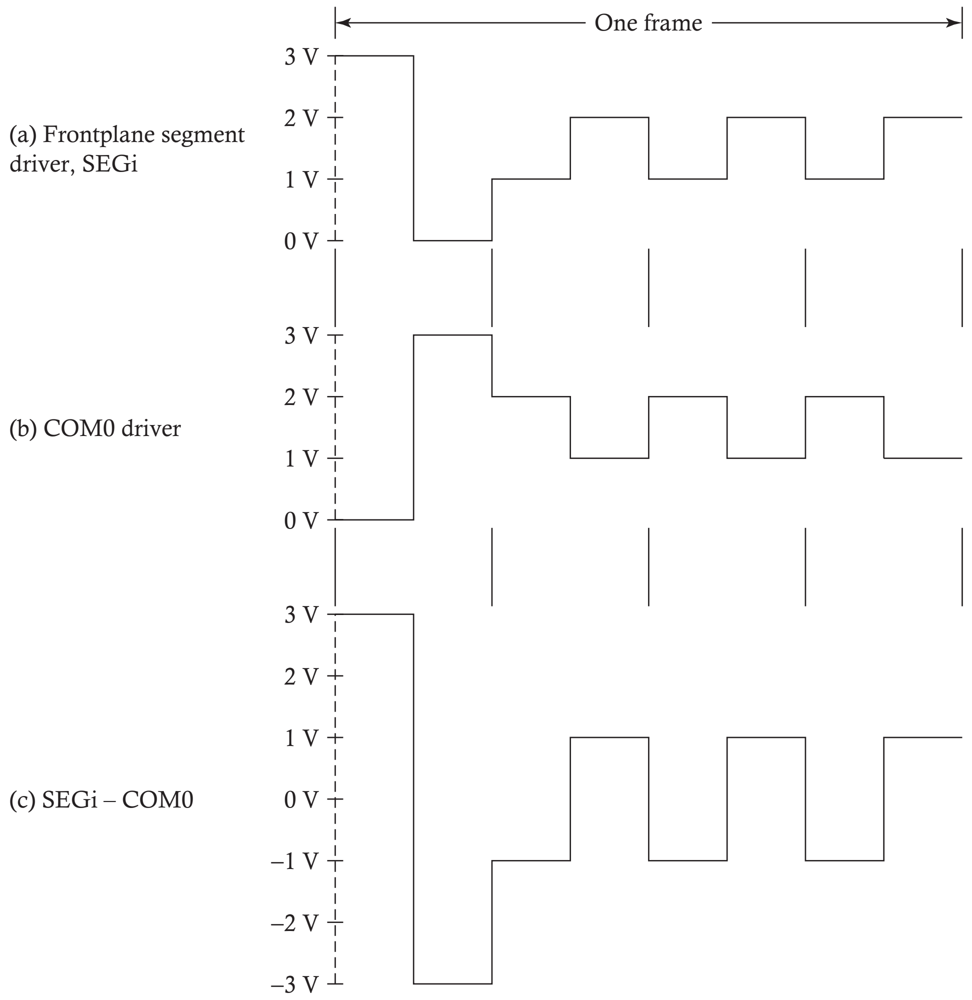
FIGURE 16-5 All four segments selected by SEGi are off

LCDDATA0 and **LCDDATA6** and **LCDDATA12** and **LCDDATA18**

The allocation of the segments for all eight characters is shown in Figure 16-7.

16.6 ASCII CODE TO LCDDATAi REPRESENTATION

Each ASCII-coded character must not only be represented by its 14-segment code of Figure 16-2a. In addition, these 14 bits must be *ordered* for loading into the **LCDDATAi** registers of Figure 16-7. To facilitate this loading, the 8-bit ASCII code is used as an



(d) $(\text{SEGi} - \text{COM0})^2$ $= 9\ V^2 \quad 9\ V^2 \quad 1\ V^2 \quad 1\ V^2 \quad 1\ V^2 \quad 1\ V^2 \quad 1\ V^2 \quad 1\ V^2$

(e) Mean of $(\text{SEGi} - \text{COM0})^2$ $= (9 + 9 + 1 + 1 + 1 + 1 + 1 + 1)/8 = 3\ V^2$

(f) RMS value $= \sqrt{3\ V^2} = 1.73\ V$

FIGURE 16-6 RMS voltage across a selected turned-on segment

offset into the table of 256 2-byte entries shown in Figure 16-8. As shown by the comment at the beginning of this table, each table entry consists of the 4 nibbles

```
JGFB  IHXA  NMDP  KLEC
```

packed into the 2 bytes of an *unsigned int* constant stored in program memory. The LCD.c file of Figure 16-13 will unpack the nibbles for each character to be displayed and load them into the appropriate **LCDDATAi** registers.

POSITION	LCD Pin	Segment	COM0		COM1		COM2		COM3	
			Bit address	Segment	Bit address	Segment	Bit address	Segment	Bit address	Segment
0	35	SEG0	LCDDATA0,0	A	LCDDATA6,0	B	LCDDATA12,0	C	LCDDATA18,0	P
	1	SEG1	LCDDATA0,1	X	LCDDATA6,1	F	LCDDATA12,1	E	LCDDATA18,1	D
	36	SEG2	LCDDATA0,2	H	LCDDATA6,2	G	LCDDATA12,2	L	LCDDATA18,2	M
	2	SEG3	LCDDATA0,3	I	LCDDATA6,3	J	LCDDATA12,3	K	LCDDATA18,3	N
1	33	SEG4	LCDDATA0,4	A	LCDDATA6,4	B	LCDDATA12,4	C	LCDDATA18,4	P
	3	SEG5	LCDDATA0,5	X	LCDDATA6,5	F	LCDDATA12,5	E	LCDDATA18,5	D
	34	SEG6	LCDDATA0,6	H	LCDDATA6,6	G	LCDDATA12,6	L	LCDDATA18,6	M
	4	SEG7	LCDDATA0,7	I	LCDDATA6,7	J	LCDDATA12,7	K	LCDDATA18,7	N
2	31	SEG8	LCDDATA1,0	A	LCDDATA7,0	B	LCDDATA13,0	C	LCDDATA19,0	P
	5	SEG9	LCDDATA1,1	X	LCDDATA7,1	F	LCDDATA13,1	E	LCDDATA19,1	D
	32	SEG10	LCDDATA1,2	H	LCDDATA7,2	G	LCDDATA13,2	L	LCDDATA19,2	M
	6	SEG11	LCDDATA1,3	I	LCDDATA7,3	J	LCDDATA13,3	K	LCDDATA19,3	N
3	29	SEG12	LCDDATA1,4	A	LCDDATA7,4	B	LCDDATA13,4	C	LCDDATA19,4	P
	7	SEG13	LCDDATA1,5	X	LCDDATA7,5	F	LCDDATA13,5	E	LCDDATA19,5	D
	30	SEG14	LCDDATA1,6	H	LCDDATA7,6	G	LCDDATA13,6	L	LCDDATA19,6	M
	8	SEG15	LCDDATA1,7	I	LCDDATA7,7	J	LCDDATA13,7	K	LCDDATA19,7	N
4	27	SEG16	LCDDATA2,0	A	LCDDATA8,0	B	LCDDATA14,0	C	LCDDATA20,0	P
	9	SEG17	LCDDATA2,1	X	LCDDATA8,1	F	LCDDATA14,1	E	LCDDATA20,1	D
	28	SEG18	LCDDATA2,2	H	LCDDATA8,2	G	LCDDATA14,2	L	LCDDATA20,2	M
	10	SEG19	LCDDATA2,3	I	LCDDATA8,3	J	LCDDATA14,3	K	LCDDATA20,3	N
5	25	SEG20	LCDDATA2,4	A	LCDDATA8,4	B	LCDDATA14,4	C	LCDDATA20,4	P
	11	SEG21	LCDDATA2,5	X	LCDDATA8,5	F	LCDDATA14,5	E	LCDDATA20,5	D
	26	SEG22	LCDDATA2,6	H	LCDDATA8,6	G	LCDDATA14,6	L	LCDDATA20,6	M
	12	SEG23	LCDDATA2,7	I	LCDDATA8,7	J	LCDDATA14,7	K	LCDDATA20,7	N
6	23	SEG24	LCDDATA3,0	A	LCDDATA9,0	B	LCDDATA15,0	C	LCDDATA21,0	P
	13	SEG25	LCDDATA3,1	X	LCDDATA9,1	F	LCDDATA15,1	E	LCDDATA21,1	D
	24	SEG26	LCDDATA3,2	H	LCDDATA9,2	G	LCDDATA15,2	L	LCDDATA21,2	M
	14	SEG27	LCDDATA3,3	I	LCDDATA9,3	J	LCDDATA15,3	K	LCDDATA21,3	N
7	21	SEG28	LCDDATA3,4	A	LCDDATA9,4	B	LCDDATA15,4	C	LCDDATA21,4	P
	15	SEG29	LCDDATA3,5	X	LCDDATA9,5	F	LCDDATA15,5	E	LCDDATA21,5	D
	22	SEG30	LCDDATA3,6	H	LCDDATA9,6	G	LCDDATA15,6	L	LCDDATA21,6	M
	16	SEG31	LCDDATA3,7	I	LCDDATA9,7	J	LCDDATA15,7	K	LCDDATA21,7	N

FIGURE 16-7 Allocation of starburst segment bits to PIC18LF6390 registers

```
// Table-entry coding of segments:   J G F B  I H X A  N M D P  K L E C
const rom unsigned int ASCII[] = {
   // ASCII column 0
   0x0100, 0x1000, 0x0001, 0x0020,      // Chris Bruhn's and Peter Ralston's
   0x0002, 0x2000, 0x4000, 0x0400,      // modification for PV.c, their
   0x0800, 0x8000, 0x0008, 0x0080,      // performance verification program
   0x0040, 0x0004, 0x0200, 0x0010,

   // ASCII column 1
   0xffff, 0xcedc, 0xffff, 0xffff,      // CB and PR again
   0xffff, 0xffff, 0xffff, 0xffff,
   0xffff, 0xffff, 0xffff, 0xffff,
   0xffff, 0xffff, 0xffff, 0xffff,

   // ASCII column 2
   0x0000, 0xffff, 0xffff, 0x0480,  // blank,!,",#
   0xffff, 0xffff, 0xffff, 0x0200,  // $,%,&,'
   0x8080, 0x0404, 0xc48c, 0x4848,  // (,),*,+
   0x0000, 0x4008, 0xb137, 0x8004,  //  , ,-,/

   // ASCII column 3
   0xb127, 0x1001, 0x512a, 0x1129,  // 0,1,2,3
   0x7009, 0x6129, 0x612b, 0x8140,  // 4,5,6,7
   0x712b, 0x7129, 0xb137, 0x0000,  // 8,9,0,
   0x8080, 0x4028, 0x0404, 0x7108,  // <,= >,?

   // ASCII column 4
   0xffff, 0x710b, 0x1969, 0x2122,  // @,A,B,C
   0x1961, 0x6122, 0x6102, 0x212b,  // D,E,F,G
   0x700b, 0x0960, 0x1023, 0xe082,  // H,I,J,K
   0x2022, 0xb403, 0x3483, 0x3123,  // L,M,N,O

   // ASCII column 5
   0x710a, 0x31a3, 0x718a, 0x6129,  // P,Q,R,S
   0x0940, 0x3023, 0xa006, 0x3087,  // T,U,V,W
   0x8484, 0x8440, 0x8124, 0xffff,  // X,Y,Z,
   0xffff, 0xffff, 0x0084, 0xffff,  //  , ,^,

   // ASCII column 6
   0xffff, 0x710b, 0x1969, 0x2122,  // @,A,B,C
   0x1961, 0x6122, 0x6102, 0x212b,  // D,E,F,G
   0x700b, 0x0960, 0x1023, 0xe082,  // H,I,J,K
   0x2022, 0xb403, 0x3483, 0x3123,  // L,M,N,O

   // ASCII column 7
   0x710a, 0x31a3, 0x718a, 0x6129,  // P,Q,R,S
   0x0940, 0x3023, 0xa006, 0x3087,  // T,U,V,W
   0x8484, 0x8440, 0x8124, 0xffff,  // X,Y,Z,
   0xffff, 0xffff, 0xffff, 0xffff,  //  , , ,

};
```

FIGURE 16-8 ASCII Table

16.7 AWAKENING VERSUS INTERRUPT VECTORING

The LCD controller expects to be awakened by a falling edge applied to its **INT0** input. This external interrupt in the PIC18LF6390 is identical in operation to the **INT0** input in the PIC18LF4321. However as used in the LCD.c code of Figure 16-13, the interrupt mechanism is initialized all the way up to the global interrupt enable bit, **GIE**, which is left disabled. The effect of the falling edge occurring on the **INT0** pin when the chip is asleep and **GIE** = 0 is to awaken the chip. However, instead of the normal interrupt response (i.e., stacking the program counter and vectoring to an interrupt service routine), the CPU simply executes the instruction that follows the **Sleep** macro.

16.8 RECEPTION OF SPI BYTES INTO VSTRING

The main loop of LCD.c is broken out to Figure 16-9. When the CPU is awakened by the **INT0** input edge, it expects to receive 9 bytes over the Serial Peripheral Interface. As each byte is received, it is loaded into the variable string, **VSTRING**. When all 9 bytes have been received, the **DisplayV** function translates each ASCII-coded byte and loads the resulting nibbles into the appropriate **LCDDATAi** registers.

If, on the other hand, a user sends fewer than 9 bytes, Timer0 is used as a *breakout timer*, to break out of the loop that awaits the reception of bytes that actually were

```
void main()
{
   Initial();                    // Initialize everything
   while (1)
   {
      Sleep();
      Nop();
      CHAR = SSPBUF;             // Clear buffer initially
      INTCONbits.INT0IF = 0;     // Clear wake up flag
      TMR0L = 0;                 // Reset breakout timer
      INTCONbits.TMR0IF = 0;     // Clear breakout timer flag
      for (RECEIVED = 0; RECEIVED < 9; RECEIVED++)  // Receive 9 chars
      {
         PIR1bits.SSPIF = 0;     // Clear SPI flag
                                 // Wait limited amount of time for character
         while ((!PIR1bits.SSPIF) && (!INTCONbits.TMR0IF)) ;
         if (INTCONbits.TMR0IF)
         {
            break;
         }
         VSTRING[RECEIVED] = SSPBUF;  // Get character and put into string
      }
      DisplayV();
   }
}
```

FIGURE 16-9 Main loop

not sent. Rather than wait indefinitely with the (F_{OSC} = 8 MHz) clock drawing its relatively heavy current, the CPU stops waiting, displays what it has received, and returns to sleep.

16.9 DECIMAL POINT

The reception of (the ASCII representation of) a decimal point into **VSTRING** must be treated differently from the reception of any other character, as discussed earlier in Section 5.5. **DisplayV** normally displays each received byte in each successive position of the display. However, **DisplayV** not only reads a byte from **VSTRING** into **CHAR**, it also looks ahead to the next byte in **VSTRING**. If this next byte represents a decimal point, **DisplayV** sets **DPFLAG** and increments past the decimal point. When **WriteCharacter** handles the byte in **CHAR**, it also sets the bit for that character's decimal point.

An apostrophe could also be handled as an exceptional character and packed in with the character it follows. Instead, LCD.c simply takes up an entire character position to display an apostrophe.

16.10 DATA STRUCTURES

To access the 4 nibbles of a selected ASCII table entry, the **WriteCharacter** function first copies the character into **LCDDATA.ALL** as shown in Figure 16-10a. **LCDDATA** is the name of a data structure in RAM, six parts of which have been assigned the members shown in Figure 16-10b. The assigning of these members is shown in Figure 16-10c.

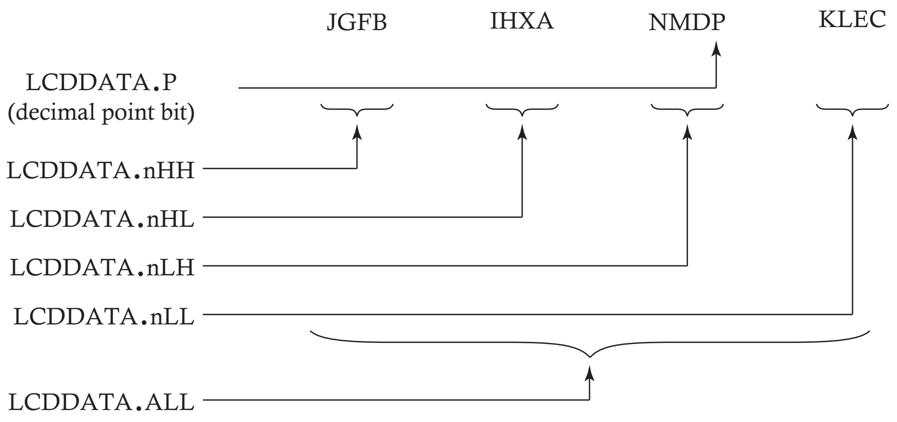
In similar fashion, Alex Singh has created another data structure with members identifying the nibble locations in the registers of Figure 16-7, where the nibbles for the ASCII table entry must be sent to display that character in a certain character position. For example, the 4 nibbles associated with the character to be displayed in the left-most character position (POSITION 0 of Figure 16-7) must be written as follows, given the source of the nibbles from Figure 16-10:

Lower nibble of LCDDATA0	= LCDDATA.nHL	(i.e., IHXA)
Lower nibble of LCDDATA6	= LCDDATA.nHH	(i.e., JGFB)
Lower nibble of LCDDATA12	= LCDDATA.nLL	(i.e., KLEC)
Lower nibble of LCDDATA18	= LCDDATA.nLH	(i.e., NMDP)

The data structure is set up so that a pointer can be used to identify the lowest of the four addresses. From Figure 16-7, it can be seen that the lowest address for POSITION 0 and POSITION 1 is the address of **LCDDATA0**. The lowest address for POSITION 2 and POSITION 3 is the address of **LCDDATA1**, etc. Then, depending on whether POSITION is even or odd, the RAM nibbles will be written to the lower nibble or upper nibble of the selected register.

LCDDATA.ALL = ASCII [CHAR]; // Load table entry to be written

(a) Reading from table into data structure



(b) Desired members within the data structure, LCDDATA

```
union
{
   struct
   {
      unsigned int ALL;   // To identify entire structure
   };
   struct
   {
      unsigned nLL:4;     // To identify 4 nibbles for each table entry
      unsigned nLH:4;
      unsigned nHL:4;
      unsigned nHH:4;
   };
   struct
   {
      unsigned:4;
      unsigned P:1;       // To identify the decimal point bit
   };
} LCDDATA;                // structure to handle table entries as nibbles
```

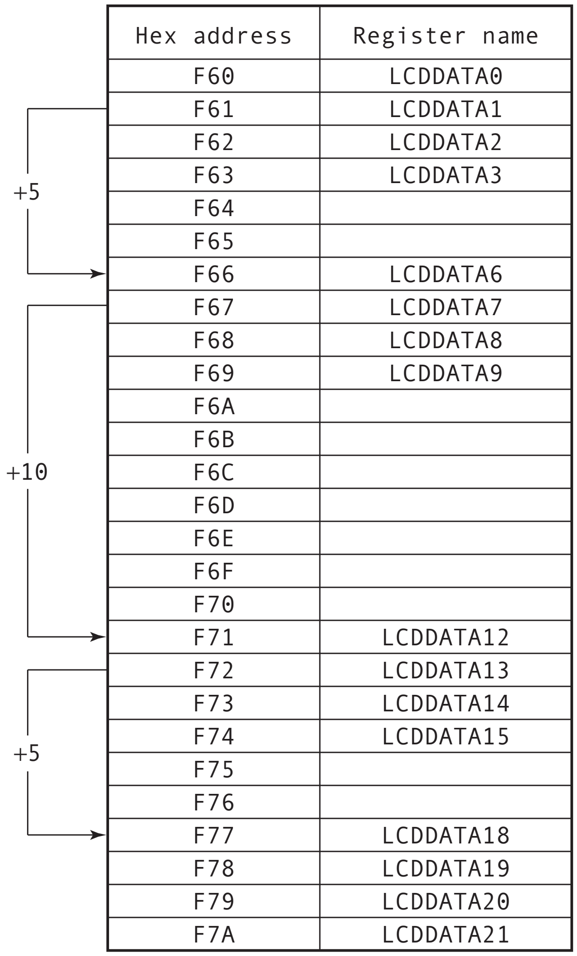
(c) Data structure definition to assign members to parts of ASCII table entry.

FIGURE 16-10 ASCII table entry members

Figure 16-11 lists the actual addresses of the **LCDDATAi** registers. The data structures of Figure 16-12 assign members that will be written to, once the pointer has been set to point to the first address (e.g., the address of **LCDDATA1** for Character Number 2).

As the **DisplayV** function reads an ASCII code from **VSTRING** into **CHAR**, it also forms **POSITION**, the character location destined to display the character. **WriteCharacter**, in turn, forms the pointer:

```
ptr = (oneLCDSEG *) (&LCDDATA0 + POSITION / 2);
```



Hex address	Register name
F60	LCDDATA0
F61	LCDDATA1
F62	LCDDATA2
F63	LCDDATA3
F64	
F65	
F66	LCDDATA6
F67	LCDDATA7
F68	LCDDATA8
F69	LCDDATA9
F6A	
F6B	
F6C	
F6D	
F6E	
F6F	
F70	
F71	LCDDATA12
F72	LCDDATA13
F73	LCDDATA14
F74	LCDDATA15
F75	
F76	
F77	LCDDATA18
F78	LCDDATA19
F79	LCDDATA20
F7A	LCDDATA21

FIGURE 16-11 Actual addresses of LCDDATAi registers

```
typedef struct
{
   long _4bytes;
   char _1byte;
} _5bytes;
```

(a) This structure will be used below to add an offset of five bytes to an address in the oneLCDSEG data structure

```
typedef struct
{
   unsigned nAL:4;              // LCDDATA(x)
   unsigned nAH:4;
   _5bytes a;
   unsigned nBL:4;              // LCDDATA(x+6)
```

FIGURE 16-12 Naming LCDDATAi nibbles

```
   unsigned nBH:4;
   _5bytes b;
   _5bytes c;
   unsigned nCL:4;              // LCDDATA(x+12)
   unsigned nCH:4;
   _5bytes d;
   unsigned nDL:4;              // LCDDATA(x+18)
   unsigned nDH:4;
} oneLCDSEG;
```

(b) oneLCDSEG is a structure that assigns members to LCDDATA.

FIGURE 16-12 *(continued)*

that it uses to write each nibble from the LCDDATA data structure to the nibble of a register with a line exemplified by:

```
ptr→nAL = LCDDATA.nHL;
```

The entire file is listed in Figure 16-13.

```
/******* LCD.c *****************
 *
 * Display a string received having a length of 9 characters (including
 * an optional decimal point).
 *
 * Because of the quirky translation of the starburst segments ABCDEFGHIJKLMNPX
 * into their positions in the LCDDATAi registers, the tables showing the coding
 * for numbers and letters are coded with two-byte table entries in the
 * following order:
 *    J G F B   I H X A     N M D P   K L E C
 * where P is the decimal point and X is the apostrophe.
 * Thus the letter "K" made up of segments EFGJN is translated into the table
 * entry   db  0xe0,0x82      because
 *    1 1 1 0   0 0 0 0     1 0 0 0   0 0 1 0
 *
 * Use Fosc = 8 MHz.                        Starburst display draws 6 uA.
 * Developed by Alex Singh.
 *
 ******* Program hierarchy *****
 *
 * main
 *   Initial
 *   DisplayV
 *     WriteCharacter
 *
 *******************************
 */
```

FIGURE 16-13 LCD.c

```
#include <p18f6390.h>

/*******************************
 * Assembler directives
 *******************************
 */

#pragma config OSC = INTIO7     // Internal osc, RA6=CLKO, RA7=I/O
#pragma config WDT = OFF        // WDT disabled (control through SWDTEN bit)
#pragma config PWRT = ON        // PWRT enabled
#pragma config MCLRE = ON       // MCLR pin enabled; RG5 input pin disabled
#pragma config XINST = OFF      // Instruction set extension disabled
#pragma config BOREN = ON       // Brown-out controlled by software
#pragma config BORV = 3         // Brown-out voltage set for 2.0V, nominal

/*******************************
 * Structure definitions
 *******************************
 */
                                // Structures to map LCDDATA (as nibbles)
typedef struct
{
   long _4bytes;
   char _1byte;
} _5bytes;

typedef struct
{
   unsigned nAL:4;              // LCDDATA(x)
   unsigned nAH:4;
   _5bytes a;
   unsigned nBL:4;              // LCDDATA(x+6)
   unsigned nBH:4;
   _5bytes b;
   _5bytes c;
   unsigned nCL:4;              // LCDDATA(x+12)
   unsigned nCH:4;
   _5bytes d;
   unsigned nDL:4;              // LCDDATA(x+18)
   unsigned nDH:4;
} oneLCDSEG;

/*******************************
 * Global variables
 *******************************
 */

union
{
   struct
```

FIGURE 16-13 *(continued)*

```
   {
      unsigned int ALL;         // To identify entire structure
   };
   struct
   {
      unsigned nLL:4;           // To identify 4 nibbles for each table entry
      unsigned nLH:4;
      unsigned nHL:4;
      unsigned nHH:4;
   };
   struct
   {
      unsigned:4;
      unsigned P:1;             // To identify the decimal point bit
   };
} LCDDATA;                      // Structure to handle table entries as
                                // nibbles

char DPFLAG;                    // Flag to handle a received decimal point
unsigned char CHAR;             // ASCII character from string
unsigned char POSITION;         // LCD character position (0 to 7)
unsigned char i;                // Used as index for loops and VSTRING
unsigned int DELAY;             // Sixteen-bit counter for obtaining a delay
unsigned int j;                 // Used for delay in Initial
unsigned int RECEIVED;          // Used to keep track of characters received
oneLCDSEG *ptr;                 // Pointer that maps to our LCD nibbles
char *CLEARptr;                 // Pointer used to clear all LCDDATA
unsigned char VSTRING[10];      // Variable string to display

/*******************************
 * Constant strings
 *******************************
 */

// Table-entry coding of segments:   J G F B  I H X A  N M D P  K L E C
const rom unsigned int ASCII[] = {

   // ASCII column 0
   0x0100, 0x1000, 0x0001, 0x0020,      // Chris Bruhn's and Peter Ralston's
   0x0002, 0x2000, 0x4000, 0x0400,      // modification for PV.c, their
   0x0800, 0x8000, 0x0008, 0x0080,      // performance verification program
   0x0040, 0x0004, 0x0200, 0x0010,

   // ASCII column 1
   0xffff, 0xcedc, 0xffff, 0xffff,      // CB and PR again
   0xffff, 0xffff, 0xffff, 0xffff,
   0xffff, 0xffff, 0xffff, 0xffff,
   0xffff, 0xffff, 0xffff, 0xffff,
```

FIGURE 16-13 *(continued)*

```
    // ASCII column 2
    0x0000, 0xffff, 0xffff, 0x0480,       // blank,!,",#
    0xffff, 0xffff, 0xffff, 0x0200,       // $,%,&,'
    0x8080, 0x0404, 0xc48c, 0x4848,       // (,),*,+
    0x0000, 0x4008, 0xb137, 0x8004,       // , ,-,/

    // ASCII column 3
    0xb127, 0x1001, 0x512a, 0x1129,       // 0,1,2,3
    0x7009, 0x6129, 0x612b, 0x8140,       // 4,5,6,7
    0x712b, 0x7129, 0xb137, 0x0000,       // 8,9,0,
    0x8080, 0x4028, 0x0404, 0x7108,       // <,= >,?

    // ASCII column 4
    0xffff, 0x710b, 0x1969, 0x2122,       // @,A,B,C
    0x1961, 0x6122, 0x6102, 0x212b,       // D,E,F,G
    0x700b, 0x0960, 0x1023, 0xe082,       // H,I,J,K
    0x2022, 0xb403, 0x3483, 0x3123,       // L,M,N,O

    // ASCII column 5
    0x710a, 0x31a3, 0x718a, 0x6129,       // P,Q,R,S
    0x0940, 0x3023, 0xa006, 0x3087,       // T,U,V,W
    0x8484, 0x8440, 0x8124, 0xffff,       // X,Y,Z
    0xffff, 0xffff, 0x0084, 0xffff,       // , ,^,

    // ASCII column 6
    0xffff, 0x710b, 0x1969, 0x2122,       // @,A,B,C
    0x1961, 0x6122, 0x6102, 0x212b,       // D,E,F,G
    0x700b, 0x0960, 0x1023, 0xe082,       // H,I,J,K
    0x2022, 0xb403, 0x3483, 0x3123,       // L,M,N,O

    // ASCII column 7
    0x710a, 0x31a3, 0x718a, 0x6129,       // P,Q,R,S
    0x0940, 0x3023, 0xa006, 0x3087,       // T,U,V,W
    0x8484, 0x8440, 0x8124, 0xffff,       // X,Y,Z
    0xffff, 0xffff, 0xffff, 0xffff,       // , , ,

};

/********************************
 * Variable strings
 ********************************
 */

/********************************
 * Function prototypes
 ********************************
 */

void Initial(void);
void DisplayV(void);
void WriteCharacter(void);
```

FIGURE 16-13 *(continued)*

```
/*******************************
 * Macros
 *******************************
 */
#define Delay(x) DELAY = x; while(--DELAY){ Nop(); Nop(); }

/*******************************
 * main()
 *******************************
 */

void main()
{
   Initial();                    // Initialize everything
   while (1)
   {
      Sleep();
      Nop();
      CHAR = SSPBUF;             // Clear buffer initially
      INTCONbits.INT0IF = 0;     // Clear wake up flag
      TMR0L = 0;                 // Reset breakout timer
      INTCONbits.TMR0IF = 0;     // Clear breakout timer flag
      for (RECEIVED = 0; RECEIVED < 9; RECEIVED++)  // Receive 9 chars
      {
         PIR1bits.SSPIF = 0;     // Clear SPI flag
                                 // Wait limited amount of time for character
         while ((!PIR1bits.SSPIF) && (!INTCONbits.TMR0IF)) ;
         if (INTCONbits.TMR0IF)
         {
            break;
         }
         VSTRING[RECEIVED] = SSPBUF;  // Get character and put into string
      }
      DisplayV();
   }
}

/*******************************
 * Initial()
 *
 * This subroutine performs all initializations of variables and registers.
 *******************************
 */

void Initial()
{
   OSCCON = 0b01110010;          // Select 8 MHz internal oscillator
   LCDSE0 = 0b11111111;          // Enable all LCD segments
   LCDSE1 = 0b11111111;
   LCDSE2 = 0b11111111;
   LCDSE3 = 0b11111111;
   LCDCON = 0b10001011;          // 1/4 mux; INTRC clock
```

FIGURE 16-13 *(continued)*

```
   LCDPS = 0b00110110;           // 37 Hz frame frequency
   CLEARptr = (char *) &LCDDATA0;  // Point to first segment
   for (i = 0; i < 28; i++)      // Turn off all segments
   {
      *CLEARptr++ = 0x00;
   }
   LCDDATA21 = 0b00010000;       // Turn on rightmost decimal point initially
   ADCON1 = 0b00111111;          // Make all ADC/IO pins digital
   TRISA = 0;                    // Make all pins outputs but RB0, SCK, SDI
   TRISB = 0b00000001;
   TRISC = 0b00011000;
   PORTA = 0;
   PORTB = 0;
   PORTC = 0;
   SSPCON1 = 0b00110101;         // Initialize SPI as slave
   SSPSTAT = 0b00000000;
   T0CON = 0b11000011;           // Use Timer0 to timeout on incomplete input
   INTCON2bits.INTEDG0 = 0;      // Wake up with falling edge on INT0
   INTCONbits.INT0IF = 0;        // Clear flag
   INTCONbits.INT0IE = 1;        // Enable INT0 source
   // Don't enable interrupt, only wakeup
   DPFLAG = 0;
   Delay(30000);                 // Initial delay is 300/2 milliseconds
   RCONbits.SBOREN = 0;          // Now disable brownout reset
}

/*******************************
 * DisplayV()
 *
 * This subroutine displays the string stored in VSTRING
 *******************************
 */

void DisplayV()
{
   // Iterate through all received
   for (i = 0, POSITION = 0; i < RECEIVED; i++, POSITION++)
   {
      CHAR = VSTRING[i];         // Save byte
      if (VSTRING[i + 1] == '.')  // Deal with decimal point
      {                          // Check next character for decimal point
         DPFLAG = 1;             // If it is, set flag
         i++;                    // and increment pointer past decimal point
      }
      WriteCharacter();          // Display current character
   }
}
```

FIGURE 16-13 *(continued)*

```
/******************************
 * WriteCharacter()
 *
 * This subroutine writes the selected character to the display
 ******************************
 */

void WriteCharacter()
{
   LCDDATA.ALL = ASCII[CHAR];   // Load table entry to be written
                                // Ppoint to corresponding LCDDATAs
   ptr = (oneLCDSEG *) (&LCDDATA0 + POSITION / 2);
   if (DPFLAG)                  // if FLAG is set
   {
      LCDDATA.P = 1;            // Write decimal point
      DPFLAG = 0;               // and clear flag
   }
   if (!(POSITION % 2))         // If even position
   {
      ptr->nAL = LCDDATA.nHL;   // write to lower nibbles
      ptr->nBL = LCDDATA.nHH;
      ptr->nCL = LCDDATA.nLL;
      ptr->nDL = LCDDATA.nLH;
   }
   else                         // If odd position
   {
      ptr->nAH = LCDDATA.nHL;   // write to upper nibbles
      ptr->nBH = LCDDATA.nHH;
      ptr->nCH = LCDDATA.nLL;
      ptr->nDH = LCDDATA.nLH;
   }
}
```

FIGURE 16-13 *(continued)*

PROBLEMS

16-1 Algorithm testing This problem requires the availability of a PICkit 2 programmer plus a 6-pin, 100-mil-spaced header to bridge between the programmer's 6-pin female output and the Qwik&Low board's unpopulated H5 "LCD PICkit 2" header.

- Modify the LCD.c code to set **RC7** before the call of **WriteCharacter** in **DisplayV** and to clear **RC7** on the return from **WriteCharacter**. Compile your modified LCD.c code with the same c18.exe utility used to compile the code for the PIC18LF4321. Note the message produced by this utility indicating its recognition (and acceptance) of this code for the PIC18LF6390 LCD controller chip.

- Use the PICkit 2 programming utility (rather than QwikProgram used to program QwikBug into the PIC18LF4321) to program the LCD controller chip. When you do this, make sure that SW2, the big 4PDT switch in the center of the board, is down or in (i.e., on, not off). Otherwise, this switch shorts the LCD controller's V_{DD} pin to ground, as can be seen in Figure 16-3.
- Monitor test point TP7 with a scope while running MCU code that writes to the LCD display. This test point is labeled on the back of the board below TP8, TP9, TP10, and TP11. It can, or course, be probed from the front of the board at the pad located below the TP8 label.
  a) Does the duration of **WriteCharacter** vary with the position of the character on the display? To test this, have the MCU send "AAAAAAAAA" to the display. What is this duration?
  b) Does the duration of WriteCharacter vary with the character code? To test this, you already know the effect of the position of the character. Now have the MCU send "AOPZ.aopz" to the display. What do you find?

Chapter

17

SPI FOR FEATURE ENHANCEMENT

17.1 OVERVIEW

The *Serial Peripheral Interface* (SPI) facility used by the MCU to send a display string to the LCD controller can also be employed for communication with other peripheral chips to enhance the features of the MCU. This chapter will begin with the clocking options used to match the SPI to another chip's SPI. It will end with the use of two peripheral chips.

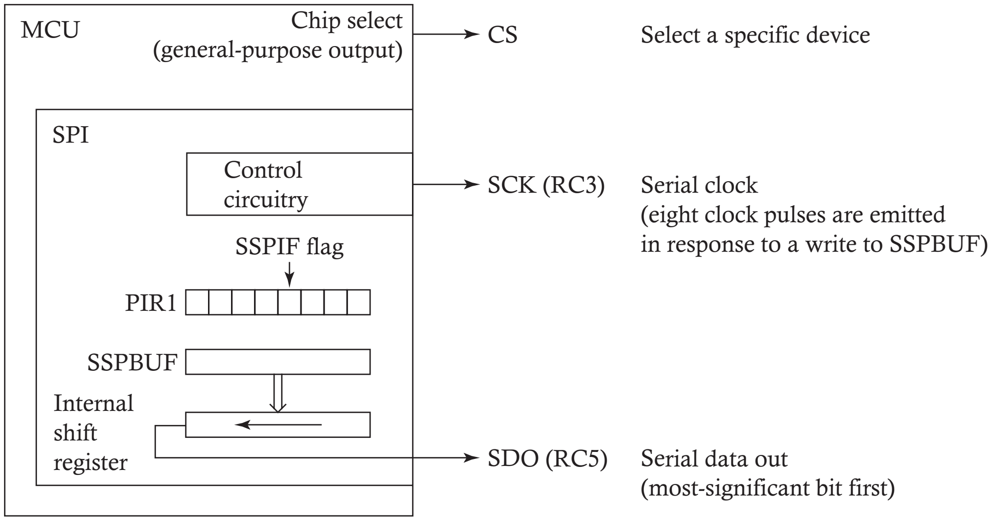
The SPI circuitry is part of the PIC18LF4321's *Master Synchronous Serial Port* (MSSP) module that supports either SPI or I^2C bus transfers. The Qwik&Low board is committed to the SPI function because:

- Two of the three SPI pins are already committed to communication with the LCD controller.
- I^2C bus use requires a relatively low 2.2-KΩ pull-up resistor for each of its two I^2C pins. These resistors present a large current draw during I^2C transfers. They must also be individually disabled for SPI transfers that use the same pins.
- SPI transfers are inherently faster than I^2C transfers, and thereby take up less CPU awake time. The SPI clock is more than twice as fast as that specified for I^2C. I^2C message strings always require the time to send one more (address) byte than a similarly functioning SPI chip would require.

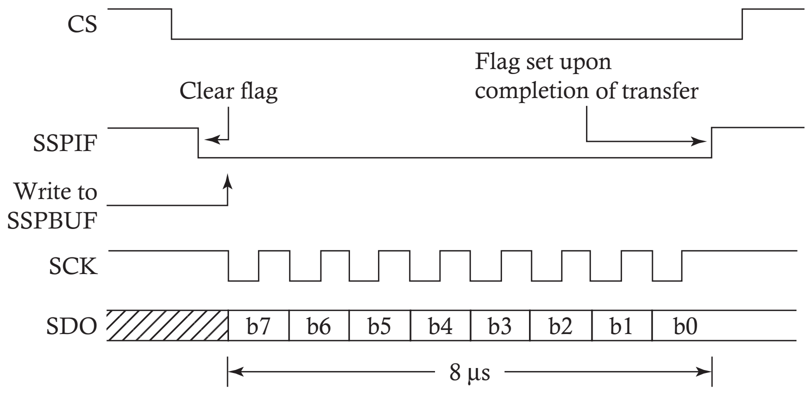
- Many peripheral functions available with an I^2C interface are also available with an SPI interface. For example, the AD5601 digital-to-analog (DAC) considered later in this chapter is the SPI counterpart of the AD5602, a DAC with an I^2C interface and an otherwise identical feature set.

17.2 SPI OUTPUT FUNCTIONALITY

The SPI functioning for transfers to the LCD controller was illustrated in Figure 5-1 and is repeated in Figure 17-1. Note that the byte that is written to the **SSPBUF** register is transferred out of the MCU *most-significant bit* first. This is a defining characteristic of the SPI interface. Peripheral chips with an SPI interface are designed to use transferred data in this most-significant-bit-first order.

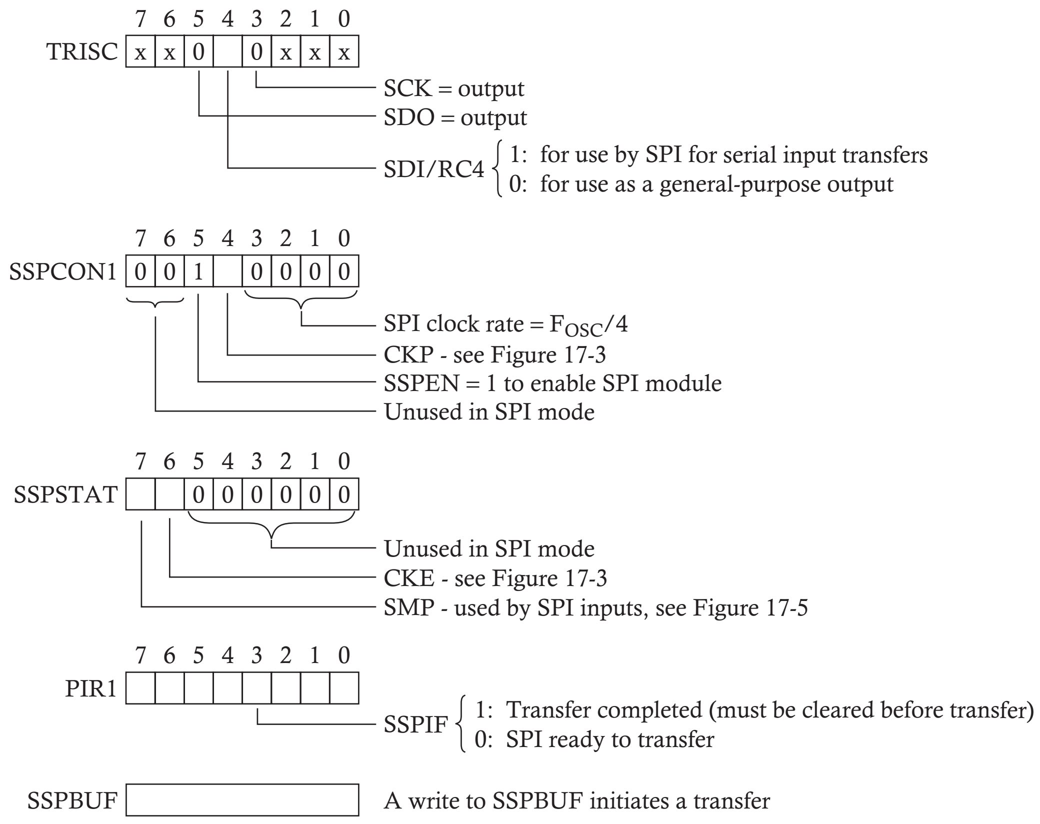


(a) Function of pins



(b) Waveforms

FIGURE 17-1 SPI use for serial output



TRISC
7 6 5 4 3 2 1 0
x x 0 0 x x x
SCK = output
SDO = output
SDI/RC4
1: for use by SPI for serial input transfers
0: for use as a general-purpose output
SSPCON1
0 0 1 0 0 0 0
SPI clock rate = $F_{OSC}/4$
CKP - see Figure 17-3
SSPEN = 1 to enable SPI module
Unused in SPI mode
SSPSTAT
0 0 0 0 0 0
Unused in SPI mode
CKE - see Figure 17-3
SMP - used by SPI inputs, see Figure 17-5
PIR1
SSPIF
1: Transfer completed (must be cleared before transfer)
0: SPI ready to transfer
SSPBUF
A write to SSPBUF initiates a transfer

FIGURE 17-2 SPI registers

The registers associated with the SPI module are shown in Figure 17-2. If the SPI is only being used for output transfers (as it is for the Qwik&Low board without add-on parts), then the initialization of bit 4 of **TRISC** to zero (as it is initialized in all of the earlier template files) allows the **SDI** pin to be used as a general-purpose output pin, **RC4**.

The SPI clock rate is controlled by the lower 4 bits of **SSPCON1**. The 0000 choice shown sets this rate to its maximum value of 1 MHz when F_{OSC} = 4 MHz so that an 8-bit transfer will take just 8 μs.

Two control bits, **CKP** and **CKE**, serve a vital role for SPI output transfers. **CKP** selects whether the clock output, **SCK**, idles high or low. These two control bits are initialized before initiating an SPI transfer to a device. If the device requires each incoming bit to be stable at the time of the *rising edge* of its **SCK** input, then either Figure 17-3a or c will serve. On the other hand, if the Figure 17-3b or d choice were made, that device would see a changing, ambiguous bit on **SDO** at the time of each rising edge of **SCK**.

For a device that requires stable data with the *rising edge* of the clock, either Figure 17-3a or c will work. For a device that requires stable data with the *falling edge* of the clock, either Figure 17-b or d will work. Between the two choices for a device, the better choice is the one that does not change **CKP** from its previous value. Changing

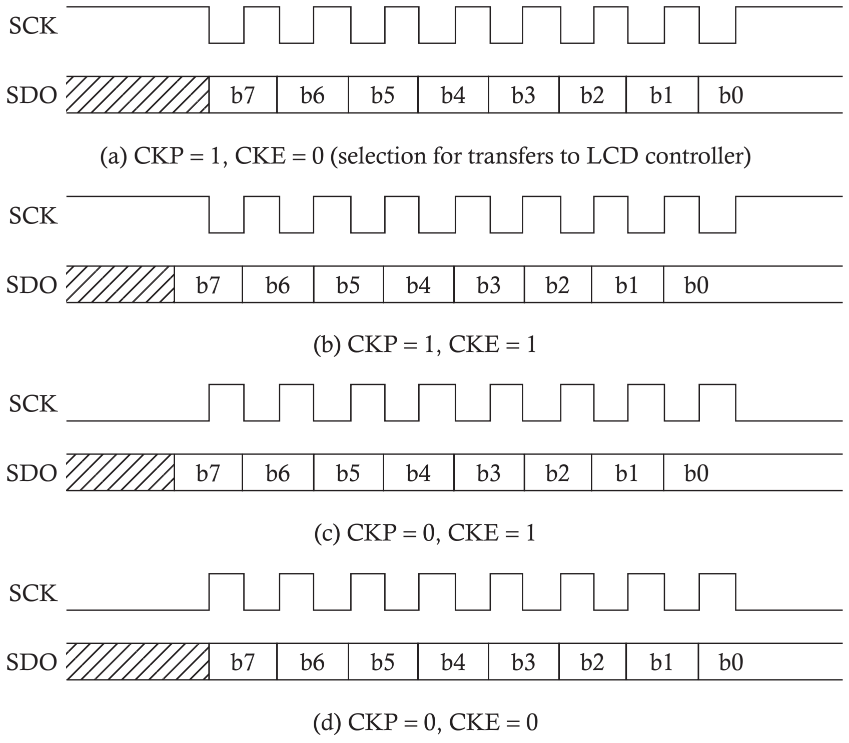


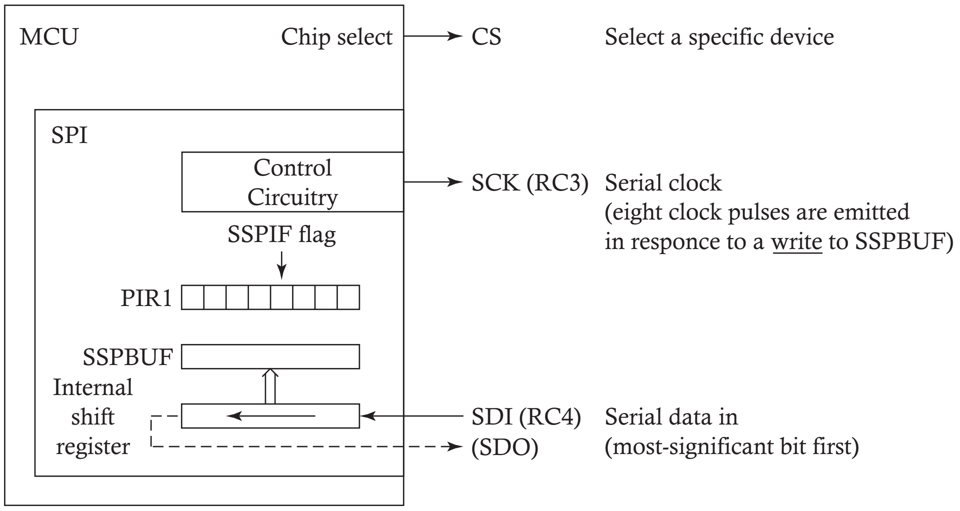
FIGURE 17-3 CKP and CKE options for output transfers

CKP introduces two extra edges, one of which will clock falling-edge-sensitive devices and one of which will clock rising-edge-sensitive devices. For example, as indicated by Figure 17-3a, the LCD controller is sensitive to rising clock edges. If **CKP** is changed to zero for an SPI transfer to another device, that change in **CKP** will drop the idle **SCK** line from high to low. That change may clock the other device and produce an inadvertent 9-bit transfer to the device. Likewise, when **CKP** is changed back to one for the next SPI transfer to the LCD controller, an extra rising edge will occur on the **SCK** input to the LCD controller. If the LCD controller's CPU has not yet been awakened by the interrupt input from the MCU (see Figure 5-1a), then this extra edge will be ignored. On the other hand, less careful dealing with this sequence by *first* dropping **RD5** to wake up the LCD controller's CPU and *then* changing **CKP** will clock a garbage bit into the LCD controller and thereby garble what is received. This potential problem never arises if **CKP** is never changed.

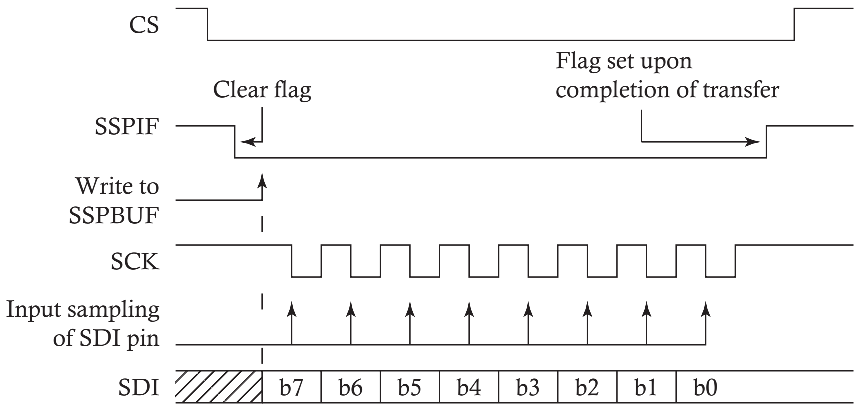
SPI output transfers to a device generally involve an extra control line in addition to the **SCK** to **SCK** connection and the **SDO** to **SDI** connection (see Figure 5-1). In the case of the MCU transfers to the LCD controller, the extra control line is the MCU's **RD5** pin used to awaken the LCD controller's CPU with a falling-edge **INT0** interrupt. Some other devices use the extra input as a chip enable, ignoring the wiggling **SCK** and **SDO** lines except when enabled. However it is used, this extra input to a device must serve the device in the manner specified in that device's data sheet.

17.3 SPI INPUT FUNCTIONALITY

Reading serial input from an SPI peripheral chip makes use of the three pins of Figure 17-4a, with a general-purpose output pin used as specified by the peripheral chip's data sheet. The output from this pin may serve as a chip enable, taking a tri-state **SDO** output from the peripheral chip out of its high-impedance state so it can drive the MCU's **SDI** input. For another chip, the output from this pin may load a register with the content to be shifted back to the MCU. In any case, after clearing the **SSPIF** flag in the **PIR1** register, a *write* to the **SSPBUF** register initiates the shifting of whatever was written to **SSPBUF** out of the **SDO** pin. At the same time, whatever the external device presents to the **SDI** pin is shifted into **SSPBUF**.



(a) Function of pins



(b) Example waveforms

FIGURE 17-4 SPI use for serial input

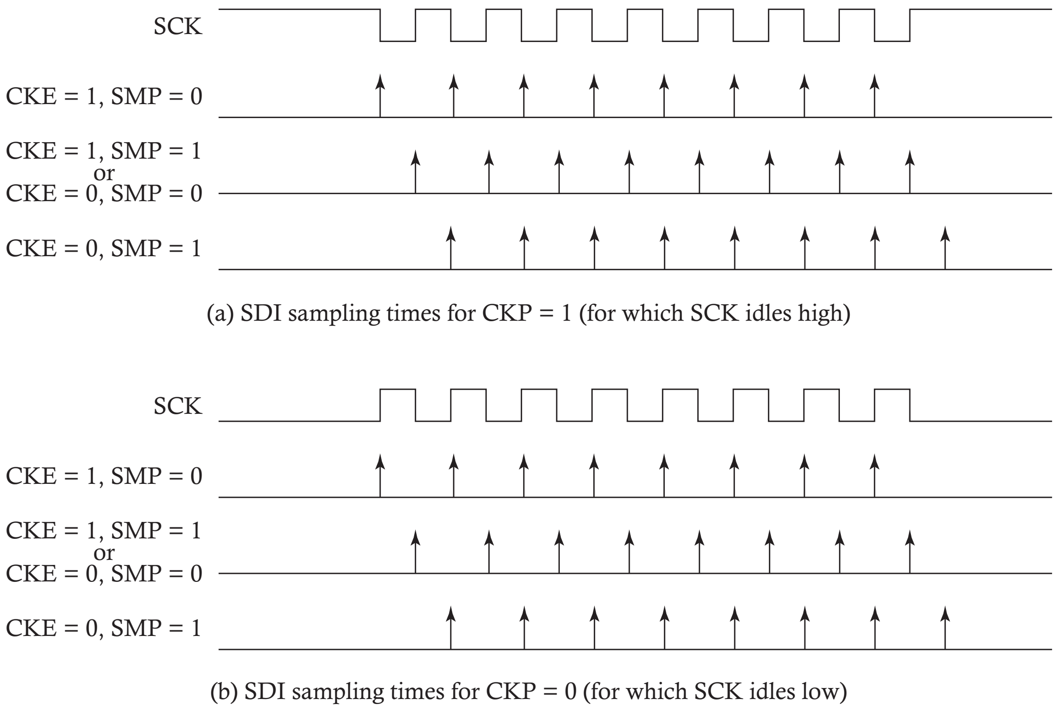


FIGURE 17-5 CKP, CKE, and SMP options for input transfers

As in the case of SPI output transfers, an SPI input transfer can use any of several timing alternatives. In addition to the role of the **CKP** and **CKE** control bits, input transfers are also controlled by an **SMP** sampling bit, as shown in Figure 17-4b. If the input transfer makes use of the same **CKP** = 1 value used for output transfers to the LCD controller, the **SCK** line will not experience any extra, spurious clock edges that might otherwise throw the input data off by 1-bit time, as discussed in the last section.

Figure 17-5a shows the three timing alternatives in the case for which **CKP** = 1. For example, the "chip select" pin to the peripheral chip may load its SPI output register so that the most-significant bit is then sitting on its **SDO** output and, thereby, on the MCU's **SDI** input. If the peripheral chip is sensitive to rising **SCK** clock edges, the use of the **CKP** = 1, **SMP** = 0 timing choice will have the MCU's SPI sampling its **SDI** input *before* the first shift takes place, just as desired. The alternative sampling time choices are obtained by loading **SSPSTAT** appropriately, as shown in Figure 17-2.

17.4 AD5601 DAC OUTPUT

Analog Devices has a family of three tiny 6-pin nanoDAC® digital-to-analog converters for 8-bit, 10-bit, or 12-bit conversions of an input value, N, to an output voltage

$$V_{OUT} = \frac{N}{2^k} \times V_{DD}$$

where k = 8 for the AD5601, k = 10 for the AD5611, and k = 12 for the AD5621. The units operate over a supply voltage range of 2.7 V–5.5 V. One of these units can be

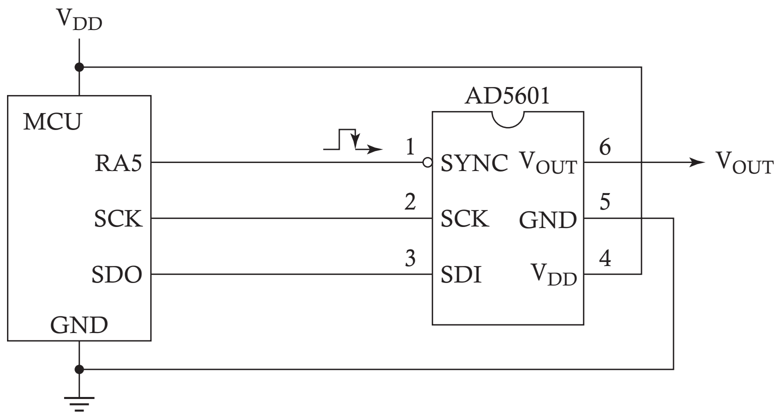


FIGURE 17-6 AD5601 DAC circuit connections

soldered to the Qwik&Low board using the 0.65 mm surface-mount pattern located on the back of the board. Using the circuit of Figure 17-6, it can be connected to the MCU with 30-gauge wirewrap wire soldered between the AD5601 pins and the pins of the H4 terminal strip.

At power-on reset, one of these units typically draws 40 μA, but when sent a standby command, the current draw drops to 0.1 μA. When sent a power-up command to produce an output voltage, the unit typically draws 60 μA, producing an output with the very low output impedance of 0.5 Ω. In standby mode, the output impedance will be

- 1 kΩ to ground
- 100 kΩ to ground
- Three-state (i.e., essentially open circuit)

depending on 2 bits of the *int* value sent to the chip.

A block diagram of the DAC is shown in Figure 17-7. The resistor string-multiplexer combination ensures a *monotonic* output; that is, an output that always increases as the digital input increases. A *relative accuracy* specification of ±0.5 LSb for the 8- and 10-bit DACs means that the output deviates from a straight line between 0 V and V_{DD} by no more than

$$\pm 0.5 \times \frac{V_{DD}}{2^k}$$

For the 12-bit DAC, the specification is relaxed slightly to ±1 LSb. This DAC family exhibits close to ideal specs, even including a typical settling time of 6 μs.

To control the DAC, 2 bytes must be sent to the chip over the SPI to load its input register, with the effect shown in Figure 17-8. The chip's SPI interface expects to see stable data on its **SDI** input at the time of the falling edges on its **SCK** input. This timing dictates the choice of Figure 17-3b. Therefore, before the transfer is initiated, **CKE** is set to one (from its value of zero dictated by the LCD controller's requirement) as shown in Figure 17-9. The DAC then expects a low→high→low pulse on its **SYNC** input. With the circuit of Figure 17-6, this is accomplished by

```
PORTAbits.RA5 = 1;

PORTAbits.RA5 = 0;
```

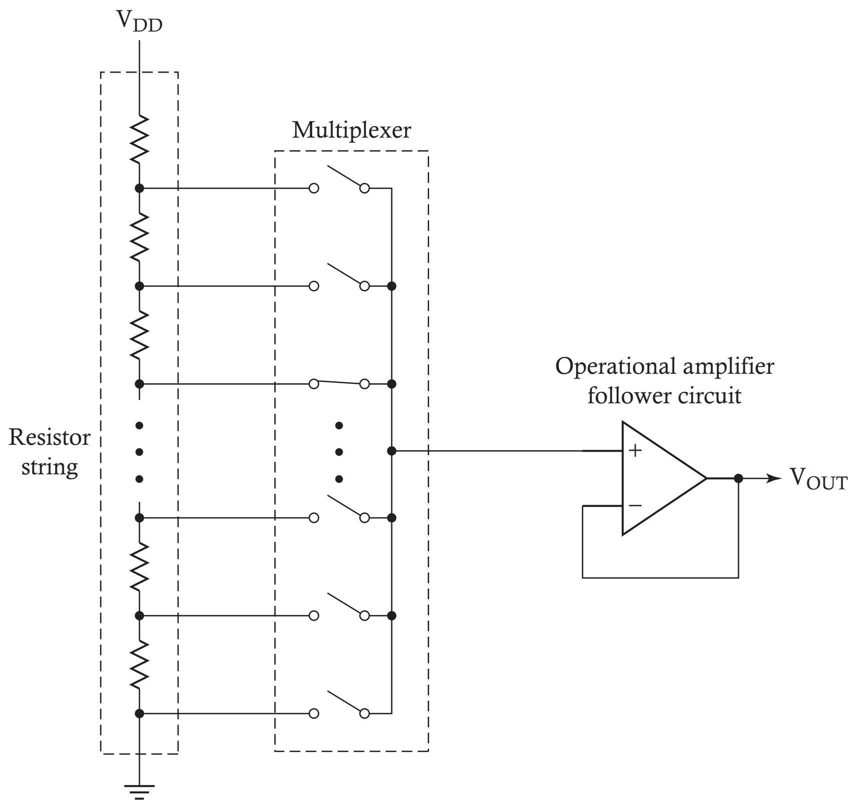


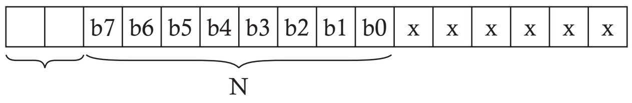
FIGURE 17-7 Block diagram of AD5601 DAC

With F_{OSC} = 4 MHz, this sequence will produce a 1-µs pulse, easily meeting the **SYNC** pulse-width specification of 20-ns minimum. The chip uses this pulse to reset, or synchronize, its SPI interface. It expects this pulse to be followed by 16 falling edges on its **SCK** input. It will shift the **SDI** input into its DAC input register in response to each of these falling edges. In response to the 16th falling edge, the DAC will act on the input register contents, as specified in Figure 17-8.

The sequence of events to update the DAC is shown in Figure 17-9. After setting the **CKE** bit, and pulsing the **SYNC** input, the MCU clears the **SSPIF** bit, writes the upper byte to **SSPBUF**, waits for **SSPIF** = 1, clears it again, writes the lower byte to **SSPBUF**, waits for **SSPIF** = 1, and finally clears the **CKE** bit to leave this bit in its default state for the LCD controller. If **DACREG** is the 16-bit *int* variable to be sent to the DAC, then

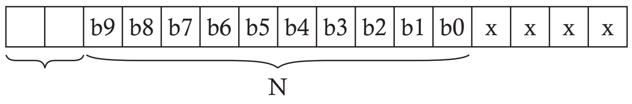
```
DACREGL = DACREG & 0x00FF;   //Form lower byte

DACREGH = DACREG >> 8;       //Form upper byte
```

forms the two 8-bit *unsigned char* variables to be written to **SSPBUF**.



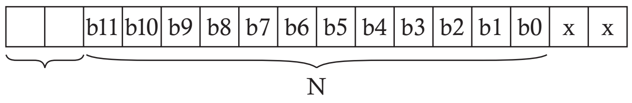
0	0	Power up; generate output
0	1	Standby with output of 1 kΩ to ground
1	0	Standby with output of 100 kΩ to ground
1	1	Standby with open-cicuit output

(a) AD5601 input register (for 8-bit DAC)



0	0	Power up; generate output
0	1	Standby with output of 1 kΩ to ground
1	0	Standby with output of 100 kΩ to ground
1	1	Standby with open-cicuit output

(b) AD5611 input register (for 10-bit DAC)



0	0	Power up; generate output
0	1	Standby with output of 1 kΩ to ground
1	0	Standby with output of 100 kΩ to ground
1	1	Standby with open-cicuit output

(c) AD5621 input register (for 12-bit DAC)

FIGURE 17-8 DAC input register

17.5 MOSI/MISO TERMINOLOGY

The Serial Peripheral Interface was developed many years ago by Motorola. The intent was to have an interface that could *swap* the contents of two 8-bit registers between two devices using the 3-wire interface shown in Figure 17-10. The term *master-out*, *slave-in* (MOSI) is applied to the line that transmits data from the *master* (i.e., the chip that drives the clock line) to the *slave* (i.e., the chip that uses its clock pin as a clock *input*). This MOSI term (and the corresponding MISO term) avoids the naming confusion associated with a line that connects an **SDO** pin on one chip to an **SDI** pin on another chip.

A fourth, active-low *slave-select* ($\overline{\textbf{SS}}$) input is also defined for the slave, driven by a corresponding slave-select output from the master. For multiple slaves (the configuration being considered in this chapter), each slave has a single ($\overline{\textbf{SS}}$) input while the master has one ($\overline{\textbf{SS}}$) for each slave. The original intent of the ($\overline{\textbf{SS}}$) input was to force a slave's **MISO** output to the high-impedance state and to keep a slave from responding to activity on the **MOSI** and **SCLK** lines unless its ($\overline{\textbf{SS}}$) line was driven low by the master for the duration of the transfer. Present usage employs a more generally defined pin to signal the slave of an impending transfer (e.g., the **SYNC** input of the

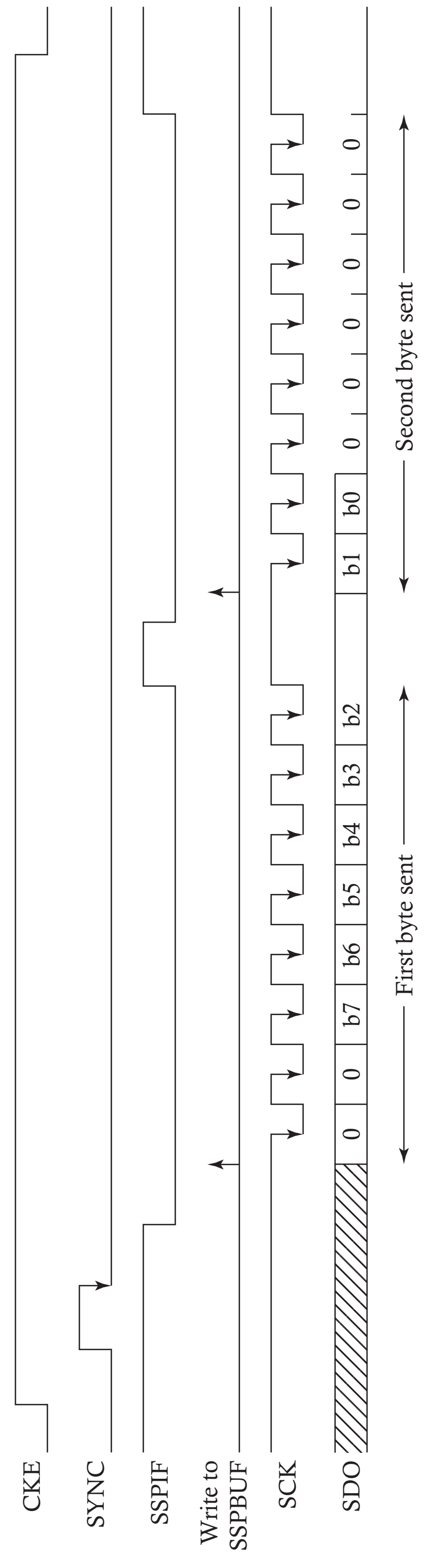


FIGURE 17-9 SPI protocol for 8-bit DAC

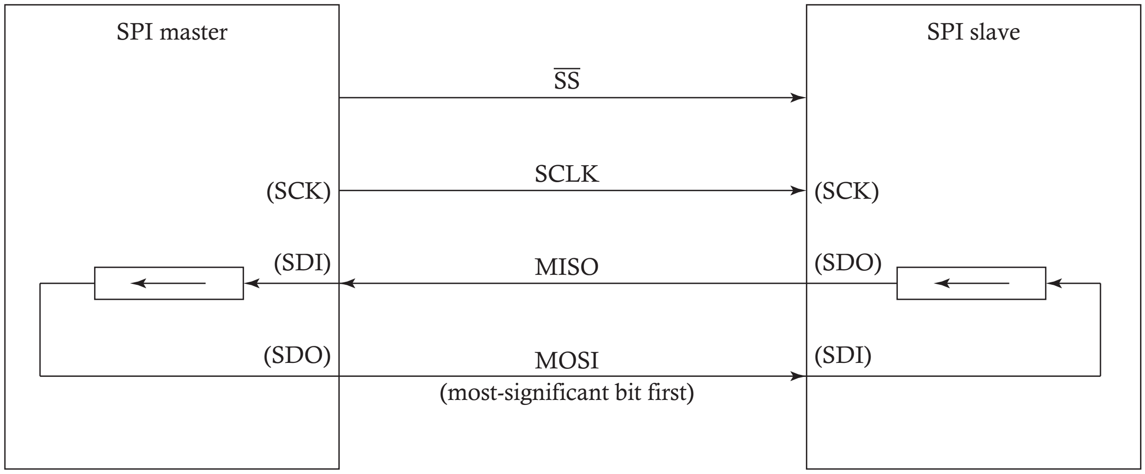


FIGURE 17-10 MOSI/MISO terminology

last section or the **INT0** input of the LCD controller) in addition to the original role of framing the transfer. For any device, it is important to read the data sheet to determine how the device expects this input to be used.

The Motorola standard defines four SPI modes. Each of these modes specifies a combination of **CKP**, **CKE**, and **SMP** of Figures 17-3 and 17-5 so that data is read by both master and slave half a clock period before data is changed. Two parameters are used to define the four modes, as shown in Figure 17-11:

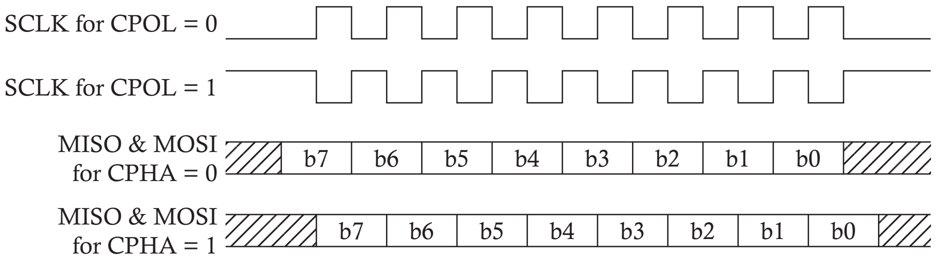
- **CPOL** = 0 means the clock line idles low (same as **CKP** = 0)
- **CPOL** = 1 means the clock line idles high (same as **CKP** = 1)
- **CPHA** = 0 means sample on the leading clock edge
- **CPHA** = 1 means sample on the trailing clock edge

The table of Figure 17-11 also lists the values of **CKP**, **CKE**, and **SMP** that produce each mode, derived from a translation of Figures 17-3 and 17-5.

17.6 ADT7301 SPI TEMPERATURE SENSOR

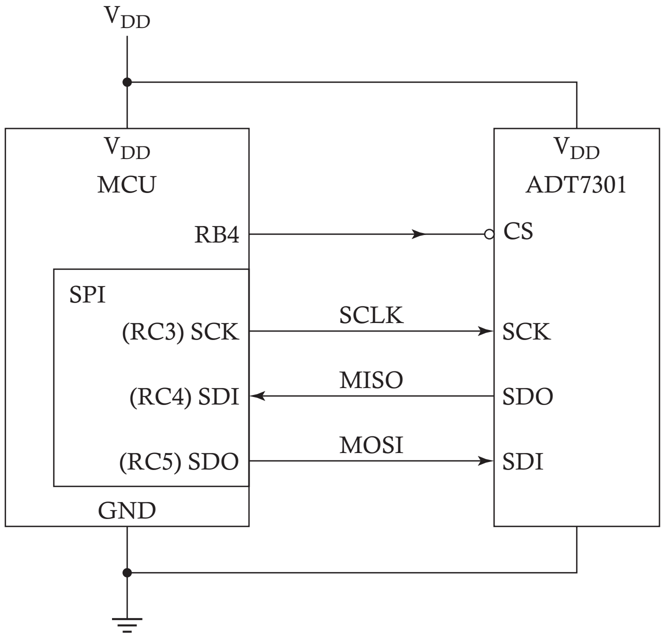
Analog Devices makes an SPI-connected temperature sensor, their ADT7301. It is available in a 6-pin SOT-23 package or an 8-pin SOIC package, either of which can be added to the surface-mount patterns in the lower-right corner of the Qwik&Low board. Because the chip powers up into an active mode, drawing almost 200 µA of current, the chip must be initialized by sending it a shutdown command, after which current drawn by the chip drops to less than a microampere.

A schematic for connecting this temperature sensor to the MCU is shown in Figure 17-12a. Features of the chip, listed in Figure 17-12b, include a resolution of well less than a tenth of a degree Fahrenheit. While the chip's current as it carries out



MODE	CPOL	CPHA	Clock idles	Clock edge upon which data is read	Clock edge upon which data is changed	CKP	CKE	SMP
0	0	0	low	rising edge	falling edge	0	1	0
1	0	1	low	falling edge	rising edge	0	0	0
2	1	0	high	falling edge	rising edge	1	1	0
3	1	1	high	rising edge	falling edge	1	0	0

FIGURE 17-11 SPI mode numbers and correlation to CKP, CKE, SMP



(a) Schematic

FIGURE 17-12 ADT7301 temperature sensor

Supply voltage	+2.7V to 5.5V
Accuracy	±0.5°C from 0°C to 70°C
Resolution	0.03125°C = 0.05625°F
Operating temperature range	-40°C to +150°C
Supply current	
Converting	1600 μA, typical
Not converting	190 μA, typical
Shutdown mode	0.2 μA, typical
Conversion time	208 μs
Package	6-lead SOT-23 or 8-lead SOIC

(b) Features

Temperature	**Output**
-0.03125°C	11 1111 1111 1111
0°C	00 0000 0000 0000
+0.03125°C	00 0000 0000 0001
1°C	00 0000 0010 0000
32°C	00 0100 0000 0000

(c) Conversion examples

FIGURE 17-12 *(continued)*

a conversion is high, a conversion once per second with the chip otherwise shut down produces an average current draw on the coin cell supply of only

$$\frac{208}{1{,}000{,}000} \times 1{,}600 = 0.33\ \mu\text{A}$$

Alternatively, the chip might be sent the serial command to power up and begin a conversion once per second. If, 10 ms later, the MCU shuts down the sensor and at the same time retrieves the converted result, the average current drawn by the temperature sensor is

$$0.33\ \mu\text{A} + \frac{10}{1{,}000} \times 190\ \mu\text{A} = 2.2\ \mu\text{A}$$

Figure 17-13a shows the two commands that the chip understands. Although it is the state of the third bit received that draws the distinction between the shutdown command and the wakeup and convert command, the chip requires the reception of all 16 clock edges before the **CS** pin is raised or it will ignore the command.

The simultaneous operation of reading out an already converted result and putting the chip in shutdown mode is illustrated in Figure 17-13b. The examples of Figure 17-12c illustrate that if the 2-byte result is formed in a *signed int* variable, **RAWTEMP**, the temperature will be expressed in units of 0.03125°C. For a temperature above 0°C, the Centigrade temperature is expressed as

```
CENTIGRADE = RAWTEMP >> 5;      //Temperature in units of degrees
```

or

```
TENTHS = (10 * RAWTEMP) >> 5;   //Temperature in units of 0.1 degree
```

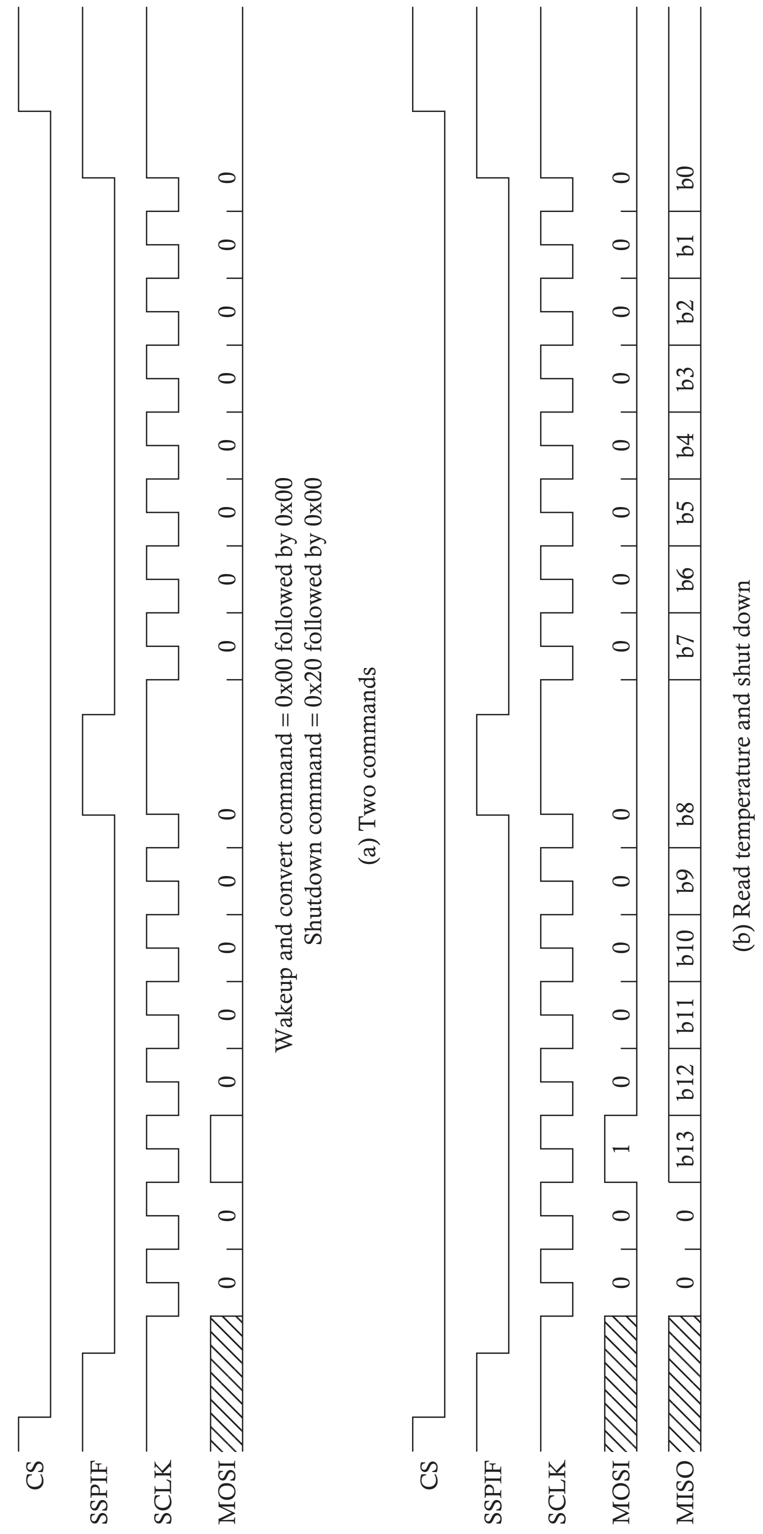

CS
SSPIF
SCLK
MOSI
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
Wakeup and convert command = 0x00 followed by 0x00
Shutdown command = 0x20 followed by 0x00
(a) Two commands
CS
SSPIF
SCLK
MOSI
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
MISO
0 0 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0
(b) Read temperature and shut down

FIGURE 17-13 SPI use with ADT7301 with Mode 3 operation (CKP=1, CKE=0, SMP=0)

PROBLEMS

17-1 DAC Output

a) Write a little **InitDAC** function that sends a standby command to the AD5601 digital-to-analog converter described in Section 17.4 for an output impedance of 1 kΩ and a current draw of less than 1 μA. Assume the chip is connected to the MCU with the circuit of Figure 17-6.

b) If you are willing to add a call of this **InitDAC** from the **Initial** function for all of the code that subsequently uses this augmented board, follow the directions at the beginning of Section 17.4 to add the part and to wire the connections. (Without running **InitDAC**, any code that uses the board will suffer an extra current draw of about 40 μA.)

c) Write a DAC.c program that powers up the pot and keeps it powered up. Then read the 8-bit output of the pot and write it to the DAC every tenth of a second. Using two DMMs, measure the DC voltage of the DAC output relative to the pot output and also measure the pot voltage as you turn the pot from full CCW to full CW. Make a plot of this voltage difference versus the pot voltage over the full range of the pot output.

17-2 Temperature sensor

a) As was done for the last problem, and for the same reason, write a little **Init7301** function that sends the shutdown command described in Figure 17-13a assuming the circuit connections of Figure 17-12 (and with no change in the SPI initialization of **SSPSTAT** and **SSPCON1** already carried out for the LCD).

b) Subject to the same caution as was raised in Part (b) of the last problem, add an ADT7301 to the SOIC surface-mount pattern on the front of the Qwik&Low board, being careful to avoid the use of any pins used by a part (such as the DAC for the last problem) on the back of the board.

c) Write a **TenthCent** function that, once a second, initiates a conversion. One loop time later it retrieves the converted result while simultaneously shutting the chip down. Then the function displays the temperature with a format exemplified by

```
27.3°C
```

right justified on the LCD. Assume that the temperature will never exceed 99.9°C.

d) Write a **TenthFahr** function analogous to the function of Part (c) that will display the Fahrenheit temperature with a format exemplified by

```
104.5°F
```

If the temperature is below 100.0°F, blank the hundreds digit.

Appendix

A1

QWIKBUG PROGRAM DEBUGGER

By Ryan Hutchinson

A1.1 INTRODUCTION

QwikBug is a small utility used to aid developers in the testing and debugging of C programs compiled for use on the Qwik&Low board. QwikBug was developed for the PIC18F4321 series of 8-bit PIC microcontrollers and makes full use of the PIC's built-in background debug mode (BDM).

A1.1.1 Architecture

There are two components to QwikBug: the QwikBug kernel, which is a program that resides in the PIC's program memory, and the QwikBug software interface, which provides the graphical user interface from a standard Windows-based PC.

The QwikBug kernel and software interface communicate using the computer's serial port, over a standard RS-232 connection. A custom protocol was developed to facilitate communications between the two components. This simple protocol defines many standard debugging commands, such as run, step, and add breakpoint. The protocol also defines commands that allow the QwikBug kernel to dynamically program the PIC's flash memory using the serial port. This allows the user to download a C program to the PIC without any additional hardware or software.

QwikBug uses the PGC (Program Clock) input pin on the microcontroller to initiate BDM (Background Debug Mode). This same pin is used by Microchip's ICD2 programmer/debugger to communicate to the PIC. This is accomplished by tying the RX input pin that is used by the PIC's UART to the PGC input pin. In this fashion, whenever data is received by the PIC on the UART, the PGC pin will trigger the microcontroller to enter BDM.

The QwikBug kernel makes use of the PIC's low-power features by placing itself into sleep mode when idle. Because of this, the software interface will "wake up" the chip when transmitting commands by first sending a "wake-up" byte that is ignored by the kernel and is only used to awaken the chip and return it to BDM.

Since the QwikBug kernel is no different from any other program that is written for a PIC microcontroller, it requires the use of various PIC resources to function. The goal behind the QwikBug program was to make it as non-intrusive as possible and to use the minimum amount of resources so that a user program could operate as if it were the only program running on the microcontroller. The QwikBug kernel occupies 1,536 bytes (768 words) of the PIC's memory. The kernel is located high in the PIC's program memory starting at address 0x1A00 and extends to the very end of its memory at address 0x1FFF. Since the PIC18LF4321 has a total of 8,192 bytes (4,096 words) of program memory available, this leaves 6,656 bytes (3,328 words) of program memory for the user's programs.

QwikBug also requires some RAM registers to operate. It utilizes a total of 56 bytes of RAM to perform its operations. As with program memory, these 56 bytes are located high in the PIC's address range to prevent collisions with user program variables. The RAM used by the QwikBug kernel begins at address 0x1C8 and extends through address 0x1FF. Since all RAM used by QwikBug is above 0x0FF, it makes exclusive use of bank 1 for all of its operations. As long as the user's programs do not write to registers in the range specified above, QwikBug can operate as intended. If the registers shared by QwikBug are written to, however, unexpected results may occur.

The third resource that is shared by QwikBug and by user programs is the UART. QwikBug uses both the transmit and receive functionality of the UART to communicate behind the scenes with the QwikBug software interface. QwikBug requires exclusive utilization of the UART's receive buffer, making it unavailable to the user except by moving a jumper on header H1, thereby assigning RX to an add-on user connection. The transmit buffer, however, can be shared between the two. The incoming user data is separated from communications-related data by the software interface and presented to the user via the QwikBug console.

To accomplish the seamless transition from BDM to user code, QwikBug shadows many of the PIC special function registers (SFRs). This means that many of the timer- and peripheral-related SFRs are saved to temporary locations before being modified by QwikBug. Once QwikBug returns from BDM, the registers are restored and operations can continue as if the user program was never interrupted. QwikBug also uses a clock frequency and serial baud rate that could be different from the user's. When BDM is first entered, QwikBug shadows the SFRs related to the clock speed and serial baud rate and uses its own, pre-defined values so that communications with the software interface is always possible.

A1.1.2 Features

The QwikBug software interface was designed to be lightweight and easy-to-use while still providing powerful debugging features. The following features are all part of the QwikBug package:

- Simple, free debugging interface
- Run, pause, and single-step through C source code
- Load and erase programs from a C18-compiled hex file using the PIC's UART
- Monitor watch variables in multiple display formats, variable types, and array sizes
- Modify watch variable values while program execution is suspended
- Soft reset of program to initial program vector
- Add a single breakpoint to stop execution at a specific C source line
- Quickly search through C source code by navigating to subroutines
- Display user data received on the PC's serial port using the built-in console tab

A1.2 INSTALLATION

A1.2.1 Prerequisites

The following prerequisites are required to run the QwikBug software interface:

- PC running Windows XP or Windows Vista
- Standard PC serial port via any COMi address, from COM1 through COM7

The following prerequisites are required to install the QwikBug kernel:

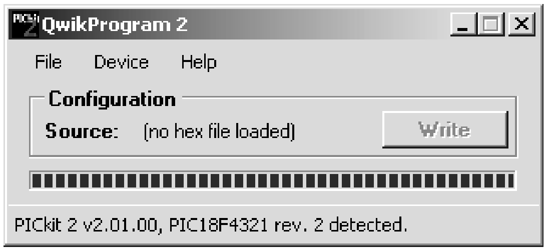
- Qwik&Low board
- Microchip PICkit 2 programmer
- QwikProgram 2 software installed (found online at www.qwikandlow.com)
- Standard PC USB port

Note that QwikBug could potentially be installed on a PIC18F4321 chip that was not part of the Qwik&Low board. However, QwikBug was designed to work with the Qwik&Low board and this guide will only cover installation for Qwik&Low boards.

A1.2.2 Installing QwikBug Kernel with QwikProgram 2

- Download the latest QwikBug kernel HEX file and HEX.VECTOR files from the webpage at www.qwikandlow.com. Place both the HEX file and the HEX. VECTOR file in the same directory.

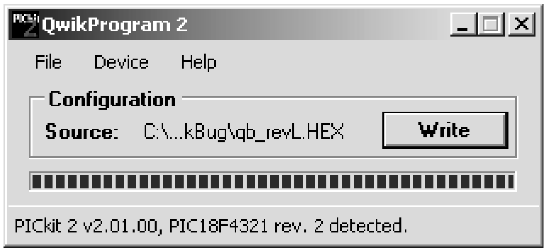
- Attach the PICkit 2 programmer to the computer's USB port.
- Attach the other end of the PICkit 2 to H5 on the Qwik&Low board. This header also has the label "MCU PICkit 2" and is specifically for the PIC18LF4321 on the Qwik&Low board. A 6-pin male-to-male header strip is required to connect the female end of the PICkit 2 to the female header on the Qwik&Low board. Ensure that the arrow on the board lines up with the arrow on the PICkit 2.
- Verify that power is turned off to the Qwik&Low board and run QwikProgram 2 on the computer. The following window should appear:



Source: (no hex file loaded)

PICkit 2 v2.01.00, PIC18F4321 rev. 2 detected.

- Verify that the PIC18F4321 was detected as labeled in the QwikProgram 2 status bar before proceeding. If the PIC was not detected, try disconnecting and reconnecting the PICkit 2 from the USB port and running Detect from QwikProgram 2's Device menu.
- Select File -> Open hex file. . . from the QwikProgram 2 menu. Locate the previously downloaded HEX file and click Open. The window should now look as shown:



Source: C:\...kBug\qb_revL.HEX

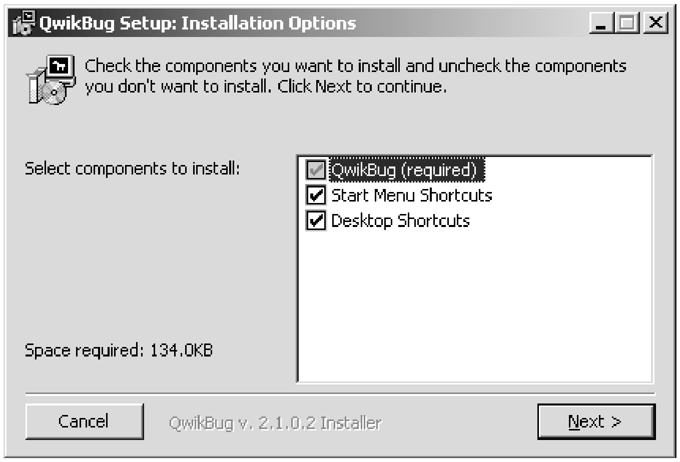
PICkit 2 v2.01.00, PIC18F4321 rev. 2 detected.

- Click the Write button once the HEX file has been loaded. This will write the QwikBug kernel to the proper vector location on the PIC. If the verification of the write did not succeed, reseat the PICkit 2 connection and try again.
- Now the QwikBug kernel has been installed and the QwikBug software interface can be used to communicate with the kernel residing on the PIC.

A1.2.3 Installing QwikBug Software Interface

- Download the QwikBug software interface install file from the website at www.qwikandlow.com.

- Launch the setup program. If QwikBug has already been installed on the computer then it will prompt to uninstall the old version before proceeding. Otherwise, click Next after selecting the desired options.

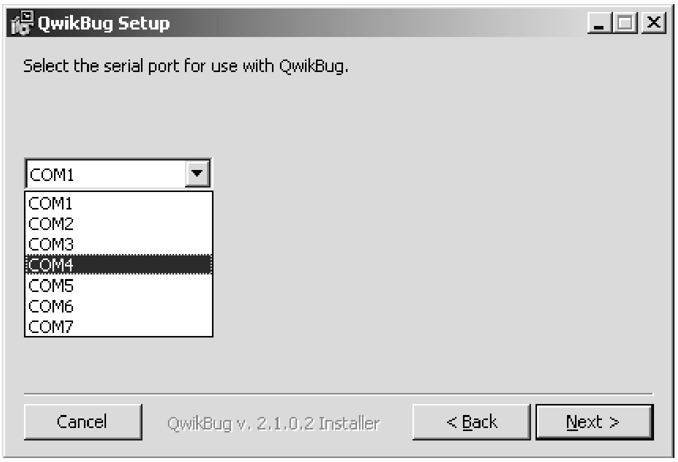

QwikBug Setup: Installation Options

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select components to install:
- [x] QwikBug (required)
- [x] Start Menu Shortcuts
- [x] Desktop Shortcuts

Space required: 134.0KB

- Select the COM port that will be used to communicate with the QwikBug kernel from the drop-down list. Click Next to continue installation.

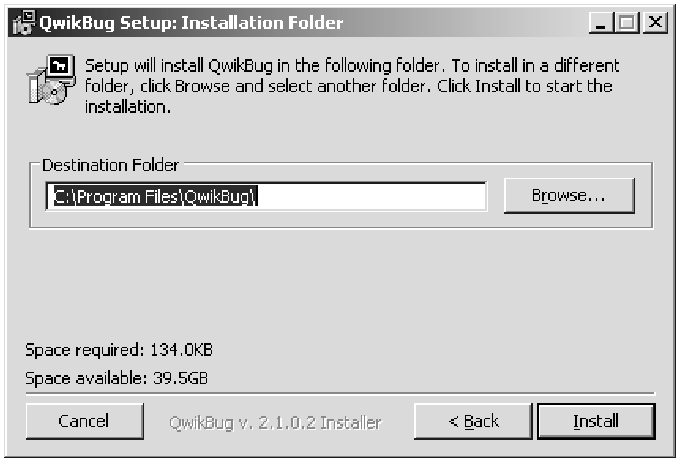

QwikBug Setup

Select the serial port for use with QwikBug.

COM1

COM1
COM2
COM3
COM4
COM5
COM6
COM7

- Select the installation folder for QwikBug. The default installation folder is recommended. Click Install to begin the installation of the software.
- After installation has successfully completed, click Close. The QwikBug software interface has now been installed.
- After installation, it is possible to tweak particular QwikBug settings. To do so, open the settings.xml file located in the QwikBug installation directory chosen above using a standard text editor such as Notepad. To change the setting values, modify the value attribute for the desired XML element. Caution; modification

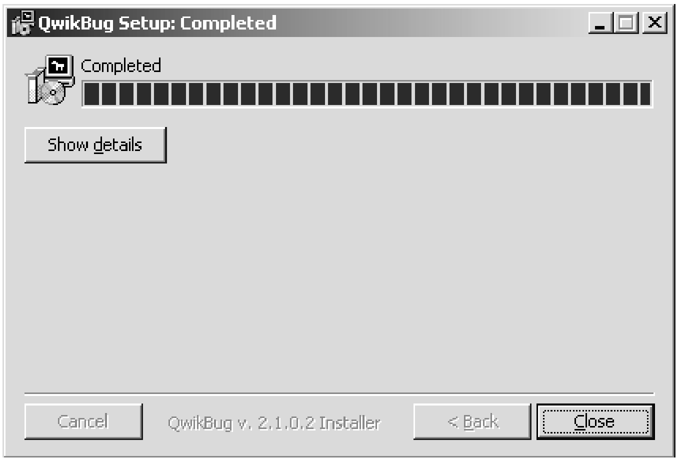

QwikBug Setup: Installation Folder

Setup will install QwikBug in the following folder. To install in a different folder, click Browse and select another folder. Click Install to start the installation.

Destination Folder: C:\Program Files\QwikBug\

Space required: 134.0KB
Space available: 39.5GB

QwikBug v. 2.1.0.2 Installer


QwikBug Setup: Completed

Completed

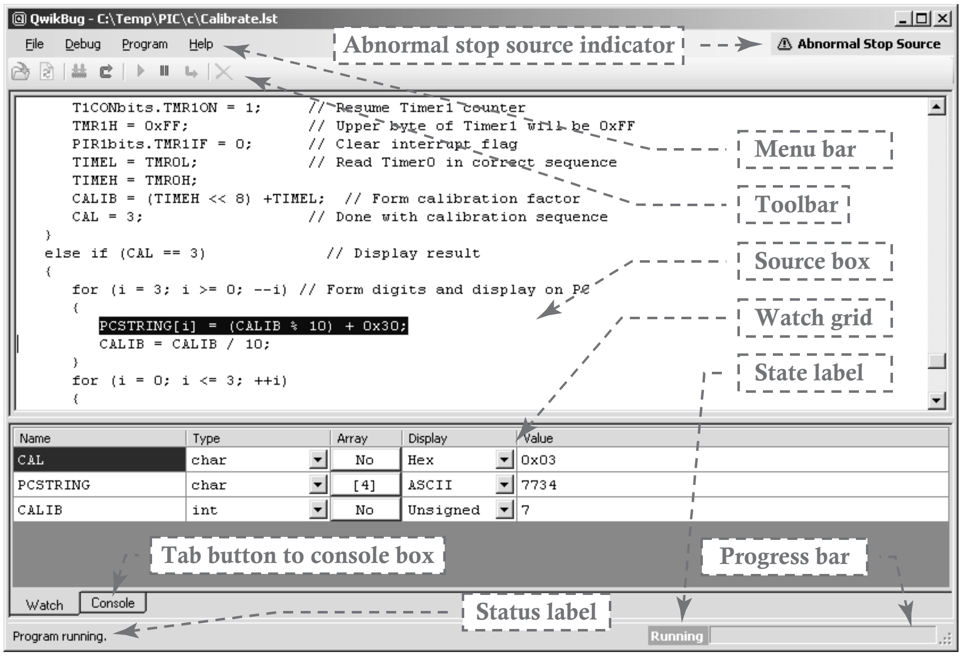
QwikBug v. 2.1.0.2 Installer

of the name or type attributes or other aspects of the file could corrupt the XML file and cause QwikBug to malfunction. The table below outlines the settings that can be tweaked:

Setting Name	Default Value	Description
COMPort	(chosen during setup)	The COM port that QwikBug uses to communicate with the kernel. This value was created during setup and can be changed if needed.
BaudRate	19,200 baud	The serial port baud rate used to communicate with the kernel. This setting should generally not be changed unless the QwikBug kernel is modified to also use the same baud rate.
SerialTimeout	300 ms	The amount of time in milliseconds to wait before a timeout error occurs when issuing a command to the QwikBug kernel.

A1.3 GRAPHICAL USER INTERFACE

The following figure outlines the various components of the QwikBug user interface. Those components that will be referred to later are called out to provide a reference.



```
      T1CONbits.TMR1ON = 1;      // Resume Timer1 counter
      TMR1H = 0xFF;              // Upper byte of Timer1 will be 0xFF
      PIR1bits.TMR1IF = 0;       // Clear interrupt flag
      TIMEL = TMR0L;             // Read Timer0 in correct sequence
      TIMEH = TMR0H;
      CALIB = (TIMEH << 8) +TIMEL;  // Form calibration factor
      CAL = 3;                   // Done with calibration sequence
   }
   else if (CAL == 3)              // Display result
   {
      for (i = 3; i >= 0; --i) // Form digits and display on PC
      {
         PCSTRING[i] = (CALIB % 10) + 0x30;
         CALIB = CALIB / 10;
      }
      for (i = 0; i <= 3; ++i)
      {
```

Name	Type	Array	Display	Value
CAL	char	No	Hex	0x03
PCSTRING	char	[4]	ASCII	7734
CALIB	int	No	Unsigned	7

Interaction with the QwikBug user interface is identical to that of most commonly used Windows programs. Several of the popup windows that are part of the QwikBug program are set as topmost windows and will always remain on top of other windows while they are open. This is to help the user to quickly relocate the popup windows when needed.

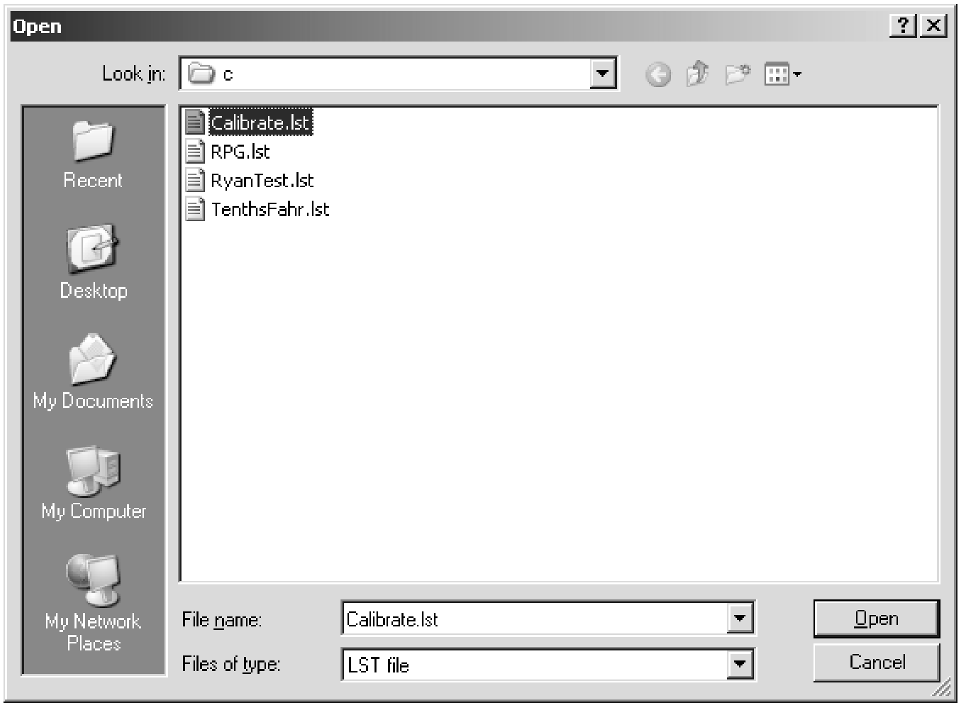
A1.4 LOADING A PROGRAM

A1.4.1 Opening a List File

QwikBug gathers all of the information it needs about the user program from the LST and COD files that are generated by the Microchip C18 compiler. It also requires the HEX file that is generated by the compiler to download the program to the PIC. It is recommended that the user downloads and uses the C18 compilation utility that is freely available on the website at www.qwikandlow.com. The C18 utility utilizes the C18 compiler and passes in compiler arguments needed to compile C programs for the PIC18LF4321. It also provides the user with program memory utilization and factors in the program memory occupied by QwikBug when doing so.

To open a list file, select Open File from the File menu or click the Open File toolbar button. A dialog box will appear prompting the user for the desired LST file. Locate the compiled LST file from the dialog and click Open. Note that the associated

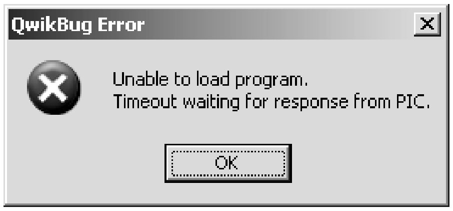
HEX and COD files must be in the same directory as the LST file and have the same beginning file name or QwikBug will produce an error.



File name: Calibrate.lst
Files of type: LST file

Once the list file has been processed, the source code for the C program will appear in the source box. The progress bar will show the progress of the file as it is being opened. Note that the state label shows "Disconnected", indicating that the user program has not yet been loaded into the PIC.

Ensure that the Qwik&Low board is powered on and that the serial cable is properly connected from the PC to the Qwik&Low board's female DB-9 serial port connector. To load the program into the PIC, select Load from the Program menu or click the Load toolbar button. The progress bar should begin to increase as the program is downloaded over the serial port. If there was an error downloading, the following message box will appear:



Unable to load program.
Timeout waiting for response from PIC.

This error message indicates that the software interface was unable to communicate with the QwikBug kernel. If this error occurs, check the power to the PIC and the serial port connections. Also verify that the QwikBug kernel has been properly installed as described in the previous section.

Upon successful load of the program, the state label will change to "Paused" and the status label should read "Program paused [Reset executed]". Note that the Reset, Run ▶, and Step ↳ toolbar buttons are all now enabled. Refer to the associated sections to learn about the use of all of QwikBug's debugging features.

A1.4.2 Erasing the Program Memory

QwikBug allows the user to erase the program memory of the PIC (up to the QwikBug kernel's vector location). To execute an erase, select Erase from the Program menu or click the Erase ✕ toolbar button. Erase is only allowed when the state is either disconnected or paused. If the erase is successful, the state will change to disconnected and the status label will read "Program memory erased".

A1.4.3 Monitoring of List File

QwikBug will constantly monitor the list file once it has been opened. If the list file changes for any reason (e.g., the source file is changed and recompiled), QwikBug will disconnect, clear out the source box, and the status label will change to read "List file has been changed, please reopen file". This is done to ensure that the user is kept up-to-date when the source file has been recompiled and that an old copy of the program is not accidentally used with QwikBug. To reopen a list file at any time after the file has initially been opened, select Reopen File from the File menu or click the Reopen File toolbar button.

A1.5 PROGRAM CONTROL

A1.5.1 Running

While the PIC is in the running state, it is executing code without any intervention by QwikBug. To place the PIC in run mode after a program has been loaded into program memory, select Run from the Debug menu or click the Run ▶ toolbar button. The PIC cannot be placed into run mode unless it is currently paused. The state label will change to "Running" while in run mode and will be highlighted green.

Running

Once in run mode, watch variable values will not be updated until the PIC reaches a paused state. If the PIC reaches a breakpoint while running it will also enter the paused state. The PIC can also be reset while in the running state.

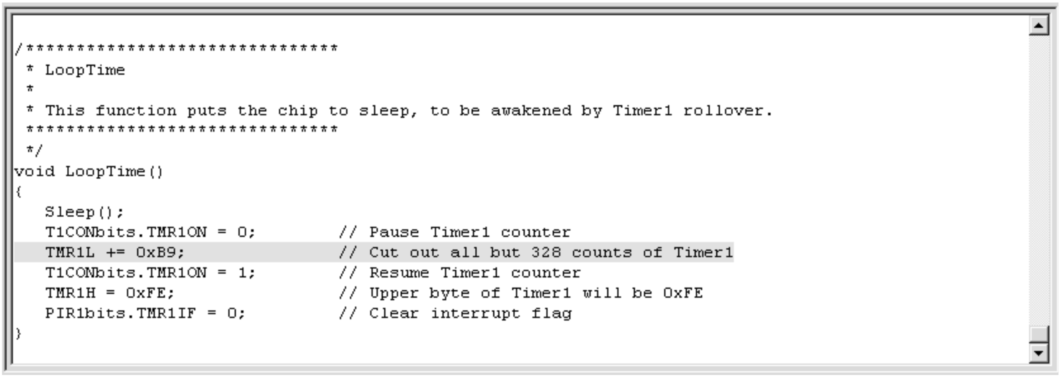
A1.5.2 Pausing

No user program code is executed by the PIC while in the paused state. To place the PIC in pause mode after a program has been loaded into program memory, select Pause from the Debug menu or click the Pause ‖ toolbar button. The PIC can be paused while it is in run or step mode. Paused is also the default state of the PIC after coming out of reset or performing a program download. The state label will change to "Paused" while in paused mode and will be highlighted yellow.

Paused

The values of the watch variables will be updated as soon as the PIC is placed in the paused state. While in pause mode, breakpoints can be added or removed and watch variables can be added. It is also possible to reload the program into the PIC while in pause mode or to erase the program memory. From pause mode, the user can place the PIC in run mode or step mode.

While the PIC is in the paused state, a yellow highlight will appear in the source box to let the user know what code is *about to be executed*. It is important to remember that the line that is highlighted has not yet been executed or has been partially executed, as is the case for C source lines that contain multiple instructions should the PIC be paused while in the middle of executing the line. The figure below shows an example of a highlighted C source line just before execution:



```
/*******************************
 * LoopTime
 *
 * This function puts the chip to sleep, to be awakened by Timer1 rollover.
 *******************************
 */
void LoopTime()
{
   Sleep();
   T1CONbits.TMR1ON = 0;        // Pause Timer1 counter
   TMR1L += 0xB9;               // Cut out all but 328 counts of Timer1
   T1CONbits.TMR1ON = 1;        // Resume Timer1 counter
   TMR1H = 0xFE;                // Upper byte of Timer1 will be 0xFE
   PIR1bits.TMR1IF = 0;         // Clear interrupt flag
}
```

A1.5.3 Stepping

While the PIC is stepping, code is being executed one instruction at a time. The PIC will continue to step until its program counter reaches a value that is outside the range of the C source code line. There are generally multiple instructions that are executed for each line of C source code. To place the PIC in step mode after a program has been loaded into program memory, select Step from the Debug menu or click the Step ↳ toolbar button. The PIC can be stepped while it is in pause mode only. The state label will change to "Stepping" while in step mode and will be highlighted orange.

Stepping

If a C source line contains loops that cause the line to continually execute, it could take significant time before the PIC steps out of one C source code line. Because of this, QwikBug allows the user to discontinue step mode by selecting Pause from the

Debug menu or clicking the Pause ⏸ toolbar button. This will cause the PIC to suspend single-step execution and the state will change back to paused.

A1.5.4 Resetting

QwikBug provides a soft reset feature. To issue a soft reset to the PIC, select Reset from the Program menu or click the Reset toolbar button. This will cause the program counter to reset to the default initialization vector of 0x0000. Additionally, all PIC special function registers (SFRs) and RAM values will be reset to their initial states. A reset can be executed while the PIC is in either run or paused mode. It cannot be executed while the PIC is stepping. QwikBug will return the PIC to the paused state at the initialization vector after a reset.

A1.6 BREAKPOINTS

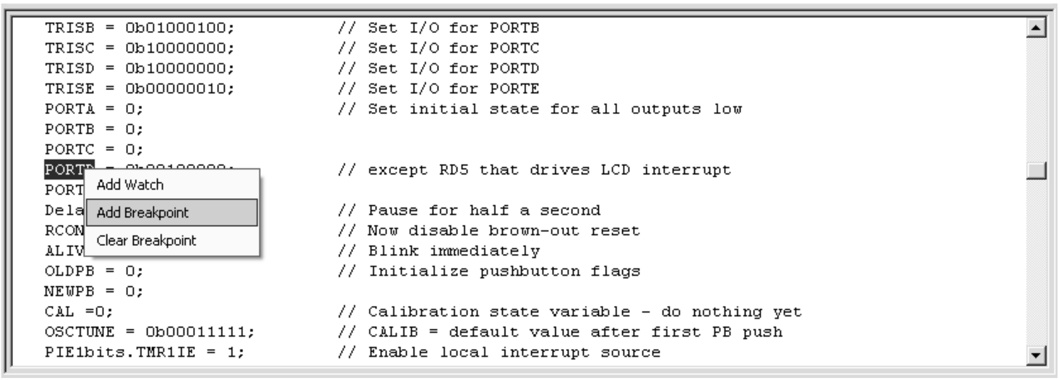
QwikBug provides the user with a single breakpoint that can be placed on any line of C source code that contains PIC instructions. Breakpoints cause an executing PIC program to halt when the program counter reaches a specified value.

A1.6.1 Breakpoint Execution Order

Breakpoint execution order is different for QwikBug than for many other debugging programs that the user may be used to. When a breakpoint is set in QwikBug, the breakpoint is actually put at the last instruction contained in the C source code line. This is because the PIC executes the line of code that the breakpoint is placed on and then increments the program counter before entering BDM. When a breakpoint is placed on a C source line is QwikBug, the PIC will execute the very last instruction contained in the line and then pause.

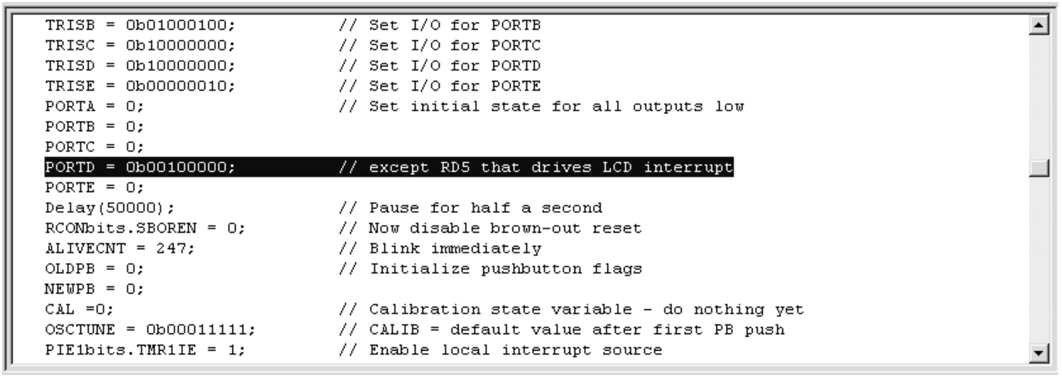
A1.6.2 Adding a Breakpoint

To add a breakpoint in QwikBug, the program must first be loaded onto the PIC and the PIC must be in a paused state. Once paused, simply right-click on a line of source code in the source box and select Add Breakpoint as shown below:



```
TRISB = 0b01000100;          // Set I/O for PORTB
TRISC = 0b10000000;          // Set I/O for PORTC
TRISD = 0b10000000;          // Set I/O for PORTD
TRISE = 0b00000010;          // Set I/O for PORTE
PORTA = 0;                   // Set initial state for all outputs low
PORTB = 0;
PORTC = 0;
PORTD                        // except RD5 that drives LCD interrupt
PORT
Dela                         // Pause for half a second
RCON                         // Now disable brown-out reset
ALIV                         // Blink immediately
OLDPB = 0;                   // Initialize pushbutton flags
NEWPB = 0;
CAL =0;                      // Calibration state variable - do nothing yet
OSCTUNE = 0b00011111;        // CALIB = default value after first PB push
PIE1bits.TMR1IE = 1;         // Enable local interrupt source
```

A line that has a breakpoint associated with it will be highlighted red as shown below:



```
TRISB = 0b01000100;          // Set I/O for PORTB
TRISC = 0b10000000;          // Set I/O for PORTC
TRISD = 0b10000000;          // Set I/O for PORTD
TRISE = 0b00000010;          // Set I/O for PORTE
PORTA = 0;                   // Set initial state for all outputs low
PORTB = 0;
PORTC = 0;
PORTD = 0b00100000;          // except RD5 that drives LCD interrupt
PORTE = 0;
Delay(50000);                // Pause for half a second
RCONbits.SBOREN = 0;         // Now disable brown-out reset
ALIVECNT = 247;              // Blink immediately
OLDPB = 0;                   // Initialize pushbutton flags
NEWPB = 0;
CAL =0;                      // Calibration state variable - do nothing yet
OSCTUNE = 0b00011111;        // CALIB = default value after first PB push
PIE1bits.TMR1IE = 1;         // Enable local interrupt source
```

Note that only one breakpoint is available with QwikBug. If another breakpoint already exists when the Add Breakpoint selection is clicked, the previous breakpoint will be automatically cleared and the newly selected breakpoint line will be highlighted red. Also note that source lines that do not have any PIC instructions associated with them (e.g., comment lines) cannot have breakpoints added to them. If a line that is not associated with any instructions is selected, the Add Breakpoint option will be grayed-out and disabled from the right-click menu.

A1.6.3 Clearing a Breakpoint

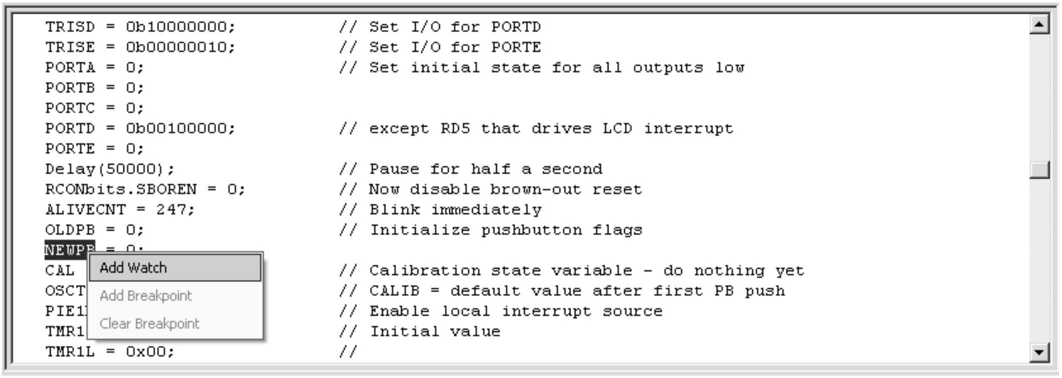
To clear the current breakpoint either select Clear Breakpoint from the Debug menu or select Clear Breakpoint from the menu that appears when a C source line is right-clicked. Breakpoints can only be cleared when the PIC is in the paused state. When a breakpoint is cleared, the red highlight will also disappear from the breakpoint line in the source box.

A1.7 WATCH VARIABLES

Watch variables are pointers into the values stored in the PIC's RAM and Special Function Registers. The variables are defined in the C source file. QwikBug uses the compiler-generated COD file to locate address information for the variable symbol names. Once a watch variable has been added to the watch grid in QwikBug, the user can view the actual value of the register prior to entering BDM. The user also has the ability to modify the value of the watch variable and display the variable in multiple formats, including decimal, hexadecimal, binary, and ASCII. QwikBug allows the user to manually choose the variable type for a specified watch variable (e.g., char, int, short long, long) as well as the number of elements if the variable is an array. In addition, QwikBug allows the user to watch PIC SFRs and modify their values just like a normal, user-defined C variable.

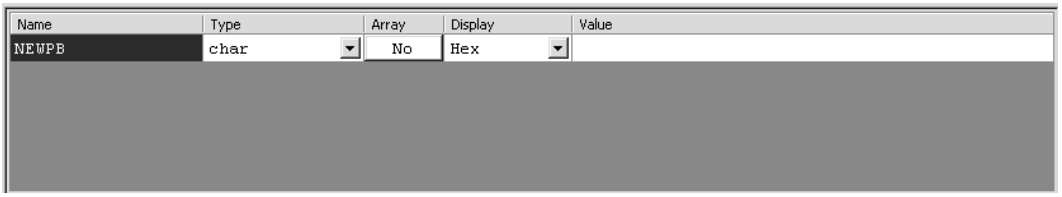
A1.7.1 Adding Watch Variables to the Watch Grid

Watch variables can be added while the PIC is in any state. The values, however, will only be updated while the PIC is in its paused state. To add a watch variable after opening a list file, right-click on the variable anywhere it is used in the C source box. QwikBug will automatically highlight the entire word that was clicked on in the source box. If the variable is found in QwikBug's symbol table the Add Watch option will be selectable. Note that QwikBug does not support watching local function variables. If a user-defined variable is intended to be watched in QwikBug, it must be a global C variable. The figure below shows an example of a highlighted variable in the source box as well as the Add Watch selection:



```
TRISD = 0b10000000;          // Set I/O for PORTD
TRISE = 0b00000010;          // Set I/O for PORTE
PORTA = 0;                   // Set initial state for all outputs low
PORTB = 0;
PORTC = 0;
PORTD = 0b00100000;          // except RD5 that drives LCD interrupt
PORTE = 0;
Delay(50000);                // Pause for half a second
RCONbits.SBOREN = 0;         // Now disable brown-out reset
ALIVECNT = 247;              // Blink immediately
OLDPB = 0;                   // Initialize pushbutton flags
NEWPB = 0;
CAL                          // Calibration state variable - do nothing yet
OSCT                         // CALIB = default value after first PB push
PIE1                         // Enable local interrupt source
TMR1                         // Initial value
TMR1L = 0x00;                //
```

Once Add Watch is selected, the variable will be added to the watch grid. The variable will be added with the default variable type and display type, and will be initialized with an array size of one element, meaning that the variable is not an array. Only one copy of the variable can exist at a time in the watch grid, and QwikBug will ignore requests to add the same watch variable multiple times. The figure below shows the watch grid after the example variable has been added for watching:

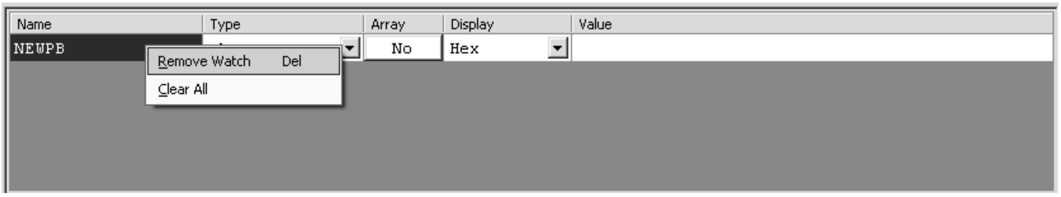


Name	Type	Array	Display	Value
NEWPB	char	No	Hex	

Now that the variable exists in the watch grid the value will be updated whenever the PIC enters the paused state. The Type, Array, and Display columns in the QwikBug watch grid will need to be manually updated for the new watch variable since they will all be set to the default values. Refer to the section on variable and display types below for more information on these columns.

A1.7.2 Removing Watch Variables from the Watch Grid

To remove a watch variable, simply right-click anywhere on the variable's row in the watch grid and select Remove Watch as shown in the figure below:

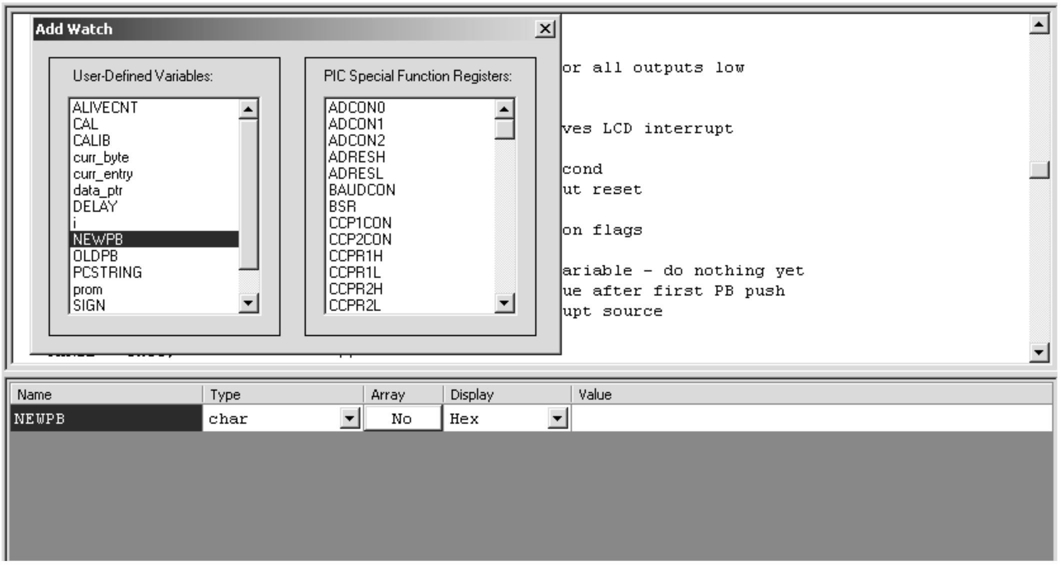


Name	Type	Array	Display	Value
NEWPB		No	Hex	

If multiple rows were selected, all of the selected watch variables will be removed from the watch grid. Multiple rows can be selected at a time by clicking the leftmost column and dragging the mouse to select the desired rows. It is also possible to use the keyboard's Del key to quickly delete selected rows from the watch grid. To clear all of the watch variables that are shown in the watch grid, select the Clear All option after right-clicking on the watch grid.

A1.7.3 Add Watch Variable Popup

The add watch variable popup is a way to alphabetically view all possible variables that can be watched and can be used as an alternate method of adding and removing watch variables from the watch grid. The add watch variable popup window can be opened by selecting Add Watch from the Debug menu and is always the topmost window. The figure below shows a typical example of the add watch variable popup in QwikBug:



```
or all outputs low

ves LCD interrupt

cond
ut reset

on flags

ariable - do nothing yet
ue after first PB push
upt source
```

Name	Type	Array	Display	Value
NEWPB	char	No	Hex	

The add watch variable popup is divided into two list boxes: user-defined variables and PIC special function registers. The user-defined variables list box contains all globally-defined C variables that are user defined in the source program. The PIC special function registers list box contains all of the defined SFRs that can be read from or written to on the PIC18LF4321.

The variables that are currently being watched are highlighted in the appropriate list box, as shown in the example above. To add or remove a watch variable simply click on the variable name in either list box. If the variable is not being watched, it will become highlighted and will be added to the watch grid. Conversely, if the variable is currently being watched, the highlight will be removed from the list box and the variable will be removed from the watch grid.

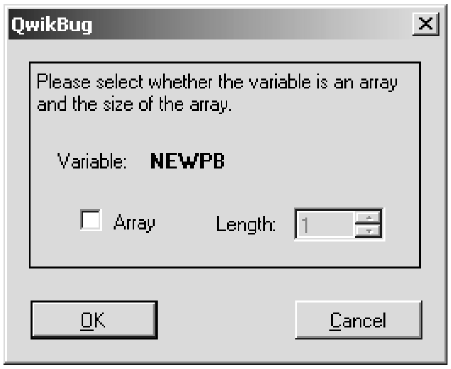
A1.7.4 Watch Variable Types and Display Types

The Type column in the watch grid represents the declared type of the C variable in the source program. The table below lists all of the possible variable types and their associated size:

Type	Size (of each element)
char	1 byte
int	2 bytes
short long	3 bytes
long	4 bytes

Note that the variable type applies to each element of an array. Structures and other multi-type variables are not supported by QwikBug. The variable type for PIC SFRs cannot be modified and will be highlighted gray because all SFRs are single-byte registers and thus have an inherent type of char.

The Array column in the watch grid displays the number of elements contained in the associated watch variable. If the variable consists of only one element, the Array column will display No, implying that the watch variable is not an array. To change the number of elements in a watch variable, click on the cell button in the column. The following popup will appear:



Please select whether the variable is an array and the size of the array.

Variable: **NEWPB**

☐ Array Length: 1

To make the variable an array of elements, check the Array box on the popup. Then, select the total length of the array and click OK to update the watch variable. Note that the Array column will be disabled and grayed-out for SFRs because they can only be single-byte registers.

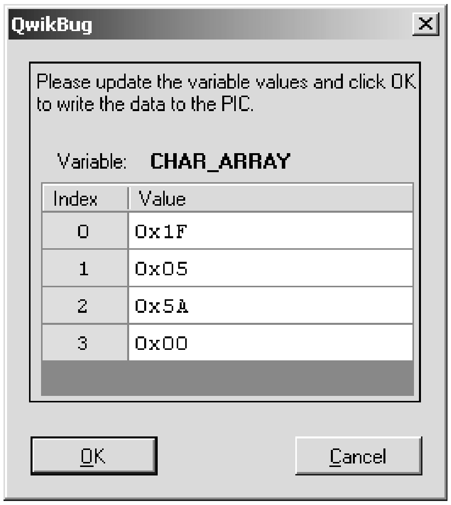
The Display column determines how the watch variable's value in the Value column should be displayed to the user. The table below lists all of the possible display type options as well as an example display of each for a single-element char:

Display	Example
Unsigned	165
Signed	–91
Hex	0xA5
Binary	10100101
ASCII	G

For variables that have been defined as arrays, the displayed value will be comma-separated with each element following the display formatting shown above. For ASCII display types, only the least significant byte is used to determine the ASCII character. For variables types larger than a char this means that all of the upper bytes are ignored when displaying the value as ASCII.

A1.7.5 Writing to Watch Variables

QwikBug allows the user to modify the values of variables that are being watched. Watch variable values can only be modified when the PIC is in the paused state. To modify the value, double-click on the cell in the Value column for the associated watch variable in the watch grid. This action will open a popup that can be used to modify the variable's value(s), as shown below:



Please update the variable values and click OK to write the data to the PIC.

Variable: CHAR_ARRAY

Index	Value
0	0x1F
1	0x05
2	0x5A
3	0x00

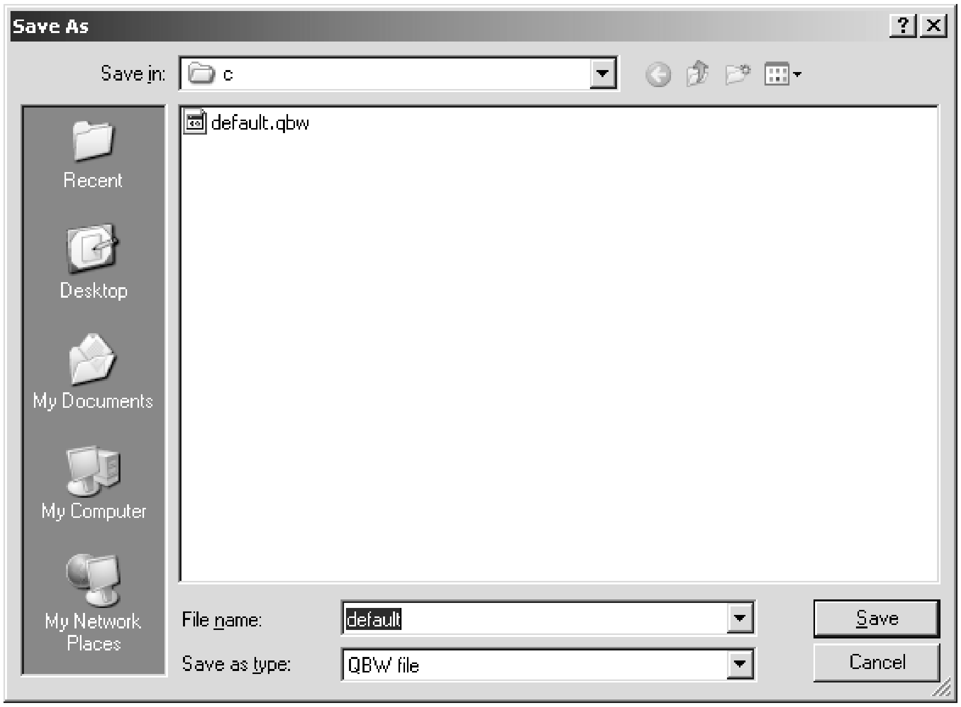
For variables that are configured as arrays, a row will be present for each element in the array and the associated index number is shown in the column to the left of an element's value. For ASCII display types only one row will be shown and the value is treated as a contiguous string. Click on each of the variable's elements that are to be modified and type in a new value for the element. Be sure to use the same display format when overwriting an element's value. QwikBug will verify all values before they are written to the PIC. If the value entered cannot be interpreted for the chosen display type or if the value is outside the range of the variable type, the old value will remain in the element's row.

Once all element values have been modified to the desired new values, click the OK button to have QwikBug update the PIC with the new values. The Value column in the watch grid will then be updated with the variable's new value(s) once they have been successfully read back from the PIC.

To discard any modifications simply click the Cancel button on the popup window and the variable will not be written to.

A1.7.6 Import/Export of Watch Variables

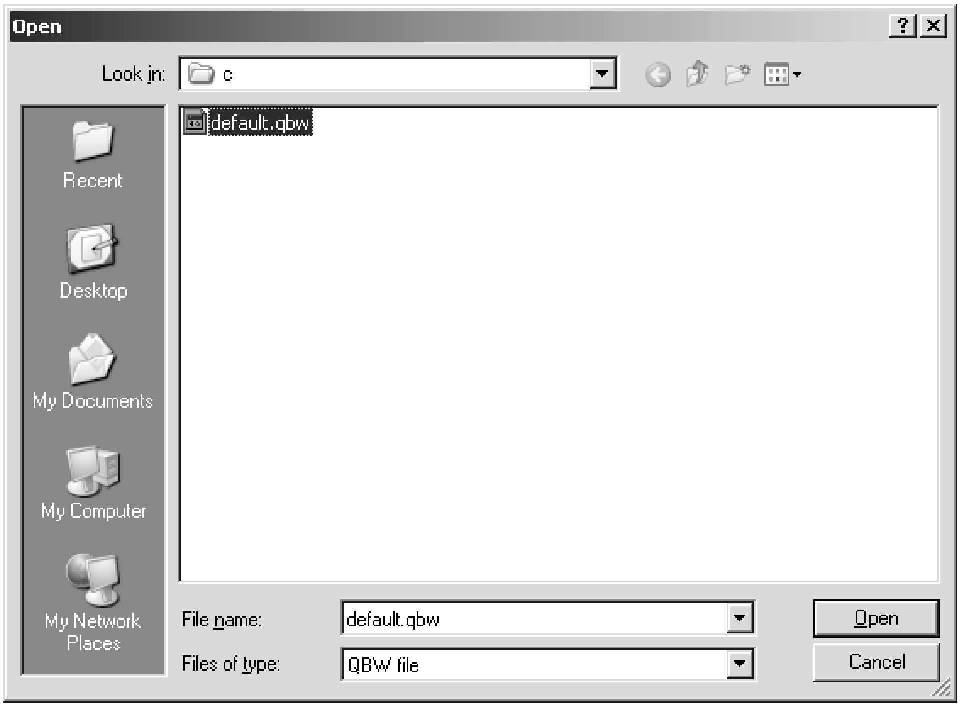
QwikBug contains a feature that allows the user to import and export watch variable configurations to the hard drive to expedite watch variable creation at different points in time. To export the current watch variable configuration once a list file has been opened and the watch grid has been configured as desired, select Export Watch Variables from the File menu. A dialog will appear prompting the user for a destination file to save to:



default.qbw
File name: default
Save as type: QBW file

Select the file to save the watch variable configuration to and click the Save button. The name of the file is insignificant to QwikBug and is only used to help the user keep track of multiple watch variable export files.

Once a configuration has been exported, the file can be imported by selecting Import Watch Variables from the File menu. The following dialog will appear:



File name: default.qbw
Files of type: QBW file

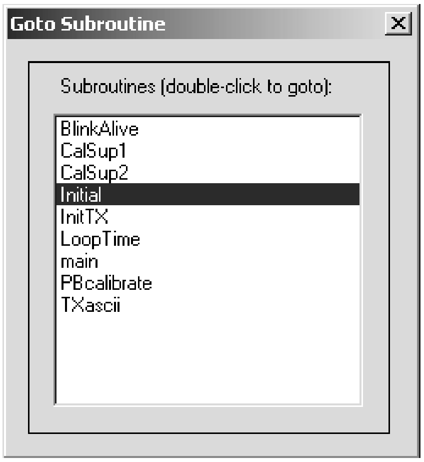
Select the file that contains the configuration that should be imported and click the Open button to begin the import process. Watch variables can only be imported after a list file has been opened. QwikBug will clear all current watch variables out of the watch grid once an import is initiated and will add the variables from the exported file to the watch grid.

QwikBug does not associate watch variable export files with specific list files. Therefore, it is possible to import any watch variable configuration with any list file. QwikBug will check each variable during an import to ensure that it exists in the current source program. QwikBug will not add the variable to the watch grid if the variable name does not exist or if there is a mismatch between the addresses of the exported variable and the current program's variable.

A1.8 ADDITIONAL FEATURES

A1.8.1 Find Subroutine Popup

The find subroutine popup allows the user to quickly locate a C subroutine in the source box. To open the find subroutine popup window, select Find Subroutine from the Debug menu; it will be opened as the topmost window. The popup window will display a list of all subroutine symbols that QwikBug has located in the source file:

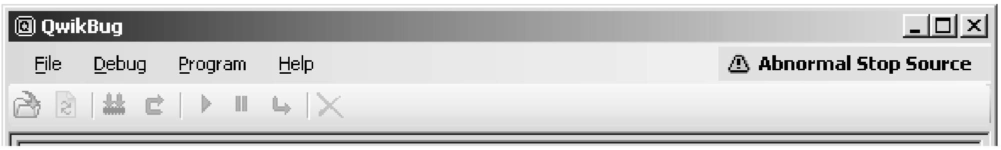


BlinkAlive
CalSup1
CalSup2
Initial
InitTX
LoopTime
main
PBcalibrate
TXascii

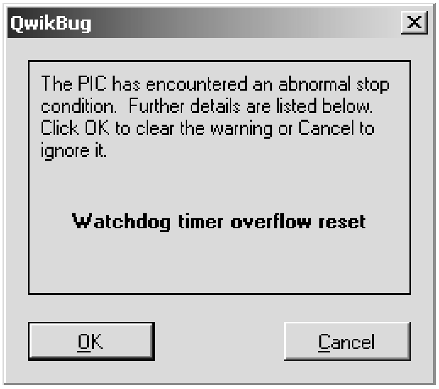
Double-clicking on the name of the subroutine in the list box will scroll the source box to the location of the first C source line in the selected subroutine and highlight the line.

A1.8.2 Abnormal Stop Sources

An abnormal stop occurs any time the PIC enters BDM for an irregular reason. Potential reasons include: brown-out reset, stack over/underflow, power-on reset, and watchdog timer overflow. If an abnormal stop occurs after a program has been loaded, the following icon will appear in QwikBug:



Click on the abnormal stop source indicator to display a popup window with details on the abnormal stop source. An example is shown below:



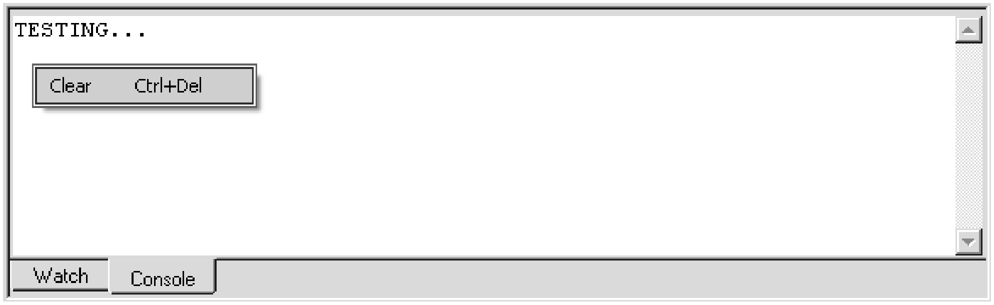
The PIC has encountered an abnormal stop condition. Further details are listed below. Click OK to clear the warning or Cancel to ignore it.

Watchdog timer overflow reset

Click the OK button to close the popup window and clear the abnormal stop source indicator warning. Click Cancel to close the popup window but keep the abnormal stop source indicator shown on the main QwikBug window. When an abnormal stop occurs, the PIC will generally be in a reset condition and the user can resume execution from the reset vector or correct the reason for the abnormal stop source in the C source code.

A1.8.3 Console Tab

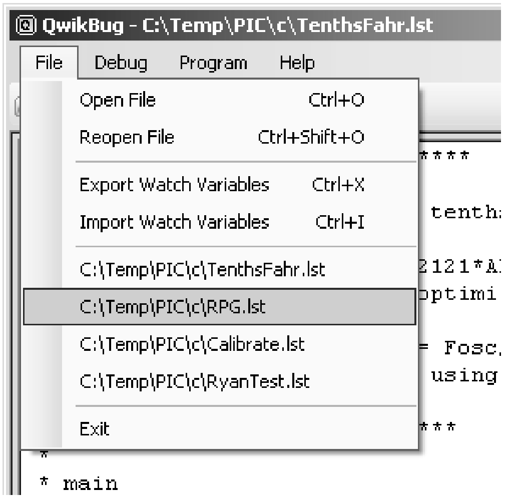
The console text box displays data received by the serial port that is not part of the QwikBug interface communications. The console tab will flash when new data is received and the console tab is not in focus. New data is appended to the end of the text box, and the box will scroll to always display the most recent data at the bottom of the console box. To clear the console box, right-click anywhere in the text box and select Clear, as shown below:



```
TESTING...
```

A1.8.4 Recently Opened Files

QwikBug displays a list of the four most recently opened files. The names of the files can be found under the File menu as shown below:



QwikBug - C:\Temp\PIC\c\TenthsFahr.lst
File
Debug
Program
Help
Open File Ctrl+O
Reopen File Ctrl+Shift+O
Export Watch Variables Ctrl+X
Import Watch Variables Ctrl+I
C:\Temp\PIC\c\TenthsFahr.lst
C:\Temp\PIC\c\RPG.lst
C:\Temp\PIC\c\Calibrate.lst
C:\Temp\PIC\c\RyanTest.lst
Exit

To open any of the previous files, simply click on its name in the File menu. The most recent file is positioned at the top of the list and the oldest is positioned at the bottom.

A1.9 KEYBOARD SHORTCUTS

The table below shows all defined QwikBug keyboard shortcuts:

Function	Keyboard Shortcut	Alternate Keyboard Shortcut
Open File	Ctrl+O	none
Reload File	Ctrl+Shift+O	none
Show Add Watch Popup	Ctrl+W	none
Show Find Subroutine Popup	Ctrl+F	none
Clear Breakpoint	Ctrl+Shift+B	none
Clear Console	Ctrl+Del	none
Export Watch Variables	Ctrl+X	none
Import Watch Variables	Ctrl+I	none
Usage Guide	F1	none
Reset PIC	F2	Ctrl+T
Load Program	F3	Ctrl+L
Pause	F5	Ctrl+P
Run	F7	Ctrl+R
Step	F8	Ctrl+S
Erase Program	F12	Ctrl+E

Both the keyboard shortcut and the alternate keyboard shortcut provide the same functionality. Note that some of the keyboard shortcuts that use Ctrl will not work with the cursor positioned in the main source code text box. Click outside the text box to change the focus so that these keyboard shortcuts will work.

Appendix

A2

INTERPRETING .LST FILES

A2.1 OVERVIEW

This appendix will describe the PIC18LF4321 CPU structure, program memory, RAM and Special Function Registers, addressing options, and instruction set. The intent is not to arm the reader to write assembly code for the MCU. Rather, it is to help with the interpretation of the list file generated by the C18 compiler. When the execution of a user program does not match the expectations of the code writer, the resulting .lst file translates the code writer's expectations into the reality of what is actually being executed by the MCU

Cody Planteen has written a qwiklst.exe utility used by Alex Singh's C18.exe utility to translate the *.lst file produced by the compiler into a *qwik.lst* file that is easier to interpret. It substitutes the name of a variable or of a Special Function Register in place of the normal .lst file's cryptic display of the address of the variable or register. For instructions that return an operand to either of two locations, it substitutes an **F** or a **W** for the more cryptic 0x1 or 0x0 (see Section A2.4).

A2.2 HARVARD ARCHITECTURE

The PIC18LF4321 CPU is organized around two buses in what is known as the Harvard architecture, shown in Figure A2-1. One bus reaches out to program memory

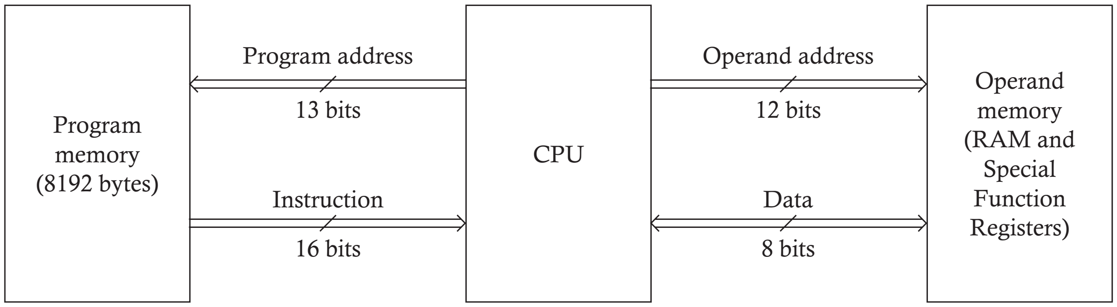

Program memory (8192 bytes)
Program address
13 bits
Instruction
16 bits
CPU
Operand address
12 bits
Data
8 bits
Operand memory (RAM and Special Function Registers)

FIGURE A2-1 Harvard architecture

and fetches one of the CPU's many 2-byte instructions during each CPU clock cycle. These fetches of 1-cycle instructions are occasionally augmented by the fetches of 2-cycle, 4-byte instructions.

At the same time that the CPU is fetching one instruction, it is executing the instruction fetched during the previous cycle. The Harvard architecture has a twofold impact on the performance of the CPU relative to the performance of a single-bus 8-bit microcontroller:

- The wide 16-bit instruction bus can fetch instructions with half the fetch cycles and thus twice the speed for a given clock rate.
- The time required to execute instructions is largely hidden beneath the ongoing fetching of successive instructions rather than requiring additional cycles.

The effect of the pipelined operation can be explained via the four-line qwik.lst example of Figure A2-2 derived from the T3.lst file. The left column lists the first of the two addresses (i.e., 0116 and 0117) holding an instruction. The next column lists the 2-byte *opcode* (i.e., 0E2C) that identifies the instruction to the CPU. The next two columns list the assembly language instruction(s) that the compiler has selected to implement the source file line shown to the right. That is,

```
MOVLW  0x2C
MOVWF  ALIVECNTL
MOVLW  0x01
MOVWF  ALIVECNTH
```

implements the source file line

```
ALIVECNT = 300;  //Blink immediately
```

The CPU fetches each 2-byte instruction during its 1-microsecond fetch cycle. Furthermore, as the **MOVWF** instruction is being fetched, the preceding **MOVLW** instruction is being executed.

```
0116   0E2C       MOVLW      0x2C          ALIVECNT = 300;     // Blink immediately
0118   6E05       MOVWF      ALIVECNTL
011A   0E01       MOVLW      0x01
011C   6E06       MOVWF      ALIVECNTH
```

FIGURE A2-2 A snippet from qwik.lst

A2.3 INSTRUCTION SET AND DIRECT ADDRESSING

The PIC18 family of microcontrollers has the instruction set shown in Figure A2-3. To understand the role of an instruction beyond the somewhat cryptic *Description* expressed in the table, it is necessary to understand the role of the operands associated with an instruction. The operand, *f*, associated with many instructions is an 8-bit field in the 16-bit instruction that accounts for 8 bits of the 12-bit *direct address* of the operand. In addition to the 8-bit address field of an instruction, another bit ("a" in Figure A2-4a) indicates how the remaining bits of an operand address are to be formed. For programs making use of less than 128 bytes of RAM, the default case for the C18.exe utility, the compiler uses the access-bank direct addressing of Figure A2-4b. The 8-bit address field in an instruction can access any of 128 RAM bytes using addresses ranging from 0 to 127 (i.e., 0x00 to 0x7F). In effect, the microcontroller behaves as if it only has 128 bytes of RAM, not the 512 actually available.

If a program were to use more than 128 named RAM addresses, some of these addresses would be accessed using the banked-memory, direct-addressing scheme of Figure A2-4c. An instruction using this scheme is preceded by a

```
MOVLB  0x00
```

or a

```
MOVLB  0x01
```

instruction to load the Bank Select Register with the number to be used as the upper *nibble* (i.e., upper four bits) of the full 12-bit address.

An application making use of more than 128 bytes of RAM is likely to use indirect addressing to access one or more large arrays. These would extend over the RAM locations that cannot be reached by access-bank direct addressing. Cody Planteen's QwikLst utility suppresses the a = 0 parameter of an access-bank directly addressed RAM variable. The result is one less parameter in the .lst file calling for interpretation.

A2.4 F/W DISTINCTION

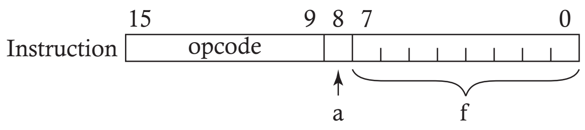
Another parameter associated with instructions that directly address an operand is the *destination* parameter, *F/W*. It is one of the distinguishing characteristics of Microchip's PIC® microcontrollers that the result of an operation on a variable can be returned

Mnemonic	Operands	Description	Words	Cycles	Status bits affected
ADDLW	k	Add literal value into WREG	1	1	C,OV,N,Z
ADDWF	f,F/W	Add WREG and f, putting result into F or W	1	1	C,OV,N,Z
ADDWFC	f,F/W	Add WREG and f and carry bit, putting result into F or W	1	1	C,OV,N,Z
ANDLW	k	AND literal value into WREG	1	1	N,Z
ANDWF	f,F/W	AND WREG with f, putting result into F or W	1	1	N,Z
BC	label	If carry or if no borrow (C=1), then branch to labeled instruction	1	2 or 1	-
BCF	f,b	Clear bit b of register f, where b = 0 to 7	1	1	-
BN	label	If negative (N=1), then branch to labeled instruction	1	2 or 1	-
BNC	label	If no carry or if borrow (C=0), then branch to labeled instruction	1	2 or 1	-
BNN	label	If ≥0 (N=0), then branch to labeled instruction	1	2 or 1	-
BNOV	label	If no overflow of signed-number operation, then branch to labeled instruction	1	2 or 1	-
BNZ	label	If not zero (Z=0), then branch to labeled instruction	1	2 or 1	-
BRA	label	Branch to labeled instruction	1	2	-
BSF	f,b	Set bit b of register f, where b = 0 to 7	1	1	-
BTFSC	f,b	Test bit b of register f, where b = 0 to 7; skip if clear	1	2 or 1	-
BTFSS	f,b	Test bit b of register f, where b = 0 to 7; skip if set	1	2 or 1	-
BTG	f,b	Toggle bit b of register f, where b = 0 to 7	1	1	-
BOV	label	If overflow of signed-number operation, then branch to labeled instruction	1	2 or 1	-
BZ	label	If zero (Z=1), then branch to labeled instruction	1	2 or 1	-
CALL	label	Call subroutine	2	2	-
CALL	label,FAST	Call subroutine; copy (WREG)→WS, (STATUS)→STATUSS, (BSR)→BSRS	2	2	-
CLRF	f	Load f with 0x00	1	1	Z
CLRWDT		Clear watchdog timer	1	1	-
COMF	f,F/W	Complement f, putting result into F or W	1	1	N,Z
CPFSEQ	f	Skip if f is equal to WREG	1	2 or 1	-
CPFSGT	f	Skip if f is greater than WREG (unsigned compare)	1	2 or 1	-
CPFSLT	f	Skip if f is less than WREG (unsigned compare)	1	2 or 1	-
DAW		Decimal adjust binary sum in WREG of two packed BCD numbers	1	1	C
DECF	f,F/W	Decrement f, putting result into F or W	1	1	C,OV,N,Z
DECFSZ	f,F/W	Decrement f, putting result into F or W; skip if zero	1	2 or 1	-
DCFSNZ	f,F/W	Decrement f, putting result into F or W; skip if not zero	1	2 or 1	-
GOTO	label	Go to labeled instruction	2	2	-
INCF	f,F/W	Increment f, putting result into F or W	1	1	C,OV,N,Z
INCFSZ	f,F/W	Increment f, putting result into F or W; skip if zero	1	2 or 1	-
INFSNZ	f,F/W	Increment f, putting result into F or W; skip if not zero	1	2 or 1	-
IORLW	k	Inclusive-OR literal value into WREG	1	1	N,Z
IORWF	f,F/W	Inclusive-OR WREG with f, putting result into F or W	1	1	N,Z
LFSR	i,k	Load FSRi with address of operand	2	2	-
MOVF	f,F/W	Move f to F or W	1	1	N,Z

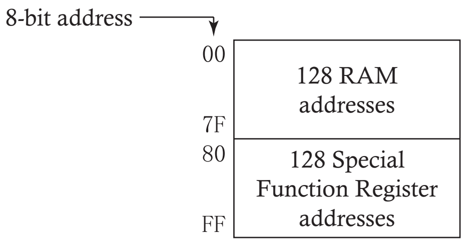
FIGURE A2-3 Assembly language instruction set

Mnemonic	Operands	Description	Words	Cycles	Status bits affected
MOVFF	fS,fD	Move fS to fD	2	2	-
MOVLB	k	Move literal value to BSR<3:0>, where k = 0 to 15, to set the bank for direct addressing with a=1	1	1	-
MOVLW	k	Move literal value to WREG	1	1	-
MOVWF	f	Move WREG to f	1	1	-
MULLW	k	Multiply unsigned WREG with literal value, putting result into PRODH:PRODL	1	1	-
MULWF	f	Multiply unsigned WREG with f, putting result in PRODH:PRODL	1	1	-
NEGF	f	Change sign of a twos-complement-coded one-byte number	1	1	C,OV,N,Z
NOP		No operation	1	1	-
POP		Discard address on top of stack	1	1	-
PUSH		Push address of next instruction onto stack	1	1	-
RCALL	label	Call subroutine	1	2	-
RESET		Software reset to same state as is achieved with the MCLR input	1	1	C,OV,N,Z
RETFIE		Return from interrupt; reenable interrupts	1	2	-
RETFIE	FAST	Return from interrupt; reenable interrupts; restore state from shadow registers (WS)→WREG, (STATUSS)→ STATUS, (BSRS)→ BSR	1	2	C,OV,N,Z
RETLW	k	Return from subroutine, putting literal value into WREG	1	2	-
RETURN		Return from subroutine	1	2	-
RETURN	FAST	Return from subroutine; restore state from shadow registers (WS)→ WREG	1	2	C,OV,N,Z
RLCF	f,F/W	Copy f into F or W; rotate left through carry bit (9-bit rotate left)	1	1	C,N,Z
RLNCF	f,F/W	Copy f into F or W; rotate left without carry bit (8-bit rotate left)	1	1	N,Z
RRCF	f,F/W	Copy f into F or W; rotate right through carry bit (9-bit rotate right)	1	1	C,N,Z
RRNCF	f,F/W	Copy f into F or W; rotate right without carry bit (8-bit rotate right)	1	1	N,Z
SETF	f	Load f with 0xFF	1	1	-
SLEEP		Normally enter sleep mode; if IDLEN=1, enter idle mode	1	1	-
SUBFWB	f,F/W	Subtract f and borrow bit from WREG, putting result into F or W	1	1	C,OV,N,Z
SUBLW	k	Subtract WREG from literal value, putting result into WREG	1	1	C,OV,N,Z
SUBWF	f,F/W	Subtract WREG from f, putting result into F or W	1	1	C,OV,N,Z
SUBWFB	f,F/W	Subtract WREG and borrow bit from f, putting result into F or W	1	1	C,OV,N,Z
SWAPF	f,F/W	Swap four-bit nibbles of f, putting result into F or W	1	1	-
TBLRD		Read from program memory location pointed to by TBLPTR into TABLAT	1	2	-
TBLRDPOSTDEC		Read from program memory location pointed to by TBLPTR into TABLAT, then decrement TBLPTR	1	2	-
TBLRDPOSTINC		Read from program memory location pointed to by TBLPTR into TABLAT, then increment TBLPTR	1	2	-
TBLRDPREINC		Increment TBLPTR, then read from program memory location pointed to by TBLPTR into TABLAT	1	2	-
TSTFSZ	f	Test f; skip if zero	1	2 or 1	-
XORLW	k	Exclusive-OR literal value into WREG	1	1	N,Z
XORWF	f,F/W	Exclusive-OR WREG with f, putting result into F or W	1	1	N,Z

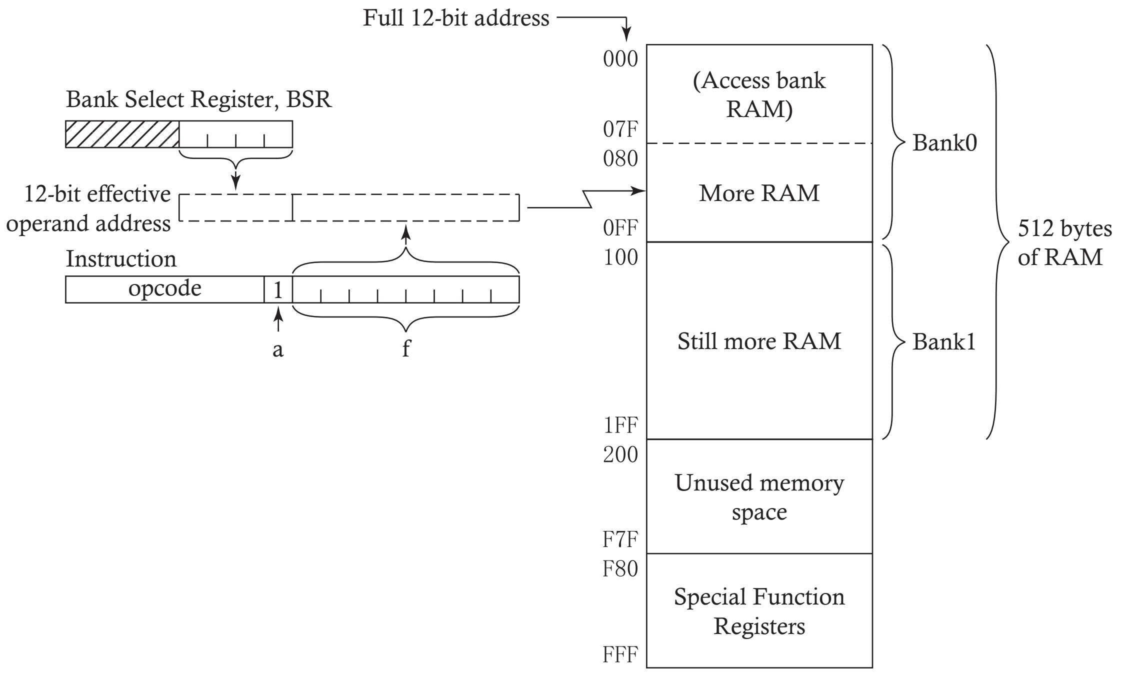
FIGURE A2-3 *(continued)*



(a) Nine bits of an instruction identifying the source of an operand.



(b) Access bank direct addressing (a = 0).

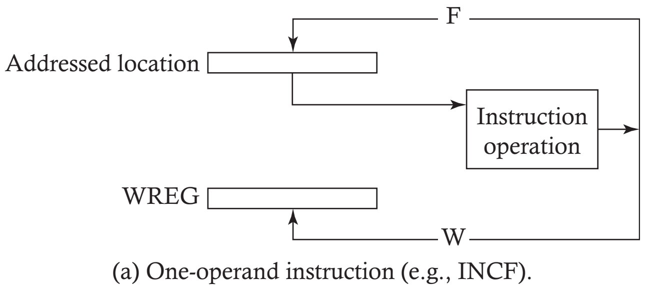


(c) Banked memory direct addressing (a = 1).

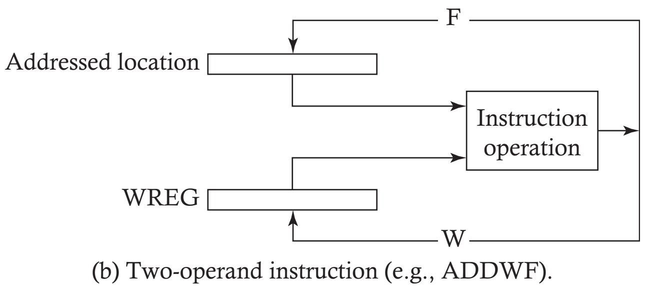
FIGURE A2-4 Direct addressing

to the variable or, alternatively, to the CPU's **WREG** working register, as shown in Figure A2-5. The .lst file generated by the linker shows both the "a" parameter of the last section and the "F/W" parameter of this section as

0x0 or 0x1



(a) One-operand instruction (e.g., INCF).



(b) Two-operand instruction (e.g., ADDWF).

FIGURE A2-5 Specifying the F or W destination of an instruction

The QwikLst utility suppresses the a = 0 parameter (as discussed in the last section) and converts the 0 or 1 value listed for the "F/W" parameter into a **W** when the destination of the operation is **WREG** and an **F** when the operation returns the result back to the source address used by the instruction.

A2.5 NAME REPLACEMENTS FOR OPERAND ADDRESSES

The final and most significant clarification performed by the QwikLst utility is the replacement of the .lst file's listing of the address of an operand by its name. This clarification is especially helpful when one source file line produces many lines of assembly code. For example, consider the raw .lst file code segment shown in Figure A2-6a and the recasting of the segment into the qwik.lst code segment of Figure A2-6b. Using a utility developed by Kenneth Kinion, the two-byte *int* variable, **ALIVECNT**, is reexpressed as **ALIVECNTL** and **ALIVECNTH**, to indicate the low and high bytes. The improvement in clarity is evident.

A2.6 COUNTING CYCLES

One reason to look at the .lst file is to discern the number of cycles that a segment of code generated by the compiler will take to execute. Consider the assembly code produced for implementing the **Delay** macro, shown in Figure A2-7. The loop that is traversed repeatedly as the delay parameter is counted down to zero ends with the line

```
00FA  D7F7     BRA  L002
```

```
                                            void BlinkAlive()
                                            {
000178   9883     BCF      0x83,0x4,0x0        PORTDbits.RD4 = 0;          // Turn off LED
00017a   2a05     INCF     0x5,0x1,0x0         if (++ALIVECNT == 400)      // Increment counter and return if not 400
00017c   0e00     MOVLW    0x0
00017e   2206     ADDWFC   0x6,0x1,0x0
000180   0e90     MOVLW    0x90
000182   1805     XORWF    0x5,0x0,0x0
000184   e106     BNZ      0x192
000186   0e01     MOVLW    0x1
000188   1806     XORWF    0x6,0x0,0x0
00018a   e103     BNZ      0x192
                                               {
00018c   6a05     CLRF     0x5,0x0                ALIVECNT = 0;            // Reset ALIVECNT
00018e   6a06     CLRF     0x6,0x0
000190   8883     BSF      0x83,0x4,0x0           PORTDbits.RD4 = 1;       // Turn on LED for one looptime
                                               }
000192   0012     RETURN   0x0              }
```

(a) Listing file

```
                                          void BlinkAlive()
                                          {
0178   9883       BCF      PORTD,4           PORTDbits.RD4 = 0;            // Turn off LED
017A   2A05       INCF     ALIVECNTL,F       if (++ALIVECNT == 400)        // Increment counter and return if not 400
017C   0E00       MOVLW    0x00
017E   2206       ADDWFC   ALIVECNTH,F
0180   0E90       MOVLW    0x90
0182   1805       XORWF    ALIVECNTL,W
0184   E106       BNZ      L003
0186   0E01       MOVLW    0x01
0188   1806       XORWF    ALIVECNTH,W
018A   E103       BNZ      L003
                                             {
018C   6A05       CLRF     ALIVECNTL            ALIVECNT = 0;              // Reset ALIVECNT
018E   6A06       CLRF     ALIVECNTH
0190   8883       BSF      PORTD,4              PORTDbits.RD4 = 1;         // Turn on LED for one looptime
                                             }
0192   0012 L003 RETURN                   }
```

(b) Translation into qwik.lst

FIGURE A2-6 QwikLst translation of operands

```
00E2   0E50       MOVLW     0x50      Delay(50000);    // Pause for half a second
00E4   6E0F       MOVWF     DELAYL
00E6   0EC3       MOVLW     0xC3
00E8   6E10       MOVWF     DELAYH
00EA   060F  L002 DECF      DELAYL,F
00EC   0E00       MOVLW     0x00
00EE   5A10       SUBWFB    DELAYH,F
00F0   500F       MOVF      DELAYL,W
00F2   1010       IORWF     DELAYH,W
00F4   E003       BZ        L001
00F6   0000       NOP
00F8   0000       NOP
00FA   D7F7       BRA       L002
00FC   9CD0  L001
```

FIGURE A2-7 Delay macro's assembly code

The **BRA** instruction tells the CPU to branch to the program address labeled L002 that holds the instruction

```
00EA  060F L002 DECF  DELAYL,F
```

This instruction decrements the low byte of the 2-byte variable **DELAY.**

Every instruction in this loop of instructions executes in one cycle except **BRA** (2 cycles) and **BZ** (2 or 1 cycles). The **BZ** instruction tests the result of an operation that preceded it. If that 8-bit result was 0x00, the CPU will take two cycles as it branches to address 0x00FC. However, while the CPU is looping for the first 49,999 times, the **BZ** instruction does not branch. Thus, the number of cycles taken to execute one of these 49,999 loops (beginning with the **DECF** instruction and ending with the **BRA** instruction) is

$$1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 2 = 10 \text{ cycles}$$

A2.7 FLAG BITS

The right-most column of Figure A2-3 lists any of four **STATUS** register flags that are affected by an instruction. For example, the **CLRF** instruction, in addition to writing 0x00 to a RAM variable or a Special Function Register, will set the **Z** bit. As another example, the **IORWF** instruction will (inclusive) OR the content of **WREG** bit by bit with the addressed operand, putting the result back into either the operand (**F**) or **WREG** (**W**). If this result is 0x00, the **Z** bit will be set to 1; otherwise, the **Z** bit will be cleared to 0.

The **C** bit will be set by an add operation of two unsigned 1-byte numbers that produces a result that is greater than 255, the largest 8-bit unsigned number. Actually, the CPU does not know if numbers are signed or unsigned. It just sets the **C** bit if that ninth bit of the operation is a 1 and clears **C** otherwise.

An add operation of two signed 1-byte numbers expressed in two's-complement code will set the **OV** (overflow) bit if the result is larger than 127, the largest 1-byte signed number. For a source file that adds two 1-byte numbers, the C compiler will look to see if they are defined as unsigned or signed numbers and then test the **C** bit or the **OV** bit to determine whether an overflow occurred.

Subtract operations of unsigned and signed numbers represent a borrow condition with the complement of the **C** and **OV** bits. That is, a borrow resulting from an unsigned subtraction will produce **C** = 0. A borrow resulting from a signed subtraction will produce **OV** = 0.

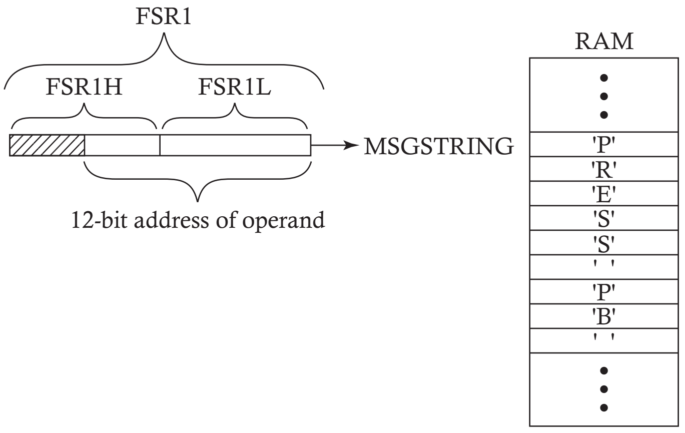
The **N** status bit signifies whether the result of an operation on signed numbers is negative (**N** = 1). If the result is 0 or positive, then **N** = 0.

A2.8 INDIRECT ADDRESSING OF RAM VARIABLES

The PIC18 family of microcontrollers includes in the CPU three 12-bit pointers, any one of which can be loaded with the *address* of an operand. Then the operand can be accessed *indirectly* via this pointer. This is especially useful for implementing pointers arising in a user program, but is also employed by the C compiler for more prosaic tasks, such as providing a function with scratchpad variables.

An example of indirect addressing is illustrated in Figure A2-8. **FSR1**, one of the three pointers in the CPU, has been initialized to point to the first byte of **MSGSTRING** in Figure A2-8a. The initialization of **FSR1** can be carried out in either of two ways, as shown in Figure A2-8b. Figure A2-8c shows the instruction that can be included in a loop of instructions to send each byte to the LCD. The instruction tells the CPU to read a byte from the location pointed to by **FSR1**, to increment **FSR1**, and to write the byte to the Serial Peripheral Interface's **SSPBUF** register.

FIGURE A2-8 Indirect addressing into a variable string



(a) FSR1 use as a variable pointer

```
MOVLW   low MSGSTRING           ;FSR1L = low byte of address of MSGSTRING
MOVWF   FSR1L
MOVLW   high MSGSTRING          ;FSR1H = high byte of address of MSGSTRING
MOVWF   FSR1H

     or

LFSR  1,MSGSTRING               ;FSR1 = address of MSGSTRING
```

(b) Loading FSR1 pointer

```
MOVFF   POSTINC1,SSPBUF         ;Send string element to PC
```

(c) Copy operand pointed to by FSR1 to SSPBUF, then increment FSR1

FIGURE A2-8 *(continued)*

The five operations associated with indirect addressing are **INDFi**, **POSTINCi**, **POSTDECi**, **PREINCi**, and **PLUSWi**, where $i = 0$, 1, or 2 to identify which of the three pointers, **FSR0**, **FSR1**, or **FSR2**, is to be used for carrying out the operation. If the operand is **INDFi**, the CPU accesses the location pointed to by **FSRi**. If the operand is **POSTINCi**, the CPU accesses the location pointed to by **FSRi** and *then* increments **FSRi**. If the operand is **POSTDECi**, the CPU accesses the location pointed to by **FSRi** and then decrements **FSRi**. If the operand is **PREINCi**, the CPU *first* increments **FSRi** and then accesses the location pointed to by the incremented **FSRi**. Finally, if the operand is **PLUSWi**, the CPU *temporarily* adds its working register, **WREG** (treated as a signed number), to the address in **FSRi** to form the address of the instruction operand.

It should be pointed out that the address associated with any one of the five indirect addressing operations is not a physical register. Rather, the CPU interprets an access of the address as a signal to carry out the specified indirect addressing operation. Thus the instruction

```
INCF  INDF1,F
```

increments the location pointed to by **FSR1**.

A2.9 CPU REGISTERS

The CPU registers are shown in Figure A2-9, together with the operands associated with indirect addressing (e.g., **INDF0**). Notice that the working register is identified in two ways. As the *destination* of an operation, it is **W**:

```
SUBWF  COUNTER,W
```

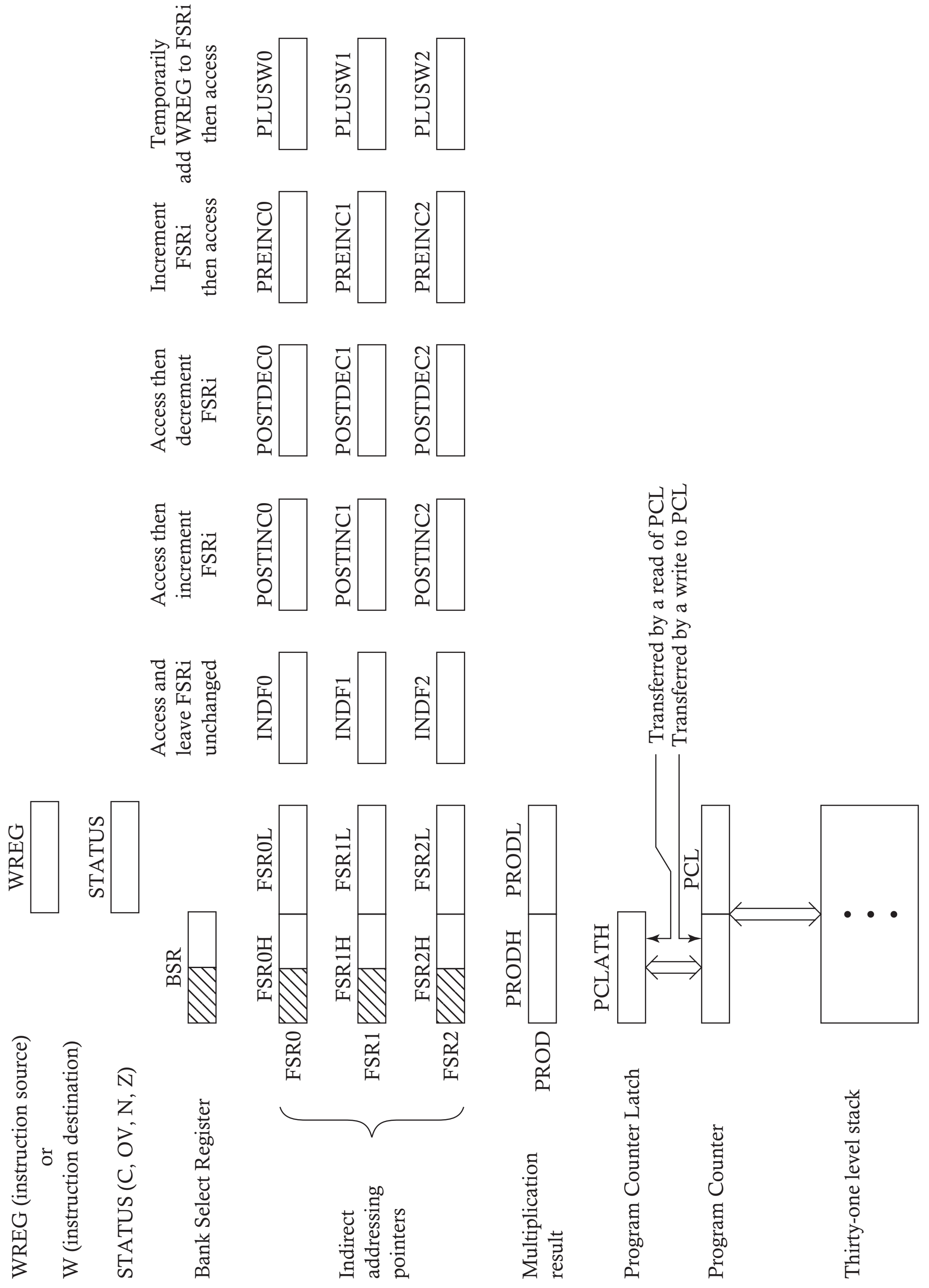


FIGURE A2-9 CPU registers

As the *source* of an operation, it is **WREG**:

```
BCF  WREG,7
```

Any multiply operation makes use of one of the two 8-bit unsigned multiply instructions, **MULLW** or **MULWF**, putting the 16-bit result into the CPU registers **PRODH:PRODL**. Any instruction that operates on the program counter operates on the lower 8 bits, **PCL**. If that instruction is going to write back to **PCL**, then, at the same time that the write takes place, the content of **PCLATH** (Program Counter LATcH) is written into the upper byte of the program counter.

Subroutine calls and interrupts nest return addresses onto the 31-level stack. The PIC18 family of microcontrollers has an extended instruction set that can be used to facilitate the extension of the stack into RAM, to support a more powerful compilation. However, for the benefit of QwikBug and also for the benefit of execution speed, that extended instruction set is disabled.

A2.10 INDIRECT ADDRESSING OF PROGRAM MEMORY

The instruction set of Figure A2-3 includes four instructions that are used for accessing constant strings, arrays, and tables stored in program memory with the mechanism shown in Figure A2-10. These instructions are

TBLRD, **TBLRDPOSTINC**, **TBLRDPOSTDEC**, **TBLRDPREINC**

They use the **TBLPTRH:TBLPTRL** register pair to identify the program memory address that is to be copied to the **TABLAT** register. They include the option of postincrementing, postdecrementing, or preincrementing **TBLPTRH:TBLPTRL**.

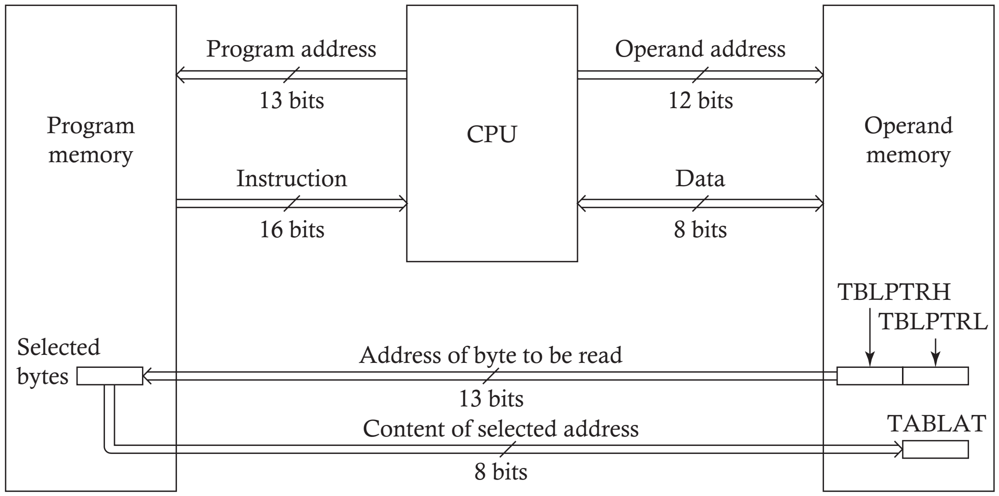
An example of the creation and use of a table is shown in Figure A2-11. This **FormHex** table expects to be handed a *char* variable containing a value ranging from 0 to 15. It returns the ASCII-coded hex character corresponding to that value. Thus, in the example whose qwik.lst file segment is shown in Figure A2-11b, the source file line

```
ASCII = FormHex[NIBBLE];
```

produces the result in **ASCII** in 10 cycles.

A2.11 SPECIAL FUNCTION REGISTERS

The special function registers are listed in Figure A2-12 along with their addresses, both in 3-nibble full address form and 2-nibble Access Bank address form.



(a) Special mechanism for reading from program memory.

TBLRD	Read from program memory location pointed to by TBLPTRH:TBLPTRL into TABLAT
TBLRDPOSTINC	Read from program memory location pointed to by TBLPTRH:TBLPTRL into TABLAT, then increment TBLPTRH:TBLPTRL
TBLRDPOSTDEC	Read from program memory location pointed to by TBLPTRH:TBLPTRL into TABLAT, then decrement TBLPTRH:TBLPTRL
TBLRDPREINC	Increment TBLPTRH:TBLPTRL, then read from program memory location pointed to by TBLPTRH:TBLPTRL into TABLAT

(b) Special instructions for reading from program memory.

FIGURE A2-10 Reading operands from program memory

FIGURE A2-11 Creation and use of a program memory table to convert a number between 0 and 15 to the ASCII code for its hex representation

```
/*******************************
* Constant table
*******************************
*/

const char rom FormHex[] = "0123456789ABCDEF";
```

(a) A table to be compiled into program memory beginning at address 0x02F8.

```
02ce   502d       MOVF      NIBBLE,W        ASCII = FormHex[NIBBLE];
02d0   6af7       CLRF      TBLPTRH
02d2   0f46       ADDLW     0xF8
02d4   6ef6       MOVWF     TBLPTRL
02d6   0e04       MOVLW     0x02
02d8   22f7       ADDWFC    TBLPTRH,F
02da   0008       TBLRD
02dc   50f5       MOVF      TABLAT,W
02de   6e2e       MOVWF     ASCII
```

(b) Assembly code for ASCII = FormHex[NIBBLE]; executes in 10 cycles.

FIGURE A2-11 *(continued)*

FIGURE A2-12a Special Function Registers and their full hex addresses

ADCON0	FC2	INDF0	FEF	PORTD	F83	T2CON	FCA
ADCON1	FC1	INDF1	FE7	PORTE	F84	T3CON	FB1
ADCON2	FC0	INDF2	FDF	POSTDEC0	FED	TABLAT	FF5
ADRESH	FC4	INTCON	FF2	POSTDEC1	FE5	TBLPTRH	FF7
ADRESL	FC3	INTCON2	FF1	POSTDEC2	FDD	TBLPTRL	FF6
BAUDCON	FB8	INTCON3	FF0	POSTINC0	FEE	TBLPTRU	FF8
BSR	FE0	IPR1	F9F	POSTINC1	FE6	TMR0H	FD7
CCP1CON	FBD	IPR2	FA2	POSTINC2	FDE	TMR0L	FD6
CCP2CON	FBA	LATA	F89	PR2	FCB	TMR1H	FCF
CCPR1H	FBF	LATB	F8A	PREINC0	FEC	TMR1L	FCE
CCPR1L	FBE	LATC	F8B	PREINC1	FE4	TMR2	FCC
CCPR2H	FBC	LATD	F8C	PREINC2	FDC	TMR3H	FB3
CCPR2L	FBB	LATE	F8D	PRODH	FF4	TMR3L	FB2
CMCON	FB4	OSCCON	FD3	PRODL	FF3	TOSH	FFE
CVRCON	FB5	OSCTUNE	F9B	RCON	FD0	TOSL	FFD
ECCP1AS	FB6	PCL	FF9	RCREG	FAE	TOSU	FFF
ECCP1DEL	FB7	PCLATH	FFA	RCSTA	FAB	TRISA	F92
EEADR	FA9	PCLATU	FFB	SPBRG	FAF	TRISB	F93
EECON1	FA6	PIE1	F9D	SPBRGH	FB0	TRISC	F94
EECON2	FA7	PIE2	FA0	SSPADD	FC8	TRISD	F95
EEDATA	FA8	PIR1	F9E	SSPBUF	FC9	TRISE	F96
FSR0H	FEA	PIR2	FA1	SSPCON1	FC6	TXREG	FAD
FSR0L	FE9	PLUSW0	FEB	SSPCON2	FC5	TXSTA	FAC
FSR1H	FE2	PLUSW1	FE3	SSPSTAT	FC7	WDTCON	FD1
FSR1L	FE1	PLUSW2	FDB	STATUS	FD8	WREG	FE8
FSR2H	FDA	PORTA	F80	STKPTR	FFC		
FSR2L	FD9	PORTB	F81	T0CON	FD5		
HLVDCON	FD2	PORTC	F82	T1CON	FCD		

ADCON0	C2	INDF0	EF	PORTD	83	T2CON	CA
ADCON1	C1	INDF1	E7	PORTE	84	T3CON	B1
ADCON2	C0	INDF2	DF	POSTDEC0	ED	TABLAT	F5
ADRESH	C4	INTCON	F2	POSTDEC1	E5	TBLPTRH	F7
ADRESL	C3	INTCON2	F1	POSTDEC2	DD	TBLPTRL	F6
BAUDCON	B8	INTCON3	F0	POSTINC0	EE	TBLPTRU	F8
BSR	E0	IPR1	9F	POSTINC1	E6	TMR0H	D7
CCP1CON	BD	IPR2	A2	POSTINC2	DE	TMR0L	D6
CCP2CON	BA	LATA	89	PR2	CB	TMR1H	CF
CCPR1H	BF	LATB	8A	PREINC0	EC	TMR1L	CE
CCPR1L	BE	LATC	8B	PREINC1	E4	TMR2	CC
CCPR2H	BC	LATD	8C	PREINC2	DC	TMR3H	B3
CCPR2L	BB	LATE	8D	PRODH	F4	TMR3L	B2
CMCON	B4	OSCCON	D3	PRODL	F3	TOSH	FE
CVRCON	B5	OSCTUNE	9B	RCON	D0	TOSL	FD
ECCP1AS	B6	PCL	F9	RCREG	AE	TOSU	FF
ECCP1DEL	B7	PCLATH	FA	RCSTA	AB	TRISA	92
EEADR	A9	PCLATU	FB	SPBRG	AF	TRISB	93
EECON1	A6	PIE1	9D	SPBRGH	B0	TRISC	94
EECON2	A7	PIE2	A0	SSPADD	C8	TRISD	95
EEDATA	A8	PIR1	9E	SSPBUF	C9	TRISE	96
FSR0H	EA	PIR2	A1	SSPCON1	C6	TXREG	AD
FSR0L	E9	PLUSW0	EB	SSPCON2	C5	TXSTA	AC
FSR1H	E2	PLUSW1	E3	SSPSTAT	C7	WDTCON	D1
FSR1L	E1	PLUSW2	DB	STATUS	D8	WREG	E8
FSR2H	DA	PORTA	80	STKPTR	FC		
FSR2L	D9	PORTB	81	T0CON	D5		
HLVDCON	D2	PORTC	82	T1CON	CD		

FIGURE A2-12b Special Function Registers and their Access Bank hex addresses

Appendix

A3

QWIK&LOW BOARD IN DETAIL

The Qwik&Low board was designed by the author, with creative insights from Rick Farmer who developed the board layout and from Bill Kaduck and Dave Cornish of MICRODESIGNS who are producing the board. The Gerber files for the board artwork are freely available from www.qwikandlow.com along with the schematic and parts list for the board.

What follows are a few comments to explain some of the circuitry. The UART circuitry shown in the top left of Figure A3-1 includes an (unpopulated) 3-pin header, H1. The cuttable link on the back of the board between the pins labeled **PC** and **RX** provide the default connection. If a user wants to add a transducer to the board that has a UART output, the link between the **PC** and **RX** pins is cut, a 3-pin header is added to the board, and the UART output from the new device is connected to the **RX'** pin of the header (using #30 wirewrap wire). Then a jumper (i.e., the 100-mil shunt, H2X, in the parts list of Figure A3-2) can be used to connect **PC** to **RX** for downloading user code. The jumper can be moved between **RX'** and **RX** to run the user code.

The **RX'** pin is connected to a 100kΩ pull-up resistor, R4, to ensure that the **RX** input does not float. The input is pulled high to match a transducer with a normal idle-high UART output. In this case, the application program must reinitialize the UART so that **RXDTP** = 0 in the **BAUDCON** register. This undoes QwikBug's inversion of

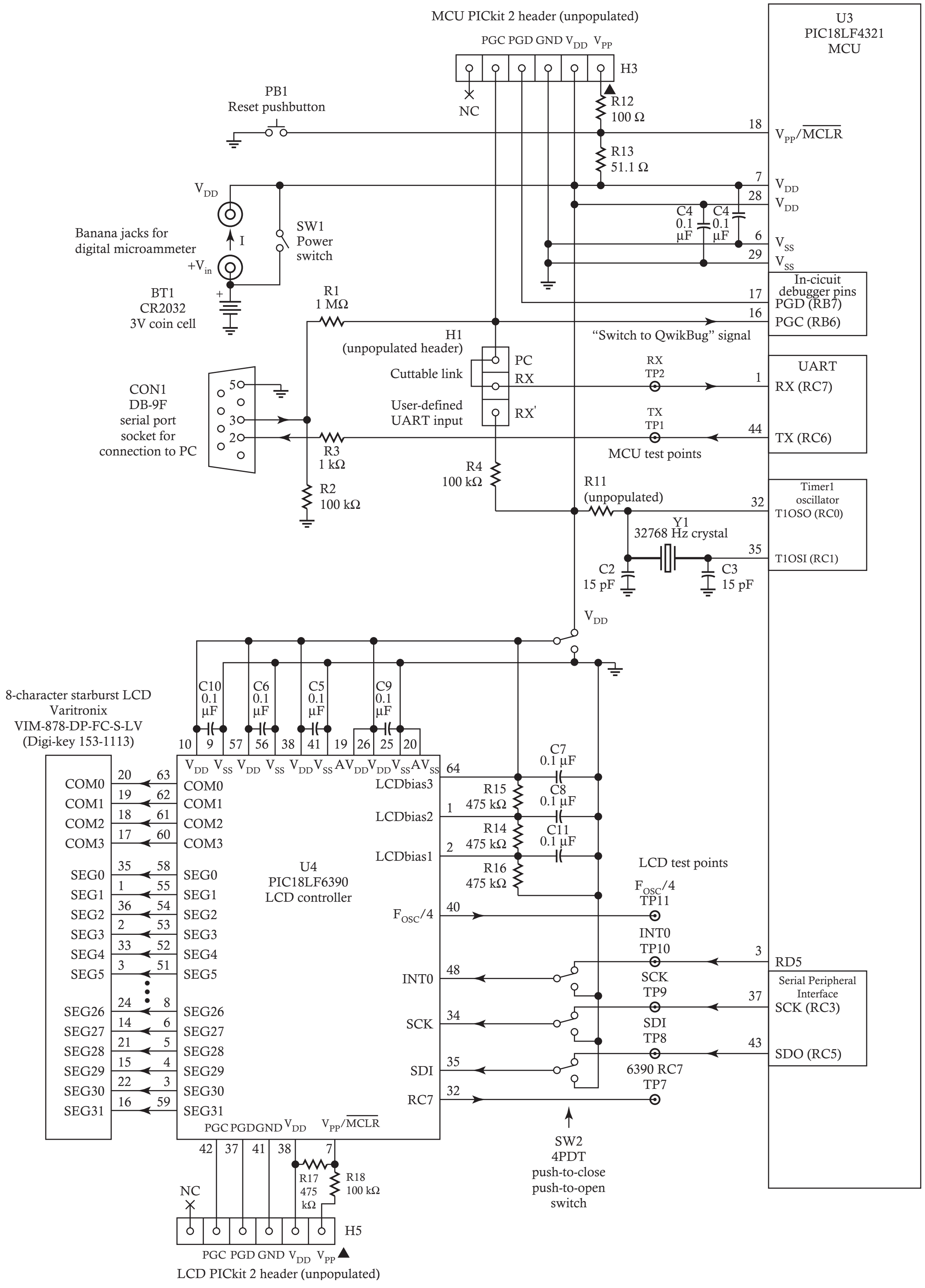


FIGURE A3-1 Qwik&Low schematic

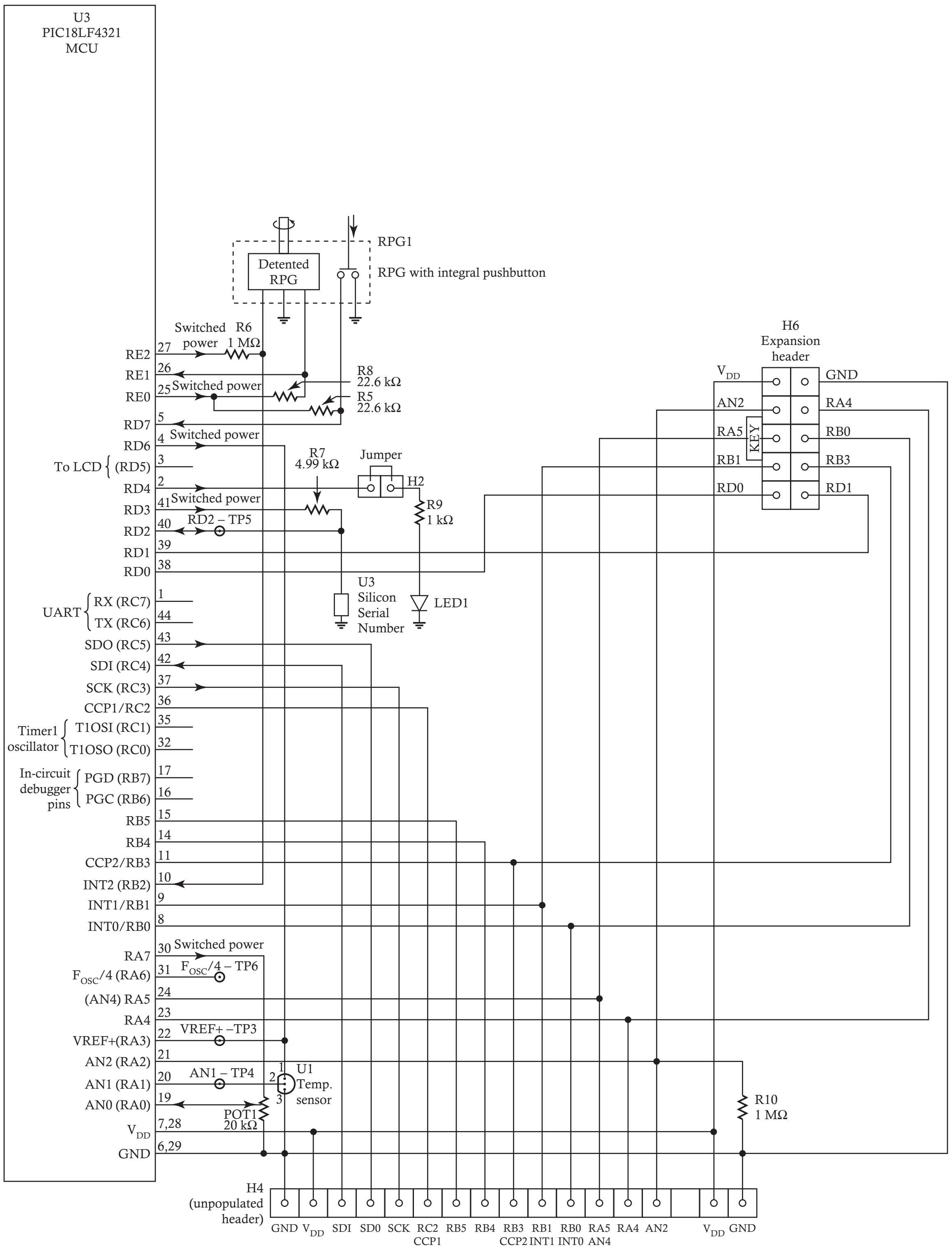


FIGURE A3-1 *(continued)*

Quantity	Order Number	Distributor	Designation	Footprint	Part Description	Manufacturer	Manufacturer's Part Number	Cost Each	Total Cost
1					Qwik&Low board	PCBCART			6.00
2	BC1305CT	Digi-Key	C2,3	1206	15 pf 50V NPO ceramic capacitor	BC Comp.	VJ1206A150JXACW1BC	0.15	0.30
9	311-1179-1	Digi-Key	C1,4,5,6,7,8,9,10,11	1206	0.1uF 50V X7R ceramic capacitor	Yageo	CC1206KRX7R9BB104	0.10	0.90
2	311-100FRCT	Digi-Key	R12,18	1206	100 ohm 1/4W 1% thick film resist.	Yageo	RC1206FR-07100RL	0.09	0.18
2	311-1.00KFRCT	Digi-Key	R3,9	1206	1.00K 1/4W 1% thick film resistor	Yageo	RC1206FR-071KL	0.09	0.18
1	311-4.99KFRCT	Digi-Key	R7	1206	4.99K 1/4W 1% thick film resistor	Yageo	RC1206FR-074K99L	0.09	0.09
2	311-22.6KFRCT	Digi-Key	R5,8	1206	22.6K 1/4W 1% thick film resistor	Yageo	RC1206FR-0722K6L	0.09	0.18
1	311-51.1KFRCT	Digi-Key	R13	1206	51.1K 1/4W 1% thick film resistor	Yageo	RC1206FR-0751K1L	0.09	0.09
2	311-100KFRCT	Digi-Key	R2,4	1206	100K 1/4W 1% thick film resistor	Yageo	RC1206FR-07100KL	0.09	0.18
4	311-475KFRCT	Digi-Key	R14,15,16,17	1206	475K 1/4W 1% thick film resistor	Yageo	RC1206FR-07475KL	0.09	0.36
3	311-1.00MFRCT	Digi-Key	R1,6,10	1206	1.00M 1/4W 1% thick film resistor	Yageo	RC1206FR-071ML	0.09	0.27
		Digi-Key	R11	1206	(leave this unpopulated)				
1	535-9166-1	Digi-Key	Y1		32768 Hz watch crystal	Abracon	ABS25-32.768KHZ-T	1.05	1.05
1	PIC18LF4321-I/PT	Digi-Key	U3	TQFP-44	Microcontroller	Microchip	PIC18LF4321-I/PT	3.37	3.37
1	DS2401P+	Digi-Key	U2	TSOC-6	Silicon Serial Number	Dallas Semi.	DS2401P+	1.52	1.52
1	PIC18LF6390-I/PT	Digi-Key	U4	TQFP-64	LCD controller	Microchip	PIC18LF6390-I/PT	3.27	3.27
								Back of board components =	17.94
1	P605	Digi-Key	LED1	T-1	Red LED, through-hole mounting	Panasonic	LN28RALXU	0.38	0.38
1	AD22103KTZ	Digi-Key	U1	TO-92	Temperature sensor	Analog Dev.	AD22103KTZ	2.86	2.86
1	P4D2203	Digi-Key	POT1	POT_HFAA	20K Thumbwheel POT	Panasonic	EVL-HFAA06B24	2.07	2.07
1	A32117	Digi-Key	CON1		Female DB-9 socket	Tyco AMP	5747844-4	2.72	2.72
1	153-1113	Digi-Key	LCD1	2" x 1"	8-char starburst LCD	Varitronix	VIM-878-DP-FC-S-LV	2.92	2.92
1	BH32T-C	Digi-Key	BT1		CR2032 coin cell holder	MPD	BH32T-C	1.04	1.04
1	P189	Digi-Key	BT1X		CR2032 coin cell	Panasonic	CR2032	0.25	0.25
2	J147	Digi-Key		1/4" Dia hole	Nickel plated banana jack	Johnson	108-0740-001	1.48	2.96
1	EG1874	Digi-Key	SW2		4PDT push-on, push-off switch	E-Switch	TL4201EEYA	2.44	2.44
1	A26512-40	Digi-Key	H1	SIP100_3P	(leave this unpopulated)				
1	A26512-40	Digi-Key	H2	SIP100_2P	Male straight 2-pin header	Tyco AMP	4-103239-0	0.10	0.10
1	S9001	Digi-Key	H2X		100 mil shunt	Sullins	SPC02SYAN	0.12	0.12
2	A26512-40	Digi-Key	H3,5	SIP100_6P	(leave these unpopulated)				
1	A26512-40	Digi-Key	H4	SIP100_17P	(leave this unpopulated)				
1	HRP10H	Digi-Key	H6		Male shrouded 5x2 pin header	Assmann	AWHW10G-0202-T-R	0.59	0.59
1	855-0017	Allied	PB1		Sealed pushbutton switch	Cannon	KSA0M211 LFT	0.34	0.34
1	688-EC11G1524402	Mouser	RPG1		Detented RPG-PB switch comb.	ALPS	EC11G1524402	2.19	2.19
1	108-0051-EVX	Mouser	SW1	SIP100_3P	SPDT toggle switch	Mountain Sw.	108-0051-EVX	1.97	1.97
4	517-SJ-5003BK	Mouser			Four rubber bumpers for feet	3M	SJ-5003 (BLACK)	0.13	0.52
								Remaining components =	23.47
								Total =	41.41

FIGURE A3-2 Qwik&Low parts list

the **RX** input. If a transducer is used with a UART output that idles low, the R4 resistor should be removed from the board.

One interesting addition to the board is a 12-key (0–9, *, #) keypad with an 80¢, 14-pin PIC16F505 microcontroller providing:

- The decoding of the keypad.
- A bit-banged UART interface.

The CPU clock of the PIC16F505 is 1 MHz ± 2%. This is sufficiently accurate so that if an output pin is changed appropriately at intervals of

$$1{,}000{,}000/19{,}200 = 52.08333\ \mu s \approx 52\ \mu s$$

then the MCU will read this serial input exactly as it would from a 19,200 baud UART.

The Timer1 crystal oscillator circuitry shown in Figure A3-1 immediately below the UART circuitry includes pads labeled R11. The intent is to make it possible to add a high-impedance (e.g., ≈ 3 MΩ) resistor, to try to get the Timer1 oscillator to run reliably with the

```
# pragma config LPT1OSC = ON
```

configuration choice. This choice, the reliability of which has not been thoroughly explored, would decrease the current draw of the Timer1 oscillator and the Timer1 counter from about 6 μA to 1 μA. That low-power oscillator configuration is meant to run with a 5 V supply, not the 3 V supply of the Qwik&Low board. A Microchip application engineer intimately involved with the oscillator design suggested this pull-up resistor modification.

The unpopulated H4 header provides test points for the pin of a scope probe, to test user-generated outputs (e.g., for a pulse-width measurement on the **RC2** or **RB0** pin). The unpopulated header also provides solder points for an add-on part installed on the surface-mount pads located on the front or the back of the board. Point-to-point wiring with #30 wirewrap wire can produce a clean job of the addition. Be sure to remove any solder flux, especially if water-soluble (i.e., conductive) flux is used.

The prototype area on the board allows DIP parts and discrete parts to be added easily. For surface-mount parts that do not fit on the available surface-mount patterns (e.g., an SOIC part with up to 28 pins), consider the use of one of the surface-mount-to-DIP adapters available from www.beldynsys.com.

The test points dispersed over the board are designed for the pin probe of a scope. $F_{OSC}/4$, the CPU clock for each of the PIC microcontrollers on the board, allows a user to gauge the awake/sleep behavior of each chip under varying circumstances.

An output pin on the MCU can be employed in a user program to monitor time intervals of interest. In like manner, one of the few free output pins of the LCD controller, **RC7**, can be monitored. TP7 is such a test point, located near the right edge of the board below TP8. It is unlabeled (but can be probed) on the front of the board. It is labeled on the back of the board. The intent is to be unobtrusive for normal use of the board, but available for users who wish to explore modifications to the LCD controller's program code.

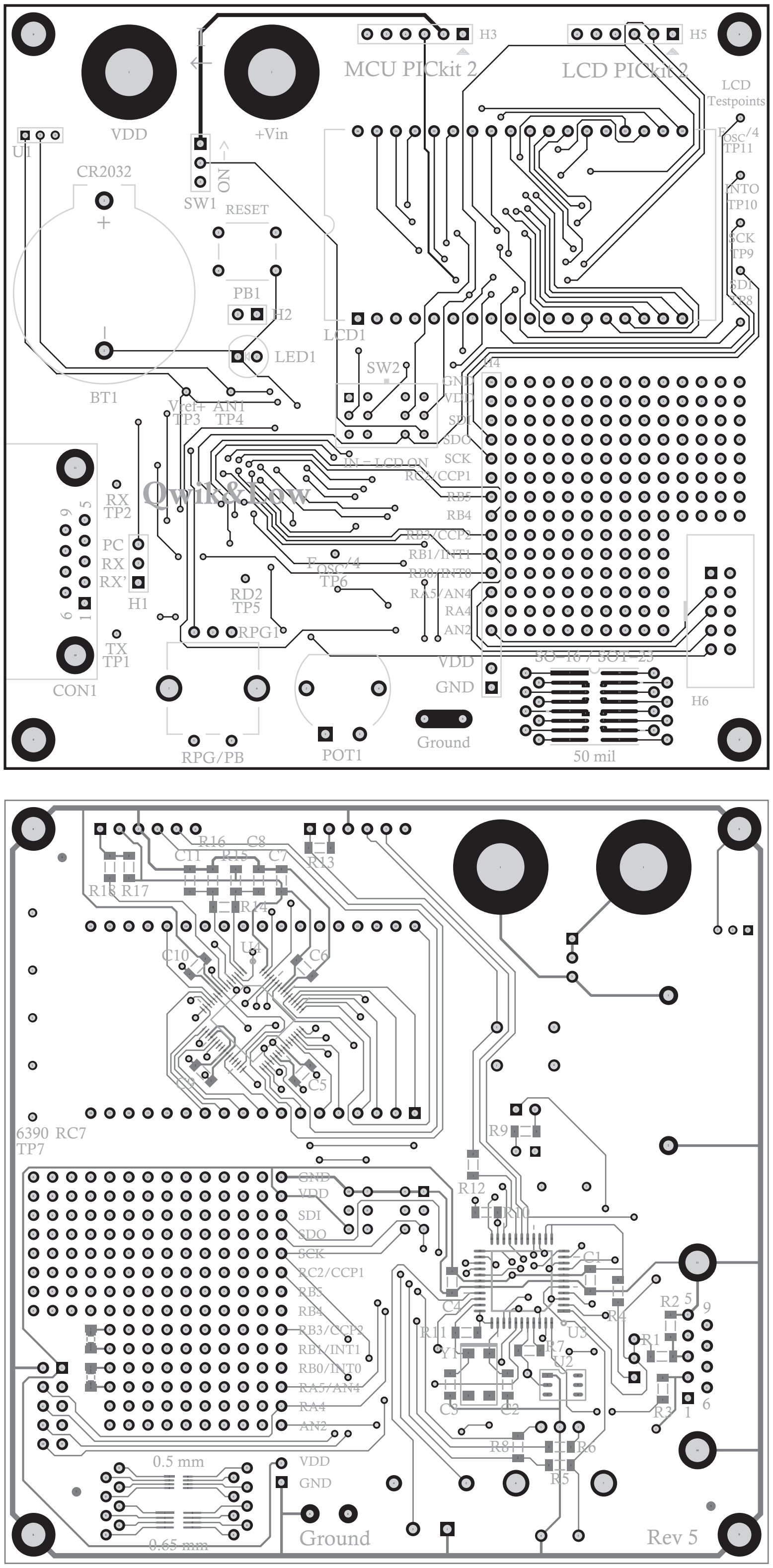


FIGURE A3-3 Front and back of Qwik&Low board

The parts list includes the same Digi-Key part number, A26512-40, for all of the single-in-line (SIP) headers, H1, H2, H3, H4, and H5. This part is actually a 40-pin strip that is designed to be cut with diagonal cutters into headers having the desired number of pins.

The parts list shows the Qwik&Low board as being manufactured by PCBCART (www.pcbcart.com), a vendor for having low-cost, high-quality boards made in China with a nominal two-week delivery schedule. The $6.00 price shown in Figure A3-2 was the approximate unit price with an order for 30 boards. The Gerber files on the www.qwikandlow.com website can be used to place an order for boards. However, proceed cautiously! As the artwork of Figure A3-3 for the front and the back of the board illustrates, the surface-mount parts for the back of the board, particularly the two PIC microcontrollers, call for considerable experience in dealing with fine-pitch parts. Also, when completed, a PICkit 2 programmer must be available to program the two chips. Before undertaking such a venture, be sure to consider the alternative of purchasing a built and tested board from www.microdesignsinc.com.

Chris Bruhn and Peter Ralston have developed a Performance Verification program, PV.c, available from the www.qwikandlow.com website. This program can be compiled, loaded via QwikBug, and run. It initially sends the serial number from the DS2401 chip (Chapter Fifteen) to the QwikBug console, verifying both the chip and the serial interface. It then runs the LCD display through four quick tests before moving on to display the temperature, and to verify the operation of the Timer1 oscillator and the operation of the potentiometer/ADC combination. If the Timer1 oscillator is working correctly, the middle number on the display will increment every second. The LED also blinks every second. As the potentiometer is turned from full CCW to full CW, the right-hand number changes from 00 to FE or FF. Turning the RPG turns on individual segments of the LCD, verifying that no adjacent pins of the display are shorted together. Turning the RPG clockwise or counterclockwise also increments/decrements a number sent to the QwikBug console. The LED is turned on in response to the pressing of the RPG's pushbutton. The current draw of the board while running this program (with the LED jumper removed and the LCD switched off) is about 20 μA.

Appendix

A4

STEPPER-MOTOR BOARD IN DETAIL

The schematic for the stepper-motor controller board discussed in Chapter Eight is shown in Figure A4-1. Its 12 V power is supplied by a wall-transformer power supply. The Schottky rectifier, D1, is present to reduce the risk of burning out the board circuitry by inadvertently plugging in a wrong wall transformer having its polarity reversed.

The 3.3 V voltage regulator, U2, derives the logic supply voltage used by the controller chip, U1, from the 12 V motor supply voltage. (If a higher motor supply voltage is used, it must not exceed the 20 V maximum input specification of this voltage regulator.) The input and output capacitors, C9 and C8, are included to meet the stability requirements of the voltage regulator. The 0.1 μF ceramic capacitor, C5, provides an RF bypass for the Allegro chip's logic supply voltage. The intent of using a supply voltage of 3.3 V (rather than 3.0 V) is to ensure that the DIR (Direction) and STEP inputs from the Qwik&Low board do not exceed this supply voltage.

The motor current can be monitored by connecting a digital milliammeter between the two pins of the H1 header labeled

$$V_m \rightarrow I$$

and opening the power switch, TB2Y. As the motor steps, the DMM will display the average current. This current is set by the values of the two current-sensing resistors, R4

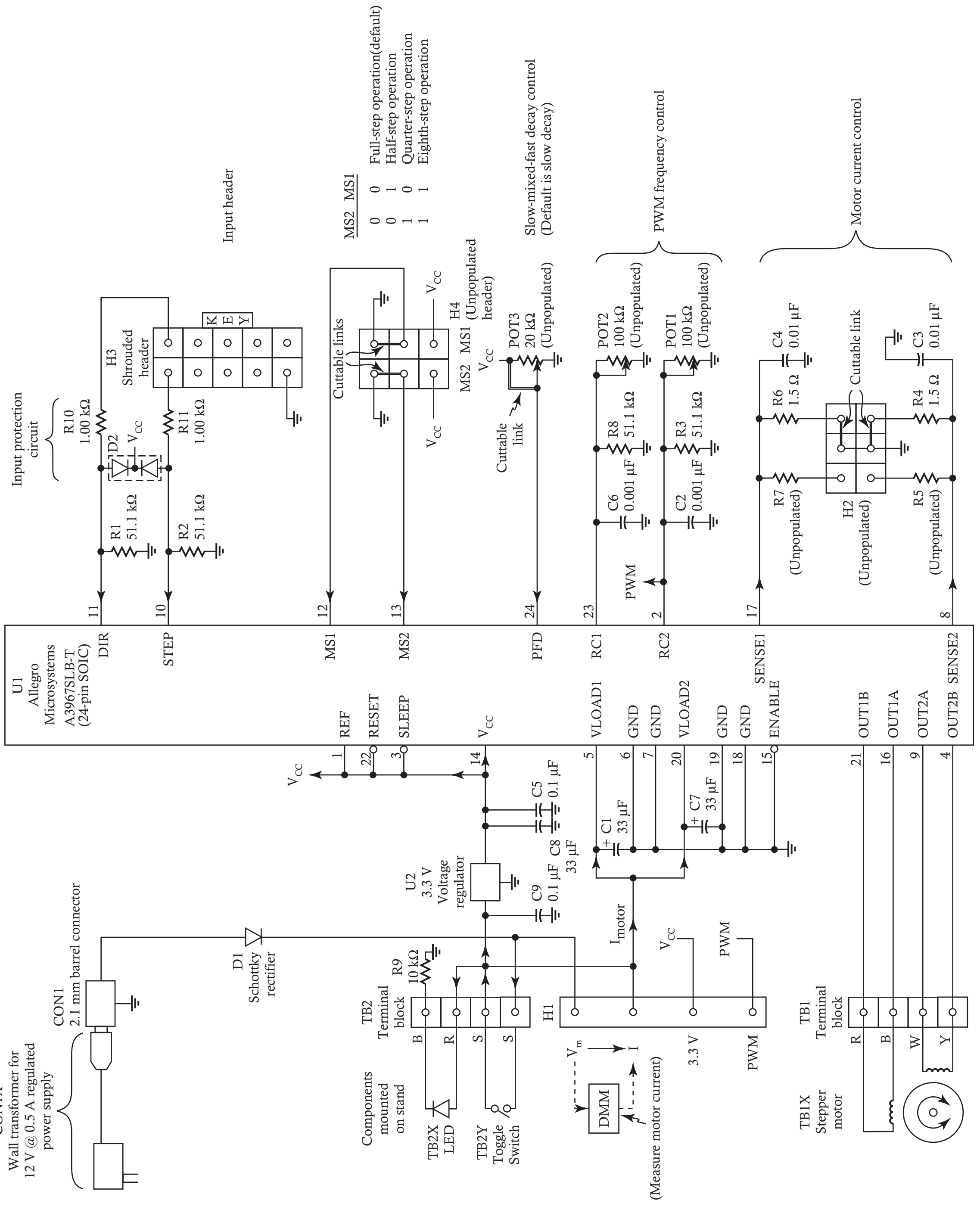


FIGURE A4-1 Stepper motor driver board

and R6, shown in the lower-right corner of Figure A4-1. The Allegro chip is designed to be able to drive each of the two stepper-motor windings with a voltage of up to 30 V and a current of up to ±750 mA. When full stepping (the default stepping mode), the current in each winding is approximated by the equation[1]

$$I_{\text{winding}} = \pm\frac{0.707 \times V_{\text{REF}}}{8 \times R_S} = \pm\frac{292}{R_S}\text{mA}$$

with VREF = 3.3 V. With R_S = 1.5Ω, Iwinding = 194 mA. When operating in the full-step mode, both windings are energized with this (±) current at each step position for a total load of 386 mA for the wall transformer. The Allegro chip carries out full stepping by alternately reversing the current in one winding and then the other winding.

The stepper motor listed in the parts list of Figure A4-2 steps 200 full steps per revolution. Even finer resolution (i.e., 400 s/r, 800 s/r, or 1,600 s/r) can be achieved by adding a 2 × 3 pin header in the H4 header pattern on the board, cutting the two links on the back of the board as shown near the upper right of Figure A4-1, and then adding two jumpers to select the stepping mode. For example, if both center pins are connected to the bottom pins of H4, eighth-step operation will result. While the total current drawn from the wall transformer for full stepping is constant, for any of the other modes, the total current varies with the step position, between a maximum equal to that found for full stepping and a minimum equal to 0.707 times that value.

The pulse-width-modulation (PWM) control circuit defaults to the nominal RC values suggested by Allegro in the data sheet for this driver chip. Each motor winding is subjected to a current that alternates between ramping up and decaying down. When one of the winding currents is low, the 12 V power supply voltage is applied across the winding until the voltage across the current-sensing resistor crosses its threshold voltage. At that point, the power supply voltage is cut off from the motor winding. The winding current decays for a time determined by C2–R3 (for one winding) or C6–R8 (for the other one). The rate of decay is determined by the voltage on the **PFD** pin of Figure A4-1. With this pin voltage defaulting to 3.3 V, the current decays relatively slowly, with minimum current ripple. The maximum stepping rate is evidently achieved with the fast decay setting of 0 V on the **PFD** pin, albeit with increased audible noise and vibration.

In contrast to the Qwik&Low board, the stepper-motor board is a simpler board to build. Figure A4-3 shows the front and back artwork. Even though the board employs size 1206 surface-mount resistors and capacitors, these are relatively large (for surface-mount parts). The resistors are stamped with their resistor values, a help for checking the board. Check with www.microdesignsinc.com for the price and availability of complete stepper-motor assemblies, ready for lab use. Check with www.qwikandlow.com for information on obtaining the bare stepper-motor controller board.

[1] Allegro Microsystems Technical Paper STP 01-2, pp. 3–4

Unpopulated	Quantity	Part Number	Distributor	Designation	Footprint	Part Description	Manufacturer	Cost Each	Total Cost
	1					Stepper motor driver board	PCBCART	4.50	4.50
	2	311-1170-1	Digi-Key	C2,6	1206	0.001uF 50V X7R ceramic capacitor	Yageo	0.09	0.18
	2	311-1174-1	Digi-Key	C3,4	1206	0.01uF 50V X7R ceramic capacitor	Yageo	0.09	0.18
	2	311-1179-1	Digi-Key	C5,9	1206	0.1uF 50V X7R ceramic capacitor	Yageo	0.10	0.20
	3	P835	Digi-Key	C1,7,8	0.25” Dia.	33uF 35V electrolytic capacitor	Panasonic	0.15	0.45
	2	311-1.00KFRCT	Digi-Key	R10,11	1206	1.00K 1/4W 1% thick film resistor	Yageo	0.09	0.18
	1	311-10.0KFRCT	Digi-Key	R9	1206	10.0K 1/4W 1% thick film resistor	Yageo	0.09	0.09
	4	311-51.1KFRCT	Digi-Key	R1,2,3,8	1206	51.1K 1/4W 1% thick film resistor	Yageo	0.09	0.36
	2	P1.5ASCT	Digi-Key	R4,6	1210	1.5 ohm 1.2W 5% thick film resistor	Panasonic	0.36	0.72
x	2	P1.8ASCT	Digi-Key	R5,7	1210	1.8 ohm 1.2W 5% thick film resistor	Panasonic	0.36	0.72
x	0	P1.2ASCT	Digi-Key	R5,7	1210	1.2 ohm 1.2W 5% thick film resistor	Panasonic	0.36	0.00
x	1	490-2839	Digi-Key	POT3	0.26”x0.27”	20K 1 turn, side adjust trimpot	Murata	0.79	0.79
x	2	490-2834	Digi-Key	POT1,2	0.26”x0.27”	100K 1 turn, side adjust trimpot	Murata	0.79	1.58
x	2	S2011E-36	Digi-Key	H2,4	3x2 pin header	Male straight 2 row x 3 pin header	Sullins ($2.74/72pins)	0.23	0.46
	1	S1011E-36	Digi-Key	H1	7x1 pin header	Male straight 7x1 header with 2,4,6 removed	Sullins ($1.51/36pins)	0.30	0.30
	1	HRP10H	Digi-Key	H3		Male shrouded 100 mil header, 5x2 pins	Assmann Elect.	0.59	0.59
	2	HKC10H	Digi-Key	H3X		Mating female ribbon cable connector	Assmann Elect.	0.50	1.00
	1	AE10G-5	Digi-Key	H3Y		10-conductor ribbon cable, 5 feet	Assmann Elect.	0.90	0.90
	1	1N5818	Digi-Key	D1	DO-41	30V,1A Schottky rectifier	Vishay	0.23	0.23
	1	BAT54C-FDICT	Digi-Key	D2	SOT-23	Dual Schottky diode	Diodes Inc.	0.60	0.60
	1	MC33269DTRK-3OSCT	Digi-Key	U2	DPAK	3.3V voltage regulator	ON Semi.	1.00	1.00
	1	620-1073	Digi-Key	U1	SOIC-24	Motor driver, A3967SLB-T	Allegro	2.75	2.75
	2	277-1736	Digi-Key	TB1,2	0.57”x0.36”	4-conductor terminal block, right angle	Phoenix	0.90	1.80
	1	CP-202A	Digi-Key	CON1	0.57”x0.36”	2.1mm barrel connector for power	CUI Stack	0.38	0.38
	1	T983-P5P	Digi-Key	CON1X		12V@0.5A regulated wall wart	CUI Inc.	5.38	5.38
								Stepper driver board + power =	20.84
	1	163395	Jameco	TB1X		Bipolar stepper motor, 8.4V, 280mA/phase	Applied Motion	4.79	4.79
	1	350-1592	Digi-Key	TB2X	4mm Dia.	Panel-mounted blue LED	Dialight	2.70	2.70
	1	EG2446	Digi-Key	TB2Y	5mm Dia.	SPDT toggle switch	E-Switch	3.45	3.45
	1					Aluminum stand		10.00	10.00
	4	517-SJ-5003BK	Mouser			Four rubber feet for aluminum stand	3M	0.52	2.08
	4	2036K	Digi-Key		1/4” round	Round spacer for 4-40 screw, 1/8” long	Keystone	0.80	3.20
	4	H781	Digi-Key			4-40 x 3/8” machine screw	Building Fasteners	0.08	0.32
	4	H216	Digi-Key			4-40 hex nut	Building Fasteners	0.08	0.32
	4	3011	Bolt Depot			Internal tooth lock washers, 4		0.02	0.08
	2	7914	Bolt Depot			Socket head screws, 6-32 x 1-1/2		0.18	0.36
	2	3012	Bolt Depot			Internal tooth lock washers, 6		0.02	0.04
	2	2643	Bolt Depot			Hex machine screw nuts, 6-32		0.02	0.04
								Parts for motor and stand =	27.38
								Total =	48.22

FIGURE A4-2 Stepper Motor Driver parts list

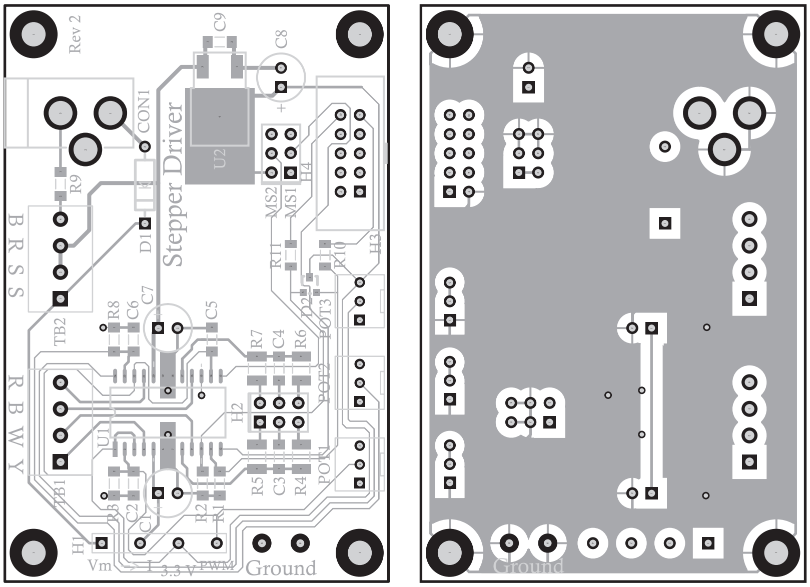


FIGURE A4-3 Front and back of Stepper Driver board

INDEX

As embedded microcontrollers reach into all corners of modern life, many applications can benefit from coin-cell battery power. Some benefits are reduced product size and cost, enhanced design simplicity, portability, and electrical isolation. Microchip Technology, the number one supplier of 8-bit microcontrollers in the world, is using their nanoWatt Technology™ features to achieve these benefits.

This book explores how these features impact the design process. It employs the Qwik&Low board shown on the cover as the learning vehicle for the reader. The board is available from MICRODESIGNS (www.microdesignsinc.com)

This book introduces the reader to code writing for a microcontroller via a series of template files and using Microchip's free version of their C compiler for their PIC18™ family of microcontrollers. Free supporting tools are available at the author's website, www.qwikandlow.com , including QwikBug, a debugging user interface for downloading code to the Qwik&Low board, running that code, and debugging it using a serial PC connection (via either a serial cable or a USB-to-serial adapter).

About the author: John Peatman, Professor of Electrical and Computer Engineering at the Georgia Institute of Technology, is the author of six earlier textbooks (two from Prentice Hall and four from McGraw-Hill).

Available as a $15.50 print-on-demand paperback book from www.lulu.com

or as a free download from www.qwikandlow.com

ISBN 978-0-9799770-0-8