INCLUDES

NEWNES ONLINE MEMBERSHIP

PIC MICROCONTROLLERS

know it all

- **A 360 degree view from our best-selling authors**
- **Key facts, designs, and applications fully detailed**
- **The ultimate hard-working desk reference: all the essential information, techniques, and tricks of the trade in one volume**

Di Jasio • Wilmshurst • Ibrahim • Morton
Bates • J. Smith • D.W. Smith • Hellebuyck

About the Authors

Martin P. Bates (Chapters 6, 7, 8, Appendices C, J) is the author of PIC Microcontrollers, 2E. He is currently lecturing on electronics and electrical engineering at Hastings College, UK. His interests include microcontroller applications and embedded system design.

Lucio Di Jasio (Chapters 24, 25, 26, 27, 28, 29) is the author of Programming 16-bit Microcontrollers in C. He joined Microchip Technology in 1995 as a Field Application Engineer. Since 2005, he has been in charge of the Application Segment Group, a cross-divisional team of engineers that develops and promotes Microchip's solutions across a wide range of application segments, including: utility metering, intelligent power conversion, motor control and lighting applications.

Chuck Hellebuyck (Chapters 15, 16, 17) is the author of Programming PIC Microcontrollers using PIC Basic. He is founder and president of Elproducts, Inc., a firm specializing in devices and project kits based on the PIC microcontroller. He writes a monthly column on the PIC microcontroller for "Nuts and Volts" magazine.

Dogan Ibrahim (Chapters 1, 14) is the author of PICBasic Projects. He works for the Transport for London in UK. He was formerly a lecturer at South Bank University and Head of Department of Computer Engineering at Near East University, Cyprus.

John Morton (Chapters 9, 10, Appendices D, E, F, G, H, I) is the author of The PIC Microcontroller. He is a Junior Research Fellow at St. John's College, Oxford, investigating experimental quantum computation using electron spins. He works in the Oxford University Materials Department and Clarendon Laboratory and in collaboration with the Quantum Information Processing IRC. His interests include PIC Microcontrollers.

D.W. Smith (Chapters 11, 12, 13) is the author of PIC in Practice. He has 30 years experience in the Electronics Industry. Before arriving at MMU he worked as an Electronics Design Engineer for ICL and Marconi. His teaching interests are focused on enabling Design and Technology students to implement microcontroller designs into their projects.

Jack Smith (Chapters 18, 19, 20, 21, 22, 23) is the author of Programming the PIC Microcontroller with MBasic. He is currently with Clifton Laboratories in Virginia. He was

PIC Microcontrollers

Newnes Know It All Series

PIC Microcontrollers: Know It All
Lucio Di Jasio, Tim Wilmshurst, Dogan Ibrahim, John Morton,
Martin Bates, Jack Smith, D.W. Smith, and Chuck Hellebuyck
ISBN: 978-0-7506-8615-0

Embedded Software: Know It All
Jean Labrosse, Jack Ganssle, Tammy Noergaard, Robert Oshana, Colin Walls, Keith Curtis,
Jason Andrews, David J. Katz, Rick Gentile, Kamal Hyder, and Bob Perrin
ISBN: 978-0-7506-8583-2

Embedded Hardware: Know It All
Jack Ganssle, Tammy Noergaard, Fred Eady, Lewin A.R.W. Edwards,
David J. Katz, Rick Gentile, Ken Arnold, Kamal Hyder, and Bob Perrin
ISBN: 978-0-7506-8584-9

Wireless Networking: Know It All
Praphul Chandra, Daniel M. Dobkin, Alan Bensky, Ron Olexa,
David Lide, and Farid Dowla
ISBN: 978-0-7506-8582-5

RF & Wireless Technologies: Know It All
Bruce Fette, Roberto Aiello, Praphul Chandra, Daniel Dobkin,
Alan Bensky, Douglas Miron, David Lide, Farid Dowla, and Ron Olexa
ISBN: 978-0-7506-8581-8

For more information on these and other Newnes titles visit: www.newnespress.com

PIC Microcontrollers

Lucio Di Jasio
Tim Wilmshurst
Dogan Ibrahim
John Morton
Martin P. Bates
Jack Smith
D. W. Smith
Chuck Hellebuyck

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Newnes is an imprint of Elsevier

Newnes is an imprint of Elsevier
30 Corporate Drive, Suite 400, Burlington, MA 01803, USA
Linacre House, Jordan Hill, Oxford OX2 8DP, UK

Library of Congress Cataloging-in-Publication Data

PIC microcontrollers : know it all / Lucio Di Jasio … [et al.].
p. cm. – (The Newnes know it all series)
ISBN-13: 978-0-7506-8615-0
1. Programmable controllers. 2. Microcomputers. 3. Microprocessors.
I. Di Jasio, Lucio.
TJ223. P76P52 2007
629.8′95416–dc22

2007025364

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

ISBN: 978-0-7506-8615-0

07 08 09 10 10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

Contents

This page intentionally left blank

Founder and a Consultant at TeleworX, Virginia, previously. He is currently working on several PIC Microcontroller amateur radio-related projects.

Tim Wilmshurst (Chapters 2, 3, 4, 5, Appendices A, B) is the author of Designing Embedded Systems with PIC Microcontrollers. He has been designing embedded systems since the early days of microcontrollers. For many years this was for Cambridge University, where he led a development team building original systems for research applications—for example in measurement of bullet speed, wind tunnel control, simulated earthquakes, or seeking a cure to snoring. Now he is Head of Electronic Systems at the University of Derby, where he aims to share his love of engineering design with his students.

SECTION I

An Introduction to PIC Microcontrollers

This page intentionally left blank

CHAPTER 1

The PIC Microcontroller Family

The PIC microcontroller family is manufactured by Microchip Technology Inc. Currently they are one of the most popular microcontrollers, used in many commercial and industrial applications. Over 120 million devices are sold each year.

The PIC microcontroller architecture is based on a modified Harvard RISC (Reduced Instruction Set Computer) instruction set with dual-bus architecture, providing fast and flexible design with an easy migration path from only 6 pins to 80 pins, and from 384 bytes to 128 kbytes of program memory.

PIC microcontrollers are available with many different specifications depending on:

- Memory Type
  - Flash
  - OTP (One-time-programmable)
  - ROM (Read-only-memory)
  - ROMless
- Input–Output (I/O) Pin Count
  - 4–18 pins
  - 20–28 pins
  - 32–44 pins
  - 45 and above pins
- Memory Size
  - 0.5–1 K
  - 2–4 K
  - 8–16 K
  - 24–32 K
  - 48–64 K
  - 96–128 K
- Special Features
  - CAN
  - USB

- LCD
- Motor Control
- Radio Frequency

Although there are many models of PIC microcontrollers, the nice thing is that they are upward compatible with each other and a program developed for one model can very easily, in many cases with no modifications, be run on other models of the family. The basic assembler instruction set of PIC microcontrollers consists of only 33 instructions and most of the family members (except the newly developed devices) use the same instruction set. This is why a program developed for one model can run on another model with similar architecture without any changes.

All PIC microcontrollers offer the following features:

- RISC instruction set with only a handful of instructions to learn
- Digital I/O ports
- On-chip timer with 8-bit prescaler
- Power-on reset
- Watchdog timer
- Power-saving SLEEP mode
- High source and sink current
- Direct, indirect, and relative addressing modes
- External clock interface
- RAM data memory
- EPROM or Flash program memory

Some devices offer the following additional features:

- Analog input channels
- Analog comparators
- Additional timer circuits
- EEPROM data memory
- External and internal interrupts
- Internal oscillator

- Pulse-width modulated (PWM) output
- USART serial interface

Some even more complex devices in the family offer the following additional features:

- CAN bus interface
- I^2C bus interface
- SPI bus interface
- Direct LCD interface
- USB interface
- Motor control

Although there are several hundred models of PIC microcontrollers, choosing a microcontroller for an application is not a difficult task and requires taking into account these factors:

- Number of I/O pins required
- Required peripherals (e.g., USART, USB)
- The minimum size of program memory
- The minimum size of RAM
- Whether or not EEPROM nonvolatile data memory is required
- Speed
- Physical size
- Cost

The important point to remember is that there could be many models that satisfy all of these requirements. You should always try to find the model that satisfies your minimum requirements and the one that does not offer more than you may need. For example, if you require a microcontroller with only 8 I/O pins and if there are two identical microcontrollers, one with 8 and the other one with 16 I/O pins, you should select the one with 8 I/O pins.

Although there are several hundred models of PIC microcontrollers, the family can be broken down into three main groups, which are:

- 12-bit instruction word (e.g., 12C5XX, 16C5X) (also referred to in this book as the 12 Series and the 16C5X Series)

- 14-bit instruction word (e.g., 16F8X, 16F87X) (also referred to in this book as the 16 Series)
- 16-bit instruction word (e.g., 17C7XX, 18C2XX) (also referred to in this book as the 17 Series and the 18 Series).

All three groups share the same RISC architecture and the same instruction set, with a few additional instructions available for the 14-bit models, and many more instructions available for the 16-bit models. Instructions occupy only one word in memory, thus increasing the code efficiency and reducing the required program memory. Instructions and data are transferred on separate buses, so the overall system performance is increased.

The features of some microcontrollers in each group are given in the following sections.

1.1 12-bit Instruction Word

Table 1.1 lists some of the devices in this group. These devices have a very simple architecture; however, as the prices of 14-bit devices have declined, it is rarely necessary to use a 12-bit device these days, except for the smaller physical size.

Table 1.1: Some 12-bit PIC Microcontrollers

Microcontroller	Program Memory	Data RAM	Max Speed (MHz)	I/O Ports	A/D Converter
12C508	512 × 12	25	4	6	–
16C54	384 × 12	25	20	12	–
16C57	2048 × 12	72	20	20	–
16C505	1024 × 12	41	4	12	–
16C58A	2048 × 12	73	20	12	–

PIC12C508: This is a low-cost, 8-pin device with 512 × 12 EPROM program memory, and 25 bytes of RAM data memory. The device can operate at up to 4-MHz clock input and the instruction set consists of only 33 instructions. The device features six I/O ports, 8-bit timer, power-on reset, watchdog timer, and internal 4-MHz oscillator capability. One of the major disadvantages of this microcontroller is that the program memory is EPROM-based and it cannot be erased or programmed using the standard programming devices. The program memory has to be erased using an EPROM eraser device (an ultraviolet light source).

The "F" version of this family (e.g., PIC12F508) is based on flash program memory, which can be erased and reprogrammed using the standard PIC programmer devices. Similarly, the "CE" version of the family (e.g., PIC12CE518) offers an additional 16-byte nonvolatile EEPROM data memory.

Figure 1.1 shows the pin configuration of the PIC12F508 microcontroller.

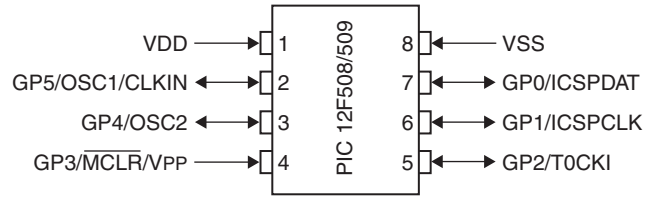


Figure 1.1: PIC12F508 Microcontroller

PIC16C5X: This is one of the earliest PIC microcontrollers. The device is 18-pin with a 384 × 12 EPROM program memory, 25 bytes of RAM data memory, 12 I/O ports, a timer, and a watchdog. Some other members in the family, such as PIC16C56, have the same architecture but more program memory (1024 × 12). PIC16C58A has more program memory (2048 × 12) and also more data memory (73 bytes of RAM). Figure 1.2 shows the pin configuration of the PIC16C56 microcontroller.

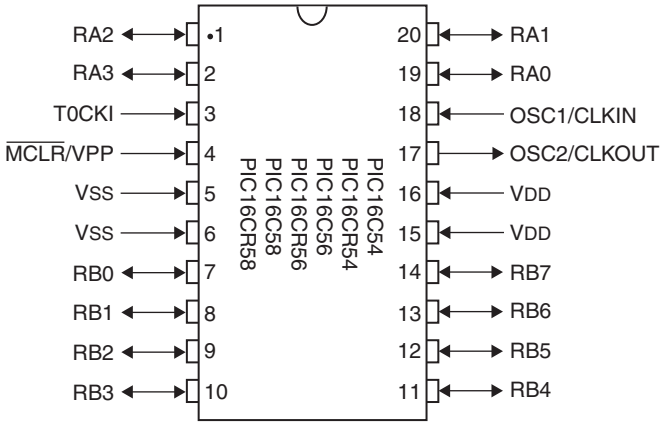


Figure 1.2: PIC16C56 Microcontroller

1.2 14-bit Instruction Word

This is a big family that includes many models of PIC microcontrollers. Most of the devices in this family can operate at up to a 20-MHz clock rate. The instruction set consists of 35 instructions. These devices offer advanced features such as internal and external interrupt sources. Table 1.2 lists some of the microcontrollers in this group.

PIC16C554: This microcontroller has similar architecture to the PIC16C54 but the instructions are 14 bits wide. The program memory is 512 × 14 and the data memory is 80 bytes of RAM. There are 13 I/O pins and each pin can source or sink 25 mA of current. Additionally, the device contains a timer and a watchdog.

Table 1.2: Some 14-bit Microcontrollers

Microcontroller	Program Memory	Data RAM	Max Speed (MHz)	I/O Ports	A/D Converter
16C554	512 × 14	80	20	13	–
16C64	2048 × 14	128	20	33	–
16F84	1024 × 14	36	10	13	–
16F627	1024 × 14	224	20	16	–
16F628	2048 × 14	224	20	16	–
16F676	1024 × 14	64	20	12	8
16F73	4096 × 14	192	20	22	5
16F876	8192 × 14	368	20	22	5
16F877	8192 × 14	368	20	33	8

PIC16F84: This has been one of the most popular PIC microcontrollers for a very long time. This is an 18-pin device and it offers 1024 × 14 flash program memory, 36 bytes of data RAM, 64 bytes of nonvolatile EEPROM data memory, 13 I/O pins, a timer, a watchdog, and internal and external interrupt sources. The timer is 8 bits wide but can be programmed to generate internal interrupts for timing purposes. PIC16F84 can be operated from a crystal or a resonator for accurate timing. A resistor-capacitor can also be used as a timing device for applications where accurate timing is not required. Figure 1.3 shows the pin configuration of this microcontroller. The pin descriptions are given in Table 1.3.

PIC16F877: This microcontroller is a 40-pin device and is one of the popular microcontrollers used in complex applications. The device offers 8192 × 14 flash program memory,

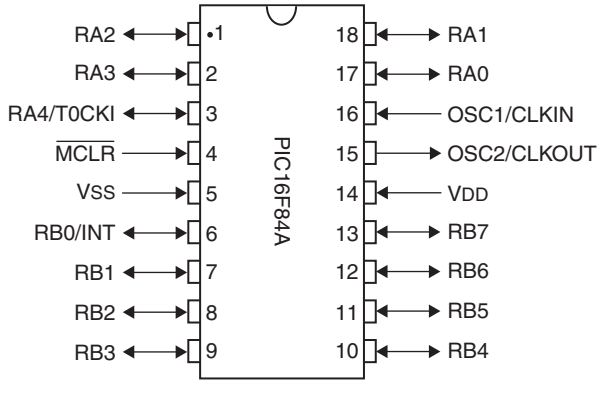


Figure 1.3: PIC16F84 Microcontroller Pin Configuration

Table 1.3: PIC16F84 Microcontroller Pin Descriptions

Pin	Description	Pin	Description
1	RA2—PORTA bit 2	10	RB4—PORTB bit 4
2	RA3—PORTA bit 3	11	RB5—PORTB bit 5
3	RA4/T0CK1—PORTA bit 4/Counter clk	12	RB6—PORTB bit 6
4	MCLR—Master clear	13	RB7—PORTB bit 7
5	Vss—Gnd	14	Vdd—+V supply
6	RB0/INT—PORTB bit 0	15	OSC2
7	RB1—PORTB bit 1	16	OSC1
8	RB2—PORTB bit 2	17	RA0—PORTA bit 0
9	RB3—PORTB bit 3	18	RA1—PORTA bit 1

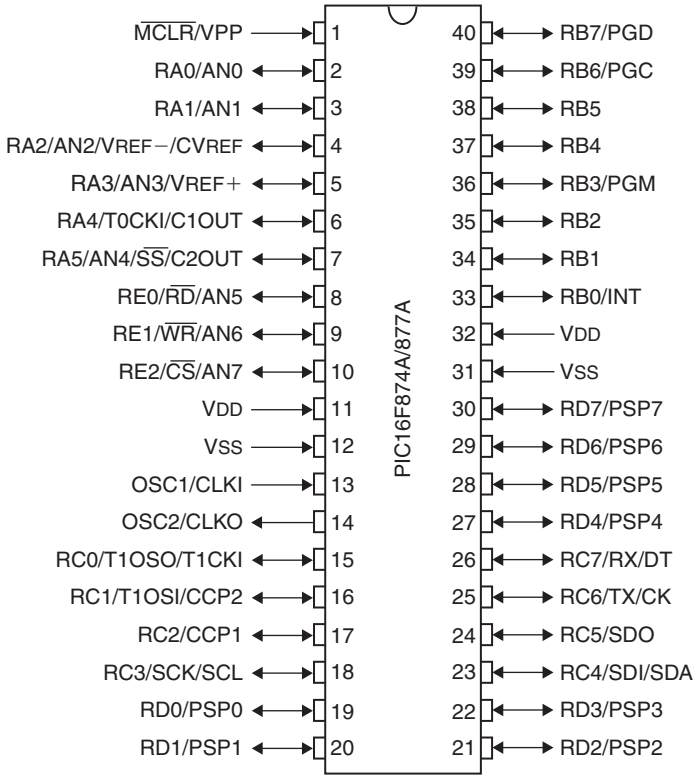


Figure 1.4: PIC16F877 Microcontroller Pin Configuration

368 bytes of RAM, 256 bytes of nonvolatile EEPROM memory, 33 I/O pins, 8 multiplexed A/D converters with 10 bits of resolution, PWM generator, three timers, an analog capture and comparator circuit, USART, and internal and external interrupt facilities. Figure 1.4 shows the pin configuration of this microcontroller.

PIC16F627: This is an 18-pin microcontroller with 1024 × 14 flash program memory. The device offers 224 bytes of RAM, 128 bytes of nonvolatile EEPROM memory, 16 I/O pins, two 8-bit timers, one 16-bit timer, a watchdog, and comparator circuits. This microcontroller is similar to PIC16F84, but offers more I/O pins, more program memory, and a lot more RAM. In addition, PIC16F627 is more suited to applications that require more than one timer. Figure 1.5 shows the pin configuration of this microcontroller.

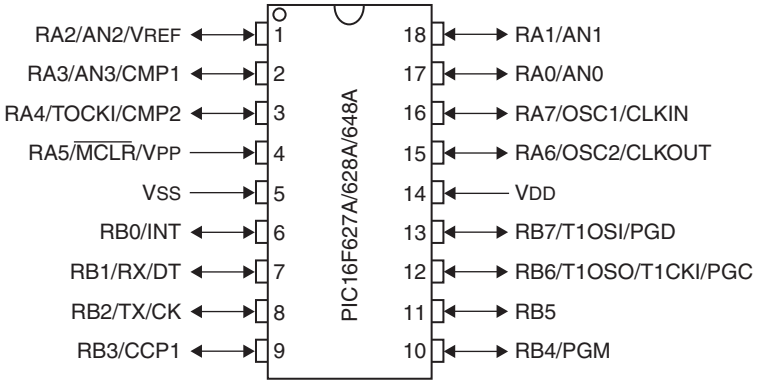


Figure 1.5: PIC16F627 Microcontroller Pin Configuration

PIC16F676: Figure 1.6 shows a 14-pin microcontroller that is becoming very popular. The device offers 1024 × 14 flash program memory, 64 bytes of RAM, 12 I/O pins, 128 bytes of EEPROM, 8 multiplexed A/D converters, each with 10-bit resolution, one 8-bit timer, one 16-bit timer, and a watchdog. One of the advantages of this microcontroller is the built-in A/D converter.

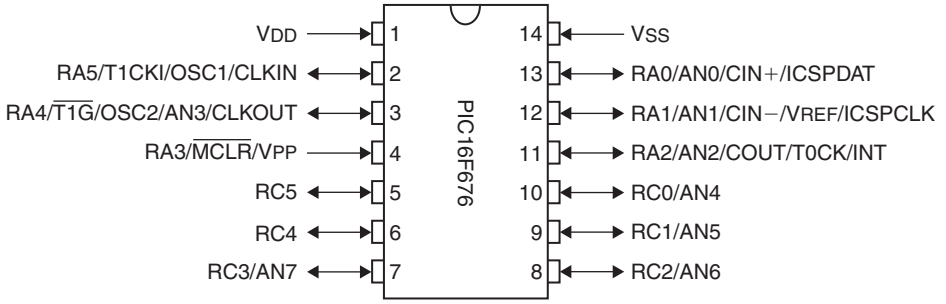


Figure 1.6: PIC16F676 Microcontroller Pin Configurationm

PIC16F73: This is a powerful 28-pin microcontroller with 4096 14 flash program memory, 192 bytes of RAM, 22 I/O pins, five multiplexed 8-bit A/D converters, two 8-bit timers, one 16-bit timer, watchdog, USART, and I^2C bus compatibility. This device combines A/D

converter, digital I/O, and serial I/O capability in a 28-pin medium size package. Figure 1.7 shows the pin configuration of this microcontroller.

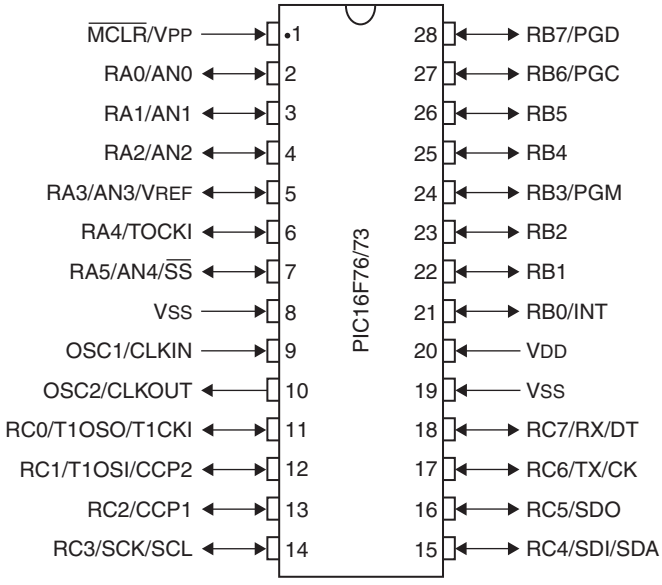


Figure 1.7: PIC16F73 Microcontroller Pin Configuration

1.3 16-bit Instruction Word

The 16-bit microcontrollers are at the high end of the PIC microcontroller family. Most of the devices in this group can operate at up to 40 MHz, have 33 I/O pins, and three timers. They have 23 instructions in addition to the 35 instructions found on the 14-bit microcontrollers. Table 1.4 lists some of the devices in this family.

All memory for the PIC microcontroller family is internal and it is usually not very easy to extend this memory externally. No special hardware or software features are provided for

Table 1.4: Some 16-bit PIC Microcontrollers

Microcontroller	Program Memory	Data RAM	Max Speed (MHz)	I/O Ports	A/D Converter
17C43	4096 × 16	454	33	33	–
17C752	8192 × 16	678	33	50	12
18C242	8192 × 16	512	40	23	5
18C252	16384 × 16	1536	40	23	5
18F4520	32768 × 16	1536	40	36	13

extending either the program memory or the data memory. The program memory is usually sufficient for small to medium size projects. But the data memory is generally small and may not be enough for medium to large projects unless a bigger and more expensive member of the family is chosen. For some large projects even this may not be enough and the designer may have to sacrifice the I/O ports to interface an external data memory, or to choose a microcontroller from a different manufacturer.

1.4 Inside a PIC Microcontroller

Although there are many models of microcontrollers in the PIC family, they all share some common features, such as program memory, data memory, I/O ports, and timers. Some devices have additional features such as A/D converters, USARTs and so on. Because of these common features, we can look at these attributes and cover the operation of most devices in the family.

1.4.1 Program Memory (Flash)

The program memory is where your program resides. In early microprocessors and microcontrollers the program memory was EPROM, which meant that it had to be erased using UV light before it could be reprogrammed. Most PIC microcontrollers nowadays are based on flash technology, where the memory chip can be erased or reprogrammed using a programmer device. Most PIC microcontrollers can also be programmed without removing them from their circuits. This process (called in-circuit serial programming, or ISP) speeds up the development cycle and lowers the development costs. Although the program memory is mainly used to store a program, there is no reason why it cannot be used to store constant data used in programs.

PIC microcontrollers can have program memories from 0.5 to over 16 K. A PicBasic program can have several pages of code and still fit inside 1 K of program memory. The width of a 14-bit program memory is actually 14 bits. It is interesting to note that PICs are known as 8-bit microcontrollers. This is actually true as far as the width of the data memory is concerned, which is 8 bits wide. Microchip calls the 14 bits a *word*, even though a *word* is actually 16 bits wide.

When power is applied to the microcontroller or when the MCLR input is lowered to logic 0, execution starts from the Reset Vector, which is the first word of the program memory. Thus, the first instruction executed after a reset is the one located at address 0 of the program memory. When the program is written in assembler language, the programmer has to use special instructions (called ORG) so that the first executable instruction is loaded into address 0 of the program memory. High-level languages such as PicBasic or PicBasic Pro compile your program such that the first executable statement in your program is loaded into the first location of the program memory.

1.4.2 Data Memory (RAM)

The data memory is used to store all of your program variables. This is a RAM, which means that all the data is lost when power is removed. The data memory is 8 bits wide and this is why the PIC microcontrollers are called 8-bit microcontrollers.

The data memory in a PIC microcontroller consists of banks, with some models having only two banks, some models four banks, and so on. A required bank of the data memory can be selected under program control.

1.4.3 Register File Map and Special Function Registers

Register File Map (RFM) is a layout of all the registers available in a microcontroller and this is extremely useful when programming the device, especially when using assembler language. The RFM is divided into two parts: the *Special Function Registers* (SFR), and the *General Purpose Registers* (GPR). For example, on a PIC16F84 microcontroller there are 68 GPR registers and these are used to store temporary data.

SFR is a collection of registers used by the microcontroller to control the internal operations of the device. Depending upon the complexity of the devices, the number of registers in the SFR varies. It is important that the programmer understands the functions of the SFR registers fully since they are used both in assembly language and in high-level languages.

Depending on the model of PIC microcontroller used, there could be other registers. For example, writing and reading from the EEPROM are controlled by SFR registers EECON1, EECON2, EEADR, and EEDATA. Fortunately, PicBasic and PicBasic Pro compilers provide simple high-level instructions for writing to and reading from the EEPROM and thus you do not need to know how to load these registers if you are programming in these languages.

Some of the important SFR registers that you may need to configure while programming using a high-level language are

- OPTION register
- I/O registers
- Timer registers
- INTCON register
- A/D converter registers

The functions and the bit definitions of these registers are described in detail in the following sections.

1.4.3.1 OPTION Register

This register is used to set up various internal features of the microcontroller and is named as OPTION_REG. This is a readable and writable register containing various control bits

7	6	5	4	3	2	1	0
RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0

Bit 7: PORTB Pull-up Enable
1: PORTB pull-ups disabled
0: PORTB pull-ups enabled

Bit 6: INT Interrupt Edge Detect
1: Interrupt on rising edge of INT input
0: Interrupt on falling edge of INT input

Bit 5: TMR0 Clock Source
1: T0CK1 pulse
0: Internal oscillator

Bit 4: TMR0 Source Edge Select
1: Increment on HIGH to LOW of T0CK1
0: Increment on LOW to HIGH of T0CK1

Bit 3: Prescaler Assignment
1: Prescaler assigned to Watchdog Timer
0: Prescaler assigned to TMR0

Bit 2-0: Prescaler Rate

000	1:2
001	1:4
010	1:8
011	1:16
100	1:32
101	1:64
110	1:128
111	1:256

Figure 1.8: OPTION_REG Bit Definitions

to configure the on-chip timer and the watchdog timer. This register is at address 81 (hexadecimal) of the microcontroller and its bit definitions are given in Figure 1.8. The OPTION REG register is also used to control the external interrupt pin RB0. This pin can be set up to generate an interrupt—for example, when it changes from logic 0 to logic 1. The microcontroller then suspends the main program execution and jumps to the interrupt service routine (ISR) to service the interrupt. Upon return from the interrupt, normal processing resumes.

For example, to configure the INT pin so that external interrupts are accepted on the rising edge of the INT pin, the following bit pattern should be loaded into the OPTION_REG:

X1XXXXXX

where X is a don't care bit and can be a 0 or a 1. We shall see later how to configure various bits of this register.

1.4.3.2 I/O Registers

These registers are used for the I/O control. Every I/O port in the PIC microcontroller has two registers: *port data register* and *port direction control register*.

Port data register has the same name as the port it controls. For example, the PIC16F84 microcontroller has two port data registers, PORTA and PORTB. A PIC16F877 microcontroller has five port data registers, PORTA, PORTB, PORTC, PORTD, and PORTE. Eight bits of data can be sent to any port, or 8 bits of data can be read from the ports. It is also possible to read or write to individual port pins. For example, any bit of a given port can be set or cleared, or data can be read from one or more port pins at the same time.

Ports in a PIC microcontroller are bidirectional. Thus, each pin of a port can be used as an input or an output pin. Port direction control register configures the port pins as either inputs or outputs. This register is called the TRIS register and every port has a TRIS register named after its port name. For example, TRISA is the direction control register for PORTA. Similarly, TRISB is the direction control register for PORTB and so on.

Setting a bit in the TRIS register makes the corresponding port register pin an input. Clearing a bit in the TRIS register makes the corresponding port pin an output. For example, to make bits 0 and 1 of PORTB input and the other bits output, we have to load the TRISB register with the bit pattern.

00000011

Figure 1.9 shows the TRISB register and the direction of PORTB pins.

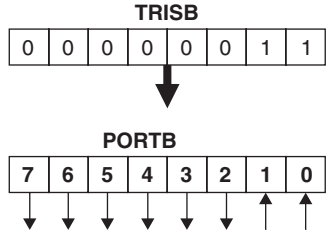


Figure 1.9: TRISB and PORTB Direction

Note that port data register and port direction control registers can be accessed directly using the PicBasic Pro compiler. For example, as we shall see later, TRISB register can be set to 3 and data can be read from PORTB into a variable named CNT by the PicBasic Pro instructions:

```
TRISB = 3
CNT = PORTB
```

The PicBasic compiler has no direct register control instructions and, as we shall see later, we have to use the PEEK and POKE instructions. PEEK is used to read data from a register and POKE is used to send data to a register.

When we use the PEEK and POKE instructions, we have to specify the register address of the register we wish to access. The register addresses of port registers are (the "$" character specifies that the number is in hexadecimal format):

Ports	Address (Hexadecimal)
PORTA	$05
PORTB	$06
PORTC	$07
PORTD	$08
PORTE	$09
TRISA	$85
TRISB	$86
TRISC	$87
TRISD	$88
TRISE	$89

Thus, for the above example, the required PicBasic instructions will be

```
POKE $86, 3
PEEK $06, CNT
```

1.4.3.3 Timer Registers

Depending on the model used, some PIC microcontrollers have only one timer, and some may have up to three timers. In this section we shall look at the PIC16F84 microcontroller, which has only one timer. The extension to several timers is similar and we shall see in the projects section how to use more than one timer.

The timer in the PIC16F84 microcontroller is an 8-bit register (called TMR0), which can be used as a timer or a counter. When used as a counter, the register increments each time a clock pulse is applied to pin T0CK1 of the microcontroller. When used as a timer, the register increments at a rate determined by the system clock frequency and a prescaler selected by register OPTION_REG. Prescaler rates vary from 1:2 to 1:256. For example, when using a 4-MHz clock, the basic instruction cycle is 1 μs (the clock is internally divided by four). If we select a prescaler rate of 1:16, the counter will be incremented at every 16 μs.

The TMR0 register has address 01 in the RAM.

A timer interrupt is generated when the timer overflows from 255 to 0. This interrupt can be enabled or disabled by our program. Thus, for example, if we need to generate interrupts at intervals of 200 μs using a 4-MHz clock, we can select a prescaler value of 1:4 and enable timer interrupts. The timer clock rate is then 4 μs. For a time-out of 200 μs, we have to send 50 clocks to the timer. Thus, the TMR0 register should be loaded with 256 – 50 = 206—i.e., a count of 50 before an overflow occurs.

The watchdog timer's oscillator is independent from the CPU clock and the time-out is 18 ms. To prevent a time-out condition the watchdog must be reset periodically via software. If the watchdog timer is not reset before it times out, the microprocessor will be forced to jump to the reset address. The prescaler can be used to extend the time-out period and valid rates are 1, 2, 4, 8, 16, 32, 64, and 128. For example, when set to 128, the time-out period is about 2 s ($18 \times 128 = 2304$ ms). The watchdog timer can be disabled during programming of the device if it is not used.

Since the timer is a very important part of the PIC microcontrollers, more detailed information is given on its operation below.

1.4.3.4 TMR0 and Watchdog

TMR0 and a watchdog are found in nearly all PIC microcontrollers. Figure 1.10 shows the functional diagram of TMR0 and the watchdog circuit. The operation of the watchdog circuit is as described earlier and only the TMR0 circuit is described in this section.

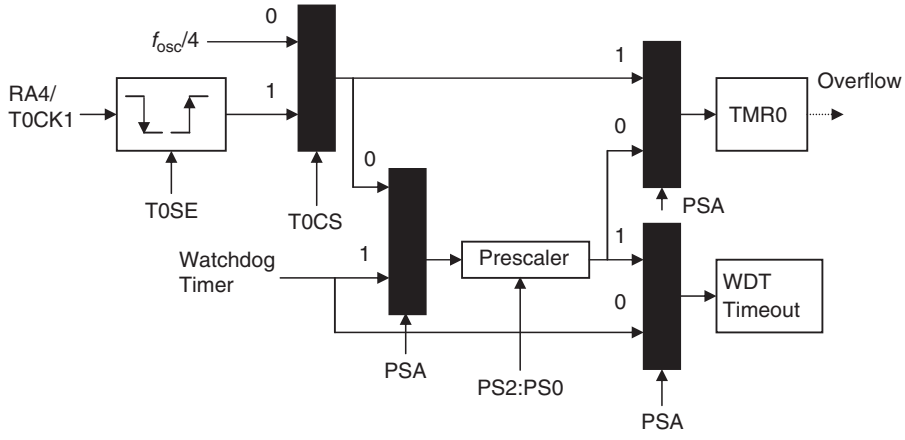


Figure 1.10: TMR0 and Watchdog Circuit

The source of input for TMR0 is selected by bit T0CS of OPTION_REG and it can be either from the microcontroller oscillator f_{osc} divided by 4, or it can be an external clock applied to the RA4/T0CK1 input. Here, we will only look at using the internal oscillator. If a 4-MHz crystal is used, the internal oscillator frequency is $f_{osc}/4 = 1$ MHz, which corresponds to

a period of T = $1/f = 10^{-6}$, or 1 μs. TMR0 is then selected as the source for the prescaler by clearing the PSA bit of OPTION_REG. The required prescaler value is selected by bits PS0 to PS2 as shown in Fig. 1.8. Bit PSA should then be cleared to 0 to select the prescaler for the timer. All the bits are configured now and the TMR0 register increments each time a pulse is applied by the internal oscillator. TMR0 register is 8 bits wide and it counts up to 255, then creates an overflow condition, and continues counting from 0. When TMR0 changes from 255 to 0 it generates a timer interrupt if timer interrupts and global interrupts are enabled (see INTCON register; an interrupt will be generated if GIE and TMR0 bits of INTCON are both set to 1). See Section 1.4.6 on Interrupts for more information.

By loading a value into the TMR0 register we can control the count until an overflow occurs. The formula given below can be used to calculate the time it will take for the timer to overflow (or to generate an interrupt) given the oscillator period, value loaded into the timer and the prescaler value.

$$\text{Overflow time} = 4 \times T_{\underline{\text{OSC}}} \times \text{Prescaler} \times (256 - \text{TMR0}) \tag{1.1}$$

where

Overflow time is in μs
$T_{\underline{\text{OSC}}}$ is the oscillator period in μs
Prescaler is the prescaler value chosen using OPTION_REG
TMR0 is the value loaded into TMR0 register

For example, assume that we are using a 4-MHz crystal, and the prescaler is chosen as 1:8 by setting bits PS2:PS0 to "010". Also assume that the value loaded into the timer register TMR0 is decimal 100. The overflow time is then given by

$$4\,\text{MHz clock has a period, } T = 1/f = 0.25\,\mu\text{s}$$

Using the above formula,

$$\text{Overflow time} = 4 \times 0.25 \times 8 \times (256 - 100) = 1248\,\mu\text{s}.$$

Thus, the timer will overflow after 1.248 μs and a timer interrupt will be generated if the timer interrupt and global interrupts are enabled.

What we normally need is to know what value to load into the TMR0 register for a required Overflow time. This can be calculated by modifying Eq. (1.1) as

$$\text{TMR0} = 256 - (\text{Overflow time})/(4 \times T_{\underline{\text{OSC}}} \times \text{Prescaler}) \tag{1.2}$$

For example, suppose that we want an interrupt to be generated after 500 μs and the clock and the prescaler values are as before. The value to be loaded into the TMR0 register can be calculated using Eq. (1.2) as

$$\text{TMR0} = 256 - 500/(4 \times 0.25 \times 8) = 193.5$$

The nearest number we can load into the TMR0 register is 193.

Table 1.5 gives the values that should be loaded into the TMR0 register for different Overflow times. In this table a 4-MHz crystal is assumed and the table gives the result as the prescaler value is changed from 2 to 256.

Table 1.5: Required TMR0 Values for Different Overflow Times

Time to Overflow (μs)	Prescaler							
	2	4	8	16	32	64	128	256
100	206	231	243	250	253	254	–	–
200	156	206	231	243	250	253	254	–
300	106	181	218	237	246	251	253	255
400	56	156	206	231	243	250	253	254
500	6	131	193	224	240	248	252	254
600	–	106	181	218	237	16	251	253
700	–	81	168	212	234	245	250	253
800	–	56	156	206	231	243	250	253
1,000	–	6	131	193	225	240	248	252
5,000	–	–	–	–	100	178	77	236
10,000	–	–	–	–	–	100	178	217
20,000	–	–	–	–	–	–	100	178
30,000	–	–	–	–	–	–	–	139
40,000	–	–	–	–	–	–	–	100
50,000	–	–	–	–	–	–	–	60
60,000	–	–	–	–	–	–	–	21

1.4.3.5 TMR1

Although TMR0 is the basic timer found in nearly all PIC microcontrollers, some devices have several timers, such as TMR0, TMR1, and TMR2. Additional timers give added functionality to a microcontroller. In this section the operation of TMR1 will be described in detail.

TMR1 is a 16-bit timer, consisting of two 8-bit registers TMR1H and TMR1L. As shown in Fig. 1.11, a prescaler is used with TMR1 and the available prescaler values are only 1, 2, 4, and 8.

Register T1CON controls the operation of TMR1. The bit definition of this register is shown in Fig. 1.12. TMR1 can operate either as a timer or as a counter, selected by bit TMR1CS of T1CON. When operated in timer mode, TMR1 increments every oscillator frequency $f_{osc}/4$.

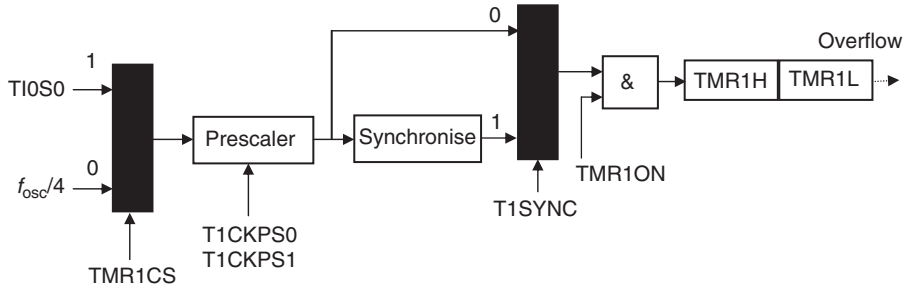


Figure 1.11: TMR1 Structure

7	6	5	4	3	2	1	0
–	–	TICKPS1	TICKPS0	TIOSCEN	TISYNC	TMR1CS	TMR1ON

Bit 7: Unused

Bit 6: Unused

Bit 5–4: Timer1 Input Clock Prescale Select Bits
- 11 1:8 prescale value
- 10 1:4 prescale value
- 01 1:2 prescale value
- 00 1:1 prescale value

Bit 3: Timer1 Oscillator Enable Bit
- 1: Oscillator is enabled
- 0: Oscillator is disabled

Bit 2: Timer1 External Clock Input Synchronization Select Bit
- When TMR1CS = 1:
- 1: Do not synchronize external clock input
- 0: Synchronize external clock input
- When TMR1CS = 0:
- This bit is ignored. Timer1 uses internal clock

Bit 1: Timer1 Clock Source Select Bit
- 1: External clock from pin TIOSO (on rising edge)
- 0: Internal clock ($f_{osc}/4$)

Bit 0: Timer1 On Bit
- 1: Enable Timer1
- 0: Stops Timer1

Figure 1.12: T1CON Bit Definitions

TMR1 can be enabled or disabled by setting or clearing control bit TMR1ON. TMR1 can count from 0 to 65,535 and it generates an overflow when changing from 65,535 to 0. A timer interrupt is generated if the TMR1 interrupt enable bit TMR1IE is enabled and also the global interrupts are enabled by register INTCON.

When TMR1 is operated in counter mode, it increments on every rising edge (from logic 0 to logic 1) of the clock input.

1.4.3.6 TMR2

TMR2 is an 8-bit timer with a prescaler and a postscaler and it has an 8-bit period register PR2. This timer is controlled by register T2CON whose bit definitions are given in Fig. 1.13. The prescaler options are 1, 4, and 16 only and are selected by T2CKPS1 and T2CKPS0 bits of T2CON. TMR2 increments from 0, until it matches PR2, and then resets to 0 on the next cycle. Then the cycle is repeated. TMR2 can be shut off by clearing TMR2ON of T2CON register to minimize power consumption.

7	6	5	4	3	2	1	0
–	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0

Bit 7: Unused

Bit 6-3: Timer2 Output Postscale Select Bits
- 0000 1:1 Postscale
- 0001 1:2 Postscale
- 0010 1:3 Postscale
- ....
- ....
- 1111 1:16 Postscale

Bit 2: Timer2 On Bit
- 1: Timer2 is On
- 0: Timer2 is Off

Bit 1-0: Timer2 Clock Prescale Select Bits
- 00 Prescaler is 1
- 01 Prescaler is 4
- 10 Prescaler is 16
- 11 Prescaler is 16

Figure 1.13: T2CON Bit Definitions

1.4.3.7 INTCON Register

This is the interrupt control register. This register is at address 0 and 8B (hexadecimal) of the microcontroller RAM and the bit definitions are given in Fig. 1.14. For example, to enable interrupts so that external interrupts from pin INT (RB0) can be accepted on a PIC16F84, the following bit pattern should be loaded into register INTCON:

1XX1XXXX

Similarly, to enable timer interrupts, bit 5 of INTCON must be set to 1.

7	6	5	4	3	2	1	0
GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF

Bit 7: Global Interrupt Enable
1: Enable all un-masked interrupts
0: Disable all interrupts

Bit 6: EE Write Complete Interrupt
1: Enable EE write complete interrupt
0: Disable EE write complete interrupt

Bit 5: TMR0 Overflow Interrupt
1: Enable TMR0 interrupt
0: Disable TMR0 interrupt

Bit 4: INT External Interrupt
1: Enable INT External Interrupt
0: Disable INT External Interrupt

Bit 3: RB Port Change Interrupt
1: Enable RB port change interrupt
0: Disable RB port change interrupt

Bit 2: TMR0 Overflow Interrupt Flag
1: TMR0 has overflowed
0: TMR0 did not overflow

Bit 1: INT Interrupt Flag
1: INT interrupt occurred
0: INT interrupt did not occur

Bit 0: RB Port Change Interrupt Flag
1: One or more of RB4-RB7 pins changed state
0: None of RB4-RB7 changed state

Figure 1.14: INTCON Register Bit Definitions

1.4.3.8 A/D Converter Registers

The A/D converter is used to interface analog signals to the microcontroller. The A/D converts analog signals (e.g., voltage) into digital form so that they can be connected to a computer. A/D converter registers are used to control the A/D converter ports. On most PIC microcontrollers equipped with A/D, PORTA pins are used for analog input and these port pins are shared between digital and analog functions.

PIC16F876 includes five A/D converters. Similarly, PIC16F877 includes eight A/D converters. There is actually only one A/D converter, as shown in Fig. 1.15, and the inputs are multiplexed and they share the same converter. The width of the A/D converter can be 8 bits or 10 bits. Both PIC16F876 and PIC16F877 have 10-bit converters. PIC16F73 has 8-bit converters. An A/D converter requires a reference voltage to operate. This reference voltage is

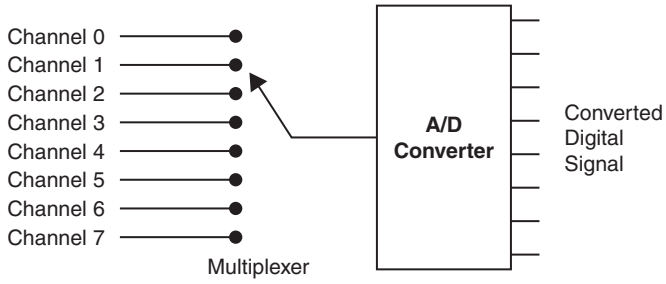


Figure 1.15: Multiplexed A/D Structure

chosen by programming the A/D converter registers and is typically 5 V. Thus, if we are using a 10-bit converter (1024 quantization levels), the resolution of our converter will be 5/1024 = 0.00488 V, or 4.88 mV; i.e., we can measure analog voltages with a resolution of 4.88 mV. For example, if the measured analog input voltage is 4.88 mV we get the 10-bit digital number "0000000001"; if the analog input voltage is $2 \times 4.88 = 9.76$ mV, the 10-bit converted number will be "0000000010"; if the analog input voltage is $3 \times 4.88 = 14.64$ mV, the converted number will be "0000000011"; and so on.

In a similar way, if the reference voltage is 5 V and we are using an 8-bit converter (256 quantization levels), the resolution of the converter will be 5/256 = 19.53 mV. For example, if the measured input voltage is 19.53 mV we get the 8-bit number "00000001"; if the analog input voltage is $2 \times 19.53 = 39.06$ mV we get the 8-bit number "00000010"; and so on.

The A/D converter is controlled by registers ADCON0 and ADCON1. The bit pattern of ADCON0 is shown in Fig. 1.16. ADCON0 is split into four parts; the first part consists of the highest two bits ADCS1 and ADCS0 and they are used to select the conversion clock. The internal RC oscillator or the external clock can be selected as the conversion clock as in the following table:

00	External clock/2
01	External clock/8
10	External clock/32
11	Internal RC clock

The second part of ADCON0 consists of the three bits CHS2, CHS1, and CHS0. These are the channel select bits, and they select which input pin is routed to the A/D converter. The selection is as follows:

CHS2:CHS1:CHS0	
000	Channel 0
001	Channel 1
010	Channel 2

7	6	5	4	3	2	1	0
ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/DONE	–	ADON

Bit 7-6: A/D Converter Clock Select
- 00 $f_{osc}/2$
- 01 $f_{osc}/8$
- 10 $f_{osc}/32$
- 11 Internal RC oscillator

Bit 5-3: A/D Channel Select
- 000 Channel 0
- 001 Channel 1
- 010 Channel 2
- 011 Channel 3
- 100 Channel 4
- 101 Channel 5
- 110 Channel 6
- 111 Channel 7

Bit 2: GO/DONE Bit
- 1: Start conversion
- 0: A/D conversion is complete

Bit 1: Not used

Bit 0: ADON Bit
- 1: Turn ON A/D circuit
- 0: Turn OFF A/D circuit

Figure 1.16: ADCON0 Bit Definitions

011 Channel 3
100 Channel 4
101 Channel 5
110 Channel 6
111 Channel 7

The third part of ADCON0 is the single GO/DONE bit. This bit has two functions: first, by setting the bit it starts the A/D conversion. Second, the bit is cleared when the conversion is complete and this bit can be checked to see whether or not the conversion is complete.

The fourth part of ADCON0 is also a single bit, ADON, which is set to turn on the A/D converter circuitry.

ADRESH and ADRESL are the A/D converter result registers. ADRESL is the low byte and ADRESH is the upper 2 bits (if a 10-bit converter is used). We shall see how to configure the result of the conversion later.

ADCON1 is the second A/D control register. This register controls the format of converted data and mode of the PORTA inputs. The bit format of this register is shown in Fig. 1.17.

Bit 7 is called ADFM and when this bit is 0 the result of the A/D conversion is left justified; when it is 1, the result of the A/D conversion is right justified. If we have an 8-bit converter, we can clear ADFM and just read ADRESH to get the 8-bit converted data. If we have a 10-bit converter, we can set ADFM to 1 and the 8 bits of the result will be in ADRESL, and 2 bits of the result will be in the lower bit positions of ADRESH. The remaining 6 positions of ADRESH (bit 2 to bit 7) will be cleared to zero.

Bits PCFG0-3 control the mode of PORTA pins. As seen in Fig. 1.17, a PORTA pin can be programmed to be a digital pin or an analog pin. For example, if we set PCFG0-3 to "0110" then all PORTA pins will be digital I/O pins. PCFG0-3 bits can also be used to define the reference voltage for the A/D converter. As we shall see in the projects section of the book, the reference voltage Vref is usually set to be equal to the supply voltage (Vdd), and Vref is set to be equal to Vss. This causes the A/D reference voltage to be +5 V.

1.4.4 Oscillator Circuits

An oscillator circuit is used to provide a microcontroller with a clock. A clock is needed so that the microcontroller can execute a program.

7	6	5	4	3	2	1	0
ADFM	–	–	–	PCFG3	PCFG2	PCFG1	PCFG0

Bit 7: A/D Converter Result Format Select
1: A/D converter output is right justified
0: A/D converter output is left justified

Bit 6: Not used

Bit 5: Not used

Bit 4: Not used

Bit 3-0: Port Assignment and Reference Voltage Selection
(see Figure 1.17)

PIC microcontrollers have built-in oscillator circuits and this oscillator can be operated in one of five modes.

- LP—Low-power crystal
- XT—Crystal/resonator
- HS—High-speed crystal/resonator
- RC resistor–capacitor
- No external components (only on some PIC microcontrollers).

PCFG3-PCFG0	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0	Vref+	Vref−
0000	A	A	A	A	A	A	A	A	Vdd	Vss
0001	A	A	A	A	Vref+	A	A	A	RA3	Vss
0010	D	D	D	A	A	A	A	A	Vdd	Vss
0011	D	D	D	A	Vref+	A	A	A	RA3	Vss
0100	D	D	D	D	A	D	A	A	Vdd	Vss
0101	D	D	D	D	Vref+	D	A	A	RA3	Vss
0110	D	D	D	D	D	D	D	D	Vdd	Vss
0111	D	D	D	D	D	D	D	D	Vdd	Vss
1000	A	A	A	A	Vref+	Vref−	A	A	RA3	RA2
1001	D	D	A	A	A	A	A	A	Vdd	Vss
1010	D	D	A	A	Vref+	A	A	A	RA3	Vss
1011	D	D	A	A	Vref+	Vref−	A	A	RA3	RA2
1100	D	D	D	A	Vref+	Vref−	A	A	RA3	RA2
1101	D	D	D	D	Vref+	Vref−	A	A	RA3	RA2
1110	D	D	D	D	D	D	D	A	Vdd	Vss
1111	D	D	D	D	Vref+	Vref	D	A	RA3	RA2

Figure 1.17: ADCON1 Bit Definitions

In LP, XT, or HS modes, an external oscillator can be connected to the OSC1 input as shown in Fig. 1.18. This can be a crystal-based oscillator, or simple logic gates can be used to design an oscillator circuit.

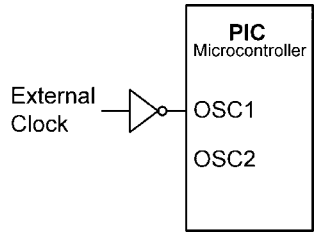


Figure 1.18: Using an External Oscillator

1.4.4.1 Crystal Operation

As shown in Fig. 1.19, in this mode of operation an external crystal and two capacitors are connected to the OSC1 and OSC2 inputs of the microcontroller. The capacitors should be chosen as in Table 1.6. For example, with a crystal frequency of 4 MHz, two 22-pF capacitors can be used.

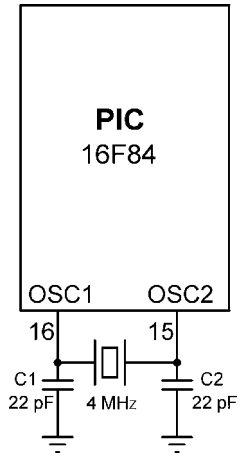


Figure 1.19: Crystal Oscillator Circuit

Table 1.6: Capacitor Selection for Crystal Operation

Mode	Frequency	C1, C2
LP	32 kHz	68–100 pF
LP	200 kHz	15–33 pF
XT	100 kHz	100–150 pF
XT	2 MHz	15–33 pF
XT	4 MHz	15–33 pF
HS	4 MHz	15–33 pF
HS	10 MHz	15–33 pF

1.4.4.2 Resonator Operation

Resonators are available from 4 to about 8 MHz. They are not as accurate as crystal-based oscillators. Resonators are usually 3-pin devices and the two pins at either side are connected to OSC1 and OSC2 inputs of the microcontroller. The middle pin is connected to the ground. Figure 1.20 shows how a resonator can be used in a PIC microcontroller circuit.

1.4.4.3 RC Oscillator

For applications where the timing accuracy is not important, we can connect an external resistor and a capacitor to the OSC1 input of the microcontroller as in Fig. 1.21. The oscillator frequency depends upon the values of the resistor and capacitor (see Table 1.7), the supply voltage, and to the temperature. For most applications, using a 5 K resistor with a 20-pF capacitor gives about 4 MHz and this may be acceptable in non-time-critical applications.

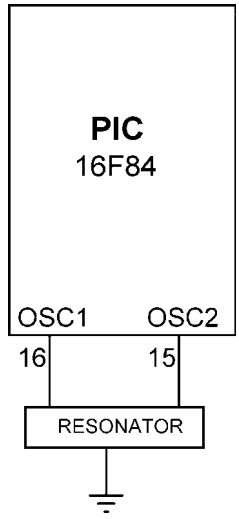


Figure 1.20: Resonator Oscillator Circuit

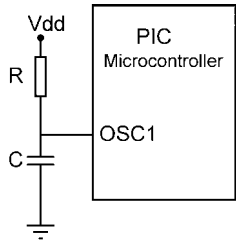


Figure 1.21: RC Oscillator Circuit

Table 1.7: RC Oscillator Component Selection

C	R	Frequency
20 pF	5 K	4.61 MHz
	10 K	2.66 MHz
	100 K	311 kHz
100 pF	5 K	1.34 MHz
	10 K	756 kHz
	100 K	82.8 kHz
300 pF	5 K	428 kHz
	10 K	243 kHz
	100 K	26.2 kHz

1.4.4.4 Internal Oscillator

Some PIC microcontrollers (e.g., PIC12C672 and PIC16F628) have built-in oscillator circuits and they do not require any external timing components. The built-in oscillator is usually set

to operate at 4 MHz and is selected during the programming of the device. For example, the PIC16F62X series of PIC microcontrollers can be operated with an internal resistor–capacitor-based 4-MHz oscillator (called mode INTRC). Additionally, a single resistor can be connected to pin RA7 of the microcontroller to create a variable oscillator frequency (called ER mode). For example, in the PIC16F62X microcontroller OSC1 and OSC2 pins are shared with the RA7 and RA6 pins, respectively. The internal oscillator frequency can be set by connecting a resistor to pin RA7 as shown in Fig. 1.22. Depending on the value of this resistance, the oscillator frequency can be selected from 200 kHz to 10.4 MHz (see Table 1.8). When used in this mode, pin RA7 is not available as a digital I/O pin.

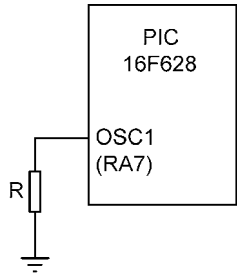


Figure 1.22: Changing the Internal Oscillator Frequency

Table 1.8: Resistor Value for the Internal Oscillator

Resistance	Frequency
0	10.4 MHz
1 K	10.0 MHz
10 K	7.4 MHz
20 K	5.3 MHz
47 K	3 MHz
100 K	1.6 MHz
220 K	800 kHz
470 K	300 kHz
1 M	200 kHz

The internal oscillator frequency of some microcontrollers (such as PIC16F630) can be calibrated so that more accurate timing pulses can be generated in time-critical applications (in serial communications, for example). In these microcontrollers an oscillator register called OSCCAL is used for the calibration of the oscillator frequency. A factory-calibrated oscillator constant is loaded into the last location of the memory. By copying this constant value into the oscillator register, we can have a more accurate 4-MHz clock frequency for our

microcontroller. It is also possible to modify the OSCCAL register values in order to fine-tune the oscillator frequency.

As a practical example, the following PicBasic and PicBasic Pro statements can be used to copy the oscillator calibration constant from the last memory location into the OSCCAL register. These commands must be declared at the beginning of the programs.

```
DEFINE OSCCAL_1 K 1     For 1 K core-size microcontrollers
DEFINE OSCCAL_2 K 1     For 2 K core-size microcontrollers
```

Note that the oscillator constant can be erased during the erasing of the program memory. You should make a note of the value at the last location of the program memory before erasing the memory. If this value is known it can be loaded directly into the OSCCAL register at the beginning of our programs, as shown below (here it is assumed that the constant is $24).

```
OSCCAL = $24
```

1.4.5 Reset Circuit

Reset is used to put the microcontroller into a known state. Normally when a PIC microcontroller is reset, execution starts from address 0 of the program memory. This is where the first executable user program resides. The reset action also initializes various SFR registers inside the microcontroller.

PIC microcontrollers can be reset when one of the following conditions occur:

- Reset during power on (POR – Power On Reset)
- Reset by lowering MCLR input to logic 0
- Reset when the watchdog overflows.

As shown in Fig. 1.23, a PIC microcontroller is normally reset when power is applied to the chip and when the MCLR input is tied to the supply voltage through a 4.7 K resistor.

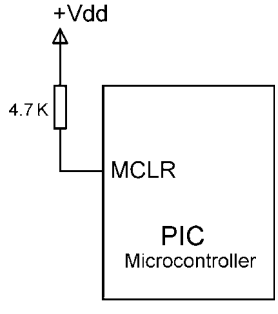


Figure 1.23: Using the Power on Reset

There are many applications where we want to reset the microcontroller, for instance by pressing an external button. The simplest circuit to achieve an external reset is shown in Fig. 1.24. In this circuit, the MCLR input is normally at logic 1 and the microcontroller is operating normally. When the reset button is pressed, this pin goes to logic 0 and the microcontroller is reset. When the reset button is released, the microcontroller starts executing from address 0 of the program memory.

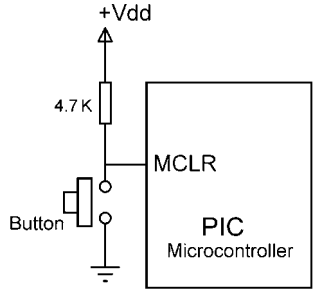


Figure 1.24: Using an External Reset Button

1.4.6 Interrupts

Interrupts are an important feature of all microcontrollers. An interrupt can either occur asynchronously or synchronously. Asynchronous interrupts are usually external events which interrupt the microcontroller and request service. For example, pin INT (RB0) of a PIC16F84 microcontroller is the external interrupt pin and this pin can be used to interrupt the microcontroller asynchronously; i.e., the interrupt can occur at any time independent of the program being executed inside the microcontroller. Synchronous interrupts are usually timer interrupts, such as the timer overflow generating an interrupt.

Depending on the model used, different PIC microcontrollers may have a different number of interrupt sources. For example, the PIC16F84 microcontroller has the following four sources of interrupts:

- External interrupt from INT (RB0) pin
- TMR0 interrupt caused by timer overflow
- External interrupt when the state of RB4, RB5, RB6, or RB7 pins change
- Termination of writing data to the EEPROM.

Interrupts are enabled and disabled by the INTCON register. Each interrupt source has two bits to control it. One enables interrupts, and the other one detects when an interrupt occurs. There is a common bit called GIE (see INTCON register bit definitions) which can be used to disable all sources of interrupts.

The INTCON control bits of various interrupt sources are

Interrupt Source	Enabled by	Completion Status
External interrupt from INT	INTE = 1	INTF = 1
TMR0 interrupt	T0IE = 1	T0IF = 1
RB4–RB7 state change	RBIE = 1	RBIF = 1
EEPROM write complete	EEIE = 1	–

Whenever an interrupt occurs, the microcontroller jumps to the ISR. On low-end microcontrollers (e.g., PIC16F84 or PIC16F628) all interrupt sources use address 4 in program memory as the start of the ISR. Because all interrupts use the same ISR address, we have to check the interrupt completion status to detect which interrupt has occurred when multiple interrupts are enabled.

The completion status has to be cleared to zero if we want the same interrupt source to be able to interrupt again.

Assuming that we wish to use the external interrupt (INT) input, and interrupts should be accepted on the low to high transition of the INT pin, the steps before and after an interrupt are summarized below.

- Set the direction of the external interrupt to be on rising edge by setting INTEDG = 1 in register OPTION_REG.
- Enable INT interrupts by setting INTE = 1 in register INTCON.
- Enable global interrupts by setting GIE = 1 in register INTCON.
- Carry out normal processing. When interrupt occurs, program will jump to the ISR.
- Carry out the required tasks in the ISR routine.
- At the end of the ISR, re-enable the INT interrupts by clearing INTF = 0.

1.4.7 The Configuration Word

PIC microcontrollers have a special register called the *Configuration Word*. This is a 14-bit register and is mapped in program memory 2007 (hexadecimal). This address is beyond the user program-memory space and cannot be directly accessed in a program. This register can be accessed during the programming of the microcontroller.

The configuration word stores the following information about a PIC microcontroller:

- Code protection bits: These bits are used to protect blocks of memory so that they cannot be read.
- Power-on timer enable bit.
- Watchdog (WDT) timer enable bit.

- Oscillator selection bits: The oscillator can be selected as XT, HS, LP, RC, or internal (if supported by the microcontroller).

For example, in a typical application we can have the following configuration word selection during the programming of the microcontroller:

- Code protection OFF
- XT oscillator selection
- WDT disabled
- Power-up timer enables.

1.4.8 I/O Interface

A PIC microcontroller port can source and sink 25 mA of current. When sourcing current, the current is flowing out of the port pin, and when sinking current, the current is flowing into the pin. When the pin is sourcing current, one pin of the load is connected to the microcontroller port and the other pin to the ground (see Fig. 1.25a). The load is then energized when the port output is at logic 1. When the pin is sinking current, one pin of the load is connected to the supply voltage and the other pin to the output of the port (see Fig. 1.25b). The load is then energized when the port output is at logic 0.

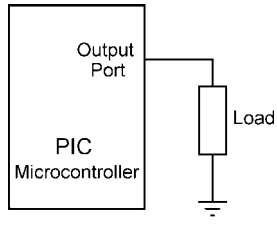


Figure 1.25a: Current Sourcing

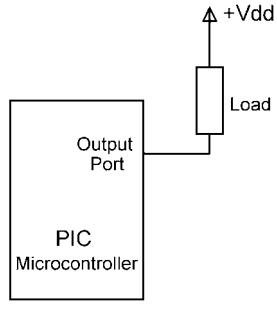


Figure 1.25b: Current Sinking

Some useful interface circuits are given in this section.

1.4.8.1 LED Interface

LEDs come in many different sizes, shapes, and colors. The brightness of an LED depends on the current through the device. Some small LEDs operate with only a few milliamperes of current, while standard size LEDs consume about 10 mA of current for normal brightness. Some very bright LEDs consume 15–20 mA of current. The voltage drop across an LED is about 2 V, but the voltage at the output of a microcontroller port is about 5 V when the port is at logic 1 level. As a result of this, it is not possible to connect an LED directly to a microcontroller output port. What is required is a resistor to limit the current in the circuit.

If the output voltage of the port is 5 V and the voltage drop across the LED is 2 V, we need to drop 3 V across the resistor. If we assume that the current through the LED is 10 mA, we can calculate the value of the required resistor as

$$R = \frac{5 - 2\,\text{V}}{10\,\text{mA}} = \frac{3\,\text{V}}{10\,\text{mA}} = 0.3\,\text{K}$$

The nearest physical resistor we can use is 330 Ω. Figure 1.26 shows how an LED can be connected to an output port pin in current source mode. In this circuit the LED will be ON when the port output is set to logic 1. Similarly, Fig. 1.27 shows how an LED can be connected to an output port pin in current sink mode. In this circuit the LED will be ON when the port output is at logic 0.

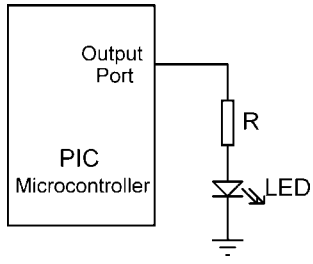


Figure 1.26: Connecting an LED in Current Source Mode

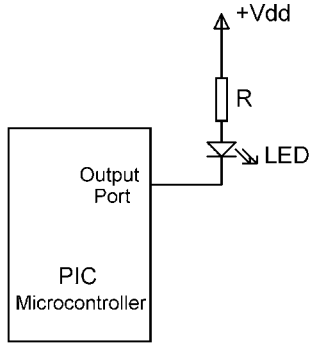


Figure 1.27: Connecting an LED in Current Sink Mode

1.4.8.2 Higher Current Load Interface

The circuits given in Figs 1.26 and 1.27 work fine for an LED, or for any other device whose current requirement is less than 25 mA. What do we do if we wish to operate a load with a higher current rating (e.g., a 12 V filament lamp)? The answer is that we have to use a switching device, such as a transistor or a relay.

Figure 1.28 shows how we can drive a small lamp from our port pin using a bipolar transistor. In this circuit, when the port output pin is at logic 1, current flows through the resistor and turns the transistor ON, effectively connecting the bottom end of the lamp to ground. It is important to realize that the positive supply to the lamp is not related to the PIC supply voltage and while the PIC is operating from 5 V, the lamp can be operated from a 12 V supply.The current capability depends upon the type of transistor used and several hundred milliamperes can be achieved with any type of small npn transistors. For higher currents, bipolar power transistors, or preferably MOSFET transistors, can be used.

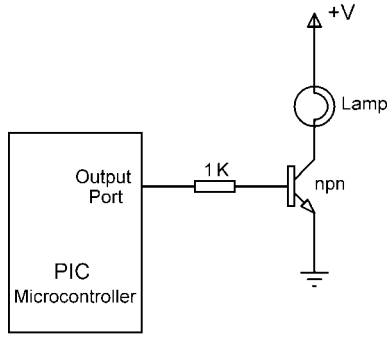


Figure 1.28: Driving a Lamp Using a Transistor

1.4.8.3 Relay Interface

When we want to switch inductive loads such as relays we have to use a diode in the circuit to prevent the transistor from being damaged (see Fig. 1.29). An inductive load can generate a

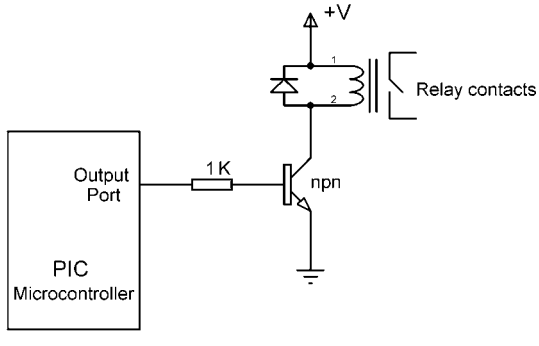


Figure 1.29: Driving an Inductive Load (e.g., a Relay)

back EMF which could easily damage a transistor. By connecting a diode in reverse bias mode this back EMF is dissipated without damaging the transistor.

Since we can drive a relay, we can connect any load to the relay outputs as long as we do not exceed the contact ratings of the relay. Figure 1.30 shows how a mains lamp can be operated from the microcontroller output port using a relay. The relay could also be operated using a MOSFET power transistor. In this circuit the mains lamp will turn ON when the output port of the microcontroller is a logic 1.

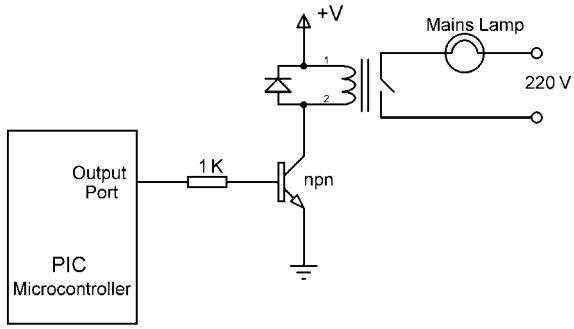


Figure 1.30: Driving a Mains Bulb Using a Relay

1.4.8.4 Button Input

One of the most common types of inputs is a button (a push-button switch) input where the user can change the state of an input pin by pressing a button. Basically, button input can use two different methods: active low and active high. As shown in Fig. 1.31, in active low implementation, the microcontroller input pin is connected to the supply voltage using a resistor (this is also called a pull-up resistor) and the button is connected between the port pin and ground. Normally the microcontroller input is pulled to logic 1 by the resistor. When the button is pressed, the input is forced to ground potential, which is logic 0. The change of state in the input pin can be determined by a program.

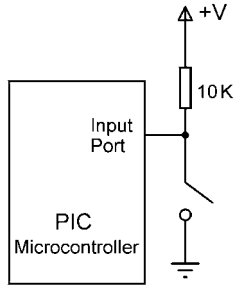


Figure 1.31: Active Low-Button Input

Some ports in PIC microcontrollers have internal pull-up resistors (e.g., PORTB) and these resistors can be enabled by clearing bit 7 (RBPU) of register INTCON to zero. When one of these port pins is used for button input, there is no need to use an external pull-up resistor and the button can simply be connected between the port pin and ground.

A button can also be connected in active high mode as shown in Fig. 1.32. In this configuration the button is connected between the supply voltage and the port pin. A resistor (this is also called a pull-down resistor) is connected between the port pin and ground. Normally, the port pin is at logic 0. When the button is pressed the port pin goes to the supply voltage, which is logic 1.

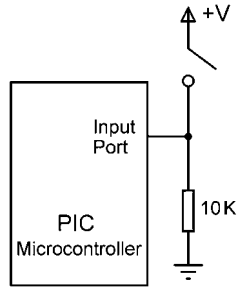


Figure 1.32: Active High-Button Input

One of the problems with mechanical switches is that, when a switch closes, its metal parts compress and relax and this causes the switch to open and close several times quickly. The problem is that the microcontroller can read the switch so fast that it can see the switch open and close during the bouncing of the metal parts and this can cause a wrong switch state to be read by the microcontroller. One way to eliminate this switch-bouncing problem is to delay reading the input after the switch state changes. For example, when we detect the switch is pressed, we may wait about 10 ms before we read the state of the switch.

In Figs 1.31 and 1.32, we have seen how simple buttons can be connected to a microcontroller port. It is also possible to connect a switching transistor to an input pin, the output of another IC, or simply the output of another PIC port pin. Figure 1.33 shows how a switching transistor can be connected as an input. In this circuit the transistor acts like an inverting switch. When the transistor input voltage is 0 V, the transistor is in OFF state and the port pin is at logic 1 level. When the transistor input voltage is 5 V the transistor turns ON and its collector-emitter voltage drops to 0 V, making the port pin logic 0. One nice thing about this circuit is that the transistor input voltage does not need to be 5 V to turn the transistor ON; it could easily be 9 or 12 V.

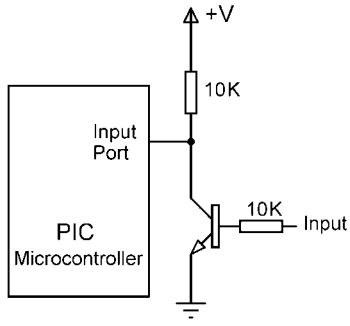


Figure 1.33: Transistor Input

The input ports of PIC microcontrollers are protected by internal diodes for over-voltage and under-voltage. Thus, the voltage on a pin can exceed the supply voltage, or it can go below the ground voltage, without causing any harm to the microcontroller. The RS232 serial communication lines operate with 12 V and we can usually connect these lines directly to the input ports using resistors without damaging the microcontroller.

CHAPTER 2

Introducing the PIC® 16 Series and the 16F84A

In Chapter 1 we surveyed the different PIC® microcontroller families that are available. We are now going to step up a gear and begin to look at the detail of the PIC 16 Series "mid-range" family. As an example device we will use the 16F84A, as it is a comparatively small member of the family. Just about everything we meet in the following chapter applies to all of the PIC 16 Series microcontrollers.

We will explore the overall architecture of the device and take time to get into some detail over its memory—both the technology and the memory maps.

In this chapter you will therefore learn about:

- The PIC 16 Series family, in overview
- The overall architecture of the 16F84A
- The 16F84A memory system, along with a review of memory technologies
- Other hardware features of the 16F84A, including the reset system.

If you wish, you will also learn about:

- Alternative approaches to microcontroller structure, through an example from another microcontroller family.

2.1 The Main Idea—the PIC 16 Series Family

2.1.1 A Family Overview

The PIC 16 Series family is growing rapidly, with a huge and almost bewildering diversity of members. Therefore, when we talk of "family" here, we are applying the concept of "extended family" and a very large one at that. Nevertheless, the 16 Series stays true to the concept that all family members have identical core and instruction sets, with the difference arising from different peripherals and other features being implemented, and different package sizes.

Table 2.1 gives summary details of some members of the 16 Series family. Even with this limitation, there is considerable diversity. Within the 16 Series extended family, we find groupings of very closely related controllers, of which two are represented here, the 16F84A and the 16F87XA. The 16F84A is listed first, with features we are about to explore in detail. It has a very close relative, the 16LF84A, whose extended supply voltage range allows operation at lower voltages. Either of these controllers is available in different packages, different operating temperature ranges, and different clock speed ranges. For example, the 16F84A is available in 4 and 20 MHz versions.

The 16F87XA is a diverse grouping, as can be seen. There are two package sizes and two memory sizes. It is easy to see that package size is driven by the number of input/outputs that are available. The 40-pin versions have five parallel ports (which translates to 33 lines of parallel digital input/output), as well as more analog input, compared with their 28-pin relatives. There is otherwise not much difference. Each package size, however, comes with two different memory sizes. The bigger memory of course gives the opportunity for longer programs and more data storage, but also costs a little more.

As is normal Microchip practice, each member of the 16 Series family has its own comprehensive data sheet, available from Microchip's website. Reference 2.1 is the data sheet for the 16F84A. As well as this, there is a manual covering all the features that are common to all members of the family [Ref. 2.2]. While it is not necessary to refer to these while reading this chapter, it is worth knowing they are there, and extremely useful for looking up the finer details of a microcontroller's design and use.

2.1.2 The 16F84A

The 16F84A, along with its direct predecessors, has been one of many PIC success stories. It first appeared as the 16C84. At a time when most microcontroller manufacturers were trying to make their products bigger, more sophisticated and more complex, Microchip made the bold decision to stay small, simple and easy to use. While many microcontrollers of the day did have on-chip program memory, it was usually EPROM (Erasable Programmable Read-Only Memory), with the attendant time-consuming EPROM erase cycle. With the 16C84, Microchip chose to use EEPROM (Electrically Erasable Programmable Read-Only Memory) for program memory. Thus, it could be programmed rapidly, and repeatedly changed. Then, as flash memory technology became more accessible, the 'C84 was reissued as the 16F84, with the new memory technology. With further upgrading it became the 16F84A. At the time of writing, this is the current version. A 16LF84A, intended for low-power applications, is also available.

2.1.3 A Caution on Upgrades

As technological expertise develops, any microcontroller design is inevitably upgraded. These are normally spelled out in documentation published by the manufacturer (e.g., Ref. 2.3).

Table 2.1: Some Members of the PIC 16 Series Family

Device number	No. of Pins*	Clock Speed	Memory (K = Kbytes, i.e. 1024 bytes)	Peripherals/Special Features
16F84A	18	DC to 20 MHz	1K program memory, 68 bytes RAM, 64 bytes EEPROM	1 8-bit timer 1 5-bit parallel port 1 8-bit parallel port
16LF84A	As above	As above	As above	As above, with extended supply voltage range
16F84A-04	As above	DC to 4 MHz	As above	As above
16F873A	28	DC to 20 MHz	4K program memory 192 bytes RAM, 128 bytes EEPROM	3 parallel ports, 3 counter/timers, 2 capture/compare/PWM modules, 2 serial communication modules, 5 10-bit ADC channels, 2 analog comparators
16F874A	40	DC to 20 MHz	4K program memory 192 bytes RAM, 128 bytes EEPROM	5 parallel ports, 3 counter/timers, 2 capture/compare/ PWM modules, 2 serial communication modules, 8 10-bit ADC channels, 2 analog comparators
16F876A	28	DC to 20 MHz	8K program memory 368 bytes RAM, 256 bytes EEPROM	3 parallel ports, 3 counter/timers, 2 capture/compare/PWM modules, 2 serial communication modules, 5 10-bit ADC channels, 2 analog comparators
16F877A	40	DC to 20 MHz	8K program memory 368 bytes RAM, 256 bytes EEPROM	5 parallel ports, 3 counter/timers, 2 capture/compare/PWM modules, 2 serial communication modules, 8 10-bit ADC channels, 2 analog comparators

* For DIP package only.
ADC, analog-to-digital converter; PWM, pulse width modulation.

While each upgrade is generally to be welcomed, the changes introduced need to be watched with care. Some are of obvious benefit. For example, the "A" version of the 16F84 can run at a higher speed than before (20 MHz maximum instead of 10 MHz). However, the technical upgrade sometimes has side effects. These are of no direct advantage and sometimes make it difficult to replace a microcontroller in an existing product with its upgraded version. For example, operating power supply voltages and logic input thresholds are different between the 'F84 and the 'F84A.

2.2 An Architecture Overview of the 16F84A

The pin connection diagram of the 16F84A is shown in Fig. 2.1 and its block diagram in Fig. 2.2. A comparison of these figures with the equivalent ones for the PIC 12F508 in Chapter 1 shows some interesting similarities and differences. With 18 pins in play, there isn't the intense pressure to squeeze several functions onto each pin. Separate and dedicated pins are now provided, for example, for clock oscillator (pins 15 and 16) and Reset (pin 4—**$\overline{\text{MCLR}}$**). Nevertheless, compared to most, the 'F84A remains a small microcontroller.

Architecturally there is clear similarity between the 12F508 and the 16F84A. In fact, the former is a direct subset of the 'F84A, with near identical CPU, memory, bus structure and counter/timer (TMR0) peripheral. Notice first, however, that the address bus sizes have been increased, to meet the needs of the whole PIC 16 Series family. As a smaller member of that family, the 'F84A doesn't fully exploit all these developments. The program address bus is now 13-bit and the instruction word size is 14-bit. Therefore, 2^{13} (i.e., 8192) memory locations *could* be addressed. Program memory size at 1 K is, however, only one-eighth of this. The larger bus size will, however, be useful in the larger 16 Series devices, as can be seen in the program memory size of the 16F876A and 16F877A (Table 2.1). RAM size has crept up cautiously to 68 locations and the Stack to eight locations.

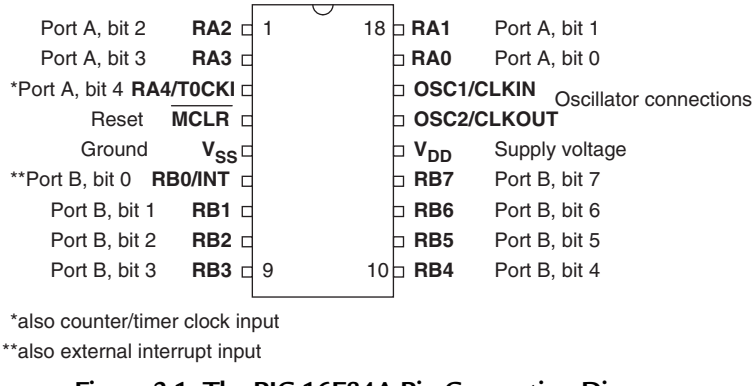


Figure 2.1: The PIC 16F84A Pin Connection Diagram

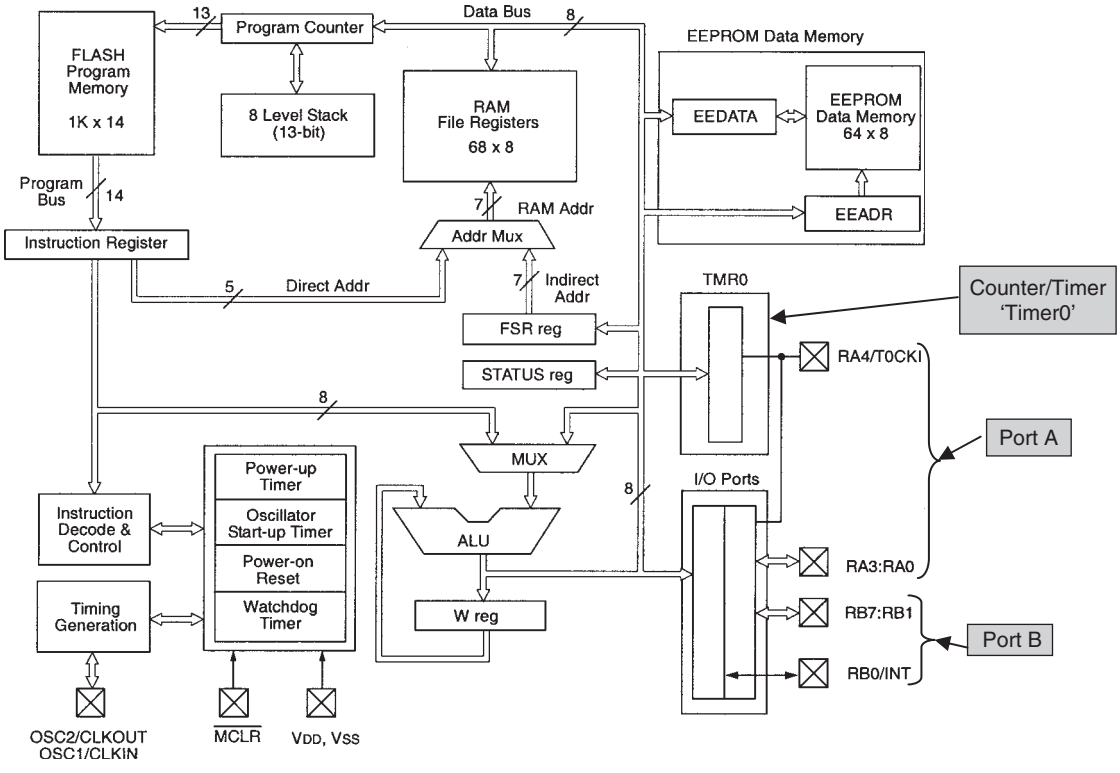


Figure 2.2: Block Diagram of the 16F84A (Supplementary Labels in Shaded Boxes Added by the Author)

A number of important new additions have appeared. The addition of an EEPROM memory gives the valuable capability of being able to store data values even when the chip is powered down. There are now two digital input/output ports. These are Port A, with five pins, and Port B, with eight. Importantly, there is the addition of an interrupt capability. This can be seen externally on pin 6, where bit 0 of Port B is shared with the external interrupt input. We will also see that there are three further internal interrupt sources, generated by the peripherals.

Overall, we have a microcontroller that, while only modestly more complex than the 12F508, has proved incredibly diverse and useful in small applications.

2.2.1 The Status Register

The result of any CPU operation is held in the Working register, but this does not necessarily tell everything about the operation which has just occurred. What if, for example, the 8-bit range has been exceeded by an addition instruction? The Working register has no way of indicating this and would simply hold an incorrect result. Therefore, a set of logic bits,

sometimes called *condition code* flags, is built in to any computer CPU. These are used to carry extra information about the result of the instruction most recently executed, for example whether the result is zero, negative or positive. For the 16F84A, these flags are held in the Status register, shown in Fig. 2.3. Only three of these Status register bits genuinely fall into the category of condition codes. These are bits 0 to 2, i.e. bits **C**, **DC** and **Z**. As the key to the figure shows, these indicate respectively whether a Carry or Digit Carry has been generated, or if the result is Zero.

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C
bit 7							bit 0

bit 7-6 **Unimplemented:** Maintain as "0"

bit 5 **RP0**: Register Bank Select bits (used for direct addressing)
01 = Bank 1 (80h - FFh)
00 = Bank 0 (00h - 7Fh)

bit 4 $\overline{\textbf{TO}}$: Time-out bit
1 = After power-up, CLRWDT instruction, or SLEEP instruction
0 = A WDT time-out occurred

bit 3 $\overline{\textbf{PD}}$: Power-down bit
1 = After power-up or by the CLRWDT instruction
0 = By execution of the SLEEP instruction

bit 2 **Z**: Zero bit
1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero

bit 1 **DC**: Digit Carry/$\overline{\text{borrow}}$ bit (ADDWF, ADDLW, SUBLW, SUBWF instructions) (for $\overline{\text{borrow}}$, the polarity is reversed)
1 = A carry-out from the 4th low order bit of the result occurred
0 = No carry-out from the 4th low order bit of the result

bit 0 **C**: Carry/$\overline{\text{borrow}}$ bit (ADDWF, ADDLW, SUBLW, SUBWF instructions) (for $\overline{\text{borrow}}$, the polarity is reversed)
1 = A carry-out from the Most Significant Bit of the result occurred
0 = No carry-out from the Most Significant Bit of the result occurred

Note: A subtraction is executed by adding the twos complement of the second operand. For rotate (RRF, RLF) instructions, this bit is loaded with either the high or low order bit of the source register.

Figure 2.3: The 16F84A Status Register

2.3 A Review of Memory Technologies

In order to examine the memory capabilities of the 16F84A, and to work with embedded systems in general, it is important to have some knowledge of the characteristics of the memory technologies in use. The following section gives just a brief overview of the different memory technologies currently used by Microchip.

An ideal memory reads and writes in negligible time, retains its stored value indefinitely, occupies negligible space and consumes negligible power. In practice no memory technology meets all these happy ideals! In general, different technologies are strong in one or more of these characteristics and weaker in others. There is not one best memory technology, and different technologies are therefore applied for different applications, according to the need.

Any memory is made up of an *array* of memory *cells*, where each cell holds one bit of data. The characteristics of the single cell reflect the characteristics of the overall array; therefore, each technology is described here simply in terms of its cell design.

2.3.1 Static RAM (SRAM)

Here each memory cell is designed as a simple flip-flop, using two back-to-back transistor pairs. Two further transistors allow the cell to connect into the main array. Data is held only as long as power is supplied. Hence the SRAM technology is volatile. With each cell taking six transistors, SRAM is not dense. However, if made from CMOS (Complementary Metal Oxide Semiconductor) it can be made to consume *very* little power and can retain its data down to a low voltage (around 2 V). It has thus been a popular technology in battery-powered systems. SRAM is mainly used for data memory (RAM) in a microcontroller.

2.3.2 EPROM (Erasable Programmable Read-Only Memory)

In this technology each memory cell is made of a single MOS transistor—but with a difference. Within the transistor there is embedded a "floating gate." Using a technique known as *hot electron injection* (HEI), the floating gate can be charged. When it is *not* charged, the transistor behaves normally and the cell output takes one logic state when activated. When it *is* charged, the transistor no longer works properly and it no longer responds when it is activated. The charge placed on the floating gate is totally trapped by the surrounding insulator. Hence, EPROM technology is nonvolatile. EPROM can, however, be erased by exposing it to intense ultraviolet light. This gives the trapped electrons the energy to leave the floating gate.

A special version of EPROM is OTP—One Time Programmable. Here the EPROM is packaged in plastic, without a window. Therefore, OTP can be programmed only once and never erased.

With a single transistor for a cell, EPROM is very high density and robust. Its requirement for a quartz window and ceramic packaging, to enable erasing, raises its price and reduces its flexibility. EPROM used to be integrated onto many microcontrollers for program memory, forcing the whole microcontroller to be ceramic-packaged, with a quartz window. As a technology, EPROM is now rapidly giving way to flash, which follows shortly.

2.3.3 EEPROM (Electrically Erasable Programmable Read-Only Memory)

EEPROM also uses floating gate technology. Its dimensions are finer, so that it can exploit another means of charging its floating gate. This is known as Nordheim Fowler tunneling (NFT). With NFT, it is possible to electrically erase the memory cell, as well as write to it. To allow this to happen, a number of switching transistors need to be included around the memory element itself, so the high density of EPROM is lost.

Generally, EEPROM can be written to and erased on a byte-by-byte basis. This makes it especially useful for storing single items of data, like television settings or mobile phone numbers. Both writing and erasing take finite time, up to several milliseconds, although a read can be accomplished at normal semiconductor memory access times, i.e., within microseconds or less. Again, like EPROM, because the charge on the floating gate is totally trapped by the surrounding insulator, EEPROM is nonvolatile. Because the EEPROM structure is now so fine, it suffers from certain wear-out mechanisms. Manufacturers usually therefore define a guaranteed minimum number of erase/write cycles that their memory can successfully undergo.

2.3.4 Flash

Flash represents a further evolution of floating gate technology. With a single transistor per memory cell, it uses both HEI and NFT to allow electrical writing and erase. It does not include the extra switch transistors that EEPROM has, so can only erase in blocks. It therefore returns to the exceptionally high density of EPROM. Like EEPROM, it has wear-out mechanisms, so cannot be written and erased indefinitely.

Apart from its inability to erase byte by byte, flash is an incredibly powerful technology. It is now a central feature of a huge range of products, including digital cameras, memory sticks, laptop computers and microcontroller program memory.

2.4 The 16F84A Memory

As Fig. 2.2 shows, there are no less than *four* areas of memory in the 16F84A, as summarized in Table 2.2. Each memory has its own distinct function and means of access.

2.4.1 The 16F84A Program Memory

The 16F84A program memory map is shown in Fig. 2.4. Looking at this diagram, we can see that it actually shows three things: the Program Counter, the Stack and the actual program memory. The three work inextricably together. The program memory is loaded with the program code that the microcontroller executes. The program is in the form of a list of instructions, and the Program Counter holds the address of the next instruction that is to be executed by the microcontroller. Therefore, it acts as a pointer to program memory, as indicated in the diagram. The value of the Program Counter can also be moved onto the Stack. This occurs when either a

Table 2.2: 16F84A Memory Features

Memory Function	Technology	Size	Volatile/Non-Volatile	Special Characteristics*
Program	Flash	1K × 14 bits	Non-volatile	10 000 erase/write cycles, typically
Data memory (file registers)	SRAM	68 bytes	Volatile	Retains data down to supply voltage of 1.5 V
Data memory (EEPROM)	EEPROM	64 bytes	Non-volatile	10 000 000 erase/write cycles, typically
Stack	SRAM	8 × 13 bits	Volatile	
* Information obtained from full 16F84A data sheet [Ref. 2.1].				

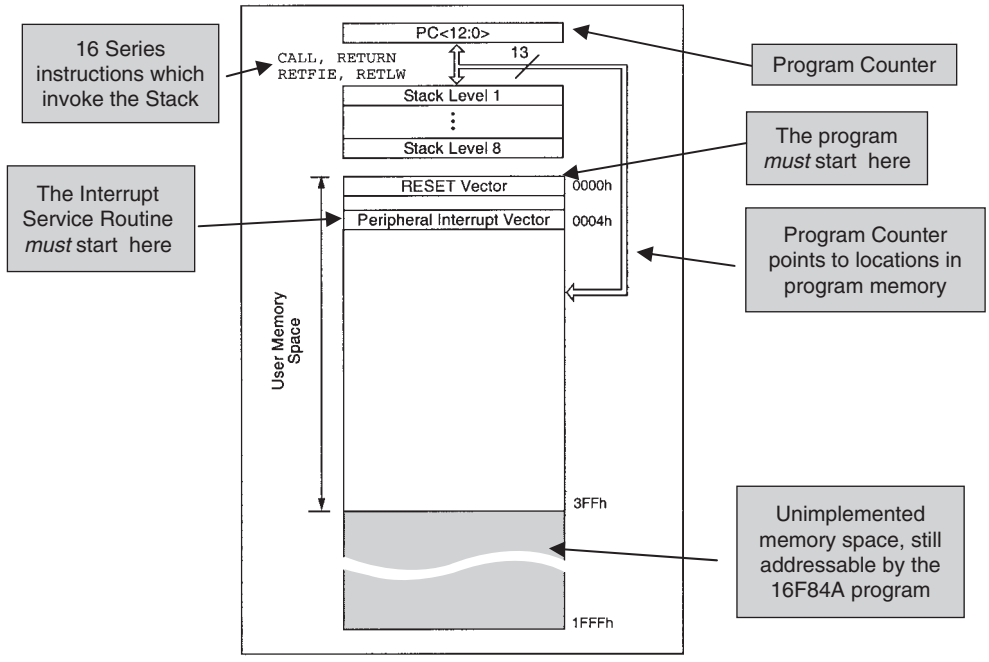


Figure 2.4: The 16F84A—Program Memory and Stack (Supplementary Labels in Shaded Boxes Added by the Author)

subroutine or an interrupt occurs. The instructions indicated in the diagram, **CALL**, **RETURN**, **RETFIE** and **RETLW**, all relate to subroutines and interrupts. We will meet them in the coming chapters—don't worry if they have no meaning to you at present!

We can see that the address range of program memory is from 0000 to $03FF_H$. With its 13-bit Program Counter, the microcontroller can theoretically address a range from 0000 to $1FFF_H$. The extra address space is shown (in grey), although it is of no use here.

The very first location in the program memory is labeled the *reset vector*. When the program starts running for the first time, for example on power-up, the Program Counter is set to 0000. Therefore, the first memory location that it points to is the reset vector. The programmer must therefore place his/her first instruction at this location. The *peripheral interrupt vector* acts in a similar way for interrupt service routines.

2.4.2 The 16F84A Data and Special Function Register Memory ('RAM')

The RAM memory map is shown in Fig. 2.5. The memory area is banked and is divided into two important areas. The first is the general-purpose data memory, which occupies locations

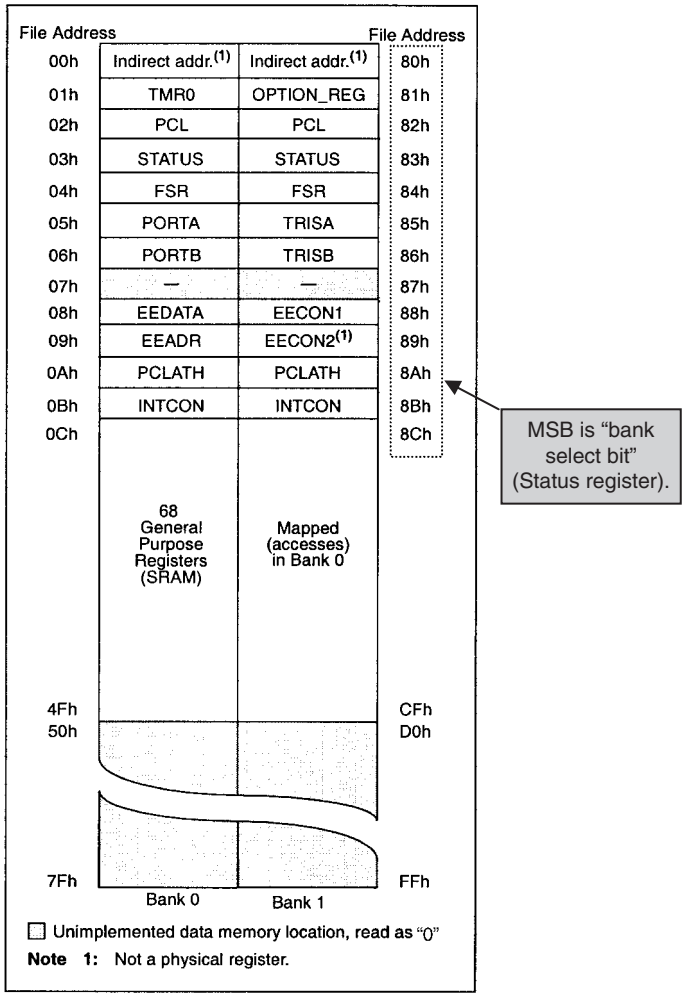


Figure 2.5: Data Memory and Special Function Register Map of the 16F84A (Supplementary Labels in Shaded Boxes Added by the Author)

$0C_H$ to $4F_H$. Above that are the Special Function Registers (SFRs). Let us explore the two concepts just mentioned, as they are likely to be unfamiliar.

2.4.2.1 Banked Addressing

A problem with any memory space is that the larger the memory is, the larger the address bus must be. One way of avoiding big address buses is to divide the memory into a number of smaller blocks called banks, each identical in size. Now a smaller address bus can be used. It can access all banks in an identical way, with just one of the banks being identified at any one time as the target of the address specified.

PIC microcontrollers adopt a banked structure for their RAM, with the 16F84A having just two banks. The address of either bank is the 7-bit RAM address (RAM addr) seen in Fig. 2.2. The active bank is selected by bit 5 in the Status register (Fig. 2.3). The programmer must ensure that the bank bit in the Status register is correctly set before making any access to memory.

2.4.2.2 Special Function Registers

The SFRs are the gateway to interaction between the CPU and the peripherals, and we will get to know them very well. To the CPU, an SFR acts more or less like a normal memory location—you can usually write to it or read from it. What makes it "special" is that the bits of that memory location have a dual purpose. Each bit is wired across to one or other of the microcontroller peripherals. Each is then used either to set up the operating mode of the peripheral or to transfer data between the peripheral and the microcontroller core. As we get to know the peripheral of the 16F84A, we will get to know each of the SFRs shown in Fig. 2.5. Note that four SFRs appear in Fig. 2.2. Can you identify what they are?

2.4.2.3 RAM Addressing

Figure 2.2 shows that there are two possible sources of the RAM address, selected through the address multiplexer (Addr Mux). Either the address forms part of the instruction, and is routed across to the address multiplexer from the Instruction register. This is called *direct* addressing. Alternatively, the address is taken from the File Select Register, or FSR, which can be found as one of the SFRs in Fig. 2.5. If the user loads an address into the FSR, that can then be used as an address to data memory, a technique known as *indirect* addressing. This will be described in Chapter 5, Section 5.4.1.

The actual memory addresses are shown in Fig. 2.5, labeled as "file address." These addresses, at least in the right-hand column, appear to be 8-bit. We know, however, from Fig. 2.2 that the RAM address bus is only 7-bit, or only 5 valid bits if direct addressing is used. It is important to understand that the addresses shown are made up of this 7-bit RAM address, *with* the bank select bit from the Status register inserted as the eighth, most significant, bit. When programming it is necessary to separate the two, ensuring that the MSB in Fig. 2.5 is used for the bank select bit. This will become clear as we start to program.

2.4.3 The Configuration Word

A special part of the 16F84A program memory is its *Configuration Word* (Fig. 2.6). This allows the user to define certain configurable features of the microcontroller, at the time of program download. These are fixed until the next time the microcontroller is programmed. This is distinct from those many selectable features, like the setting of SFRs, which are under normal program control. While the Configuration Word is part of program memory, it is not accessible within the program, or in any way while the program is running. The actual features it controls, which can be read on the diagram, are explained in this and later chapters.

R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u
CP	CP	CP	CP	CP	CP	CP	CP	CP	CP	$\overline{\text{PWRTE}}$	WDTE	F0SC1	F0SC0
bit13													bit0

bit 13-4 **CP:** Code Protection bit
1 = Code protection disabled
0 = All program memory is code protected

bit 3 **$\overline{\text{PWRTE}}$**: Power-up Timer Enable bit
1 = Power-up Timer is disabled
0 = Power-up Timer is enabled

bit 2 **WDTE**: Watchdog Timer Enable bit
1 = WDT enabled
0 = WDT disabled

bit 1-0 **FOSC1:FOSC0**: Oscillator Selection bits
11 = RC oscillator
10 = HS oscillator
01 = XT oscillator
00 = LP oscillator

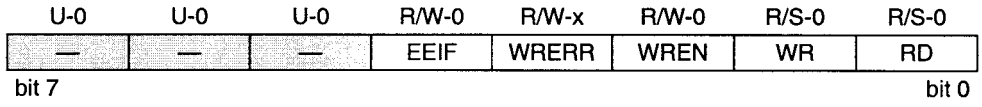
Figure 2.6: 16F84A Configuration Word

2.4.4 EEPROM

The EEPROM is nonvolatile and is particularly useful for holding data variables that can be changed, but are likely to be needed for the medium to long term. Examples include TV tuner settings, phone numbers stored in a cell phone or calibration settings on a measuring instrument.

In the 16F84A (and indeed any PIC microcontroller), the EEPROM is not placed in the main data memory map. Instead (as the top right of Fig. 2.2 neatly shows) it is addressed through the **EEADR** register and data is transferred through the **EEDATA** register. These are both SFRs, seen in Fig. 2.5.

As the review of memory technology earlier suggests, reading from EEPROM is a simple process, but writing to it is not. This latter takes significant time in electronic terms (i.e., milliseconds) and care must be taken to avoid accidental writes. A set of controls is therefore



bit 7-5 **Unimplemented:** Read as "0"

bit 4 **EEIF:** EEPROM Write Operation Interrupt Flag bit
1 = The write operation completed (must be cleared in software)
0 = The write operation is not complete or has not been started

bit 3 **WRERR:** EEPROM Error Flag bit
1 = A write operation is prematurely terminated
(any $\overline{\text{MCLR}}$ Reset or any WDT Reset during normal operation)
0 = The write operation completed

bit 2 **WREN:** EEPROM Write Enable bit
1 = Allows write cycles
0 = Inhibits write to the EEPROM

bit 1 **WR:** Write Control bit
1 = Initiates a write cycle. The bit is cleared by hardware once write is complete. The WR bit can only be set (not cleared) in software.
0 = Write cycle to the EEPROM is complete

bit 0 **RD:** Read Control bit
1 = Initiates an EEPROM read RD is cleared in hardware. The RD bit can only be set (not cleared) in software.
0 = Does not initiate an EEPROM read

Figure 2.7: The EECON1 Special Function Register (Address 88_H)

required, to start the process and (for write) to detect when it is ended. These are found in the bits of the **EECON1** register (Fig. 2.7). To *read* an EEPROM location, the required address must be placed in **EEADR** and the **RD** bit set in **EECON1**. The data in that memory location is then copied to the **EEDATA** register and can be read immediately. To *write* to an EEPROM location, the required data and address must be placed in **EEDATA** and **EEADR** respectively. The write process is enabled by the **WREN** (Write Enable) bit being set high, followed by the bytes 55H followed by AA_H being sent to the **EECON2** register. The built-in requirement for these codes helps to ensure that accidental writes do not take place, for example on power-up or down. The WR bit is then set high and writing actually commences. The write completion is signaled by the setting of bit EEIF in EECON1.

2.5 Some Issues of Timing

2.5.1 Clock Oscillator and Instruction Cycle

Any microprocessor or microcontroller is a complex electronic circuit, made up of sequential and combinational logic. At fantastic speed it steps in turn through a series of complex states, each state being dependent on the instruction sequence it is executing. While the detail of

this process is invisible to us, it is still necessary to provide the "clock" signal, a continuously running fixed-frequency logic square wave. The overall speed of the microcontroller operation is entirely dependent on this clock frequency. It is not just the CPU that is dependent on the clock. In most microcontrollers many essential timing functions are also derived from it, ranging from counter/timer functions to serial communications. Furthermore, the overall power consumption of the microcontroller has a strong dependence on clock frequency, with high-speed operation being much more power hungry than slow speed.

As Table 2.1 shows, every microcontroller has a specified range for its clock frequency. It is up to the designer to determine the clock frequency needed and to select a means of generating the clock source. With so many things depending on the clock frequency and its stability, these can be challenging decisions.

Within any microprocessor, the main clock signal is immediately divided down by a fixed value into a lower-frequency signal. Each cycle of this slower signal is called either a machine cycle or an instruction cycle. Microchip uses the latter terminology. The instruction cycle becomes the primary unit of time in the action of the processor, for example being used as a measure for how long an instruction takes to execute. The original clock signal is retained to create phases or time stages within the instruction cycle. In PIC 16 Series microcontrollers the main oscillator signal is divided by 4 to produce the instruction cycle time.

Table 2.3 gives some popular clock frequencies, with their resulting instruction cycle durations. For the fastest clock frequency, 20 MHz, the instruction cycle frequency is 5 MHz, with a period of 200 ns. The slightly cheaper version of the controller, the 16F84-04, with maximum clock frequency of 4 MHz, has at this frequency an instruction cycle time of 1 μs. As we will see, this unsurprisingly is a convenient value for a range of simple timing applications, using software delay loops and the counter/timer. A popular clock frequency for very-low-power applications, including wristwatches, is 32.768 kHz. This has an instruction cycle period of 122.07 μs. The result is very low power, but strictly no high-speed calculations!

Table 2.3: PIC 16 Series Instruction Cycle Durations for Various Clock Frequencies

Clock Frequency	Instruction Cycle	
	Frequency	Period
20 MHz	5 MHz	200 ns
4 MHz	1 MHz	1 μs
1 MHz	250 kHz	4 μs
32.768 kHz	8.192 kHz	122.07 μs

2.5.2 Pipelining

The combination of the RISC instruction set and the Harvard memory map used by PIC microcontrollers has an added advantage: instructions can be *pipelined*. Every instruction in a computer's program memory has first to be fetched and then executed. In many CPUs these two steps are done one after the other—first the CPU fetches and then it executes. If, however, program memory has its own address and data bus, separate from data memory (i.e., a Harvard structure), then there is no reason why a CPU cannot be designed so that while it is executing one instruction, it is already fetching the next. This is called *pipelining*. Pipelining works best if fetch and execute cycles are always of the same duration, such as a RISC structure gives. This fairly simple design upgrade gives a doubling in execution speed!

All PIC microcontrollers implement pipelining, which is one of the reasons for their comparatively high speed of operation. Each instruction is fetched while the previous one is being executed. Pipelining fails only for instructions that cause the value in the Program Counter to be changed, for example a program branch or jump. In this case, the instruction fetched is no longer the one needed. The pipelining process must then start again, with the consequent loss of an instruction cycle.

A diagram representing the pipelining process in 16 Series microcontrollers is shown in Fig. 2.8. Here we can see that while instruction 1 is being executed, instruction 2 is already being fetched, the same happening as instruction 2 is executed, and so on. An example sequence of instructions is shown to the left of the diagram. It is not, however, necessary to understand their meaning to understand the diagram, except to know that the **CALL** instruction causes a program branch. The instruction following it, instruction 4, is fetched while instruction 3 is being executed. Due to the program branch, however, instruction 4 is no longer needed, and a cycle has to be lost while the new instruction is fetched.

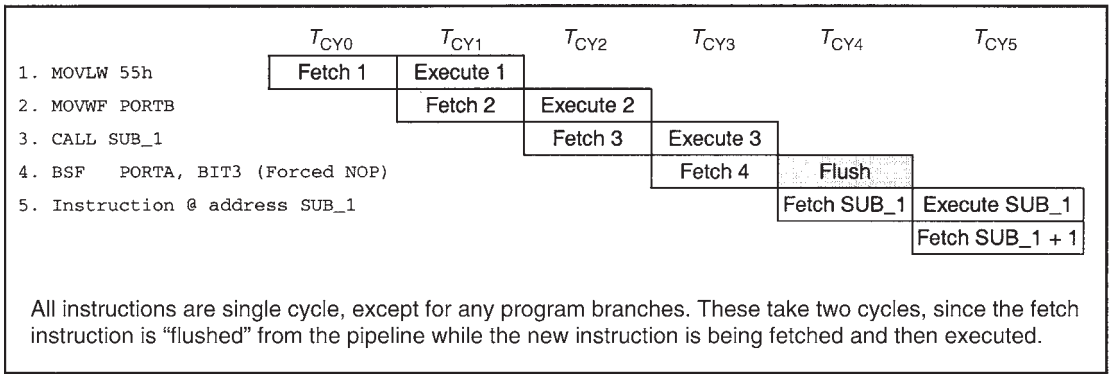


Figure 2.8: Instruction Pipelining

2.6 Power-Up and Reset

When the microcontroller powers up, it must start running its program from its beginning (i.e., for the 16F84A from its reset vector, seen in Fig. 2.4). This will only happen if explicit circuitry is built in to detect power-up and force the Program Counter to zero. Along with this, it is also very useful to set SFRs so that peripherals are initially in a safe and disabled state. This "ready-to-start" condition is called *Reset*. The CPU starts running its program when it leaves the Reset condition.

In the 16F84A there is a Reset input, $\overline{\textbf{MCLR}}$ (Master Clear, on pin 4, Fig. 2.1). As long as this is held low, the microcontroller is held in Reset. When it is taken high, program execution starts. If the pin is taken low while the program is running, then program execution stops immediately and the microcontroller is forced back into Reset mode.

There remains the question of when program execution should actually be allowed to start. The moment power is applied is a dangerous one for any embedded system. Both the power supply and the clock oscillator take finite time to stabilize, and in a complex system power to different parts of the circuit may become stable at different times. Clearly, this situation takes some careful handling. How can the start of program execution be delayed until power has stabilized?

A simple way to resolve the "what do we do as power is applied?" question is shown in Fig. 2.9, illustrated here for any microcontroller which has an active low Reset input. If a resistor capacitor circuit is connected to the Reset input, then when power is applied the capacitor voltage rises according to the RC time constant, which can be made as big as is wanted. For a certain period of time, because it is rising comparatively slowly, the $\overline{\textbf{Reset}}$ input is at Logic 0. Thus, the microcontroller can be held in Reset while its power supply stabilizes and while the clock oscillator starts up. The diagram for a simple external Reset circuit is shown in Fig. 2.9(a).

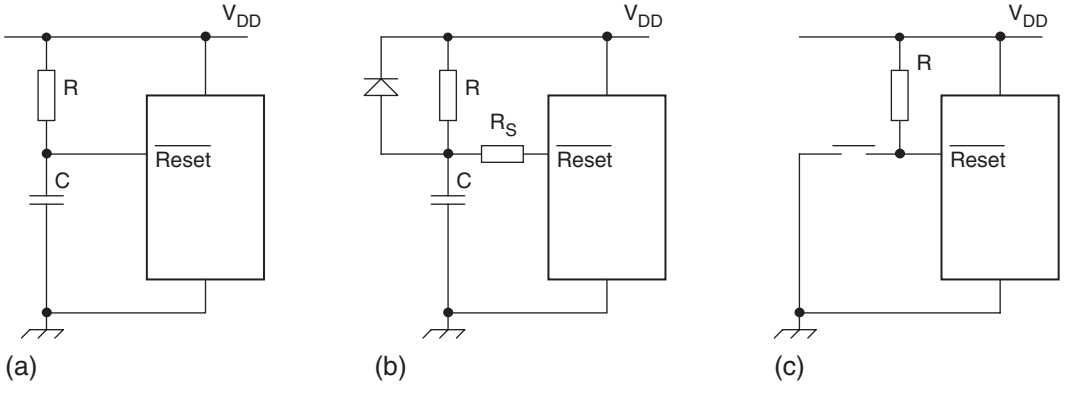


Figure 2.9: External Reset Circuits—Generic Microcontroller with $\overline{\text{Reset}}$ Input: (a) Power-on Reset, Simplest Possible, (b) Power-on Reset, with Discharge Diode and Protective Resistor and (c) User Reset Button

A small problem arises with this circuit if the power is switched off and then on again quickly (a cruel and challenging thing to do to any electronic device). With the circuit of Fig. 2.9(a) the capacitor wouldn't have time to discharge and the Reset condition might not be properly applied when power is applied again. More dangerously, the capacitor voltage might exceed the voltage supplied to the microcontroller and excessive current could then flow from the capacitor into the **$\overline{\text{Reset}}$** input. By adding a simple discharge diode, as shown in the circuit in Fig. 2.9(b), we can ensure that the capacitor discharges more or less at the same rate as the V_{DD} supply. The resistor R_S is also included, to limit current into the **$\overline{\text{Reset}}$** input if the capacitor voltage does inadvertently exceed the voltage supplied to the microcontroller or another fault condition occurs.

If the designer wishes to include a Reset button, then the circuit of Fig. 2.9(c) can be applied. This is particularly useful for prototype circuits, where a large amount of testing is expected. Then it is convenient to be able to reset a program that may have crashed. R is a pull-up resistor, whose value can be in the range 10–100 k. In a commercial device it is usually undesirable to have a Reset button; the aim here is to design the product so reset by the user is never needed.

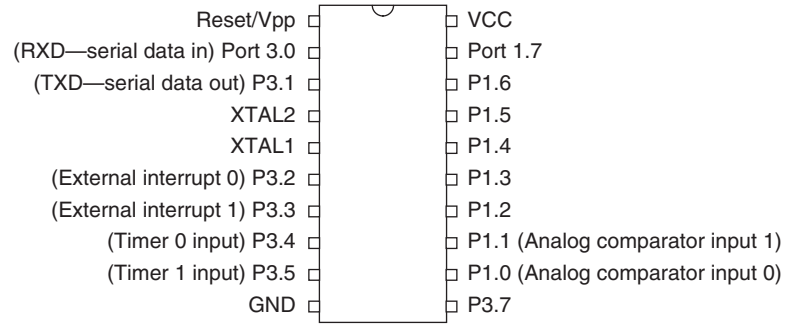
One of Microchip's goals is to minimize the number of external components needed for their microcontrollers, and the components of Fig. 2.9 fall exactly into this category. Therefore, the 16F84A includes some sophisticated on-chip reset circuitry, which in many situations makes the components of Fig. 2.9(a) or (b) unnecessary. A Power-up Timer (PWRT) is included on-chip, which can be enabled by the user with bit 3 of the Configuration Word (Fig. 2.6). The 16F84A detects that power has been applied and the Power-up Timer then holds the controller in Reset for a fixed period of time. Once this is over the microcontroller leaves Reset and program execution begins. In practice, the circuit of Fig. 2.9(b) need only be applied if the supply voltage rises very slowly. The Power-up Timer, and further details of the internal Reset circuit, are covered in greater detail in Section 2.8.

So what should be done with the 16F84A **$\overline{\text{MCLR}}$** input if we don't want to use it? It is essential to recognize that this input must not just be left unconnected. The simplest thing to do is to tie it to the supply rail and then forget about it.

2.7 What Others Do—the Atmel AT89C2051

Atmel produces a range of small (but not tiny) microcontrollers, comparable in size to the smaller PIC 16 Series devices, though somewhat larger than the 16F84A. They are based on an 8051 core, originally developed by Intel, and since adopted by a number of other manufacturers, including both Atmel and Philips.

The Atmel AT89C2051 [Refs 2.4, 2.5] is summarized in Fig. 2.10. With 20 pins it is just slightly larger than the 16F84A, and this is reflected in its internal capability. Despite the



- 8051 core (CISC)
- 2.7–6 V operating range
- 15 programmable I/O lines
- Six interrupt sources
- Direct LED drive outputs
- 2 Kbytes of reprogrammable Flash memory
- 128 × 8-bit internal RAM
- Two 16-bit timer/counters
- Programmable serial UART channel
- On-chip analog comparator

Figure 2.10: The Atmel AT89C2051 Pin Connection Diagram and Summary

differences, it is interesting to notice the similarities. Both have on-chip program and data memory, flash for program and SRAM for data. Crystal inputs, reset and power supply are all extremely similar. The PIC microcontroller has 13 digital input/output lines, the Atmel 15, both with direct LED drive capability. The PIC microcontroller has one external interrupt, the Atmel two. The Atmel has two 16-bit timers, to the one 8-bit timer of the PIC, and the Atmel has a serial capability that the PIC microcontroller just doesn't have. One would not have to look far in the PIC 16 Series family to find relatives of the 'F84A, just slightly larger than it, which have serial capability. An interesting further feature of the '2051 is the on-chip comparator, which allows the controller to host simple analog functions. However, it doesn't all go the way of the Atmel device, as the PIC microcontroller has the EEPROM memory that Atmel does not match.

All this discussion, however, is mainly just comparing peripherals, which manufacturers can add or subtract with ease. What other differences would we find if we put both controllers to use? The answer lies in the core and instruction set. The Atmel is a CISC device, with the 8051 instruction set. Let's just note here the advantages that the PIC microcontroller core enjoys. As a RISC processor, it requires just four oscillator cycles per instruction, as described earlier in this chapter. The 2051 requires 12 for each of its machine cycles, and even then many instructions require more than one machine cycle. The PIC microcontroller also uses pipelining, which the Atmel does not.

2.8 Taking Things Further—the 16F84A On-Chip Reset Circuit

Let's take a closer look at the 16F84A on-chip reset circuitry, shown in simplified form in Fig. 2.11. This takes some understanding, but it is worth doing.

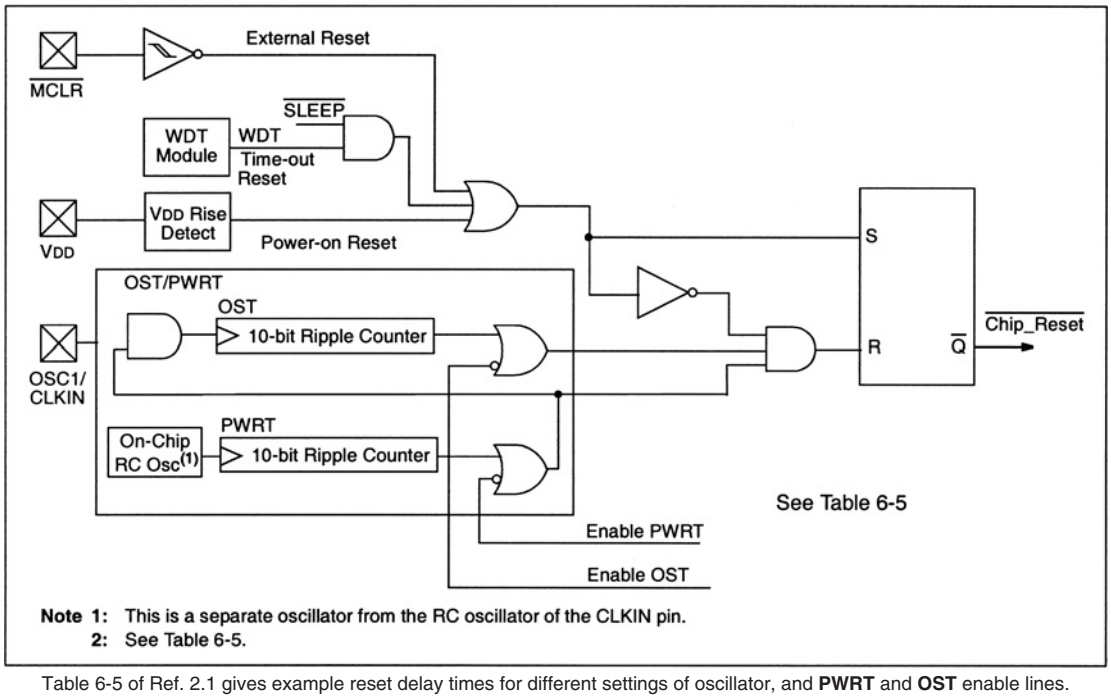


Table 6-5 of Ref. 2.1 gives example reset delay times for different settings of oscillator, and **PWRT** and **OST** enable lines.

Figure 2.11: The 16F84A Reset Circuitry

The actual reset to the CPU, **Chip_Reset**, is generated by a flip-flop, which appears to the right of the diagram. This has two inputs, **S** (Set) and **R** (Reset). The CPU enters Reset mode when **Chip_Reset** goes low, which is caused by the **S** line going high. It stays there until the flip-flop is cleared, caused by the **R** line going high.

So what causes a reset? The **S** input to the flip-flop goes high, via a three-input OR gate, if any of the following goes high:

- **External Reset**, from the **MCLR** line we have already seen
- **Time-out Reset**, from the Watchdog Timer (WDT); this is designed to occur if a program crash occurs
- **Power-on Reset**, output of the circuit that detects power being applied (VDD Rise Detect).

Once any of these occurs, the flip-flop is set, the **Chip_Reset** line goes low and the PIC microcontroller is held in Reset.

The **Chip_Reset** line returns to 1 (and the PIC microcontroller is enabled) if the **R** input to the flip-flop is activated. The three requirements to be satisfied here, determined by the inputs to the

associated AND gate, are that both power supply and oscillator have stabilized, and that any demand for Reset has been cleared. The first two of these requirements are achieved by two interesting timers, the Power-Up Timer (PWRT) and the Oscillator Start-up Timer (OST). The Power-up Timer can be enabled by setting its bit in the Configuration Word (Fig. 2.6). The Oscillator Start-up Timer is enabled via the **Enable OST** line. This is set automatically by the user oscillator setting in the Configuration Word, which enables it for all oscillator modes except RC. The Power-Up Timer is clocked by its own on-chip RC oscillator, and when enabled counts 1024 cycles of its oscillator before setting its output to 1. This time duration turns out to be around 72 ms. This is long enough for the average power supply to have stabilized, though is not enough for a slowly rising supply. Once the Power-Up Timer has completed its count, the Oscillator Start-up Timer is then activated, which in turn counts 1024 cycles of the main oscillator signal. This tests for a reliably running clock oscillator—if the oscillator isn't running, then of course it can't count. The outputs of both counters, and the inverse of the **S** input to the flip-flop, are ANDed together, to form the **R** input to the flip-flop. If all lines are high, i.e., both counters have completed their count and there is no demand for a Reset, then the flip-flop is cleared. The CPU accordingly leaves the Reset condition and starts running.

The reset sequence just described is shown in Fig. 2.12, for the common situation of $\overline{\textbf{MCLR}}$ being tied to V_{DD}. The application of power is seen in the rise of the V_{DD} trace, which brings the $\overline{\textbf{MCLR}}$ line with it. This change is detected, as seen in the change of state of the internal POR line. This in turn triggers the Power-Up Timer, which runs for a period T_{PWRT}. When T_{PWRT} is up, the Oscillator Start-up Timer, whose time delay is T_{OST}, is activated. Notice that T_{OST} depends on the main oscillator running successfully, and is dependent on its frequency. For a 4-MHz oscillator, it will be 1024×250 ns, or 256 μs. When T_{OST} is complete, the **R** line in Fig. 2.11 goes high and the microcontroller leaves the Reset state.

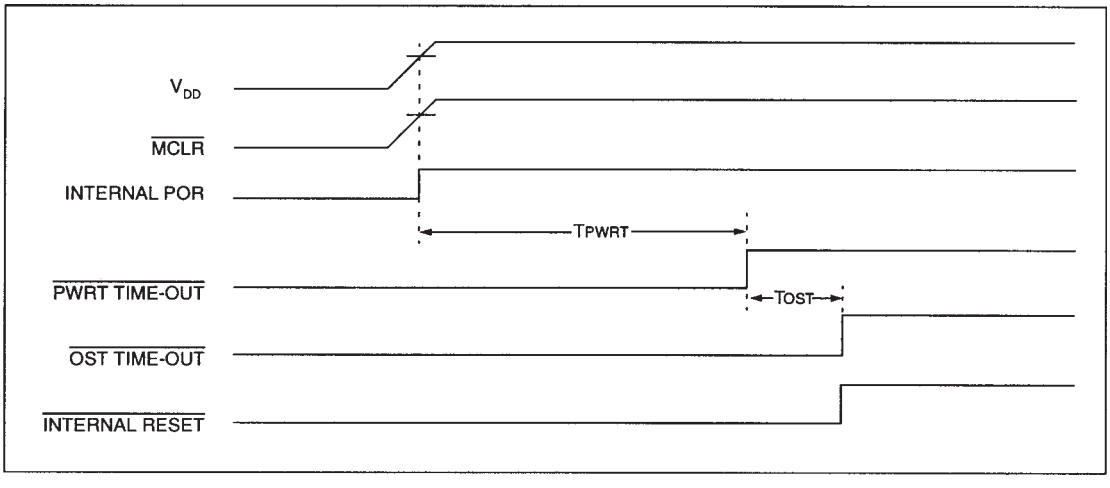


Figure 2.12: Reset Sequence on Power-Up, with $\overline{\text{MCLR}}$ Tied to V_{DD}

2.9 Summary

- The PIC 16 Series is a diverse and effective family of microcontrollers.
- The 16F84A architecture is representative of all 16 Series microcontrollers, with Harvard structure, pipelining and a RISC instruction set.
- The PIC 16F84A has a limited set of peripherals, chosen for small and low-cost applications. It is thus a smaller member of the family, with features that are a subset of any of the larger ones.
- The 16F84A uses three distinct memory technologies for its different memory areas.
- A particular type of memory location is the Special Function Register, which acts as the link between the CPU and the peripherals.
- Reset mechanisms ensure that the CPU starts running when the appropriate operating conditions have been met, and can be used to restart the CPU in case of program failure.

References

[2.1] PIC 16F84A Data Sheet (2001). Microchip Technology Inc., Reference no. DS35007B; www.microchip.com

[2.2] PICmicro Mid-Range MCU Family Reference Manual (1997). Microchip Technology Inc., Reference no. DS33023A; www.microchip.com

[2.3] PIC16F84 to PIC16F84A Migrati on (2001). Microchip Technology Inc., Reference no. DS30072B; www.microchip.com

[2.4] 8-bit Microcontroller with 2K Bytes Flash (2000). Atmel Corporation, AT89C2051, Rev. 0368E-02/00; http://www.atmel.com/

[2.5] Atmel 8051 Microcontrollers Hardware Manual (2004). Atmel Corporation, Ref. 4316C-8051-05/04; http://www.atmel.com/

This page intentionally left blank

CHAPTER 3

Parallel Ports, Power Supply and the Clock Oscillator

So far we have looked a little at the theory of microcontroller architecture, and its implementation in PIC® microcontrollers. This chapter now begins to move from that theory to the practice of small-scale hardware design.

As we have seen, the microcontroller core has internal data and address buses. In a way these are like motorways, or interstate freeways, carrying large amounts of traffic in both directions to a variety of different destinations. The microcontroller needs to be provided with a way of allowing that data flow to connect with the outside world, so that it can read in external digital values or output other values. In other words, it needs the equivalent of motorway junctions, where data can leave (or enter) the bus at designated times and locations. In the microcontroller world these junctions have many forms, as there are many different ways that data can be input or output. The most general purpose of these is the parallel input/output port. This is one of the microcontroller's most essential peripherals, and is the opening subject of this chapter.

Given a working car engine, two essentials that it needs to run are fuel and a stream of sparks from the plugs. A microcontroller has similar needs. Its fuel is the low-level electrical power supply that it requires, and instead of a flow of sparks, it needs a regular sequence of clock oscillator pulses. A study of these forms the second half of this chapter.

In this chapter you will learn about:

- Why we need parallel input/output
- How simple logic circuits can be developed to give a flexible interface between the microcontroller data bus and the outside world—these are the parallel ports
- How external devices can be connected to the parallel port
- The parallel input/output available on the PIC 16F84A
- The essential hardware features of power supply and clock oscillator
- The Microchip approach to power supply and oscillator, with the 16F84A
- The hardware design of the electronic ping-pong game.

3.1 The Main Idea—Parallel Input/Output

Almost any embedded system needs to transfer digital data between its CPU and the outside world. This transfer falls into a number of categories, which can be summarized as:

- *Direct user interface*, including switches, keypads, light-emitting diodes (LEDs) and displays
- *Input measurement information*, from external sensors, possibly being acquired through an analog-to-digital converter
- *Output control information*, for example to motors or other actuators
- *Bulk data transfer* to or from other systems or subsystems, moving in serial or parallel form, for example sending serial data to an external memory.

With this plethora of data coming and going, it is likely we will need to have a variety of digital inputs and/or outputs. These are divided broadly into serial and parallel. In serial data transfer, the information is transferred one bit at a time. Only a single interconnection is used to carry the data itself, although other lines are usually included for synchronization and control. In parallel data transfer, a set (for example, eight) of interconnections is used. Each of these can carry 1 bit, and each works in parallel with the others. Data can thus be transferred in groups of bits, for example in bytes. Parallel input/output (I/O) is the workhorse for all the basic data interchange of a microcontroller, including interfacing with switches, LED, displays and so on. A group of parallel I/O interconnections, appearing on the pins of the microcontroller, is called a *parallel port*.

3.2 The Technical Challenge of Parallel Input/Output

Our immediate challenge is how to provide the required interface between the microcontroller data and address buses and the outside world. As suggested above, we start with the data bus, a multi-purpose data highway. How can we grab the data we want from the bus and transfer it to the outside world, via the parallel port? Alternatively, how can we take external input data and introduce it onto the data bus, at the right time and place, so that it gets to the right place within the microcontroller? Finally, given a port that can do these things, how can we make it really flexible, so that it can be used for input, or output, or a mixture of both, and can transfer a combination of data with possibly very different end uses.

3.2.1 Building a Parallel Interface

It should be simple to create a set of output pins to create an *output* port (Fig. 3.1). Let us assign an address in the memory map to the port. Whenever that address is selected by an instruction in the program, it activates a line called *Port Select*. A further line, *Read/$\overline{Write}$*, indicates whether the CPU is undertaking a Read (line is high) or Write (line is low)

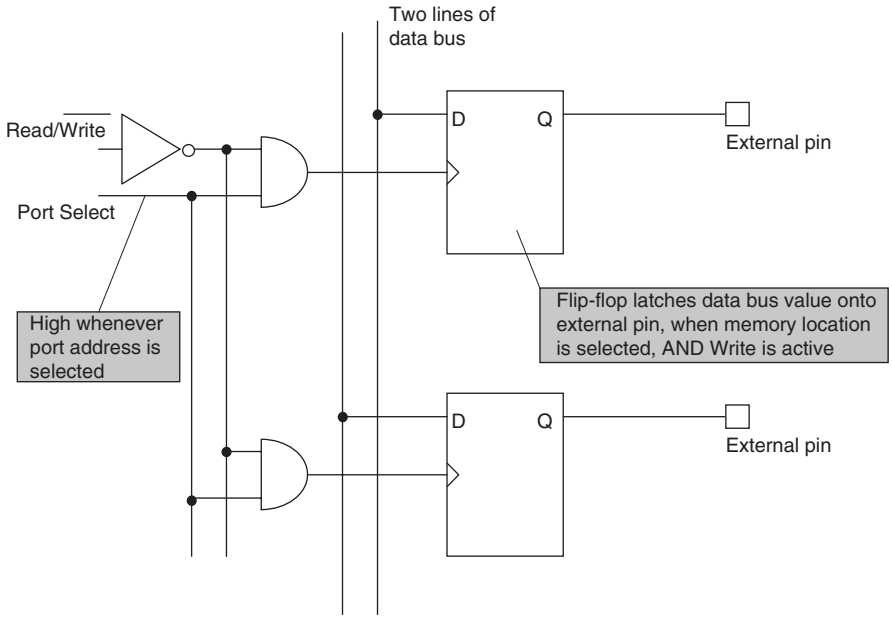


Figure 3.1: Two Bits of a Possible Digital Output Port

operation. This is gated with the Port Select line. Each line of the data bus is connected to a bistable, and all of these are clocked by the Port Select line. Then the value of the data bus is latched into the bistable whenever the port memory location is addressed, in Write mode. The outputs of the bistables are made available for connection to the outside world.

It is equally simple to create a set of input pins (Fig. 3.2). All that is needed is a tristate buffer gate connected between an external pin and a line of the data bus. When the buffer is

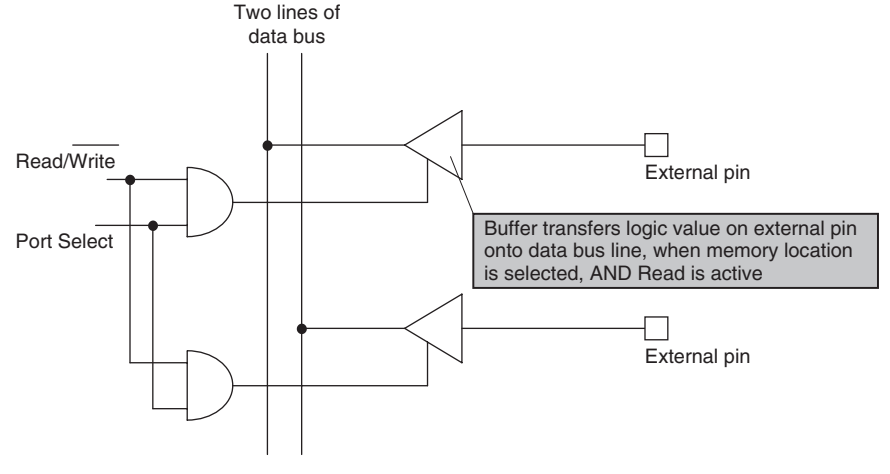


Figure 3.2: Two Bits of a Possible Digital Input Port

enabled, again by a logical combination of Port Select line and Read/$\overline{\text{Write}}$ control, the logic value of the external pin is briefly connected to the data bus line, and can be read by the CPU. Note that in this design the external data is not latched by the port, it must be held at a stable value by the external source.

These ideas are quite attractive, but the reality is that it is inflexible to limit an external pin of an IC to just one function, whether input or output. It would be much neater to combine somehow the two circuits used for input and output, and let the user decide which direction he/she wants the data to move. The diagram of Fig. 3.3 does just that. It shows a possible "pin driver" circuit

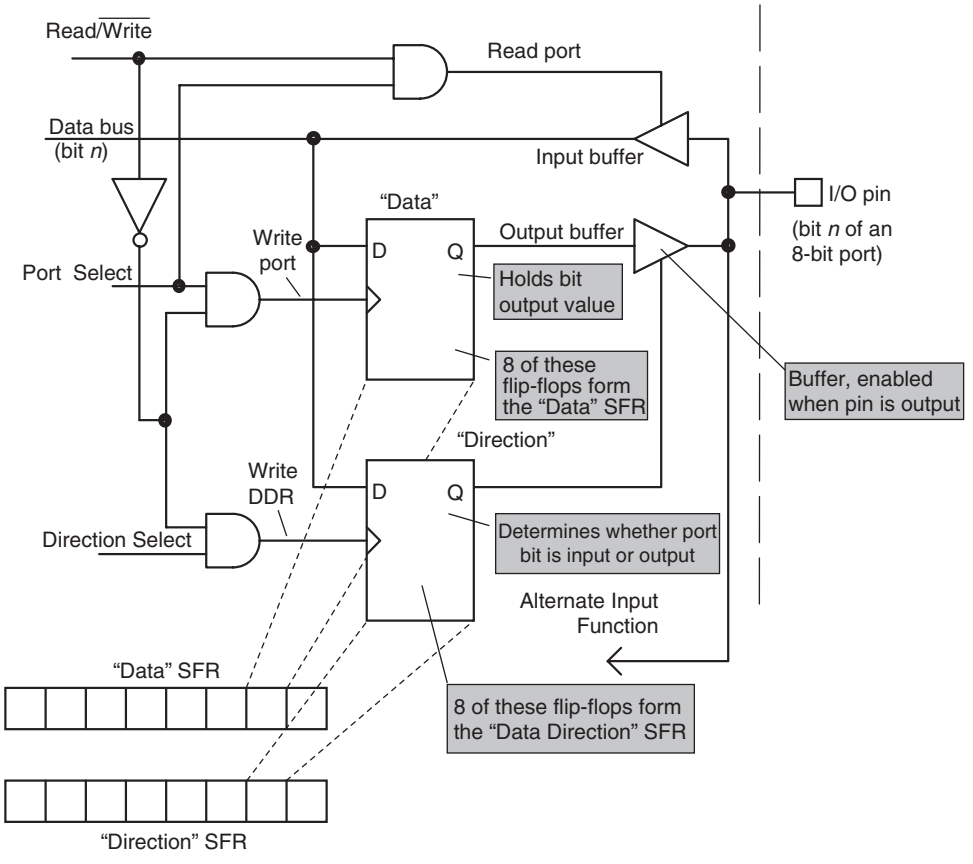


Figure 3.3: A Bidirectional Port Pin Driver Circuit

for one bit of a parallel port. It is easy to pick out in it the circuits of Figs 3.1 and 3.2. What must be added, however, is a further flip-flop ("Direction"), which is set to determine whether this microcontroller pin is to act as an input or output. The state of this flip-flop is set by the program. It controls the "output buffer", which is enabled when the port bit is in output mode.

This circuit forms the basis for a very useful bidirectional input/output pin driver, and it is easy to find versions of it in many popular microcontrollers. Sets of I/O pins are grouped together to form a parallel I/O port. Each Data flip-flop then forms one bit of a Data SFR (Special Function Register), and each Direction flip-flop forms one bit of a Direction SFR, as seen in Fig. 3.3. Each SFR is memory mapped, with its own unique address. Derived from that address is its select line, which goes high when that location is addressed. *Port Select* selects the Data SFR and *Direction Select* selects the Direction SFR.

By writing to the Direction SFR the user can determine which bits are to be input and which are to be output. By writing to the Data SFR he/she can set the value of *all* Data flip-flops, whether that pin is actually set as an output or not. This value is transferred to the I/O pin through the buffer for those pins which are enabled as outputs. By reading from the Data SFR the program can acquire the logical value of the I/O pin. If the pin is set as output, this value is simply the value held by the Data flip-flop and asserted on the I/O pin through the Output Buffer. If the pin is set as an input, then an external signal should be connected to the pin, and the controller will read its value.

Having established this basic design, it is possible to extend it further to add other features. We will see this when we look at some PIC microcontroller examples. One simple extension is already indicated in Fig. 3.3, however. This is the Alternate Input Function line, which allows an internal peripheral to share the I/O pin.

3.2.2 Port Electrical Characteristics

Logic gates are designed to interface easily with each other, and if we connect logic gates from just one family together then we usually don't need to worry about the electrical details of what is going on. If, however, we are connecting logic devices (in this case microcontroller port bits) to *non*-logic elements (like LEDs or switches) then we *do* need to understand the electrical characteristics of the logic. In particular, we need to understand their input and output characteristics.

The output of a logic gate *can* be visualized, or "modeled," as in Fig. 3.4(a). If the output is at Logic high (or 1), then the internal switch is in the upper position. It is in the lower position for Logic 0. In either case, the output is modeled as a voltage source in series with a resistor (in circuit theory this is called a Thevenin equivalent circuit). V_{LH} is the logic high output voltage, with an output resistance of $R_{S(high)}$. V_{LL} is the logic low output voltage, with an output resistance of $R_{S(low)}$.

In the case of CMOS (Complementary Metal Oxide Semiconductor) the situation is quite simple, as V_{LH} is equal to the supply voltage and V_{LL} is equal to 0 V. This is illustrated in Fig. 3.4(b). Thus, if the supply voltage is 5 V, then Logic 0 and 1 will be 0 and 5 V respectively, if no current is being drawn from the gate output.

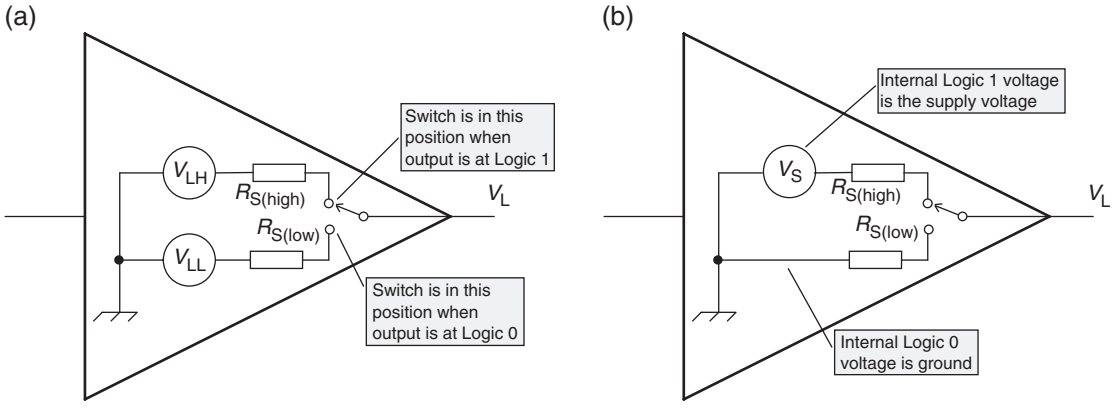


Figure 3.4: Modeling a Logic Gate Output: (a) Generalized Model and (b) Model of CMOS Logic Gate Output

In practice, $R_{S(high)}$ and $R_{S(low)}$ are not constant, but depend to some extent on the current being sourced or sunk from the gate output. Therefore, manufacturers frequently publish graphical information on the output characteristics. We will see this shortly for the 16F84A.

3.2.3 Some Special Cases

We review now two special types of I/O characteristic, which will be important as we explore the 16F84A parallel ports.

3.2.3.1 Schmitt Trigger Inputs

A Schmitt trigger (Fig. 3.5) is a certain type of logic gate input which is designed to "clean up" a corrupted logic signal. It has two input thresholds, with the positive-going higher than the negative-going. A signal starting from a low value has to pass the negative-going threshold (at which point nothing happens) and then cross the positive-going threshold, at which point the output changes state. The output will not reverse until the input (now negative-going) has got right back down to the negative-going threshold. Thus, small fluctuations that recross a threshold just crossed do not cause any change in output.

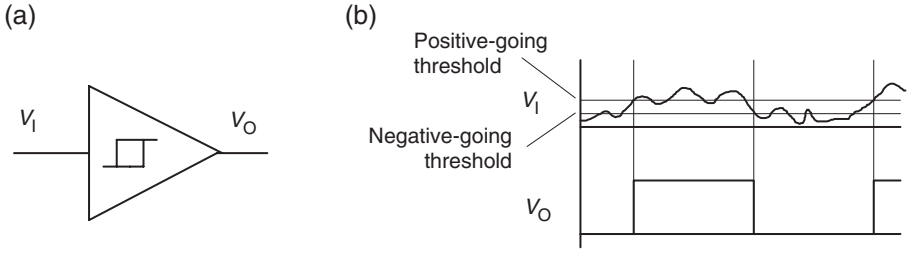


Figure 3.5: Schmitt Trigger Characteristics: (a) Buffer with Schmitt Trigger Input and (b) Input/Output Characteristic

3.2.3.2 The Open Drain Output

The Open Drain output is a flexible style of output that can be adapted either as a standard logic output, *or* as a direct drive for small loads, *or* used for a special logic function known as Wired-OR. The output itself is as illustrated in Fig. 3.6(a). A logic gate drives the gate of a MOSFET (Metal Oxide Semiconductor Field Effect Transistor), whose unconnected Drain terminal forms the output. When the MOSFET gate drive is high, the FET conducts and a logic zero is asserted at the terminal. When the gate is low the FET will not conduct and

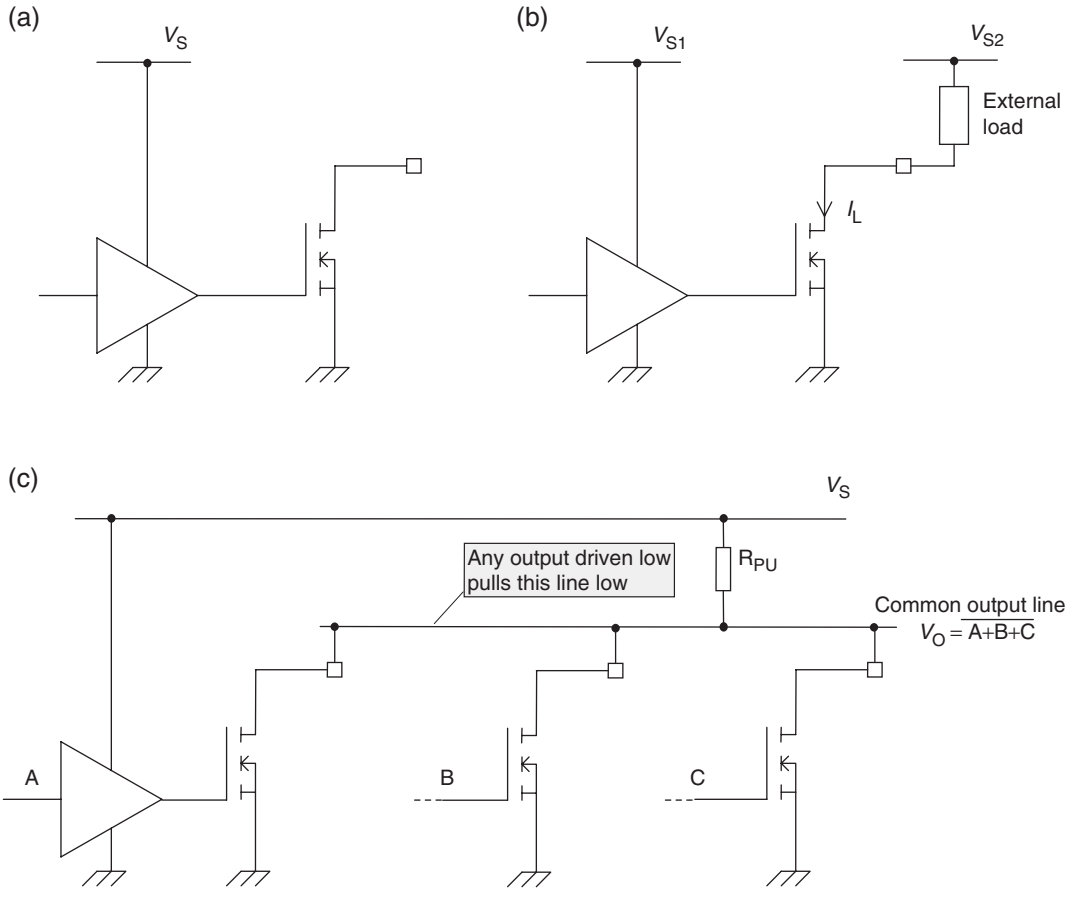


Figure 3.6: The Open Drain Output and Some Applications: (a) An "Open Drain" Output, (b) Open Drain Output Driving Load Resistor and (c) The "Wired-OR" Connection

(with no other connection) the terminal will be at an undefined voltage. If a pull-up resistor is connected from the Drain to the supply voltage, then the output acts more or less as a normal logic output. Without the active pull-up of a normal logic output, however, its rise time will be a little sluggish and the amount of current it can source will be limited by the resistor value.

The Open Drain output can also be used to drive a simple load, acting as illustrated in Fig. 3.6(b). Usefully, the load does not have to be supplied from the same voltage as the logic supply, although it would have to be of the same polarity. Therefore, for example, a microcontroller supplied from 5 V (V_{S1} in the diagram) could drive a load supplied from 12 V (V_{S2} in the diagram), if all operating requirements are met.

Another important application of the Open Drain output is the Wired-OR connection, shown in Fig. 3.6(c). Here several Open Drain outputs are connected together and tied high through a single pull-up resistor, R_{PU}. If all outputs are off, then the common line (V_O) is high. If *any* one output goes low, then the common line is pulled low. This is a possible way of achieving the OR or NOR logic function, and important for certain types of serial link, as we shall see later.

3.3 Connecting to the Parallel Port

3.3.1 Switches

Switches are extensively used in embedded systems. Our main initial interest is not to switch directly a voltage or current, but to convert the switch position to a logic level that can be read by a microcontroller port bit. Switches are used as direct user interface in the form of push-buttons, toggle switches, slide switches, or as thumbwheel or rotary switches. They are also used, in the form of microswitches, to detect certain types of mechanical movement.

The simplest way of deriving a logic level from a switch is shown in Fig. 3.7(a). This shows a single-pole, double-throw (SPDT) switch, with one terminal connected to ground and the other to the supply. The switch wiper simply selects one of these two as the logic input. Some logic families advise against direct connection of a logic input to the supply voltage, so a series resistor (shown dotted) might be in order.

There is a slight disadvantage to the connection of Fig. 3.7(a), as it requires the SPDT switch. A simpler option, using just a single-pole, single-throw (SPST) switch, for example a push-button, is shown in Fig. 3.7(b). Here a pull-up resistor is connected to one terminal of the switch, with the other terminal connected to ground. If the switch is closed, then the input to the logic gate, V_I, is 0 V and a current V_S/R flows to ground. If the switch is open then V_O is equal to V_S. To reduce wasted current when the switch is closed, the value of R should be high. If it is too high, however, then the Logic 1 level that it is meant to define may not be properly sustained. To evaluate the upper limit of the pull-up resistor, the input leakage current and logic thresholds need to be applied. For PIC microcontrollers, pull-up values in the range 10–100 k are usually appropriate. The circuit of Fig. 3.7(b) is very useful and widely applied, as many simple switches (e.g., PCB-mounting slide switches and push-buttons) are only available as SPST.

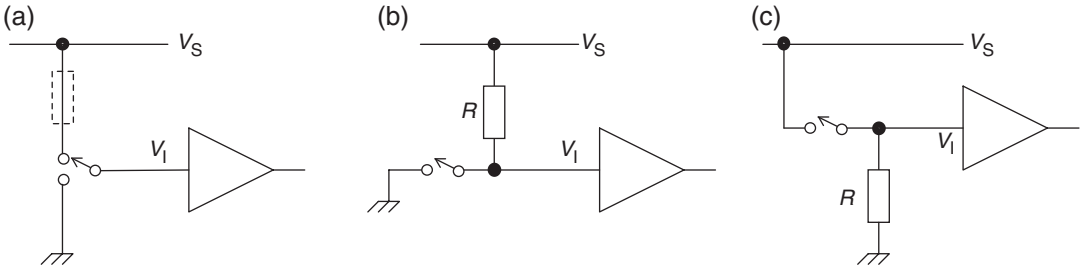


Figure 3.7: Connecting Switches to Logic Inputs: (a) SPDT Connection, (b) SPST With Pull-Up Resistor and (c) SPST with Pull-Down Resistor

The switch circuit of Fig. 3.7(b) *can* be reconnected as in Fig. 3.7(c). The characteristics of some logic families (for example, TTL) do, however, place restrictions on the use of this circuit, as the current sourced from the gate input significantly affects the action of the pull-down resistor. The circuit can be applied with PIC microcontrollers.

3.3.2 Light-Emitting Diodes

In certain semiconductor materials light is emitted as current flows across a forward-biased p-n junction. LEDs exploit this phenomenon. LEDs made of gallium arsenide (GaAs) emit light in the infrared, and if phosphorus is added in increasing proportions, the light moves to visible red and ultimately to green. LEDs are widely available in red, green and yellow, as single devices, and as arrays, bargraphs and alphanumeric displays.

Because they are diodes, LEDs display the normal voltage–current relationship of a forward-biased diode. This means that, to a reasonable approximation, the voltage across an LED is constant, if it is conducting. Note, however, that this forward voltage is considerably higher for GaAs than it is for silicon. Example LED characteristics, for red and green Kingbright LEDs, are shown in Fig. 3.8. From these graphs it can be seen that the voltage across the red LED changes from 1.90 to 2.00 V if the current increases from 5 to 20 mA. For the green it changes from 1.95 to 2.20 V for the same current range. These voltage values are typical for all LEDs of similar type, with red having a slightly lower forward voltage, compared to green or yellow.

The different colors also do not give equal intensities for equal drive currents, as shown in the data in Fig. 3.8. Red is the most efficient, which may account for its greater popularity. For a single LED to be comfortably visible, it typically requires around 10 mA of current. Brighter ones may require up to 20 mA, but special low-power devices (such as the high-efficiency red) need as little as 1 or 2 mA to be seen.

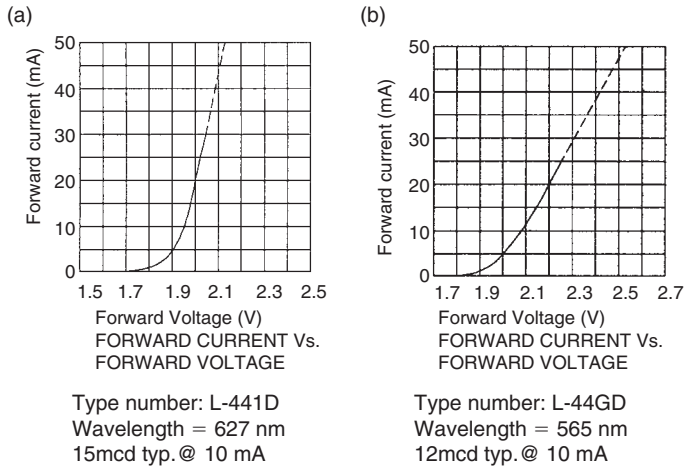


Figure 3.8: Example Kingbright LED Characteristics [Ref. 3.1]: (a) High-Efficiency Red and (b) Green (Reproduced with Permission of Kingbright Elec. Co. Ltd)

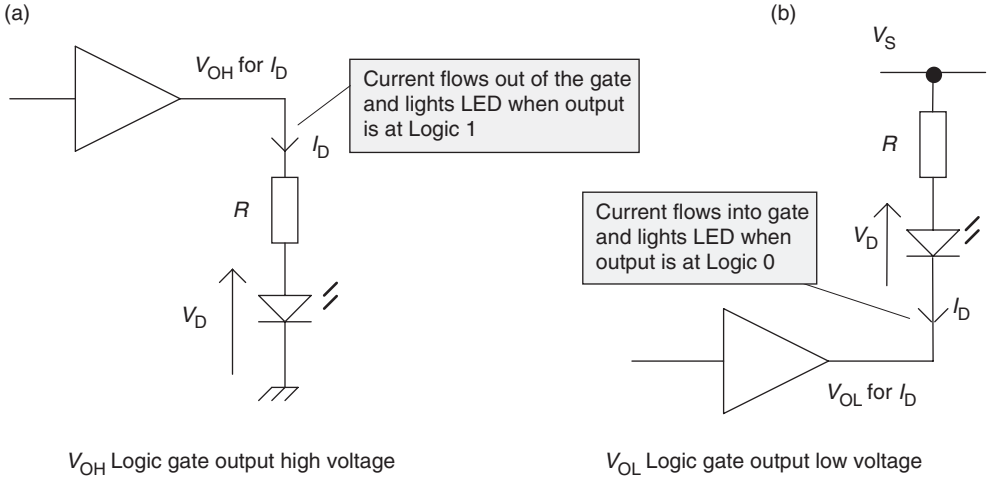


Figure 3.9: Driving LEDs from Logic Gates: (a) Gate Output Sourcing Current to LED and (b) Gate Output Sinking Current from LED

An LED can be driven from a logic output, for example a microcontroller port, as long as its current requirements can be met. Depending on the capabilities of the port output they can be connected so that the output is sourcing current (Fig. 3.9(a)) or sinking current (Fig. 3.9(b)).

CMOS logic families have symmetrical outputs and can source or sink almost equally well, so either of these circuits can be applied. In contrast, TTL logic can source little current but can sink a comparatively large amount, and therefore the configuration of Fig. 3.9(b) is preferred in this case.

A current-limiting resistor must normally be included in series with the LED. This is calculated as shown below by considering the voltages in the circuit. Precise values are not usually required.

$$\text{For current source: } V_{OH} = RI_D + V_D$$

$$R = \frac{V_{OH} - V_D}{I_D} \tag{3.1}$$

$$\text{For current sink: } V_S = V_{OL} + RI_D + V_D$$

$$R = \frac{V_S - V_D - V_{OL}}{I_D} \tag{3.2}$$

An exception to the need for a series resistor, which must be cautiously applied, is when the logic is powered from a comparatively low voltage, and its internal output resistance itself forms an appropriate value for the current-limiting resistor.

3.4 The PIC 16F84A Parallel Ports

We saw in Chapter 2 that the 16F84A has two ports, A and B. A is 5-bit, while B is 8-bit. Notice from Fig. 2.1 that some port bits have more than one function. We will see that the 16F84A adapts the generic pin driver circuit of Fig. 3.3 and cleverly weaves in these extra functions.

The SFRs that relate to the ports are seen in Fig. 2.5. In each case the Port data itself appears in the **PORTA** or **PORTB** register (i.e., these act as the Data SFR of Fig. 3.3), while the data direction is determined by the bit values set in the **TRISA** or **TRISB** registers (i.e., these act as the Direction SFR of Fig. 3.3).

We will now explore the ports in some further detail. Perhaps the most straightforward is Port B, with which we accordingly start.

3.4.1 The 16F84A Port B

This is a general-purpose 8-bit bidirectional port, with pin driver circuit similar to that in Fig. 3.3. The simplest bits, 0 to 3, are illustrated in Fig. 3.10(a). The data latch can be seen

in each circuit, while the TRIS latch in Fig. 3.10 replaces the Direction latch of the earlier diagram. It can be seen that if the TRIS latch output is set to 0, then the buffer that it drives is enabled and the port bit is in output mode.

There are four enhancements to the simple pin driver circuit we saw earlier:

- The incoming data is latched, through the lowest latch in the diagram, rather than just its instantaneous value being read.
- The state of the TRIS latch can be read, via the buffer controlled by the **RD TRIS** line. It follows that the **TRIS** register acts as a normal read/write memory location, and the program can check, if necessary, the values previously stored there.
- Bit 0 is also the external interrupt input and has a Schmitt trigger interface.
- Weak pull-up resistors can be switched on, for all port bits used as inputs. These can be applied to replace the resistor in circuits like in Fig. 3.7(b). The pull-up is implemented with a p-channel MOSFET, seen at the top of the diagram. They are enabled for all port bits set as input by clearing the bit **$\overline{\text{RBPU}}$** in the **OPTION** register. (This is seen memory mapped in Fig. 2.5.)

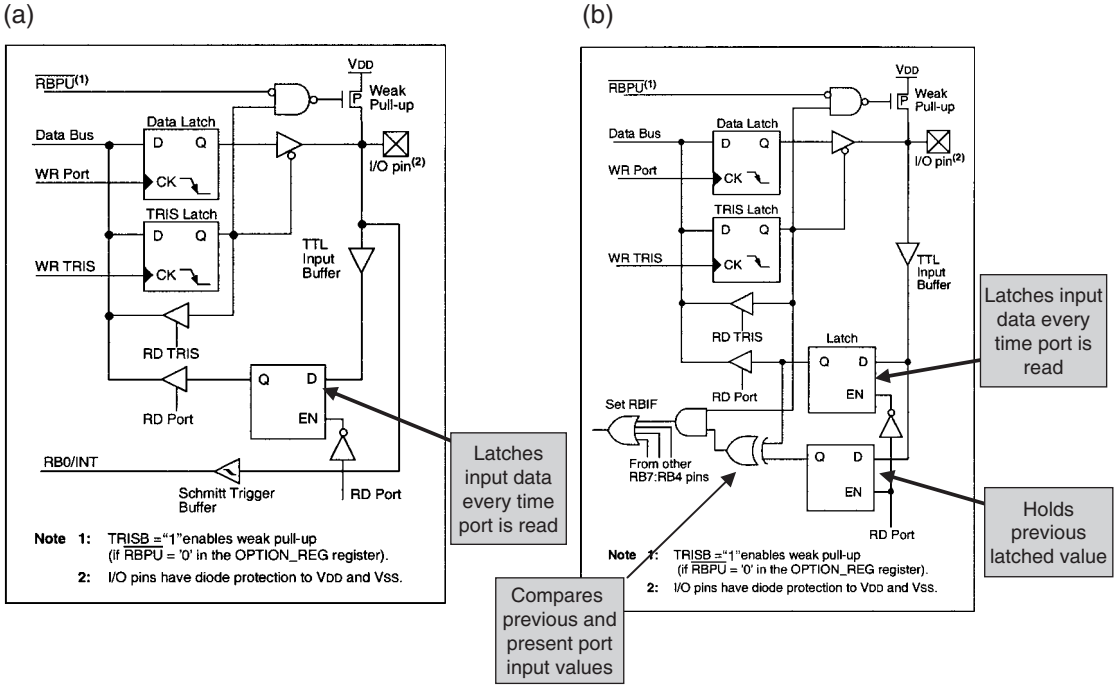


Figure 3.10: Block Diagram of Port B Pin Driver Circuits: (a) Pins RB3 to RB0 and (b) Pins RB7 to RB4 (Supplementary Labels in Shaded Boxes Added by the Author)

Bits 4 to 7 of Port B are seen in Fig. 3.10(b). They have a useful interrupt on change facility. As with the lower numbered bits, the data value is latched as input data is read. On these bits, however, the previous input value, from the last time the port was read, is retained on another latch. Its stored value is compared with the current input value. Any difference is detected by an Exclusive OR gate, whose output can generate an interrupt.

3.4.2 The 16F84A Port A

Like Port B, this can be used as a general-purpose bidirectional digital port. The basic port pin driver (Fig. 3.11(a)) is very similar to the Port B pin. The diagram this time draws out in full the output tristate buffer. Bit 4 (Fig. 3.11(b)) doubles as the Timer 0 clock input. It also has a Schmitt trigger input characteristic and an Open Drain output, as described in Section 3.2.3. The full device data indicates that the absolute maximum permissible voltage applied to this Open Drain pin is 8.5 V. Therefore, the ability to drive an external load from a supply higher than the microcontroller itself can only be applied in a limited way.

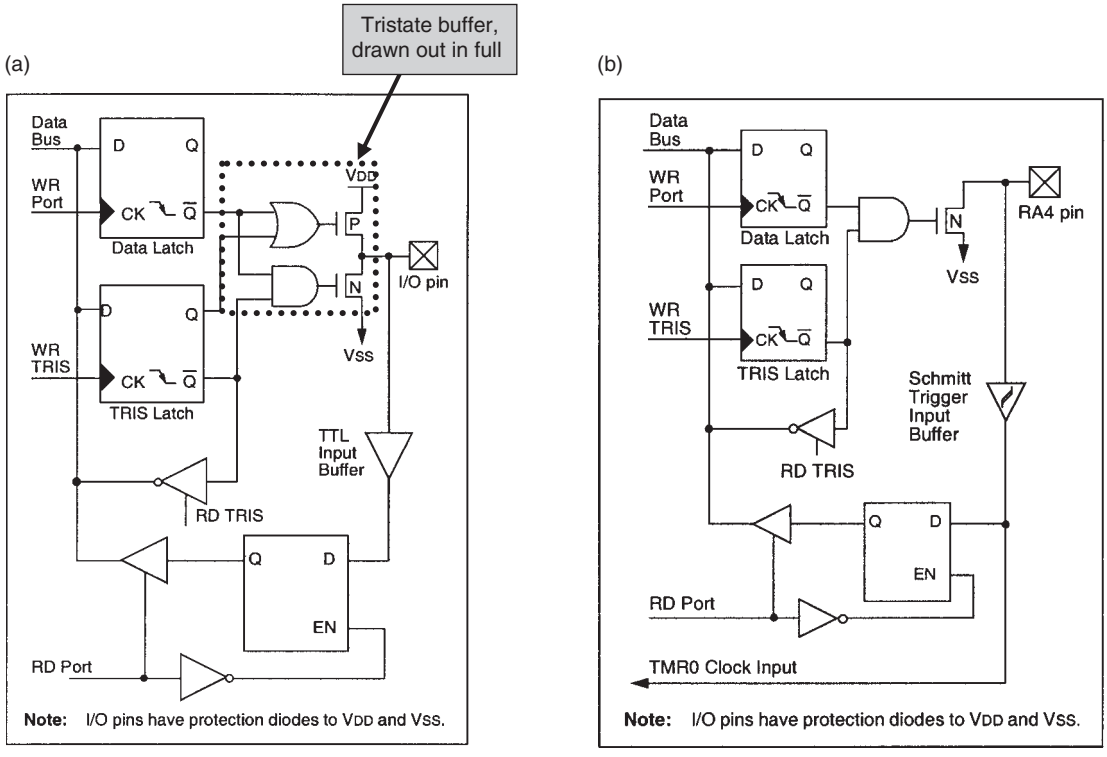


Figure 3.11: Block Diagram of Port A Pin Driver Circuits: (a) Pins RAO, 1, 2, and 3 and (b) Pin RA4/TOCKI (Supplementary Labels in Shaded Boxes Added by the Author)

3.4.3 Port Output Characteristics

The 16F84A port output characteristics are shown in Fig. 3.12, for a supply voltage of 3.0 V. In Fig. 3.12(a) we see (at 25°C) how the output voltage for Logic 1 is 3 V when the output current is 0, but falls to around 1.7 V when the output current is 10 mA, flowing out of the gate. Similarly, in Fig. 3.12(b) we see (at 25°C) how the output voltage for Logic 0 is 0 V when the output current is 0, but *rises* to around 0.8 V when the output current is 22.5 mA (flowing into the gate). It is curves like these that can be used to find the V_{OL} and V_{OH} values used in Eqs (3.1) and (3.2), once a value for I_D is known. Graphs are also given in the full data for characteristics with a 5 V supply.

Another way of applying these curves is to deduce from them an approximate output resistance. This can be done by measuring the gradient of the curve at a particular point. A simple construction to do this has been added to each plot. By dividing vertical (voltage) by horizontal (current) for each of these, output resistances of approximately 130 Ω when at Logic high and 36 Ω when at Logic 0 can be deduced. If we call these two values R_{OH} and R_{OL}, respectively, Eqs (3.1) and (3.2) can be written in a different form:

$$\text{For current source: } V_S = (R + R_{OH})I_D + V_D$$

$$R = \frac{V_S - V_D}{I_D} - R_{OH} \tag{3.3}$$

$$\text{For current sink: } V_S = (R + R_{OH})I_D + V_D$$

$$R = \frac{V_S - V_D}{I_D} - R_{OH} \tag{3.4}$$

3.5 The Clock Oscillator

The choice of microcontroller clock source determines some of its fundamental operating characteristics. While "faster is better" in terms of operating speed and programming execution, faster is definitely worse in terms of power consumption, and also possibly in terms of electromagnetic interference. All timed elements within the microcontroller almost invariably depend on the clock characteristics. If stable and accurate timing is required, then the clock oscillator must be stable and accurate. With these points in mind, the clock source must be chosen with care and understanding. This section starts with a review of the clock technologies available, before moving on to looking at the options offered with the 16F84A.

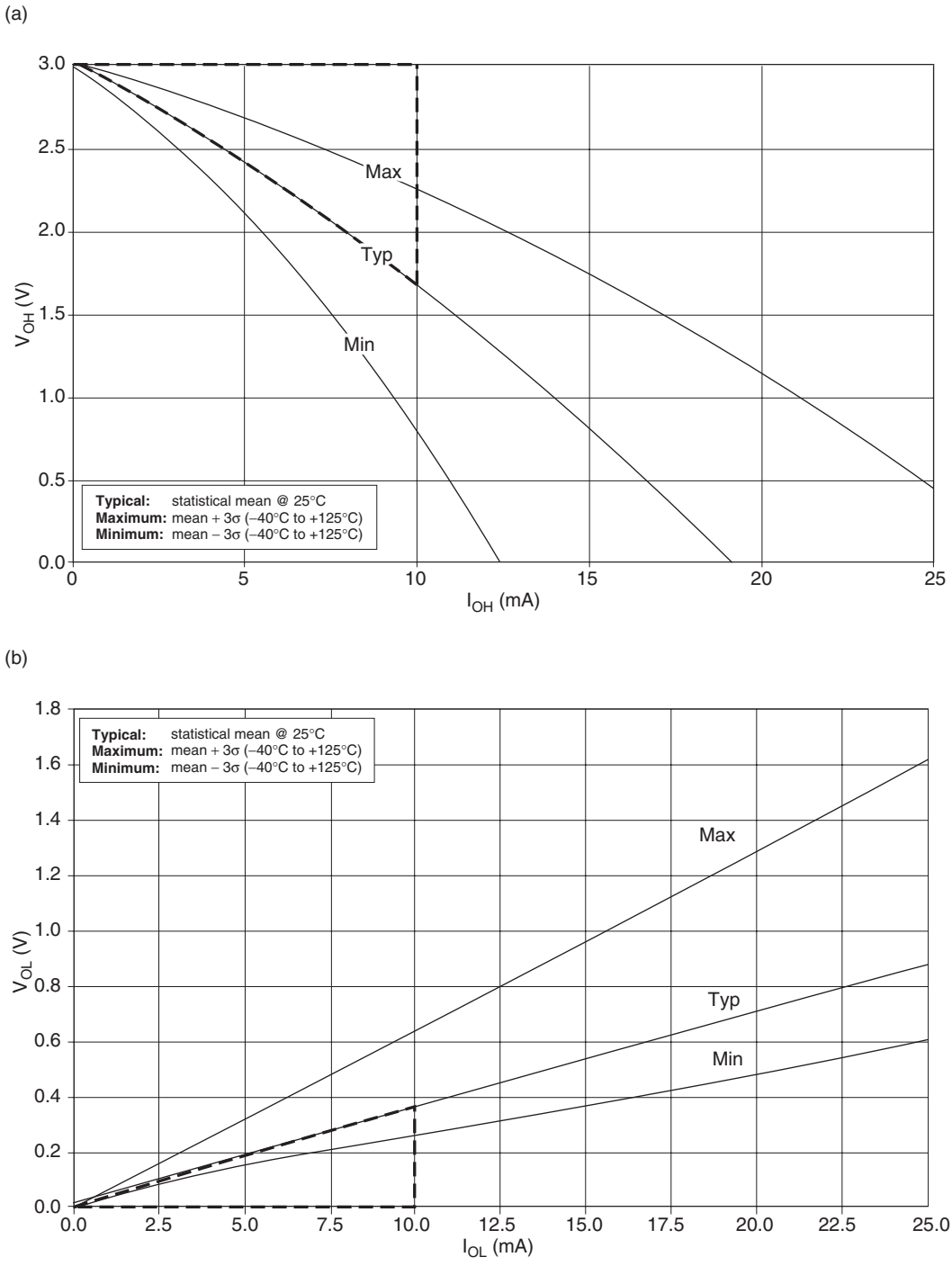


Figure 3.12: 16F84A Port Output Characteristics: (a) V_{OH} *vs.* I_{OH} (V_{DD} = 3 V, −40 to 125°C) and (b) V_{OL} vs. I_{OL} (V_{DD} = 3 V, −40 to 125°C) (Dashed Lines Added by the Author)

3.5.1 Clock Oscillator Types

Broadly, there are two types of oscillator circuit in common use in microcontrollers, as illustrated in Fig. 3.13. In the resistor–capacitor (RC) type (Fig. 3.13(a)), a capacitor is charged through a resistor from the supply rail. The capacitor voltage drives the input of a Schmitt trigger buffer. When the Schmitt trigger threshold is exceeded, its output goes high, switching on the MOSFET transistor to which it is connected. The capacitor is quickly discharged, the Schmitt output goes low, the MOSFET is switched off and the charging process starts again. This continues for as long as power is maintained. The clock signal is taken from the rectangular waveform generated at the Schmitt output. This simple circuit is integrated onto many larger ICs requiring a clock signal. Users are then usually required to connect resistor and capacitor externally, choosing these to set the desired frequency. It is important to note, however, that RC oscillators can be implemented entirely on-chip. RC oscillators are very low cost and produce a clock signal very reliably. As resistor, capacitor, power supply and Schmitt trigger threshold values all vary with temperature, their frequency is not very stable. They cannot therefore be used where precise timing is required.

The crystal oscillator (Fig. 3.13(b)) depends on the piezo-electric properties of quartz crystal. Any mechanical distortion of the material causes a voltage to be produced across opposite sides of it; similarly, if a voltage is applied to the material, a mechanical distortion results. Crystals are carefully cut into very thin slices (usually discs), have tiny electrodes attached and are mounted so that they can vibrate. When connected in the feedback path across a logic inverter, as the figure shows, the crystal can be forced through piezo-electric action into mechanical vibration. This translates into electrical oscillation, an oscillation that is sustained by the action of the logic gate. Small value capacitors connected from either side of the capacitor to ground optimize the electrical conditions needed for this oscillation.

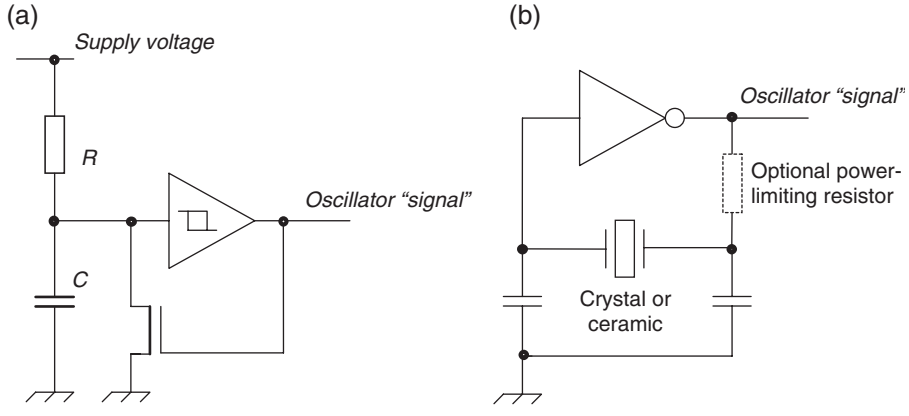


Figure 3.13: Microcontroller Oscillator Generator Circuits: (a) Resistor–Capacitor (RC) and (b) Crystal or Ceramic

Crystal vibration occurs at a fixed and remarkably stable frequency—this is the great advantage of the crystal oscillator. The crystals themselves tend to be on the expensive side (although cost continues to fall) and mechanically fragile. An alternative is the ceramic resonator. This has similar piezo-electric properties to the crystal and is connected in an identical way. It is, however, both lower in cost and rather less stable in frequency. Crystals are the only option when precise timing functions, derived from the clock oscillator, are required.

3.5.2 Practical Oscillator Considerations

All microcontroller manufacturers go a long way to making it easy to create a clock waveform for their microcontrollers. Usually, this is done by including the circuits of Figure 3.13, possibly in merged form, on-chip. One may be forgiven, therefore, for thinking that setting up the oscillator on a microcontroller is a straightforward thing—in fact, it isn't, and unreliable or nonfunctioning oscillators are a cause of real frustration with novice builders. Oscillator frequency shows greater or lesser dependence on supply voltage, temperature, humidity, printed circuit board (PCB) layout and possibly other factors. Crystals in particular are sensitive to poor PCB layout. It is important to exclude parasitic resistance, capacitance or inductance by having very short PCB tracks, therefore locating the crystal close to the body of the microcontroller.

3.5.3 The 16F84A Clock Oscillator

The 16F84A can be configured to operate in four different oscillator modes, allowing implementation of RC, crystal or ceramic oscillators. These are detailed below. It can also accept an external clock source. The user selects which mode is to be used by setting bits in the Configuration Word (Fig. 2.6).

- *XT—crystal*. This is the standard crystal configuration. It is intended for crystals or resonators in the range 1–4 MHz.
- *HS—high speed*. This is a higher drive version of the XT configuration. It recognizes that higher frequency crystals, and ceramic resonators in general, require a higher drive current. It is intended for crystal frequencies in the region of 4 MHz or greater, and/or ceramic resonators. It leads to the highest current consumption of all the oscillator modes.
- *LP—low power*. This mode is intended for low-frequency crystal applications and gives the lowest power consumption possible. In many cases this will be 32.768 kHz (i.e., 2^{15}), which is the most popular frequency for low-power, time-sensitive applications, for example wristwatches. It will, however, operate at any frequency below around 200 kHz.
- *RC—resistor–capacitor*. For this an external resistor and capacitor must be connected to pin 16, replicating the circuit of Fig. 3.13(a). This is the lowest cost way of getting an

oscillator, but should not be used when any timing accuracy is required. The nominal frequency of oscillation can be predicted with limited accuracy only, and even then it will drift with changing temperature, supply voltage and time. An example of use of the RC oscillator appears in the electronic ping-pong case study at the end of this chapter.

As seen in Fig. 2.1, the 16F84A has two oscillator pins, OSC1 (pin 16) and OSC2 (pin 15). Between these lies a logic inverter and associated circuitry. Figure 3.14 shows the possible oscillator configurations that can be connected using these pins. Either a crystal or a ceramic can be connected to create the oscillator circuit of Fig. 3.14(a). Any of the three speed ranges outlined above can be invoked through the Configuration Word. An RC oscillator can also be used, as shown in Fig. 3.14(b). The approximate oscillation frequency can be selected by consulting graphical information given in the Electrical Characteristics section of the data sheet, for example as seen in Fig. 3.15. Finally, an external clock source can simply be connected to the OSC1 pin (Fig. 3.14(c)). Further guidance on oscillator design for Microchip microcontrollers can be found in Ref. 3.2.

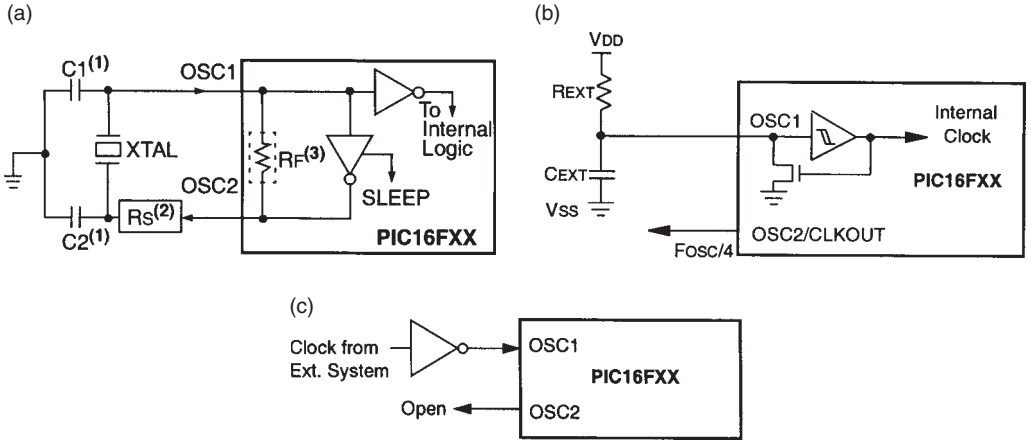


Figure 3.14: Ways of Supplying a Clock Waveform to the 16F84A: (a) Crystal or Ceramic, HS, XT or LP, (b) Resistor–Capacitor, and (c) Externally Supplied Clock

3.6 Power Supply

3.6.1 The Need for Power, and its Sources

Like any electronic circuit, a microcontroller and the overall embedded system need to be supplied with electrical power. Traditionally, much logic circuitry is supplied at 5 V, arising from the voltage specified for the TTL (Transistor Transistor Logic) logic family. With the growth in battery-powered equipment and developments in electronic technology, supply voltages have been pushed down, and 3.3 and 3.0 V supplies are now common.

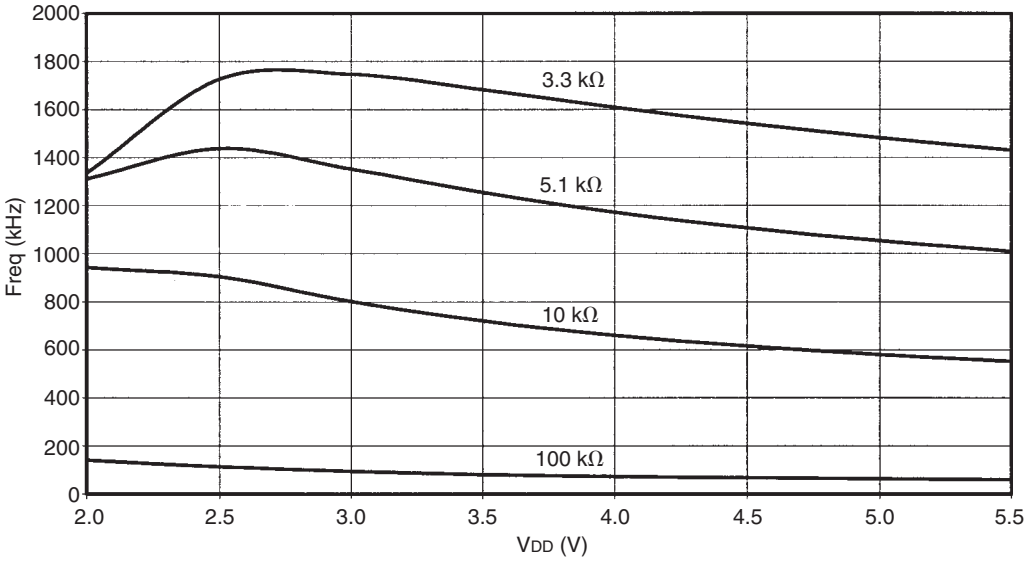


Figure 3.15: Average RC Oscillator Frequency vs. V_{DD} for variable *R*, *C* = 100 pF, 25°C

Operating conditions for electronic components are specified in the manufacturer's data sheet. In terms of power supply there are two important issues: the supply voltage required and the current that the device will then take from the supply. This supply current will be dependent on operating frequency. Also given are *absolute maximum ratings*, which give voltage and power dissipation levels beyond which the device must not be taken.

3.6.2 16F84A Operating Conditions

The essential operating conditions of the 16F84A are shown in Fig. 3.16. From this it can be seen that a supply voltage of between 4.0 and 5.5 V is required, unless the HS oscillator mode is used. In this case the supply voltage must not be below 4.5 V. In Sleep mode (when all program execution is suspended and the oscillator is switched off), the supply voltage can be dropped right down to 1.5 V and the data in RAM is still retained. If operation from lower supply voltages is required, then the 16LF84A should be used.

Looking further down the table, we see how much supply current depends on oscillator frequency. A typical supply current of 1.8 mA can be expected when running at 4 MHz with a supply voltage of 5.5 V. If the oscillator frequency is increased to 20 MHz, then the supply current rises to 10 mA. It's worth mentioning that both these values are actually very good, and compare well with many other, more power-hungry microcontrollers. If we want to operate at really low currents, however, then look what the 16LF84A offers at low frequency—a staggering 15 μA!

Param No.	Symbol	Characteristic	Min	Typ†	Max	Units	Conditions
	VDD	**Supply Voltage**					
D001		16LF84A	2.0	—	5.5	V	XT, RC, and LP osc configuration
D001		16F84A	4.0	—	5.5	V	XT, RC and LP osc configuration
D001A			4.5	—	5.5	V	HS osc configuration
D002	VDR	**RAM Data Retention Voltage (Note 1)**	1.5	—	—	V	Device in SLEEP mode
D003	VPOR	**VDD Start Voltage** to ensure internal Power-on Reset signal	—	Vss	—	V	See section on Power-on Reset for details
D004	SVDD	**VDD Rise Rate** to ensure internal Power-on Reset signal	0.05	—	—	V/ms	
	IDD	**Supply Current (Note 2)**					
D010		16LF84A	—	1	4	mA	RC and XT osc configuration **(Note 4)** FOSC = 2.0 MHz, VDD = 5.5V
D010		16F84A	—	1.8	4.5	mA	RC and XT osc configuration **(Note 4)** FOSC = 4.0 MHz, VDD = 5.5V
D010A			—	3	10	mA	RC and XT osc configuration **(Note 4)** FOSC = 4.0 MHz, VDD = 5.5V (During FLASH programming)
D013			—	10	20	mA	HS osc configuration (PIC16F84A-20) FOSC = 20 MHz, VDD = 5.5V
D014		16LF84A	—	15	45	µA	LP osc configuration FOSC = 32 kHz, VDD = 2.0V, WDT disabled

Note 1: This is the limit to which V_{DD} can be lowered without losing RAM data.
Note 2: Gives further information on factors that influence supply current.
Note 3: Gives guidance on how to calculate current consumed by the external RC network, when this is used.

Figure 3.16: The 16F84A Basic Operating Conditions

You may recognize that, for a battery-powered system, the required supply voltage of the 16F84A makes a three-cell alkaline battery supply a useful option. This gives a supply of around 4.5 V. Suppose you powered with three AA cells, each with a nominal capacity of 800 mAh. Running at 1.8 mA would give a battery life of 444 hours, or 18.5 days. Running at 10 mA would give 80 hours, or 3.3 days, while 15 µA consumption would lead to 53,333 hours, or 2222 days, or just over six years! In this case battery self-discharge would potentially be significant. The above calculations of course only take account of the consumption of the microcontroller, and not of any other parts of the circuit.

An important opportunity for conserving power is through the Sleep mode.

3.7 The Hardware Design of the Electronic Ping-Pong

This little game, shown in Fig. 3.17, is one of several projects used to illustrate the material of this book. It is a game for two players, who each have a push button paddle. Either player

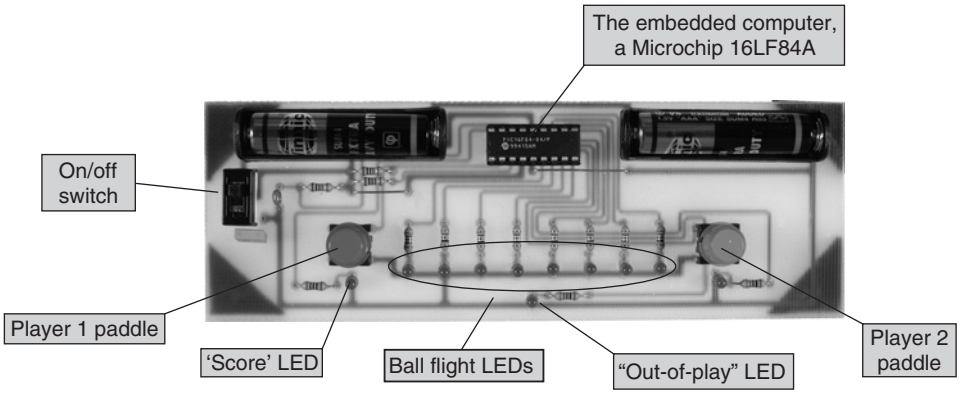


Figure 3.17: The Electronic Ping-Pong

can start the game by pressing his/her paddle. The ball, represented by the row of eight LEDs (light-emitting diodes), then flies through the air to the opposing player, who must press his paddle only when the ball is at the end LED and at none other. The ball continues in play until either player violates this rule of play. Once this happens, the nonviolating player scores and the associated LED is briefly lit up. When the ball is out of play, an "out-of-play" LED is lit. All of the action is controlled by a PIC16LF84A.

The circuit diagram can be seen in Appendix B, Fig. B.1. We are now in a position to understand every detail of its circuit design. Power is supplied from two AAA cells, which are connected to the V_{SS} and V_{DD} pins of the microcontroller via an on–off switch. Because the power supply is only 3 V, an LF version of the microcontroller is used. A 100 nF decoupling capacitor across the power supply smoothes voltage spikes that may be induced due to the action of the microcontroller internal circuitry. **MCLR** is simply tied to the supply rail, as no Reset function is needed for this simple game.

It can be seen that an RC oscillator is used. This is reasonable, as it is a cost-conscious application, with no time-critical elements. Figure 3.15 shows that, for the values used, and with a supply voltage of 3.0 V nominal, the oscillator frequency will be 800 kHz.

Let us now look at the use of the parallel ports. It can be seen that the two player paddles, connected to bits 3 and 4 of Port A, follow the pattern of Fig. 3.7(b), with 10k pull-up resistors. The score and out-of-play LEDs take up the remaining bits of Port A, and the "ball flight" LEDs are all connected to Port B. All LEDs are high-efficiency types and are connected according to Fig. 3.9(a). Noting the approximate 130 output resistance derived in Section 3.4.3 of this chapter, the total resistance in series with each LED is (560 + 130). With a forward voltage across the LED of around 1.8 V, the current is given by applying equation (3.3), i.e.

$$I = (3 - 1.8)/(560 + 130)$$
$$= 1.7 \text{ mA approx.}$$

This current value is just adequate for this type of application and LED, where only close-up viewing is expected. It would in general be viewed as low.

3.8 Summary

- The parallel port allows ready exchange of digital data between the outside world and the controller CPU.
- It is important to understand the electrical characteristics of the parallel port and how they interact with external elements.
- While there is considerable diversity in the logic design of ports, they tend to follow similar patterns. The internal circuitry is worth understanding, as it leads to effective use of ports.
- The 16F84A has diverse and flexible parallel ports.
- A microcontroller needs a clock signal in order to operate. The characteristics of the clock oscillator determine speed of operation and timing stability, and strongly influence power consumption. Active elements of the oscillator are usually built in to a microcontroller, but the designer must select the oscillator type, and its frequency and configuration.
- A microcontroller needs a power supply in order to operate. The requirements need to be understood and must be met by a supply of the appropriate type.

References

[3.1] Kingbright Elec. Co. Ltd. Taiwan; http://www.kingbright.com.tw

[3.2] Overview, Design Tips and Troubleshooting of the PICmicro™ Microcontroller Oscillator (2001). Microchip Technology Inc., Reference no. DS33023A; www.microchip.com

SECTION II

Programming PIC Microcontrollers Using Assembly Language

This page intentionally left blank

CHAPTER 4

Starting to Program—An Introduction to Assembler

Embedded system design is made up of two main aspects, the hardware and the software. In the early days of microprocessors, systems were built up laboriously using a large number of integrated circuits (ICs). Memory was very limited, so only small programs could be written. Slowly, the available ICs became more and more sophisticated, and the designer had to do less to get a working hardware system. Meanwhile memory was growing, so longer programs could be written. Now we are in a situation where memory is plentiful and cheap, and the hardware is sophisticated and readily available. Complex hardware systems can be built up with comparative ease, and in many projects software development is now the main creative activity. In this chapter we start down the long but exciting road to developing good programs. We start that road using the Assembler programming language.

We have one problem if we are to start programming. What will the program run on? Ultimately, of course, embedded systems programs are written to run on the target system hardware. You may be working with an educational PIC® hardware system, or something else. In many cases, however, we don't want to be dependent on hardware to try out a programming idea. What can really cause a study of programming to spring to life is a simulator—a program running on a desktop computer that will run the program we have developed. Therefore, we make it a priority in this chapter to introduce the Microchip MPLAB® Integrated Development Environment, and the simulator in it. Once you have the skill to use this, then most program ideas can be tried out very quickly, and you should be able to make rapid progress in the noble but tricky art of microcontroller programming!

In this chapter you will learn about:

- Some aspects of the underlying issues of computer programming
- The essentials of assembler programming and how to write simple assembler programs
- Development environments for programming and the Microchip MPLAB Integrated Development Environment
- The PIC 16 Series instruction set in overview

- The use of certain PIC 16 Series instructions
- Simulating software and the MPLAB software simulator MPSIM™.

You will also, if you wish, be able to learn about:

- How the RISC instruction set of the PIC 16 Series compares with the instruction set of a comparable CISC microcontroller
- The details of how the PIC 16 Series instruction word is constructed.

4.1 The Main Idea—What Programs Do and How We Develop Them

The four main ideas of computer programming, according to this author, are listed here:

1. A computer has an *instruction set*; it can recognize each instruction and *execute* it.
2. The program that the computer executes is a list of instructions drawn from its instruction set; it reads these in binary from its program memory. The program in this form is called *machine code*.
3. To execute, the computer works relentlessly through the instructions of the program, from the beginning, doing exactly what each instruction tells it to do—nothing more, nothing less—except when temporarily diverted by an interrupt.

So far this is simple, but here is the difficult one:

4. The programmer must find a means of breaking down and translating his/her ideas into steps that the computer can undertake, where each step ultimately must be an instruction from its instruction set.

4.1.1 The Problem of Programming and the Assembler Compromise

The problem of programming is summarized in Fig. 4.1. We as humans express our ideas in complex and often loosely defined linguistic forms. A computer reads and "understands" binary, and responds in a precise way to precise instructions. It is ruthlessly logical and does exactly what it is told.

Given this linguistic divide, how can a programmer write programs for a computer? Three ways of bridging the gap present themselves:

1. *The human learns machine code*. This is what programmers used to do sometimes in the very early days, laboriously writing each instruction in the binary code of the computer, exactly as the computer would then read it. This is incredibly slow, tedious and error-prone, but at least the programmer relates directly to the needs and capabilities of the computer.

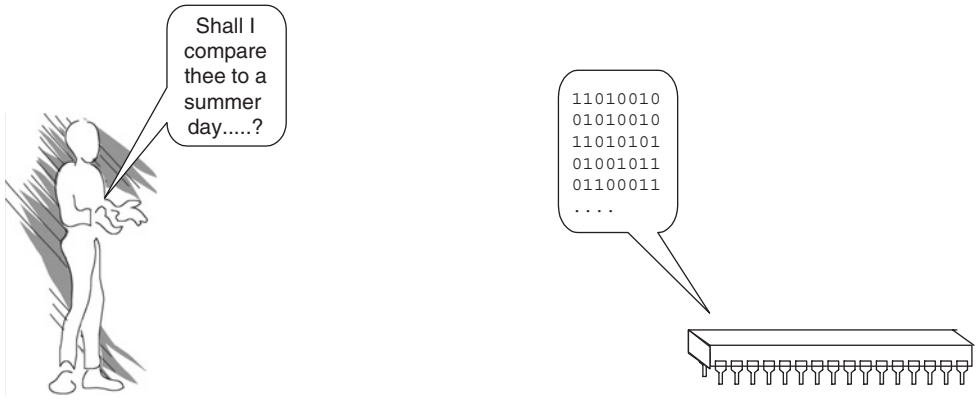


Figure 4.1: The Problem of Programming

2. *Use a high-level language (HLL).* This is as if we go some way to asking the computer to learn our language. In an HLL, instructions are written in a form that relates in a recognizable way to our own language—in the case of people reading this book, probably English. Another computer program, either a compiler or an interpreter, then converts that program into the machine code that the computer can comprehend. The programmer now has a much easier time and can write very sophisticated programs. He/she is now, however, separated from the resources of the computer, and the program may be comparatively inefficient in terms of its use of memory and in its execution speed.

3. *Use assembler (assembly language).* This is a compromise position. Every one of the computer's instruction set is given a mnemonic. This is usually a three-or four-letter word that can be used to represent directly one instruction from the instruction set. The programmer then writes the program using the instruction mnemonics. The programmer has to think at the level of the computer, as he/she is working directly with its instructions, but at least the programmer has the mnemonics to use, rather than actually working with the computer machine code. A special computer program called a cross-assembler, usually these days running on a PC, converts the code written in mnemonics to the machine code that the computer will see. Because there is a computer doing the conversion from the assembler code to machine code, a number of other benefits can be built into the process. For example, the cross-assembler can look after most of the business of allocating memory space in program memory, and it can accept labels for numbers and memory locations, greatly easing the programmer's task.

In the early days of computing, assembler was used to program almost any type of computer. These days, however, it is pretty much the preserve of embedded designers, particularly when using smaller 8-bit devices. For the embedded designer assembler offers the huge advantage

that it allows him/her to work directly with the resources of the computer, and leads to efficient code, which executes quickly. Because it is so directly linked to the computer structure, working in assembler helps the user to learn the structure of the computer. Programming in assembler has the disadvantage that it is rather slow, error-prone and does not always produce well-structured programs. For now, in order to write simple programs and understand the microcontroller more, we will learn assembler.

4.1.2 The Process of Writing in Assembler

The actual process of writing in assembler is illustrated in Fig. 4.2. The programmer writes in the microprocessor or microcontroller assembly language. This *can* be done using nothing more than a text editor. We will soon recognize the two lines of assembler program in Fig. 4.2 as being from the PIC 16 Series instruction set. The computer he/she is writing on runs the cross-assembler. The terminology *cross*-assembler implies that one computer is assembling code for one of another type, not for itself. Usually, and somewhat confusingly, cross-assembler is shortened simply to assembler. The cross-assembler *assembles* the program—i.e., it converts it from assembler mnemonics into machine code ready for the microcontroller. In Fig. 4.2 the cross-assembler is seen converting the two lines of assembler code into the 14-bit machine code words of the PIC 16 Series. For most microcontrollers there are then special programming tools that can download the program in machine code from the main PC and program it into the microcontroller program memory.

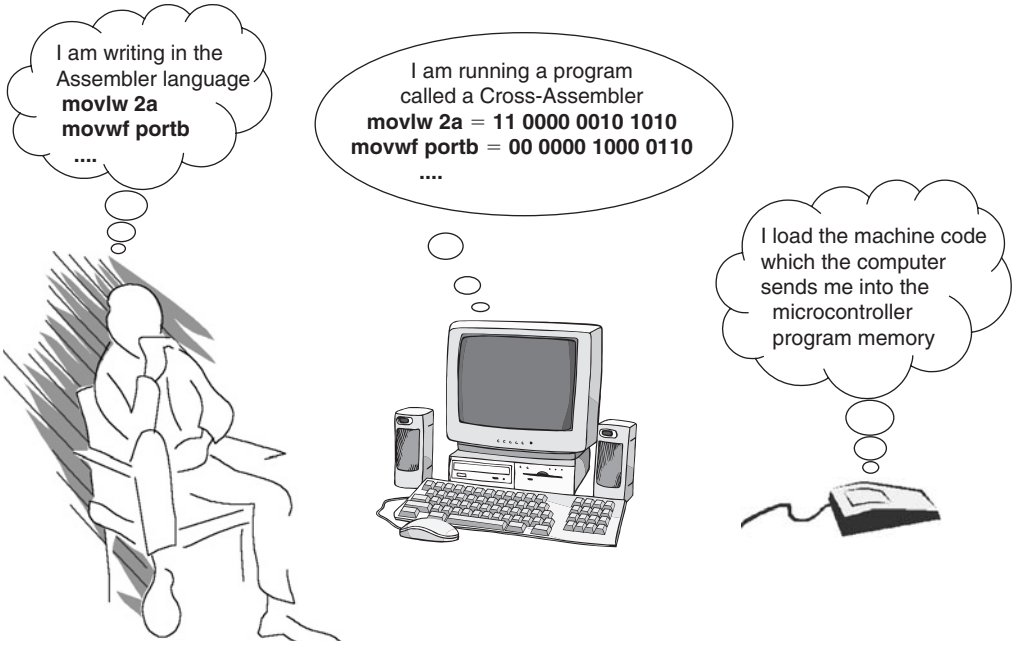


Figure 4.2: Programming in Assembler

4.1.3 The Program Development Process

The process of writing in assembler needs to be placed in the broader context of project development. The possible stages in the development process for the program of a simple embedded system project are shown in Fig. 4.3. The programmer writes the program, called the *source code*, in Assembler language.

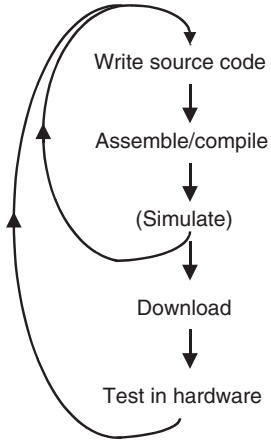


Figure 4.3: Developing a Simple Project

This is then assembled by the cross-assembler running on the host computer. If the programmer has access to a simulator then he/she may choose to test the program by simulation. This is likely to lead to program errors being discovered, which will require alteration to the original source code. When satisfied with the program, the developer will then download it to the program memory of the microcontroller itself, using either a stand-alone "programmer" linked to the host computer or a programming facility designed into the embedded system itself. He/she will then test the program running in the actual hardware. Again, this may lead to changes being required in the source code.

Clearly, to develop even a simple project, a selection of different software tools is beneficial. These are usually bundled together into what is called an *Integrated Development Environment* (IDE).

4.2 The PIC 16 Series Instruction Set, with a Little More on the ALU

4.2.1 More on the PIC 16 Series ALU

Before looking at the 16 Series instruction set, it is worth taking a more detailed look at the ALU (Fig. 4.4). Understanding this will aid in understanding the instruction set. Looking at

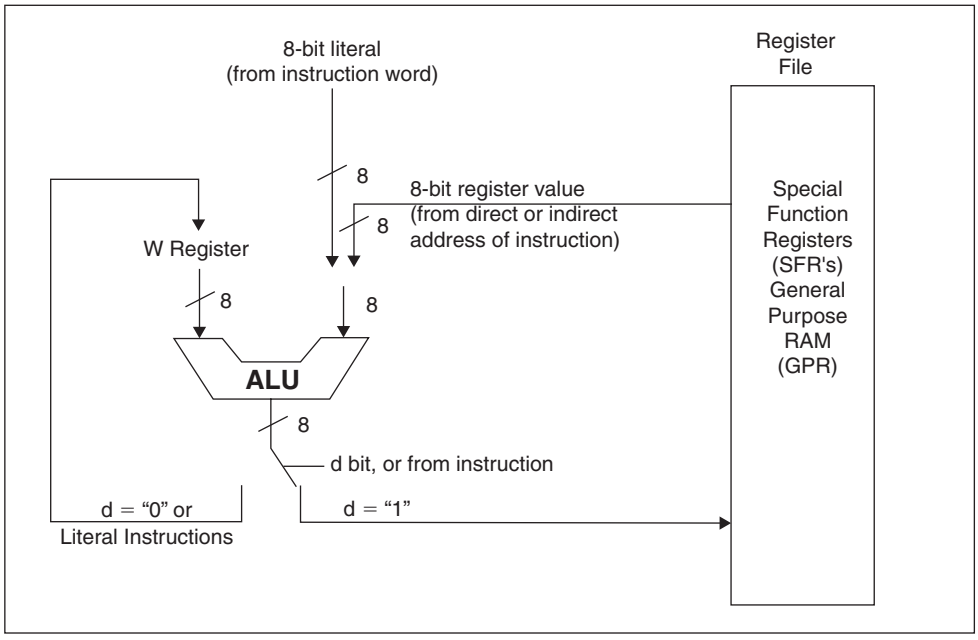


Figure 4.4: Block Diagram of the PIC 16 Series ALU

this, we see that the ALU can operate on data from *two* sources. One is the W (or Working) register. The other is *either* a *literal* value *or* a value from a data memory (whose memory locations Microchip calls *register files*). A literal value is a byte of data associated with a particular instruction that the programmer embeds in the program.

Thus, we can expect to see some instructions that call on data memory and others that require literal data to be specified whenever they are used. Examples of all are coming! The data that the instruction operates on, or uses, is called the *operand*. Operands can be data or addresses. We will see that some types of instructions always need an operand to be specified with them, others do not.

Once an instruction has been executed, where is the result stored? For many instructions Microchip offer a choice, whereby the result can *either* be held in the W register *or* stored back in data memory. Which one is used is fixed by certain instructions; in others it is determined by the state of a special **d** bit, which is specified within the instruction.

4.2.2 The PIC 16 Series Instruction Set—an Introduction

Turn now to the PIC 16 Series instruction set, which can be found in Appendix A. Take a long hard look at it—we are aiming to get to know it extremely well! You can see that the table is divided into six columns, and each of the 35 instructions gets one line. The first column gives

the actual mnemonic, together with the code specifying the type of operand it acts on. There are four such operand codes:

- **f** for file (i.e., memory location in RAM), a 7-bit number
- **b** for bit, to be found within a file also specified, a 3-bit number
- **d** for destination, as described above, a single bit
- **k** for literal, an 8-bit number if data or 11-bit if address.

The second column summarizes what the instruction does. In some cases this gives adequate information. A much fuller description of how each instruction works can also be found in the full microcontroller data [Ref. 2.1]. The third column shows how many instruction cycles the instruction takes to execute. As a RISC processor, we expect this to be a single cycle. This turns out to be the case, apart from those instructions that cause a branch in the program. We discuss their use in Chapter 5. The fourth column gives the actual 14-bit opcode of each instruction. This is the code that the cross-assembler produces, as it converts the original program in Assembler language to machine code. It is interesting to see here how the operand codes, listed above, become embedded within the opcode. The fifth column shows which bits in the Status register (Fig. 2.3) are affected by each instruction.

Let us immediately look at five example instructions, to see how the information is presented. As an aside, let us note now that assembler programming does not have to be case sensitive, and that all the examples in this book are *not* case sensitive. Therefore, do not worry if you see instruction mnemonics and operands appearing in either upper or lower case in different references. In this book, for stylistic reasons, we choose to write assembler programs in lower case. Find now each of the instructions below in the Instruction Set Table in Appendix A:

- **clrw**—this clears the value in the W register to zero. There are no operands to specify. Column 5 tells us that the Status register **Z** bit is affected by the instruction. As the result of this instruction is always zero, the bit is always set to 1. No other Status register bits are affected.
- **clrf f**—this clears the value of a memory location, symbolized as **f**. It is up to the programmer to specify a value for **f**. Again, because the result is zero, the Status register **Z** bit is affected.
- **addwf f,d**—this adds the contents of the W register to the contents of a memory location symbolized by f. It is up to the programmer to specify a value for f. There is a choice of where the result is placed, as discussed above. This is determined by the value of the operand bit d. Because of the different values that the result can take, all three condition code bits, i.e. Z, the Carry bit C, and the Digit Carry bit DC are affected by the instruction.
- **bcf f,b**—this instruction clears a single bit in a memory location. Both the bit and the location must be specified by the programmer. The bit number **b** will take a value from 0 to 7, to identify any one of the 8 bits in a memory location. No Status register

flags are affected, even though it is possible to imagine that the result of the instruction could be to set a memory location to zero.

- **addlw k**—This instruction adds the value of a literal, whose value **k** must be specified by the programmer, to the value held in the W register. The result is stored in the W register; there is no choice. Like **addwf**, all condition code bits can be affected by this instruction.

4.3 Assemblers and Assembler Format

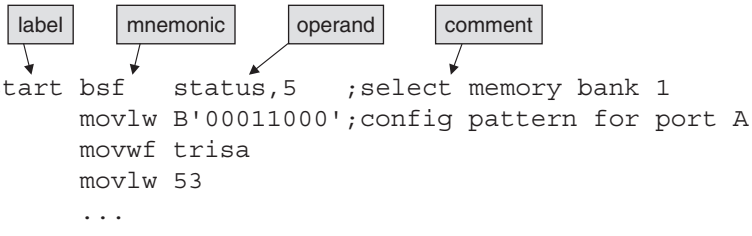
4.3.1 Introducing Assemblers and the Microchip MPASM™ Assembler

For any microprocessor or microcontroller, there are a large number of (cross-) assemblers available. Some are distributed free by the makers of the processors to encourage people to buy their products. Others, usually more sophisticated, are written by specialist software houses and sold commercially. Many these days come as part of an IDE, as mentioned earlier. This book uses MPASM, the assembler offered by Microchip. It is usually used as part of the MPLAB IDE, and both MPASM and MPLAB are introduced in some detail later in this chapter.

While many aspects of assembler programming are common across all cross-assemblers, some are specific to the particular assembler that is in use.

4.3.2 Assembler Format

Having taken a first look at the instruction set, we need now to understand how we can build these instructions into a program. Assembler programs have a simple format, which must be understood and followed. This is shown in Program Example 4.1.



Program Example 4.1: Assembler Format

There are four possible elements to an assembler line of code:

- *Label*. A label for a line is optional. When it is first specified, the label must start in the left-most space of the line. The assembler will interpret anything starting in this space as a label. Once defined in this way, a label can be used as an operand. Labels must start with an alphabetic character or underscore, but not a number. Labels can stand alone on a line, in which case the label is adopted by the next line that contains an instruction.

- *Instruction mnemonic*. This is drawn from the instruction set. It may be placed anywhere on the line, except starting at the far left. It should be separated from any label by at least one blank space.
- *Operand*. These must conform exactly to the format specified in the instruction set. For better intelligibility, labels are often used rather than numbers. If there is more than one operand they are separated by a comma.
- *Comment*. This is optional, and is used to add information to the program and improve its intelligibility to the human reader. A comment must always start with a semicolon. The cross-assembler ignores everything that follows a semicolon in any line. Comments can follow instructions on a line; alternatively, a whole line can be used just for commenting.

A line of the assembler program can contain an instruction, properly formatted as above, it can be a comment only, or it can be left completely blank (this sometimes helps to improve layout and readability).

4.3.3 Assembler Directives

While the assembler program is written for the target microcontroller, it has to be processed by the assembler first. To aid this process and make it more powerful and flexible, a way is needed of passing information and instructions to the assembler, which it recognizes as being for its attention only. These instructions are called *assembler directives*. They are used for very diverse applications, for example defining the target processor or specifying where the program must be placed in memory. A few MPASM examples are shown in Table 4.1. These are written in the code and appear almost like mnemonics from the instruction set. Their very distinct role must, however, be recognized.

4.3.4 Number Representation

One of the features of working close to the inner operations of a microcontroller is that sometimes one is thinking in binary, sometimes in decimal and sometimes in hexadecimal,

Table 4.1: Some Common MPASM Assembler Directives

Assembler Directive	Summary of Action
list	Implement a listing option*
#include	Include additional source file
org	Set program origin
equ	Define an assembly constant; this allows us to assign a value to a label
end	End program block
* Listing options include setting of radix and of processor type.	

or even octal. Therefore, it is helpful for the assembler program to be able to recognize and respond to different number bases. MPASM does this first by allowing a default to be set. Thus, for example, if one wants to work only (or mainly) in hexadecimal, then all numbers can be interpreted as such. Any number that the programmer wants to represent in an alternative radix must be prefixed, as shown in Table 4.2. In Program Example 4.1, the programmer is writing for a default radix of hexadecimal. In the second line of the example, however, he wishes to specify a number in binary, so therefore uses the appropriate format from Table 4.2. In the fourth line he is using the hexadecimal number 53 as an operand. As hexadecimal is the default radix, its number base does not need to be specified explicitly.

Table 4.2: Number Representation in MPASM Assembler

Radix	Example Representation
Decimal	`D'255'`
Hexadecimal	`H'8d' or 0x8d`
Octal	`O'574'`
Binary	`B'01011100'`
ASCII	`'G' or A'G'`

Note that a hexadecimal number must not start with an alphabetic character, otherwise it might be interpreted as a label. Therefore, any hexadecimal number starting with a, b, c, d, e or f must be preceded with a zero. Thus, for example, the number b2H must be entered as 0b2.

4.4 Creating Simple Programs

4.4.1 A Simple Data Transfer Program

Let's now look at a simple example program, written for the MPASM Assembler, using the MPLAB IDE. It is shown in Program Example 4.2. The program is written for the electronic ping-pong hardware in Appendix B, but we will soon use it for simulation.

```
;*************************************************************
;ELECTRONIC PING-PONG DATA MOVE
;This program moves push button switch values from Port A to the
;leds on Port B
;TJW 21.2.05                                        Tested 22.2.05
;*************************************************************
;
;Configuration Word: WDT off, power-up timer on,
;                      code protect off, RC oscillator
;     list p=16F84A
;
```

Program Example 4.2: A Simple Data Transfer Program

```
; specify SFRs
status    equ     03
porta     equ     05
trisa     equ     05
portb     equ     06
trisb     equ     06
;
          org     00
;Initialize
start     bsf     status,5       ; select memory bank 1
          movlw   B'00011000'
          movwf   trisa          ;port A according to above pattern
          movlw   00
          movwf   trisb          ;all port B bits output
          bcf     status,5       ;select bank 0
;
;The "main" program starts here
          clrf    porta          ;clear all bits in ports A
loop      movf    porta,0        ;move port A to W register
          movwf   portb          ;move W register to port B
          goto    loop
          end
```

Program Example 4.2: Continued

The program starts with a header made up of five comment lines, each starting with a semicolon. These define the program title, briefly describe what it does, and give the date the program was written and the author. Information on Configuration Word settings (Fig. 2.6), fundamental to the running of the program, is then given. We will find that there is more than one way to get this information programmed into the microcontroller. The first active line of the program follows—all lines to here have been comments. This defines the microcontroller to be used, using the **list** directive.

A section follows that uses the **equ** directive to define the memory locations of the SFRs that will be used. It comes as some surprise to many people that it is necessary to do this. Haven't we just "told" the assembler what the processor is, so shouldn't it "know"? The answer is that it doesn't, so we must supply this information. This program just uses the Status register, Ports A and B, and their control registers **TRISA** and **TRISB**. Labels for these are therefore defined, taking memory addresses directly from the memory map of Fig. 2.5. Remember (Section 2.4.2) that the Bank Select bit is held in the Status register. Once this is removed from the SFR addresses shown in Fig. 2.5, then the labels **porta** and **trisa**, and **portb** and **trisb** have the same values. In the program it would make some sense to use just one label for each of these pairs, instead of the two. We choose not to do this in this program example, for better clarity when the different locations are used.

Before the actual program starts it is essential to use the **org** directive to define the program start address. We have no choice over the address used—it must be the reset vector address, as seen in Fig. 2.4.

The program that follows makes use of seven instructions, all of which manipulate bits and bytes of data, except for one branch instruction. These are:

- **clrf f**—this clears to zero the value in memory location **f**
- **movwf f**—this moves the contents of the W register to the memory location **f**
- **movf f,d**—this instruction moves the contents of the memory location **f** to the W register, *if* the **d** bit is set to 0; if it is set to 1, then the contents of **f** are just returned to **f**
- **movlw k**—this instruction moves the literal value **k**, an 8-bit number which accompanies the instruction, into the W register
- **bcf f,b**—this clears (i.e., sets to Logic 0) the bit **b** in memory location **f**
- **bsf f,b**—this sets to Logic 1 the bit **b** in memory location **f**
- **goto k**—this transfers program execution to the instruction in memory location **k**.

The initialization section of the program follows. This sets up the direction of each bit of the two ports that are used, and this requires access to the port control registers **TRISA** and **TRISB**. As these are placed in RAM memory bank 1, it is necessary first of all to set bit 5 of the Status register to 1 (as explained in Chapter 2). This is done in the first line of actual program, labeled **start**, using the **bsf** instruction. The label **status** can be used, because it was defined earlier in the program. If this had not been done, then it would have been necessary to write:

```
start     bsf   3,5    ;select memory bank 1
```

which would have been somewhat less intelligible.

The port pin directions needed are derived from the circuit diagram (Fig. B.1). From this we can see that the two push-buttons connect to bits 3 and 4 of Port A, which must accordingly be set up as inputs. The three other bits of Port A are all connected to LEDs, so must be set up as outputs. As described in Section 3.4, to be an output a port pin must havea0inits corresponding **TRIS** register bit. It must have a 1 for the bit to be an input. Therefore, we must send the word 00011000 to **TRISA**. Note that **TRISA** is an 8-bit location, even though Port A only has 5 bits. It is therefore necessary to specify a complete 8-bit word to be sent, even for those 3 bits that are not implemented. There is no instruction that allows us to transfer a byte of data directly from the program into a memory location, so two lines of code must be used. First, the required word 00011000 is placed into the W register, using a **movlw** instruction. In doing this, the binary radix

is used (Table 4.2), instead of the default hexadecimal. The contents of the W register are then moved to **trisa** using a **movwf** instruction. A similar process is followed for setting up Port B. A quick look at the circuit diagram shows that all Port B pins are connected to LEDs, so all must be set as output. Therefore, the word sent to **trisb** is 00_H. The initialization section ends with memory bank 0 being selected in the Status register, as from here on the ports themselves will be accessed, whose locations are in bank 0.

Finally, we reach the effective program itself, all five lines of it! The program continuously reads the value of Port A and transfers it to Port B. Thus, if either push-button is pressed, this should be seen on the LEDs connected to bits 3 and 4 of Port B. When Port A is read, all of its 5 bits are read, even though three are set as outputs. For these, the values of the internal data latches (Fig. 3.11) are read. All Port A bits are therefore initially cleared to 0 in the program, using a **clrf** instruction.

The actual data transfer part of the program uses a **movf** instruction to shift the value of Port A to the W register, followed by a **movwf** instruction to move the W register value to Port B. A **goto** instruction creates a continuous loop, making use of the earlier defined label **loop**.

It is worth observing here that label values are assigned in two different ways. Some, like **porta** or **portb**, are assigned a specific value by the programmer, using the **equ** directive. Others, like **loop**, are inserted into the program, and the assembler allocates them a value.

The program ends with an **end** directive.

4.5 Adopting a Development Environment

4.5.1 Introducing MPLAB

MPLAB is an IDE that can be downloaded free from Microchip's website. There is also a copy on the book CD. It contains all the software tools necessary to write a program in Assembler, assemble it, simulate it and then download it to a programmer. The latter must be built or bought, or designed in to the target system. Further software tools can be bought and then integrated with MPLAB, both from Microchip and from other suppliers. This includes alternatives to what MPLAB already offers—e.g., assemblers or simulators, as well as tools which offer much greater development power, like C compilers or emulator drivers.

MPLAB is a continuously evolving package, with its own manuals [Refs 4.1, 4.2] and on-line Help facility. Therefore, this book does not aim to act as a full MPLAB manual. It will, however, aim to give a clear introduction to its use, so that you can begin to apply it with confidence. Screen images from MPLAB Versions 7.00 and 7.22 are used in this chapter and the next.

4.5.2 The Elements of MPLAB

MPLAB is made up of a number of distinct elements, which work together to give the overall development environment. These are:

- *Text editor*. This allows entry of the source code. It behaves to some extent like a simple text editor such as Notepad, but it can recognize the main elements of the programming language that is being used. Thus, in Assembler it color-codes instructions in one color, labels in another and comments in a third. In this way the programmer can immediately see if there is a misconception in his placing or use of text within the assembler line.
- *Project manager*. The preferred way of developing programs in MPLAB is by creating a *project*. An MPLAB project groups all the files together that relate to any one project, and ensures that they interact with each other in an appropriate way and are updated as needed.
- *Assembler and Linker*. The function of the assembler has already been discussed. So far we have assumed that there is a single source file. In advanced projects, however, the code may be created from a number of different files. The role of the Linker is to put these together, give each its correct location in memory, and ensure that branches and calls from one file to the other are correctly established.
- *Software simulator and debugger*. A software simulator allows the program that has been developed to be tested, by running it on a simulated CPU in the host computer. Inputs can also be simulated, and outputs and memory values can be observed. The debugger contains the tools which allow program execution to be fully examined, for example by single stepping through the program, or running at slow speed, or halting at a particular location.

Table 4.3: Some File Extensions Used in MPLAB IDE

File Extension	Function
.asm	Assembly language source file
.err	Error file
.hex	Machine code in hex format file
.inc	Assembly language include file
.lib	Library file
.lst	Absolute listing file
.o	Object file
.mcp	Project information file
.mcw	Workspace information file

4.5.3 The MPLAB File Structure

Even with simple projects, a significant number of files are rapidly generated in MPLAB. Each type is designated by the file extension used, examples of which are given in Table 4.3. Whenever a project is set up, files of type **.mcp** and **.mcw** are created. When using assembler, the original source code is written in a file with the **.asm** extension. The source code may include an **.inc** file. When the source code is assembled successfully, the output appears in **.lst** and **.hex** files. If there is an error, that is placed in an **.err** file.

4.6 An Introductory MPLAB Tutorial

This tutorial takes you through the stages of creating a project, writing simple source code and assembling that to create output files. To follow the tutorial, you should download and install the current version of MPLAB, if it is not available in your place of work or study.

Open the MPLAB IDE, which should appear as in Fig. 4.5. If a blank Output window also opens, close it. The main screen is blank, apart from the Workspace window at top left, which cannot be closed.

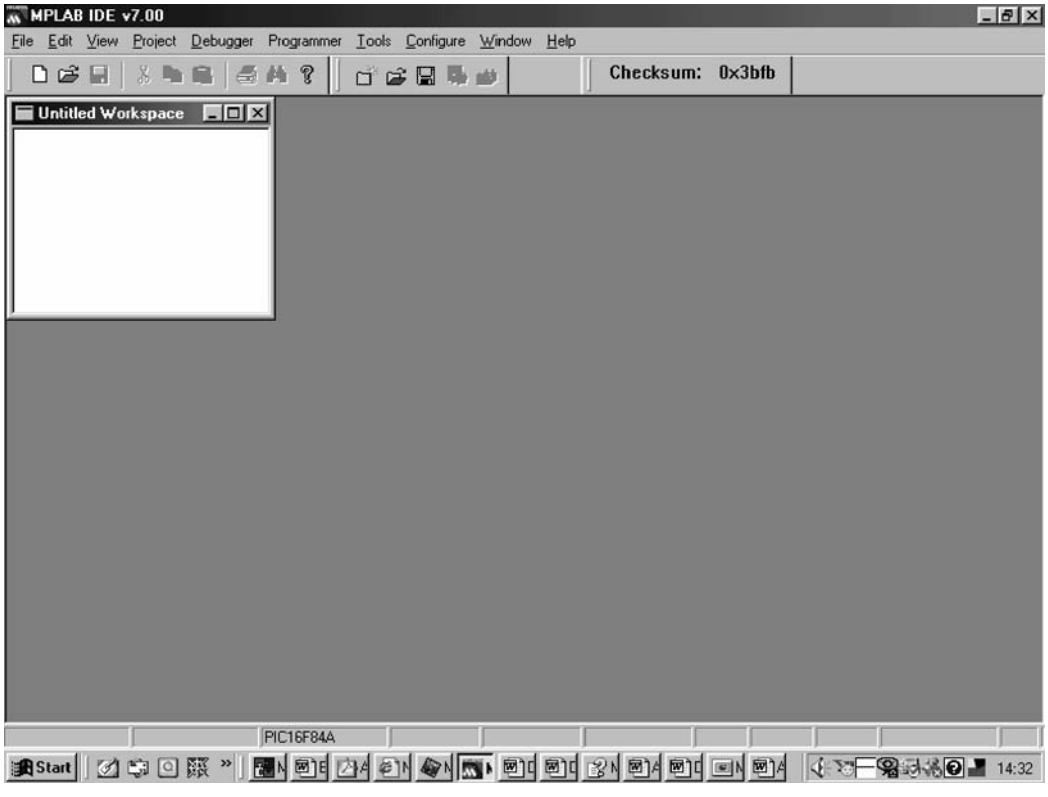


Figure 4.5: The MPLAB IDE Screen

4.6.1 Creating a Project

Click the Project button on the toolbar to access the pull-down menu, as shown in Fig. 4.6.

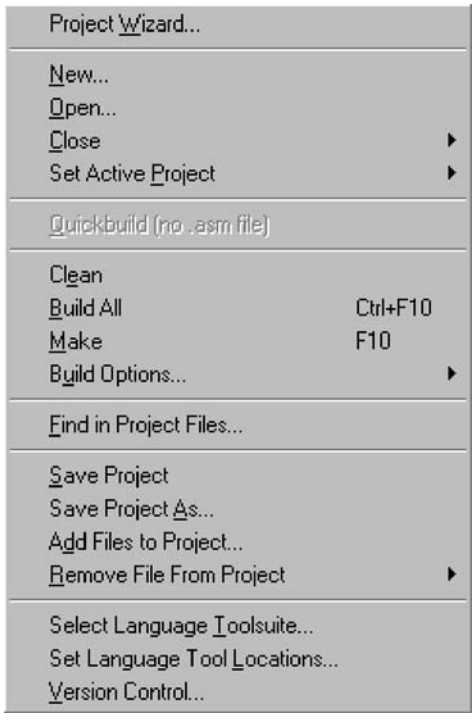


Figure 4.6: Project Pull-Down Menu

There are two ways to create a project, both accessible from this menu. One is by using the **Project Wizard** and the other by selecting **New. . .**. Try following the Project Wizard route, making the following selections as you work through the dialogue boxes:

Device: **PIC16F84A**
Active Toolsuite: **Microchip MPASM Toolsuite**
which will display:
MPASM Assembler
MPLINK Object Linker
MPLIB Librarian
as Toolsuite contents

Project Name <your own choice>
Project Directory <your own choice>

Add existing files. . . Make no additions

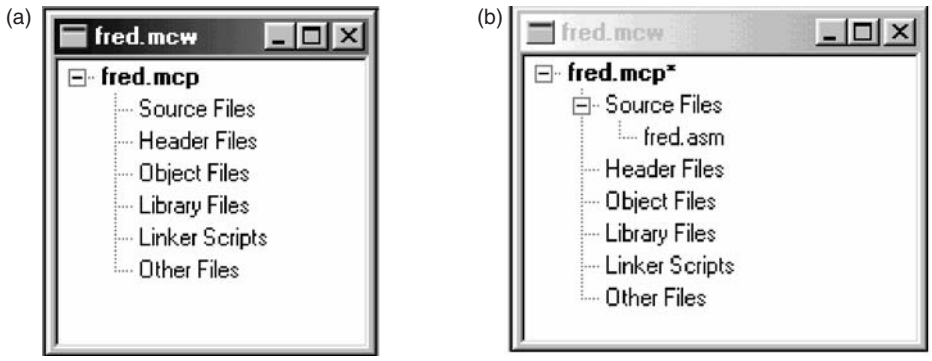


Figure 4.7 Workspace Window: (a) For a Newly Created Project and (b) For a Project with Source file Added

When you click **Finish**, the Workspace window should be updated to show the filename you have selected, as seen in Fig. 4.7(a), for a project called **fred**.

4.6.2 Entering Source Code

Now open a new file by clicking **File** > **New**, and start to enter into it Program Example 4.2. After a few lines save this using **File** > **Save As. . .**. Select file type **Assembly Source File**, and save as <*your project name*>**.asm**. Continue entering the code and notice now that MPLAB has identified this file as an Assembler Source File. It applies color-coding to label, instruction mnemonic, numerical data, Assembler directive and comment. When complete, go to the Project menu again, click **Add Files to Project. . .** and select the one you have just saved. Your Workspace window should now appear as in Fig. 4.7(b), with of course your own file name. You will now begin to appreciate how valuable this window is to become, as it shows a complete picture of the files associated with your project.

4.6.3 Assembling the Project

Now comes one of the testing moments in any project development. You have entered new source code, and you need to know if it assembles correctly. The assembler subjects your code to a series of checks. It returns errors if it finds incorrect use of assembler format, instruction mnemonics, labels or a range of other things. Remember, however, that the assembler can effectively only check that your program is correct grammatically; it cannot assure you that it is a viable program. Above all else, it has no knowledge of the target hardware, beyond the fact that the microcontroller has been specified. Correct assembly does not guarantee correct program operation!

Check that the default radix is correctly set by clicking **Project** > **Build Options** > **Project** > **MPASM Assembler** and ensuring that **Hexadecimal** is selected in the dialogue box. In the same dialogue box you can enable or disable case sensitivity for all the source code. This is not necessary if you have directly copied Program Example 4.2. You may need to use it in future, however.

Invoke the MPASM Assembler by pressing **Project** > **Build All**. This also ensures all files are updated as needed. The **Output** window will open, reporting on the progress of the build. In the Output window you will either get a "Build succeeded" message or a "Build failed," together with a fleeting box showing a green (for success) or red (for errors) bar.

Whether your build has initially succeeded or failed, open the file <*your project name*>**.lst**. This should be in the directory you specified for the project. Use **File** > **Open**, and ensure you select **All** Files in the dialogue box against **Files of Type**. The **.lst** file is very informative and gives you the original source code, alongside the assembled machine code, together with any errors and warnings that may have been generated. Part of the list file for Program Example 4.2 is shown in Program Example 4.3. Notice both how the machine code is represented and that, after the main program listing, there is essential underlying program information.

```
                00029     ;The "main" program starts here
0006   3000    00030       movlw 00   ;clear all bits ports A and B
0007   0085    00031       movwf porta
0008   0086    00032       movwf portb
0009   0805    00033 loop  movf    porta,0  ;move port A to W register
000A   0086    00034       movwf   portb    ;move W register to port B
000B   2809    00035       goto    loop
                00036       end

MPASM 03.90 Released              DATA MOVE.ASM   3-10-2005
15:55:03      PAGE 2
SYMBOL TABLE
   LABEL              VALUE
__16F84A             00000001
loop                 00000009
porta                00000005
portb                00000006
start                00000000

status               00000003
trisa                00000005

trisb                00000006
```

. . . . *etc*

Program Example 4.3: Part of the Data_Move List File

If you have errors, each will be accompanied by an error number and message in the list file. In most cases these are simple typographical errors that can be easily fixed by correcting the source code and building again. If an error proves difficult, then full details on the error should be sought, either in the Help menu or in Ref. 4.1, by looking up its number. At the end of a development session, close the current project using **Project > Close**.

Once you have a source file that builds correctly, you are in a position either to download to microcontroller memory or to simulate. We continue now on the simulation path.

4.7 An Introduction to Simulation

The following subsection introduces the MPLAB simulator, MPSIM™, by means of a tutorial, simulating the program that has just been assembled.

4.7.1 Getting Started

Once in MPLAB, select the simulator by invoking **Debugger > Select Tool > MPLAB SIM**. The simulator menu, as seen in Fig. 4.8, then appears under Debugger.

Figure 4.8: Simulator Menu

4.7.2 Generating Port Inputs

This program applies the ping-pong hardware, so to simulate we will need to create simulated inputs for the two ping-pong paddles, on Port A, pins 3 and 4. There are two ways of generating

simulated inputs, which depends on whether the input is to be synchronous with instruction execution or not. We will choose the simpler—asynchronous under user control.

Select **Debugger** > **Stimulus Controller** > **New Scenario**. Explore the dialogue box that appears—it allows you to set up different types of inputs at the Port pins, which are initiated by pressing the Fire button at the appropriate moment. Under **Pin**, select **RA3** and then **RA4**, with Toggle under Action for each. When you close the project, your settings are saved as a **.stc** file.

4.7.3 Viewing Microcontroller Features

You can observe a number of microcontroller features during simulation, including program memory, SFRs, data memory and so on. A window can be opened for each of these, using the **View** menu. If you do this, however, you will find that the screen very quickly becomes cluttered. A Watch window allows you to make selections of only those variables you want to watch, while leaving out the others. Items for the **Watch** window are selected by using the pull-down menus at the top of the window. Open a Watch window, and select **PCL**, **TRISA**, **PORTA**, **TRISB** and **PORTB**. Looking ahead, the Watch window will appear as seen in Fig. 4.10.

4.7.4 Resetting and Running the Program

You can reset the simulated CPU either by the F6 button, or by using **Debugger** > **Reset**. Using the latter, four Reset categories are offered, reflecting the Reset capabilities of the PIC microcontroller (Section 2.8). Alternatively (and more simply), you can use the Reset button of the Debugger toolbar (Fig. 4.9). If this is not displayed, invoke it by **View** > **Toolbars** > **Debug**.

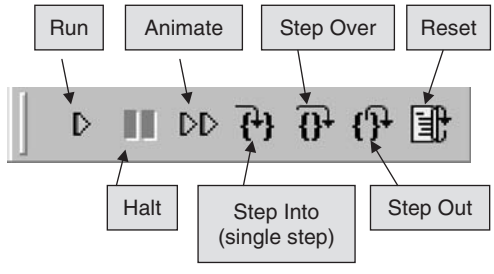


Figure 4.9: MPSIM Software Simulator Debugger Toolbar

There are three ways to run the program. Each can be selected under the **Debugger** pull-down menu or by selecting the buttons on the toolbar. These are:

- *Single Step*. This allows you to step through the program one instruction at a time. This version of MPLAB uses the terminology **Step Into** for this mode.

- *Animate*. This is like an automated single step. The program runs slowly but continuously, with the screen being updated after each instruction. The speed it runs at can be set by invoking **Debugger > Settings > Debugger Animation**.
- *Run*. This runs the program, but does not update on-screen windows as it runs. It does, however, accept stimulus input.

It is also possible to **Step Over** a subroutine or **Step Out** of one. Each of these also has a button on the toolbar. These are especially useful for delay routines, which on a simulator may take an unacceptably long time to simulate.

If you have followed the instructions to here, you should have a computer screen similar to that shown in Fig. 4.10, although you are likely to have arranged the windows differently. In this image we see the Watch window top left, the Stimulus Controller top right and the source file bottom left. The simulated CPU has been reset, so the arrow representing the Program Counter is pointing to the first instruction. This can be confirmed by checking the **PCL** value in the Watch window.

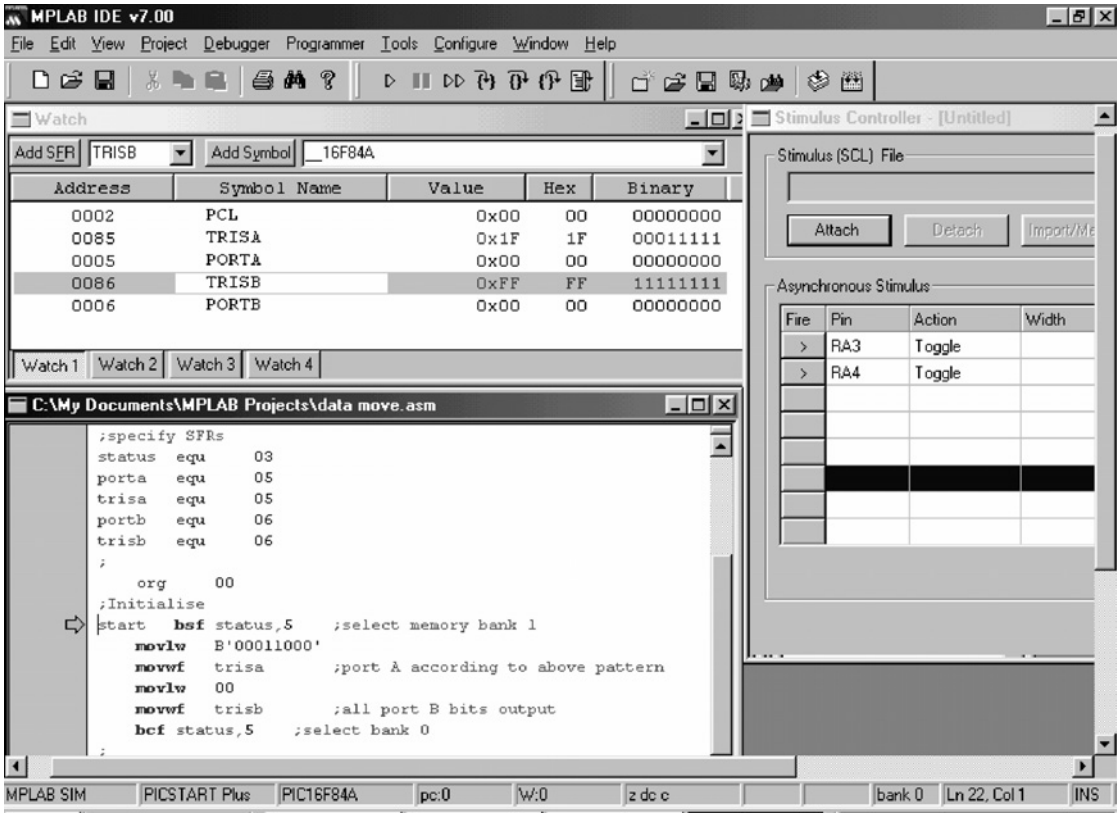


Figure 4.10: MPSIM Set-up for a Simple Simulation

Now, by repeated pressing of the **Step Into** button, single-step through the program. As the program moves through the initialization, you will see the SFR values being changed in the Watch window and the **PCL** value being incremented. Program execution then circles around the loop, formed by the last three instructions. Now try "firing" RA3 or RA4. Display windows are not updated with the value you have forced, until the next instruction execution. Observe Port A and Port B being updated in value, as you continue to execute the program with single-stepping. Try now pressing the **Animate** button. Notice that the program runs continuously, but still responds to stimulus inputs.

4.8 Downloading the Program to a Microcontroller

Most modern microcontrollers are equipped with on-chip program memory using flash technology. The process of programming requires data to be transferred into the chip in a precisely timed way and certain programming voltages to be applied, usually higher than the normal supply voltage. Certain microcontroller pins therefore usually have a secondary function, being used in programming mode to transfer the program data onto the chip and transmit the programming voltages.

In times past, the process of programming always used to require the IC carrying the memory (whether a stand-alone device or memory in a microcontroller) to be placed in a *programmer*. This was linked to a desktop computer for the process to be carried out. As memory technology has improved, however, the process has become simpler and it has become increasingly easy to design the necessary programming circuitry into the target system. This means that many microcontrollers can now be programmed *in situ*, i.e., within the target system. We will see these techniques in later chapters. In this chapter we will stay with traditional programmers, which require the microcontroller to be removed from the target circuit and placed into the programmer.

A popular and low-cost programmer, supplied by Microchip, is the PICSTART® Plus, shown in Fig. 4.11. There are many alternatives to this, including many designs intended for home-build, which are available on the Web. The PICSTART programmer is connected to the host computer by a serial cable and MPLAB has the software to communicate with it. The PICSTART programmer can accept a wide range of dual-in-line microcontroller packages, from 18 to 40 pins. With adaptors it can program other package types.

The following steps take you through actually downloading code to the microcontroller, using the PICSTART Plus programmer. If you have a programmer and the ping-pong hardware, you can immediately download the program you have just created in the preceding tutorial.

You will need to power your PICSTART programmer and connect it to the serial port of your computer. From within MPLAB IDE, select **Programmer** > **Select Programmer** > **PICSTART Plus**. Then enable the programmer with **Programmer** > **Enable Programmer**.

Figure 4.11: The PICSTART Plus Programmer

A positive response should be given via the Output window. If there is a problem, you may need to check **Programmer > Settings > Communications**. Ensure the programmer toolbar (Fig. 4.12) is displayed. If not, find it with **View > Toolbars > Picstart**.

Put the Zero Insertion Force (ZIF) socket on the PICSTART programmer in the Open position. Place a 16F84A into it, ensuring from the legend that the chip is in the right place and is the right way round.

Then close the ZIF with the lever. With the project you want open on MPLAB, you should now be able to apply the features available to you, as summarized in Fig. 4.12.

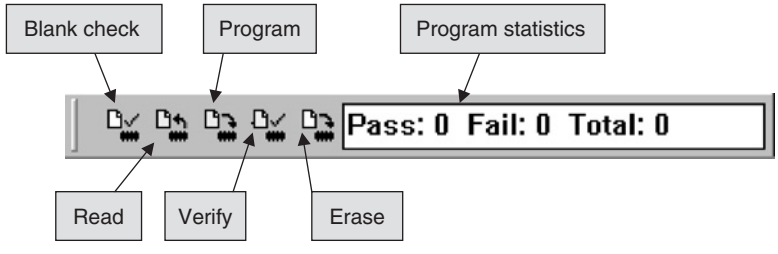


Figure 4.12: The MPLAB Programmer Toolbar

4.9 What Others Do—A Brief Comparison of CISC and RISC Instruction Sets

As they are CISC CPUs, we expect the Atmel 8051 microcontrollers to have a much larger instruction set, compared to the RISC 16 Series family. Furthermore, we might expect some of those instructions to be more powerful. The price to be paid for these is slower execution time. This expectation is proved correct. The 8051 core has 111 instructions. Most instructions execute in one machine cycle, each of which takes 12 oscillator cycles (compared with the four oscillator cycles for the PIC instruction cycle). Some instructions take two cycles. The two "advanced" instructions **MUL** (multiply) and **DIV** (divide) take four cycles each.

The versatility offered by a CISC instruction set appears attractive when viewed from a RISC world. Simple actions that require several RISC instructions are reduced to a single instruction. Two simple examples are shown in Program Example 4.4. In the first, a byte of constant data (called literal data in the PIC world as we know, and *immediate data* in the 8051 world) is moved to a memory location called **mem_loc**. This requires two PIC instructions or one 8051 instruction. The programmer gains the advantage with the 8051 of writing less lines of code, but we can see that there is no apparent timing advantage in execution, as each ultimately takes two cycles. Indeed, as the 8051 machine cycle is longer, the timing advantage ultimately lies with the PIC microcontroller.

Interestingly, the advantage is less clear in the next example. The PIC 16 Series only has conditional skip instructions and not branch. Thus, conditional branches have to be built from a skip instruction followed by a **goto**. Here, three cycles are taken if the conditional branch is executed for the PIC microcontroller or two for the 8051, while a single line of code is required for the latter. Now the timing advantage lies with the 8051 CPU, as long as instruction cycles are equal in length.

PIC 16 Series

```
movlw  22        ;1 cycle
mov mem_loc      ;1 cycle
```

Atmel 8051

```
mov mem_loc,#22  ;2 cycles
```

(a)

```
btfsc status,0   ;1 cycle (no skip)
goto new_place   ;2 cycles
```

```
jc new_place     ;2 cycles
```

(b)

Program Example 4.4: Comparing RISC and CISC Instruction Capabilities: (a) Moving Immediate/Literal Data to a Memory Location and (b) Branching if Carry Bit Set

The disadvantage of the extra lines of code required for the RISC processor is not a great one. We will see in the next two chapters that even this slight disadvantage can be ameliorated, in assembler by using macros and also by using a high-level language like C, where the programmer is no longer directly concerned by the number of lines of assembler code produced.

4.10 Taking Things Further—The 16 Series Instruction Set Format

It is interesting to take a little time here to understand further the way the instruction code is made up of different component parts. The PIC 16 Series has four possible instruction word formats, as shown in Fig. 4.13. The instruction word, which is transferred down the program bus (Fig. 2.2), is made up of 14 bits. These appear as bits 0 to 13 in the figure. The opcode, the actual instruction part of the instruction word, always occupies the highest bits of the instruction word.

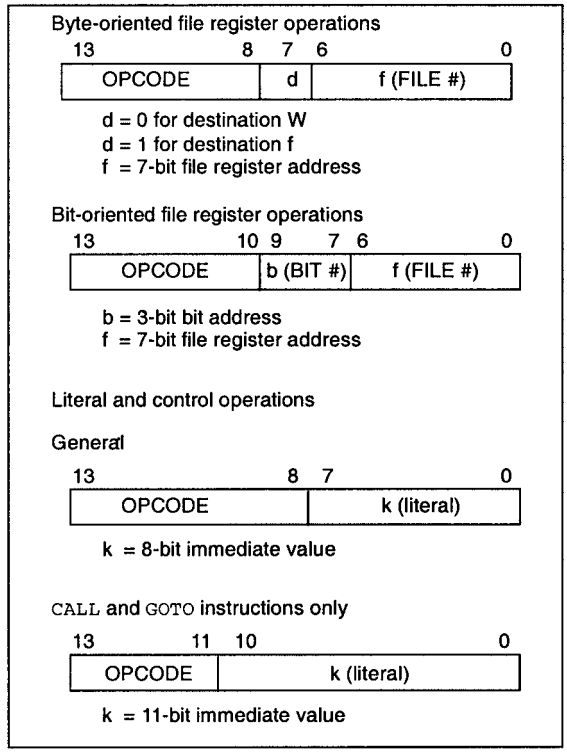


Figure 4.13: Instruction Formats of the PIC Mid-Range Microcontrollers

This is the part of the instruction word that ends up in the "Instruction Decode and Control" unit of Fig. 2.2, but it is not always the same length.

If the instruction is the type that contains a file address, then it is of the first format shown. The most significant 6 bits hold the opcode, while the least significant 7 bits are used to hold the address. These bits are transferred onto the Direct Addr bus of Fig. 2.2. In fact, because the F84A only has a small memory, only the least significant 5 bits are used, as can be seen from the Direct Addr bus size indicated. Bit 7 holds the **d** bit. Different instruction word patterns are used for the other instruction categories. These can be seen and understood by reading the information in the figure.

4.11 Summary

If you have followed the material of this chapter then you have taken an enormous step forward—you are on your way to becoming a programmer of embedded systems! The key points are:

- Assembler is a programming language that is part of the toolset used in embedded systems programming. It comes with its own distinct set of rules and techniques.
- It is essential to adopt and learn an IDE when developing programs. The MPLAB IDE is an excellent tool for PIC microcontrollers, both for learners and professionals. And it can't be beaten on price!
- While some people are eager to get programs into the hardware immediately, it is extremely useful to learn the features of a simulator. The simulator in MPLAB allows the user to test program features with great speed, and is an invaluable learning tool.

References

[4.1] MPASM User's Guide, with MPLINK and MPLIB (1999). Microchip Technology Inc., Document No. DS33014G.

[4.2] MPLAB User's Guide (2005). Microchip Technology Inc., Document No. DS51519A.

CHAPTER 5

Building Assembler Programs

In Chapter 4 the basic rules of assembler programming were introduced, along with some of the instructions from the PIC® 16 Series instruction set. It's as if we have now learned some introductory skills in bricklaying. We need to develop those skills further, but we need to begin to think about the structures that the bricks are going to be built into. Therefore, we now need to develop this introductory knowledge so that we can actually build up programs that have structure, and are functional and reliable.

In this chapter you will learn about:

- How to visualize a program and represent it diagrammatically
- How to use subroutines
- How to implement delays
- How to use look-up tables
- Logical and arithmetic instructions
- How to simplify and optimize assembler programming
- More advanced features of software simulators.

5.1 The Main Idea—Building Structured Programs

When we actually design a program that is to do anything more than some minimalist task, it is important to think about and plan its structure *before* starting to write the code. This is especially true in assembler—Chapter 4 warned that one of the problems of assembler programming was that it leads to unstructured "spaghetti" programs. Therefore, we must consider means of representing the program diagrammatically. Let us consider how we might do this, with two examples of commonplace domestic products.

5.1.1 Flow Diagrams

A well-established diagramming technique is the flow diagram. While this has many symbols that can be used, we can develop good flow diagrams with just two—rectangle for process or action and diamond for decision.

Figure 5.1 shows a simple flow diagram example, for a refrigerator controller. The user has a single control, an adjustable potentiometer that allows him/her to set a desired temperature. Within the fridge there is a temperature sensor. Temperature is controlled by switching the compressor on or off—the temperature will fall when it is running. The program reads both the actual and demand temperatures, and determines which is higher. If it is the actual temperature, then the compressor is switched on. If the difference between the two is very great, then an alarm will sound. The flow diagram shows this action, using just the two symbols mentioned above. Notice how each diamond decision symbol contains a question within it, with a yes/no answer. Its two exit points then correspond to the two possible answers. It can be seen that this example program will loop indefinitely. This is a common embedded system program structure and is sometimes called a *super loop*.

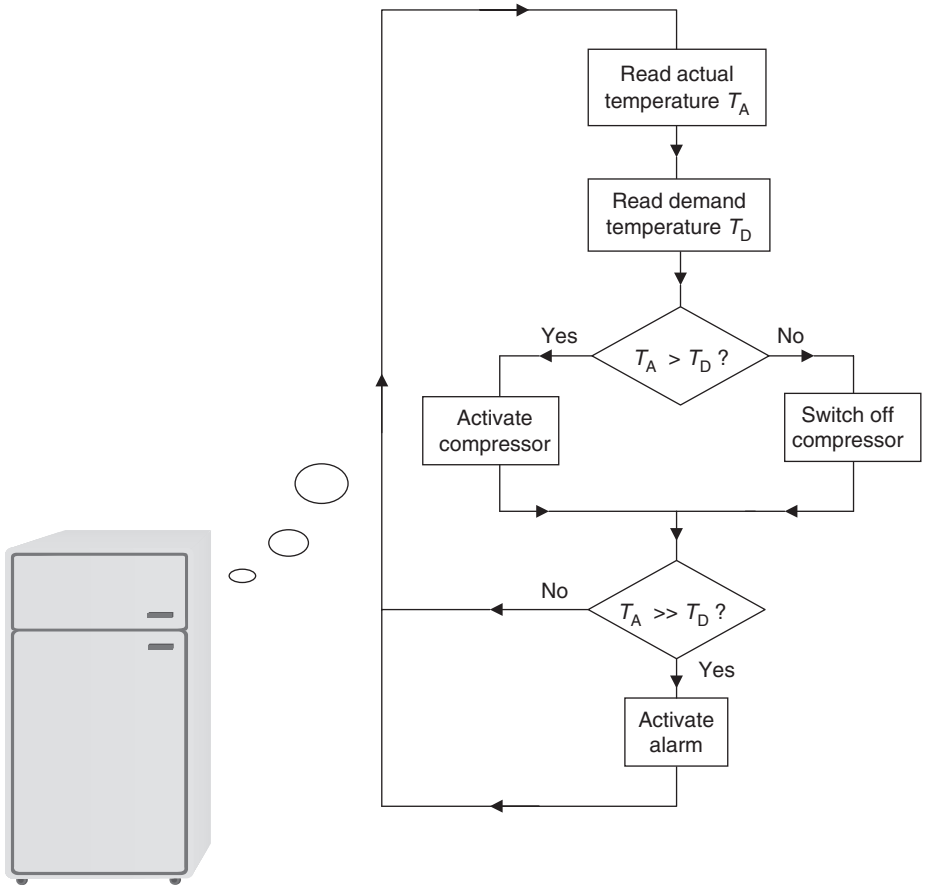


Figure 5.1: Flow Diagram of Simple Refrigerator Controller

It is possible to draw flow diagrams with too much detail or too little. With care and experience, however, it is possible to draw them at a level such that the diagram can be converted to an assembler program without too much difficulty. In some cases, an overview flow diagram might be appropriate, while different sections within that are then developed as separate diagrams.

Flow diagramming is considered by some to be an old-fashioned technique as, somewhat like assembler itself, it does not encourage a structured program. Nevertheless, it is easy to learn and use, and it can clearly represent simple program ideas. Therefore, for our purposes we will make use of it.

5.1.2 State Diagrams

The flow diagram views life as a series of actions or events which are rapidly passed through. Many products, however, behave in a different sort of way. They tend to move from one state to another, maybe spending a significant period of time in that state and leaving it only when a time period is completed or a specific event occurs. These are best represented by a *state diagram*, which forms an alternative to the flow diagram. As with flow diagrams, there is some sophistication in using state diagrams in their full form. For our purposes, however, all we need to do is to draw each state as a labeled circle and interconnect these with arrows. These show under what condition(s) one state can move to another. Each arrow is labeled with the condition that causes the state to change.

Figure 5.2 shows the function of a domestic washing machine represented as a simple state diagram. When switched on, the machine first enters a Ready state. If the door is closed and the user initiates a wash, then the machine first loads with water. A level sensor detects when this is complete. However, the machine will also measure the time taken to fill. If it does not fill within the allotted time, then a fault is assumed.

This may be due to inadequate water pressure or a faulty valve mechanism. The use of time-out is a low-cost alternative to sensing the water flow or pressure itself. The fill state is followed by a water heating state. Again, a time-out occurs if the water is not heated in an allotted time. The process continues as shown, each state having a "successful" exit condition, as well as one which leads to the fault state.

From a programming point of view, state diagrams are more abstract than flow diagrams and cannot so easily be translated directly into assembler code. In fact, it is often useful to convert each state into its own flow diagram. To retain clarity of structure and ensure good programming practice, each state should have very clearly defined entry and exit points. The use of both flow diagrams and a state diagram to represent program structure is illustrated later in this chapter, with the electronic ping-pong program.

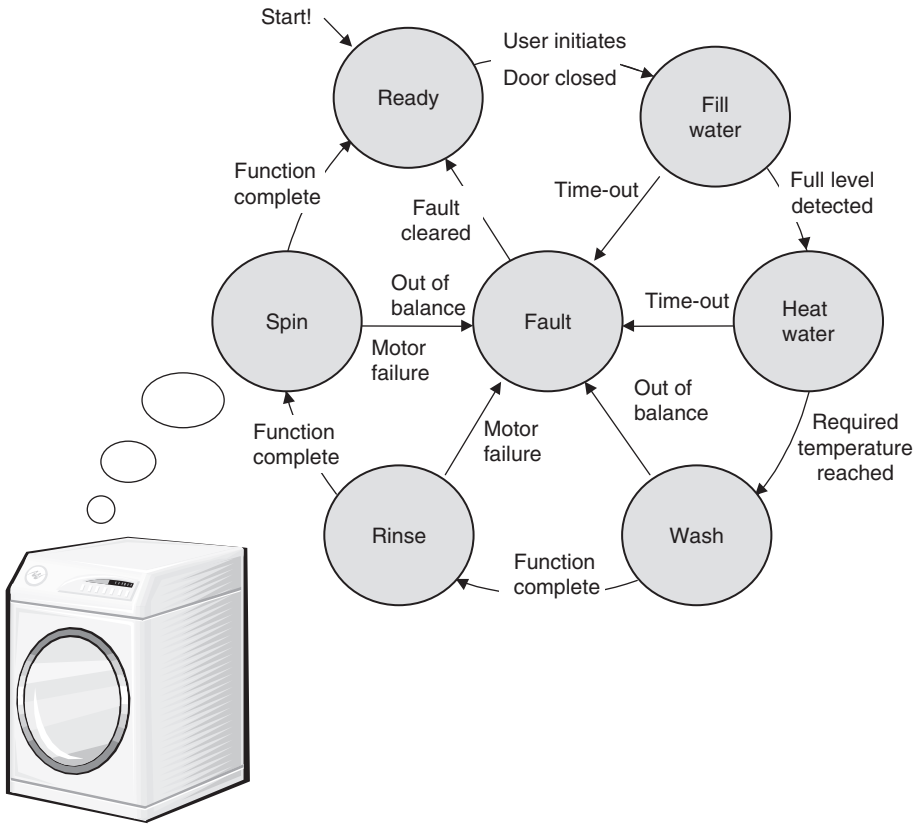


Figure 5.2: A Washing Machine Control Program, Visualized as a State Diagram

5.2 Flow Control—Branching and Subroutines

As Figs 5.1 and 5.2 show, programs rarely execute in one continuous and unbroken sequence of instructions. The techniques used to allow program execution to move to different program sections are collectively termed *flow control*, and are the subject of this section. We have already met the PIC instruction **goto**, which unconditionally transfers program execution from one place in the program to another. Now we explore the use of conditional branching and subroutines.

5.2.1 Conditional Branching and Working with Bits

One of the most important features of any microprocessor or microcontroller program is its ability to make "decisions," i.e., to act differently according to the state of logical variables. Many microprocessors have within their instruction sets a number of instructions that allow them to test a particular bit, and either continue program execution if the condition is not met

or branch to another part of the program if it is. This is illustrated in Fig. 5.3. Most commonly these variables are bit values in condition code or Status registers.

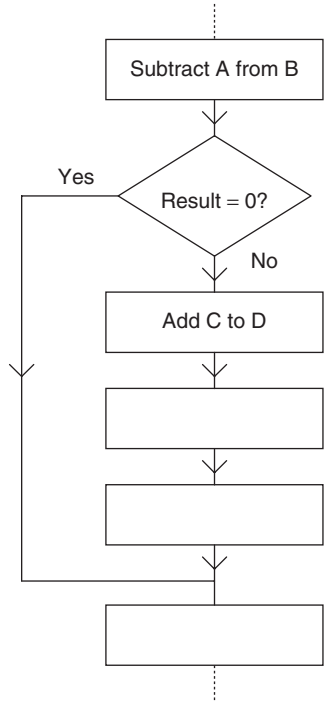


Figure 5.3: Conditional Branching

The PIC 16 Series microcontrollers are a little unusual when it comes to conditional branching, as they do not have branch instructions as such. They have instead four conditional "skip" instructions. These test for a certain condition, and skip just one instruction if the condition is met and continue normal program execution if it is not. The most versatile and general purpose of these are the instructions:

btfsc f,b
btfss f,b

The first of these tests bit **b** in memory location **f** and skips just one instruction if the bit is set (i.e., at Logic 1). The second does a similar thing, but skips if the tested bit is clear (i.e., at Logic 0). Let us explore this in an example program.

Our first example, the simple data moving program of Program Example 4.2, works well. Suppose, however, we don't want to move the whole of Port B to Port A. Maybe we wanted to transfer just one bit or move a bit from one position in Port B to a different position in Port A.

Then we could use the "bit-oriented" instructions of the 16 Series instruction set, of which there are four: the two we have just seen above, and **bsf** and **bcf**.

The program fragment in Program Example 5.1 performs a near identical function to that of Program Example 4.2, but applies the bit manipulation instructions just identified. As just single bits are manipulated, however, it does not affect any of the other bits in the port to which it is writing. It lights an LED if the associated microswitch is pressed. Even this simple task requires some thought, however. As the port input goes low when the button is pressed, the program needs to *set* the output bit (to light the LED) if the input is low and *clear* it if it is high. This implies a selection process—in a high-level language we might call this an *if. . .else* structure. The simple skip instruction is not able to do this on its own. One way to do this is to *preset* the output bit with one value and then change it if we find it has been set wrong.

```
;The "main" program starts here
       movlw 00            ;clear all bits in port A and B
       movwf porta
       movwf portb
loop   bcf   portb, 3      ;preclear port B, bit 3
       btfss porta, 3
       bsf   portb, 3      ;but set it if button pressed
;
       bcf   portb, 4      ;preclear port B, bit 4
       btfss porta, 4
       bsf   portb, 4      ;but set it if button pressed
       goto loop
       end
```

Program Example 5.1: Testing and Manipulating Single Bits

Programming Exercise 5.1

Open a new project under the suggested name Bit_Set, or choose your own name. Copy Program Example 4.2 into it as source file, but replacing the main section of code with Program Example 5.1. Build the project and simulate. With the Stimulus Controller create inputs signals for Port A, pins 3 and 4, selecting Toggle for Action. Open a Watch window with **PCL, PORTA, PORTB** and **W register** as observed variables. Step through the program, "firing" the inputs at appropriate moments, noting the effect. Change the program so that:

(1) Port B, bits 3 and 4 are *set* if the respective buttons are pressed
(2) Different bits in Port B are set when the buttons are pressed.

5.2.2 Subroutines and the Stack

As we develop bigger programs, we quickly find that there are program sections that are so useful that we would like to use them in different places. Yet it is tedious, and space and memory consuming, to write out the program section whenever it is needed. Enter the subroutine.

The subroutine is a program section structured in such a way that it can be called from anywhere in the program. Once it has been executed the program continues to execute from wherever it was before. The idea is illustrated in Fig. 5.4. At some point in the main program there is an instruction Call SR1. Program execution then switches to Subroutine 1, identified by its label. The subroutine must be terminated with a "return from subroutine" instruction. Program execution then continues from the instruction *after* the Call instruction. A little later in the program another subroutine is called, followed a little later again by another call to the first routine.

The action of the Call instruction is two-fold. It saves the contents of the Program Counter onto the Stack, so that the CPU will know where to come back to after it has finished the subroutine. It then loads the subroutine start address into the Program Counter. Program execution thus continues at the subroutine. The return instruction complements the action of the Call. It loads the Program Counter with the data held at the top of the Stack, which will be the address of the instruction following the Call instruction. Program execution then continues at this address. Subroutine Call and Return instructions *must always* work in pairs.

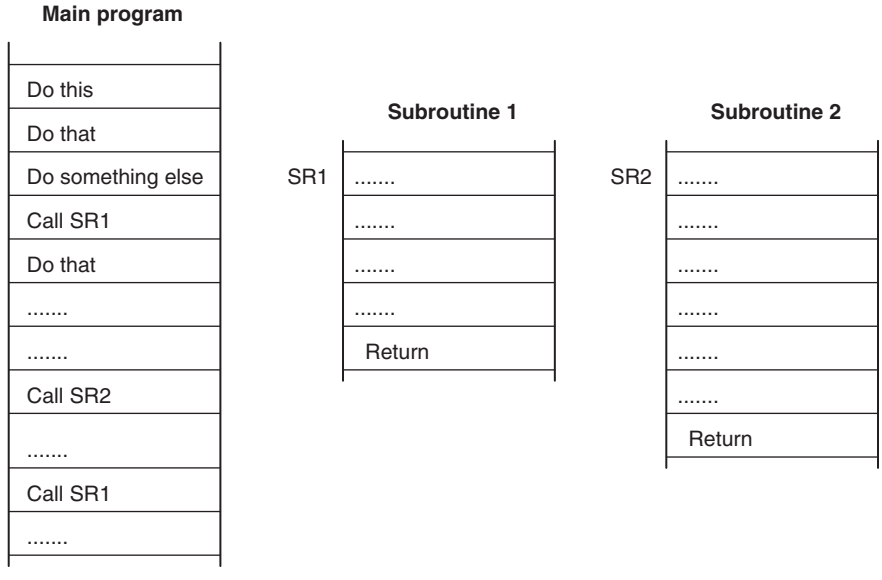


Figure 5.4: Subroutine Calling

The PIC 16 Series subroutine call and return instructions can be seen in Appendix A, and are simply called **call** and **return**. A special return instruction, **retlw**, is also available. Example subroutines will be explored in the following section.

A subroutine called from *within* another subroutine is called a nested subroutine. In doing this, it must be remembered that every time a subroutine is called one Stack location is taken up, which becomes free again on the subroutine return. If we call a subroutine from within another, then two Stack locations are used up, or three if there is another nested call. As the 16 Series microcontrollers only have an eight-level stack, care must taken that there is not "stack overflow."

5.3 Generating Time Delays and Intervals

A recurring theme of embedded systems is how we deal with time—how they respond in a timely way to external events, and how they can measure time and generate time delays. Even with only a limited grasp of programming, we can begin to address the issue of timing by developing program loops that give time delays of known and accurate duration.

The initial concept is simple. A memory location is set up to act as a counter, loaded with a certain value and then decremented repeatedly in a loop until it reaches zero. The time taken will depend on the number first placed in the counter and then the time taken for each decrement loop.

To implement accurate delays, the oscillator frequency needs to be accurate and stable, and we need to know what that frequency is. Here lies one advantage of using a crystal oscillator, as that gives a frequency of excellent accuracy and stability. Approximate delays can of course be implemented with other oscillator sources, as we do with the ping-pong. With the PIC, we need to remind ourselves that each instruction cycle takes four oscillator cycles.

A simple example of a delay loop, taken from the ping-pong program, is shown in Program Example 5.2. It takes the form of a subroutine called **delay5**. The subroutine opens by moving a number into the memory location **delcntr1**. In this case the number is 200_D, although this can be varied to produce delays of different lengths (up to a maximum, for an 8-bit location, of 255_D). The actual delay loop is that section of code starting with the label **del1**. Two **nop** instructions, which do nothing at all but take up time, are used to extend the time taken for one loop iteration. The **decfsz** instruction is then implemented. This decrements memory location **delcntr1**, which has been previously loaded. If the result of the decrement is zero, then the subsequent instruction is skipped and program execution moves on to the **return** instruction.

For 199 cycles, however, the decrement result will not be zero, there will be no skip and program execution will go back to **del1**.

```
;Delay of 5ms approx. Instruction cycle time is 5us.
delay5  movlw      D'200'      ;200 cycles called,each taking 5x5=25us
        movwf      delcntr1
del1    nop                    ;1 inst. cycle
        nop                    ;1 inst. cycle
        decfsz     delcntr1,1  ;1 inst. cycle, when no skip
        goto del1              ;2 inst. cycles
        return
```

Program Example 5.2: A Delay Subroutine

The time duration of this delay subroutine can be worked out with ease, by considering the time taken by each instruction in the loop (taken from the instruction set, Appendix 1). This is shown in the program comments. While **delcntr1** is counting down from its initial value, the loop is made up of two **nop** instructions, a **decfsz** and a **goto**. As the **decfsz** instruction does not skip, it takes only one cycle, whereas the **goto** always takes two. Therefore, the total per loop is five. The electronic ping-pong has a clock frequency of approximately 800 kHz, and therefore an instruction cycle frequency of 200 kHz or instruction cycle period of 5 μs. Therefore each loop, with its five instruction cycles, takes 25 μs. Two hundred loops are called; hence the overall duration is 5 ms.

For a precise delay, it is necessary also to take into account the time duration of the final cycle, and the entry and exit to the subroutine. On the final loop iteration, for example, the **decfsz** causes a skip and hence takes two cycles. The **goto** is, however, missed.

The simple delay loop of Program Example 5.1 is useful for comparatively short delays, say up to tens of milliseconds. There are many situations, however, when we want something much longer. A simple way to extend it is to create a second subroutine, similar to the first, but which calls the first from within its loop. An example is shown in Program Example 5.3. This loop makes 100_D calls to the subroutine **delay5**, which we have just seen. The resultant delay is therefore around 500 ms. A further way of writing a longer delay loop, using a "loop within a loop" within a single subroutine, appears in Program Example 5.4.

```
;500ms delay (approx)        ;100 calls to delay5
delay500  movlw      D'100'
          movwf      delcntr2
del2      call       delay5
          decfsz     delcntr2,1
          goto       del2
          return
```

Program Example 5.3: Nested Subroutines for Greater Delay

Delay routines are very useful things and are widely used. However, they need to be used with care, as when the delay routine is running, the CPU can do nothing else. A delay routine is a bit like asking the scientist Einstein to sit and count beans—it's just not very good use of a powerful resource.

5.4 Dealing with Data

We have seen that it is easy to move single bytes of data to or from data memory, whether the memory is SFRs or memory locations dedicated to holding particular variables. We did this in the most recent program example, with the instructions:

```
movlw  D'100'
movwf  delcntr2
```

In using these instructions, we specify the actual address of the memory location required. In this example, **delcntr2** is a label clearly specifying an address. If, however, we want to work with blocks of data, simple data moves like this become restrictive. If working through a list of numbers, for example, it is quite inconvenient to have to specify a fixed address for every memory location. Instead, it is useful to have a pointer that shows where we are in the list, but which can be changed to point to successive locations. It is useful to be able to do this in data memory. In this case the program can develop a block of data and then manipulate it. It is also useful to do it in program memory, in which case the program is able to access a predetermined block of data, but not modify it. This section considers two techniques used for working with blocks of data, in data memory and program memory.

5.4.1 Indirect Addressing and the File Select Register

In Figs 2.2 and 2.5 we see a register called the FSR, the File Select Register. As Fig. 2.2 suggests, instead of embedding an address in the instruction word, the number stored in the FSR can be used as an address to data memory. This is called an *indirect address.* The FSR is invoked whenever the memory location **INDF** is addressed. **INDF** doesn't actually exist as a register; its use simply forces the CPU to implement the indirect addressing mode. The beauty of this mode is that the FSR can be manipulated as a normal memory location, to point to anywhere in data memory that is wanted. When **INDF** is invoked, the instruction used then acts on the memory location pointed to by the FSR.

An example of the use of indirect addressing appears in Program Example 5.7, where we have developed a program that develops a data list. It is discussed in Section 5.6.4.

5.4.2 Look-Up Tables

The instruction **movlw** allows us to embed within the program a byte of constant data that is then applied in any way we want. We have already seen this in the previous program examples. This is fine for manipulating single bytes, or just a few. But suppose we want

to place in the program a whole list of numbers that are needed during program execution, maybe to generate a waveform or to produce output patterns on a display. Suppose also that we wanted to be able to remember where we were in the list with some sort of marker. The **movlw** instruction is then not really up to the job, and we need to apply a way of setting up data known as a *look-up table*.

A look-up table is a block of data that is held in program memory, which can be accessed by the program and used within it. In a Von Neumann structure, with its single address and data buses, it is rather easy to set up and use look-up tables, as all memory locations are of equal size and all can be accessed with equal ease. In a Harvard structure it is more difficult, as data must be moved from one distinct memory map to another. The situation is made worse by the difference in memory location size that usually exists between data and program memories. Therefore, in a Harvard structure like the PIC's, a special technique is used to create look-up tables. This introduces several important new ideas.

The PIC 16 Series approach to look-up tables is shown in Fig. 5.5. The table is formed as a subroutine. Every byte of data in the table is accompanied by a special instruction, **retlw**. This instruction is another "return from subroutine," but with a difference—it requires an 8-bit literal operand. As it implements the subroutine return, it picks up its operand and puts it into the W register. The table is essentially a list of **retlw** instructions, each with its byte of data.

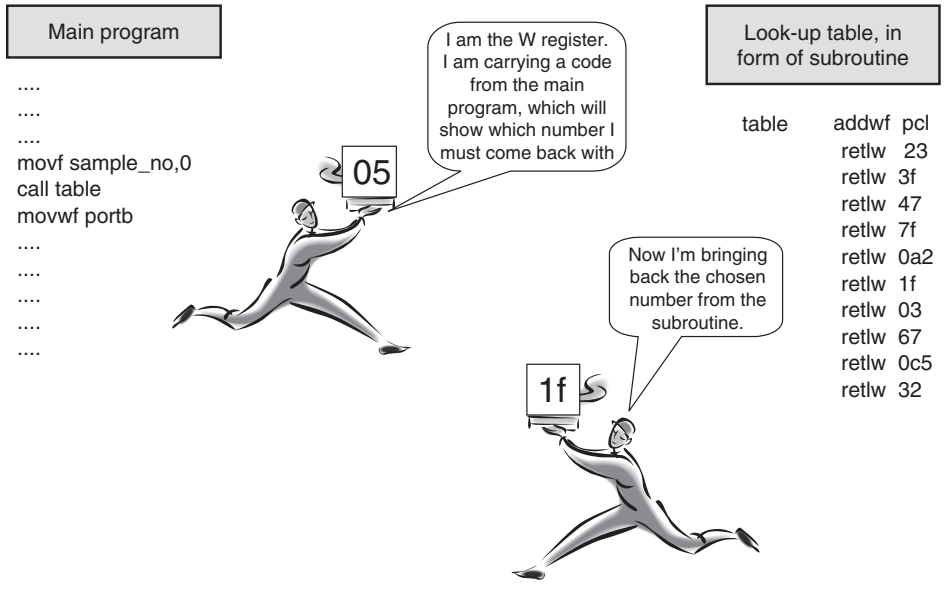


Figure 5.5: Fetching Data from a Look-Up Table

What we need now is a technique that allows just one of those **retlw** instructions to be selected from the list—we use something called *computed go to*. Look carefully at the very

first instruction in the subroutine, **addwf pcl**. The contents of the W register are added to **pcl**, which is the lower byte of the Program Counter. This sounds like a pretty dangerous sort of thing to do, and for sure it must be done with care. The effect, however, is that once a number has been added to the Program Counter, then program execution jumps forward by whatever that number was. In this example the CPU then executes the **retlw** instruction it lands on and thus goes back to the main program. Note carefully that, however long it appears, on any one iteration *only two instructions in the subroutine are executed*, the **addwf pcl** and the chosen **retlw**. It is obviously up to the programmer to ensure that as the subroutine is called, the W register is already loaded with the offset that is needed.

Let's see this at work in the example of Fig. 5.5. Using the **movf** instruction, the main program transfers into the W register the contents of a memory location called **sample_no**. It then calls the subroutine **table**. In this example it is assumed that **sample_no** was holding the number 5, which the W register is holding as the subroutine is entered. As the subroutine starts program execution, the number 5 is added to **pcl**. Program execution therefore jumps forward by 5, to instruction **retlw 1f**. This causes a return from the subroutine, with the number **1f** now placed in the W register. The main program immediately makes use of this number, in this case transferring it to Port B.

In summary, the W register is like a messenger being sent to the subroutine. It goes to the subroutine carrying the code (which acts as a pointer) showing which number is wanted and it comes back carrying that number.

There is one possible problem with this approach—by manipulating only the lower byte of the Program Counter we can only operate within the first 256 words of program memory, or within any page following. If the look-up table is very long, or if it is situated across a page boundary, then problems with the computed go to will occur. In this case it is essential to calculate a fuller version of the computed go to [Ref. 5.1].

5.4.3 Example Program with Delays and Look-Up Table

Program Example 5.4 is a simple example combining delay loops and use of a look-up table, applied to the ping-pong hardware. It takes 8-bit values from a look-up table and transfers them to the ping-pong LEDs, with a delay between each data transfer. The overall effect is a display of randomly flashing LEDs. The opening sections are very similar to Program Example 4.2, although now we need to specify the address for **pcl** from Fig. 2.5. Also specified is an RAM location, called **pointer**, whose address is chosen to lie within the available range of 0C to 4F (also Fig. 2.5).

The core of the program starts at the label **loop**. Just as in Fig. 5.5, the value of **pointer** is moved to the W register and a subroutine called **table** is called. Upon return from the subroutine, the value held in the W register is transferred to Port B. This lights a certain pattern of LEDs. A delay loop, **delay**, is then called. Note the structure of this, which is similar to the concept implied in Program Example 5.3, but uses a single subroutine and a

loop within a loop to extend the delay. The program continues to loop, checking whether **pointer** has reached its maximum value. If so, it is reset to zero before continuing.

```
;*************************************************************
;FLASHING LEDs!
;This program continuously outputs a series of led patterns,
;using simulation or ping-pong hardware.
;TJW 5.3.05.              Tested in simulation 11.3.05.
;*************************************************************
;Clock is 800kHz
;Configuration Word: WDT off, power-up timer on,
;                       code protect off, RC oscillator
;
        list p=16F84A
;
;specify SFRs
pcl       equ  02
status    equ  03
porta     equ  05
trisa     equ  05
portb     equ  06
trisb     equ  06
;
pointer   equ  10
delcntr1  equ  11
delcntr2  equ  12
;
      org     00
;Initialize
start bsf     status,5      ;select memory bank 1
      movlw   B'00011000'
      movwf   trisa         ;port A according to above pattern
      movlw   00
      movwf   trisb         ;all port B bits output
      bcf     status,5      ;select bank 0
;
;The "main" program starts here
      movlw   00            ;clear all bits in port A
      movwf   porta
      movwf   pointer       ;also clear pointer
loop  movf    pointer,0     ;move pointer to W register
      call    table
      movwf   portb         ;move W register, updated from table SR, to
                            ;port B
      call    delay
```

Program Example 5.4: Using a Look-Up Table

```
      incf  pointer,1
      btfsc pointer,3    ;test if pointer has incremented to 8
      clrf  pointer      ;if it has, clear pointer to start over
      goto  loop
;
;*************************************************************
;Subroutines
;*************************************************************
;Introduces delay of 500ms approx, for 800kHz clock
delay movlw  D'100'
      movwf  delcntr2
outer movlw  D'200'
      movwf  delcntr1
inner nop
      nop
      decfsz delcntr1,1
      goto   inner
      decfsz delcntr2,1
      goto   outer
      return
;Holds Lookup Table
table addwf pcl
      retlw 23
      retlw 3f
      retlw 47
      retlw 7f
      retlw 0a2
      retlw 1f
      retlw 03
      retlw 67
;
      end
```

Program Example 5.4: Continued

Programming Exercise 5.2

People often find the concept of the PIC look-up table quite difficult to understand. It is therefore a particularly good idea to simulate an example. Create a project called Flashing LEDs, copy the source code of Program Example 5.4 from the book CD and include it in the project. Then simulate. Open a Watch window with **PCL, PORTB, WREG** and **pointer** as observed variables. Step through the program and see carefully how the value of the W register changes as the subroutine is entered and left. Use Step Over to avoid getting stuck in the delay subroutine. Ensure that you understand all stages of the program and the new instructions that have been used.

5.5 Introducing Logical Instructions

So far we have seen a good selection of the 16 Series instructions, but have yet to see any logical ones. These instructions, like **andwf**, **andlw**, **iorwf** or **xorwf**, perform logical operations between the contents of the W register and either a literal value or a value held in a memory location. They do it on a *bitwise* basis. For example, if the **andlw k** instruction was applied, then bit 0 of the literal value is ANDed with bit 0 of the W register, bit 1 is ANDed with bit 1 and so on. These instructions are useful for actual logical operations. Commonly, **and** instructions are used for suppressing unwanted bits in a word and **or** instructions are used for setting individual bits in a word.

As an example, let's look at an alternative way of resetting the pointer in Program Example 5.4. Remember, every time the pointer increments to value 8 (0000 1000), it needs to be reset to 0. Instead of doing this, why not just suppress the higher 5 bits of the word, which are of no use in this application?

The alternative, using logical instruction **andlw**, is shown in Program Example 5.5. Instead of testing the value of **pointer** every time it is incremented, it is now 'ANDed' with the number 07, or 0000 0111_B. Now when it increments to 0000 1000_B, bit 3 (which has been set to 1) is ANDed to 0 and the value of **pointer** returns to zero.

```
loop   movf    pointer,0      ;move pointer to W register
call table
movwf portb           ;move W register,updated from table SR, to port B
call delay
incf pointer,0        ;increment pointer, place result in W reg
andlw 07
movwf pointer
goto  loop
```

Program Example 5.5: Introducing a Logical Instruction

5.6 Introducing Arithmetic Instructions and the Carry Flag

The 16 Series instruction set has six arithmetic instructions: **addwf, addlw, subwf, sublw, incf** and **decf**. Their use is central to any arithmetic processing that the microcontroller may have to do, as we will see shortly in the next program example.

5.6.1 Using Add Instructions

The use of the two add instructions is straightforward. Using **addwf** the contents of the W register are added to the contents of the memory location specified; using **addlw** a literal value is added to the contents of the W register. The situation becomes slightly more complex when one realizes that two 8-bit results added together can lead to a 9-bit result. This ninth bit is the Carry flag in the Status register.

5.6.2 Using Subtract Instructions

The subtract instructions follow a similar pattern to the add. The Carry bit now acts as a Borrow, except the polarity is reversed (see the Status register, Fig. 2.3). Therefore, if a subtract occurs and the result is positive, then the Carry bit is *set*. If the result is negative, then the Carry bit is *clear*.

5.6.3 An Arithmetic Program Example

Program Example 5.6 demonstrates the use of some simple arithmetic techniques. It generates a Fibonacci series by an adding process, as described in the header. A counter indicates how many numbers in the series have been calculated. When it exceeds the 8-bit range, it reverses the series by subtracting. It detects the range overflow by checking the Carry bit after each addition.

The program starts by preloading the three first numbers in the series into the memory store. It then starts moving up the series, from the label **forward**. The two most recent numbers are added and the Carry bit then checked. If it is set, the 8-bit range has been exceeded and the program will need to reverse. Assuming Carry was not set, the program then increments the counter and shuffles the numbers in the memory store, discarding the oldest. The program then loops up to **forward**. If, however, the Carry had been set earlier, the program branches to **reverse**. Now it works down the series, by subtraction. It tests the counter number to determine when it should return to **forward**.

```
;*****************************************************************
;In a Fibonacci series each number is the sum of the two previous
;ones, e.g. 0,1,1,2,3,5,8,13,21.....
;This program calculates Fibonacci numbers within an 8-bit range,
;first going up and then down.
;Program intended for simulation only, hence no input/output.
;The program demonstrates addition, subtraction, compare.
;TJW 17.3.05.                          Tested by simulation 18.3.05
;*****************************************************************

    list p=16F84A
;no i/o  ports used
status   equ 03
c        equ 0
z        equ 2
;these memory locations hold the three highest values of the
;Fibonacci series
fib0     equ 10       ;lowest number (oldest when going up,
                             ;newest when reversing down)
fib1     equ 11       ;middle number
fib2     equ 12       ;highest number
```

Program Example 5.6: Generating a Fibonacci Series

```
fibtemp   equ 13       ;temporary location for newest number
counter  equ 14       ;indicates value reached, opening value is 3
org 00
;preload initial values
       movlw 0
       movwf fib0
       movlw 1
       movwf fib1
       movwf fib2
       movlw 3
       movwf counter ;we have preloaded the first three numbers,
                     ;so start count at 3
;
forward movf fib1,0
      addwf  fib2,0
      btfsc  status,c    ;test if we have overflowed 8-bit range
      goto   reverse     ;here if we have overflowed, hence reverse down
      movwf  fibtemp     ;latest number now placed in fibtemp
      incf   counter,1
;now shuffle numbers held, discarding the oldest
      movf   fib1,0      ;first move middle number, to overwrite oldest
      movwf  fib0
      movf   fib2,0
      movwf  fib1
      movf   fibtemp,0
      movwf  fib2
      goto   forward
;when reversing down,    subtract fib0 from fib1 to form new fib0
reverse movf fib0,0
      subwf  fib1,0
      movwf  fibtemp     ;latest number now placed in fibtemp
      decf   counter,1
;now shuffle numbers held, discarding the oldest
      movf   fib1,0      ;first move middle number, to overwrite oldest
      movwf  fib2
      movf   fib0,0
      movwf  fib1
      movf   fibtemp,0
      movwf  fib0
;test if counter has reached 3, in which case return to forward
      movf   counter,0
      sublw  3
      btfsc  status,z
      goto   forward
      goto   reverse
;
      end
```

Program Example 5.6: Continued

Programming Exercise 5.3

Create a project in MPLAB® called Fibonacci. Copy from the book CD the source file of Program Example 5.6 into it and simulate. In the Watch window display **counter, fib0, fib1, fib2, fibtemp, WREG** and **STATUS**. Single-step initially and watch the Fibonacci series develop, in **fib0, fib1** and **fib2**. How many numbers in the series fit into the 8-bit range? Watch the Carry bit being set as the range is exceeded and see the program reverse down the series. Notice now that the Carry bit (now acting as Borrow) is *set* after each subtraction. Try halting the program at **reverse**, and forcing two values for **fib0** and **fib1** that will give a negative result. Single-step through the subtraction, check the result in the W register and notice that the Carry bit is clear. See how the comparison of **counter** with the literal number 3 is achieved, and see the program return to **forward**.

5.6.4 *Using Indirect Addressing to Save the Fibonacci Series*

Program Example 5.7 extends the Fibonacci series program to incorporate storage of the series, using indirect addressing, with the main section of the program shown. The extra lines of code are highlighted in bold. The data block is stored in data memory, starting at location 20_H. The first three numbers in the series are entered using conventional addressing (indirect addressing could already be used here, but this would require slightly more lines of code). The FSR is then loaded with 23_H, the address of the next location in the data block to be loaded. Entering the main loop, it can be seen that every time a new number is generated, it is stored in the data block, using the **movwf indf** instruction. The value of the FSR is then incremented until it reaches a predetermined maximum value.

```
;...
(opening lines of program omitted)

;...
;these memory locations hold the most recent numbers in the Fibonacci
;series
fib0     equ 10 ;lowest number (oldest when going up; newest when
;reversing down)
fib1     equ 11 ;middle number
fib2     equ 12 ;highest number
fibtemp  equ 13 ;temporary location for newest number
counter equ 14 ;indicates which value we have reached, opening value
;is 2

       org 00
```

Program Example 5.7: Storing the Fibonacci Series with Indirect Addressing

```
;preload initial values
       movlw 0
       movwf fib0
       movwf 20          ;save at start of list
       movlw 1
       movwf fib1
       movwf 21          ;save in list
       movwf fib2
       movwf 22          ;save in list
       movlw 3
       movwf counter ;have preloaded the first three numbers, so start
                     ;at 3
       movlw 23          ;Initialize File Select Register, with next
                         ;location for list of numbers to be saved
       movwf fsr

;
forward movf  fib1,0
       addwf  fib2,0
       btfsc  status,c   ;test if we have overflowed 8-bit range
       goto reverse      ;here if we have overflowed, hence reverse
                         ;down the series
       movwf  fibtemp    ;latest number now placed in fibtemp
       movwf  indf       ;save by indirect addressing
       movf   fsr,0      ;test to see if FSR is at top of range
       sublw  30
       btfss  status,z
       incf   fsr,1      ;increment FSR, if available range not full
       incf counter,1
;now shuffle numbers held, discarding the oldest
       movf   fib1,0     ;first move middle number, to overwrite oldest
       movwf  fib0
       movf   fib2,0
       movwf  fib1
       movf   fibtemp,0
       movwf  fib2
       goto   forward
```

Program Example 5.7: Continued

Programming Exercise 5.4

Create a project in MPLAB called Fibo+storage, or use a name of your choice. Copy from the book CD the source file of Program Example 5.7 into it and simulate. Display the File Registers window using View > File Registers. Using "Step Into", single-step

Programming Exercise 5.4 Continued

through the program and watch the Fibonacci series being built up in the data memory block starting at location 20_H. On completion you should have a window similar to that in Fig. 5.6. Notice in this also the two banks of SFRs, and how **PCL, STATUS** and **FSR** are mirrored across both banks. What does the program do when **FSR** has reached its maximum value? Try changing the definition of the top of the block by varying the literal value in the instruction **sublw 30**. Note the change in program action as you do this.

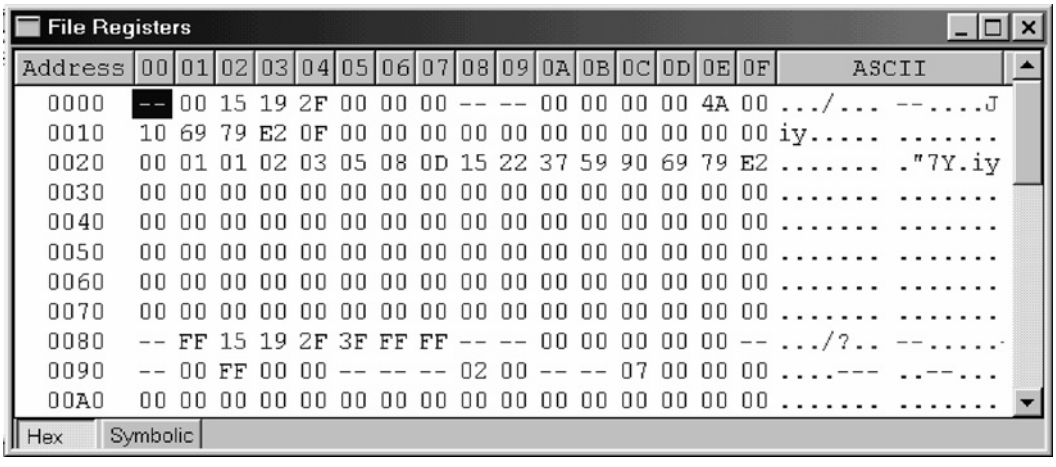


Figure 5.6: The Fibonacci Series Stored in a Data Block Starting to Address 20_H

5.7 Taming Assembler Complexity

You are now beginning to see how complex even a simple assembler program can become. We need every means possible of keeping the program as short and understandable as possible. A few options are now described.

5.7.1 Include Files

The assembler directive **#include** allows any file to be embedded within a program, thereby saving the trouble and space of pasting in large program sections which already exist elsewhere. A file so included is called an *Include File*. Initially, the most useful way to use this is to replace all the microcontroller-specific memory definitions that must occur at the start of a program. Like other assemblers, MPLAB contains Include files for each microcontroller, containing **equ** statements for all SFRs and their bits. These can be found in one of the MPLAB folders.

Use of an Include file is useful for a small microcontroller like the 16F84A, where the file is several pages long. It becomes almost essential for larger processors, which have a huge array of SFRs and hence very long Include files. Once an Include file is used, it is of course

essential to ensure that the microcontroller-specific labels that are referenced in the program are identical to the ones in the Include file. The advantage of an Include file, even for a very small application, is illustrated in Program Example 5.8.

```
;specify SFRs
timer     equ   01
status    equ   03
porta     equ   05                    #include p16f84A.inc
trisa     equ   05                    OR
portb     equ   06                    include p16f84A.inc
trisb     equ   06
intcon    equ   0B
```

the above, *and much more besides, can be replaced by:* ***the above***

Program Example 5.8: Using Include Files

5.7.2 *Macros*

We are finding in every program we see that program development for a RISC processor is laborious, due to the limited function of each individual instruction. A CISC instruction set, with its somewhat more powerful instructions, offers some modest advantage, as we saw in Section 4.9, but not much. Is there a way we can get around the minimalist nature of the instruction set while remaining in the assembler environment?

One answer to this problem is by the use of *macros*. A macro is a grouping of instructions, defined by the programmer and given a name. Once defined, the macro can be used in the program at any time. In some ways a macro offers the convenience of a subroutine, but it is used differently. When the source code is assembled, the macro is expanded out into the original instructions that made it up. Therefore, using macros is a form of shorthand in programming rather than a way of structuring the program.

Program Example 5.9 shows three macros inserted at the start of the ping-pong program (Appendix B). The macro itself is contained within the directives **macro** and **endm**. *Arguments* are defined for the macro, which are data values that the macro can apply. The macro **movlf** moves a data constant into a memory location. It applies two arguments, **const** and **address**. The macros **bfbset** (branch if file bit set) and **bfbclr** (branch if file bit clear) are similarly defined. All three macros are then applied within the first few lines of program, each time saving one line of code. Thus, the eight original lines of code in loop **wait** are reduced to four.

```
;now ready for action
;macro to move a literal value to a file
movlf       macro const,address
            movlw  const
            movwf  address
            endm
```

Program Example 5.9: Applying Macros to the Ping-Pong Program

```
;macro to branch if a specified bit is set
bfbset macro file,bit,target
           btfsc  file,bit
           goto   target
           endm
;macro to branch if a specified bit is clear
bfbclr macro file,bit,target
           btfss  file,bit
           goto   target
           endm
wait       movlf  04,porta      ;at rest, "out of play"
           movlf  00,portb      ;all play leds off
;both paddles must initially be clear before play allowed to commence
           bfbclr porta,4,wait ;go to wait if right paddle pressed
           bfbset porta,3,wait ;go to wait if left paddle pressed
;
```

Program Example 5.9: Continued

Programming Exercise 5.5

Enter the code of Program Example 5.8 into a copy of the ping-pong program, removing the lines of code it replaces. Assemble the code and open the list file. Notice how the original macro definition occupies no memory space and observe how the macro is expanded out into its original form whenever it is invoked, thus replicating the original ping-pong program. Continue through the program, applying these two macros wherever you can. How many times can you do this and how many lines of code do you save? Are there other macros that could usefully be defined?

5.7.3 MPLAB Special Instructions

Microchip further eases the problem of the restrictive RISC instruction set by defining a set of "special instructions." These are recognized by the assembler and expanded out to the equivalent instructions shown. Examples are given in Table 5.1, while a full listing appears in appendix B.11 of Ref. 4.1. Most are operations using or manipulating the **Z** or **C** bits in the Status register. Some, like **bc** or **bnc**, offer no saving in lines of code, but improve the clarity of programming. Others, like **addcf**, create new and useful functions not originally available in the instruction set, which are very similar to CISC instructions.

5.8 More Use of the MPLAB Simulator

We have already seen the enormous value of the software simulator as a means of running through a program and observing outputs. We did this just using the simple controls, of

Table 5.1: Example MPASM™ "Special" Instructions

Mnemonic	Description	Equivalent	Status Flags Affected
addcf f,d	Add Digit Carry to file	BTFSC 3,1 INCF f,d	Z
bc k	Branch on Carry	BTFSC 3,0 GOTO k	–
bnc k	Branch on No Carry	BTFSS 3,0 GOTO k	–
clrc	Clear Carry	BCF 3,0	–
movfw f	Move file to W	MOVF f,0	Z
subcf f,d	Subtract Carry from file	BTFSC 3,0 DECF f,d	Z
tstf f	Test file	MOVF f,1	Z

single-step, animate or run. As programs grow, however, we need greater sophistication in the way we can run them and how we observe their behavior.

5.8.1 Breakpoints

Once programs become long, it becomes increasingly tedious to step through them when simulating. We need a means of getting them to run through the code that we may not be interested in, but stopping where we need to take a closer look at what is happening. Breakpoints allow this functionality. In their simplest form, breakpoints allow you to run a program up to a specified instruction. Program execution then stops, and memory and register values can be inspected. In MPSIM™ you can set a breakpoint simply by double-clicking on an instruction in the program window, and remove it in the same way. The number of breakpoints is unlimited, so they can be used freely.

Programming Exercise 5.6

The Fibonacci program is perhaps the longest example we have look at so far, and it is annoying to have to step through it if we wish to see something happen deep inside the program. Open the Fibonacci project you created earlier (or create it for the first time) and select the MPLAB simulator using Debugger > Select Tool > MPLAB SIM. Scroll through the **.asm** source file and double-click on the line labeled **reverse**. A breakpoint symbol should appear, as seen in Fig. 5.7. Check that you can remove this by double-clicking again. Reset the simulator and run. See how the program stops at the breakpoint. You can inspect all windows at this point and then proceed any way you wish, for example by single-stepping. Try setting another breakpoint at the second line shown in Fig. 5.7 and running to here.

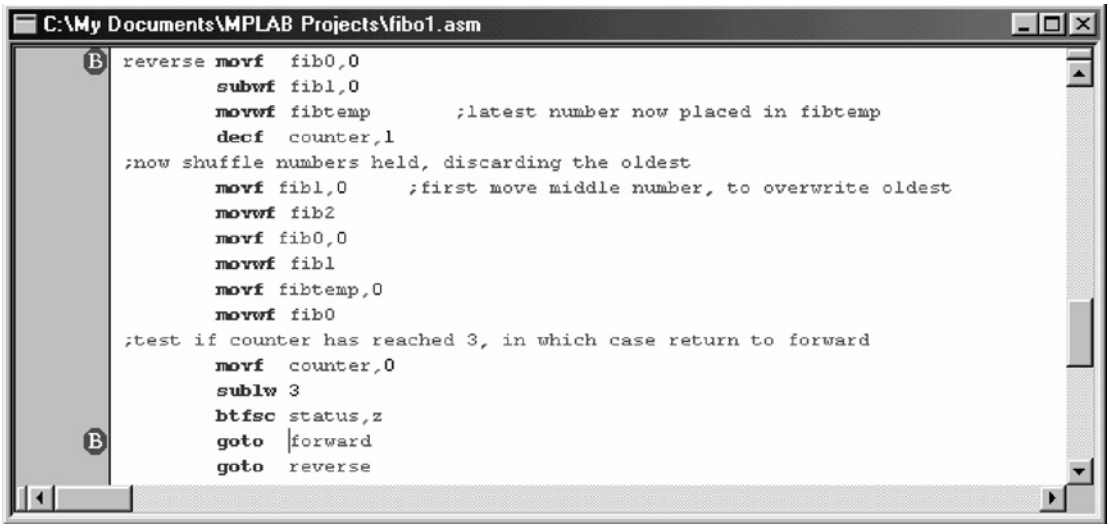


Figure 5.7: Breakpoints Inserted in Fibonacci Program

5.8.2 Stopwatch

A weakness of the software simulator is that it does not run in real time, yet in embedded systems we have a strong desire to understand the timing behavior of our programs. The Stopwatch facility of the simulator allows accurate time measurements to be simulated. It simply requires that the simulator "knows" what the oscillator frequency is. As it can record the number of instruction cycles executed, it can then calculate time taken.

In MPSIM the oscillator frequency is set through the Simulator Settings window, found using Debugger > Settings > Osc/Trace (Fig. 5.8(a)). The Stopwatch (Fig. 5.8(b)) is displayed using Debugger > Stopwatch.

Programming Exercise 5.7

Still with the Fibonacci project open, set the processor frequency to 4 MHz. This usefully gives an instruction cycle time of 1 μs. Leave the breakpoints as set in Programming Exercise 5.6. Press Debugger > Stopwatch, and Zero the Stopwatch. Reset the simulator and run the program to the breakpoint. The Stopwatch should show 166 μs. Can you account for this value?

5.8.3 Trace

The various windows available in MPSIM give a good picture of the state of the processor status and memory locations at any time, but they do not tell us the history of program

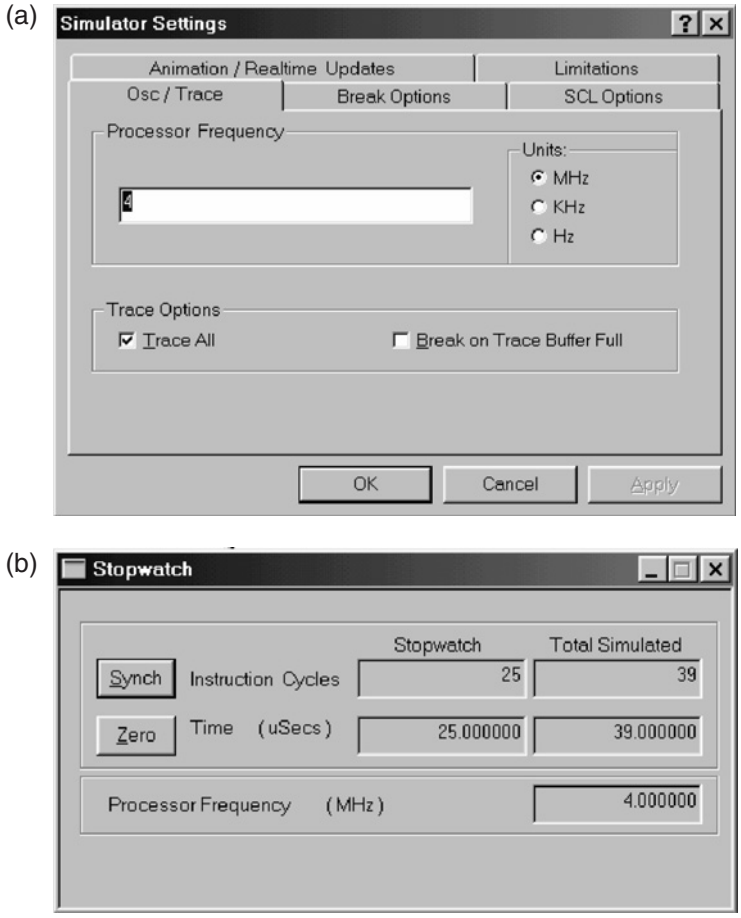


Figure 5.8: Using the Stopwatch: (a) Simulator Settings Window and (b) Stopwatch Window

execution. Even if program execution has halted at a breakpoint, there may have been a number of program paths for it to go down to reach that point. The *Trace* function is there to give a record of the recent past of the program execution. In Trace memory the simulator keeps a continuous record of all instructions that have been executed. This can be inspected when program execution stops.

MPLAB has a Trace function, with memory size of 32 767 lines. The Trace function is enabled in the Simulator Settings window (Fig. 5.8(a)), found by following Debugger > Settings > Osc/Trace. Note that it slows down simulator speed somewhat if it is enabled. The Trace window is viewed using View > Simulator Trace. Here the columns are all self-explanatory, except for:

SA = Source address—address or symbol of the source data
SD = Source data—value of the source data

DA = Destination address—address or symbol of the destination data
DD = Destination data—value of the destination data.

Programming Exercise 5.8

Return again to the settings of Programming Exercise 5.6. Ensure the Trace is enabled as described above, reset the simulator and run to the breakpoint at label **reverse**. Now open the Trace window, which should appear as in Fig. 5.9. See that it is a list of all the instructions recently executed, finishing (in line 142) with the **goto** instruction which takes execution to the breakpoint line. Looking back over the SD and DD columns, we see the Fibonacci series being formed. The value of the Status register in line 141 is 19_H, or 0001 1001_B. This shows that bit 0, the Carry flag, has been set and the **goto** instruction is accordingly invoked.

Trace

Line	Addr	Op	Label	Instruction	SA	SD	DA	DD	Cycles
134	000F	0812		MOVF 0x12, W	0012	90	0012	90	00000009B
135	0010	0091		MOVWF 0x11	----	--	0011	90	00000009C
136	0011	0813		MOVF 0x13, W	0013	E9	0013	E9	00000009D
137	0012	0092		MOVWF 0x12	----	--	0012	E9	00000009E
138	0013	2807		GOTO 0x7	----	--	----	--	00000009F
139	0007	0811	forward	MOVF 0x11, W	0011	90	0011	90	0000000A1
140	0008	0712		ADDWF 0x12, W	0012	E9	0012	79	0000000A2
141	0009	1803		BTFSC 0x3, 0	0003	19	----	--	0000000A3
142	000A	2814		GOTO 0x14	----	--	----	--	0000000A4

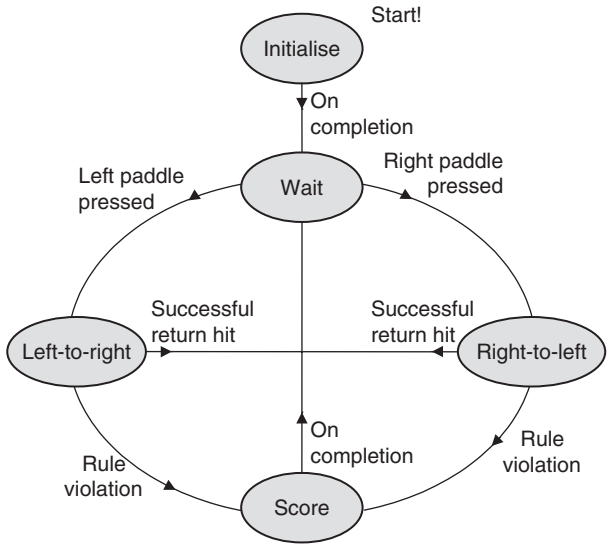
Figure 5.9: Trace Window for Section of Fibonacci Program

5.9 The Ping-Pong Program

It is useful now to look at the full ping-pong program, as seen in Appendix B. It is never simple looking at assembler code written by someone else (in fact, it's often difficult looking at your own assembler code!), so you should not feel worried if initially it appears difficult.

5.9.1 A Structure for the Ping-Pong Program

Let us first of all try to get a feel for the overall structure. For this program a state diagram gives a clear overall representation, which would be difficult to achieve with a flow diagram. This is seen in Fig. 5.10.



Rule violation:
1. Attempting to hit the ball when it is not at your end.
2. Not hitting ball when it *is* at your end.

Scoring
You score when your opponent violates a rule.

Successful return hit
Hitting the ball when it is at your end.

Note: ball is at "your end" when LED nearest you is lit.

Figure 5.10: The Ping-Pong Program Visualized as a State Diagram

The program starts in the *Initialize* state. When this has completed it immediately enters a *Wait* state, where it stays until play commences. If the left player presses a paddle then a *Left-to-right* state is entered. In this the "ball" starts at the left-most position and starts moving towards the right. Exit from this state occurs either if there is a rule violation (for which definitions are given) or if there is a successful return hit, when the ball has reached the right-most position. With no rule violation play continues, with the state alternating between *Left-to-right* and *Right-to-left*. When either player makes a mistake this is classified as a rule violation and the game enters a *Score* state. It leaves this when scoring is complete and returns to the *Wait* state.

Having grasped the ping-pong state diagram, try to find each state in the Assembler listing. Three of the five states, Initialize, Wait and Score, should be easy to follow. Indeed, we have met the first in Chapter 4. The two states where play is actually in progress, Left-to-right and Right-to-left, are a little more difficult to grasp. Each is a mirror image of the other, so when one is understood, the other immediately follows.

While the program overview is best represented as a state diagram, the actual Right-to-left/ Left-to-right states are essentially looping structures, and are most easily represented as a flow diagram (Fig. 5.11).

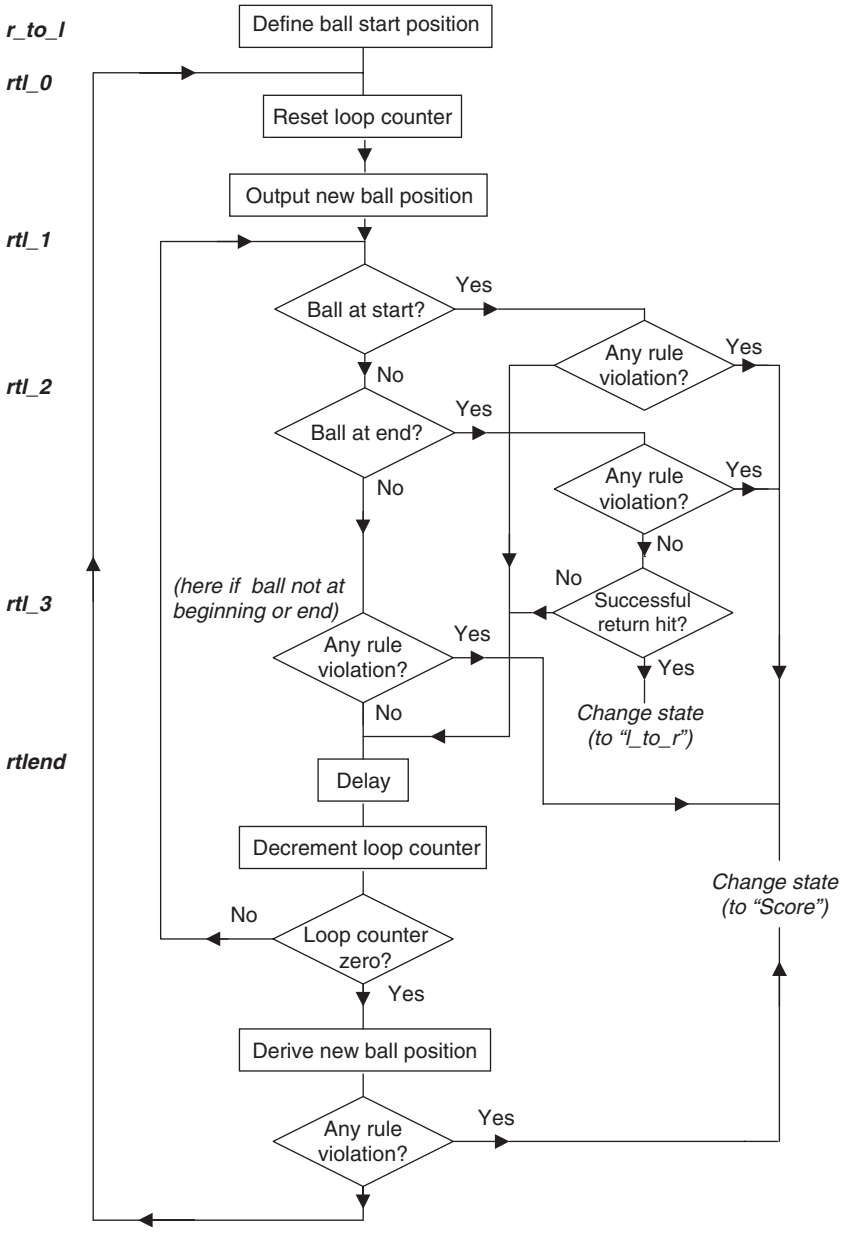


Figure 5.11: Flow Diagram of Right-to-Left/Left-to-Right States

Here we are confronted, perhaps for the first time, with the detailed complexity that such a program requires, even in a product that appears so simple. There are certainly a number of requirements to be met within the state. The ball is to move by lighting a series of LEDs, each to be illuminated for a set period of time. The state of the paddles is to be continuously checked; at certain times a paddle press is a legal action, at others it represents a rule violation. If timing were to be achieved simply by entering a timing loop, the function of input checking could not be carried out. Hence each LED time duration is made up of a certain number of loop iterations—within each the inputs are tested, followed by a short delay.

5.9.2 Exploring the Ping-Pong Program Code

As an aid to further understanding, certain sections of the ping-pong code are now described.

5.9.2.1 Opening Section and Memory Allocation

In the opening comments the program gives details on hardware allocation. This is followed by a section on memory allocation. Here names are given to memory locations in the general-purpose RAM area. The names used, **delcntr1**, etc., are chosen by the programmer and are placed as labels, i.e., starting fully left on the program line. The **equ** directive is used and the memory location is chosen from the memory map of Fig. 2.5, which shows that available memory locations range from address $0C_H$ to address $4F_H$.

5.9.2.2 The Wait State

Let's explore this by looking at the opening part of the actual ping-pong program, which follows the initialization section.

In the first four lines the program switches on the out-of-play LED and switches off all others. It then tests the state of both paddles. Remember that, when pressed, the switch input bit goes to Logic 0. If neither is pressed then program execution skips forward to **wait1**. However, if one or both are pressed, then program execution just returns to **wait** and loops until the button is released. This is to stop a "false start" to play, which would otherwise occur if a player switched on the game while a paddle was pressed. As this is the point where play restarts after a score, it also ensures that the previous round of play is completed before starting again. Note that the loop execution includes the setting of the LEDs. This is not strictly necessary but does no harm and minimizes the number of labels used.

Program execution then enters another loop, **wait1**. Both buttons have been cleared, so the game can now start. Again both paddles are tested. This time, however, if a button is pressed, instead of looping back play goes forward, to either **l_to_r** or **r_to_l**.

5.9.2.3 The Main Play States

Let us start by looking at the **r_to_l** section. This opens with the "out-of-play" LED being switched off and the opening LED position being defined. The larger loop then starts at the

line labelled **rtl_0**. Here the loop counter **loop_cntr** is loaded with the number **led_durn**. This number was defined in the opening section of the program and represents the number of times the inner loop is to be iterated. This inner loop starts at line **rtl_1**. Much of it is concerned with checking for rule violation, the interpretation of which depends on the position of the ball. The general structure is shown in the flow diagram, while the actual rule interpretation can be determined from looking at the source code. Scoring occurs when any rule violation is detected. At the end of the loop the 5 ms delay subroutine is called. The loop counter is decremented. If zero, then a new ball position is set up, by rotating the **led_posn** memory location. A score occurs if this causes the ball to go off the end of the 8-bit number; this happens if there has not been a successful return hit while the ball has been at the end position.

The Score state is divided into two parts and is simple. It lights the appropriate Score LED, calls a half-second delay and switches off the LED. The state is then left and execution returns to the Wait state.

5.10 Simulating the Ping-Pong Program—Tutorial

The ping-pong program is not exactly complex, but it is full of loops and delays, and therefore illustrates the problems of thoroughly testing a program with a simulator. The art lies in using each feature where needed. Generally, for a program segment that is to be explored in detail, you will single-step or animate. For the sections of code you want to get through quickly, you will simply run through, heading for a breakpoint you have already inserted. The following is a tutorial that guides you through the simulation of this program.

Ensure that you have created and built a project, which contains a copy of the ping-pong program.

5.10.1 Setting up Input Stimulus

The ping-pong program has two digital inputs, the two player paddles, which need to be simulated. Go to Debugger > Stimulus Controller. This will give you the Stimulus Controller dialogue box. Under Pin select RA3, and under Action select Set High. Repeat this for RA4. Then create two more lines, for RA3 and RA4 again, this time with Pulse Low as the action for each. Set the duration to 50 ms, which is representative of a fast switch push. Can you work out from the program what is the maximum theoretical duration a player can press a button?

5.10.2 Setting up the Watch Window

Click View > Watch to set up a Watch window. A useful selection for the ping-pong is **PCL** (to track where you are in the program), **PORTA** and **PORTB** using the Add SFR button, and **duration, led_posn** and **delcntr1** using the Add Symbol button. By right-clicking on the title

bar near the top of the Watch window, you will see that you can display further columns, for example the data shown in binary. The Watch window you set up will be saved for you at the end of your session.

5.10.3 Single Stepping

Press F6 to reset the Program Counter and then try single stepping the ping-pong program. You will be able to see registers changing under program instruction, with the changes being highlighted in red.

If one or both of the user paddles are set low (i.e., "pressed"), then the simulation will get stuck in the first wait loop. (You can see their logic state by inspecting the Port A display in the Watch window.) Set these lines high by pressing the Fire buttons on the Stimulus Controller box. You should see the change reflected in the Watch window.

Now you should be able to single step on to the **wait1** loop. Having looped round here once or twice, Fire the RA3 pulse. You should now exit the loop and move on to **l_to_r**. See the Port B value change to 01 as the ball position is set up. You can continue stepping from here, and either step over, or enter, the **delay5** subroutine. Once in, you can Step Out of it at any time. Clearly it would be tedious to single step all the way through this subroutine. Even if we step over it, the loop repetitions become endless and the limitations of single stepping are revealed.

5.10.4 Animate

Press F6 again and try running the program in Animate mode. Adjust the speed to one you are comfortable with by adjusting the setting in Debugger > Settings > Debugger Animation. Now you can't use the Step Over function and you will find yourself stuck again in the delay subroutine. Here you can watch the **delcntr** value being decremented in the File Registers window.

5.10.5 Run

If you select Run there is not much to watch, as the memory windows are no longer updated as the program runs. Stimulus inputs are, however, still accepted. Hence it is helpful to start using breakpoints.

5.10.6 Breakpoints

Set a breakpoint initially at the **l_to_r** label. Now reset the simulated CPU, set RA3 and RA4 high in the Stimulus Controller box, and enter Run. Fire the pulse on RA3. The program should then halt at your breakpoint. Without resetting, try setting another breakpoint at the **ltr_1** label and press Run again. You have an unlimited number of breakpoints in MPLAB simulator, so use them freely.

5.10.7 Stopwatch

Using Debugger > Settings > Osc/Trace, set the processor frequency to 800 kHz, which is the nominal frequency for the ping-pong. Set a breakpoint at the first line of the **delay5** subroutine and run to there. Now press Debugger > Stopwatch. Zero the Stopwatch and insert a breakpoint at the Return instruction of that subroutine. Run the program to there. Does the Stopwatch value agree with the calculated value of the delay routine? This is a very useful facility for measuring durations of program execution.

5.10.8 Trace

Try enabling the Trace, running the program to a breakpoint and then inspecting Trace memory in View > Simulator Trace.

Table 5.2: Suggested Breakpoint Locations for Ping-Pong Debug

Breakpoints	Action when Breakpoint Reached
wait	Check value of Port A—are bits 3 and 4 high? Press Run to loop here and see "out-of-play" LED being set high by the program. Set bits 3 and 4 high with the Stimulus generator. Press Run.
wait1	Port A should now be 0001 1100. Pulse RA4 low with Stimulus generator (i.e. a player initiates the game); this will stay low for 50 ms of run time. Press Run.
r_to_l l_to_r	Here if RA4/RA3 has been pressed, note that this bit is now low in Port A. Press Run.
rtl_1 ltr_1	Single step a few lines here and see **led_posn** set to new value. This represents the ball position. Press Run. Watch **loop_cntr** step down on each loop iteration. Then watch **led_posn** change.
rtlend ltrend	Have completed one internal cycle, will now call a delay. Press Run. Remove breakpoint for faster running.
score_left	Here if an "illegal" paddle press forces a Score. Press Run.
score_right	Here if an "illegal" paddle press forces a Score. Press Run.
delay5	Single step a few times through this loop. See value of **delcntr1** set and then decremented. Press Run. Remove breakpoint for faster running.
delay500	Here if a Score LED is lit. It will be illuminated for 0.5 s. Press Run.

5.10.9 Debugging the Full Program

Try inserting breakpoints at the positions shown in Table 5.2. Note that all breakpoints must be entered on lines containing instructions. Reset the Program Counter with F6 and run the program. Step from breakpoint to breakpoint, and observe the Watch window values shown in the table.

Having looped around a few times and understood the program, try in the simulator:

- An "illegal" paddle press to force a score
- Looping until the ball reaches the far end and returning it with a "legal" paddle press.

5.11 What Others Do—Graphical Simulators

These past two chapters have aimed to give a good introduction to MPLAB and its simulator, MPSIM. While powerful in their own way, it is worth reminding oneself that they are free. What if we are willing to spend some money on a simulator?

There are a number of simulators available which go well beyond the simple text-based interface of MPSIM. An example is the simulator found in Ref. 5.2, shown in Fig. 5.12. Here a 16F84 microcontroller is being simulated. The W register, pipelined instruction, current instruction, Stack, Status register, ports and program listing are all clearly displayed. The program can be run or be single stepped, with the internal status being clearly updated and displayed.

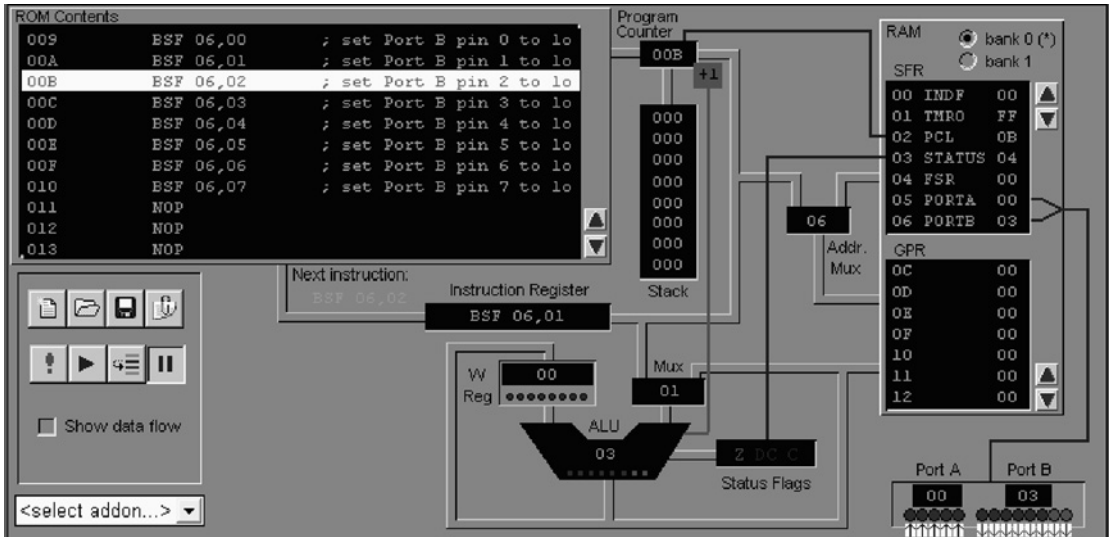


Figure 5.12: The "Virtual PICmicro" Screen, Matrix Multimedia

5.12 Summary

- It is important to devise good structures for programs as they are developed. Flow and state diagrams can help with this.
- A number of techniques assist in producing clear and well-structured programs. These include subroutines, look-up tables, macros and the use of Include Files.

- The full 16F84A instruction set can be applied to build big and sophisticated programs.
- More complex programs require greater expertise in the use of simulation techniques.

References

[5.1] *Implementing a Table Read*. Microchip, Application Note AN556; www.microchip.com.

[5.2] Assembly for PICmicro™ Microcontrollers, V3.0. John Becker, Matrix Multimedia Ltd; http://www.matrixmultimedia.co.uk or http://www.labvolt.com/

CHAPTER 6

Further Programming Techniques

Now that the basic programming methods have been introduced, we can look at some more advanced techniques. Sample programs demonstrating use of the timer, interrupts and data table are included in this chapter.

6.1 Program Timing

The microcontroller program execution is driven by the clock signal generated by an internal oscillator whose frequency is controlled by either an external RC or crystal (XT) network. This signal is divided into four internal clocks (Q1–Q4) which run at a quarter of the oscillator frequency ($F_{OSC}/4$). These provide four separate pulses during each cycle to trigger the processor operations. These include fetching the instruction code from the program memory, and copying it to the instruction register. The instruction code is then used by the decoder to set up the control lines to carry out the required process. The four clocks are used to operate the data gates and latches within the MCU to complete the data movement and processing.

This instruction timing is illustrated in Fig. 6.1. Note that, if the CR clock option is used, an output instruction clock signal at $F_{OSC}/4$ is available at the CLKOUT pin to operate external circuits synchronously. It can also be used in hardware testing to check that the clock is running, and to measure its frequency.

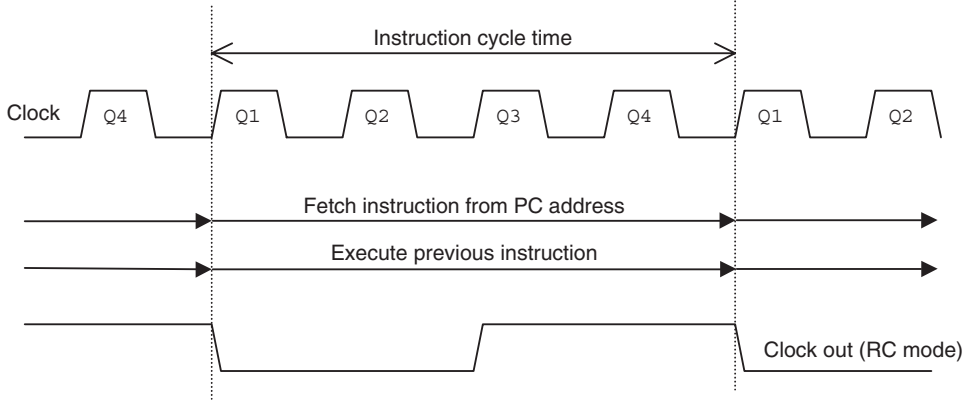


Figure 6.1: PIC Instruction Timing

The result of this clocking scheme is that each instruction takes four clock cycles to execute, unless a jump (GOTO or CALL) occurs. These will take eight clock cycles, because the program counter contents have to be replaced, and this takes an extra instruction cycle.

6.1.1 Pipelining

The instruction fetch and execute cycles can be carried out simultaneously, because the data is being transferred on separate data paths. While one instruction is being executed, the next is being fetched from the program memory into the instruction register. This overlapping of execution stages is called *pipelining*, with the PIC having a two-stage pipeline. The CISC microprocessors such as Pentium use more elaborate pipelining to break the instruction processing into multiple stages, and thereby boost performance.

6.1.2 Execution Time

We can now predict how long a particular sequence will take to execute. A clock rate of 4 MHz is a convenient default value because it is the maximum operating frequency in standard XT mode (see the PIC 16F84A data sheet, Table 6.1), and also gives an instruction execution rate of 1 MIP (millions of instructions per second) and an instruction cycle time of 1 s.

A delay loop is shown in Table 6.1. The move instructions take one cycle each, and the DECFSZ instruction is then repeated 254 times. The GOTO takes two cycles, because each time the GOTO is executed, the RETURN is prefetched, and then not executed, so a cycle is wasted. On the 255th loop, the register becomes zero and the GOTO is skipped, and the RETURN executed. This also takes two cycles, because of another wasted prefetch cycle,

Table 6.1: Sequence Execution Time

Label	*Instruction*	*Operand*	*Time (cycles)*
delay	**MOVLM**	**0xFF**	1
	MOVWF	**timer**	+1
down	**DECFSZ**	**timer**	+(1 × 255)
	GOTO	**down**	+(2 × 254)+1
	RETURN		+2
		Total	768

Clock frequency = 4 MHz
Instruction frequency = 1 MHz
Instruction period = 1 μs
Total delay time = 768 μs

but is only executed once per delay sequence. The total loop time can then be calculated, by totaling the time taken for each instruction and the loop. As we can see, this comes to 768 s, at 4 MHz. This figure can be confirmed if the program containing the loop is run in the simulator, using the stopwatch, with the clock frequency set to 4 MHz.

The block execution time for a section of code can thus be predicted before testing in simulator or hardware. Alternatively, the timing can be checked and modified using the simulator. Incidentally, NOP (No OPeration) is useful here. For time critical sequences, NOP may be used to insert a delay of one instruction cycle, that is, four clock cycles; it has no other effect. Using this, a delay of 1 ms can be created using the delay loop with the count set to 246 and a NOP in the loop to make the loop execution time 4 s. The total loop time is then (246 × 4s) plus a few cycles for the loop initialization and return.

6.2 Hardware Counter/Timer

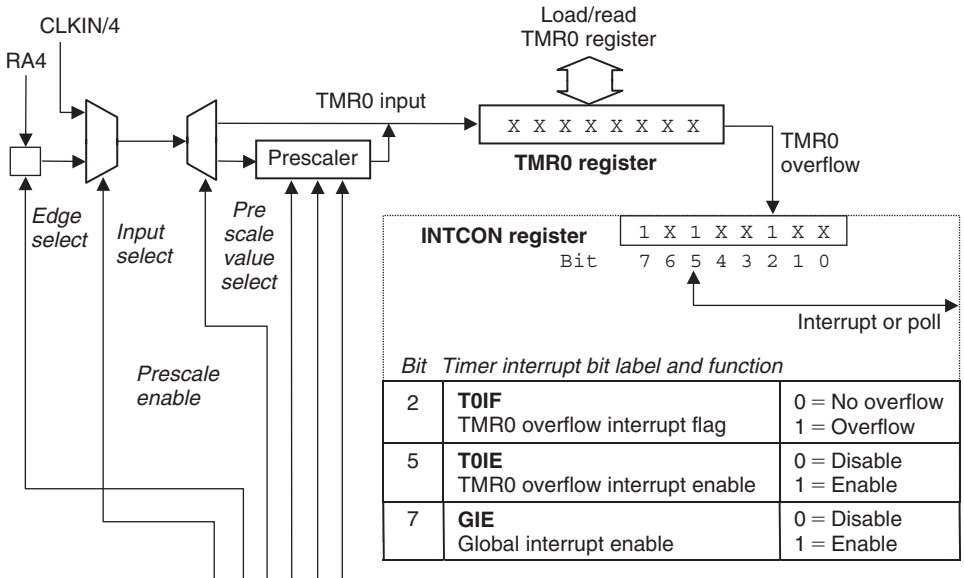
Accurate event timing and counting is often needed in microcontroller programs. For example, if we have a sensor on a motor shaft which gives one pulse per revolution of the shaft, the number of pulses per second will give the shaft speed. Alternatively, the interval between pulses can be measured, using a timer, to obtain the speed by calculation. A process for doing this would be:

1. wait for pulse,
2. read and reset the timer,
3. restart the timer,
4. process previous timer reading,
5. go to 1.

If an independent hardware timer is used to make the measurement, the controller program can carry on with other operations, such as processing the timing information, controlling the outputs and checking the sensor input, while the timer keeps an accurate record of the time elapsed.

6.2.1 Using TMR0

The special file register 01 in the 16F84 is called timer zero (TMR0); it is an 8-bit counter/timer register which, once started, runs independently. This means it can count inputs or clock pulses concurrently with (at the same time as) the main program execution. The counter/timer can also be set up to generate an interrupt when it has reached its maximum value, so that the main program does not have to keep checking it to see if a particular count has been



Bit	Timer interrupt bit label and function	
2	**T0IF** TMR0 overflow interrupt flag	0 = No overflow 1 = Overflow
5	**T0IE** TMR0 overflow interrupt enable	0 = Disable 1 = Enable
7	**GIE** Global interrupt enable	0 = Disable 1 = Enable

X X X 0 0 0 0 0 **OPTION register**

Bit 7 6 5 4 3 2 1 0

Bit	Timer control bit label and function	Prescaler division ratio (2 4 8 16 32 64 128 256)
0	**PS0** Prescaler rate select bit 0	0 1 0 1 0 1 0 1
1	**PS1** Prescaler rate select bit 1	0 0 1 1 0 0 1 1
2	**PS2** Prescaler rate select bit 2	0 0 0 0 1 1 1 1
3	**PSA** Prescaler assignment bit	0 = Select prescaler for TMR0 1 = Deselect prescaler for TMR0
4	**T0SE** TMR0 source edge select bit	0 = Increment on rising edge of RA4 1 = Increment on falling edge of RA4
5	**T0CS** TMR0 clock source select bit	0 = Instruction clock = Ext. clock/4 1 = Select RA4 input

Figure 6.2: Hardware Counter/Timer Setup and Operation

reached. A block diagram of TMR0 and its associated hardware and control registers is shown in Fig. 6.2.

As an 8-bit register, TMR0 can count from 00 to FF (255). The operation of the timer is set up by moving a suitable control code into the OPTION register. The counter is then clocked by an external pulse train, or, more usually, from the chip oscillator. When it reaches its maximum value, FF, and is incremented again, it "rolls over" to 00. This register overflow is recorded by the INTCON (interrupt control) register, bit 2 (T0IF), going to "1" (assuming that it has been previously enabled and cleared). This condition can be checked by bit testing in the program, or can trigger an interrupt (Section 6.3).

6.2.2 Counter Mode

The simplest mode of operation of TMR0 is counting pulses applied to RA4, which has the alternate name T0CKI, Timer Zero Clock Input. These pulses could be input manually from a push button, or, more likely, would be produced by some other signal source, such as the sensor on the motor shaft mentioned above. If the sensor produces one pulse per revolution of the shaft, and one of the PIC outputs controls the motor, the microcontroller could be programmed to rotate the shaft by a set number of revolutions. If the motor were geared down, a positioning system could be designed to move the output through a set angle, in a robot, for example.

In order to increase the range of this kind of measurement, the prescaler allows the number of pulses received by the TMR0 register to be divided by a factor of 2, 4, 8, 16, 32, 64, 128 or 256. The ratio is selected by loading the least significant three bits in the OPTION register as follows: 000 selects divide 2, 001 divide by 4 and so on up to 111 for divide by 256. TMR0 can also be preloaded with a value, and the overflow detected when it has been "topped up" by a set number of pulses.

6.2.3 Timer Mode

The internal clock is selected by setting the OPTION register, bit 5, to 0. To use TMR0 as an accurate hardware timer, a crystal oscillator must be used as the chip clock source. A convenient crystal frequency is 4 MHz, because it is divided by four before it is fed to the input of TMR0, giving a pulse frequency of 1 MHz. The counter would then be clocked every 1 s exactly, and would take 256 s to count from zero to zero again. Again, by preloading with a suitable value, a smaller time interval could be selected, with time out indicated by the timer interrupt flag. For example, by preloading with the value 156 (6C), the overflow would occur after 100 s. Alternatively, the time period measured can be extended by selecting the prescaler. The maximum timer period would then be 512 s, 1024 s and so on to 65.536 ms. Crystals are also available in frequencies that are more conveniently divisible by 2. For example, a 32.768 kHz crystal frequency will produce a time-out every 1.0000 s, if the prescale value of 32 is selected.

In Fig. 6.1, TMR0 is set up with $xxx00000_2$ in the option register, selecting the internal clock source, with a prescale value of 2. The INTCON register has been set up with the timer interrupt enabled and the timer overflow interrupt flag has been set (overflow has occurred).

6.2.4 TIM1 Timer Program

Program TIM1, which demonstrates the use of the timer, is listed as Program 6.1. It is designed to increment a binary output once per second. The program uses the same demonstration BIN hardware as the previous programs, with eight LEDs displaying the contents of Port B.

```
; ***************************************************************
;       TIM1.ASM        M. Bates       6/1/66      Ver 1.2
; ***************************************************************
;
; Minimal program to demonstrate the hardware timer operation.
;
; The counter/timer register (TMR0) is initialized to zero and
; driven from the instruction clock with a prescale value of 64.
;
; T0IF is polled while the program waits for time out.
; When the timer overflows, the Timer Interrupt Flag (T0IF) is set.
; The output LED binary display is then incremented. With the clock
; adjusted to 65536 Hz, the LSB LED flashes at 1 Hz.
;
;       Processor:      PIC 16F84
;
;       Hardware:       PIC BIN Demo Hardware
;       Clock:          CR = 65536 Hz (approx)
;       Outputs:        RB0 -RB7: LEDs (active high)
;       WDTimer:        Disabled
;       PUTimer:        Enabled ; Interrupts: Disabled
;       Timer:          Internal clock source
;                       Prescale = 1:64
;       Code Protect:   Disabled
;
;       Subroutines:    None
;       Parameters:     None
;
; ***************************************************************

; Register Label Equates.....................................

TMR0    EQU     01              ; Counter/Timer Register
PORTB   EQU     06              ; Port B Data Register (LEDs)
INTCON  EQU     0B              ; Interrupt Control Register

T0IF    EQU     2               ; Timer Interrupt Flag

; ***************************************************************

; Initialize Port B (Port A defaults to inputs).............

        MOVLW   b'00000000'     ; Set Port B Data Direction
        TRIS    PORTB
```

Program 6.1: TIM1 Source Code

```
        MOVLW     b'00000101'     ; Set up Option register
        OPTION                    ; for internal timer/64
        CLRF      PORTB           ; Clear Port B (LEDs Off)
; Main output loop. ..................................
next    CLRF      TMR0            ; clear timer register
        BCF       INTCON,T0IF     ; clear timeout flag
check   BTFSS     INTCON,T0IF     ; wait for next timeout
        GOTO      check           ; by polling timeout flag
        INCF      PORTB           ; Increment LED Count
        GOTO      next            ; repeat forever...
        END                       ; Terminate source code
```

Program 6.1: Continued

An adjustable CR clock is used, set to give a frequency of 65536 Hz (approximately). This frequency is divided by four, and is then divided by 64 in the prescaler, giving an overall frequency division of $4 \times 64 = 256$. The timer register is therefore clocked at 65536/256 = 256 Hz. The timer register counts from zero to 256 and so overflows every second. The output is then incremented; it will take 256 s to complete the 8-bit binary output count.

6.2.5 Timing Problems

Each instruction in the program takes four clock cycles to complete, with jumps taking eight cycles. If the program sequence is studied carefully, extra time is taken in completing the program loop before the timer is restarted. In this application, it will cause only a small error, but in other applications it may be significant. Also notice that the program has to keep checking to see if the time-out flag has been set by the timer overflowing. It is more efficient to allow the processor to carry on with some other process while the timer runs, and allow the time-out condition to interrupt the main program when it has finished.

6.2.6 More Timers

Because they are so useful, some larger PIC chips have more than one timer/counter. The 16F877, for example, has three. In addition to Timer 0 (TMR0), it has Timer 1, a 16-bit counter (using two registers in cascade) which provides an accurate count up to 64 535. It can also operate with its own independent oscillator, which can operate with a 32.768 kHz crystal, which can be used to give an accurate time interval up to 2 s. Timer 2 is another 8-bit counter which is designed to be used in generating a pulse-width-modulated output signal. This can be used to drive motors and other loads which require a variable power input. Obviously, additional control registers are needed to setup and operate these extra timers.

6.3 Interrupts

Interrupts are generated by an internal or external asynchronous (not linked to the program timing) event, and the interrupt signal can be received at any time during the execution of the main process. For example, when you hit the keyboard or move the mouse on a PC, an interrupt signal is sent to the processor from the keyboard interface to request that the key be read in, or the mouse movement transferred to the screen. The code which is executed as a result of the interrupt is called the "interrupt service routine" (ISR). When the ISR has finished its task, the process which was interrupted must be resumed as though nothing has happened. This means that any information being processed at the time of the interrupt may have to be stored temporarily, so that it can be recalled later. The program counter is saved automatically on the stack, as when a subroutine is called, so that the program can return to the original execution point after the ISR has been completed. This system allows the CPU to get on with other tasks without having to keep checking all the possible input sources.

6.3.1 Interrupt Setup

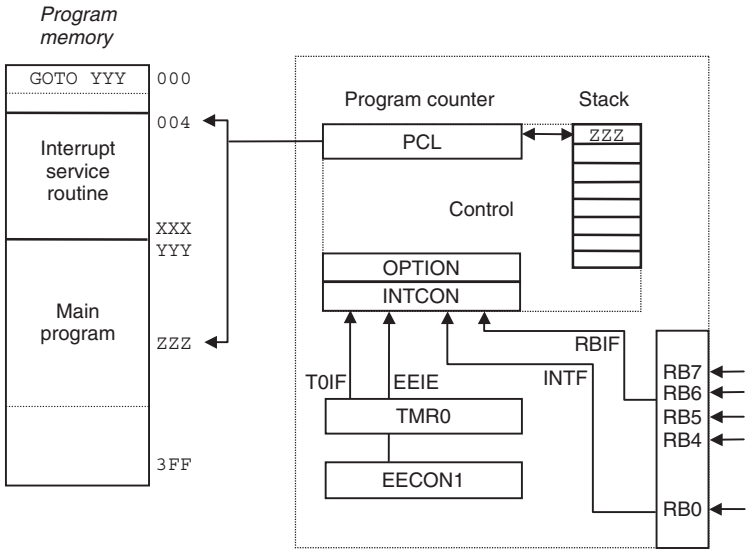
A block diagram detailing the 16F84 interrupt system is given in Fig. 6.3. The PIC has four possible interrupt sources:

1. RB0 can be selected as an edge-triggered interrupt input by setting INTCON, 4 (INTE), with the active edge selected by OPTION,6 (INTEDG).
2. RB7–RB4 can be selected to trigger an interrupt if any of them changes state, by setting INTCON,3 (RBIE).
3. TMR0 overflow interrupt can be selected by setting INTCON,5 (T0IE).
4. Completion of an EEPROM (nonvolatile data) write operation can be used to trigger the interrupt.

If interrupts are required, the interrupt source must be enabled in the INTCON (interrupt control) register. Then, the global interrupt enable bit, which enables all interrupts, must be set (INTCON,7) and finally the specific interrupt bit must be set. Note that, although there are four interrupt sources, they will all call an ISR at location 0004. If more than one interrupt source is to be used, a mechanism for identifying which is active must be included in the application program. There is no hardware interrupt priority system, as is available in more complex processors.

6.3.2 Interrupt Execution

Interrupt execution is also illustrated in Fig. 6.3. Each interrupt source has a corresponding flag, which is set if the interrupt event has occurred. For example, if the timer overflows, T0IF (INTCON,2) is set. When this happens, and the interrupt is enabled, the current instruction is



Interrupt control bit functions

	Bit	Label	Function	Settings
INTCON	0	**RBIF**	Port B (4:7) Interrupt flag	0 = No change 1 = Bit change detected
	1	**INTF**	RB0 Interrupt flag	0 = No interrupt 1 = Interrupt detected
	2	**T0IF**	TMR0 overflow Interrupt flag	0 = No overflow 1 = Overflow detected
	3	**RBIE**	Port B (4:7) Interrupt enable	0 = Disabled 1 = Enabled
	4	**INTE**	RB0 Interrupt enable	0 = Disabled 1 = Enabled
	5	**T0IE**	TMR0 overflow Interrupt enable	0 = Disabled 1 = Enabled
	6	**EEIE**	EEPROM write complete interrupt enable flag	0 = Disabled 1 = Enabled
	7	**GIE**	Global interrupt enable	0 = Disabled 1 = Enabled
OPTION	6	**INTEDG**	RB0 interrupt Active edge select	0 = Falling edge 1 = Rising edge

Figure 6.3: Interrupt Setup and Operation

completed and the next program address is saved on the stack. The program counter is then loaded with 004, and the routine found at this address is executed. Alternatively, location 004 can contain a "GOTO addlab" (address label) if the ISR is to be placed elsewhere in program memory. If interrupts are to be used, a GOTO must also be used at the reset vector address, 000, to redirect the program counter to the start of the main program at a higher

memory address, because the ISR (or GOTO addlab) will occupy address 004. The ISR must be created and allocated to address 004 (ORG 004) as part of the program source code.

The ISR must be terminated with the instruction RETFIE (return from interrupt). This causes the original program address to be pulled from the stack, and program execution resumes at the instruction following the one which was interrupted. It may be necessary to save other registers as part of the ISR, so that they can be restored after the interrupt. This is called "context saving." This is illustrated in INT1 program below by saving and restoring the contents of Port B data register as part of the ISR.

6.3.3 INT1 Interrupt Program

A demonstration program, Program 6.2, illustrates the use of interrupts. The BIN hardware must be modified to run this program, with the push buttons connected to RB0 and RA4. This is necessary because only Port B pins can be used for external interrupts.

The program outputs the same binary count to Port B, as seen in the BINx programs, to represent its normal activity. This process is then interrupted by RB0 being pulsed manually. The interrupt service routine causes all the outputs to be switched on, and then waits for the button on RA4 to be pressed. The routine then terminates, restores the value in Port B data register and returns to the main program at the original point. The program structure and sequence can be represented by the flowcharts in Fig. 6.4.

```
;*********************************************************
;       INT1.ASM        M. Bates       12/6/66     Ver 2.1
; *********************************************************
;       Minimal program to demonstrate interrupts.
;
;       An output binary count to LEDs on PortB, bits 1-7
;       is interrupted by an active low input at RB0/INT.
;       The Interrupt Service Routine sets all outputs high,
;       and waits for RA4 to go low before returning to
;       the main program.
;       Connect push button inputs to RB0 and RA4
;
;
;       Processor:          PIC 16F84
;       Hardware:           PIC Modular Demo System
;                           (reset switch connected to RB0)
```

Program 6.2: INT1 Interrupt Program

```
;       Clock:              CR ~100kHz
;       Inputs:             Push Buttons
;                           RB0=1=Interrupt
;                           RA4 = 0 = Return from Interrupt
;       Outputs:            RB1 - RB7: LEDs (active high)
;
;       WDTimer:            Disabled
;       PUTimer:            Enabled
;       Interrupts:         RB0 interrupt enabled
;       Code Protect:       Disabled
;
;       Subroutines:        DELAY
;       Parameters:         None
;
;*****************************************************************

; Register Label Equates.....................................

PORTA   EQU     05          ; Port A Data Register
PORTB   EQU     06          ; Port B Data Register
INTCON  EQU     0B          ; Interrupt Control Register
timer   EQU     0C          ; GPR1 = delay counter
tempb   EQU     0D          ; GPR2 = Output temp. store

; Input Bit Label Equates ...................................

intin   EQU     0           ; Interrupt input = RB0
resin   EQU     4           ; Restart input = RA4
INTF    EQU     1           ; RB0 Interrupt Flag

;
; ***************************************************************

; Set program origin for Power On Reset.....................

        org     000         ; Program start address
        GOTO    setup       ; Jump to main program start

; Interrupt Service Routine at address 004..................

        org     004         ; ISR start address

        MOVF    PORTB,W     ; Save current output value
        MOVWF   tempb       ; in temporary register
```

Program 6.2: Continued

```
        MOVLW   b'11111111'   ; Switch LEDs 1-7 on
        MOVWF   PORTB

wait    BTFSC   PORTA,resin   ; Wait for restart input
        GOTO    wait          ; to go low

        MOVF    tempb,w       ; Restore previous output
        MOVWF   PORTB         ; at the LEDs
        BCF     INTCON,INTF   ; Clear RB0 interrupt flag
        RETFIE                ; Return from interrupt

; DELAY subroutine.....................................

delay   MOVLW   0xFF          ; Delay count literal is
        MOVWF   timer         ; loaded into spare register
down    DECFSZ  timer         ; Decrement timer register
        GOTO    down          ; and repeat until zero then
        RETURN                ; return to main program


; Main Program ********************************************

; Initialize Port B (Port A defaults to inputs).................

setup   MOVLW   b'00000001'   ; Set data direction bits
        TRIS    PORTB         ; and load TRISB

        MOVLW   b'10010000'   ; Enable RB0 interrupt in
        MOVWF   INTCON        ; Interrupt Control Register

; Main output loop ....................................

count   INCF    PORTB         ; Increment LED display
        CALL    delay         ; Execute delay subroutine
        GOTO    count         ; Repeat main loop always

        END                   ; Terminate source code

; ********************************************************
```

Program 6.2: Continued

The program is in three parts: the main sequence which runs the output count, the delay subroutine which controls the speed of the output count, and the interrupt service routine. The delay process in the main program is implemented as a subroutine, and expanded in a separate flowchart. The ISR must be shown as a separate chart because it can run at any time within the

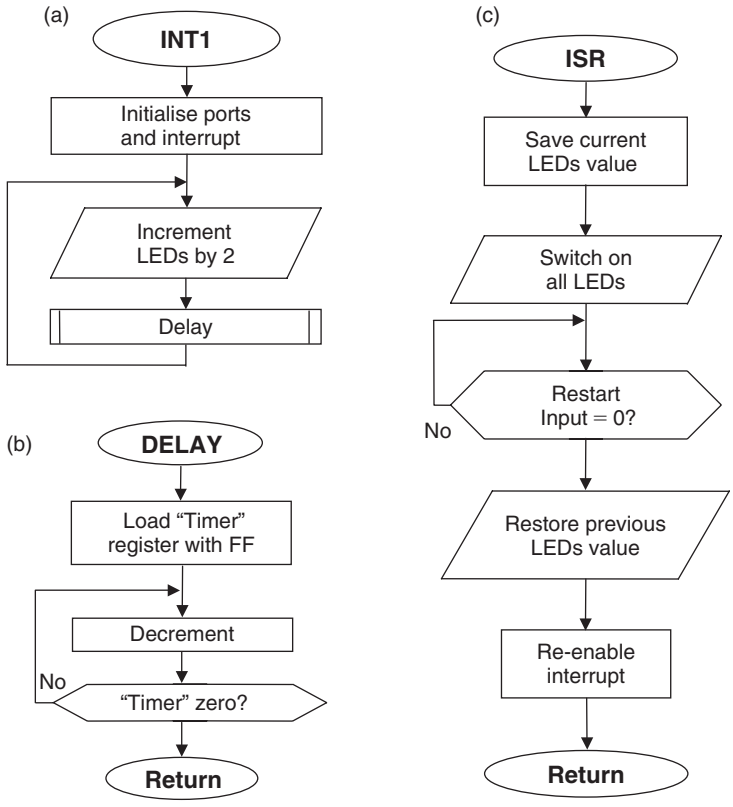


Figure 6.4: INT1 Interrupt Program Flowcharts: (a) Main Sequence (b) Delay Routine, and (c) Interrupt Service Routine

program sequence. In this particular program, most of the time is spent executing the software delay, so this is the process that is most likely to be interrupted.

The interrupt routine is placed at address 004. The instruction 'GOTO setup' jumps over it at run time to the initialization process at the start of the main program. The interrupt and delay routines must be assembled before the main program, because they contain the subroutine start address labels referred to in the main program, so they are entered first in the source code. The last instruction in the ISR must be RETFIE. This instruction pulls the interrupt return address from the stack and places it back in the program counter, where it was stored at the time of the interrupt call.

To illustrate context saving, the state of the LEDs is saved in register "tempb" at the beginning of the interrupt, because Port B is going to be overwritten with "FF" to switch on all the LEDs. Port B is then restored after the program has been restarted. Note that writing a "1" to the input bit has no effect. During the ISR execution, the stack will hold both the ISR return address and the subroutine return address.

6.3.4 More Interrupts

In larger PIC chips, additional interrupt sources are typically present, such as analog inputs, serial ports and additional timers. These all have to be setup and controlled via additional special function registers. As an example, the 16F877 has 14 interrupt sources, but still has only one interrupt vector address, 0004, to handle them. The interrupt bits must be checked in software to see which is active before calling the appropriate ISR. Also, the stack can still only hold eight return addresses, despite the program memory being 8k. The limit of eight levels of subroutine or interrupt can easily be exceeded if the program is too highly structured, so this must be borne in mind when planning the program implementation.

6.4 More Register Operations

6.4.1 Data Destination W

The default destination for operations that generate a result is the file register specified in the instruction. For example:

6.4.1.1 INCF spare

Increments the register labeled "spare," with the result being left in the register. The above syntax generates a message when the program is assembled to remind the user that the "default" destination is being used. This is because the full syntax is

```
INCF    spare,1
```

where "1" indicates the file register itself as the destination. If the result of the operation were required in the working register W, it could be moved using a second instruction

```
MOVF    spare,W
```

However, the whole operation can be done in one instruction by specifying the destination as W as follows:

```
INCF    spare,0
```

or

```
INCF    spare,W
```

The label W is automatically given the value 0 by the assembler. The result of the operation is stored in W, while the original value is left unchanged in the file register. All the register arithmetic and logical byte operations have this option, except CLRF (Clear File Register) and CLRW (Clear Working Register) which are by definition register specific, MOVWF and NOP (No operation). This option offers significant savings in execution time and memory requirements, which in PIC applications may be quite significant, and compensates for the lack of instructions to make direct moves between file registers.

6.4.2 Register Bank Select

The 16F84 file register set (Fig. 6.5) is organized in two banks, with the most commonly used registers in the default bank 0. Some of the control registers, such as the port data direction registers, TRISA and TRISB, and the OPTION register, are mapped into bank 1. Many of the SFRs can be accessed in either bank. Others have special access instructions, namely TRIS to write the Port A and B data direction registers, and OPTION which is used to set up the real time clock counter.

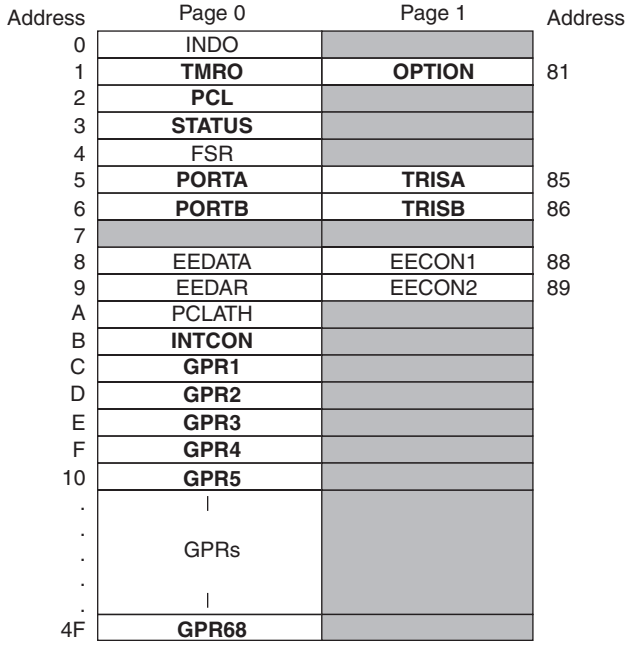


Figure 6.5: PIC 16F84 File Register Set

The manufacturer recommends using bank selection to access all these registers, and the instruction set warns that the instructions TRIS and OPTION may not be supported by future assemblers. Bank 0 is enabled by default, and bank 1 registers OPTION, TRISA, TRISB, EECON1 and EECON2 can be selected by setting bit 5, RP0, in the STATUS register, prior to accessing the corresponding register number. The alternative method to set Port B to output is therefore as follows:

```
STATUS   EQU     03              ; label for status register
TRISB    EQU     86              ; label for data direction register
         BSF     STATUS,5        ; select bank 1
         CLRW                    ; load W with data direction code
         MOVWF   TRISB           ; set Port B as outputs
         BCF     STATUS,5        ; re-select bank 0
```

It is a good idea to reselect bank 0 immediately, as this is the most commonly used. However, if further bank 1 access is required, leave this step until later. Once a bank has been selected, it remains accessible until de-selected. Larger PIC chips which have more special function registers and provide more data registers have four register banks, requiring two bits for bank selection, status bits 5 and 6.

An alternative is to use the pseudo-operation "BANKSEL" as follows:

```
BANKSEL   TRISB      ; select bank containing TRISB, bank 1
CLRW                 ; load code for all outputs
MOVWF     TRISB      ; set Port B as outputs
BANKSEL   PORTB      ; re-select bank containing PORTB, bank 0
```

BANKSEL selects the bank that the specified register is in, so, to change banks, any register in the required bank will do. The temperature control program (Program 8.1) uses this technique, and it is recommended as the best option for accessing registers not in bank 0.

Pseudo-operations, or special instructions, are explained in Section 6.8.

6.4.3 File Register Indirect Addressing

File register 04 is the File Select Register (FSR). It is used for indirect or indexed addressing of the other file registers, particularly the GPRs. If a file register address (00–4F) is loaded into FSR, the contents of that file register can be read or written through file register 00, the Indirect File Register (INDF). This method can be used for accessing a set of data RAM locations, by reading or writing the data via INDF, and selecting the next file register by incrementing FSR (see Fig. 6.5). This indexed, indirect file register addressing is particularly useful for storing a set of data which has been read in at a port, in, for example, a data logging application. An output data table of predefined values, such as seven-segment display codes, can use the program data table method described in Section 6.6.

The demonstration Program 6.3 loads a set of file registers, 20–2F, with dummy data (AA), using FSR as the index register. Here, FSR operates as a pointer to a block of locations and is incremented between each read or write operation. Notice that the data actually has to be moved into INDF each time.

6.4.4 EEPROM Memory

Many PIC chips have a block of electrically erasable read only memory (EEPROM) which operates as non-volatile read and write memory; the data written to this block is retained when the power is off. This is useful, for example, for security applications such as an electronic lock, where the correct combination can be stored and changed as required. Access to EEPROM is illustrated in Fig. 6.6.

The four registers used to access the memory are EEDATA, EEADR, EECON1 and EECON2. The data to be stored is placed in EEDATA, and the address at which it is to be written

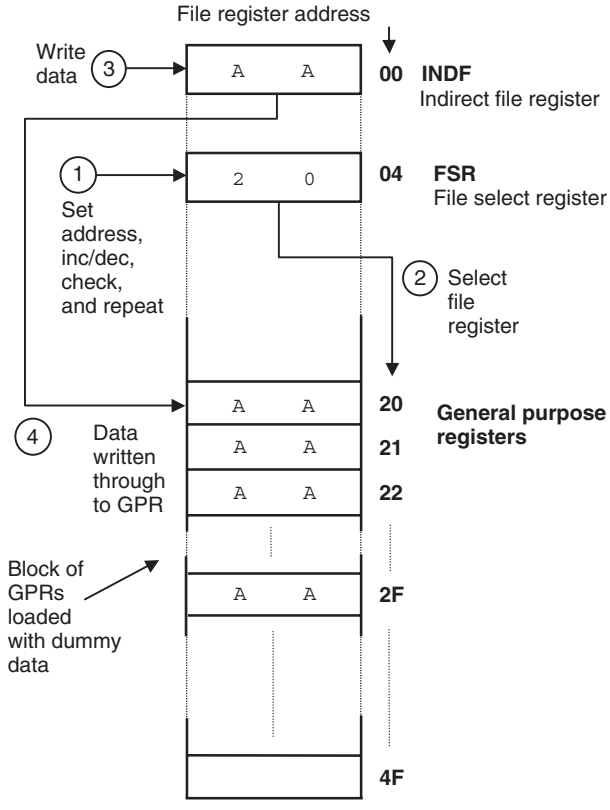


Figure 6.6: Indirect File Register Addressing

(00–3 F) in EEADR. Bank 1 must then be selected, and a read or write sequence included in the program as specified in the data sheet, Section 3. The complex write sequence is designed to reduce the possibility of an accidental write to EEPROM, whereby valuable data is lost. Reading the EEPROM is more straightforward. The LOCK application program in Appendix B includes examples of the code sequences required to read and write EEPROM. Other devices use a different technique to access the EEPROM; the 8-pin PIC 12CE518/6 devices use serial access via the unused bits of the port register. The individual device data sheet must therefore be studied carefully to use this feature.

6.4.5 Program Counter High Register, PCLATH

The 16F84 has 1 k of program memory (000–3FF), requiring a 10-bit address; the 8-bit PCL (program counter low byte) can only select one of 256 addresses. The 1 k of program memory is therefore divided into four 256 word blocks (pages), one of which is selected with 2 extra bits in the PCLATH (program counter latch high) register. The PCL provides the address within each page of memory and is fully readable and writable. When a program jump is executed, PCL and PCLATH are modified automatically, that is, CALL and GOTO use a full 10-bit

operand for jumps, so do not require any special manipulation of the address for jumping across page boundaries. However, if PCL is modified by a direct write under program control, PCLATH bits 0 and 1 may need to be manipulated to cross page boundaries successfully.

In other PIC devices, there may be other limitations to program branching operations. For example, CALL instructions in the 12C5XX group are limited to the first 256 locations of the program, even though the overall memory may be up to 1 k. Check the data sheet carefully to avoid problems with this limitation.

```
;  index.asm       M Bates       26-10-03  ;
;  ..................................................
;  Demonstrates indexed indirect addressing by
;  writing a dummy data table to GPRs 20-2F
;  ..................................................

    PROCESSOR 16F84          ; select processor

FSR     EQU     04          ; File Select Register
INDF    EQU     00          ; Indirect File Register


        MOVLW   020         ; First GPR = 20h
        MOVWF   FSR         ; to FSR

        MOVLW   0AA         ; Dummy data
next    MOVWF   INDF        ; to INDF and GPRxx

        INCF    FSR         ; Increment GPR Pointer
        BTFSS   FSR,4       ; Test for GPR = 30h
        GOTO    next        ; Write next GPR

        SLEEP               ; Stop when GPR = 30h

        END                 ; of source code
```

Program 6.3: Indexed File Register Addressing

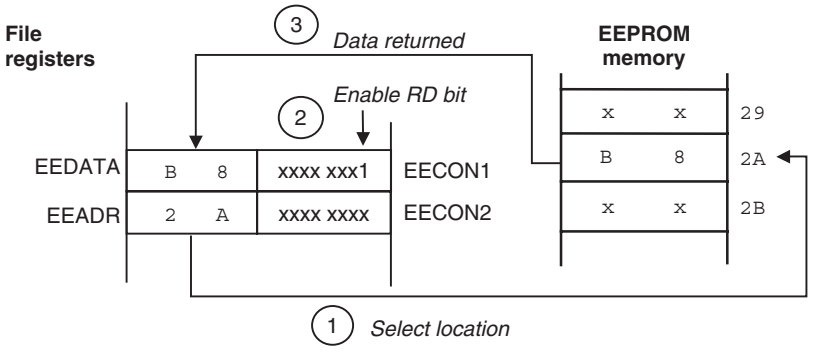


Figure 6.7: EEPROM Read Operation

6.5 Special Features

PIC chips have a number of special features that enhance its flexibility and range of applications. Different oscillator types can be used, timers enabled to ensure reliable program start up and recovery, and in-circuit programming and code protection are available.

6.5.1 Oscillator Type

PIC chips can be operated with an external RC network, a crystal oscillator and an externally or internally generated clock signal. Typical oscillator circuits are illustrated in Fig. 6.8.

For applications where the precise timing of the program is not important, an inexpensive RC clock circuit (Fig. 6.8(a)) can be used. This requires only a resistor and capacitor connected

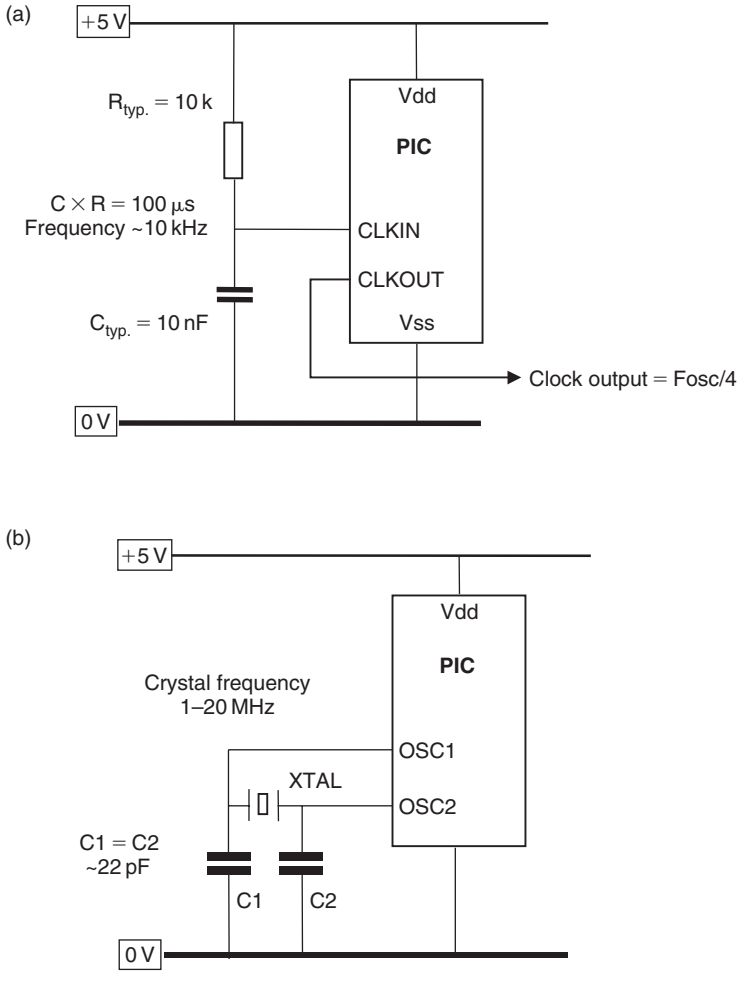


Figure 6.8: PIC Clock Circuits: (a) RC Oscillator and (b) Crystal Oscillator

as shown to the CLKIN pin of the chip. If a variable resistor is used, as in the BIN hardware, the clock rate can be adjusted, within limits, and therefore all output signal frequencies can be changed simultaneously (for example, the outputs from the program BIN1). The clock and output frequency can thus be "trimmed" to a required value. On the other hand, the clock signal will not be very accurate or stable.

The crystal is slightly more expensive, but is far more precise than the RC clock. In the XT oscillator circuit (Fig. 6.8(b)) the crystal resonates at a fixed frequency, with an accuracy of around 50 ppm (parts per million), or 0.005%. This will allow the hardware timer to measure exact intervals and to generate accurate output signals. The overall execution time of the program blocks can also be predicted; this can be done by calculation, or, more readily, by use of the stopwatch in MPLAB.

If the PIC chip is part of a larger system, or one with more than one processor, a system clock signal generated by a master oscillator can be input at CLKIN. One of the crystal options must then be selected. The clock type must be selected when programming the chip, to match the target system hardware design. There are three types of crystal that can be used: standard (XT), low power (LS) or high speed (HS). XT mode should be used for clock speeds up to 4 MHz, and HS used up to 20 MHz.

In order to minimize the number of external components required, some PIC chips now have an on-board oscillator option, which provides a 4 MHz clock and 1s instruction cycle. Because this is not a precise oscillator, it is tested in production and a calibration value supplied preprogrammed in the first program memory location. This value must then be loaded into the oscillator calibration register OSCCAL. Even so, the accuracy achieved is specified as only about 5% (3.8–4.2 MHz).

6.5.2 Power-on Timers

When a power supply is switched on, the voltage and current initially rise in an unpredictable way, depending on the design of the supply and the circuits connected to it. If the processor program tries to start immediately, before the supply had settled down, it may not start correctly. In a conventional microprocessor, an external circuit is typically connected to the CPU reset input, which provides a delay between the power being switched on and the processor starting.

The PIC has the required power-on timers built in to the chip. The reset input can therefore simply be connected to the positive supply (+5 V) for many applications, as is the case in the examples in this book. When the PIC is powered up, a power-on reset pulse is generated when the supply voltage detected at Vdd rises through about 1.5 V. This starts a power-up timer which times out after 72 ms, which in turn triggers an oscillator start-up timer, which delays

for another 1024 clock cycles, to allow the internal clock to stabilize. An internal reset is then generated, and the program starts executing. The power-up timer should normally be enabled when programming the chip, as the resulting delay on start up will normally be insignificant.

6.5.3 Watchdog Timer (WDT)

This is an internal independent timer which, by default, forces the PIC to automatically restart after a fixed period (about 18 ms). The idea is to allow the processor to escape from an endless loop or other error condition, without having to be reset manually. This facility would be used by more advanced programs, so our main concern here is to prevent watchdog timeout occurring when not required, because it will disrupt the sequence and timing of our programs.

The WDT can be disabled by selecting the appropriate configuration setting during program downloading, and this is the usual option for simple programs. If the watchdog is to be employed, the WDT must be regularly reset within the program loop using the instruction CLRWDT. If this happens at least every, say, 1 ms (1000 instructions at 4 MHz), the WDT auto-reset can be prevented. If a program misbehaves in the simulator, check that WDT is disabled.

6.5.4 Sleep Mode

The instruction SLEEP causes normal operation to be suspended and the clock oscillator to be switched off. Power consumption is minimized in this state, which is useful for battery-powered applications. The PIC is woken up by a reset or interrupt; for example, when a key connected to Port B is pressed.

The SLEEP instruction is used (see Program 6.3) if the program is not required to loop continuously. If the program execution is allowed to run on into unprogrammed locations, there is a problem. The bits in the empty memory locations after the last instruction code default high. In the ’84, this is in fact a valid PIC instruction, ADDLW FF, which means add literal “FF” to W, so this instruction will be repeated throughout the unused locations. The program counter will roll over to zero after executing these meaningless instructions up to address 3FF, and the program at 000 will be restarted, so the program will loop by default. It is therefore a sensible precaution to terminate the program with a SLEEP instruction if does not run in continuous loop. If SLEEP is used to stop the program at the end, a power-on reset, external reset or an interrupt can then restart the processor.

6.5.5 In-Circuit Programming and Debugging

In-circuit programming allows the chip to be programmed without being removed from the circuit, which avoids possible mechanical (broken/bent legs) and electrical (static) damage. The programming module is connected to the serial port of the host PC and to the chip via

two port pins (RB6 and RB7 in the 16XXX) via a suitable connector (Fig. 6.8). The program can then be downloaded in the usual way, in serial form. Note, however, that the circuit must be designed so that the normal operational connections to the port pins do not interfere with the downloading process. If in doubt, leave these pins exclusively for programming. When programming is complete, the connector can be removed, the board set to run and the port pins used for their normal function.

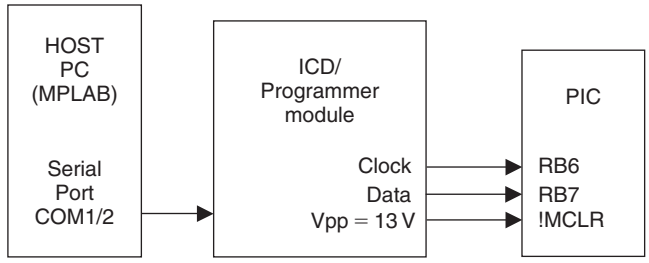


Figure 6.9: Serial Programming and ICD Connections

PIC chips are now designed to allow this programming link to be used in a cheap but effective debugging system. If ICD is supported by the target hardware, an ICD module acts as programmer and debugging interface. The same MPLAB simulator tools can be used to test the program as it runs in the actual chip, rather than in the purely software model, with the real hardware acting as inputs and outputs. This allows the hardware to be verified, and timing critical operations to be tested easily and reliably. All the usual techniques are available: single stepping, breakpoints, register monitoring and so on. The finished program can be tested at full speed in the actual hardware, and any final bugs removed. Previously, an expensive in-circuit emulator would be needed for this type of testing. It is anticipated that this feature will be extended to more PIC devices, as it is a valuable low-cost tool for PIC program development.

By incorporating the programming interface into the target hardware, it is also possible for microcontrollers to be reprogrammed remotely after final installation. If a suitable communication link is available, a new control program can be downloaded while the target processor remains at its remote site. This is a great advantage in, for example, distributed sensing and monitoring applications, where a site visit would be expensive or time-consuming. Obviously, the new program would need to be fully tested on an identical local system before being downloaded to the remote system.

6.5.6 Code Protection

In commercial applications, the PIC program designer does not want the software supplied with a product to be copied by a market competitor. The “Code Protect” fuse, selected during

programming, is designed to prevent unauthorized copying. The chip can also be given a unique identification code during programming, if required. For our purposes, the code protection should not be enabled, as the program cannot then be read back for verification.

6.5.7 Configuration Word

The oscillator selection bits (2), watchdog timer, power-up timer and code protection are all selected by setting the bits of a configuration word, located at a special address which is only accessible when the chip is being programmed. These bits can be set via the programming dialogue in MPLAB. Alternatively, the configuration options can be set by including an assembler directive in the source code.

6.6 Program Data Table

A program may be required to output a set of predefined data bytes, for example, the codes to light up a seven-segment display with the correct pattern for each display digit, as in Program 7.2. The data set can be written into the program as a table within a subroutine, and the data list accessed using CALL and RETLW. To fetch the table value required, the position in the table is placed in W. "0" will access the first item, "1" the second and so on. At the top of the subroutine, ADDWF PCL is used to add the table pointer value to the program counter register so that the execution point jumps to the required item in the list. RETLW is then used to return the table value in W, and it can then be moved to the required file register.

Program 6.4, TAB1, shows how such a table may be used to generate a sequence at the LEDs in our BIN demonstration hardware. In this case, it is a bar graph display which lights the LEDs from one end, using the binary sequence 0, 1, 3, 7, 15, 31, 63, 127, 255.

Spare registers labeled "timer" and "point" are used. Port B is set as outputs and subroutines defined for a delay and to provide a table of output codes. In the main loop, the table pointer register "point" is initially cleared, and will then be incremented from zero to 6 as each code is output. The value of the pointer is checked each time round the loop to see if it is 6 yet. When 6 is reached, the program jumps back to "newbar" and the pointer reset to zero.

For each output, the pointer value (0–8) is placed in W and the "table" subroutine called. The first instruction "ADDWF PCL" adds the pointer value to the program counter. At the first call, this value is zero, so the next instruction "RETLW 000" is executed. The program returns to the main loop with the value 00 in W. This is output to the LEDs, the delay run, and the pointer value incremented. The new value is tested to see if it is 6 yet, and if not, the call is made to the table with the next value, 1, and so on to 8. Each time the pointer value is added to PCL, so that the program jumps to the second, then third, then fourth code and so on, until finally the ninth code, which is 0FF, is returned to the main output loop for display. After this,

the test of the pointer being equal to 6 succeeds, the jump back to "newbar" taken, and the process repeats. Note the use of "W" as the destination for the result of the subtract (SUBWF) instruction. This is necessary to avoid the pointer value being overwritten with the result of the subtraction.

```
;*********************************************************
;       TAB1.ASM        M. Bates      13/6/66      Ver 1.3
; *********************************************************
;
;       Output binary sequence gives a demonstration of a
;       bar graph display, using a program data table...
;
;       Processor:      PIC 16F84
;
;       Hardware:       PIC Demo System
;       Clock:          CR ≈10kHz (Cycle time ≈0.7s)
;       Inputs:         none
;       Outputs:        LEDs (active high)
;
;       WDTimer:        Disable
;       PUTimer:        Enable
;       Code Protect:   Disable
;
;       Interrupts:     Disabled
;       Subroutines:    'delay' (no arguments)
;                       'table' (argument 'point')

; *********************************************************

; Register Label Equates.................................

PCL     EQU     02          ; Program Counter Low Register
PORTB   EQU     06          ; Port B Data Register
timer   EQU     0C          ; GPR1 used as delay counter
point   EQU     0D          ; GPR2 used as table pointer

; *********************************************************

        ORG     000
        GOTO    start       ; Jump to start of main prog

; Define DELAY subroutine................................
```

Program 6.4: TABI Table Program

```
delay   MOVLW   0xFF        ; Delay count literal
        MOVWF   timer       ; loaded into spare register

down    DECFSZ  timer       ; Decrement timer register
        GOTO    down        ; and repeat until zero
        RETURN              ; then return to main program

; Define Table of Output Codes ..........................

table   ADDWF   PCL         ; Add pointer to PCL
        RETLW   000         ; 0 LEDS on
        RETLW   001         ; 1 LEDS on
        RETLW   003         ; 2 LEDS on
        RETLW   007         ; 3 LEDS on
        RETLW   00F         ; 4 LEDS on
        RETLW   01F         ; 5 LEDS on
        RETLW   03F         ; 6 LEDS on
        RETLW   07F         ; 7 LEDS on
        RETLW   0FF         ; 8 LEDS on

;       Initialise Port B (Port A defaults to inputs).........

start   MOVLW b'00000000'   ; Set Port B Data Direction Code
        TRIS    PORTB       ; and load into TRISB

; Main loop ..............................................

newbar  CLRF    point       ; Reset pointer to start of table

nexton  MOVLW   006         ; Check if all outputs done yet
        SUBWF   point,W     ; (note: destination W)
        BTFSC   3,2         ; and start a new bar
        GOTO    newbar      ; if true...

        MOVF    point,W     ; Set pointer to
        CALL    table       ; access table...
        MOVWF   PORTB       ; and output to LEDs

        CALL    delay       ; wait a while...

        INCF    point       ; Point to next table value
        GOTO    nexton      ; and repeat...

; End of main loop .....................................
        END                 ; Terminate source code
```

Program 6.4: Continued

6.7 Assembler Directives

Assembler directives are commands inserted in PIC source code that control the operation of the assembler. They are not part of the program itself and are not converted into machine code. Many assembler directives will only be used when a good knowledge of the programming language has been achieved, so we will refer to a small number of selected examples at this stage. The use of some of these is illustrated in Program 6.5, ASD1. The assembler directives are placed in the second column, with the instruction mnemonics. We have already met some of the most commonly used directives, but END is the only one

```
;*************************************************************
;    ASD1.ASM          M. Bates        17/12/03     Ver 1.1
;*************************************************************
;    Assembler directives, a macro and a pseudo-operation are
;    illustrated in this counting program ...
; ************************************************************

;    Directive sets processor type:
     PROCESSOR 16F84

;    Set configuration fuses:
     __CONFIG B'1111111111110011'
;    Code protection off, PuT on, WDT off, RC clock

;    SFR equates are inserted from disk file:
     INCLUDE "C:\PIC\REG84.EQU"
;    Constant values can be predefined by directive:
     CONSTANT maxdel = 0xFF, dircb = b'00000000'

timer    EQU     0C                  ; delay counter register
; Define DELAY macro ****************************************

DELAY    MACRO

         MOVLW   maxdel         ; Delay count literal
         MOVWF   timer          ; loaded into spare register

down     DECF    timer          ; Decrement spare register
         BNZ     down           ; Pseudo-Operation:
                                ; Branch If Not Zero
         ENDM

;*************************************************************
```

Program 6.5: ASD1 Assembler Directives Program

```
;       Initialize Port B (Port A defaults to inputs)

        MOVLW   dircb           ; Port B Data Direction Code
        TRIS    PORTB           ; Load the DDR code into F86

;       Start main loop ..................................

        CLRF    PORTB           ; Clear Port B Data & restart
again   INCF    PORTB           ; Increment count at Port B
        DELAY                   ; Insert DELAY macro
        GOTO    again           ; Repeat main loop always


        END                     ; Terminate source code
```

Program 6.5: Continued

which is essential, all the others are simply available to make the programming process more efficient. For definitive information refer to the documentation and help files supplied with your current assembler version.

6.7.1 Control Directives Processor

Specifies the PIC processor for which the program has been designed, and allows the assembler to check that the syntax is correct for that processor. The simulator also uses this to automatically select the right processor. The processor can be selected in the assembler command line; if so, this supersedes the source code directive.

6.7.1.1 Config

The configuration directive allows the configuration bits to be specified in the source code, so that they do not have to be set up each time when downloading. This is obviously useful if the program has to be downloaded several times before completion of debugging. The significance of each bit is shown in the data sheet, Section 6.1. A 16-bit word is loaded into the configuration register by this directive. Bits 0 and 1 set the clock type (11 = RC, 01 = XT), bit 2 disables the watchdog timer if cleared and bit 3 enables the power-up timer if cleared. All the other bits are set to 1 to disable code protection. The double underscore that starts the directive indicates an operation on the MCU registers.

6.7.1.2 Org

Sets the code "origin," meaning the address which will be allocated to the first instruction following this directive. We have already seen (Program 6.2) how it is necessary to set the origin of the interrupt service routine as 004. The default origin is 000, the first program

memory location, so if not specified, the program will be placed here. This is the reset address where the processor always starts on power up or reset. If using interrupts, an unconditional jump "GOTO addlab" should be used as the first instruction at the reset address 000. This will jump over the ISR (or a jump to it) placed at address 004. The main program can then be placed at a higher address using the ORG directive.

6.7.1.3 End

Informs the assembler that the end of the source code has been reached. This is the one directive that must be present.

6.7.2 Conditional Directives

These directives allow selective assembly of source code blocks. That is, sections of code can be omitted during assembly, or repeated, by use of high level language type statements such as IF… ELSE… ENDIF. Assembler "variables" are used to define the conditions for assembly.

6.7.3 Listing Directives

6.7.3.1 List

This directive has a number of options that allow the format and content of the list file to be modified, e.g., number of lines and columns per page, error levels reported, processor type and so on.

6.7.3.2 Page

Forces a page break when printing.

6.7.3.3 Title

Defines the program name printed in the list file header line, if you want it to be different from the source code file name (see also SUBTITL).

6.7.4 Data Directives

6.7.4.1 Equ

EQU is probably the second most commonly used directive, because it allows literal and register labels to be defined, and we have already used it routinely. It assigns a label to any numerical value (hex, binary, decimal or ASCII), and the assembler then replaces the label with the number. This allows recognizable labels to be used instead of numbers.

6.7.4.2 Include

This directs the assembler to include a block of source code from a named file on disk. If necessary, the full file path must be given. The text file is included as though it had been typed into the source code editor, so it must conform to the usual assembler syntax, but any program block, subroutine or macro could be included in the same way. This allows separate source code files to be included, and opens the way for the user to create libraries of reusable program modules. In the example ASD1, it is used to include a standard header file (REG84.EQU) which defines labels for all the special function registers in the PIC. Use of this option is recommended when the basics have been mastered; standard header files, which use labelling which is consistent with the SFR labels used in the register monitoring windows in MPLAB, are supplied with the development system files for all processors.

6.7.4.3 Data, Zero, Set, Res

Allow program constants and data blocks to be defined and memory allocated for specified purposes.

6.7.5 Macro Directives

6.7.5.1 Macro. . . . Endm

A macro is a block of source code that is inserted into the program when its name is used as an instruction. In ASD1, for example, DELAY is the name of the macro, and its insertion in the main program can be seen in the list file. Thus using a macro is equivalent to creating a new instruction from standard instructions, or an automatic copy and paste operation. The directive MACRO defines the start of the block (with a label), ENDM terminates it. It effectively allows you to create your own instruction mnemonics (see also LOCAL and EXITM).

6.8 Special Instructions

Special instructions are essentially macros that are predefined in the assembler. A typical example is shown in the program ASD1, "BNZ down," which stands for "Branch if Not Zero to label." It is replaced by the assembler with the instruction sequence Bit Test and Skip and GOTO:

```
BNZ down = BTFSS 3,2
GOTO down
```

These two instructions are inserted into the program in place of the special instruction. The zero flag (bit 2) in the status register (register 3) is tested, and the GOTO skipped if it is set as a result of the previous operation being zero. If the result was not zero, the GOTO is executed, and the program jumps to the address label specified. Special instructions are designed to

simplify operations using the carry or zero flag, and are equivalent to conditional branch instructions in complex instruction set processors. This type of instruction is included in the main instruction set of the more powerful 18XXX series of PICs.

6.9 Numerical Types

Literal values given in PIC source code can be written using different number systems. The default is hexadecimal, so if the type is not specified, the assembler will assume it is hex. However, it is very important to note that the assembler will still get confused between numbers and labels if the hex number starts with a letter (i.e., A, B, C, D, E, or F). The literal must start with a number, so use a leading zero at all times. Then 8-bit literals will be written as three digits, with the first always zero (000–0FF).

The numerical types supported by the MPASM assembler are:

- hexadecimal
- decimal
- binary
- octal
- ASCII

To specify a type, the initial letter of the type can be used with quotes, such as:

```
H'3F'
D'47'
B'10010011'
A'K'
```

Binary is useful for specifying register values that are bit-oriented, as is the case for many SFRs; the state of each bit can be clearly seen. In particular, we have used binary to define port data direction codes in our demonstration programs.

If an ASCII character is specified, the corresponding 8-bit code in the range 00–7F will be loaded representing the code for each character in the set (see Table 6.2). This option is used in sending data to alphanumeric liquid crystal displays, for example. The character itself may then be used in the program, and the assembler does the code conversion:

```
MOVLW    'Y'        ; Converted to binary 01011001
MOVWF    PortB      ; send to display
```

Note that the A for ASCII can be left out, and the character still be correctly recognized by the assembler.

Table 6.2: ASCII Character Set

	High Bits					
Low Bits	**0010**	**0011**	**0100**	**0101**	**0110**	**0111**
0000	Space	0	@	P	`	p
0001	!	1	A	Q	a	q
0010	″	2	B	R	b	r
0011	#	3	C	S	c	s
0100	$	4	D	T	d	t
0101	%	5	E	U	e	u
0110	&	6	F	V	f	v
0111	′	7	G	W	g	w
1000	(	8	H	X	h	x
1001	)	9	I	Y	i	y
1010	*	:	J	Z	j	z
1011	+	;	K	[	k	{
1100	′	<	L	\	l	|
1101	-	=	M	]	m	}
1110	.	>	N	^	n	~
1111	/	?	O	–	o	Del

6.10 Summary

- Each PIC instruction takes four clock periods to execute (instruction cycle time). Jumps take two instruction cycles. Block execution times can therefore be calculated.
- The hardware counter/timers can be used to count inputs or time intervals. Programmable prescalers extend the range of the counter. Timer overflow sets the time-out flag, which can be used to trigger an interrupt.
- Interrupts allow an internal or external event to change the program sequence, and force the execution of an ISR. There are multiple interrupt sources, but no interrupt priority.
- Register and memory bank selection are sometimes necessary; EEPROM is available for nonvolatile storage.
- The clock signal that drives the chip can be obtained from an RC or crystal circuit, or master system clock. Power-on timers, watchdog timer, sleep mode, in-circuit programming and code protection are available.

- Program data tables can be operated using CALL and RETLW, with an incrementing PCL offset.
- Assembler directives are instructions to the assembler which are not converted into machine code.
- Macros are user-defined instructions. Special instructions are predefined macros.
- Numerical types hex, decimal, binary, octal and ASCII character codes can be used in the source code.

CHAPTER 7

Prototype Hardware

We now come to the stage where we need to look at the techniques available for designing and building our PIC circuits. Circuit design, simulation and layout software has developed to the point where powerful packages are now available at a reasonable cost. Current software allows the circuit to be drawn, tested by simulation, and the circuit netlist (list of components and connections) produced. This is then imported into a PCB design package where the circuit is laid out on screen; this can be printed onto a masking sheet, or a file generated which can be used to automatically produce a PCB.

7.1 Hardware Design

Traditionally, circuits have been designed as sketches on paper and a final version produced by a draftsman. This relied heavily on the experience of the electronics engineer to be able to predict the circuit performance from theoretical knowledge and practical experience. Numerous prototypes would typically be needed to arrive at a working solution.

This process has, since the development of increasingly powerful desktop computers, been radically improved. The designer still has to come up with the original ideas, but the proposed circuits can now be quickly drawn and tested on-screen, and a working design produced without a pencil touching paper or any component being inserted in a prototype board. The design cycle is much faster and the time taken from design concept to market is a major competitive factor in a rapidly changing industry. Therefore, electronic computer aided design (ECAD) is now a vital tool for the electronics engineer, just as CAD has become for the mechanical engineer.

Thus, a circuit diagram can be drawn and converted into a PCB layout within a single software package. The circuit design can be tested for correct function by software simulation which incorporates mathematical models for the behavior of each component, and their interaction. Libraries of microprocessor and microcontroller models, as well as interactive on-screen components, are now included; a circuit can be drawn on-screen, the application program attached to the microcontroller and the program tested by operating the on-screen inputs, such as switches and/or a keypad, with the mouse. The results are then displayed on simulated displays (LED and LCD) or operate animated output devices such as relays and motors.

As an example, a simple circuit design created in the schematic capture package ISIS is shown in Fig. 7.1. It is an electronic dice board with a push button, seven-segment display and buzzer controlled by a PIC 16F84. It can be programmed to display a random number between one and six when the button is pressed.

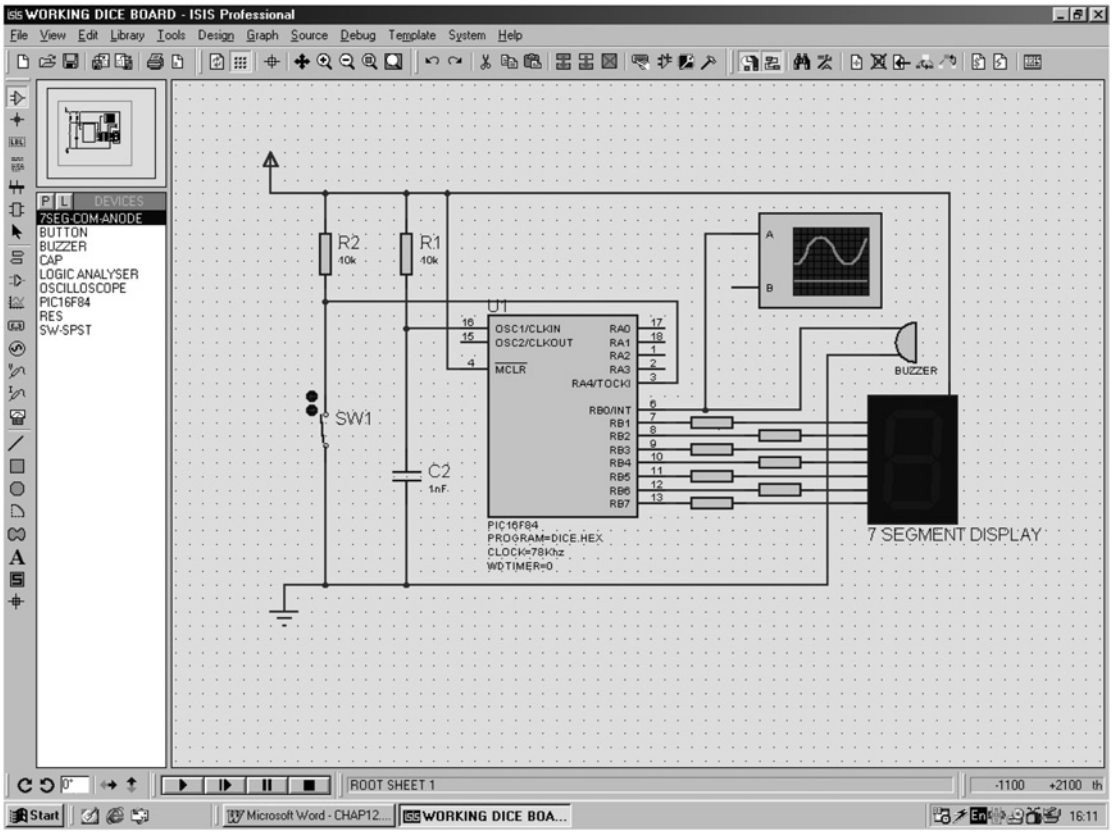


Figure 7.1: Circuit Design for DICE Board

When a suitable program is assigned to the PIC in the simulation, the circuit becomes interactive on screen. When the switch is operated, the display will operate in the same way as the real device. If the chip is programmed to make a sound, the waveform can be displayed on a virtual oscilloscope, and even be reproduced from the PC audio output, if the CPU is fast enough.

7.2 Hardware Construction

First, we will look briefly at some traditional techniques suitable for building one-off boards and prototypes. A general purpose demonstration board will be then designed and laid out in prototype form, and some programs provided to demonstrate its features and the related programming

principles. The DIZI board (display and buzzer with interrupt) has a seven-segment display and an audio output for some simple display and sound applications (see Appendix C).

7.2.1 Printed Circuit Board

The PCB is the standard method for making electronic circuits. In its basic form, it starts life as a sheet of insulating fiberglass board with a layer of copper on one side. The circuit connections are made by photographically transferring a pattern of conducting tracks and pads for the component connections onto the copper. For complex circuits, such as a PC motherboard, multilayer boards are used to accommodate the large number of parallel data connections.

The layout for a simple PIC circuit is shown in Fig. 7.2. It has a PIC 16F84, push button, seven-segment display, buzzer and associated components and can be programmed to operate as an electronic dice, generating a random number between one and six. The pattern of the copper tracks is shown, as well as the "silk screen" printing which will be applied to the component side of the board to show where to place the components.

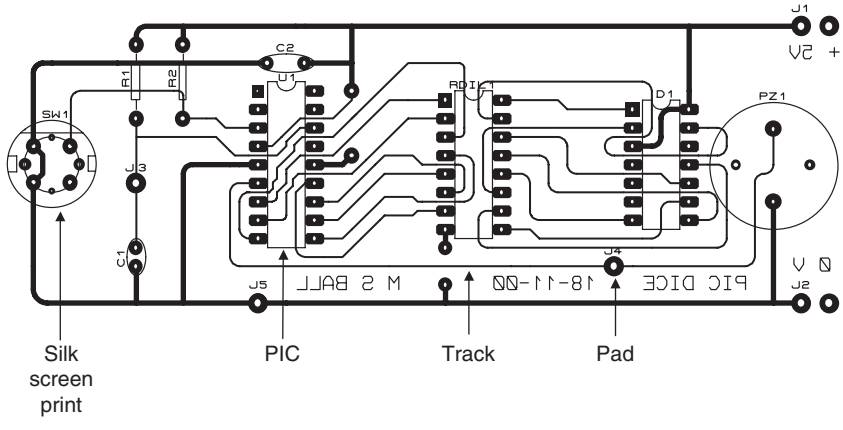


Figure 7.2: PIC Dice Board Layout

The layout is reversed as it will be printed onto a translucent mask, which is then used to create the pattern of connections on the copper side of the board. The copper layer is coated with a light-sensitive material, which is exposed to ultraviolet light through the mask. In the exposed areas of the board, the photo-sensitive material becomes soluble and is removed by a caustic solvent, exposing the copper below. This is then dissolved (etched) in an acid bath, leaving behind the copper layout where it was protected by the etch resisting layer. The components are then fitted to the silk screen side of the board, and the leads and pins soldered to the pads.

Once the layout has been designed, it can be used for batch production of the application hardware. Specialist companies are often used to manufacture the boards direct from the file output of the PCB design software. The final PCB-based product is shown in Fig. 7.3.

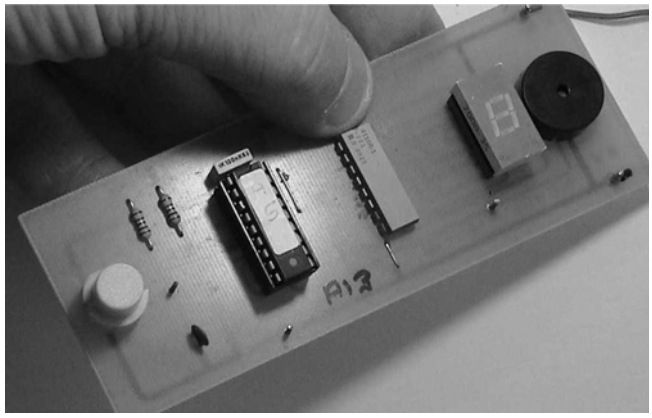


Figure 7.3: PIC 16F84 Dice Board

Even with the current ECAD packages, the PCB layout can take some time to create, and a considerable amount of skill is needed to use the software. Therefore, we will also look at how to prototype our hardware using traditional methods, which do not require specialist software or PCB fabrication equipment.

7.2.2 Breadboard

A common method of constructing prototype circuits uses breadboard (plugblock) to wire up prototype circuits. The connecting wires are pushed into an array of interconnected sockets, allowing circuits to be quickly built and modified. A typical circuit, which will be discussed later, is illustrated in Fig. 7.9.

The breadboard module has sets of terminals laid out on a 0.1″ grid which will accept the manual insertion of component leads and insulated tinned copper wire (TCW) links. It has rows of contacts interconnected in groups placed either side of the centerline of the board, where the ICs are inserted, giving typically four contacts on each IC pin. At each side of the board, there are longitudinal rows of common contacts which are normally used for the power supplies. Some types of breadboard can be supplied in blocks that plug together to accommodate larger circuits, or are mounted on a base with built-in power supplies.

The layout for a simple circuit is shown in Fig. 7.4, with a PIC 16F84 driving an LED at RB0 via a current-limiting resistor. The only other components required are a capacitor and a resistor to form the clock circuit, but we must not forget to connect the !MCLR (master clear) pin to the positive supply, or the chip will not run. The chip could now be programmed to flash the output at a specified rate. To connect up the circuit, we will need to refer to the chip pinout, which is given in Fig. 7.5.

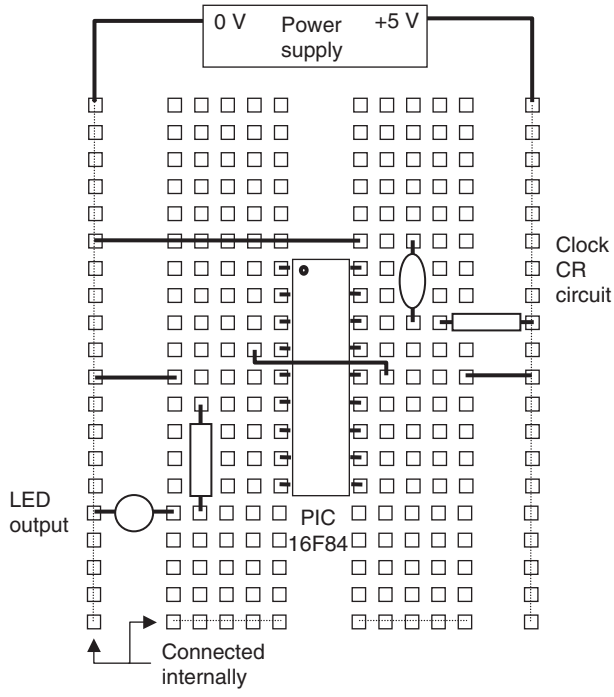


Figure 7.4: Breadboard Layout

	Function	*Label*	*Pin*	*Pin*	*Label*	*Function*	
I/O	Port A bit 2	RA2	1	18	RA1	Port A bit 1	I/O
I/O	Port A bit 3	RA3	2	17	RA0	Port A bit 0	I/O
I/O	RA4 or timer input	RA4/T0CKI	3	16	OSC1/CLKIN	Crystal or RC oscillator	In
In	Master clear (Reset)	!MCLR	4	15	OSC2/CLKOUT	Crystal circuit (if used)	Out
In	Supply 0 V	VSS	5	14	VDD	Supply +5 V	In
I/O	Port B bit 0 + interrupt	RB0/INT	6	13	RB7	Port B bit 7 (+interrupt)	I/O
I/O	Port B bit 1	RB1	7	12	RB6	Port B bit 6 (+interrupt)	I/O
I/O	Port B bit 2	RB2	8	11	RB5	Port B bit 5 (+interrupt)	I/O
I/O	Port B bit 3	RB3	9	10	RB4	Port B bit 4 (+interrupt)	I/O

Figure 7.5: 16F84 Pinout

Breadboard circuits can be built quickly, with no special tools required. However, the connections are relatively unreliable, so bad connections are likely in more complicated circuits. Therefore, method of producing prototype circuits with more reliable soldered connections is useful.

7.2.3 Stripboard

Stripboard is a prototyping method which requires no special tools or chemical processing. The components are connected via copper tracks laid down in strips on a 0.1″ grid of pin holes in an insulating board. The components are soldered in place and the circuit completed using wire links placed on the component side and soldered to the tracks on the copper side. The tracks must then be cut where the same strip is used for separate connections in the circuit. The components are generally placed across the tracks, so that each pin connects with a separate track. The tracks must be cut between opposite DIL chip pins, and other required positions, using a hand drill. An example is shown in Fig. 7.11.

Care is required to avoid dry joints (too little solder) or short circuits between tracks due to solder splashes and whiskers (too much solder!). A manual drawing may be used to draft the layout, if necessary, but a reasonably experienced constructor can build the circuit directly onto the board, with maybe some additional wastage of board area. It is also possible to work out the layout using a simple drawing package, or the drawing tools in a word processor.

Figure 7.6 shows how the simple PIC circuit can be laid out for construction on stripboard using general purpose drawing tools, such as those provided with Word. In the wordprocessor, the drawing toolbar needs to be switched on, and page layout view selected. In the "Draw" menu, the grid should be switched on and set it to 0.1″; this allows layouts to be drawn actual size, since this is the spacing between standard in-line pins. The circuit can then be drawn using suitable line styles, text boxes and so on. When finished, use the Select Objects tool to

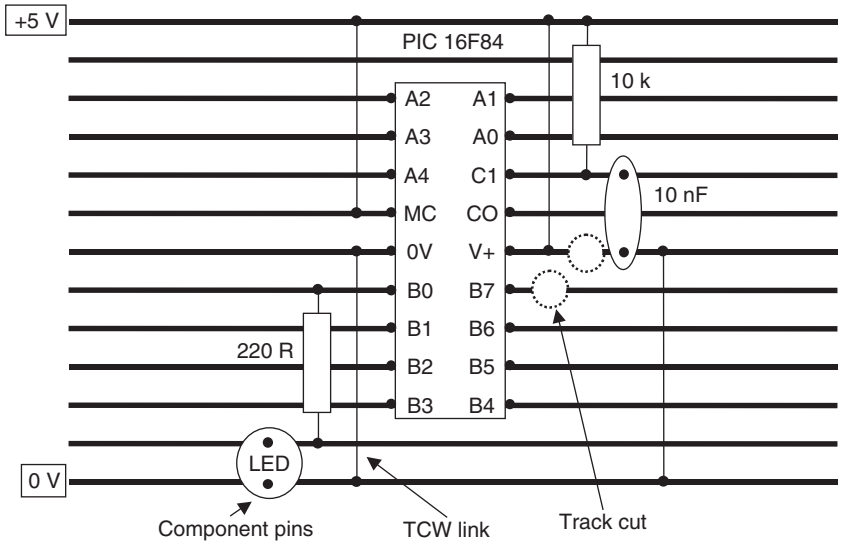


Figure 7.6: Stripboard Connections

select the whole drawing and Group it in the Draw menu. This prevents text cursor movement from disrupting the drawing, and the whole diagram can be repositioned on the page if required.

7.3 Demo Board

A circuit will now be designed, and a set of programs outlined, to illustrate the hardware design process and programming principles discussed in previous chapters. The DIZI board will allow the user to experiment with the various features of the PIC hardware and programming techniques within a single hardware module.

7.3.1 Hardware Specification

The microcontroller demonstration board will be suitable for demonstrating a wide range of processes incorporating display, audio, counting, timing and interrupt operations. The board will have a single digit seven-segment display for showing output data in hexadecimal and decimal form, and a low power audio transducer. Manually operated toggle switches will provide a 4-bit parallel input. Two input push buttons should be available; one to simulate input events to be counted, the other to simulate an external interrupt input. Timed events should be measured or generated with an accuracy of better than 1%. The circuit will be battery powered, with a push button power switch to ensure that the power cannot be left on, and a power on indicator.

The board will be as small as possible, and the microcontroller must be easily reprogrammable, with flash memory.

7.3.2 Hardware Implementation

The seven-segment display will require seven outputs from the microcontroller. Active high operation can be provided by a common cathode LED display, and the display decimal point can be used as the power indicator. The audio transducer requires one output; a piezo buzzer was tested for suitability, since its power consumption is low. Although the device is specified to operate at a fixed frequency, it was found to be satisfactory in its frequency response. A miniature dip switch bank will be used for 4-bit input, and miniature push buttons used, to conserve space.

Fourteen I/O pins are required; the PIC 16F84A has only thirteen, so a chip with more I/O could be considered. However, the audio output and interrupt input could use the same I/O pin, because the high impedance of the buzzer will not interfere with input signals on the same pin. RB0 will be used as the dual function pin, since it is defined as the principal interrupt input, but can also be used as an output. The outputs can source up to 25 mA, but current-limiting resistors will restrict the current per display segment to 10–15 mA to control the maximum load on the port when all the segments are on.

The I/O allocation for the project is therefore as follows:

```
Seven-Segment display    Outputs    RB1–RB7
4-Bit switch bank        Inputs     RA0–RA3
Push button              Input      RA4
Push button interrupt    Input      RA0 (dual function)
Audio transducer         Output     RA0 (dual function)
```

A crystal clock of 4 MHz will be used to obtain the required timing precision, and the convenience of a 1 s instruction cycle. The 16LF84A-04 (LF = low voltage) can operate from a supply of between 2.0 and 5.5 V, so the circuit will be powered from 2 × 15 V dry cells, giving a 3.0 V supply. The "04" suffix indicates that a maximum 4-MHz clock frequency can be used.

A block diagram of the proposed system is shown in Fig. 7.7. The inputs and outputs are given the labels which will be assigned in the application programs.

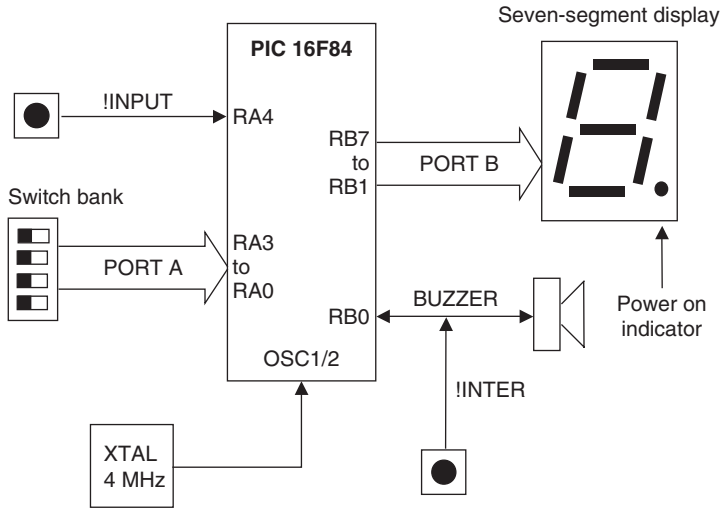


Figure 7.7: Block Diagram of DIZI Demonstration Board

7.3.3 Implementation

A circuit for the DIZI board is shown in Fig. 7.8. The PIC 16LF84A drives an active high (common cathode) low current seven-segment LED display at Port B, RB1–RB7, via a block of 270 R current-limiting resistors. RB0 drives an audio sounder when set as an output, but can also be used to detect the "Interrupt" push button when set as an input and the chip is initialized for this option. To prevent RB0 being shorted to ground if set as an output, the spare 270 R resistor is connected between the push button and RB0. This does not affect the operation of the sounder, which has a relatively high resistance. The flying lead is suggested because this would allow the all output pins to be monitored for audio output if, for example, BIN2 were run on this hardware.

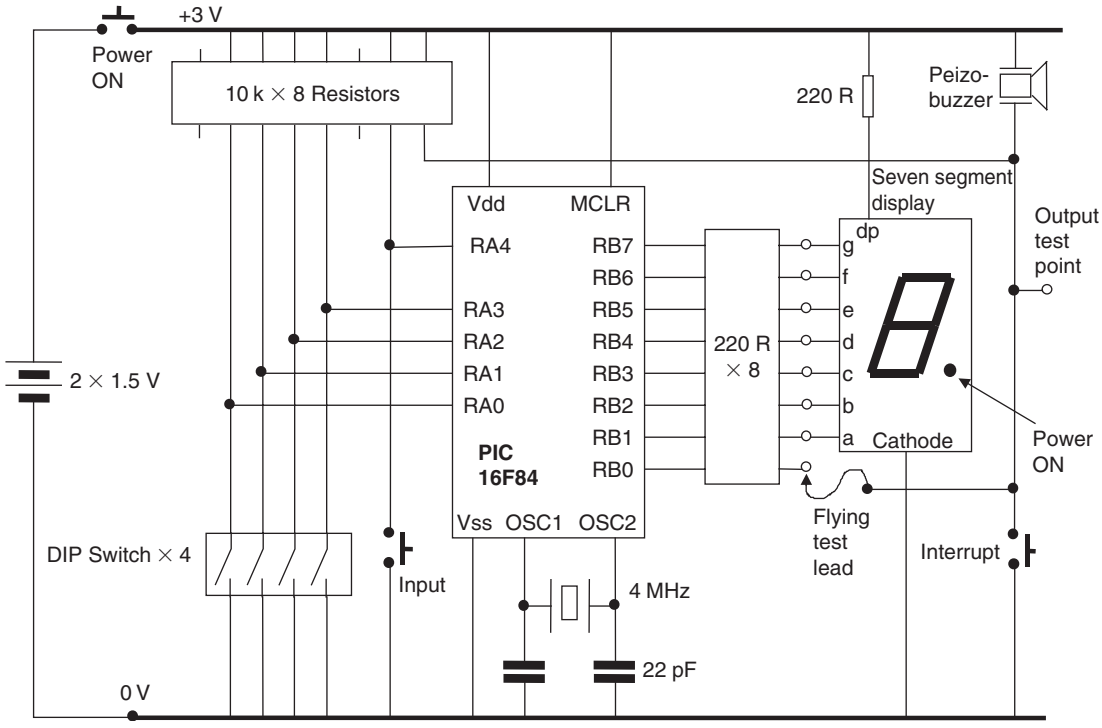


Figure 7.8: DIZI Board Circuit Diagram

A 4-bit DIP switch input is connected to Port A, RA0–RA3, with a push button connected to RA4, which can be used as an external pulse input to the counter/timer register RTCC. These operate as active low inputs with 10 k pull-up resistors, as does the interrupt push button.

A breadboard version of the circuit is shown in Fig. 7.9. A stripboard layout for the DIZI board is shown in Fig. 7.10. The detail of the component pin connections has been omitted

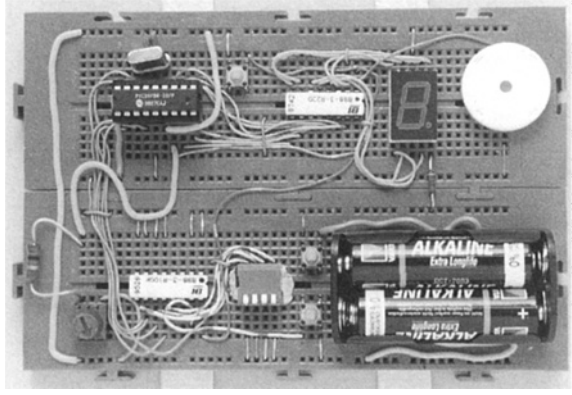


Figure 7.9: DIZI Breadboard Prototype Circuit

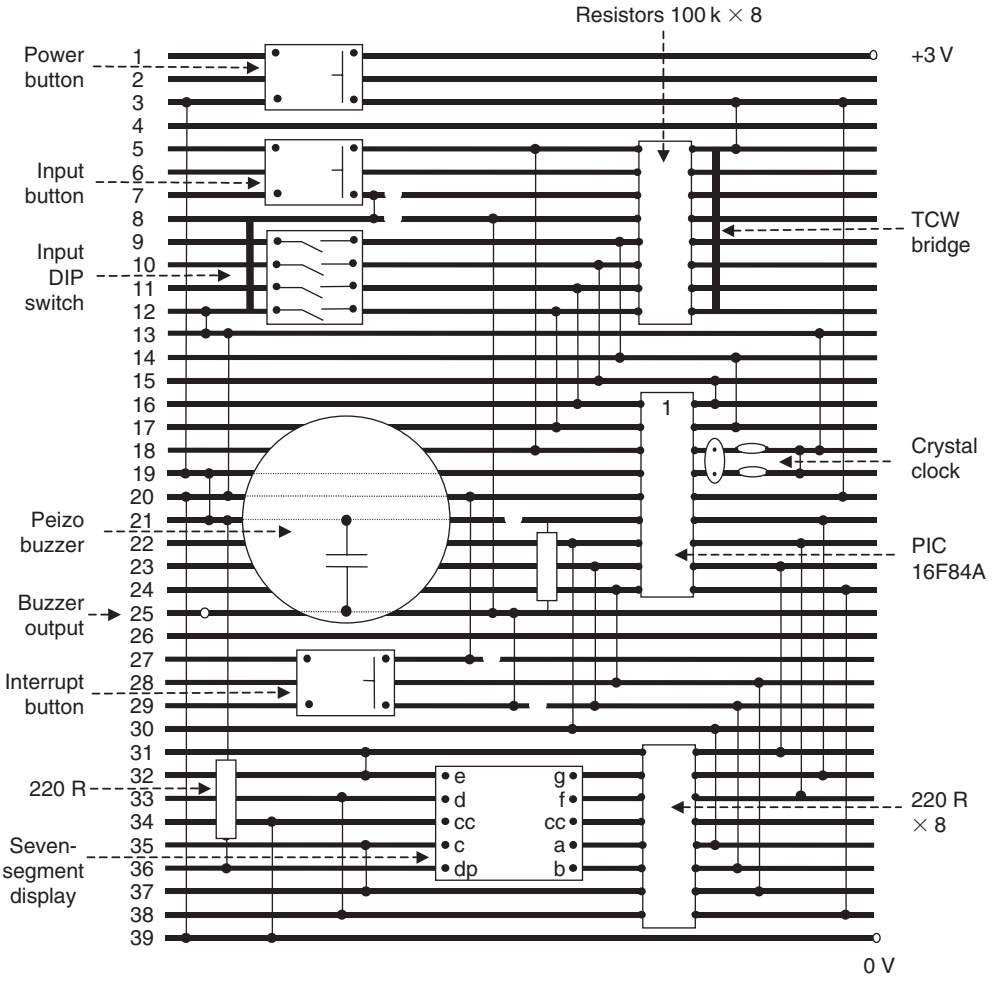


Figure 7.10: Stripboard Layout for DIZI Board

due to the reduced scale of the illustration, but this information can be obtained from the component pin out data, when selecting particular components.

The finished stripboard circuit is shown in Fig. 7.11.

7.4 Demo Board Applications

A set of programs to run on this hardware is listed below. Selected applications will be developed and coded (*), and the reader is invited to investigate the others, using the techniques covered thus far.

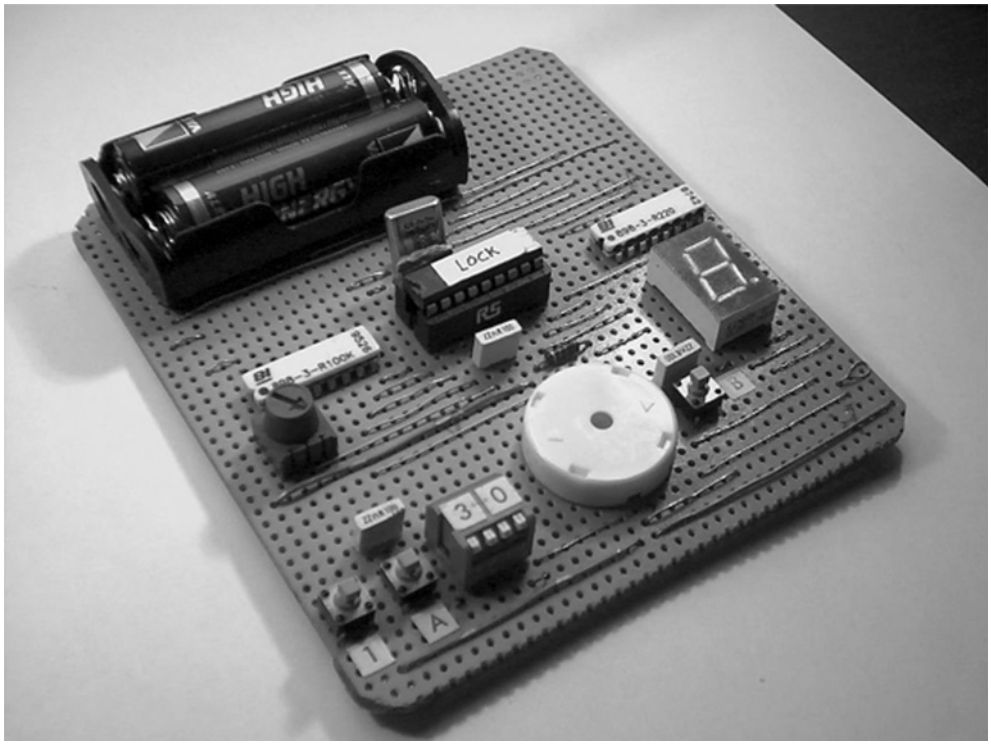


Figure 7.11: DIZI Stripboard Circuit

Display

	FLASH1	Flash all segments
	STEP1	Step through segments
	HEX1	Binary to Hex converter
	MESS1	Message display
	SEC1	One-Second Clock
	REACT1	Reaction timer
*	*DICE1*	Electronic dice

Sound

	BUZZ1	Output single tone
	SWEEP1	Sweep tone frequency
*	*TONE1*	Switch tone on/off
	SEL1	Select tone on switches
	GEN1	Audio frequency generator
	MET1	Metronome
	GIT1	Guitar tuner
*	*SCALE1*	Musical scale
	BELL1	Doorbell tune

Interrupts

STEP1	Step through scale
STEP2	Step scale and display note
BUZZ2	Output tone using TMR0
REACT2	Reaction timer using TMR0
SEC2	One-second clock using TMR0
MET2	Metronome using TMR0

EEPROM

STORE1	Store a display sequence in EEPROM
STORE2	Store a tone sequence in EEPROM
LOCK1	Store a code and buzz if matched

7.4.1 Program BUZZ1

A flowchart for the program BUZZ1 is shown in Fig. 7.12. It will generate a single tone at the buzzer when the input button is operated, by toggling the output to the buzzer, with a delay between each change of output state. If a count of 255 is used with a 1 μs instruction cycle time, we have seen that the loop itself will take:

$$255 \times 3 \times 1 = 765\,\mu s$$

which will give a frequency of approximately:

$$1000000/765 \times 2 = 650\,Hz$$

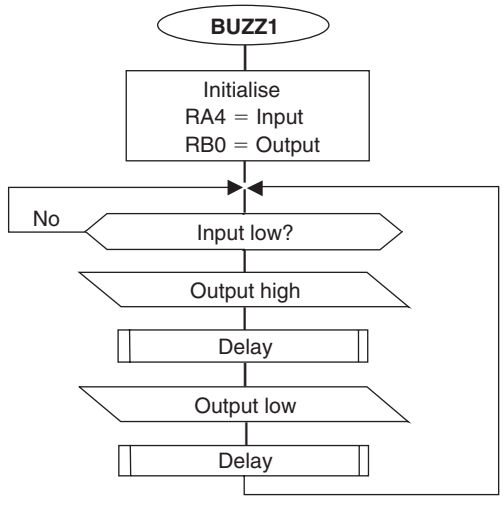


Figure 7.12: BUZZ1 Flowchart

The frequency is not critical, so we will ignore the additional loop instructions, because they will only make a small difference. The result is well within the audio range, and therefore is

suitable. It can be adjusted by simply reducing the count value in the delay loop; 650 Hz is the minimum frequency available. A more precise calculation of the delay loop can be used to obtain a more exact frequency, or the hardware timer can be used. In either case, the period can be checked using the stopwatch in the simulator before downloading (see Program 7.1).

```
;
;        BUZZ1.ASM              M.Bates               6/4/99
; ********************************************************
;
;       Generates an audio tone at Buzzer when the
;       Input button is operated..
;
;       Hardware:      PIC 16F84 DIZI Demo Board
;       Clock:         XTAL 4MHz
;       Inputs:        RA4       : Input (Active Low)
;       Outputs:       RB0       : Buzzer
;       MCLR:          Enabled
;
;       PIC Configuration Settings:
;       WDTimer:       Disable
;       PUTimer:       Enable
;       Interrupts:    Disable
;       Code Protect:  Disable
;
        PROCESSOR 16F84    ; Declare PIC device

; Register Label Equates.................................

PORTA     EQU          05  ; Port A
PORTB     EQU          06  ; Port B
Count     EQU          0C  ; Delay Counter

; Register Bit Label Equates ............................
Input     EQU          4   ; Push Button Input RA4
Buzzer    EQU          0   ; Buzzer Output RB0

; Start Program *****************************************

; Initialize (Default = Input) ..........................

          MOVLW        b'00000000'  ; Define Port B outputs
          TRIS         PORTB        ; and set bit direction
          GOTO         check
```

Program 7.1: BUZZ1 Source Code

```
; Delay Subroutine ....................................

delay    MOVLW       0FF          ; Standard Routine
         MOVWF       Count
down     DECFSZ      Count
         GOTO        down
         RETURN
; Main Loop ..........................................

check    BTFSC       PORTA,Input     ; Check Input Button
         GOTO        check           ; and wait if not 'on'
         BSF         PORTB,Buzzer    ; Output High
         CALL        delay           ; run delay subroutine
         BCF         PORTB,Buzzer    ; Output Low
         CALL        delay           ; run delay subroutine
         GOTO        check           ; repeat always
         END                         ; Terminate source code
```

Program 7.1: Continued

7.4.2 Program DICE1

This program will generate a random number at the display between 1 and 6 when the input button is pressed. A continuous loop will increment a register from 1 to 6, and back to 1. The loop is stopped when the button is pressed and the number displayed. The display is retained when the button is released. A table is required to work out the display digit codes.

First, the allocation of the segments to the pins on the display chip must be established. The segments of the display are labeled from a–g, as shown in Fig. 7.13. They must be lit in the appropriate combinations to give the required digit display; for instance, segments "b" and "c" must be lit for the digit "1" to be displayed. A table is useful here to work out the codes required for output to the display (Table 7.1).

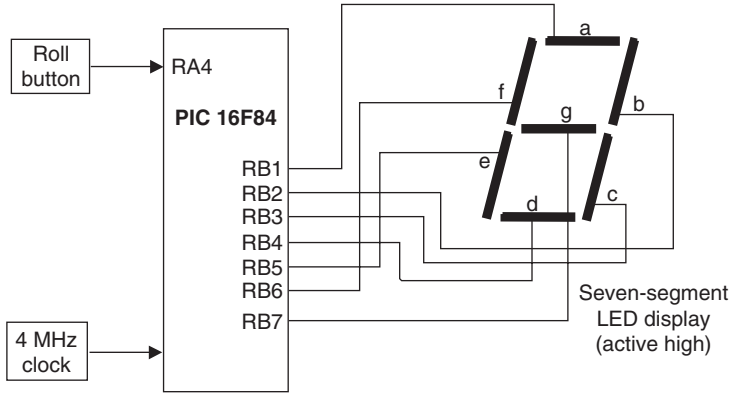


Figure 7.13: Block Diagram for DICE1 System

Table 7.1: DICE1 Display Encoding Table

Displayed Digit	Segment Code (1 = Segment On)							
	g RB7	f RB6	e RB5	d RB4	c RB3	b RB2	a RB1	Hex (RB0 = 0)
1	0	0	0	0	1	1	0	0C
2	1	0	1	1	0	1	1	B6
3	1	0	0	1	1	1	1	9E
4	1	1	0	0	1	1	0	CC
5	1	1	0	1	1	0	1	DA
6	1	1	1	1	1	0	1	FA

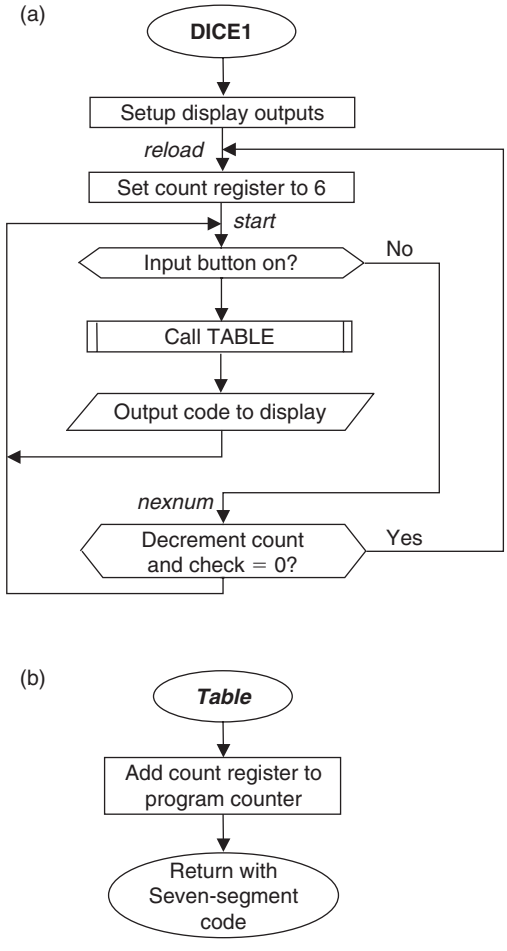


Figure 7.14: DICE1 Program Flowcharts

The display is "active high" in operation. This means a "1" at the pin will light that segment. This arrangement is also described as common cathode, as all the LED cathodes are connected together at the common terminal. A common anode display will therefore operate "active low." The binary or hexadecimal code for each digit will be included in the program in the form of a program data table.

The program represented in the flowchart, Fig. 7.14, uses a spare register as a counter which is continuously decremented from 6 to 0. When the button is pressed, the current number is used to select from the table of codes using the method described in Program 9.4. This results in the pseudo-random number code being displayed, and remaining visible until the button is pressed again. Because the number is selected by manually stopping a fast loop, the number cannot be predicted. In the flowchart, the jump destinations have been labeled, and these labels will be used in the program source code. The table subroutine is also named "table" to match the source code subroutine start label (see Program 7.2).

```
;       DICE1.ASM           M. Bates                    6/4/99
; ************************************************************
;
;       Displays pseudo-random numbers between 1 and 6 when
;       a push button is operated.
;
;
        Hardware:           PIC 16F84 DIZI Demo Board
;       Clock:              XTAL 4MHz
;       Inputs:             RA4       : Roll (Active Low)
;       Outputs:            RB1-RB7   : 7-Segment LEDs (AH)
;       MCLR:               Enabled
;
;       PIC Configuration Settings:
;       WDTimer:            Disable
;       PUTimer:            Enable
;       Interrupts:         Disable
;       Code Protect:       Disable
;
; Set Processor Options.....................................
        PROCESSOR 16F84     ; Declare PIC device

; Register Label Equates....................................
PCL     EQU       02        ; Program Counter
PORTA   EQU       05        ; Port A
```

Program 7.2: DICE1 Source Code

```
PORTB    EQU       06        ; Port B
Count    EQU       0C        ; Counter (1-6)

; Register Bit Label Equates.................................

Roll     EQU       4         ; Push Button Input

; Start Program ***********************************************

; Initialize (Default = Input)

         MOVLW     b'00000001'       ; Define RB1-7 outputs
         TRIS      PORTB             ; and set bit direction

         GOTO      reload            ; Jump to main program

; Table subroutine ....................................

table    MOVF      count,W           ; Put Count in W
         ADDWF     PCL               ; Add to Program Counter
         NOP                         ; Skip this location
         RETLW     00C               ; Display Code for '1'
         RETLW     0B6               ; Display Code for '2'
         RETLW     09E               ; Display Code for '3'
         RETLW     0CC               ; Display Code for '4'
         RETLW     0DA               ; Display Code for '5'
         RETLW     0FA               ; Display Code for '6'

; Main Loop ..........................................

reload   MOVLW     06                ; Reset Counter
         MOVLF     Count             ; to 6

start    BTFSC     PORTA,Roll        ; Test Button
         GOTO      nexnum            ; Jump if not pressed
         CALL      table             ; Get Display Code
         MOVWF     PORTB             ; Output Display Code
         GOTO      start             ; start again

nexnum   DECFSZ    Count             ; Dec & Test Count=0?
         GOTO      start             ; Start again
         GOTO      reload            ; Restart count if zero

         END                         ; Terminate source code
```

Program 7.2: Continued

7.4.3 Program SCALE1

This program will output a musical scale of eight tones. The frequencies for a musical scale from middle C upwards are:

262, 294, 330, 349, 392, 440, 494, 523 (Hz)

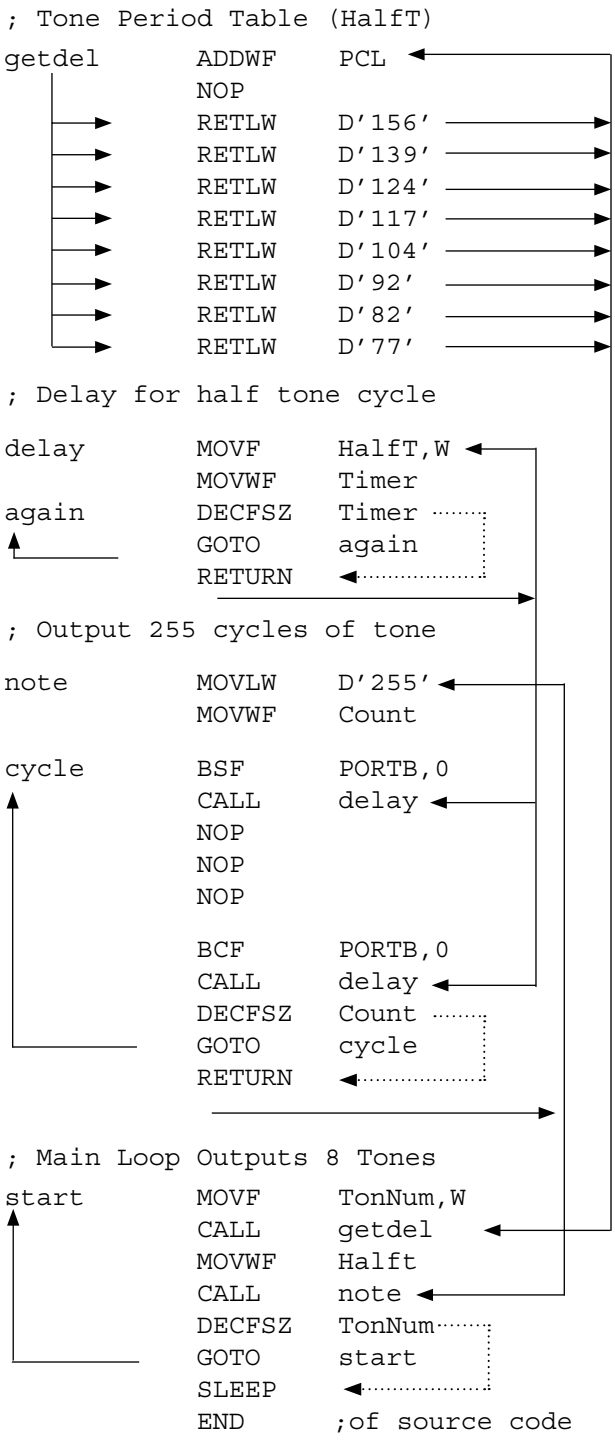
These can be translated into a table of delay counts which gives the required tone period, since:

Period, $T = 1/f$ (s)
where f = frequency (Hz)

The buzzer on the DIZI board is driven from RB0, so this needs to be toggled at a rate determined by the frequency of each tone. We therefore need to use a counter register or the hardware timer to provide a delay corresponding to half the period of each tone. We have seen in Section 6.1.2 how to calculate the delay time for a loop. Using a formula for the count value derived from this analysis, figures were calculated for a half cycle of each tone, which were then placed in the data table in SCALE1.ASM. To keep the program simple, each tone will be output for 255 cycles, so we will use another register to count the number of cycles competed during each tone. The scale will then be played over a period of about 5 s. The table of values can later be modified to play a tune in the doorbell program (see Program 7.3).

```
; SCALE1.ASM              M.Bates              6/4/99
; ************************************************

; Outputs a scale of 8 tones, 255 cycles per tone,
; tone duration of between a half and one second.
; Hardware: PIC 16F84
; XTAL 4MHz, !MCLR to start
; Audio Output: RB0

; Assign Registers *********************************

PCL         EQU      02             ; Program Counter
PORTB       EQU      06             ; Port B for Output
HalfT       EQU      0C             ; Half Period of Tone
Timer       EQU      0D             ; Delay Time Counter
Count       EQU      0E             ; Cycle Count
TonNum      EQU      0F             ; Tone Number (1–8)

; Initialize Registers

             MOVLW   B'11111110'    ; RB0 set..
             TRIS    PORTB          ; as output
             MOVLW   08             ; Set intial value of..
             MOVWF   TonNum         ; Tone Number
             GOTO    start          ; Jump to main program
```

Program 7.3: SCALE1 Source Code



```
; Tone Period Table (HalfT)
getdel      ADDWF    PCL
            NOP
            RETLW    D'156'
            RETLW    D'139'
            RETLW    D'124'
            RETLW    D'117'
            RETLW    D'104'
            RETLW    D'92'
            RETLW    D'82'
            RETLW    D'77'

; Delay for half tone cycle

delay       MOVF     HalfT,W
            MOVWF    Timer
again       DECFSZ   Timer
            GOTO     again
            RETURN

; Output 255 cycles of tone

note        MOVLW    D'255'
            MOVWF    Count

cycle       BSF      PORTB,0
            CALL     delay
            NOP
            NOP
            NOP

            BCF      PORTB,0
            CALL     delay
            DECFSZ   Count
            GOTO     cycle
            RETURN

; Main Loop Outputs 8 Tones
start       MOVF     TonNum,W
            CALL     getdel
            MOVWF    Halft
            CALL     note
            DECFSZ   TonNum
            GOTO     start
            SLEEP
            END      ;of source code
```

Program 7.3: Continued

Instead of a flowchart, the SCALE1 program source code listing has been annotated with arrows to show the execution sequence. This informal method of analysis can be used to check the program logic prior to simulation. The eight tone frequencies are controlled by the value of "Half T" obtained from the program data table at "getdel." "HalfT" is a counter value which will give a delay corresponding to a half cycle of the frequency required when the chip is clocked at 4 MHz. The eight tones are selected in turn by the value of "TonNum," which is initialized to 8. This is used as the program counter offset in the data table fetch operation. It is decremented in the main loop after each tone has finished to select the next. The "HalfT" values are thus selected from the bottom of the table upwards.

The tone is generated in the routine "note," where RB0 is set high, the delay using "HalfT" runs, RB0 is cleared, and the second half cycle delay executed. No operation instructions (NOP) have been inserted to equalize the duration of each half cycle. RB0 is toggled 255 times using the "Count" register, which gives duration of around half a second, depending on which tone is being generated (the lower frequencies are output for longer). The main loop thus selects each of the eight values of "HalfT" in turn, and outputs 255 cycles of each tone. The program is terminated with the SLEEP instruction to stop program execution running into the unused locations following the program.

7.4.4 DIZI Application Outlines

Some further applications are outlined below, for the reader to develop for the DIZI hardware.

7.4.4.1 HEX1 Hex Converter

The hexadecimal number corresponding to the binary setting of the DIP switch inputs is displayed. The input switches select from a table of 16 seven-segment codes which light up the segments in the required pattern for each hex digit display:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, b, C, d, E

Note that numbers B and D are displayed in lower case on a seven-segment display so that they can be distinguished from 8 and 0, respectively. Use the switch input number to select one of the 16 codes from a seven-segment table.

7.4.4.2 MESS1 Message Display

A sequence of characters is displayed for about 0.5 s each. Most letters of the alphabet can be obtained on the seven-segment display in either upper or lower case, for instance "HI tHErE." Output a character code table with delay. The number of characters must be set in a counter, or a termination character used.

7.4.4.3 SEC1 One-Second Timer

An output is displayed which increments exactly once per second, from 0 to 9, and then repeats. A table of display codes is required as in the "Hex Converter." A one-second

time delay can be achieved using the hardware timer (see Chapter 6) and spare register. A "tick" could be produced at the audio output by pulsing the speaker at each step.

7.4.4.4 REACT1 Reaction Timer

The user's reaction time is tested by generating a random delay of between 1 and 10 s, outputting a sound, and timing the delay before the input button is pressed. A number representing the time between the sound and the input, in multiples of 100 ms, is then displayed as a number 0–9, giving a maximum reaction time of 900 ms.

7.4.4.5 GEN1 AF Generator

An audio frequency generator outputs frequencies in the range 20 Hz–20 kHz. The sounder output is toggled with a delay between each operation determined by the frequency required, as in the BUZZ program. For example, for a frequency of 1 KHz, a delay of 1 ms is required, which is 1000 instruction cycles at a cycle time of 1 s. The delay time, and hence the frequency, can then be incremented using the input button, and range selection with the input switches might be incorporated, as there are only 255 steps available when using an 8-bit register as the period counter.

7.4.4.6 MET1 Metronome

An audible pulse is output at a rate set by the DIP switches or input buttons. The output tick can be adjustable from, say, 1 up to 4 beats per second, using the interrupt button to step the speed up and down, and the input button to select up or down. A software loop or the TMR0 register can be used to provide the necessary time delays.

7.4.4.7 BELL1 Doorbell

A tune is played when the input button is pressed, using a program look-up table for the tone frequency and duration. Each tone must be played for a suitable time, or number of cycles, as required by the tune. The program can be elaborated by selecting a tune using the DIP switches, and displaying the number of the tune selected.

7.4.4.8 GIT1 Guitar Tuner

The program will allow the user to step through the frequencies for tuning the strings of a guitar, or other musical instruments using the input button, or selecting the tone at the DIP switches. The program could be enhanced by displaying the string number to be tuned. The tone frequencies will be generated as for the doorbell application. The digit display codes would also be required in a table.

7.5 Summary

- Methods of circuit construction available include PCB, breadboard and stripboard.
- Design software is available to draw the application circuit, test it by interactive simulation and create a board layout, but it requires time to learn to use the software, and etching equipment is needed to produce a PCB.
- Breadboard is reusable and allows circuits to be prototyped quickly and easily, but is unreliable for complex circuits.
- Stripboard is more reliable, but not reusable. Connections can be laid out on paper, using computer drawing tools or directly onto the board.
- The DIZI demonstration board has a seven-segment display and audio output, with push button input and interrupt and a 4-bit switched input, and can be used for demonstrating a range of simple applications.

CHAPTER 8

More PIC Applications and Devices

The PIC 16F84 has been used as a reference device so far because its architecture and operation are relatively simple compared with other PIC microcontrollers. The range of flash memory PIC chips has now expanded such that alternative devices are now available which have more features at a lower unit cost. We will continue to concentrate on PICs that have flash program memory, since these are the best choice for learning about application development, prototyping and producing one-off or low volume products. The main groups of PIC flash devices are shown in Table 8.1. They are divided into three groups, with a different prefix number: the 12XXXX series are 8-pin miniature PICs, the 16XXXX group might be described as the standard series and the 18XXXX devices as the high performance group. Full details of all these devices and the rest of the PIC range are provided at www.microchip.com, from where the individual data sheets can be downloaded as PDF files.

In this chapter, an application for the 16F877 will be described in some detail, and we'll show how similar applications could be implemented using a range of other devices.

8.1 16F877 Application

The 16F877 is at the top of the range in this group, and will therefore be used to illustrate the range of features available within the 16 series. Other chips in this group have different combinations of these features; the intention is to help the reader to make the best choice of chip for any given application.

The temperature controller described here uses most of the available I/O provided, including analog inputs. The 8k memory should be sufficient for most application programs which might be developed for this hardware. A demonstration program is provided which will exercise the hardware for test purposes, but it will be left to the reader to develop a fully functional application.

8.1.1 Temperature Controller System

A temperature controller is required to control a system such as a greenhouse where the temperature must be kept within set limits (0–50ºC) by a heating and ventilation system (Fig. 8.1).

Table 8.1: PIC Flash Microcontrollers

12FXXX

- Low cost and small size
- 8-pin packages
- 6 I/O pins
- 33/35 × 12/14-bit instructions
- 1 k word program memory
- 20 MHz clock
- 4 MHz internal oscillator
- 8-bit and 16-bit timer
- Up to 4 analogue inputs*
- In-circuit programming and debugging*

16FXXX

- Mid-range cost and performance
- 14–40-pin packages
- 12–33 I/O pins
- 35 × 14-bit instructions
- 1–8 k word program memory
- 20 MHz clock
- 4/8 MHz internal oscillator*
- 2 × 8-bit and 1 × 16-bit timers*
- Up to 8 analogue inputs
- Serial communication ports, parallel slave port*
- 1/2 pulse width modulation outputs, capture & compare inputs*
- In-circuit programming and debugging*

18FXXX

- High performance
- 18–80-pin packages
- 13–68 I/O pins
- 58 × 16-bit instructions
- 2–64 k word program memory
- 40 MHz clock
- 8/10 MHz internal oscillator*
- 2–64 k program memory
- Up to 2 × 8-bit and 3 × 16-bit timers
- Up to 16 analogue inputs
- Serial communication ports, parallel slave port*
- Up to 14 pulse width modulation outputs, capture and compare inputs*
- CAN communication interface*
- In-circuit programming and debugging*

*Selected devices in the range

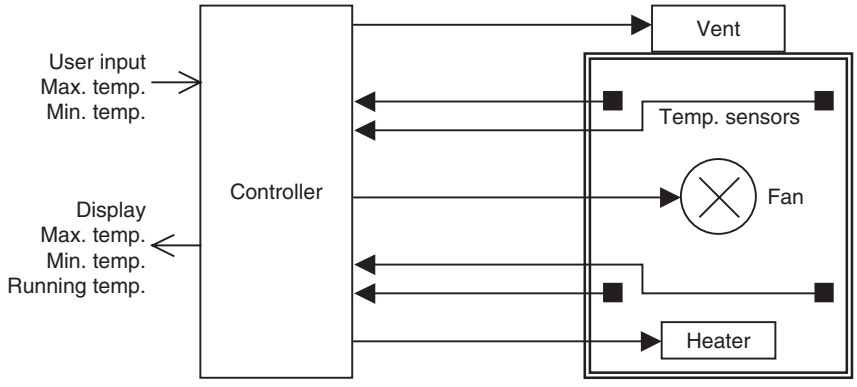


Figure 8.1: Temperature Control System

The unit will be programmed to accept a maximum and minimum temperature, or a set temperature and operating range. The system operates on the average temperature reading from four sensors to give a more accurate representation of the overall temperature in the enclosure. Using more than one temperature sensor also allows the system to tolerate a fault in one sensor, if the application software includes a check to see if one sensor is out of range. The temperature is maintained by a switched heater, switched vent and a fan which can be speed-controlled. The system should operate as specified in Table 8.2. The fan is fitted to the heater, so that it can be used for forced heating or cooling, depending on whether the

Table 8.2: Temperature Controller Function Table

Measured Temp.	Heater	Vent	Fan	Action
Temp. ⩽ Min.	**ON**	OFF	**ON**	Forced heating
Temp. < Min.	**ON**	OFF	OFF	Heating
Min. < Temp. < Max.	OFF	OFF	OFF	Correct temp.
Temp. > Max.	OFF	**ON**	OFF	Cooling
Temp. ⩾ Max.	OFF	**ON**	**ON**	Forced cooling

heater is on. It needs to be connected to a PWM output on the controller if the rate of forced heating and cooling is to be varied. A demonstration system was constructed, where the heater was represented by a pair of filament lamps and the fan by a 5 V CPU fan. The temperature sensors were standard LM35 devices which have a built-in amplifier which outputs 10 mV per ºC. Figure 8.2 shows the interfacing requirements for the application. The temperature sensor readings can be averaged, or processed with a weighting factor for each, to give a representative value for the measured temperature. A heater is then controlled via a suitable

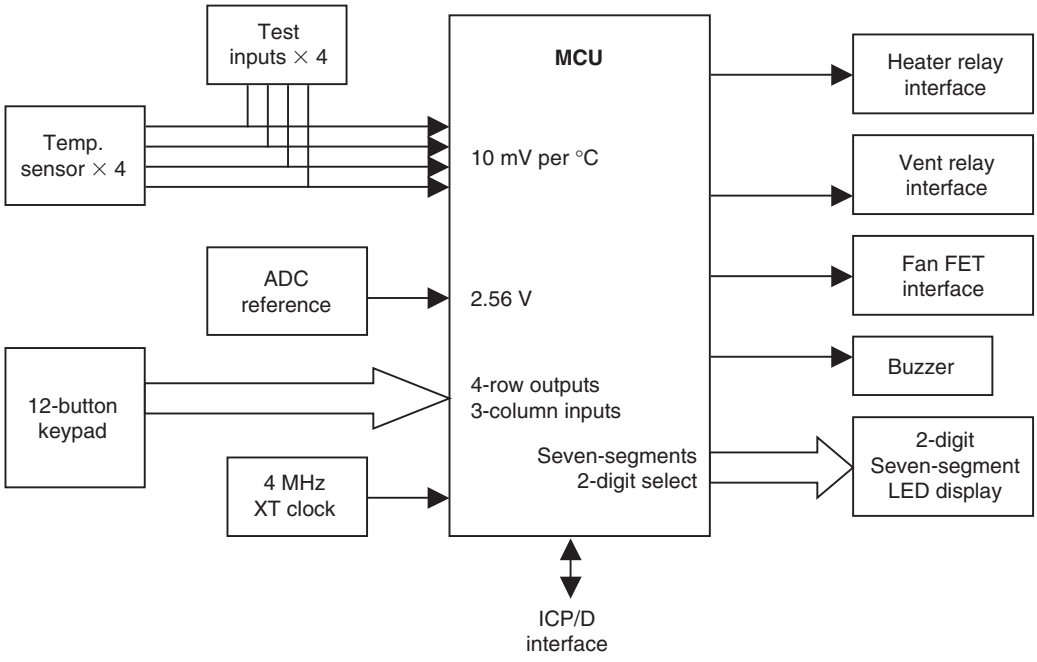


Figure 8.2: Temperature Controller Interfacing

interface. A relay can be used if on/off control is sufficient. If proportional control is required, a PWM output would be required. In the hardware design provided here, the heater and vent interfaces are implemented as normally open switched relays, so that an external power supply can be used. The fan output demonstrates the alternative solid state interface, using a general purpose power FET. This would allow proportional control, but the external circuit must be operated at 5 V. For PWM control, the FET output would have to be re-allocated to one of the PWM outputs on the PIC 16F877.

8.1.2 I/O Allocation

The I/O functions provided by the PIC 16F877 are detailed in Table 8.3. These were then mapped against the requirements of the application, and the most convenient grouping decided, giving the I/O allocation in Table 8.4.

The interfacing for this application is typical of that required for simple control systems using microcontrollers. Fortunately, this application does not need any analog signal conditioning on the input side, as the temperature sensors can be connected directly to the PIC. Other references on interfacing will cover the range of design techniques needed for the most common sensors and output devices.

Table 8.3: 16F877 Pin Functions

Pin label	Function
	Port A (6 bits)
RA0/AN0	Digital I/O or analog input 0
RA1/AN1	Digital I/O or analog input 1
RA2/AN2/Vref−	Digital I/O or analog input 2 or positive reference voltage for ADC
RA3/AN3/Vref+	Digital I/O or analog input 0 or negative reference voltage for ADC
RA4/T0CKI	Digital I/O or input to timer 0
RA5/AN4/SS	Digital I/O or analog input 0 or slave select input (SPI mode)
	Port B (8 bits)
RB0/INT	Digital I/O or external interrupt input
RB1	Digital I/O
RB2	Digital I/O
RB3/PGM	Digital I/O or select serial programming mode
RB4	Digital I/O (interrupt on change)
RB5	Digital I/O (interrupt on change)
RB6/PGC	Digital I/O or in-circuit debugger or serial programming clock input (interrupt on change)
RB7/PGD	Digital I/O or in-circuit debugger or serial programming data input (interrupt on change)
	Port C (8 bits)
RC0/T1OSO/T1CKI	Digital I/O or Timer 1 oscillator output or Timer 1 clock input
RC1/T1OSI/CCP2	Digital I/O or Timer 1 oscillator input or Capture 2 input or Compare 2 output or PWM2 output
RC2/CCP1	Digital I/O or Capture 1 input or Compare 1 output or PWM1 output
RC3/SCK/SCL	Digital I/O or Synchronous serial clock input or output in SPI and I^2C modes
RC4/SDI/SDA	Digital I/O or SPI data input or I^2C data I/O
RC5/SDO	Digital I/O or SPI data output
RC6/TX/CK	Digital I/O or USART asynchronous transmit or USART synchronous clock I/O
RC7/RX/DT	Digital I/O or USART asynchronous receive or USART synchronous data I/O
	Port D (8 bits)
RD0/PSP0	Digital I/O or parallel slave port bit 0
RD1/PSP1	Digital I/O or parallel slave port bit 1
RD2/PSP2	Digital I/O or parallel slave port bit 2
RD3/PSP3	Digital I/O or parallel slave port bit 3
RD4/PSP4	Digital I/O or parallel slave port bit 4
RD5/PSP5	Digital I/O or parallel slave port bit 5
RD6/PSP6	Digital I/O or parallel slave port bit 6
RD7/PSP7	Digital I/O or parallel slave port bit 7
	Port E (3 bits)
RE0/RD/AN5	Digital I/O or PSP read select or analog input 5
RE1/WR/AN6	Digital I/O or PSP write select or analog input 6
RE2/CS/AN7	Digital I/O or PSP chip select or analog input 7

Table 8.4: Temperature Controller I/O Allocation

Device	Function	16F877 Pin	Initialization
Temperature sensors	10 mV per °C 0–512 mV = 0–51.2°C	RA0, RA1, RA2, RA5	AN0, AN1, AN2, AN4
ADC reference voltage	2.048 V	RA3	VREF+
Heater	Switched output	RE0	Digital output
Vent	Switched output	RE1	Digital output
Fan	Switched output	RE2	Digital output
4 × 3 keypad	Read column Scan row	RD3, RD4, RD5, RD6 RD0, RD1, RD2	Digital output Digital input
2 × 7-segment display	Segments Digit select	RC1–RC 7 RB1, RB2	Digital output Digital output
Buzzer	Audio alarm	RB0	Digital output
ICP/D interface	Program & debug	RB3, RB6, RB7	N/A

8.1.3 Temperature Controller Circuit Description

Figure 8.3 shows the circuit for the temperature controller. Each section of the external circuits can be described separately.

8.1.3.1 Analog Inputs

The four temperature sensors are allocated to four of the eight analog inputs available on the chip. In the demo system, standard sensors with an output of 10 mV per ºC were used (0ºC = 0 mV). The controller is designed to operate at up to 50ºC, at which temperature the sensor output is 500 mV. This relatively low voltage is acceptable if the sensors are not connected on long leads, which could pick up electrical noise. For more remote operation, a DC amplifier should be used at the sensor end of the connection to increase the voltage to, say, 5.00 V at 50ºC. Alternatively, or in addition, screened leads could be used. The inputs are protected by a low pass RC filter; the input impedance at the ADC is high enough for this to have negligible effect on the input voltage.

The ADC normally operates at 10-bit resolution, giving output values 0–1024. It needs reference voltages to set the maximum and minimum values for the input conversion. These can be provided internally as Vdd and Vss (supply values), but Vdd does not give a convenient conversion factor. Therefore, an external reference value is provided from a 2.7 V zener diode and potential divider, giving Vref+ which is adjusted to 2.048 V. This then gives a conversion factor of 2048/1024 = 2 mV per bit. To simplify the software, and to cover the correct range, only the low eight bits of the ADC result will be used, with a maximum value of 255. At 50ºC,

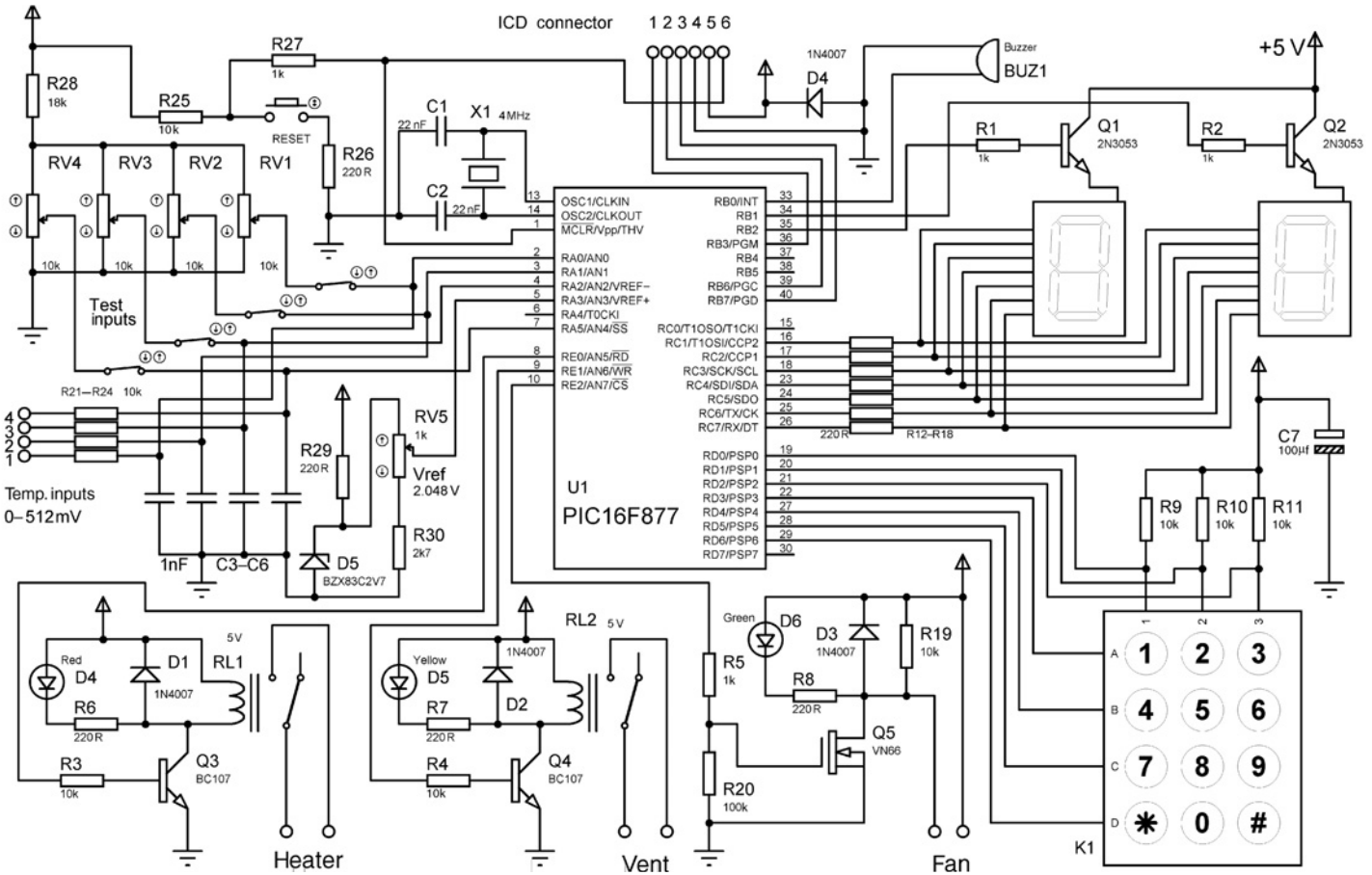


Figure 8.3: 16F877 Temperature Controller Circuit Diagram

the input will be 500 mV/2 mV = 250, giving a resolution of 0.2ºC per bit. For test purposes, a set of four on-board pots are provided, so that input voltages in this range can be input manually, to check the operation of the software without having to heat and cool the target system. These can be switched in and out as required via a bank of DIP switches.

8.1.3.2 Outputs

Two types of output are provided, relay and FET. The relay gives a switched output that is isolated (electrically separated) from the controller. The external circuit operates with a separate supply, so the load (heater in this case) can be powered from a high-voltage supply if necessary. The relay also provides a high off resistance (an air gap). The FET interface, on the other hand, is more reliable, as it is solid state. The problem is that the load has to operate from the same supply as the FET, the 5 V board supply. It also does not provide electrical isolation between the controller and the load, unless an opto-isolator is included between the FET and the MCU. However, the FET can be switched at high frequency, while the relay cannot. The outputs include an on-board LED to indicate their status, in case the state of the outputs cannot be seen or are not connected.

8.1.3.3 Keypad

The 12-button keypad allows the user to input the required temperature and other operating parameters as required by the application program. The target temperature would typically be input as a 2-digit number. It may also be desirable to input upper and lower limits, alarm levels and so on. These should be displayed as they are entered, to ensure that the correct figures are stored. The keypad is simply a set of switches connected in a row and column arrangement (see Chapter 7) and accessed by a scanning routine. If the row inputs (A, B, C, D) are all set high initially, and no button is pressed, all the column outputs (1, 2, 3) will be high (pulled up to 5 V). If a 0 is output on each row in turn, and a button pressed, that '0' will appear at the column output, and can be read in to the PIC. The combination of active row and column identifies the key. The demonstration program (Program 8.1) includes a simple keypad scanning routine.

8.1.3.4 Display

A seven-segment display is used as it is relatively easy to drive, compared with a liquid crystal display, and is self-illuminating. The decoding process has been covered in Chapter 7, where a code to illuminate the segments to display digits 0–9 is looked up in a program data table. In this case, two digits are required, but they can both be operated from the same set of outputs by multiplexing. The digits are switched on alternately via Q1 and Q2; because they are switched too fast for the eye to perceive, they appear to be on at the same time, albeit at reduced brightness.

8.1.3.5 Other Circuit Elements

A buzzer is fitted to provide an audible alarm output. This can be used to signal system failure, temperature too low for too long and so on. Audible feedback from keystrokes is also desirable. A 4 MHz clock is used to give a convenient instruction execution time, although no timing-critical operations are required. In the context of this circuit, cost saving on clock components would be insignificant. A manual reset is provided, so that the program can be restarted without powering down. This will be useful for testing as well as in normal operation. In-circuit programming and debugging are provided for via the ICD connector. The ICD module must be connected between the host PC and the application board. MPLAB IDE can then be used for testing the program in software initially, and then finally in ICD mode.

8.1.4 Hardware Development

The circuit was developed using Labcenter™ ISIS schematic capture software, which provides interactive components for circuit testing, and integrated software and hardware testing. When the circuit had been tested by interactive simulation, a stripboard implementation was devised (Fig. 8.4).

A demo target system was then constructed, comprising two filament lamps as the heaters, operating from a high current 5 V supply, controlled by the relay output on the application board. A 5 V CPU fan was fitted as the cooling element, and the temperature sensors were arranged symmetrically inside the enclosure. The wiring of the target hardware is shown in Fig. 8.5. Note that there was a sensor output on the fan which could be used to monitor the actual fan speed, if a suitable interface were designed to convert the fan sensor pulse to TTL levels. The vent was not physically implemented at this stage.

The photo of the prototype system (Fig. 8.6) shows the simulator at the right of the picture, with the ICD module (enclosed in ABS box), which is connected to ICD input of the TEMPCON board. 5V power supplies and a host PC would complete the system.

When final hardware testing was completed, an application board was created using Labcenter ARES™ PCB layout software, shown in Fig. 8.7. This incorporated an on-board +5 V supply for operation from a mains adapter.

8.1.5 Temperature Controller Test Program

Program 8.1 was written to exercise the hardware and to help get the reader started in developing applications for the TEMPCON hardware.

The program will read in the analog inputs and display the raw data on the displays. An apparently random pattern results, which changes if the analog inputs (test pots) are varied, indicating that the hardware input and display interfaces are working. Pressing a key on the

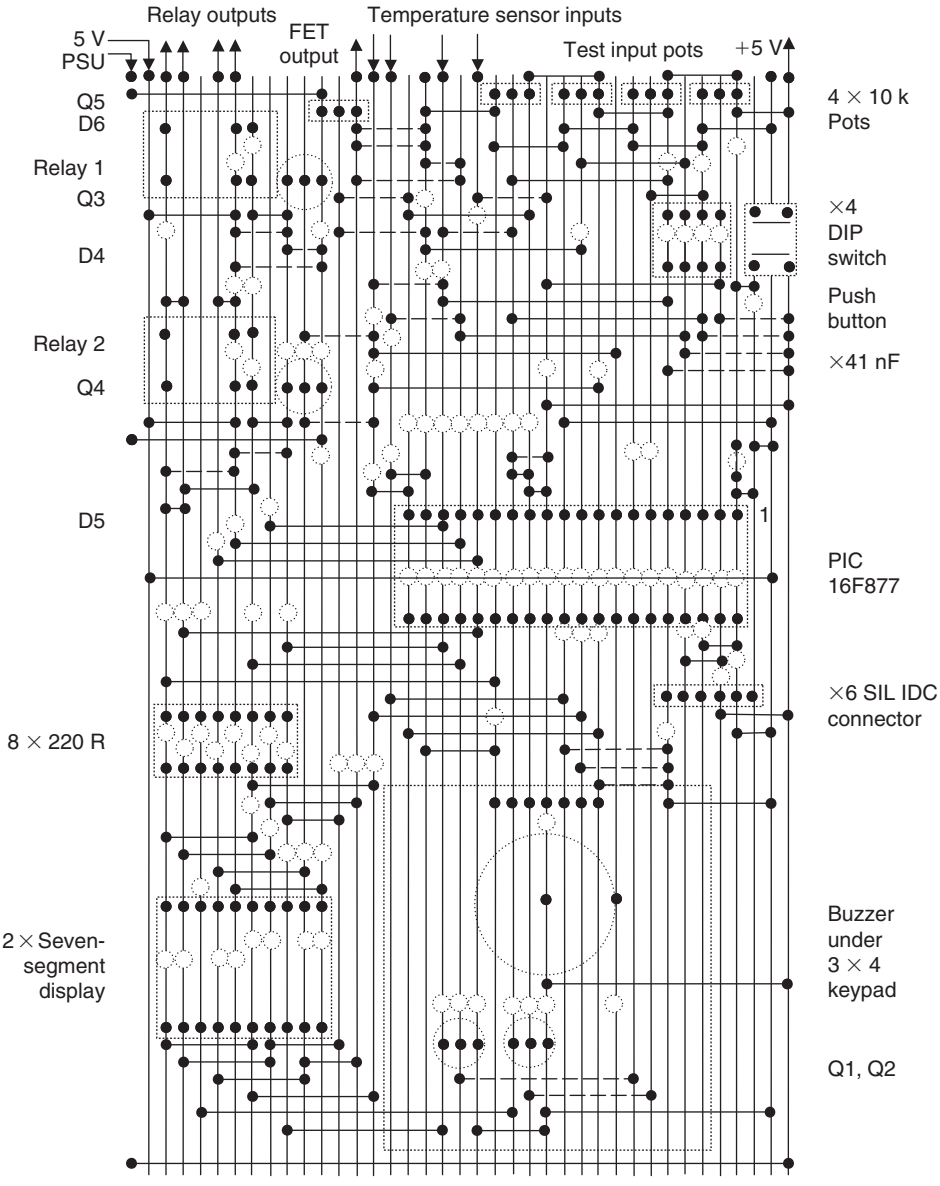


Figure 8.4: Stripboard Layout for TEMPCON Board

keypad will select an analog input for display; key 1 for input 1, and so on to 4, then repeating for keys 5–8. Key 9 will enable the buzzer test, while *, 0 and # will operate the heater, fan and vent, respectively. A full header has been included with as much information as possible; details of target system, program description, register initialization, port allocation and so on. The ports and analog control registers have been initialized using bank selection, as recommended.

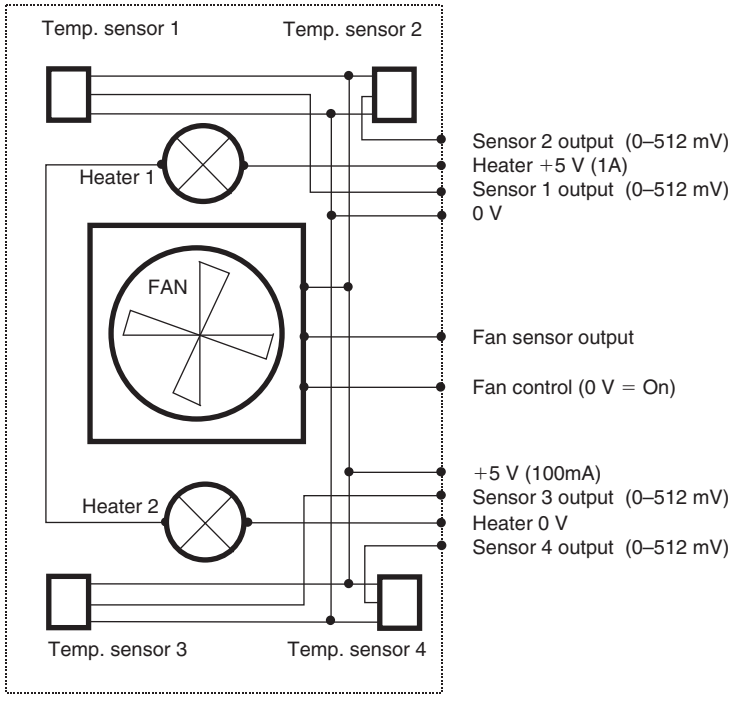


Figure 8.5: Greenhouse Simulator Wiring

(a)

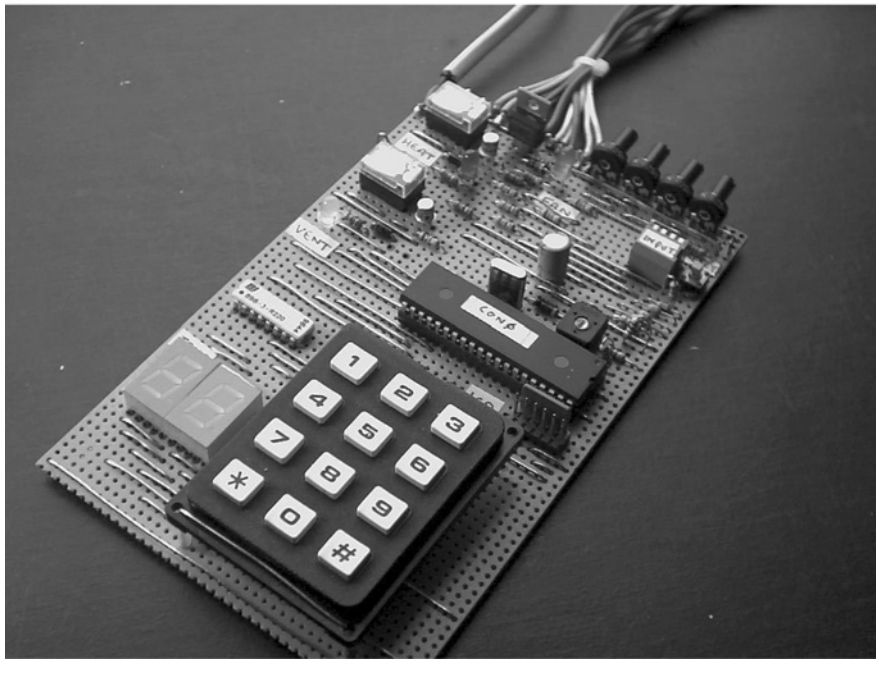


Figure 8.6: TEMPCON System: (a) Stripboard Version of Temperature Controller Board

(b)

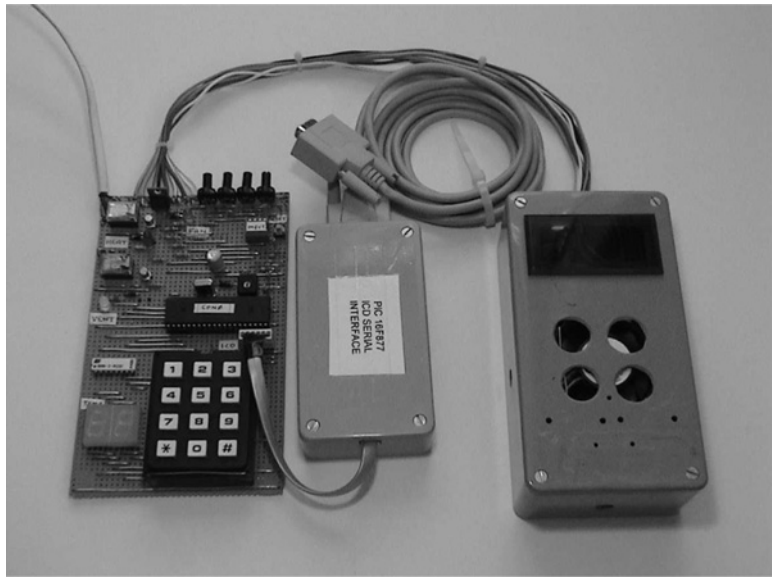


Figure 8.6: (Continued) (b) Temperature Controller System with ICD Module and Dummy Load

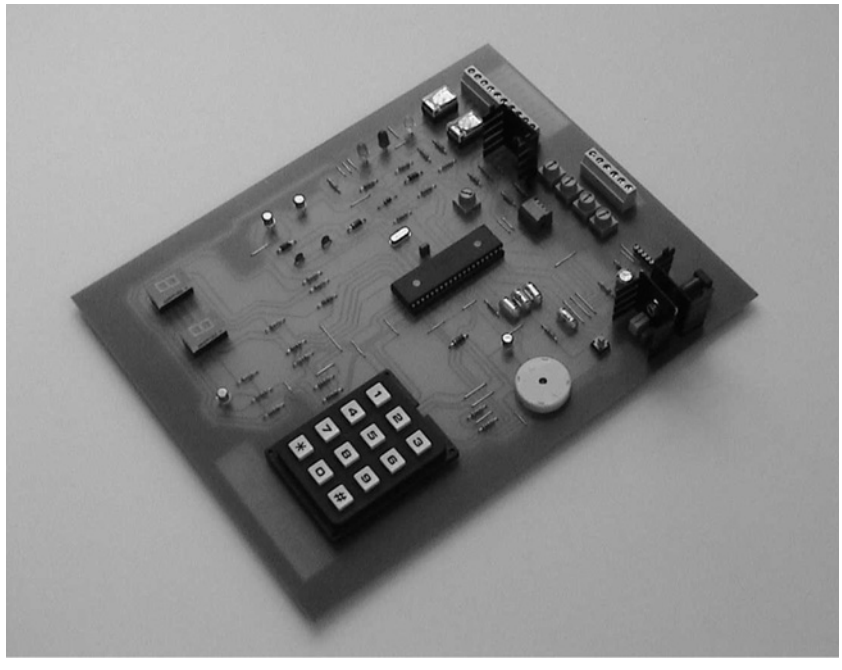


Figure 8.7: Temperature Controller Board

```
;*****************************************************************
;   Source File:            CON0.ASM
;   Design & Code:          M Bates
;   Date:                   29-6-02
;   Version:                1.0
;   Customer:               HCAT
;
;   Target Hardware:        16F877 Controller Board (TEMPCON Board)
;   Design & Layout:        M Bates
;   ISIS Design File:       CON0.DSN
;   Layout:                 Stripboard prototype
;
;   Development System:     MPLAB ICD Evaluation kit
;                           MPLAB IDE Ver 5.30.00
;   Assembler:              PICSTART Plus 2.40.00
;
;*****************************************************************
;
;   Test program for PIC877 Controller Board using ICD system
;   Use with ISIS CON0.DSN for simulation testing
;
;   Circuit description:
;   PIC 16F877 flash microcontroller gets 4 analogue inputs from temp
;   sensors (or test input pots) to control Heater, Vent and Fan in a
;   target system such as a greenhouse. Target temp will be set up
;   using keypad input and displayed on 2-digit multiplexed LED display.
;   Test program:
;   Checks all inputs and outputs for correct hardware operation.
;   -Press keypad buttons 1-4 to display raw data from input pots
;   -Buttons 5-8 ditto
;   -Button 9 to sound buzzer
;   -Button * to operate HEATER output
;   -Button 0 to operate VENT output
;   -Button # to operate FAN output
;
; PROGRAMMING OPTIONS ***********************************************
;
;   When programming PIC 16F877 in ICD mode, select:
;
;   XT clock mode (4MHz, 1us per instruction)
;   Power-up timer enabled
;   Watchdog timer disabled
;   Code Protection off
```

Program 8.1: Test Program for TEMPCON Board

```
;
; I/O ALLOCATION ***************************************************
;
;  INPUTS.....................................................
;
;  Analogue temp sensors      AN0 - AN3 (0 – 5V)
;  Keypad column detect       RD0 - RD2 = 0
; OUTPUTS....................................................
;
;  Buzzer                                                RB0 = toggle
;  Keypad row select                                     RD3 – RD6 = 0
;  7-segment display            Select lo digit          RB1 = 1
;                               Select hi digit          RB2 = 1
;                               Segments                 RC1 – RC7 = 0
;  Relay interfaces Heater                               RE0 = 1
;                                                        Vent RE1 = 1
;  FET interface                                         Fan RE2 = 1
;
;  PORT DATA DIRECTION CODES REQUIRED ..........................
;
;  TRISA = 11111111
;  TRISB = 11111000
;  TRISC = 00000000
;  TRISD = 00000111
;  TRISE = 00000000
;
;  RB3, RB6, RB7 reserved for ICD operation
;
; ADC SETUP ********************************************************
;
;  ADCON0   Bits 76         01 = A/Dclock = f/8)
;           Bits 543        Channel Select (AN0 – AN7)
;           Bit 2           Go = 1 / Done = 0
;           Bit 0           A/D module enable = 1
;
;  ADCON0 = 01xxx001 depending on channel required
;
;  ADCON1   Bit 7           0 = left justify result in ADRESH/ADRESL
;           Bits 3210       0010 = RA0-RA5 analogue, RE0-RE2 digital
;
;  ADCON1 = 00000010
; ASSEMBLER DIRECTIVES *********************************************
;
;       Create list file and select processor:
       list p = 16f877
```

Program 8.1: Continued

```
;
;       Include file containing register labels:
        include "p16f877.inc"
;
count   EQU       020         ; assign GPR1 for counter
;
;       Set origin at address 000:
        org       0x000
;
; START PROGRAM ***************************************************

        nop                   ; No op. required at 000 for ICD mode

; Initialise control registers ...................................

        banksel   TRISA       ; Select DDR resgister bank 1
        movlw     b'11111000' ; Setup buzzer and display digit
                              ; select..
        movwf     TRISB       ; ..as outputs
        clrf      TRISC       ; Setup 7-segment driver port as outputs
        movlw     b'00000111' ; Setup keyboard port for..
        movwf     TRISD       ; .. row outputs and column inputs
        clrf      TRISE       ; Setup relay port as outputs

; Setup ADC ......................................................

        banksel   ADCON1      ; Select register bank 1
        movlw     b'00000010' ; Set A/D mode left justify,4 channels
        movwf     ADCON1      ; and write A/D control word

        banksel   ADCON0      ; Select register bank 0
        movlw     b'01000001' ; Set A/D frequency Fosc/8, select AN0
        movwf     ADCON0      ; and write A/D control word

; Initialise outputs ....... ....................................

        banksel   PORTA       ; select port data register bank 0
        clrf      PORTE       ; switch off all outputs
        goto      start       ; jump over subroutines to main loop

; Subroutine to wait about 0.8 ms ................................

del8    clrf      count       ; Load time delay of 256 x 3=768 us
again   decfsz    count       ; Decrement and test counter
        goto      again       ; until zero
        return                ;

; Subroutine to get analogue input ...............................
; Wait 20us ADC aquisition settling time ..
```

Program 8.1: Continued

```
getAD   movlw    007          ; Load time delay of 7 × 3=21 us
        movwf    count        ; Load counter
down    decfsz   count        ; Decrement and test counter
        goto     down          ; until zero (3us per loop)

; Get analogue input ..

        bsf      ADCON0, GO   ; Start A/D conversion
wait    btfsc    ADCON0,GO    ; Wait for conversion to complete
        goto     wait         ; by testing GO/DONE bit
        return                ; from subroutine with result in
                              ; ADRESH

; Subroutines to process keys ..................................

proc1   movlw    b'01000001'  ; Select analogue channel 1
        movwf    ADCON0       ; and
        call     getAD        ; and get analogue input
        return                ; for next key

proc2   movlw    b'01001001'  ; Select analogue channel 2
        movwf    ADCON0       ; and
        call     getAD        ; and get analogue input
        return                ; for next key

proc3   movlw    b'01010001'  ; Select analogue channel 3
        movwf    ADCON0       ; and
        call     getAD        ; and get analogue input
        return                ; for next key

proc4   movlw    b'01011001'  ; Select analogue channel 4
        movwf    ADCON0       ; and
        call     getAD        ; and get analogue input
        return                ; for next key

proc5   movlw    b'01000001'  ; Select analogue channel 1
        movwf    ADCON0       ; and
        call     getAD        ; and get analogue input
        return                ; for next key

proc6   movlw    b'01001001'  ; Select analogue channel 2
        movwf    ADCON0       ; and
        call     getAD        ; and get analogue input
        return                ; for next key

proc7   movlw    b'01010001'  ; Select analogue channel 3
        movwf    ADCON0       ; and
        call     getAD        ; and get analogue input
        return                ; for next key
```

Program 8.1: Continued

```
proc8   movlw     b'01011001' ; Select analogue channel 4
        movwf     ADCON0      ; and
        call      getAD       ; and get analogue input
        return                ; for next key

proc9   bsf       PORTB,0     ; Toggle buzzer on
        call      del8        ; delay about 0.8ms
        bcf       PORTB,0     ; Toggle buzzer off
        call      del8        ; delay about 0.8ms
        return                ; for next key

procs   bsf PORTE,0           ; switch on heater output
        return                ;

proc0   bsf PORTE,1           ; switch on vent output
        return                ;

proch   bsf PORTE,2           ; switch on fan output
        return                ;

; Routine to scan keyboard ....................................

scan    movlw     0FF         ; Deselect...
        movwf     PORTD       ; ...all rows on keypad

; scan row A of keypad ..........

        bcf       PORTD,3     ; select row A of keypad

        btfsc     PORTD,0     ; test key 1
        goto      key2        ; next if not pressed
        call      proc1       ; process key 1

key2    btfsc     PORTD,1     ; test key 2
        goto      key3        ; next if not pressed
        call      proc2       ; process key 2

key3    btfsc     PORTD,2     ; test key 2
        goto      key4        ; next if not pressed
        all       proc3       ; process key 3

; scan row B of keypad ..........

key4    bsf       PORTD,3     ; deselect row A
        bcf       PORTD,4     ; select row B

        btfsc     PORTD,0     ; test key 4
        goto      key5        ; next if not pressed
        call      proc4       ; process key 4
```

Program 8.1: Continued

```
key5    btfsc     PORTD,1     ; test key 5
        goto      key6        ; next if not pressed
        call      proc5       ; process key 5

key6    btfsc     PORTD,2     ; test key 6
        goto      key7        ; next if not pressed
        call      proc6       ; process key 6

; scan row C of keypad .........

key7    bsf       PORTD,4     ; deselect row B
        bcf       PORTD,5     ; select row C

        btfsc     PORTD,0     ; test key 4
        goto      key8        ; next if not pressed
        call      proc7       ; process key 4

key8    btfsc     PORTD,1     ; test key 5
        goto      key9        ; next if not pressed
        call      proc8       ; process key 2

key9    btfsc     PORTD,2     ; test key 2
        goto      keys        ; next if not pressed
        call      proc9       ; process key 3

; scan row D of keypad ......

keys    bsf       PORTD,5     ; deselect row C
        bcf       PORTD,6     ; select row D

        btfsc     PORTD,0     ; test key *
        goto      key0        ; next if not pressed
        call      procs       ; process key *

key0    btfsc     PORTD,1     ; test key 0
        goto      keyh        ; next if not pressed
        call      proc0       ; process key 0

keyh    btfsc     PORTD,2     ; test key #
        goto      done        ; next if not pressed
        call      proch       ; process key #
```

Program 8.1: Continued

```
; all done ..................
done    return    ; to main loop
; Main program   ************************************************

start   bcf       PORTB,2     ; switch off high digit of display
        bsf       PORTB,1     ; and switch on low digit
        call      scan        ; and read keypad
        movf      ADRESH,W    ; move ADC result
        movwf     PORTC       ; to display

        bcf       PORTB,1     ; switch off low digit of display
        bsf       PORTB,2     ; and switch on high digit
        call      scan        ; and read keypad

        movf      ADRESH,W    ; move ADC result
        movwf     PORTC       ; to display

        goto      start       ; repeat main loop

        END                   ; of source code ..................
```

Program 8.1: Continued

The routine to read in an analog input is based on the model routine provided in the data sheet, where a delay of about 20 μs is included to ensure that the input has had time to settle, in case the input is changing rapidly. The conversion is then started by setting the GO bit in the ADC control register, and then waiting for it to be cleared by the ADC to indicate that the conversion is complete. In this program, only 8 of the 10 bits of the ADC result are used, so the result is left justified to place the most significant 8 bits in the ADRESH register for output to the display. This allows the full range of the input (0–2.048 V) to be checked.

In a working program, the analog input value would be converted into a 2-digit decimal value for the display. Using the conversion scaling calculated above, a temperature of 50ºC would give a result of 250 (in binary) in ADRESL, with the result right justified. Only a quarter of the full ADC range is then being used. This result can then be converted into the corresponding display digits 5 and 0, and so on down to zero. The keyboard-scanning routine uses a simple method to check if each key in each row has been pressed, calling the required action if it has. A more elegant and compact keyboard-scanning method is possible when reading in numerical values.

A full working program would allow the user to enter the maximum and minimum values for the target temperature, and then go into run mode, where the temperature would be controlled within the set range by operation of the heater, vent and fan. Pseudocode for the application software is shown in Program 8.2.

```
TEMPCON
          Initialize
                    Ports
                              Port A = Temp sensor inputs (4)
                              Port B = Display digit select (2),
                                ICP/D (3)
                              Port C = Display segments (7)
                              Port D = Keypad (4 outputs, 3 inputs)
                              Port E = Heater, Vent, Fan outputs

                    ADC       Left justify, 4 channels
                              ADC frequency Fosc/8, select input AN0

          GetMaxMin
                    Scan keyboard
                    Store & display first digit of maxtemp
                    Scan keyboard
                    Store & display second digit of maxtemp
                    Convert to byte MaxTemp (0-200)

                    Scan keyboard
                    Store & display first digit of mintemp
                    Scan keyboard
                    Store & display second digit of mintemp
                    Convert to byte MinTemp

          Cycle

                    Read tempsensor1
                    Read tempsensor2
                    Read tempsensor3
                    Read tempsensor4

                    IF sensor out of range
                              replace with previous value
                    Calculate AverageTemp

                    Display AverageTemp
                              MSD = AverageTemp/10
                              Get 7-seg code & display MSD
                              LSD = Remainder
                              Get 7-seg code & display LSD
                    IF AverageTemp > Mintemp
                              switch heater OFF
                              ELSE switch heater ON
```

Program 8.2: Pseudocode for TEMPCON Control Software

```
                IF AverageTemp > Maxtemp
                        switch vent ON
                        ELSE switch vent OFF
                IF AverageTemp > Maxtemp + 4
                        switch fan ON
                        ELSE switch fan OFF
GOTO Cycle
```

Program 8.2: Continued

8.1.6 Application Modifications

There are some features of the microcontroller that are not yet utilized in this application, which could enhance it. As mentioned above, the PWM module could be used to control the speed of the fan. In addition, a serial communication port could send the temperature data to a master controller, and receive new operating parameters. If a PC were acting as the host, the USART could be used, and the PC could display the operating data, perhaps as in graphical form, or as a plot of temperature variation over time. This data could then be saved on disk, and sent via a network to a supervisory system.

When designing the application initially, a top of the range device such as the 16F877 is a good choice, as it has most of the available features. When the design has been finalized, it may turn out that some features are not required, some I/O is unused or the program could be fitted into a smaller memory. The designer can then review the alternative, cheaper or otherwise more suitable devices, and transfer the application to that device as long as the hardware redesign required is not excessive. The software reconfiguration should also not be too much of a problem within the same PIC group, since the chips are designed to be interchangeable. This idea is illustrated below, where the temperature controller is redesigned for other PIC chips.

8.2 16F818 Application

The PIC 16F818 is a replacement part for the 16F84. It has a compatible pin-out, and additional features at a lower cost. Sixteen I/O pins are available, including five analog inputs. It has 1k words of program memory; if extra memory is needed, the 16F819 has the same features but 2k program memory.

As can be seen from the pin-out (Fig. 8.8), each pin has multiple functions, other than the two supply pins. Analog inputs can be selected on RA0–RA4, or external reference voltages. There is a CCP module and a synchronous serial port offering SPI or I^2C modes. Other special features are a variety of power saving modes in addition to the usual sleep, an internal oscillator which obviates the need for external clock components, and in-circuit programming and debugging.

RA2/AN2/Vref–	1	18	RA!/AN1
RA3/AN3/Vref+	2	17	RA0/AN0
RA4/AN4/T0CKI	3	16	RA7/OSC/CLKI
RA5/!MCLR/Vpp	4	15	RA6/OSC2/CLKO
Vss	5	14	Vdd
RB0/INT	6	13	RB7/T1OSI/PGD
RB1/SDO/CCP1	7	12	RB6/T1OSOT1CKI/PGC
RB2/SDO/CCP1	8	11	RB5/!SS
RB3/CCP1/PGM	9	10	RB4/SCK/SCL

Figure 8.8: The PIC 16F818 Pin-Out

Thus, many of the features of the more powerful 16F87X group are now available in the smaller 18-pin package. It is recommended that, when the user is familiar with all the options available on this chip, it can be used as a default choice when developing new PIC applications, if the number of I/O is sufficient.

This chip could be used in the temperature controller if the keyboard were eliminated, and the set temperature input from a pot via one of the analog inputs (see Fig. 8.9). A fixed control range might be necessary, as there would be no facility for entering maximum and minimum temperatures. The display digit selection can be reconfigured to use only one output. The application then only needs 16 I/O pins. Operational parameters could be transferred via the serial interface if the display were left out (RB1, RB2 and RB4).

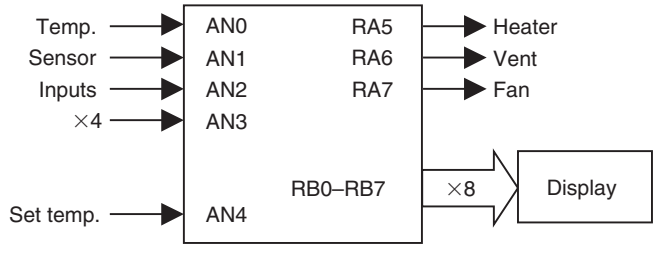


Figure 8.9: PIC 16F818 Temperature Control Block Diagram

8.3 12F675 Application

The 12-series of PIC mini-chips offers flash memory in 8-pin packages. At the current time there is limited choice, but no doubt the range will be expanded. The pin-out for the 12F675 illustrates the I/O features available (see Fig. 8.10).

The chip can be configured with six plain digital I/O pins, but also offers two timers, an analog comparator or four analog input channels. The 12F629 is the same, except that it does not include the ADC and is therefore a little cheaper. An internal oscillator and in-circuit programming are also featured.

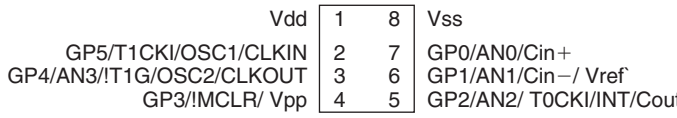


Figure 8.10: The PIC 12F675 Pin-Out

A temperature controller could be implemented using this chip if only two analog inputs are used (see Fig. 8.11). It could operate with a fixed set temperature, or another analog input could be used as a set temperature input. With no display, a dial on the set temperature pot may be necessary.

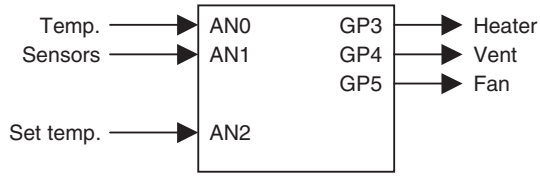


Figure 8.11: PIC 12F675 Temperature Controller

8.4 18F452 Application

The 18-series of PIC microcontrollers is the most powerful in the flash family, ranging from the 16 I/O, 4k memory 18F1220 to the 68 I/O, 64k memory 18F8720 at the current time. The group offers different combinations of the advanced features, and the larger memory size means that 'C' can be used for application programming. The instruction set of 75 16-bit instructions is designed with this in mind.

A small selection of the available 18FXXXX devices are listed in Table 8.5. At the time of writing there are a total of 33 in production, with several others listed as future products. The architecture is somewhat more complex than the 14-bit devices, with extra blocks for multiplication, a hardware data table access, additional file select registers and other advanced features. The data bus is still 8 bit. Taking the 18F452 as an example, in terms of peripheral features it is comparable to the 16F877 described in Section 8.1, so a comparison of the two devices will be made to illustrate the differences and similarities of the two groups (Table 8.6).

As can be seen, the 18-series device has some advantages: 40-MHz clock rate, 16k program memory and more data memory. However, bear in mind that a program written in C will not be as code-efficient as an assembly language equivalent, so these advantages may or may not translate into extra performance, depending on the application and the way that it is structured. The main advantage is that more complex operations such as mathematical functions are easier to program in C. The 18-series PIC has a richer instruction set, including instructions

Table 8.5: PIC Flash Microcontroller Features

PIC device number	Total pins	I/O pins	Program ROM words	File RAM bytes	EEPROM bytes	Analogue inputs	Timers 8 16 bit bit	Max. clock (MHz)	Internal osc. (MHz)	In-circuit debug	CCP/ PWM modules	Serial comms	Relative cost
12F629	8	6	1k	64	128	–	1 + 1	20	4	✓	–	–	1.02
12F675	8	6	1k	64	128	4 × 10-bit	1 + 1	20	4	✓	–	–	1.26
16F627A	18	16	1k	224	128	–	2 + 1	20	4	–	–	UART	1.49
16F628A	18	16	2k	224	128	–	2 + 1	20	4	–	1	UART	1.70
16F630	14	12	1k	64	128	–	1 + 1	20	4	✓	–	–	1.20
16F648A	18	16	4k	256	256	–	2 + 1	20	4	–	–	UART	1.83
16F676	14	12	1k	64	128	8 × 10-bit	1 + 1	20	4	✓	–	UART	1.38
16F72	28	22	2k	128	–	4 × 8-bit	2 + 1	20	–	–	1	–	2.10
16F73	28	22	4k	192	–	5 × 8-bit	2 + 1	20	–	–	2	All	3.27
16F74	40	33	4k	192	–	8 × 8-bit	2 + 1	20	–	–	2	All	3.97
16F76	28	22	8k	368	–	5 × 8-bit	2 + 1	20	–	–	2	All	4.10
16F77	40	33	8k	368		8 × 8-bit	2 + 1	20	–	–	2	All	4.58
16F818	18	16	1k	128	128	5 × 10-bit	2 + 1	20	8	✓	1	I^2C, SPI	1.71
16F819	18	16	2k	256	256	5 × 10-bit	2 + 1	20	8	✓	1	I^1C, SPI	1.71
16F84	18	13	1k	64	64	–	1	10	–	–	–	–	4.39
16F84A	18	13	1k	64	64	–	1	20	–	–	–	–	3.42
16F87	18	16	4k	398	256	–	2 + 1	20	8	✓	1	All	2.26
16F88	18	16	4k	368	256	7 × 10-bit	2 + 1	20	8	✓	1	All	2.41
16F873A	28	22	4k	192	128	5 × 10-bit	2 + 1	20	–	✓	2	All	3.98
16F874A	40	33	4k	192	128	8 × 10-bit	2 + 1	20	–	✓	2	All	4.35
16F876A	28	22	8k	256	368	5 × 10-bit	2 + 1	20	–	✓	2	All	4.28
16F877A	40	33	8k	256	368	8 × 10-bit	2 + 1	20	–	✓	2	All	4.68
18F1220	18	16	2k	256	256	7 × 10-bit	1 + 3	40	8	✓	1	UART	2.78
18F2320	28	25	4k	512	256	10 × 10-bit	1 + 3	40	8	✓	1	All	4.85
18F4320	40	36	4k	512	256	13 × 10-bit	1 + 3	40	8	✓	2	All	5.29
18F6520	64	52	16k	2048	1024	12 × 10-bit	1 + 3	40	–	✓	5	All	6.52
18F8621	80	68	32k	3840	1024	16 × 8-bit	1 + 3	40	10	✓	14	I^2C, SPI	8.25
18F8720	80	68	64k	3840	1024	16 × 10-bit	1 + 3	40	–	✓	5	All	10.90

Table 8.6: Comparison of the 16F877 and the 18F458

Feature	16F877	18F458
Total pins	40	40
Input/output pins	33	33
Ports	A, B, C, D, E,	A, B, C, D, E
Clock	20 MHz	40 MHz
Instruction bits	14	16
Program memory (instructions)	8k	16k
Instruction set size	35	75
Data memory (bytes)	368	1536
EEPROM (bytes)	256	256
Interrupt sources	14	21
Timers	3	4
Capture, compare, PWM modules	2	2
Serial communications	MSSP, USART	MSSP, USART, CAN
Parallel port	Yes	Yes
Analog inputs	8 × 10 bits	8 × 10 bits
Resets	POR, BOR	POR, BOR, Stack, Programmed
In-circuit serial programming and debugging	Yes	Yes

such as multiply, compare and skip, table read, conditional branch and move directly between registers, so still has advantages even when programmed in assembly language.

8.4.1 PIC C Programming

For those readers unfamiliar with C programming, a simple example is shown in Program 8.3. The program will give the same output as BIN1.ASM assembly language program. The program must be converted to PIC 16-bit machine code using a compiler such as MPLAB C18 Compiler, which is supplied as an add-on to the development system. This compiler recognizes ANSI C, the standard syntax for microcontrollers (ANSI = American National Standards Institute). The C compiler must be selected in the development mode dialogue when building the application.

The main elements of the program functioning are as follows.

```
/* comment */
```

Comments in C source code are enclosed between /* and */, and can be run over several lines.

```
#include<p18F456.h>
```

```
/*    BIN1.C          M Bates          Version 1.0

      Program to output a binary count to Port B LEDs

*****************************************************************/

#include <p18f458.h>          /* Include port labels for this chip */
#include <delays.h>

int counter                   /* Label a 16-bit variable location  */

void main(void)               /* Start main program sequence       */
{
     counter = 0;             /* Initialize variable value          */
     TRISB = 0;               /* Configure Port B for output        */
     while (1)                /* Start an endless loop              */
     {
          PORTB = counter;    /* Output value of the variable       */
            Counter++;        /* Increment the variable value       */
          Delay10KTCY(100);   /* Wait for 100 x 10,000 cycles       */
     }

}                             /* End of program                     */
```

Program 8.3: A Simple PIC C Program

This is a compiler directive which calls up a header file named "p18F458.h". This contains predefined register labels for that particular processor, such as TRISB and PORTB, and the corresponding addresses 06h and 86h.

```
int counter;
```

This assigns a label to a register and declares that it will store an integer, or whole number. A standard integer in C is stored as a 16-bit number, requiring two data RAM (GPR) locations.

```
void main(void)
```

This rather peculiar syntax simply indicates, as far as we are concerned here, the start of the main program sequence. The following brace (curly bracket) encloses the main program with a matching brace at the end. These are lined up in the same column and the main program tabbed in between them, so that they can be matched up correctly.

```
counter = 1;
```

A value of 1 is initially placed in the variable location (low byte).

```
TRISB = 0;
```

A value 0 is loaded into the data direction register of Port B to initialise the port bits for output to the LEDs.

```
while(1)
```

This starts a loop that will run endlessly. A condition is placed in the brackets that controls the loop. For example, the statement could read "while(count<256)", in which case the following group of statements within the curly brackets (braces) would execute 255 times, counting up to the maximum binary value and stopping. The value 1 means the condition is "always true", so the loop is endless, until reset.

```
PORTB = counter;
```

The value in counter is copied to Port B data register for display on the LEDs

```
counter++;
```

The variable value is incremented each time the loop is executed. This causes the output to be incremented the next time.

```
Delay10KTCY(100);
```

This calls a predefined block of code that provides a delay, so that the LED output changes are visible. At a maximum clock rate, the processor instruction cycle time is 0.1 μs, so the delay works out to 0.1 s (10000 × 100 cycles). The overall count cycle will then take 25.6 s. This function will require initialization and loading of hardware timers and associated operations which will clearly be quite complex in machine code.

The delay function is an example of a function call, which is one of the biggest advantages of C—the collection of standard routines, which are automatically available, means that the programmer does not have to keep "re-inventing the wheel," or even invent it for the first time; it is ready-made.

The layout of the program, with tabs, is important for understanding the program and checking the syntax if there are logical errors. However, the layout does not affect the program function, only the sequence of characters. However, the statements must be all on one line; line returns are not allowed within a statement.

Each complete statement is terminated with a semicolon; note that some are not complete in themselves and do not have a semicolon. For example, "while(1)" is not complete without the loop statements, or at least the pair of braces. The close brace terminates the "while" statement. The whole of the main loop, and any functional subblock, must be enclosed between braces.

8.4.2 Advantages of C Programming

The C compiler converts the program into PIC 16-bit machine code. Most of these C statements translate into more than one machine code instruction. This can be confirmed by studying the list file produced by disassembling the machine code.

The pseudocode for the temperature controller above can probably be more easily translated into C than assembly language. For example, the conditional control operations defined using IF. . .THEN statements will translate directly, whereas, in assembler, it has to be implemented by suitable combinations of "Bit Test and Skip" with "Goto" or "Call". In addition, the comparison of the "average temperature" with the set values can be done in one statement in C, but needs a subtract or compare prior to a bit test, which is much more complicated. On the other hand, checking bit inputs is not so easy in C as in assembler, as ANSI C contains no individual bit operations. Bit status in a register has to be checked by using a logical or numerical range check.

There are many references on C programming. To program a microcontroller in C, only the basic set of statements and simple data structures will probably be needed, so if the reader has some knowledge of C already, using it to develop PIC applications should not be too difficult. However, a full treatment will not be attempted here. We can now see the advantages of using C with the 18-series PIC. The chips themselves have a good range of peripheral interfaces and other features, and can be programmed more easily using the high level language. For more details on programming PICs in C, refer to Chapters 24–29.

8.5 Summary

- The PIC 16F877 has a good range of peripheral interfaces, including analog inputs, serial ports, CCP and PWM, and in-circuit debugging.
- The application designed around the PIC 16F877 operates as a temperature controller, with four sensors, three outputs, a keypad and 2-digit display.
- The temperature controller can be programmed to maintain the temperature in a heating/cooling system within a set range, display these parameters and operate alarms.
- A similar application can be implemented using the 16F818 without the keypad.
- A similar application can be implemented using the 12F675 without the display.
- The 18F458 has comparable I/O features to the 16F877, but can be programmed in "C" and runs at twice the speed.

CHAPTER 9

The PIC12F50x Series (8-pin PIC Microcontrollers)

There is a range of PIC microcontrollers that manages to squeeze a large number of features into a tiny 8-pin package. The 8-pin device most like the PIC16F54 is the PIC12F508 (the 12 in the name tells us that this is an 8-pin device). Surprisingly, this little PIC microcontroller offers up to 6 I/O pins (the other two are power supply pins). It needs no external oscillator (e.g., crystal or RC), as it has an in-built 4-MHz oscillator, and even offers a feature that allows external signals to wake it up from the sleep state. For any application where a small size is advantageous and 6 I/O pins are sufficient, these PIC microcontrollers are invaluable.

The PIC12F50x series consists of two models (the PIC12F5**08** and PIC12F5**09**) shown in Fig. 9.1, with a third model (the PIC12F5**10**) under development at the time of publication. The 'F509 has more memory (more program memory, and more GPFs) than the 'F508. The 'F510 will be similar to the 'F509 but with the added feature of built-in analog-to-digital conversion (this is discussed further in Chapter 10).

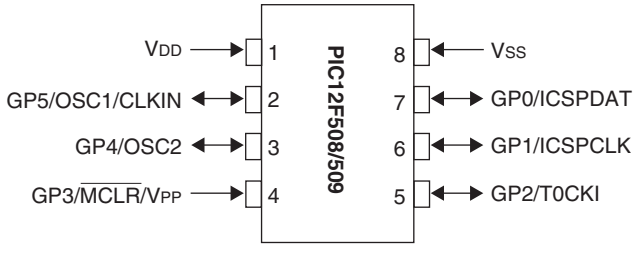


Figure 9.1

9.1 Differences from the PIC16F54

There are a few differences in the way these PIC microcontrollers work, most of which are illustrated in the file registers. Figure 9.2 shows the file register arrangement for the PIC12F50x series.

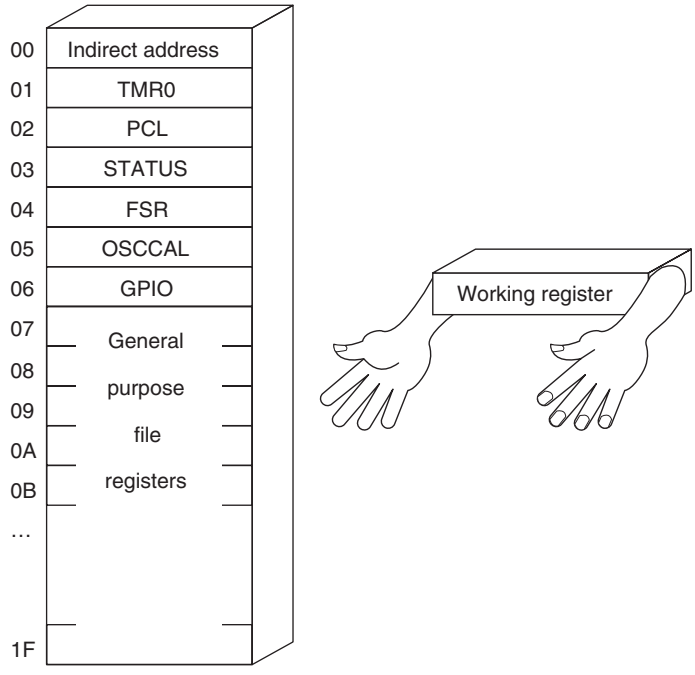


Figure 9.2: Map of File Registers for PIC12F508

9.1.1 The STATUS Register

The first difference is found in the STATUS register. This PIC series offers the option of waking up from sleep if one of three I/O pins changes state (GP0, GP1 or GP3). The previously unused bit 7 of the STATUS register can now be used to see whether the PIC microcontroller was woken up from sleep due to one of these pins changing state (bit 7 is *set*), or whether it was some other reason (bit 7 is *cleared*).

9.1.2 The OSCCAL Register

The second difference you will notice is that there is a new file register at address **05**, the **OSCCAL** file register. This is used for **osc**illator **cal**ibration, and is really only used at the start of your program (address **0x000**). To make the internal 4 MHz internal oscillator more accurate, a special number should be moved into the **OSCCAL** register. As with the PIC16F5x series, the PIC processor first executes the instruction at the last address of the program memory (1FFh for 'F508, and 3FFh for 'F509). However, when the PIC microcontrollers are made in the factory, a special instruction is programmed into them at the last address. This instruction moves a particular number (the calibration value) into the working register, i.e. it takes the form:

```
movlw   xx   ; moves calibration value into w. reg
```

After executing this instruction, the program loops back and starts at address **0x000** – remember, all this happens automatically. (By the way, if you are erasing a new PIC microcontroller, you should first read the program memory and make a note of the factory-programmed value, so that you can insert this line yourself.) If you wish to make the internal oscillator more accurate, the instruction at address **0x000** should be:

```
movwf OSCCAL    ; uses the pre-programmed value
                ;   to calibrate the internal oscillator
```

If you are not interested in oscillator accuracy, you can omit this instruction and simply place **goto start** at program at address **0x000** using the **org** command. The program template used previously should be modified as follows:

```
; Program Description: _____________________________________________
; ___________________________________________________________________
      list       P=12F50x
      include    "c: pic p12f50x.inc"

; ============
; Declarations:

porta   equ      05
portb   equ      06
        org      0              ; first instruction to be executed
        movwf    OSCCAL         ; calibrates oscillator
        goto     Start          ;

;============
; Subroutines:

Init    clrf     GPIO           ; resets input/output port
        movlw    b'xxxxxx'      ; sets up which pins are inputs
        tris     GPIO           ;   and which are outputs

        movlw    b'xxxxxxxx'    ; sets up timer and some pin
        option                  ; settings
        retlw    0              ;

;============
; Program Start
```

9.1.3 Inputs and Outputs

The PIC12F50x series has only one I/O port called the **GPIO** (the **g**eneral **p**urpose **i**nput/**o**utput file register). It works in exactly the same way as Port A and Port B on the PIC54—certain pins on the PIC microcontroller correspond to bits in this file register. One important thing to note is that *GP3 is in fact only an INPUT*, and cannot be configured as an output.

9.1.4 The OPTION Register

As previously mentioned, the PIC microcontroller can be configured to wake up from sleep when one of GP0, GP1 or GP3 changes state. This is controlled by bit 7 of the **OPTION** register—the feature is *enabled* when bit 7 is *clear*, and *disabled* when bit 7 is *set*.

Bit 6 of **OPTION** has also been given a purpose (you may remember that these two bits were unused in the PIC54 and 57). When set, the PIC microcontroller will make pins GP0, GP1 and GP3 *float* high when not connected to anything. These are known as *weak pull-ups*. These are useful when the pins are being used as inputs which are pulled low when something happens (e.g. you've attached a push button between the pin and 0V, pulling the input low when the button is pressed). If you enable the *pull-ups* on the PIC microcontroller, you don't need an external pull-up resistor. If you don't want to use this feature then make sure you set this bit.

Note that both of these features require bits to be *set* in order to disable the feature—don't forget to do this! The rest of the **OPTION** register is as in the PIC54.

9.1.5 The TRIS Register

Nothing much is new in this file register. Just remember that there are now 6 bits in the I/O file register, and the number you use to select inputs and outputs should reflect this. Also remember that GP3 cannot be configured as an output. Finally, note that GP2 is also the T0CKI pin. This means that if the TMR0 is configured (in **OPTION**) to count signals from the T0CKI pin, GP2 is automatically set to be an input, overriding the value of the bit in the **TRIS** register.

9.1.6 The General Purpose File Registers

The PIC12F508 is identical to the PIC54 in terms of GPFs. The PIC12F509 has an extra set at addresses 30-3Fh in the data memory. These are accessed in the same way as described for the PIC57 – by setting bit 5 of the **FSR** (see page 82).

9.1.7 The $\overline{MCLR}$

The PIC12F50x series still has an $\overline{MCLR}$ pin, but if you don't need a reset pin, it can be used as an input pin (GP3). You can enable or disable the $\overline{MCLR}$ when programming the PIC microcontroller (it is one of the configuration bits). In MPLab, select "Internal" to disable the $\overline{MCLR}$, or use the **__config** command.

9.1.8 Configuration Bits

There are some new configuration options relating to the ability to disable the MCLR feature, and the use of the internal oscillator. Use **_MCLRE_OFF** or **_MCLRE_ON** to disable/

enable the MCLR feature. The four allowed oscillator options are **_LP_OSC_**, **_XT_OSC**, **_IntRC_OSC** and **_ExtRC_OSC**, where the latter two options refer to the internal RC oscillator and external RC oscillator, respectively. An example configuration command would be:

```
__config _MCLRE_OFF & _IntRC_OSC & _CP_OFF & _WDT_OFF
```

9.2 Example Project: PIC Dice

Our example project to demonstrate the PIC12F508 will be a pair of dice, with fourteen LEDs and one button. The LEDs will be arranged as shown in Fig. 9.3. When the button is pressed, the LEDs will flash randomly, and when it is released, the LEDs gradually slow down until they finally display a pair of numbers (in the traditional dice format). It will display this number for 5 seconds, then go to sleep.

The PIC12F508 supports up to five outputs, so controlling fourteen LEDs is going to be a real challenge! Looking at Fig. 9.3, we notice that we don't need individual control over each LED die in order to display a number (1–6). Instead, we can split these into four groups of LEDs which I've labeled A, B, C and D. This cuts the requirement down to 8 outputs (4 per die). Finally, we can use one output to select which die is on—if the output is 0, the left die is on, and if the output is 1, the right die is on. This means we can get away with 5 outputs (1 controller, and 4 for the LEDs). The button will be connected to GP3, which will be set to wake the PIC microcontroller up from sleep. The program flowchart is shown in Fig. 9.4, and the circuit diagram in Fig. 9.5. As you can see from Figure 9.1, if you wish to use in-circuit serial programming, the ICSPDAT line should be connected to GP0, and the ICSPCLK line to GP1, at the programming stage. However, these pins should be disconnected from the ICSP lines during circuit operation.

In **Init** we should set up the inputs and outputs (all outputs, except GP3 which is the button). We then need to turn off all the LEDs. Looking at Fig. 9.3, we see that one die's LEDs are

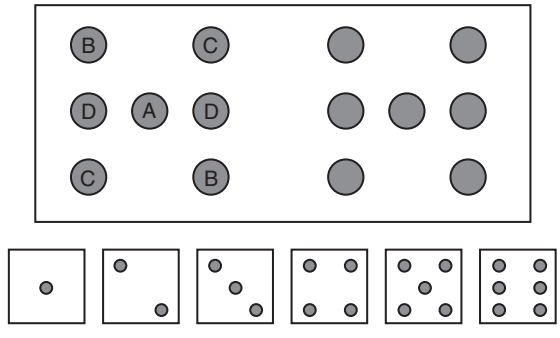


Figure 9.3

on when their corresponding GPIO bits are 1, and the other die's LEDs are on when their corresponding GPIO bits are 0 (i.e., one has common cathode, and one common anode). Therefore to turn the LEDs off, we move **b'100000'** into GPIO. Setting Bit 5 selects the common anode group of LEDs, and so the other GPIO bits should be cleared to turn off the LEDs. Finally, set up the OPTION register with TMR0 prescaled by the maximum amount, weak pull-ups disabled, the wake-up featured enabled on pins GP0, 1 and 3.

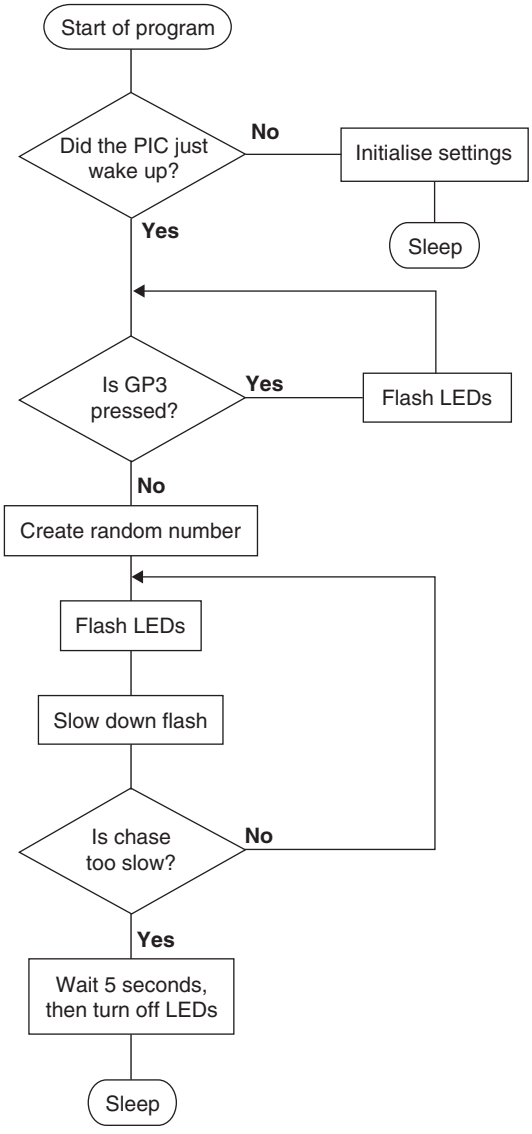


Figure 9.4

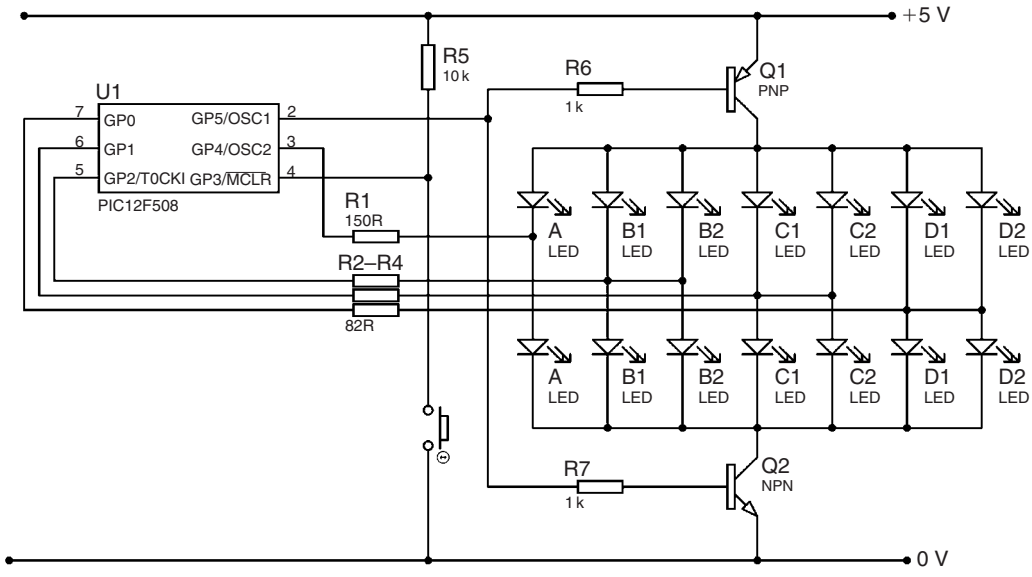


Figure 9.5

There are three main loops in the main section of the program. In the first, we are waiting for the button to be released, the displays are randomly flashing, and a random number is being selected. In the second, when the button is released, the displays slow down until a critical point is reached. Finally, the random number is displayed, and we wait for 5 seconds before going back to sleep.

9.2.1 Random Digression

There are two approaches to generating random numbers: we can use some user input (e.g., the length of time a button is pressed) or another external component, or alternatively we can use an algorithm to generate a pseudo-random number. For example, if we increment a register continually (and very quickly) during a loop in which we wait for a button to be released, the register will be overflowing constantly and will end up at a random value. If we don't have the luxury of an external input, there are methods ranging in complexity for generating random numbers. A simple algorithm is the Linear Congruential Method developed by Lehmer in 1948, and has the following form:

$$I_{n+1} = \mathrm{mod}_m(aI_n + c)$$

This generates the next number in the sequence by multiplying the previous number by a, adding c, and taking the result modulo m. Here $\mathrm{mod}_m(x)$ is equal to the remainder left when you divide x by m. Conveniently, the result of every operation performed in a PIC program is effectively given modulo 256. For example, we add 20 to 250. The "real" answer is 270,

however, the result given in a PIC program is 14. 14 *is* "270 modulo 256" or $\text{mod}_{256}(270)$. There are a number of restrictions on the choice of a and c in the above equation that maximize the randomness of the sequence. For example, we could pick $a = 3$ and $c = 63$. You also have to pick a "seed"—the first number in the sequence (I_0). You can set up this model on a spreadsheet and examine its *quasirandom* properties. First, you should notice that the randomness of the sequence does not appear to be sensitive to the seed. You should also observe that the sequence repeats itself every 256 numbers—this is an unfortunate consequence of the algorithm, but picking a larger modulus will increase the period accordingly.

In this example project, we will use the first method (increment quickly while a button is pressed) to pick the final random number for the dice. However, for the random flashing that occurs prior to the answer being displayed, we will use the algorithm given above. The program begins:

```
Start    call    Init           ; initialization procedure

Pressed  btfsc   GPIO, 3        ; tests button
         goto    Released       ; branches when released
         call    RandomScroll   ; quickly increments numbers
         call    Timing         ; keeps flashing going
         call    Display        ; keeps displays changing
         goto    Pressed        ;
```

In this loop we wait for the button to be released. In the **RandomScroll** subroutine, the dice result (a number between 0 and 35) is incremented. This number is stored over two file registers called **Ran1** and **Ran2** which each hold a number between 0 and 5.

The **Timing** subroutine creates the delay between the displays changing. When the button is released, this delay increases so that the dice slow down. The basic unit of time will be 1/50th of a second (hence for a 4 MHz oscillator and TMR0 prescaled by 256, we use a marker of 78). The postscaler will be set to 4 while the button is pressed (corresponding to the displays changing at a rate of about 12 times per second). When the button is released, we will set a bit called **slow**, which will tell the **Timing** subroutine to increment the postscaler up to a maximum of 31 (i.e., over the course of about 10 seconds it will slow down to a rate of about 1 per second). You can play with these values to create the type of behavior you desire. This subroutine starts as follows:

```
Timing   movfw    Mark78       ; base unit = 1/50th second
         subwf    TMR0, w      ;
         btfss    STATUS, Z    ;
         retlw    0            ;

         movlw    d'78'        ; resets marker
         addwf    Mark78, f    ;

         decfsz   PostX, f     ; variable postscaler
         retlw    0            ;
```

PostX is the variable postscaler that is reset with a value given by **PostVal**. Thus, to slow down the flashing, we increment **PostVal**. At the point following the above code, the variable length delay has elapsed and we need to change the display values. We have a file register called **Random** containing a random number between 0 and 255, which is generated using the algorithm given above: $\text{Random}_{n+1} = \text{mod}_{256}(3\ \text{Random}_{n+63})$. This is generated by calling the subroutine **RandomGen**.

This random number then needs to be changed into a number between 0 and 7 (as well as displaying numbers 1–6, "all-on" and "all-off" will be options during the random flashing). This is best done as follows:

```
swapf     Random, w      ;
andlw     b'00000111'    ; converts to 0-7 and moves
movwf     Die1num        ;   into Die1num
```

The file registers **Die1num** and **Die2num** will be used to hold the number to be displayed on the corresponding set of LEDs. Note that we do *not* simply take the 3 least significant bits of Random, as this leads to a periodicity of 8 in the random flashing, which will be very noticeable. By taking bits 4 to 6 of Random we get a period of 128, which will be much harder to spot. We use a similar set of four lines to move a random number into **Die2num**.

We then test the bit called **slow**, and call a subroutine named **Slowdown** if it is set (remember to clear it in **Init**).

Finally, the variable postscaler **PostX** is reset with the value in **PostVal**, and we return from the subroutine.

In the display subroutine, we handle the strobing of the two sets of LEDs. We use TMR0 to control strobing (in particular, bit 4 of TMR0). We'll need two look-up tables to take the number to be displayed (a number between 0 and 7 stored in **Die1num** and **Die2num**) and return the appropriate code for GPIO. "0" will correspond to all LEDs off, "1–6" correspond to the images shown in Fig. 9.3, and "7" corresponds to all LEDs on.

When the button is released, we jump to the **Released** section. The loop is much the same, with the exception that the **slow** bit is set, and we test for PostVal to reach 0 before skipping out of the loop:

```
Released    bsf      slow          ; tells Timing to slow down
            call     Timing        ; handles variable delays
            call     Display       ; updates displays
            movf     PostVal, f    ; has PostVal been cleared?
            btfss    STATUS, Z     ;
            goto     Released+1    ;
```

At this point, the numbers from **Ran1** and **Ran2** are incremented and moved into **Die1num** and **Die2num**, respectively. In the final loop we display the result for 5 seconds (which we create using **Mark78** and a postscaler of 250).

When GP3 changes again (i.e., the button is pressed), the PIC microcontroller will wake up, so the **sleep** command needs to be followed with the line **goto Start**.

This completes the dice project, which gives an example of what can be achieved on the tiny 8-pin PIC microcontrollers. The full program is shown in Appendix 4; however, note that the display codes used are dependent on how you wire up the LEDs in your circuit board, and these may not necessarily match my values. A nice extension of this project would be to change the time at which the two dice finish "rolling," such that one finishes before the other, to create a greater air of suspense. You may also need to add some element of debouncing, depending on the type of button you use.

CHAPTER 10

Intermediate Operations Using the PIC12F675

Studying devices such as the baseline PIC5x series (by which I mean PIC16F5x and PIC12F50x chips) allows us to learn about the basics behind PIC programming. The simplicity and low cost of these entry-level devices are definite advantages; however, this also means they lack some useful features. These features include *analog to digital conversion* (measuring an analog voltage), *interrupts* (which save having to test inputs manually), and an *EEPROM* (a bank of data that stays intact even when you remove power). These features are all found on a rather handy little 8-pin device called the PIC12F675. It is worth noting that this is a more "typical" kind of PIC microcontroller (rather than the simple PIC5x series) and so if you come across a new PIC microcontroller it is more likely to behave like this one. If you decide that 6 I/O pins are too few, there is a 14-pin version called the PIC16F676, which is essentially identical to the PIC12F675 but has 12 I/O pins.

Looking at the pin layout of the PIC12F675 in Fig. 10.1, you should notice similarities and differences between it and the PIC12F508 of the previous chapter. You will also see that some

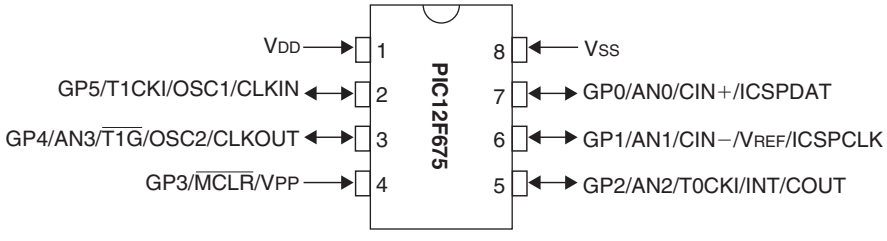


Figure 10.1

of the pins are labelled AN0, AN1, AN2 and AN3: these can be made analog inputs. VREF (pin 6) can be made the voltage reference for the other analog inputs (i.e., the PIC microcontroller compares the voltage at the other pins with the voltage on the VREF pin). INT (pin 5) can be set to interrupt normal program flow when it goes high or low. The pins labeled CIN, CIN and COUT are part of a comparator module. A comparator compares the voltage on two inputs, and tells you which one is greater. Finally, this PIC microcontroller has not one timer but two! The

second is called TMR1 (in addition to the TMR0 we have been using). The pins labeled T1CKI and T1G are associated with this second timer.

Due to the compact nature of the PIC12F675, many of these different pin functions are squeezed onto the same pins and we often have to make a choice of which particular function we wish to use. On larger models these features are spread over more pins and we have more choice over which ones can be used at the same time. Each of the pins described above, and their associated features, will be covered in detail in this chapter.

10.1 The Inner Differences

Having looked at the external differences, we now need to examine the inside of this PIC microcontroller. Figure 10.2 shows the arrangement of file registers on the PIC12F675. The first thing to observe is that all the extra features bring with them a load of extra special function registers (SFRs). Do not be overwhelmed by the large quantity of these SFRs – we will go over each one in due course. The greyed file registers are unused areas of the data memory. If you try reading the values in these locations, you will get a 0.

The second thing you might notice is that there are two *banks*. Whereas the PIC16F54 had only one bank ("filing cabinet"), the PIC12F675 has two sets of file registers. You should also take note that some file registers are the same in Bank 1 as in Bank 0. Think of a bank as a "frame of mind" of the PIC microcontroller, where file registers may (or may not) be different depending on the "frame of mind." File register 03 will always be the STATUS register, regardless of the "frame of mind" the PIC microcontroller is in. However, in Bank 0, file register 05 will be GPIO, and in Bank 1 file register 05 actually corresponds to a file register called TRISIO. Even if I actually write "GPIO" in the program, the PIC microcontroller will still act on TRISIO, if it is in the Bank 1 "frame of mind."

To switch from one bank to another we use one of the bits in the STATUS register (now you see why STATUS must be the same in both banks—if it didn't exist in Bank 1 there would be no way of getting back to Bank 0!). This bit is called RP0 and is bit number 5. To go to Bank 1, we set the bit. To return to Bank 0 we clear it.

Example 10.1 We want to clear the file register called TRISIO, however, the PIC microcontroller is currently in Bank 0.

```
bsf     STATUS, RP0     ; goes to Bank 1
clrf    TRISIO          ; clears the TRISIO register
bcf     STATUS, RP0     ; goes to Bank 0
```

Note that the following performs the same task:

```
bsf     STATUS, RP0     ; goes to Bank 1
clrf    GPIO            ; clears the TRISIO register
bcf     STATUS, RP0     ; goes to Bank 0
```

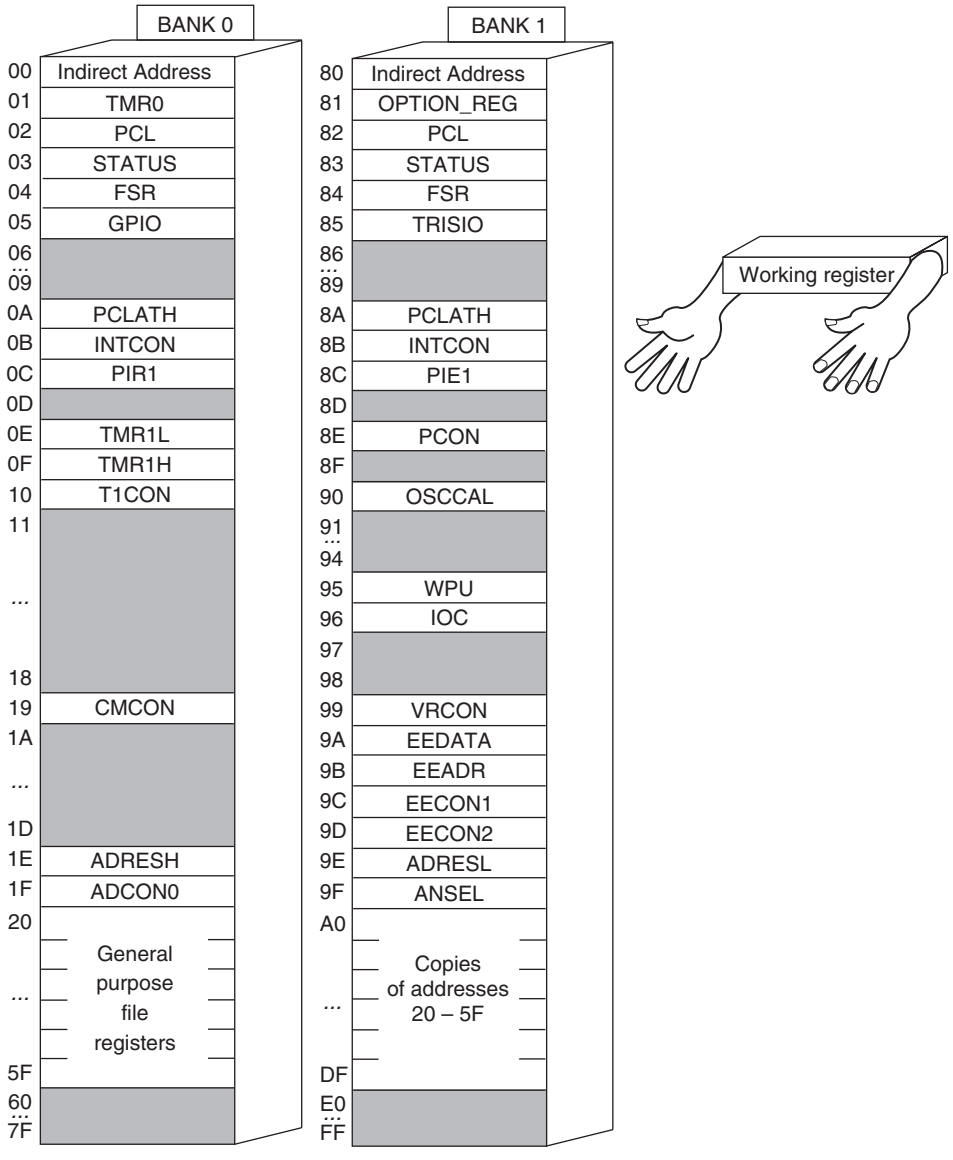


Figure 10.2

Naturally writing "TRISIO" makes far more sense, but the point is to highlight the fact that if you try to do something to GPIO when in Bank 1, you will actually do it to TRISIO.

In many cases, a Bank 1 file register is in some way related to its Bank 0 counterpart (e.g., the OPTION_REG register is largely a setup register for TMR0). Because the Bank 1 file registers tend to be involved in setting up, you may only need to go into Bank 1 during the **Init** subroutine. Finally, please note that the PIC microcontroller starts up in Bank 0.

10.1.1 The OPTION and WPU Registers

From the top, the first new file register we come across is the OPTION_REG register. It isn't strictly a new file register, because there was an OPTION register on the PIC5x chips, but we did not have direct access to it. Remember how to move a number into the OPTION register (e.g., in order to set up TMR0) with the PIC5x? Below is a reminder:

```
movlw     b'xxxxxxxx'    ; moves the number into w. reg
option                   ; moves w. reg into OPTION
```

With most other PIC microcontrollers (including the PIC12F675) there is no need for this **option** instruction as we can simply move the number into the OPTION register as we would with any other:

```
movlw     b'xxxxxxxx'    ; moves the number into w. reg
movwf     OPTION_REG     ; moves the w. reg into OPTION
```

First you should note that we use the term **OPTION_REG** to describe the OPTION register in the program—this distinguishes it from the (now defunct) **option** instruction. Secondly, I should remind you that if you don't switch into Bank 1 before you perform the above two lines, you will actually move the number into TMR0.

Bit 7 of the OPTION register now controls internal *pull-ups*, which are available on all the I/O pins (except GP3). As before, when the bit is *set* all the pull-ups are totally *disabled*, and when this bit is *clear*, the pull-ups are *enabled* (in general). If pull-ups are enabled in general, then they can be individually enabled or disabled using the WPU register (**W**eak **P**ull-**u**p register). Each bit in the WPU controls the correspond bit in GPIO (e.g., setting bit 0 of WPU *enables* the pull-up on bit 0 of GPIO, and clearing bit 4 of WPU *disables* the pull-up on bit 4 of GPIO).

Bit 6 of the OPTION register is associated with interrupts and will be discussed later. The remaining bits of the OPTION register are the same as before.

10.1.2 The TRISIO Register

The same new method applies to writing to the TRIS file register. Rather than using the **tris** instruction (which doesn't exist on this PIC microcontroller), we can move the number directly into TRISIO (again, this has to take place when in Bank 1):

```
movlw     b'xxxxxx'      ; moves a number into w. reg
movwf     TRISIO         ; sets up inputs and outputs on GPIO
```

10.1.3 Calibrating the Internal Oscillator

Finally, if we wish to use the 4-MHz internal oscillator we need to calibrate it (as we did with the PIC12F508). There are a few important differences to note:

1. The reset vector of this device is **0x000** (i.e., it starts at the beginning of the program memory).
2. The program memory is no longer split into pages. We have the freedom to **goto** or **call** to anywhere without worrying about page bits.
3. OSCCAL is now in Bank 1.
4. The following calibration instruction has been placed at address **0x3FF** (the last address of program memory):

```
retlw    XX    ; returns with calibration value in w. reg
```

Therefore, the code to set up the internal oscillator should now be placed in the **Init** subroutine and consists of:

```
bsf      STATUS, RP0    ; goes into Bank 1
call     3FFh           ; calls calibration address
movwf    OSCCAL         ; moves w. reg into OSCCAL
bcf      STATUS, RP0    ; goes back to Bank 0
```

After executing the line **call 3FFh**, the program returns with the factory-programmed calibration value in the working register, which is then moved into OSCCAL.

10.1.4 PCLATH: Higher Bits of the Program Counter

While PCL holds the lower eight bits of the program counter (bits 0 to 7), the higher bits are not directly accessible. With the PIC5x series we had some handle on the higher bits using *page* bits in the STATUS register. On the PIC12F675 these page bits are largely unnecessary, but are effectively stored in PCLATH. You don't need to worry about the upper bits of the program counter during **goto**s and **call**s, but you have to be careful when doing variable jumps (i.e., adding numbers to the program counter). When you do this, as well as performing the operation on the PCL, the PIC processor will load the state of PCLATH into the upper bits of the program counter (PCLATH feeds directly into the upper byte of the PC). For example, if I have a lookup table which starts at address **0x240**, I need to move **2** into PCLATH before adding anything to the PCL. In the example below, the lookup table starts at address **0x045** so we need to clear PCLATH first.

Example 10.2

```
0045    clrf     PCLATH     ; makes sure PCLATH is 0
0046    movfw    Marx       ; reads in value from file
0047    addwf    PCL, f     ; adds to PCL for variable jump
0048    goto     Groucho    ; branches accordingly
0049    goto     Harpo      ;
0050    goto     Chico      ;
```

10.1.5 Remaining Differences

The remaining new SFRs can be divided into a number of categories, which will be dealt with in turn:

INTCON, PIR1, PIE1, IOC:	Interrupts
EEDATA, EEADR, EECON1, EECON2:	EEPROM
CMCON, VRCON:	Comparator
ADRESH, ADRESL, ADCON0, ANSEL:	Analog to Digital Conversion
TMR1L, TMR1H, T1CON:	Timer 1 (a second timer)

The PIC12F675 also boasts a stack that is *8 levels* deep (compared with 2 levels deep on the PIC5x series). This means you can call a subroutine within a subroutine within a subroutine within a subroutine . . . etc., etc.! Having the third level is particularly useful; the others may not be used that often.

There are two more instructions found on the PIC12F675 and most other PIC microcontrollers (but not on the PIC5x series):

```
addlw    number    ;
```

(Not for PIC5x series) – **add**s a **l**iteral (**number**) to the number in the **w**orking register.

```
sublw    number    ;
```

(Not for PIC5x series) – **sub**tracts the number in the **w**orking register from a **l**iteral (**number**), leaving the result in the working register.

Finally, note that the watchdog timer (WDT) timeout behaves slightly differently when this PIC microcontroller is in sleep mode. Rather than causing a full reset, as on the PIC5x series, a WDT timeout during sleep causes this PIC microcontroller to wake up and continue executing the program from the line after the **sleep** command. When not in sleep, a WDT timeout causes a full reset, as usual.

10.2 Interrupts

An *interrupt* tells the PIC microcontroller to drop whatever it's doing and go to a predefined place (the *interrupt service routine* or *ISR*) when a certain event occurs. Think of it as a fire alarm that goes off when something is detected, and makes the PIC microcontroller go to a

particular meeting point. This event could be receiving a signal on the INT (GP2) pin, or perhaps the state of one of the other I/O pins changing. An interrupt can be set to occur when one of the timers (TMR0 or TMR1) overflows, and there are interrupts associated with the EEPROM, analog to digital converter and the comparator. Each of these interrupts can be enabled or disabled individually, and many can be active at the same time. As they all interrupt the program and make the program jump to the same place (the *ISR*), you may be wondering how we can tell which event triggered the interrupt. Fortunately, as well as having individual enable bits, each interrupt also has an associated flag that can be tested to see if that particular interrupt has occurred. At the start of the ISR you should test the flags of all enabled interrupts and branch off to difference sections accordingly. Note also that these interrupt flags *must be cleared by you*, so somewhere during the ISR you should clear the flag so it's ready to trigger next time. Finally, note that interrupt flags will get set *regardless of the state of the interrupt enable*—the interrupt enable only dictates whether or an interrupt flag going high will actually trigger an interrupt.

The majority of the interrupt enable bits and flags are held in the **INTCON** (**Int**errupt **Con**trol) register. A few further interrupts, known as "peripheral interrupts" have individual enable bits in the **PIE1** (**P**eripheral **I**nterrupt **E**nable) register, and flags in **PIR1** (**P**eripheral **I**nterrupt **R**egister). Let's start with INTCON:

10.2.1 INTCON

Bit no.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Bit name	**GIE**	**PEIE**	**T0IE**	**INTE**	**GPIE**	**T0IF**	**INTF**	**GPIF**

Port Change flag
1: A GPIO change interrupt occurred
0: It hasn't
[**Note:** Must be cleared by you]

External INT flag
1: An INT (GP2) interrupt has occurred
0: It hasn't
[**Note:** Must be cleared by you]

TMR0 Overflow Interrupt flag
1: TMR0 has overflowed
0: TMR0 has not overflowed
[**Note:** Must be cleared by you]

(Continued)

Bit no.	7	6	5	4	3	2	1	0
Bit name	GIE	PEIE	T0IE	INTE	GPIE	T0IF	INTF	GPIF

Port Change Interrupt Enable
1: Enables GPIO port change interrupt
0: Disables it

External INT Interrupt Enable
1: Enables the INT (GP2) interrupt
0: Disables it

TMR0 Overflow Interrupt Enable
1: Enables TMR0 overflow interrupt
0: Disables it

Peripheral Interrupt Enable
1: Enables any enabled "peripheral interrupts"
0: Disables all "peripheral interrupts"

Global Interrupt Enable
1: Enables any enabled interrupts
0: Disables ALL interrupts

Bit 7 (GIE) is the global interrupt enable, which is the master switch for all interrupts. Turn it off and no interrupts are enabled (regardless of the state of their individual enable bits). Turn it on and interrupts whose individual enable bits are set will be enabled.

Bit 6 (PEIE) is a mini-master switch for a group of interrupts which are known as "peripheral interrupts". These interrupts have their own enable bits in the PIE1 register. Therefore, in order to use these interrupts you have to enable three bits—the individual enable bit in PIE1, this bit, and the GIE.

Set bit 5 (T0IE) to use the TMR0 overflow interrupt—this simply triggers an interrupt whenever TMR0 overflows from 255 to 0. In the interrupt service routine you can test bit 2 (T0IF) to see if a TMR0 overflow interrupt has occurred (remember that you need to clear it yourself!).

Bit 4 (INTE) controls the "External Interrupt" which depends on the state of the pin labeled INT (GP2). The interrupt can be set to trigger on the rising edge *or* falling edge of the signal on this pin. This is done using bit 6 of the OPTION register: if bit 6 of OPTION is *clear*, the INT interrupt will occur on the *falling* edge of the INT pin. If bit 6 of OPTION is *set*, the INT interrupt will occur on the *rising* edge.

Finally, bit 3 (GPIE) of the INTCON register controls the GPIO change interrupt. This interrupt can trigger when any one of the GPIO pins *changes*. To use this interrupt you need to set this

bit, and also select which GPIO pin should be able to trigger the interrupt. This is done with the **IOC** (**I**nterrupt **O**n **C**hange) register. Each bit in the IOC corresponds to a bit in GPIO—set the bit to enable interrupts when that pin changes. For example, to enable an interrupt to occur whenever pins GP0, GP2 and GP4 change, you should write the following:

```
bsf     STATUS, RP0    ;moves into Bank 1
movlw   b'00010101'    ;enables GP0, GP2 and GP4
movwf   IOC            ;for the GPIO change interrupt
movlw   b'10001000'    ;enables GPIO change interrupt,
movwf   INTCON         ;and enables global interrupts
```

10.2.2 The Interrupt Service Routine

When an interrupt takes places, the PIC processor will jump to the instruction at address **0x004**. What's more, it actually *calls* a subroutine that starts at address **0x004**. This is so that after dealing with the interrupt, the processor can return to where it left off before the interrupt occurred. In our previous programs, address **0x004** has been five lines into our **Init** subroutine, so we will have to make some changes to the template. At address **0x004** we want to **goto** somewhere, which we will call **isr**. When the processor comes across the return instruction in **isr**, it will return to the point in the program where it was when the interrupt occurred. Remembering that the reset vector for the PIC12F675 is **0x000**, we could write:

```
        org     0
        goto    Start
        org     4
        goto    isr
Init    etc.
```

The only problem with this is that you are wasting addresses **0x001** to **0x003**, but this is not serious. Alternatively, you could write the following:

```
          org     0
          goto    Start

Init      clrf    GPIO          ;
          movlw   b'xxxxxxxx'   ;
          goto    InitCont      ;skips address 0x004
          goto    isr           ;at address 0x004 goes to isr
InitCont  etc.                  ;carries on with rest of Init
```

Counting down you should see that the line **goto isr** is still at address **0x004**, and rather than losing three lines (**0x001** to **0x003**), we really only waste one line (**goto InitCont**).

The start of the interrupt service routine should begin by checking which particular event triggered the interrupt (if more than one input is enabled).

```
isr     btfss    INTCON, 0    ; did GPIO change interrupt occur?
        goto     GPchange     ; yes, it was a GPIO change
        btfss    INTCON, 1    ; did the INT/GP2 occur?
        goto     External     ; yes, it was the INT interrupt
        btfss    INTCON, 2    ; did the TMR0 overflow?
        goto     Timer        ; yes, it was the TMR0 interrupt
        etc.
```

Fortunately, the processor automatically clears the GIE bit in the INTCON register when an interrupt occurs. This means that no interrupt can take place in the ISR – you can imagine the havoc that would take place should this not be the case! Thus, at the end of the ISR we would have to set the GIE just before returning, but even if we did this, an interrupt could take place immediately afterwards, before actually returning from the ISR. We can't set the global enable *after* returning because we don't know where the processor is going to return to. Fortunately, there is a new instruction that solves this problem:

```
retfie                        ;
```

This **ret**urns **f**rom a subroutine and sets the global **i**nterrupt **e**nable bit *at the same time*. In certain cases you may want to return from the ISR (or indeed any subroutine) without setting the global interrupt enable. On the PIC5x series, **retlw** is the only available instruction. On the PIC12F675 we can use:

```
return                        ;
```

This simply **return**s from a subroutine.

10.2.3 Interrupts During Sleep

If an interrupt that has been individually enabled occurs during sleep, the PIC microcontroller will wake up and do one of two things, depending on the state of the GIE. If the GIE is off, it will just wake up from sleep and carry on running through the program from the line after the **sleep** command. If the GIE is set, the processor will execute the instruction after **sleep**, and then call the ISR (address **0x004**). Therefore, if you just want to use an interrupt to wake up the PIC microcontroller, you should clear the GIE *before* the **sleep** instruction. If you want the program to respond in some other way, you should make sure GIE is set. Note that the TMR0 is off during sleep, so the TMR0 interrupt cannot be used to cause a wake-up from sleep.

Example 10.3 Make the PIC microcontroller go to sleep until triggered by a change of state of inputs GP0 or GP1 (assume these two have already been enabled in the IOC register). It should then carry on with the rest of the program with the TMR0 and GPIO change interrupts enabled.

```
movlw    b'00001000'    ; only enables GPIO change interrupt
movwf    INTCON         ; and disables GIE
sleep                   ; goes to sleep
movlw    b'10101000'    ; enables TMR0, GPIO change, and
movwf    INTCON         ; global interrupts
```

That's all there is to interrupts; just remember to make the ISR fairly short, because you can't get an interrupt while you're in it. Think clearly when writing this part of the program, particularly if you have more than one interrupt enabled.

10.2.4 Maintaining the STATUS Quo

Remember that interrupts can occur at any point during the program. We could be moving something into the working register, and be about to move it into another file register when *WHAM!*, an interrupt occurs. When we return from the ISR, there is likely to be a new number in the working register—what happens now?

Example 10.4

```
movlw    d'15'              ; has MinutesFame reached 15?
subwf    MinutesFame, w     ;
btfss    STATUS, Z          ;
```

In Example 10.4, what happens if an interrupt occurs after the second line? Upon returning from the ISR, the zero flag may be in a different state. In order to ensure that interrupts don't disrupt the functioning of the program, we have to store the contents of the working register and the STATUS register at the beginning of the ISR. At the end of the ISR we copy these values back and then return. To store the original values we use:

```
movwf    W_temp         ; stores w. reg in temp register
movfw    STATUS, w      ; stores STATUS in temp
movwf    STATUS_temp    ; register
```

And to restore the two registers at the end of the ISR we use:

```
movfw    STATUS_temp   ; restores STATUS register to
movwf    STATUS        ; original value
swapf    W_temp, f     ; restores working register to
swapf    W_temp, w     ; original value
```

This may seem a little puzzling. Why not simply move **W_temp** directly into the working register using the **movfw** command? The reason is that the **movfw** instruction affects the zero flag, and so has the potential of altering the original value of the STATUS register. Fortunately, the **swapf** instruction does not affect the zero flag, and so is suitable in this case. Note that

swapping twice results in no net change to the value, and so these two instructions move the value from **W_temp** into the working register with no change to STATUS.

10.2.5 New Program Template

With all these new file registers it is clear that our program template needs to be updated. A good practice is to clear any control file registers that you are not using. The only exception to this is the comparator module, which is in a low-power mode if the **CMCON** (**c**omparator **m**odule **con**trol) register is clear, but is turned completely off if you *set* bits 0–2, as shown in the template below. Even if you are going to use interrupts in the program, you should *not* set the global interrupt enable until everything else is configured. You can then use the **ret-fie** instruction to leave **Init** and enable global interrupts. If you don't want to enable interrupts at this point, end **Init** with the **return** instruction. If you are not using interrupts, you can remove the ISR.

```
;************************************
; written by:                       *
; date:                             *
; version:                          *
; file saved as:                    *
; for PIC ...                       *
; clock frequency:                  *
;************************************

; Program Description: ____________________________________________

; ________________________________________________________________

               list     P=12F675
               include "c:\pic\p12f675.inc"
;============
;Declarations:

W_temp         equ      20 h
STATUS_temp    equ      21 h

               org      0             ; first instruction to be executed
               goto     Start         ;

               org      4             ; interrupt service routine
               goto     isr           ;

;============
;Subroutines:

Init           bsf      STATUS, RP0   ; goes to Bank 1
               call     3FFh          ; calls calibration address
               movwf    OSCCAL        ; moves w. reg into OSCCAL
```

```
                movlw   b'xxxxxx'    ; sets up which pins are inputs
                movwf   TRISIO       ;   and which are outputs
                movlw   b'xxxxxx'    ; sets up which pins have
                movwf   WPU          ;   weak pull-ups enabled

                movlw   b'xxxxxxxx'  ; sets up timer and some pin
                movwf   OPTION_REG   ;   settings
                clrf    PIE1         ; turns off peripheral ints
                clrf    IOC          ; disables GPIO change int.
                clrf    VRCON        ; turns off comparator V. ref.
                clrf    ANSEL        ; makes GP0:3 digital I/O pins

                bcf     STATUS, RP0  ; back to Bank 0
                clrf    GPIO         ; resets input/output port
                movlw   b'00000111'  ; turns off comparator
                movwf   CMCON        ;
                clrf    T1CON        ; turns off TMR1
                clrf    ADCON0       ; turns off A to D conv.
                movlw   b'0xxxxxxx'  ; sets up interrupts
                movwf   INTCON       ;

                retfie or return     ;

isr             movwf   W_temp       ; stores w. reg in temp register
                movfw   STATUS       ; stores STATUS in temporary
                movwf   STATUS_temp  ;   register

               (Write the interrupt service routine here)

                movfw   STATUS_temp  ; restores STATUS register to
                movwf   STATUS       ;   original value
                swapf   W_temp, f    ; restores working register to
                swapf   W_temp, w    ;   original value
                        retfie or    ; returns, enabling GIE
                          return
;=============
;Program Start
Start           call    Init         ; initialization routine
Main            (Write your program here)
                END
```

10.2.6 Example Project: Quiz Game Controller

The project to practice interrupts will be a quiz game device. There will be three push buttons (one for each player), three LEDs (one by each button to show which player pressed first), and a buzzer to show that a button has been pressed, which stays on for 1 second. There will also be a button for the quizmaster to reset the system (this can be connected to the GP3/ $\overline{\text{MCLR}}$ pin). You may wonder why we are going to the trouble of using interrupts for this project, which looks as if it may be viable on the PIC16F54. However, without interrupts we would

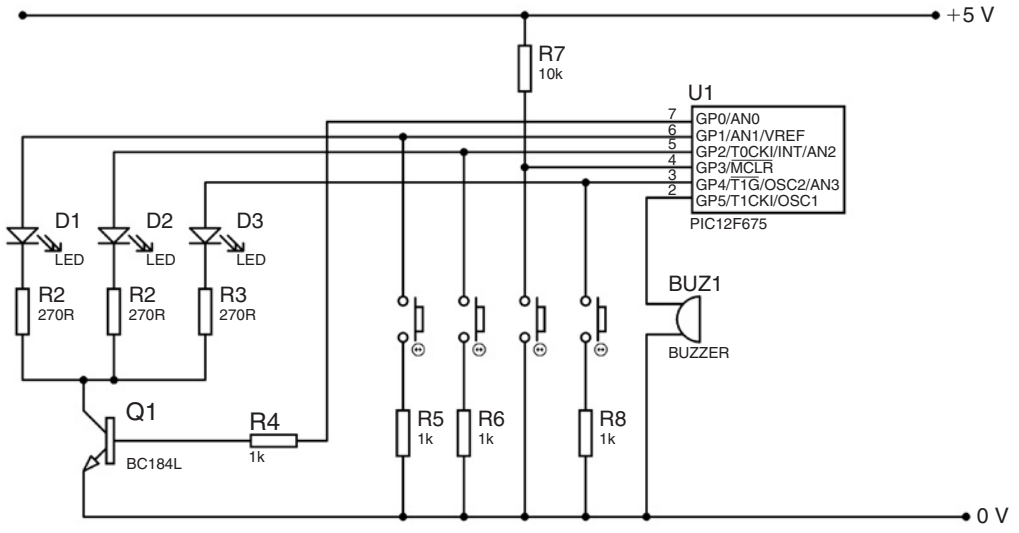


Figure 10.3

have to test each button in turn, one after the other. Let us say, for example, that the program had just finished testing the first button, and then immediately afterwards, the first button is pressed. The program then tests the second button, after which the third player responds. The third player's button is now tested, and as far as the program is concerned, he responded first. The times we are dealing with are millionths of a second, but if we want to be really exact we can use interrupts. The three players' buttons could be connected to pins which have the GPIO change interrupt enabled, so that this interrupt would trigger the moment any button is pressed. The circuit diagram for this project is shown in Fig. 10.3. Because the PIC12F675 does not exactly have an abundance of I/O pins, we have to double up the button pins and LED pins. In order for this to work, we need an extra pin (GP0) to act as a master switch for the LEDs. With this output set (5V), pins GP1, 2 and 4 can be made outputs to control whether the LEDs are on or off, irrespective of whether a button is pressed. When this pin is clear, pins GP1, 2 and 4 can be made inputs (with weak pull-ups enabled) which read the state of the buttons. We could avoid this complication by moving to a larger device (such as the PIC16F676), but this illustrates what can be achieved with a relatively small number of pins. In summary, the two modes of operation with regards to pin settings are:

1. Waiting for button to be pressed: GP0 is an output, and off (so LEDs are disabled). GP1, 2 and 4 are inputs, with weak pull-ups enabled.

2. Displaying correct LED: GP0 is an output, and on (so LEDs are enabled). GP1, 2 and 4 are outputs.

The buzzer will be connected to pin GP5. The flowchart is shown in Fig. 10.4—as you can see, the main body of the program is nothing at all, just a constant loop. All the clever stuff happens in the interrupt service routine.

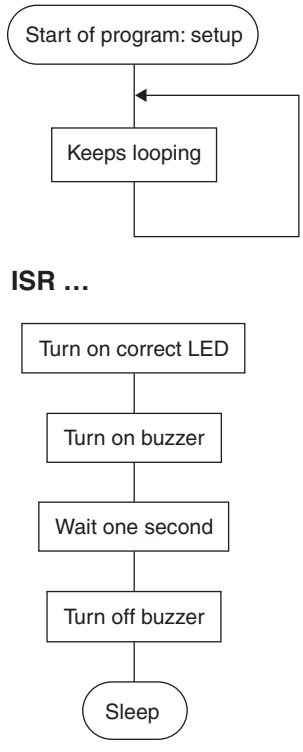


Figure 10.4

The main body of the program is just a loop, waiting for the GPIO change interrupt to occur. The program, from **Start**, is therefore:

```
Start    call     Init     ; sets everything up
Main     goto     Main     ; keeps looping
```

This leaves the **isr** to complete. In this project, we are using two interrupts, so we need to check the interrupt flags to determine which interrupt occurred. Note that in this particular project, it isn't essential to include the code at the start and end of the **isr** which store and recover the contents of the working and STATUS registers, as nothing is happening while we're waiting for an interrupt.

Interrupt flags need to be reset in the program, otherwise, upon returning from the ISR, the same interrupt will trigger again. The GPIO interrupt flag is reset by clearing bit 0 of INTCON. We now need to record which button was pressed, and turn on the corresponding LED. The buttons are *active low*, so the bit goes to 0 when a button is pressed. To turn this into an *active high* signal, we invert the state of GPIO, moving the result into the working register, and then mask all bits except the ones we're interested in: GP1, 2 and 4.

```
bcf      INTCON, 0    ; resets GPIO interrupt flag
comf     GPIO, w      ; inverts state of GPIO
andlw    b'010110'    ; masks all except GP1, 2 and 4
movwf    temp         ; stores result
```

The number in **temp** should now be all 0s, with a 1 at the bit corresponding to the button which was pressed. This number can then be used to turn on the correct LED. As a safety precaution to guard against problems like button bounce, etc., we can do a quick check; if the number in **temp** is 0, this was a false alarm and we should ignore it. The zero flag would have been triggered by the **andlw** instruction, and so we can test it immediately:

```
btfss    STATUS, Z    ; is a button actually pressed?
retfie                ; no - false alarm,so returns
```

Now we are sure a button was pressed, and **temp** holds a number corresponding to which one it was. We now need to change pins GP1, 2 and 4 to outputs (which automatically disables weak pull-ups. The contents of **temp** can then be moved back into GPIO, which will turn on the LED corresponding to the button that had been pressed. We also need to enable the LEDs by setting GP0, and turn on the buzzer by setting GP5. This can be achieved through separate **bsf** commands, but we do the same by adding the number **b'100001'** to **temp**, before moving it into GPIO.

We should then disable the GPIO change interrupt, and enable the TMR0 interrupt, and return from the isr, enabling global interrupts.

We will use the TMR0 interrupt to time the 1 second delay period for the buzzer. If we use the 4 MHz internal oscillator, instructions are executed at a frequency of 1 MHz, and TMR0 counts up at a frequency of 3.9 kHz. The frequency of the TMR0 interrupt is therefore 15.3 Hz, so if we want to time an approximate 1-second delay, we should use a postscaler of 16. We set up a file register with the number 16 (do this in **Init**), and decrement it each time the TMR0 interrupt occurs. After the 16th interrupt, the buzzer is turned off and the PIC microcontroller goes to sleep. Upon going to sleep, the states of the outputs stay the same, so the correct LED stays on. Before going to sleep, the program should set up INTCON so that all the interrupts are disabled. Therefore, only a reset on the $\overline{\text{MCLR}}$ pin (to which the quiz-master's button is connected) will wake up the device.

This completes the program, which is given in Appendix 5. So far we have looked at two interrupts, the GPIO change and TMR0 interrupts. Remaining interrupts on this PIC microcontroller (external interrupt on the INT pin, A/D conversion interrupt, EEPROM write interrupt and comparator interrupt) will be dealt with in subsequent parts of this chapter.

10.3 EEPROM

EEPROM (**E**lectrically **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory) can be seen as a large collection of general purpose file registers whose contents remain intact even after power has

been removed. We used the analogy of a filing cabinet to describe the file registers. When the PIC microcontroller is turned off, the filing cabinets are left exposed and there is little guarantee that the numbers in the file registers will be intact when you turn it on again. The EEPROM behaves like the office safe—a secure place to store data which will not be affected by removing power.

We can define this in more rigorous terms and say that the 64 file registers in the data memory are RAM (**r**andom **a**ccess **m**emory). In addition to these, there are 128 data locations which are ROM (**r**ead-**o**nly **m**emory)—the EEPROM. Reading and writing to these secure locations requires a bit more effort than with the file registers in the RAM.

The file register EEADR holds the address in the EEPROM which you wish to read or write to, while EEDATA holds the data that you have just read, or which you wish to write to the EEPROM. EECON1 holds settings for the EEPROM, and EECON2 is a special register used in the EEPROM writing process. Note that all these EEPROM file registers are found in Bank 1.

10.3.1 EECON1

bit no.	**7,6,5,4**	**3**	**2**	**1**	**0**
bit name	unused	**WRERR**	**WREN**	**WR**	**RD**

Read Control Bit
1: Starts an EEPROM read
0: EEPROM read has finished

Write Control Bit
1: Starts an EEPROM write operation (stays high until write operation finishes)
0: EEPROM write has finished

EEPROM Write Enable Bit
1: Allows writing to the EEPROM
0: Forbids writing to the EEPROM

EEPROM Write Error Flag
1: An EEPROM write has prematurely terminated
0: The write operation completed without error

10.3.2 Reading from the EEPROM

Let's pretend for the moment that you have already written something to the EEPROM and you now wish to read it.

Example 10.5 You wish to read the number stored in address 4Eh of the EEPROM and move it into the working register.

```
bsf      STATUS, RP0    ; go to Bank 1
movlw    4Eh            ; selects EEPROM address
movwf    EEADR          ;
bsf      EECON1, 0      ; starts EEPROM read operation
                        ; storing result in EEDATA
movfw    EEDATA         ; moves read data into w. reg
```

After moving into Bank 1, the next two instructions tell the PIC microcontroller which address in the EEPROM you wish to read. The EEPROM Read bit is set to initiate a read from the EEPROM, putting the result in EEDATA. This file register can be read directly immediately after the read command.

10.3.3 Writing to the EEPROM

Writing to the EEPROM is made slightly more complicated by the fact that it is a more "dangerous" operation. Reading from the EEPROM is quite harmless—all you are doing is moving a number into EEDATA (photocopying some documents that are in the safe). Writing to the EEPROM, on the other hand, involves actually *changing* the data in the EEPROM (altering the documents in the safe). Because of this distinction, steps are taken to minimize the risk of accidentally writing to the EEPROM. You have to provide a type of "combination for the safe" in the program before you are allowed to write to the EEPROM.

Example 10.6 You wish to write the decimal number 69 into the EEPROM address space 78h. First ensure the write enable bit (EECON1, bit 2) is set, then provide the "safe combination"—a series of four instructions which must immediately precede the write operation. As the execution of this procedure must not be interrupted, the global interrupt enable should be cleared for the duration of the write operation.

```
bsf         STATUS, RP0    ; goes to Bank 1
movlw       d'69'          ; moves the number to be written, into
movwf       EEDATA         ;  EEDATA
movlw       78h            ; moves the address to be written to
movwf       EEADR          ;  into EEADR
bsf         EECON1, 2      ; enables a write operation
bcf         INTCON, 7      ; disables global interrupts

movlw       55h            ; now follows the 'safe combination'
movwf       EECON2         ;
movlw       AAh            ;
movwf       EECON2         ;
bsf         EECON1, 1      ; starts the write operation
etc.
```

There is still a little more to the writing operation because, although we have started the write, it will take quite a few clock cycles to complete. This is in contrast to the read operation, which takes place immediately. If there is something in particular we want to do when the write finishes we can wait until the write completes by testing EECON1, 1 (the write control bit) which gets cleared when the write operation finishes:

```
EELoop
     btfsc   EECON1, 1      ; has write operation finished?
     goto    EELoop         ; no, still high, so keeps looping
```

If we don't want to get tied up in some loop, but want to be able to get on with other things, we can use the *EEPROM Write Complete* interrupt, which (as you may have guessed) triggers when the write operation finishes. This interrupt is a so-called "peripheral interrupt" and can be enabled using bit 7 of the PIE1 register. Don't forget that the peripheral interrupt enable (INTCON, bit 6) as well as the global interrupt enable need to be set in order for the interrupt to occur.

A final point to note is that if you are not using interrupts at all (and have therefore disabled the global interrupt bit at the beginning of the program) you may remove the relevant lines in the EEPROM write procedure.

10.3.4 Example Project: Telephone Card Chip

You are no doubt familiar with the so-called "smart cards" that have found their way into a variety of applications. These cards have tiny chips embedded inside them, and either have contacts to communicate with the outside world or have a loop antenna inside the card. To demonstrate the use of the EEPROM, we will write the program for a PIC microcontroller which has been embedded within a "smart" telephone card. We shall assume the card has 8 contact pins with which to communicate with the public telephone box. Two of these will be power pins; one will indicate that a call is in process and another will be from the card to alert the phone box when the card runs out of minutes. There will also be a pin used to reset the number of minutes left on the card to a specified value (this could be used for top-ups), and three pins will hold the 3-bit "top-up" value, therefore allowing one of 8 possible values to be written to the card. The time remaining (in minutes) will be stored in the EEPROM so that the data remains intact when power is removed (the card is removed from the phone box). For simplicity's sake minutes are used as the basic unit of time, but you can adjust the program to count down in seconds, if you want.

The circuit diagram for this arrangement is shown in Fig. 10.5. The interface with the phone box can be simulated using a number of switches and LED. GP0 is an output and will go high if there are minutes on the card, and low when the time runs out. GP1 is an input from the phone box which goes high when a call is in progress. GP2 is an input which will be made

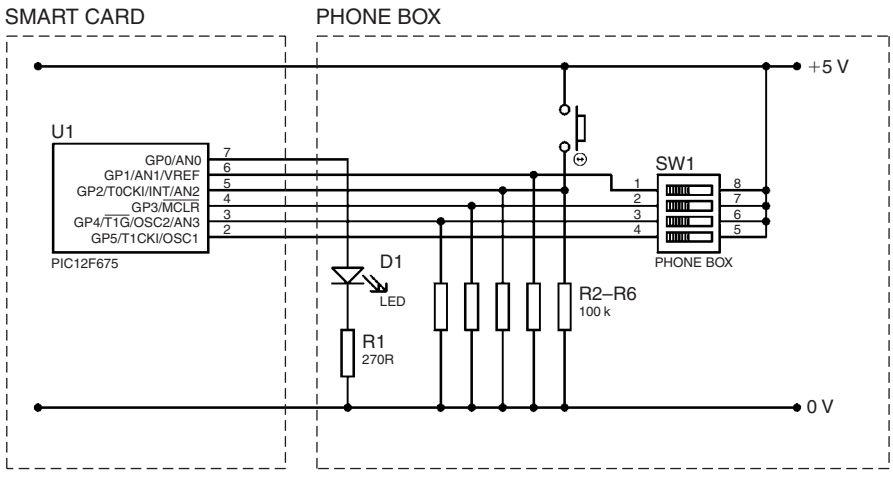


Figure 10.5

high when the card value is to be reset, and is otherwise low. GP3, 4 and 5 store the 3-bit "top-up" value. The flowchart is shown in Fig. 10.6.

Starting from the program template developed earlier, we have to select values for INTCON, TRISIO, WPU and OPTION_REG. We will use the INT interrupt on GP2 (*rising* edge) to trigger the reset of the time remaining, and require no other interrupts. We will not use the weak pull-ups.

From the flowchart, at **Main** we should first read the EEPROM and see if there are any minutes left on the card. The number of minutes will be stored in EEPROM address **00h**. If there are minutes left, the program should skip forward to a section called **Active**, and if not, it should turn off GP0 and go to sleep. If the card is topped up, the INT interrupt will trigger, the line following **sleep** will be executed and then the ISR will be called. After returning from this, the program should loop back to **Main**.

In **Active**, you should set GP0, and then enter a loop where you wait until there is a call in progress. When the call is active, the program should count time and see if one minute has passed. Because the intermediate timing registers (markers and postscalers) are not stored in the EEPROM, the card will only count down complete minutes, and not fractions of minutes. This could easily be rectified by storing these registers in the EEPROM as well, but this is left as a possible development. Using the internal 4 MHz oscillator and TMR0 prescaled by 256, we can use a marker of 125 and postscalers of 125 and 15 to time one minute.

Don't forget to set up the timing registers in **Init**. After one minute has passed we should reset the final postscaler to its correct value, and then decrement the number of minutes stored in the EEPROM. This involves reading the data in, decrementing it, and then writing it back. Finally, the program should loop back to **Main**.

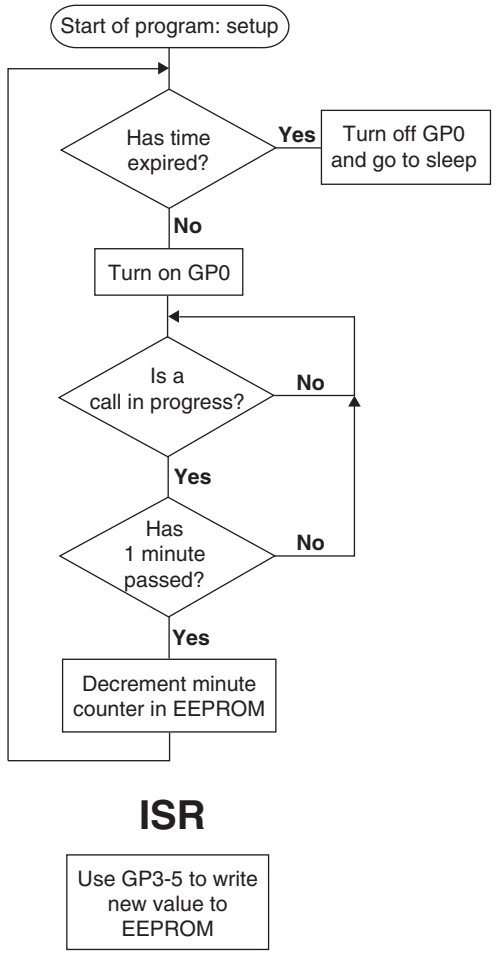


Figure 10.6

All that remains is the handling of the INT interrupt, upon which the number of minutes stored should be reset to the value determined by bits GP3:5. The eight possible values are 2, 5, 10, 20, 40, 60, 120 and 0 minutes—assigned to (000, 001 . . ., 111) respectively. We should begin by clearing the INT interrupt flag (INTCON, 1), and then read in the state of GPIO. Don't forget that the interrupt may have occurred anywhere in the program, so we need to make sure we switch into Bank 0 in order to read the GPIO register. The bits of interest in GPIO are bits 3:5, so we should rotate this three times to the right. This can't be done directly to GPIO, so we have to use an intermediate register called **temp**. Finally, to turn this into a number between 0 and 8 (b'000' and b'111') we should mask bits 3–5.

We can create a lookup table in a subroutine called **CardValue** which is called with a number between 0 and 8 in the working register, and which returns with the appropriate number of minutes.

After calling **CardValue**, the number in the working register should be written into the EEPROM—you should be well practiced at this by now. Finally, you should wait until the write operation has completed before restoring the original values of STATUS and the working register. Upon returning, global interrupts should be enabled, so the **retfie** instruction should be chosen.

All that remains is a quick check to make sure you have declared all file registers, and set them up with appropriate values in **Init**. The entire program is shown in Appendix F. When simulating this project in MPLab, you can view the contents of the EEPROM by going to View l EEPROM. When you program a PIC microcontroller, the contents of the EEPROM window will be written to the EEPROM on the chip. Similarly, when you read the program from a chip (Programmer l Read device), the contents of the chip's EEPROM will be shown in the EEPROM window.

10.3.5 Further EEPROM Examples: Music Maker

For further EEPROM practice, you could make a device on which you can store musical notes and play back a melody. You can cover 8 octaves which correspond to 96 different notes, so each note is assigned a byte in the EEPROM. With an EEPROM of 128 bytes, a melody of up to 128 notes can be stored. A speaker can be connected to one of the outputs, and the different notes are obtained by producing square waves of different frequencies. The frequencies of the notes in the scale are shown in Table 10.1.

Table 10.1

Middle C	C#	D	D#	E	F	F#	G	G#	A	A#	B
262 Hz	277	294	311	330	349	370	392	415	440	466	494

The notes of the other octaves are produced by multiplying or dividing these numbers by two. For example, the next C above middle C would be 524 Hz. Human hearing goes from about 10 Hz to 20 kHz, so rather than storing the frequency of the note in the EEPROM, it would be more sensible to store the note (e.g., D# or G) and the octave number (i.e., a number between 1 and 8). Each of these would be stored in a nibble, so for example, the hexadecimal number **56h** in the EEPROM could mean an E in the 6th octave.

10.3.6 Power Monitor

A second possible EEPROM project is a device which is powered by the mains (indirectly of course!) and which counts and displays the time, storing the latest values in the EEPROM. Then, if the mains cuts out, when the device is next powered up it will display the time stored in the EEPROM—i.e., the time at which the power cut began.

Other applications for the EEPROM include devices concerning security where passwords are involved.

10.4 Analog to Digital Conversion

Analog to digital conversion (ADC) is the ability to measure the voltage at an analog input and convert this reading into a number between 0 and 1024 (for 10-bit conversion). This translates into a precision of about 5 mV when a 5 V supply is used. For example, if the result of an A/D conversion was 10, the input voltage was 0.05 V, and if it was 400, the input was about 1.95 V. This allows much greater flexibility than digital inputs, which can only tell whether an input is high or low (more than 2.5 V or less than 2.5 V). Some PIC microcontrollers support 8-bit ADC, which leads to lower precision. The voltage can be measured relative to the supply voltage (VDD), or relative to the voltage on another pin (the VREF pin—GP1).

A/D conversion can be a fairly lengthy process (compared with the speed at which most instructions are executed). The time that an A/D conversion takes can be changed by you, though if you make it too short the accuracy of the result will be affected. This and other aspects of the ADC are controlled in the registers **ADCON0** and **ANSEL**.

10.4.1 ADCON0

Bit no.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Bit name	**ADFM**	**VCFG**	**–**	**–**	**CHS1**	**CHS0**	**GO/DONE**	**ADON**

A/D on bit
1: ADC is on
0: ADC is off (consuming no current)

GO/DONE
1: Starts A/D conversion. Stays high until finished
0: A/D conversion finished

Channel select bits
00: Channel 00 (AN0)
01: Channel 01 (AN1)
10: Channel 02 (AN2)
11: Channel 03 (AN3)

Voltage reference bit
1: Measures relative to V_{REF} pin
0: Measures relative to V_{DD} (supply voltage)

A/D result formed select
1: Right justified – result stored in ADRESL and ADRESH (bits 0:2)
0: Left justified – result stored in ADRESL (bits 6:7) and ADRESH

Bit 0 of ADCON0 is the on/off switch for the A/D converter. When it is set, the ADC is on and the PIC microcontroller consumes extra current. Bit 1 is set to start an A/D conversion, and stays set for the duration of the process, after which it automatically clears. This bit can therefore be tested to see when the A/D conversion finishes. Bits 2 and 3 together select which analog input you want to measure. For example, to test the voltage on AN2 (GP2) you should set bit 3 and clear bit 2. The voltage reference (V_{REF} pin or V_{DD}) can be selected using bit 6 of ADCON0. The measured 10-bit answer is held over two registers: **ADRESH** and **ADRESL** (**A/D Res**ult, **h**igher and **l**ower bytes). You have a choice in how the 10-bit number is stored over these two registers. Either it can be shifted to the right, so that bits 0:7 of the answer are held in ADRESL and bits 8:9 of the answer stored in bits 0:1 of ADRESH, or it can be shifted to the left, so that bits 0:1 of the answer are held in bits 6:7 of ADRESL, and bits 2:9 of the answer are held in ADRESH. This is illustrated in Fig. 10.7.

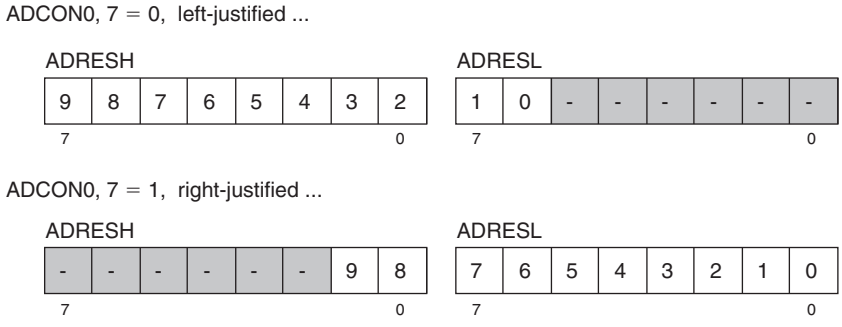


Figure 10.7

10.4.2 ANSEL: Analog Select Register

The ANSEL register has two purposes: setting the A/D conversion speed and selecting whether particular GPIO pins should be acting as analog inputs, or standard digital I/O pins. Bits 0:3 refer to pins AN0:AN3—when they are clear the relevant pin behaves like a digital I/O pin, however when set, the corresponding pin acts as an analog input, and cannot be used as a digital input.

Example 10.7 Push buttons are connected to pin GP0 (AN0) and GP2 (AN2), while a thermometer input (analog input) is connected to GP1 (AN1) and a microphone input (analog input) to GP4 (AN3). The number b'1010' should be moved into bits 0:3 of ANSEL.

Bits 4:6 of ANSEL determine the A/D conversion clock, as shown in Table 10.2. Accurate A/D conversion requires a time of 1.6 s or greater, however there is no point in making it much longer than this. The internal oscillator provides a conversion time of about 4 s, though this can vary between 2 and 6 s.

Table 10.2: Use of the ANSEL Bits 2:0 to Select the A/D Conversion Time. *Italic* Numbers Represent Conversion Times Which Are Too Fast, or Needlessly Slow

ANSEL Bits 6:4	A/D Conversion Clock	Device Frequency			
		1.25 MHz	2.46 MHz	4 MHz	20 MHz
000	Fosc/2	1.6 μs	*800 ns*	*500 ns*	*100 ns*
001	Fosc/8	6.4 μs	3.2 μs	2 μs	*400 ns*
010	Fosc/32	*25.6 μs*	*12.8 μs*	8 μs	1.6 μs
011	FRC: Internal oscillator	~4 μs	~4 μs	~4 μs	~4 μs
100	Fosc/4	3.2 μs	1.6 μs	*1 μs*	*200 ns*
101	Fosc/16	*12.8 μs*	6.4 μs	4 μs	*800 ns*
110	Fosc/64	*51.2 μs*	*25.6 μs*	*16 μs*	*3.2 μs*
111	FRC: Internal oscillator	~4 μs	~4 μs	~4 μs	~4 μs

10.4.3 A/D Conversion Interrupt

To wait for an A/D conversion to complete, we could just keep testing ADCON0, bit 1 (which we used to start the conversion) and wait for it to clear. The A/D conversion interrupt frees up the program from this loop, and triggers upon completion of the conversion. This interrupt is a "peripheral interrupt" and so it is enabled in the PIE1 register (bit 6) and its interrupt flag is found in the PIR1 register (bit 6). In order for the interrupt to trigger, both the peripheral interrupt enable and the global interrupt enable bits in INTCON (bits 6 and 7) must be set.

10.4.4 Example Project: Bath Monitor

To practice A/D conversion, our next project will be a temperature-sensing device that indicates whether the temperature of your bath is too high, too low, or just right (i.e., within an acceptable temperature range). There will be three LEDs to indicate these three possible conditions, connected to GP0, GP1 and GP2. GP4 (AN3) will be the analog input connected to the temperature sensor LM35 which varies its output linearly according to temperature. The circuit diagram is shown in Fig. 10.8, and the flowchart in Fig. 10.9.

As with the quiz game controller, the main loop of the program is practically nothing at all. In this case we simply need to keep starting the A/D conversions, and the response to the measurement will be handled in the ISR. The program from **Start** is therefore:

```
Start     call      Init          ; sets everything up
Main      bsf       ADCON0, 1     ; start A/D conversion
          goto      Main          ;
```

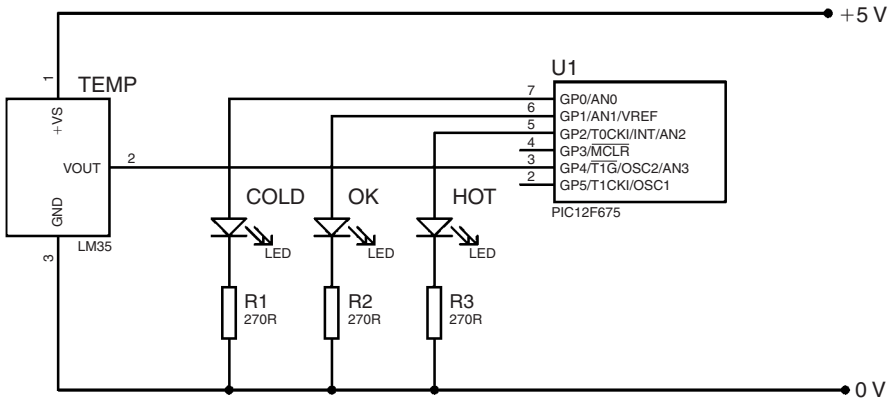


Figure 10.8

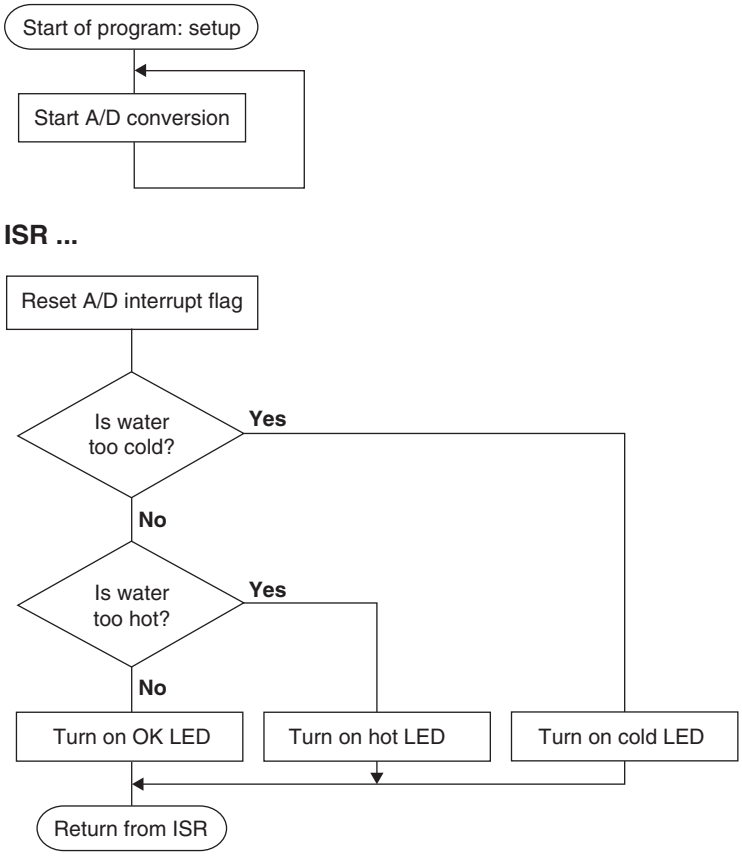


Figure 10.9

Looking at the above code we can see that using an A/D conversion interrupt in this case isn't actually necessary. We could simply have written the following:

```
Start     call      Init         ; sets everything up
Main      bsf       ADCON0, 1    ; starts A/D conversion
ADLoop    btfsc     ADCON0, 1    ; has conversion finished?
          goto      ADLoop       ; no, so keeps looping
          etc.                   ; yes, so exits loop
```

However, in a more advanced version of this program we may want to have a more complex main loop, and in such a case the A/D conversion interrupt may be very useful. We will therefore keep using the interrupt method in this project and write the program as if the interrupt may have occurred during a more complex program (i.e., use the working and STATUS register storage/recovery code in the **isr**).

In the ISR, we needn't test the A/D interrupt flag as it is the only interrupt which has been enabled, but we do need to reset it (clear it).

Following the flowchart, we see that the next step is to see whether the temperature is too cold (i.e., whether the measured analog voltage is below a certain threshold). Depending on the required accuracy, we have two choices. We can choose to discard the two *least significant* bits of the answer (bits 0 and 1) which are held in ADRESL, and use only the 8 bits held in ADRESH. We can simply see whether ADRESH is below the "cold" threshold. This particular temperature sensor gives an output voltage of 0.01 V per degree Celsius. If we say the minimum bath water temperature is 36°C, this means the minimum input voltage is 0.36 V (which is compared with the reference voltage, V_{DD}, which is 5V). $0.36/5 = 0.072$ and $0.072 \times 256 = 18$. We've multiplied by 256 because we use only the number in ADRESH (the *eight* most significant bits). We would therefore write:

```
movlw     d'18'        ; is ADRESH less than 18?
subwf     ADRESH, w    ;
btfss     STATUS, C    ; carry flag is 0 when result is -ve
goto      Cold         ; C is 0, therefore ADRESH < 18
```

If, on the other hand, we want to take advantage of the full 10-bit precision of the A/D converter, we need to test the full 10-bit number which is spread over ADRESH and ADRESL. For the sake of the more discerning bather, we will take this 10-bit approach in this project. The threshold voltage for 36°C is still 0.36 V, which when divided by 5V leaves 0.072. We multiply this by 1024 to get the 10-bit value for the cold threshold: **d'74'**. If you write this number out in binary as it will appear in registers ADRESH and ADRESL you get: **b'00010010 10000000'** (remember, bits 0:1 of the A/D result are stored in ADRESL bits 6:7, and bits 2:9 of the A/D result are stored in ADRESH). Thus, the upper byte is (**0x12**) and the lower byte is (**0x80**). To test the result of the 10-bit A/D conversion we subtract a number from the result, as we have done before, except this time the number happens to be split over two file registers. We handle

this in the same way we handle normal arithmetic—first subtract the lower bytes, subtracting one from the higher byte if you need to borrow, then subtract the higher bytes.

```
bsf     STATUS, RP0    ; goes to Bank 1
movlw   0x80           ; subtracts lower byte
subwf   ADRESL, w      ;
comf    STATUS, w      ; inverts carry flag (bit 0 of STATUS)
andlw   b'00000001'    ; masks all other bits
bcf     STATUS, RP0    ; goes to Bank 0
addlw   0x12           ; add this to the number we are
subwf   ADRESH, w      ; subtracting from the higher byte
btfss   STATUS, C      ;
goto    Cold           ; ADRESH:L < 0x1280, < "cold!"
etc . . .
```

We first subtract the lower byte of the threshold from the lower byte of the answer (ADRESL), leaving ADRESL unaffected. Note that we don't care what the answer is—only whether we had to borrow or not. If we borrow, the carry flag is clear. We invert all the bits of STATUS (including the carry flag), moving the result into the working register, then mask all bits other than bit 0 (which is now the carry flag—*inverted*). This means the working register is now 0 if there was no borrow, and 1 if there was a borrow. We can therefore add the working register (0 or 1) to the number we want to subtract from the higher byte, then subtract the total from ADRESH. Again, we're not interested in the answer itself, only in how the carry flag was affected. If it's clear, there was a borrow, meaning that the overall number split over ADRESH and ADRESL was less than **0x1280**, and the bath is therefore cold.

Each section (**Cold**, **OK** and **Hot**) should turn the correct LED on, and then return while enabling the global interrupt enable. Rather than copying out the code for restoring the values of the working register and STATUS, you can label this section **prereturn**, and jump to **prereturn** at the end of the three different sections.

The entire program is now complete, and shown in Appendix G. If you are using the *PICKit™ 1 Flash Starter Kit*, you can use the components already on the board to test the program, though you will have to change some of the pin assignments. You should change the analog input to GP0, and you can then simulate different temperatures by turning the potentiometer. The LEDs can be controlled on by making pins GP1, 2, 4 and 5 inputs, outputs 0 or outputs 1, as required. For example, to turn on LED0, make TRISIO **b'001111'** and GPIO **b'010000'**; to turn on LED1, make TRISIO **b'001111'** and GPIO **b'100000'**; and to turn on LED2, make TRISIO **b'101011'** and GPIO **b'010000'**.

10.5 Comparator Module

On the surface, the comparator module looks like a simplified version of analog to digital conversion. A comparator measures two analog inputs, called V_{IN+} and V_{IN-}, and produces a

digital output V_{OUT} depending on which voltage is bigger. In the standard configuration, $V_{OUT} = 1$ if $V_{IN+} > V_{IN-}$ and $V_{OUT} = 0$ if $V_{IN+} < V_{IN-}$. However, this behavior can be inverted when configuring the comparator.

Within this fairly basic type of operation, the PIC microcontroller offers a wide range of different possible forms of behavior which are controlled by bits 2:0 of **CMCON** (**C**omparator **M**odule **Con**trol register), and summarized in Table 10.3. On the PIC12F675, comparator inputs can be chosen from GP0/C_{IN}, GP1/C_{IN} or even a programmable internal voltage reference. V_{OUT} can be directly connected to pin GP2/C_{OUT}, or else can be released and used as a standard digital I/O pin. In the latter case, the comparator output can be read by the program as bit 6 of CMCON.

Table 10.3

CMCON 2:0		V_{IN+}	V_{IN-}	V_{OUT}
000		GP0/C_{IN+}	GP1/C_{IN-}	*Disabled: CMCON, 6 = 0*
001	**A**	GP0/C_{IN+}	GP1/C_{IN-}	GP2/C_{OUT} *and* CMCON, 6
010		GP0/C_{IN+}	GP1/C_{IN-}	CMCON, 6
011	**B**	Internal ref.	GP1/C_{IN-}	GP2/C_{OUT} *and* CMCON, 6
100		Internal ref.	GP1/C_{IN-}	CMCON, 6
101	**C**	Internal ref.	GP0 *or* GP1	GP2/C_{OUT} *and* CMCON, 6
110		Internal ref.	GP0 *or* GP1	CMCON, 6
111	*Comparator off and consumes no current (CMCON, 6 = 0)*			

If GP0 is not being used by the comparator (e.g., type **"B"** behavior), it can be used as a standard digital I/O pin. In the case where either GP0 or GP1 can be used as the V_{IN} input (i.e., type **"C"**), *both* are set as analog inputs. Note that when the PIC microcontroller is powered up or reset, CMCON 2:0 is 000, and even though the comparator is disabled (V_{OUT} is set to 0), pins GP0 and GP1 remain analog inputs and cannot be used as digital inputs. Hence, if you are not using the comparator, it should be turned off by setting CMCON 2:0 to 111. As is the case for any analog inputs, the voltage must be within the supply voltages V_{SS} and V_{DD}.

We've already discussed bits 0:2 and bit 6 of CMCON, and we will now examine the remaining bits. In type **"C"** behavior, bit 3 is the Comparator Input Switch which selects whether GP0 or GP1 is being measured. Finally, bit 4 is the Comparator Output Inversion bit—when this is set, any output from the comparator is inverted.

10.5.1 Voltage Reference

As well as comparing the states of external analog inputs, the comparator can use an internal programmable voltage reference. In order to use it, we must first turn on the voltage reference

Table 10.4

VRCON, 5 = 1 (Low Range)		VRCON, 5 = 0 (High Range)	
VRCON 3:0	VRef (V_{DD} = 5 V)	VRCON 3:0	VRef (V_{DD} = 5V)
0000	0.00	0000	1.25
0001	0.21	0001	1.41
0010	0.42	0010	1.56
0011	0.63	0011	1.72
0100	0.83	0100	1.88
0101	1.04	0101	2.03
0110	1.25	0110	2.19
0111	1.46	0111	2.34
1000	1.67	1000	2.50
1001	1.88	1001	2.66
1010	2.08	1010	2.81
1011	2.29	1011	2.97
1100	2.50	1100	3.13
1101	2.71	1101	3.28
1110	2.92	1110	3.44
1111	3.13	1111	3.59

module by setting bit 7 of the **VRCON** register (**V**oltage **R**eference **Con**trol). The voltage reference can take one of 32 distinct values, as given by bits 5 and 3:0 of VRCON. Bit 5 selects one of two voltage ranges; when set, the lower range of voltages is selected, and the reference is equal to $V_{DD} \times$ (VRCON 3:0)/24. When bit 5 is clear, the upper range is selected, and the reference is equal to $V_{DD}/4 + V_{DD} \times$ (VRCON 3:0)/32. Table 10.4 above shows example values for voltage references, for V_{DD} = 5V.

As with the comparator module, don't forget to turn off the voltage reference to save power if you aren't using it, or when going into sleep mode. On some PIC microcontrollers (e.g., the PIC16F627), this reference voltage can be output through an I/O pin.

10.5.2 Comparator Interrupts

The comparator interrupt triggers when the state of the comparator output changes. The corresponding interrupt enable is found in the PIE1 register (bit 3), and the interrupt flag is stored in the PIR1 register (bit 3)—this must be reset to 0 after the interrupt occurs. As with all peripheral interrupts, both the PIE Enable and Global Interrupt Enable bits must be set for this interrupt to occur. If you plan to change the comparator behavior during the program, you should disable the comparator interrupt during the change, to avoid the possibility of a false interrupt.

10.5.3 Comparator Example: Sun Follower

As a short example of the comparator feature, we will look at the program for a solar cell "sun follower." There are two sensors, on either side of the solar cell, which measure the light level and produce analog voltages between 0 and 5 V. As the sun rises and sets during the day, we want the solar cell to point directly towards the sunlight. The device therefore compares the signal coming from each sensor, and then drives a motor to make the two equal. The two light sensors are connected to the GP0/C_{IN} and GP1/C_{IN} pins. The motor is connected to pins GP2 and GP4; if GP2 is high and GP4 is low, the motor is driven forward, if GP4 is high and GP2 is low, the motor is driven backward. Every ten minutes a subroutine will be run to adjust the solar cell position. We begin by turning on the comparator module, and making the correct settings. We wish to compare GP0 and GP1 (i.e., type **"A"** operation) and do not require the C_{OUT} pin, hence CMCON 2:0 should be **b'010'**. We aren't using type **"C"** operation so bit 3 doesn't matter, and we don't wish to invert the comparator output so bit 4 should be 0:

```
FollowSun    bcf      STATUS, RP0    ; Bank 0
             movlw    b'00000010'    ; turns on comparator, compares
             movwf    CMCON          ; GP0 & GP1, not using C_OUT pin
```

The comparator response time can be as long as 10 s when a new input or voltage reference has been chosen, or when just turned on, so we need to insert a short delay before responding to the comparator output. An easy way to do this is to create a subroutine, **delay**, which immediately returns:

```
delay    return    ; immediately returns
```

Calling this subroutine and returning will take four clock cycles (or 4 s, given the internal 4 MHz internal oscillator). Depending on the comparator output, the motor is driven forward or in reverse:

```
             call     delay          ; kills four clock cycles (4 µs)
             call     delay          ; kills four clock cycles (4 µs)
             call     delay          ; kills four clock cycles (4 µs)
             bcf      PIR1, 3        ; resets comparator interrupt flag
             btfss    CMCON, 6       ; reads comparator output
             goto     Forward        ; drives motor forward
Reverse      bsf      GPIO, 4        ; drives motor in reverse
             goto     Continue       ;
Forward      bsf      GPIO, 2        ; drives motor forward
Continue     . . .
```

We then wait for a comparator interrupt (we don't actually need to use an interrupt – it's easier to simply test the comparator interrupt flag, which will get set even if the relevant interrupt enable bits are disabled). The comparator interrupt will take place when the comparator output changes – i.e., just when the values from the light sensors are approximately equal. At this

point we can stop the motors, and then return from the subroutine. This assumes the motors are sufficiently slow that overshoot isn't a problem.

```
Continue    btfss     PIR1, 3     ; waits for comparator to change
                                  ; output
            goto      Continue    ;
            bcf       GPIO, 2     ; turns off motor
            bcf       GPIO, 4     ;
            return                ; returns from 'FollowSun'
                                  ; subroutine
```

As well as showing how the comparator may be used, the above example illustrates how an interrupt flag may be used without actually involving a jump to the interrupt service routine.

10.5.4 Comparator Example: Reading Many Buttons from One Pin

We can also use the comparator to read a large number of buttons from only one input. If we connect the buttons as shown in Fig. 10.10, there is a different resistance between the GP1/C_{IN} pin and V_{DD}, depending on which button is pressed. We also place a capacitor between GP1 and ground, so that when a button is pressed there is a slow rise time dictated by the values of resistance and capacitance. Therefore, the rise time is different for each button. By using the comparator, we can set a particular threshold voltage for the GP1 input. We discharge the capacitor by making GP1 an output and setting it to 0, then we make it an input and set a timer going. By measuring the time at which the comparator output changes, we can identify which button was pressed (if any).

With the values given in Fig. 10.10, the discharge is very fast. The value of R × C for each button is: 8, 23, 38 or 53 μs, and is 68 μs if there is no button pressed. If we set up the comparator to trigger at about $(1 - 1/e) \times V_{DD} = 3.16$ V, the trigger times should be equal to the RC products given above. The code below could therefore be used to read the buttons:

```
Setup       bsf     STATUS, RP0   ; Bank 1
            movlw   b'10001000'   ; TMR0 not prescaled (counts up
            movwf   OPTION_REG    ; once every 1 s for 4 MHz clock)
            movlw   b'10001100'   ; programs an internal voltage
            movwf   VRCON         ; reference of 3.13 V
            bcf     STATUS, RP0   ; Bank 0
            movlw   b'00000100'   ; turns on comparator, to compare
            movwf   CMCON         ; GP1 with internal VREF

ButtonTest  bsf     STATUS, RP0   ; Bank 1
            bcf     TRISIO, 1     ; make GP1 output
            bcf     STATUS, RP0   ; Bank 0
            bcf     GPIO, 1       ; discharge capacitor
            clrf    TMR0          ; resets TMR0
            bsf     STATUS, RP0   ; Bank 1
```

```
                bsf     TRISIO, 1       ; make GP1 input ? TMR0 = 0
                bcf     STATUS, RP0     ; Bank 0
Loop            btfsc   CMCON, 6        ; waits until VIN > VREF
                goto    Loop            ;
                swapf   TMR0, w         ; takes current value of TMR0
                andlw   b'00000111'     ; takes only bits 4:6 of TMR0
                addwf   PCL, f          ; skips between 0 and 4
                goto    Button1         ; instructions, depending on
                goto    Button2         ; which button, if any, was
                goto    Button3         ; pressed
                goto    Button4         ;
                goto    NoButton        ; no button was pressed
```

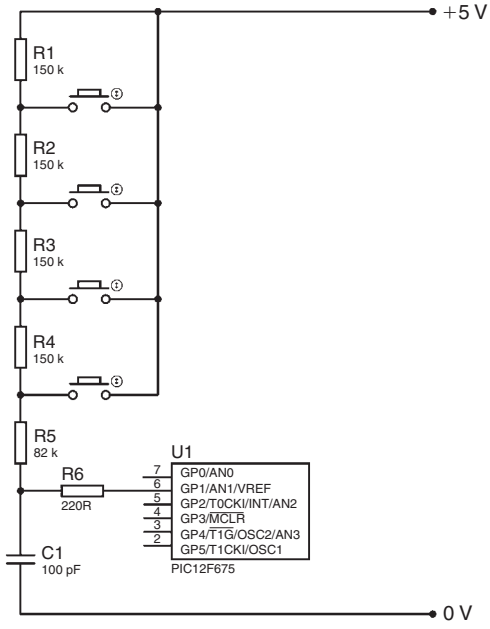


Figure 10.10

In the setup section we choose no prescaler for TMR0 (so, given a 4-MHz clock, it counts up every 1 s). We choose an internal voltage reference of 3.13 V, and turn on the comparator, selecting GP1/C_{IN} and V_{REF} as the comparator inputs, and using only CMCON, 6 as the comparator output. During the button test we first discharge the capacitor, then reset TMR0. Note that when you write to TMR0 (e.g., clear it), the change takes *two cycles* to take effect. Therefore TMR0 isn't actually cleared until the line in which GP1 is made an input, which is just what we want. We then enter a loop which waits until the comparator input goes high (the voltage has risen above the threshold). We then take the number from TMR0 and use it to jump to the appropriate section.

You will now notice that the resistor values were not chosen at random—if Button1 was pressed, the expected rise time is 8 μs. TMR0 counts up once every 1 s, so the expected value of TMR0 is 8. Even if there is a small error, we can be pretty certain the number in TMR0 is **b'0000????'**, i.e., the value of TMR0 bits 4:6 should be **000**. If Button2 was pressed, the expected rise time is 32 μs, which means TMR0 bits 4:6 are expected to be **001** (work it out!). So, our choice of resistor values means that even with some small timing errors, the values of TMR0 bits 4:6 will be: 000, 001, 010, 011 or 100 depending on which button (if any) has been pressed. Therefore, when the threshold voltage is reached, we *swap* the nibbles in TMR0 (making the bits of interest bits 0:2), leaving the result in the working register, and then mask the answer using the AND operation. This provides a number between 0 and 4 which can be added to the PC to branch to the different sections. The whole read operation therefore takes less than 100 μs, though an important drawback of this method is the inability to detect when more than one button is pressed. Finally, in practice, you should check carefully the real values of the resistors and capacitor you are using, and be prepared to play with the voltage reference value to achieve the desired behavior.

10.6 Final Project: Intelligent Garden Lights

We will bring together some of the ideas covered in this chapter in a final project: an intelligent garden lights unit (thanks to Max Horsey for the idea and original design). This device detects the ambient light level, and according to user programming, turns on the garden lights when it gets dark. The lights are automatically turned off around midnight. The user-programmed settings will be stored in the EEPROM, in case of loss of power. There will be an override button that allows the lights to be manually turned on and off, and a switch used to tell the device whether we are currently on "daylight savings time." The key behind this project is the rule-of-thumb that midnight roughly coincides, within 20 minutes or so, with the "solar midnight"—the halfway point between sunrise and sunset. In other words, the midpoint between the time for a particular light level in the evening and time for *the same* light level in the early morning is approximately midnight. For example, if it gets dark around 7 p.m., the light level should be approximately the same at 5 a.m. (so that the midpoint between these times is 12 a.m.). This means the device can calculate midnight without the need for the user to input a time (and the clumsy interface this may entail).

The override button will trigger an external INT interrupt and so will be connected to the GP2/INT pin. The garden lights will be controlled through GP5 (using a relay in the real device, or simply an LED in the test version). GP4 will control whether the day or night LED is on—which tells the user whether the device thinks it is currently day or night. The light sensor will be attached to GP0/AN0, and the summer/winter switch to GP1. Finally, when the PIC microcontroller is reset (using the GP3/$\overline{\text{MCLR}}$ pin) it will measure the current light level and use this as the threshold light level at which to turn on the garden lights. A button attached to this pin is therefore pressed to program the light level at which garden lights should be turned on. The flowchart for this project is shown in Fig. 10.11, and the circuit diagram in Fig. 10.12.

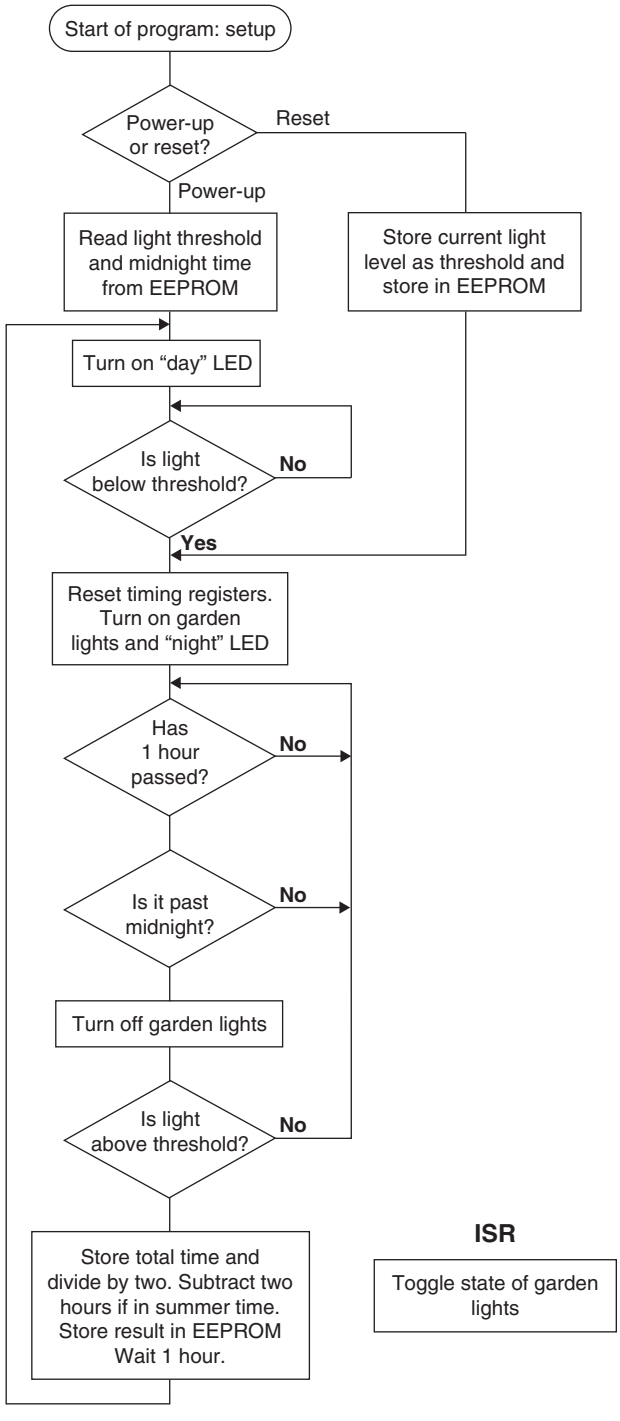

Start of program: setup
Power-up or reset?
Reset
Power-up
Read light threshold and midnight time from EEPROM
Store current light level as threshold and store in EEPROM
Turn on "day" LED
Is light below threshold?
No
Yes
Reset timing registers. Turn on garden lights and "night" LED
Has 1 hour passed?
No
Is it past midnight?
No
Turn off garden lights
Is light above threshold?
No
Store total time and divide by two. Subtract two hours if in summer time. Store result in EEPROM Wait 1 hour.
ISR
Toggle state of garden lights

Figure 10.11

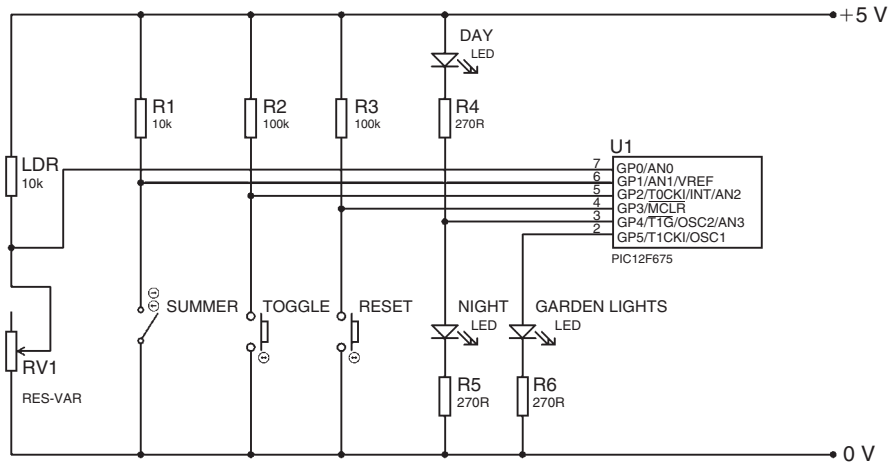


Figure 10.12

We'll go through the key steps of the flowchart and the program, but the actual writing of the program is left as an exercise. The program I wrote is shown in Appendix H, but yours may differ in parts. First, the POR bit in the **PCON** (**P**ower **Con**trol) register is used to determine whether the device has just powered up, or been reset by the $\overline{\text{MCLR}}$ pin. If the device has just been powered up, the POR bit will be 0 (and needs to be reset to 1), and the last-saved values can be read out from the EEPROM. The day LED is turned on and the program waits for the light to fall below the threshold (i.e., wait for dusk). However, if a reset occurred, the light level is measured and stored as the new threshold. The value for midnight is also reset. In this application, 10-bit accuracy is not required from the A/D converter, so the 8 bits in ADRESH can be used (given a left-justified A/D result).

Once the garden lights come on, a timer is started both to determine when to turn off the lights and also to measure when tomorrow's midnight will be. You can time in units of five minutes (using a marker of 125, and postscalers of 125 and 75). Five-minute accuracy is sufficient, and it allows you to time a whole night using one file register (it times a maximum of 5 mins $\times$ 256 = over 21 hours, which should be enough for most places, except perhaps a winter in Lapland!). Given that light levels may fluctuate slightly, we won't do anything at all for the first hour. After this point, we wait until midnight, in other words, we wait until time elapsed equals the previously estimated value for the time between dusk and midnight. We then turn off the garden lights (leaving the night LED on).

Finally, we test the light levels to wait for dawn (while keeping the timing going). When the light levels exceed the threshold, we store the elapsed time (the time from dusk until dawn) and divide it by two. This gives our estimate for the time between dusk and midnight. If we are in summer time (the clocks have gone forward one hour), our guess is out by two hours

and so we should subtract two hours from the estimate. This value should then be stored in the EEPROM, and the program should loop back. We should wait one hour before testing for dusk again, to minimize the risk of errors. The override button will trigger the INT interrupt, which simply has to toggle the state of GP5 to turn the garden lights on and off. This will not affect the normal operation of the device.

This program has combined the use of interrupts, A/D conversion and the EEPROM, and provided you with the opportunity to tackle a program on your own, with only basic guidance. You should now have the confidence and the tools with which to design and build your own PIC projects.

This page intentionally left blank

CHAPTER 11

Using Inputs

A control program usually requires more than turning outputs on and off. They switch on and off because an event has happened. This event is then connected to the input of the microcontroller to "tell" it what to do next. The input could be derived from a switch or it could come from a sensor measuring temperature, light levels, soil moisture, air quality, fluid pressure, engine speed, etc. In this chapter we will concern ourselves with digital on/off inputs. As an example let us design a circuit so that switch SW1 will turn an LED on and off. The circuit diagram is shown in Fig. 11.1. This circuit uses the 16F84 microcontroller with a 32-kHz crystal.

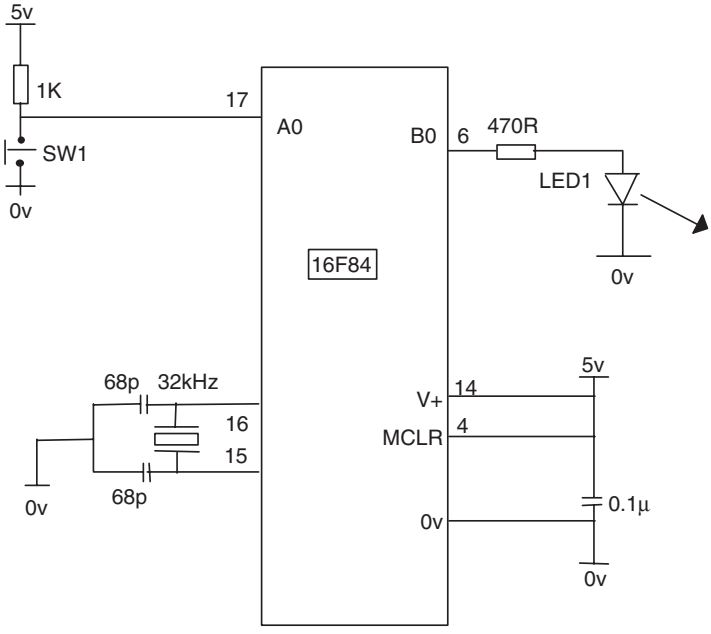


Figure 11.1: Circuit Diagram of the Microcontroller Switch

It can of course also be performed with any of the microcontrollers discussed previously, including the 16F818 using its internal oscillator, in which case the crystal and 2×68pF capacitors are not required.

The program to control the hardware would use the following steps:

1. Wait for SW1 to close.
2. Turn on LED1.
3. Wait for SW1 to open.
4. Turn off LED1.
5. Repeat.

In the circuit diagram SW1 is connected to A0 and LED1 to B0. When the switch is closed A0 goes low or clear, so we wait until A0 is clear. The code for this is:

```
BEGIN    BTFSC    PORTA,0 (test bit 0 in file PORTA skip if clear)
         GOTO     BEGIN
         BSF      PORTB,0
```

The command BTFSC is Bit Test in File Skip if Clear, and the instruction BTFSC PORTA,0 means Test the Bit in the File PORTA—i.e., Bit0, Skip the next instruction if Clear. If A0 is Clear Skip the next instruction (GOTO BEGIN); if it isn't Clear then do not Skip and GOTO BEGIN to check the switch again.

The program will check the switch thousands, perhaps millions, of times a second, depending on your clock. When the switch is pressed the program moves on and executes the instruction BSF PORTB,0 to turn on the LED.

We then wait for the switch to open. When the switch is open A0 goes Hi or Set, and we then wait until A0 is Set; i.e.,

```
SWOFF    BTFSS    PORTA,0
         GOTO     SWOFF
         BCF      PORTB,0
         GOTO     BEGIN
```

The command BTFSS is Bit Test in File Skip if Set, and the instruction BTFSS PORTA,0 means Test the Bit in the File PORTA; i.e., Bit0, Skip the next instruction if Set. If A0 is Set, Skip the next instruction (GOTO SWOFF); if it isn't Set, then do not Skip and GOTO SWOFF to check the switch again.

When the switch is set the program moves on and executes the instruction BCF PORTB,0 to switch off the LED. The program then goes back to the label BEGIN, to repeat. The program is now added to the header. (Use the TAB to make your listing easy to read.) It is then saved as SWITCH.ASM.

```
;SWITCH.ASM
;*******************************************************
;Program starts now.
BEGIN      BTFSC      PORTA,0    ;Wait for SW1 to be pressed
           GOTO       BEGIN
           BSF        PORTB,0    ;Turn on LED1.
SWOFF      BTFSS      PORTA,0    ;Wait for SW1 to be released.
           GOTO       SWOFF
           BCF        PORTB,0    ;Switch off LED1.
           GOTO       BEGIN      ;Repeat sequence.
END
```

11.1 Switch Flowchart

It will be obvious from the program listing of the solution to the switch problem that listings are difficult to follow. "A picture is worth a thousand words" has never been more apt than it is with a program listing. The picture of the program is shown in the flowchart for the solution to our initial switch problem; see Fig. 11.2. Before a programming listing is attempted it is very worthwhile drawing a flowchart to depict the program steps. Diamonds are used to show a decision (i.e., a branch) and rectangles are used to show a command. Each shape may take several lines of program to implement. But the idea of the flowchart should be evident. Note that the flowchart describes the problem—you can draw it without any knowledge of the instruction set.

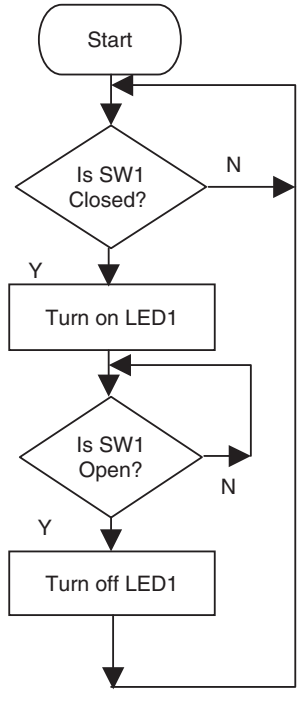


Figure 11.2: Flowchart for the Switch

11.2 Program Development

From our basic switch circuit, an obvious addition would be to include a delay so that the LED would go off automatically after a set time. Suppose we wish to switch the light on for 5 seconds, using A0 as the switch input. Figure 11.3 shows this delay flowchart.

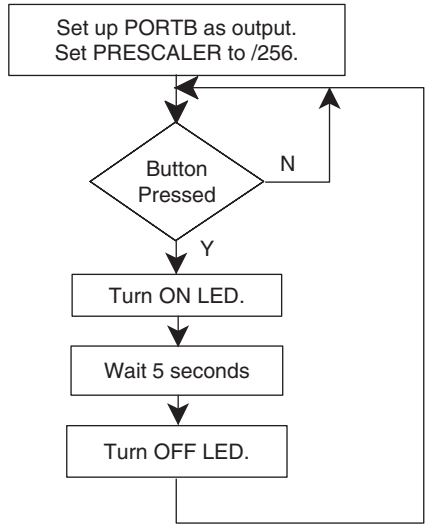


Figure 11.3: Delay Flowchart

The complete listing for this program for the 16F84 is shown below. I have given the complete code including the header because I have added a 5-second delay in the subroutine section.

```
;DELAY.ASM

;EQUATES SECTION

TMR0          EQU     1         ;TMR0 is FILE 1.
PORTA         EQU     5         ;PORTA is FILE 5.
PORTB         EQU     6         ;PORTB is FILE 6.
STATUS        EQU     3         ;STATUS is FILE3.
TRISA         EQU     85H       ;TRISA (the PORTA I/O selection)
TRISB         EQU     86H       ;TRISB (the PORTB I/O selection)
OPTION_R      EQU     81H       ;the OPTION register is file 81H
ZEROBIT       EQU     2         ;ZEROBIT is Bit 2.
COUNT         EQU     0CH       ;USER RAM LOCATION.
;*************************************************************
LIST          P=16F84           ;We are using the 16F84.
ORG           0                 ;0 is the start address.
GOTO          START             ;goto start!
```

```
;*********************************************************
;Configuration Bits
__CONFIG H'3FF0'        ;selects LP oscillator, WDT off, PUT on,
                        ;Code Protection disabled.
;*******************************************************
;SUBROUTINE SECTION.

;5 second delay.
DELAY5      CLRF      TMR0              ;Start TMR0.
LOOPA       MOVF      TMR0,W            ;Read TMR0 into W.
            SUBLW     .160              ;TIME -160
            BTFSS     STATUS,ZEROBIT    ;Check TIME-W=0
            GOTO      LOOPA             ;Time is not = 160.
            RETLW     0                 ;Time is 160, return.
;*********************************************************
;CONFIGURATION SECTION.

START       BSF       STATUS,5        ;Turn to BANK1
            MOVLW     B'00011111'     ;5 bits of PORTA are I/Ps.
            MOVWF     TRISA
            MOVLW     B'00000000'
            MOVWF     TRISB           ;PORTB IS OUTPUT
            MOVLW     B'00000111'
            MOVWF     OPTION_R        ;PRESCALER is /256
            BCF       STATUS,5        ;Return to BANK0
            CLRF      PORTA           ;Clears PORTA
            CLRF      PORTB           ;Clears PORTB
            CLRF      COUNT

;*********************************************************
;Program starts now.
ON      BTFSC     PORTA,0     ;Check button pressed.
        GOTO      ON
        BSF       PORTB,0     ;Turn on LED.
        CALL      DELAY5      ;CALL 5 second delay
        BCF       PORTB,0     ;Turn off LED.
        GOTO      ON          ;Repeat
END
```

11.2.1 How Does it Work?

We check to see if the switch has been pressed (clear). If not GOTO ON and check again. If it has, skip that line and turn on the LED on B0. The code is:

```
ON      BTFSC     PORTA,0     ;Check button pressed.
        GOTO      ON
        BSF       PORTB,0     ;Turn on LED.
```

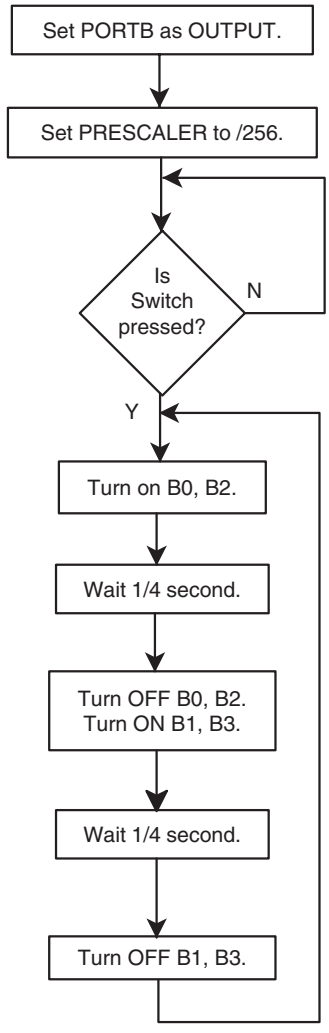


Figure 11.4: Flowchart for Problem

Wait 5 seconds. The 5-second delay has been included for you in the subroutine section. Here is the code:

```
CALL    DELAY5
```

Turn the LED off and go back to the beginning. The code is:

```
BCF     PORTB,0    ;Turn off LED.
GOTO    ON
```

Try this next problem for yourselves, before looking at the solution.

Problem 1: Using Port A bit 0 as a start button and outputs on PortB bits 0-3. Switch on Port B bits 0 and 2 for ¼ second, switch off bits 0 and 2.
Switch on Port B bits 1 and 3 for ¼ second, switch off bits 1and 3.
Repeat continuously. The ¼ second delay is provided for you.

The flowchart for the solution to Problem 1 is shown in Fig. 11.4.

11.2.1.1 Program Solution to Problem1 for the 16F84

```
;PROBLEM1.ASM

;EQUATES SECTION

TMR0          EQU      1         ;TMR0 is FILE 1.
PORTA         EQU      5         ;PORTA is FILE 5.
PORTB         EQU      6         ;PORTB is FILE 6.
STATUS        EQU      3         ;STATUS is FILE 3.
TRISA         EQU      85H       ;TRISA (the PORTA I/O selection)
TRISB         EQU      86H       ;TRISB (the PORTB I/O selection)
OPTION_R      EQU      81H       ;the OPTION register is file 81H
ZEROBIT       EQU      2         ;ZEROBIT is Bit 2.
COUNT         EQU      0CH       ;USER RAM LOCATION.

;*********************************************************

LIST          P=16F84             ;we are using the 16F84.
ORG           0                   ;the start address in memory is 0
GOTO          START               ;goto start!

;*********************************************************
;Configuration Bits

__CONFIG H'3FF0'        ;selects LP oscillator, WDT off, PUT on
                        ;Code Protection disabled.
;*********************************************************
;SUBROUTINE SECTION.

;0.25 second delay.
DELAY     CLRF      TMR0              ;START TMR0.
LOOPA     MOVF      TMR0,W            ;READ TMR0 INTO W.
          SUBLW     .8                ;TIME -8
          BTFSS     STATUS,ZEROBIT    ;Check TIME-W=0
          GOTO      LOOPA             ;Time is not =8.
          RETLW     0                 ;Time is 8, return.

;*********************************************************
```

```
;CONFIGURATION SECTION
START    BSF      STATUS,5        ;Turn to BANK1
         MOVLW    B'00011111'     ;5 bits of PORTA are I/Ps.
         MOVWF    TRISA
         MOVLW    B'00000000'
         MOVWF    TRISB           ;PORTB IS OUTPUT
         MOVLW    B'00000111'
         MOVWF    OPTION_R        ;PRESCALER is /256
         BCF      STATUS,5        ;Return to BANK0
         CLRF     PORTA           ;Clears PORTA
         CLRF     PORTB           ;Clears PORTB
;*********************************************************
;Program starts now.
ON         BTFSC    PORTA,0        ;Check button pressed.
           GOTO     ON
REPEAT     MOVLW    B'00000101'
           MOVWF    PORTB          ;Turn on bits 0 and 2
           CALL     DELAY          ;¼ second delay
           MOVLW    B'00001010'
           MOVWF    PORTB          ;Turn on bits 1 and 3
           CALL     DELAY          ;¼ second delay
           GOTO     REPEAT         ;Repeat
END
```

11.2.2 How Does it Work?

- Wait for the switch on PORTA,0 to clear, with BTFSC PORTA,0 then skip to
- MOVLW B'00000101' sets up the data in the W register.
- MOVWF PORTB transfers the W register to PORTB and puts 5v on B0 and B2 only.
- CALL DELAY waits for ¼ second.
- MOVLW B'00001010' sets up the data in the W register.
- MOVWF PORTB transfers the W register to PORTB and puts 5v on B1 and B3 only.
- CALL DELAY waits for ¼ second.
- GOTO REPEAT sends the program back to (my) label, REPEAT. This will keep the lights flashing all the time without checking the switch again.

Question: How do we make the program look at the switch, so that we can control whether or not the program repeats?

Answer: Instead of GOTO REPEAT use GOTO BEGIN. The program will then go to the label BEGIN instead of REPEAT and will wait for the switch to be Clear before repeating.

11.3 Scanning (Using Multiple Inputs)

Scanning (also called polling) is when the microcontroller looks at the condition of a number of inputs in turn and executes a section of program depending on the state of those inputs. Applications include:

- Burglar alarms—when sensors are monitored and a siren sounds either immediately or after a delay depending on which input is active.
- Keypad scanning—a key press could cause an LED to light, a buzzer to sound or a missile to be launched. Just do not press the wrong key!

Let's consider a simple example.

11.4 Switch Scanning

Design a circuit so that if a switch is pressed, a corresponding LED will light. I.e.,

If SW0 is Hi, (logic1 or Set) then LED0 is on.

If SW0 is Low, (logic 0 or Clear) then LED0 is off.

If SW1 is Hi, (logic1 or Set) then LED1 is on.

If SW1 is Low, (logic 0 or Clear) then LED1 is off.

etc.

The circuit diagram for this is shown in Fig. 11.5 and the corresponding flowchart in Fig. 11.6.

The program for this switch scan is:

```
;SWSCAN.ASM using 16F84 and 32kHz crystal.

;EQUATES SECTION

TMR0        EQU     1     ;TMR0 is FILE 1.
PORTA       EQU     5     ;PORTA is FILE 5.
PORTB       EQU     6     ;PORTB is FILE 6.
STATUS      EQU     3     ;STATUS is FILE3.
TRISA       EQU     85H   ;TRISA (the PORTA I/O selection)
TRISB       EQU     86H   ;TRISB (the PORTB I/O selection)
```

```
OPTION_R   EQU     81H   ;the OPTION register is file 81H
ZEROBIT    EQU     2     ;ZEROBIT is Bit 2.
COUNT      EQU     0CH   ;USER RAM LOCATION.
;*********************************************************
LIST       P=16F84       ;We are using the 16F84.
ORG        0             ;0 is the start address.
GOTO       START         ;goto start!
;*********************************************************
;Configuration Bits
__CONFIG H'3FF0'      ;selects LP oscillator, WDT off, PUT on,
                      ;Code Protection disabled.

;*****************************************************
;CONFIGURATION SECTION.

START     BSF       STATUS,5        ;Turn to BANK1
          MOVLW     B'00011111'     ;5 bits of PORTA are I/Ps.
          MOVWF     TRISA
          MOVLW     B'00000000'
          MOVWF     TRISB           ;PORTB IS OUTPUT
          MOVLW     B'00000111'
          MOVWF     OPTION_R        ;PRESCALER is /256
          BCF       STATUS,5        ;Return to BANK0
          CLRF      PORTA           ;Clears PORTA
          CLRF      PORTB           ;Clears PORTB
          CLRF      COUNT
;*********************************************************
;Program starts now.

SW0         BTFSC     PORTA,0    ;Switch0 pressed?
            GOTO      TURNON0    ;Yes
            BCF       PORTB,0    :No, Switch off LED0.

SW1         BTFSC     PORTA,1    ;Switch1 pressed?
            GOTO      TURNON1    ;Yes
            BCF       PORTB,1    :NO Switch off LED1.

SW2         BTFSC     PORTA,2    ;Switch2 pressed?
            GOTO      TURNON2    ;Yes
            BCF       PORTB,2    :NO Switch off LED2.

SW3         BTFSC     PORTA,3    ;Switch3 pressed?
            GOTO      TURNON3    ;Yes
            BCF       PORTB,3    :NO Switch off LED3.
            GOTO      SW0        ;Rescan.
```

```
TURNON0     BSF       PORTB,0    ;Turn on LED0
            GOTO      SW1

TURNON1     BSF       PORTB,1    ;Turn on LED1
            GOTO      SW2

TURNON2     BSF       PORTB,2    ;Turn on LED2
            GOTO      SW3

TURNON3     BSF       PORTB,3    ;Turn on LED3
            GOTO      SW0
END
```

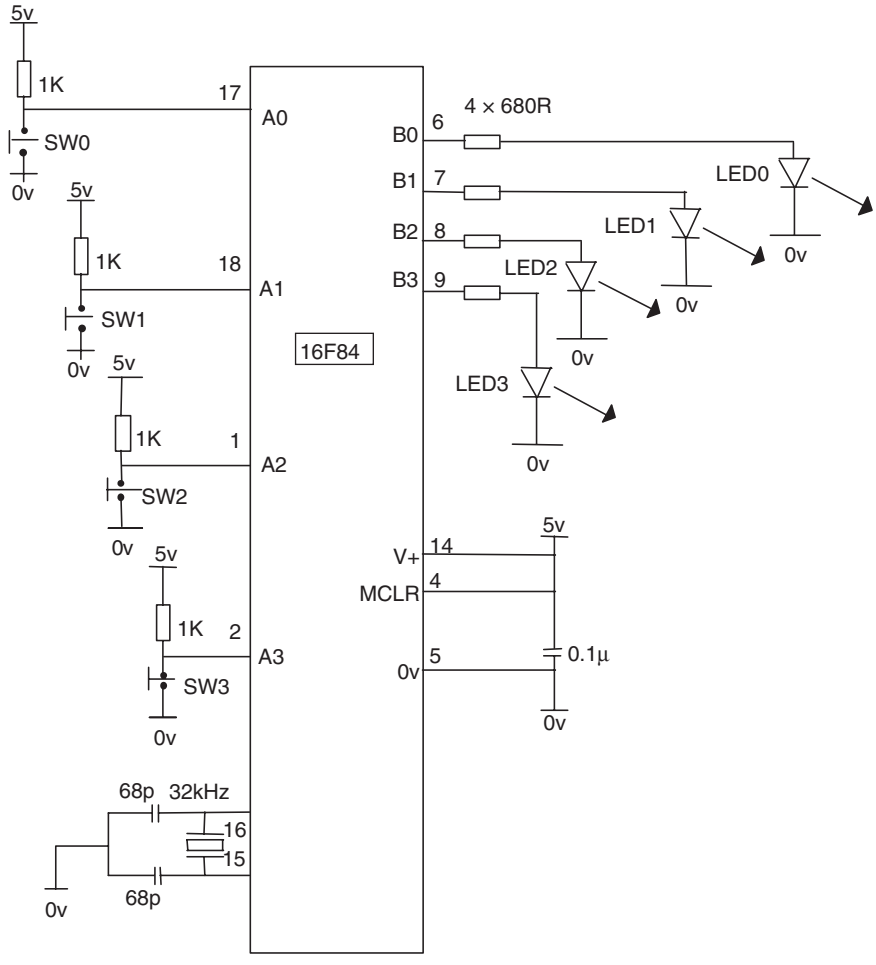


Figure 11.5: Switch Scanning Circuit

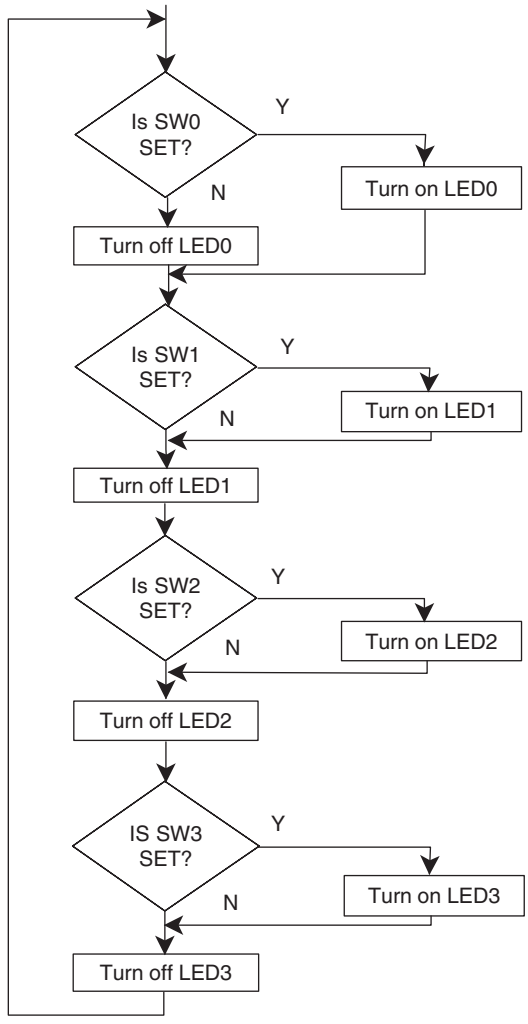


Figure 11.6: Flowchart for Switch Scan

11.4.1 How Does it Work?

SW0 is checked first with the instruction BTFSC PORTA,0. If the switch is closed when the program is executing this line, then we GOTO TURNON0. That is, the program jumps to the label TURNON0 which turns on LED0 and then jumps the program back to check SW1 at, of course, the label, SW1. SW1 is then checked in the same manner and then SW2 and SW3.

Suppose we press the switch when the program is not looking at it. The program lines are being executed at ¼ of the clock frequency— i.e., 32,768 Hz, that is, 8192 lines a second. The program will always catch you!

Try modifying the program so that the switches can flash four different routines; e.g., SW0 flashes all lights on and off 5 times for 1 second.

11.5 Control Application—A Hot Air Blower

The preceding section outlined how to monitor inputs by looking at them in turn. This application will "read" all the bits on the port at once, because we will be concerned with particular combinations of inputs rather than individual ones. The bits on the Input Port will be 0s or 1s and we can treat this binary pattern like any other number in a file.

Consider a controller for a hot air radiator. When the water is warm the fan will blow the warm air into the room. The heater and fan are controlled by three temperature sensors: (a) a room temperature sensor, (b) a boiler water temperature sensor and (c) a safety overheating sensor. The truth table for the system is shown in Table 11.1, where a 1 means hot and a 0 means cold for the sensors. The block diagram for the system is shown in Fig. 11.7.

Table 11.1: Truth Table for the Hot Air System

INPUTS								OUTPUTS	
A7	A6	A5	A4	A3	Room A2	Water A1	OverH A0	Heater B1	Fan B0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1
0	0	0	0	0	0	1	1	0	1
0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	1	0	1	0	1
0	0	0	0	0	1	1	0	0	0
0	0	0	0	0	1	1	1	0	1

Note A3, A4, A5, A6 and A7 are inputs and need to be connected to 0V. Do not leave them floating—you would not know if they were 0 or 1! Even though they are not being used they are still being read. (The inputs A5, A6 and A7 do not exist on the 16F84.)

There are eight input conditions from our three sensors. So all eight must be checked to determine which condition is true. Consider the first condition A2=A1=A0=0; i.e., PORTA reads 0000 0000. How do we know that PORTA is 0000 0000? We do not have an instruction that says "is PORTA equal to 0000 0000" or any other value, for that matter. So we need to look at our 35 instructions and come up with a way of finding out what is the binary value of PORTA.

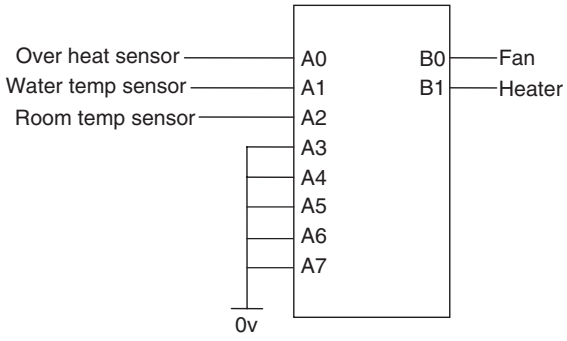


Figure 11.7: Block Diagram for the Hot Air System

We check for this condition by subtracting 00000000 from it; if the answer is zero then PORTA reads 00000000. I.e., 0000 0000 − 0000 0000=0 (obviously). But how do we subtract the two numbers and how do we know if the answer is zero?

This is a very important piece of programming, so read the next few lines carefully:

- We first of all read PORTA into the W register with the instruction MOVF PORTA,W that moves the data (setting of the switches, 1s or 0s) into W.
- We then subtract the number we are looking for, in this case 00000000, from W.
- We then need to know if the answer to this subtraction is zero. If it is, then the value on PORTA was 00000000. If the answer is not zero, then the value of the data on PORTA was not zero.
- So is the answer zero? Yes or No? The answer is held in a register called the Status Register, in bit 2 of this register, called the zero bit. If the zero bit, called a flag, is 1, it is indicating that the statement is true and the calculation was zero. If the zero bit is 0, that indicates the statement is false and the answer was not zero.
- We test the zero bit in the status register just like we tested the bit on the switch connected to PORTA at the start of this chapter. We use the command BTFSC and the instruction BTFSC STATUS,ZEROBIT. If the zero bit is clear, we skip the next instruction; if it is set, we have a match and do not skip.

The code for this is:

```
MOVLW      B'00000000'        ;put 000000 in W
SUBWF      PORTA              ;subtract W from PORTA
BTFSC      STATUS,ZEROBIT     ;PORTA =00000000?
CALL       CONDA              ;yes
```

CONDA is short for condition A, where we require the heater on and the fan off.

To check for A2 = A1 = 0 and A0 = 1 we subtract 00000001. To check for the next condition A2 = 0, A1 = 1, A0 = 0 we subtract 00000010, and so on for the other five conditions.

```
MOVLW     B'00000001'       ;put 00000001 in W
SUBWF     PORTA             ;subtract W from PORTA
BTFSS     STATUS,ZEROBIT    ;PORTA=00000001?
CALL      CONDB             ;yes

etc.
```

The opcode for this program CONTROL.ASM is:

```
;CONTROL.ASM

;SUBROUTINE SECTION.

CONDA     BCF       PORTB,0    ;turns fan off
          BSF       PORTB,1    ;turns heater on
          RETLW     0

CONDB     BSF       PORTB,0    ;turns fan on
          BCF       PORTB,1    ;turns heater off
          RETLW     0

CONDC     BSF       PORTB,0    ;turns fan on
          BSF       PORTB,1    ;turns heater on
          RETLW     0

CONDD     BCF       PORTB,0    ;turns fan off
          BCF       PORTB,1    ;turns heater off
          RETLW     0
;*********************************************************
;Program starts now.

BEGIN     MOVLW     B'00000000'       ;put 00000000 in W
          SUBWF     PORTA             ;PORTA -W
          BTFSC     STATUS,ZEROBIT    ;PORTA=00000000?
          CALL      CONDA             ;yes

          MOVLW     B'00000001'       ;put 00000001 in W
          SUBWF     PORTA             ;PORTA -W
          BTFSC     STATUS,ZEROBIT    ;PORTA=00000001?
          CALL      CONDB             ;yes

          MOVLW     B'00000010'       ;put 00000010 in W
          SUBWF     PORTA             ;PORTA -W
          BTFSC     STATUS,ZEROBIT    ;PORTA=00000010?
          CALL      CONDC             ;yes
```

```
        MOVLW   B'00000011'       ;put 00000011 in W
        SUBWF   PORTA             ;PORTA -W
        BTFSC   STATUS,ZEROBIT    ;PORTA=00000011?
        CALL    CONDB             ;yes

        MOVLW   B'000000100'      ;put 00000100 in W
        SUBWF   PORTA             ;PORTA -W
        BTFSC   STATUS,ZEROBIT    ;PORTA=00000100?
        CALL    CONDD             ;yes

        MOVLW   B'00000101'       ;put 00000101 in W
        SUBWF   PORTA             ;PORTA -W
        BTFSC   STATUS,ZEROBIT    ;PORTA =00000101?
        CALL    CONDB             ;yes

        MOVLW   B'00000110'       ;put 00000110 in W
        SUBWF   PORTA             ;PORTA -W
        BTFSC   STATUS,ZEROBIT    ;PORTA=00000110?
        CALL    CONDD             ;yes

        MOVLW   B'00000111'       ;put 00000111 in W
        SUBWF   PORTA             ;PORTA -W
        BTFSC   STATUS,ZEROBIT    ;PORTA=00000111?
        CALL    CONDB             ;yes

        GOTO    BEGIN

END
```

Notice that the SUBROUTINE SECTION needs to be changed to include the conditions CONDA, CONDB, CONDC and CONDD. The DELAY subroutines are not required in this example. The program can be checked by using switches for the input sensors and LEDs for the outputs.

There is more than one way of skinning a cat; another way of writing this program is shown in Chapter 13, in the section on look-up tables.

CHAPTER 12

Keypad Scanning

Keypads are an excellent way of entering data into the microcontroller. The keys are usually numbered, but they could be labeled as function keys—for example, in a remote control handset in a TV to adjust the sound or color, etc. As well as remote controls, keypads find applications in burglar alarms, door entry systems, calculators, microwave ovens, etc. So there is no shortage of applications for this section.

Keypads are usually arranged in a matrix format to reduce the number of I/O connections. A 12-key keypad is arranged in a 3×4 format requiring 7 connections. A 16-key keypad is arranged in a 4×4 format requiring 8 connections.

Consider the 12-key keypad. This is arranged in 3 columns and 4 rows as shown in Table 12.1. There are 7 connections to the keypad—C1, C2, C3, R1, R2, R3 and R4.

This connection to the microcontroller is shown in Fig. 12.1. The keypad works in the following way: If key 6 is pressed, then B2 will be joined to B4. For key 1, B0 would be joined to B3, etc., as shown in Fig. 12.1. The microcontroller would set B0 low and scan B3, B4, B5 and B6 for a low to see if keys 1, 4, 7 or * had been pressed.

The micro would then set B1 low and scan B3, B4, B5 and B6 for a low to see if keys 2, 5, 8 or 0 had been pressed. Finally B2 would be set low and B3, B4, B5 and B6 scanned for a low to see if keys 3, 6, 9 or # had been pressed.

12.1 Programming Example for the Keypad

As a programming example, when key 1 is pressed, display a binary 1 on PORTA, when key 2 is pressed display a binary 2 on PORTA, etc. Key 0 displays 10. Key * displays 11. Key # displays 12. This program could be used as a training aid for decimal to binary conversion. The flowchart is shown in Fig. 12.2.

Table 12.1: 12-key Keypad

	Column1, C1	Column2, C2	Column3, C3
Row1, R1	1	2	3
Row2, R2	4	5	6
Row3, R3	7	8	9
Row4, R4	*	0	#

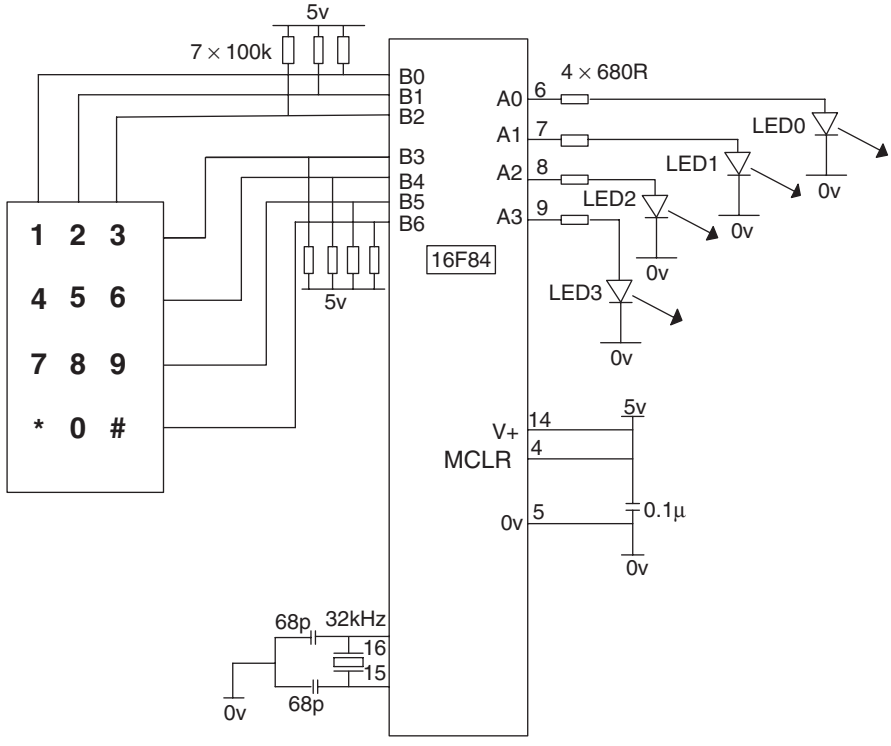


Figure 12.1: Keypad Connection to the Microcontroller

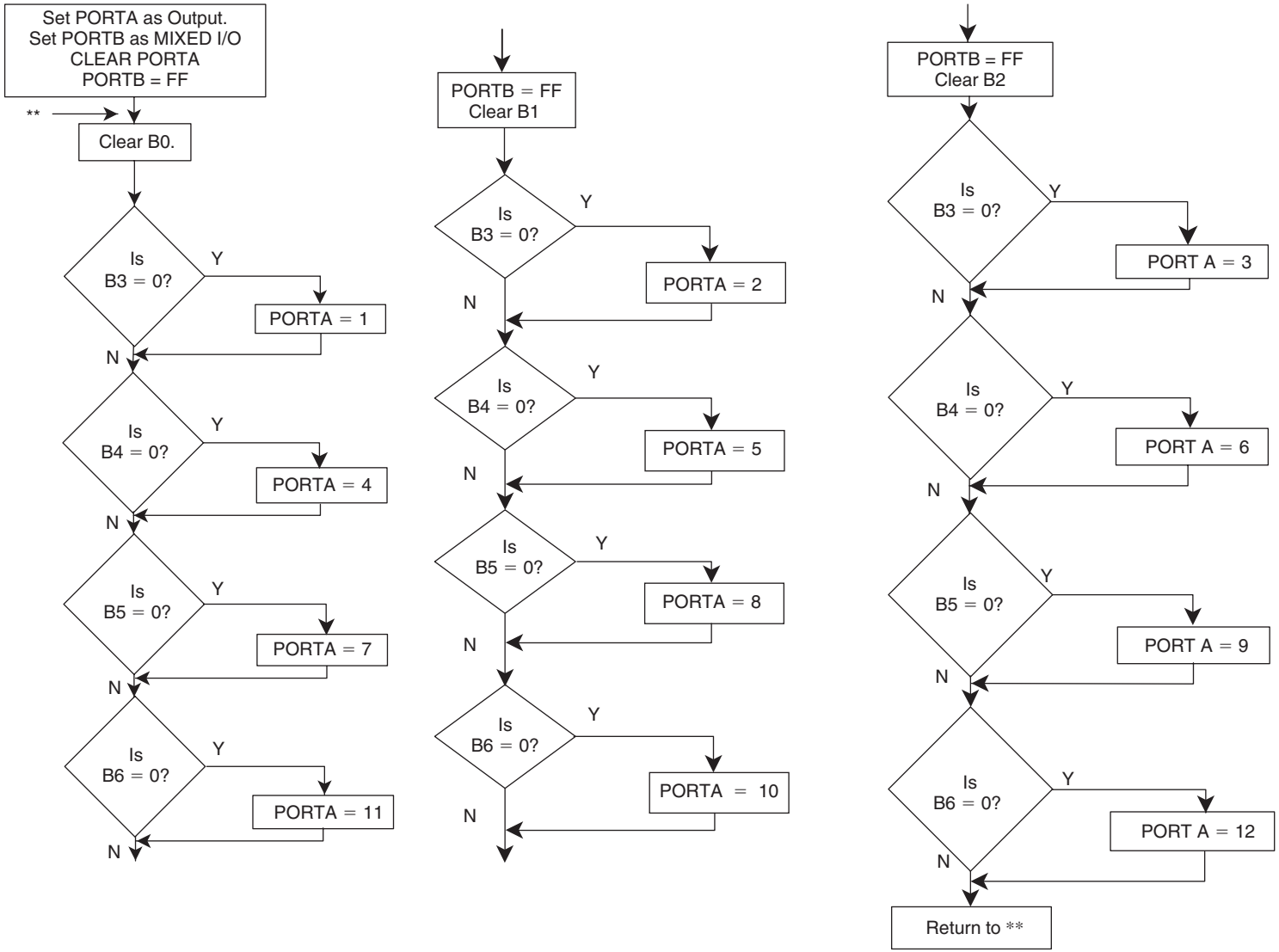


Figure 12.2: Keypad Scanning Flowchart

PORTB has internal pull-up resistors so that the resistors connected to PORTB in Fig. 12.1 are not required.

```
;KEYPAD.ASM

;EQUATES SECTION

STATUS      EQU    3       ;means STATUS is file 3.
PORTA       EQU    5       ;means PORTA is file 5.
PORTB       EQU    6       ;means PORTB is file 6.
TRISA       EQU    85H
TRISB       EQU    86H
OPTION_R    EQU    81H

;*********************************************************
LIST      P=16F84       ;we are using the 16F84.
ORG       0             ;the start address in memory is 0
GOTO      START         ;goto start!

;*********************************************************
;CONFIGURATION BITS

__Config H'3FF0'    ;selects LP Oscillator, WDT off,
                    ;Put on,
                    ;code protection disabled.
;*********************************************************
;CONFIGURATION SECTION

START       BSF       STATUS,5      ;Turns to Bank1.
            MOVLW     B'00000000'   ;PORTA is OUTPUT
            MOVWF     TRISA
            MOVLW     B'11111000'
            MOVWF     TRISB         ;PORTB is mixed I/O.
            BCF       OPTION_R,7    ;Turn on pull ups.
            BCF       STATUS,5      ;Return to Bank0.
            CLRF      PORTA         ;Clears PortA.
            CLRF      PORTB         ;Clears PortB.
;*********************************************************
;Program starts now.

COLUMN1     BCF       PORTB,0       ;Clear B0
            BSF       PORTB,1       ;Set B1
            BSF       PORTB,2       ;Set B2
CHECK1      BTFSC     PORTB,3       ;Is B3 Clear?
            GOTO      CHECK4        ;No
            MOVLW     .1            ;Yes, output 1.
            MOVWF     PORTA
```

```
CHECK4      BTFSC     PORTB,4       ;Is B4 Clear?
            GOTO      CHECK7        ;No
            MOVLW     .4            ;Yes, output 4.
            MOVWF     PORTA
CHECK7      BTFSC     PORTB,5       ;Is B5 Clear?
            GOTO      CHECK11       ;No
            MOVLW     .7            ;Yes, output 7.
            MOVWF     PORTA
CHECK11     BTFSC     PORTB,6       ;Is B6 Clear?
            GOTO      COLUMN2       ;No
            MOVLW     .11           ;Yes, output 11.
            MOVWF     PORTA
COLUMN2     BSF       PORTB,0       ;Set B0
            BCF       PORTB,1       ;Clear B1
            BSF       PORTB,2       ;Set B2
CHECK2      BTFSC     PORTB,3       ;Is B3 Clear?
            GOTO      CHECK5        ;No
            MOVLW     .2            ;Yes, output 2.
            MOVWF     PORTA
CHECK5      BTFSC     PORTB,4       ;Is B4 Clear?
            GOTO      CHECK8        ;No
            MOVLW     .5            ;Yes, output 5.
            MOVWF     PORTA
CHECK8      BTFSC     PORTB,5       ;Is B5 Clear?
            GOTO      CHECK10       ;No
            MOVLW     .8            ;Yes, output 8.
            MOVWF     PORTA
CHECK10     BTFSC     PORTB,6       ;Is B6 Clear?
            GOTO      COLUMN3       ;No
            MOVLW     .10           ;Yes, output 10.
            MOVWF     PORTA

COLUMN3     BSF       PORTB,0       ;Set B0
            BSF       PORTB,1       ;Set B1
            BCF       PORTB,2       ;Clear B2
CHECK3      BTFSC     PORTB,3       ;Is B3 Clear?
            GOTO      CHECK6        ;No
            MOVLW     .3            ;Yes, output 3.
            MOVWF     PORTA
CHECK6      BTFSC     PORTB,4       ;Is B4 Clear?
            GOTO      CHECK9        ;No
            MOVLW     .6            ;Yes, output 6.
            MOVWF     PORTA
CHECK9      BTFSC     PORTB,5       ;Is B5 Clear?
            GOTO      CHECK12       ;No
            MOVLW     .9            ;Yes, output 9.
            MOVWF     PORTA
```

```
CHECK12     BTFSC     PORTB,6       ;Is B6 Clear?
            GOTO      COLUMN1       ;No
            MOVLW     .12           ;Yes, output 12.
            MOVWF     PORTA
            GOTO      COLUMN1       ;Start scanning again.

END
```

12.1.1 How Does the Program Work?

12.1.1.1 Port Configuration

The first thing to note about the keypad circuit is that the PORTA pins are being used as outputs. On PORTB, pins B0, B1 and B2 are outputs and B3, B4, B5 and B6 are inputs. So PORTB is a mixture of inputs and outputs. The HEADER84.ASM program has to be modified to change to this new configuration.

To change PORTA to an output port, the following two lines are used in the Configuration Section:

```
MOVLW     B'00000000'     ;PORTA is OUTPUT
MOVWF     TRISA
```

To configure PORTB as a mixed input and output port, the following two lines are used in the Configuration Section:

```
MOVLW     B'11111000'
MOVWF     TRISB        ;PORTB is mixed I/O. B0,B1,B2 are O/P.
```

12.1.2 Scanning Routine

The scanning routine looks at each individual key in turn to see if one is being pressed. Because it can do this so quickly, it will notice we have pressed a key even if we press it quickly. The scanning routine first of all looks at the keys in column1—i.e., 1, 4, 7 and *. It does this by setting B0 low, B1 and B2 high. If a 1 is pressed, the B3 will be low; if a 1 is not pressed then B3 will be high, because pressing a 1 connects B0 and B3. Similarly, if 4 is pressed B4 will be low; if not B4 will be high. If 7 is pressed B5 will be low; if not B5 will be high. If * is pressed B6 will be low; if not B6 will be high.

In other words, when we set B0 low if any of the keys in column1 are pressed then the corresponding input to the microcontroller will go low and the program will output the binary

number equivalent of the key that has been pressed. If none of the keys in column 1 are pressed, then we move on to column2.

The code for scanning column1 is as follows:

These 3 lines set up PORTB with B0=0, B1=1 and B2=1.

```
COLUMN1     BCF          PORTB,0      ;Clear B0
            BSF          PORTB,1      ;Set B1
            BSF          PORTB,2      ;Set B2
```

These next 4 lines test input B3 to see if it is clear; if it is, then a 1 is placed on PORTA, and then the program continues. If B3 is set, then we proceed to check to see if key 4 has been pressed, with CHECK4.

```
CHECK1      BTFSC       PORTB,3      ;Is B3 Clear?
            GOTO        CHECK4       ;No
            MOVLW       .1           ;Yes, output 1
            MOVWF       PORTA        ;to PORTA
```

These next 4 lines test input B4 to see if it is clear; if it is, then a 4 is placed on PORTA, and then the program continues. If B4 is set, then we proceed to check to see if key 7 has been pressed, with CHECK7.

```
CHECK4      BTFSC       PORTB,4      ;Is B4 Clear?
            GOTO        CHECK7       ;No
            MOVLW       .4           ;Yes, output 4.
            MOVWF       PORTA
```

These next 4 lines test input B5 to see if it is clear; if it is then a 7 is placed on PORTA, and then the program continues. If B5 is set then we proceed to Check to see if key * has been pressed, with CHECK11.

```
CHECK7      BTFSC       PORTB,5      ;Is B5 Clear?
            GOTO        CHECK11      ;No
            MOVLW       .7           ;Yes, output 7.
            MOVWF       PORTA
```

These next 4 lines test input B6 to see if it is clear; if it is then an 11 is placed on PORTA, then the program continues. If B5 is set then we proceed to check the keys in column2, with COLUMN2.

```
CHECK11     BTFSC       PORTB,6      ;Is B6 Clear?
            GOTO        COLUMN2      ;No
            MOVLW       .11          ;Yes, output 11.
            MOVWF       PORTA
```

These 3 lines set up PORTB with B0=1, B1=0 and B2=1.

```
COLUMN2    BSF         PORTB,0      ;Set B0
           BCF         PORTB,1      ;Clear B1
           BSF         PORTB,2      ;Set B2
```

We then check to see if key2 has been pressed by testing to see if B3 is clear, if it is then a 2 is placed on PORTA and the program continues. If B3 is set then we proceed with CHECK5. This code is:

```
CHECK2     BTFSC       PORTB,3      ;Is B3 Clear?
           GOTO        CHECK5       ;No
           MOVLW       .2           ;Yes, output 2.
           MOVWF       PORTA
```

The program continues in the same manner checking 5, 8 and 10 (0), then moving onto column3 to check for 3, 6, 9 and 12 (#). After completing the scan, the program then goes back to continue the scan again.

It takes about 45 lines of code to complete a scan of the keypad. With a 32,768-Hz crystal, the lines of code are executed at a quarter of this speed—i.e., 8192 lines per second, so the scan time is 45/8192=5.5 ms. This is why, no matter how quickly you press the key, the microcontroller will be able to detect it.

12.1.3 Security Code

One of the most useful applications of a keypad is to enter a code to turn something on and off, such as a burglar alarm or door entry system. In the following program, KEYS3.ASM, the subroutine SCAN scans the keypad, waits for a key to be pressed, waits 0.1 seconds for the bouncing to stop, waits for the key to be released, waits 0.1 seconds for the bouncing to stop and then returns with the key number in W, which can then be transferred into a file. This is then used as a security code to turn on an LED (PORTA,0) when 3 digits (137) have been pressed, and turn the LED off again when the same 3 digits are pressed. You can, of course, use any 3 digits.

```
;KEYS3.ASM

;EQUATES SECTION

ZEROBIT      EQU     2
TMR0         EQU     1
STATUS       EQU     3          ;means STATUS is file 3.
PORTA        EQU     5          ;means PORTA is file 5.
PORTB        EQU     6          ;means PORTB is file 6.
TRISA        EQU     85H
TRISB        EQU     86H
```

```
OPTION_R    EQU     81H
NUM1        EQU     0CH
NUM2        EQU     0DH
NUM3        EQU     0EH
;*********************************************************
LIST        P=16F84       ;we are using the 16F84.
ORG         0             ;the start address in memory is 0
GOTO        START         ;goto start!
;*********************************************************
;SUB-ROUTINE SECTION

SCAN        NOP

COLUMN1     BCF         PORTB,0     ;Clear B0

            BSF         PORTB,1     ;Set B1
            BSF         PORTB,2     ;Set B2

CHECK1      BTFSC       PORTB,3     ;Is B3 Clear?
            GOTO        CHECK4      ;No
            CALL        DELAYP1
CHECK1A     BTFSS       PORTB,3
            GOTO        CHECK1A
            CALL        DELAYP1
            RETLW       .1

CHECK4      BTFSC       PORTB,4     ;Is B4 Clear?
            GOTO        CHECK7      ;No
            CALL        DELAYP1
CHECK4A     BTFSS       PORTB,4
            GOTO        CHECK4A
            CALL        DELAYP1
            RETLW       .4

CHECK7      BTFSC       PORTB,5     ;Is B5 Clear?
            GOTO        CHECK11     ;No
            CALL        DELAYP1
CHECK7A     BTFSS       PORTB,5
            GOTO        CHECK7A
            CALL        DELAYP1
            RETLW       .7

CHECK11     BTFSC       PORTB,6     ;Is B6 Clear?
            GOTO        COLUMN2     ;No
            CALL        DELAYP1
```

```
CHECK11A    BTFSS       PORTB,6
            GOTO        CHECK11A
            CALL        DELAYP1
            RETLW       .11

COLUMN2     BSF         PORTB,0     ;Set B0
            BCF         PORTB,1     ;Clear B1
            BSF         PORTB,2     ;Set B2

CHECK2      BTFSC       PORTB,3     ;Is B3 Clear?
            GOTO        CHECK5      ;No
            CALL        DELAYP1
CHECK2A     BTFSS       PORTB,3
            GOTO        CHECK2A
            CALL        DELAYP1
            RETLW       .2          ;Yes, output 2.

CHECK5      BTFSC       PORTB,4     ;Is B4 Clear?
            GOTO        CHECK8      ;No
            CALL        DELAYP1
CHECK5A     BTFSS       PORTB,4
            GOTO        CHECK5A
            CALL        DELAYP1
            RETLW       .5          ;Yes, output 5.

CHECK8      BTFSC       PORTB,5     ;Is B5 Clear?
            GOTO        CHECK0      ;No
            CALL        DELAYP1
CHECK8A     BTFSS       PORTB,5
            GOTO        CHECK8A
            CALL        DELAYP1
            RETLW       .8          ;Yes, output 8.

CHECK0      BTFSC       PORTB,6     ;Is B6 Clear?
            GOTO        COLUMN3     ;No
            CALL        DELAYP1
CHECK0A     BTFSS       PORTB,6
            GOTO        CHECK0A
            CALL        DELAYP1
            RETLW       0           ;Yes, output 10.

COLUMN3     BSF         PORTB,0     ;Set B0
            BSF         PORTB,1     ;Set B1
            BCF         PORTB,2     ;Clear B2

CHECK3      BTFSC       PORTB,3     ;Is B3 Clear?
            GOTO        CHECK6      ;No
            CALL        DELAYP1
```

```
CHECK3A     BTFSS       PORTB,3
            GOTO        CHECK3A
            CALL        DELAYP1
            RETLW       .3          ;Yes, output 3.

CHECK6      BTFSC       PORTB,4     ;Is B4 Clear?
            GOTO        CHECK9      ;No
            CALL        DELAYP1
CHECK6A     BTFSS       PORTB,4
            GOTO        CHECK6A
            CALL        DELAYP1
            RETLW       .6          ;Yes, output 6.

CHECK9      BTFSC       PORTB,5     ;Is B5 Clear?
            GOTO        CHECK12     ;No
            CALL        DELAYP1
CHECK9A     BTFSS       PORTB,5
            GOTO        CHECK9A
            CALL        DELAYP1
            RETLW       .9          ;Yes, output 9.

CHECK12     BTFSC       PORTB,6     ;Is B6 Clear?
            GOTO        COLUMN1     ;No
            CALL        DELAYP1
CHECK12A    BTFSS       PORTB,6
            GOTO        CHECK12A
            CALL        DELAYP1
            RETLW       .12         ;Yes, output 12.

;3/32 second delay.
DELAYP1   CLRF        TMR0          ;Start TMR0.
LOOPD     MOVF        TMR0,W        ;Read TMR0 into W.
          SUBLW       .3            ;TIME-3

          BTFSS       STATUS,ZEROBIT  ;Check TIME-W=0
          GOTO        LOOPD         ;Time is not=3.
          RETLW       0             ;Time is 3, return.
;*****************************************************
;CONFIGURATION SECTION

START     BSF         STATUS,5      ;Turns to Bank1.
          MOVLW       B'00000000'   ;PORTA is OUTPUT
          MOVWF       TRISA
          MOVLW       B'11111000'
          MOVWF       TRISB         ;PORTB is mixed I/O.
```

```
          MOVLW       B'00000111'
          MOVWF       OPTION_R
          BCF         STATUS,5        ;Return to Bank0.
          CLRF        PORTA           ;Clears PortA.
          CLRF        PORTB           ;Clears PortB.
;*****************************************************
;Program starts now.
;Enter 3 digit code here
          MOVLW       1               ;First digit
          MOVWF       NUM1
          MOVLW       3               ;Second digit
          MOVWF       NUM2
          MOVLW       7               ;Third digit
          MOVWF       NUM3
BEGIN     CALL        SCAN            ;Get 1st number
          SUBWF       NUM1,W
          BTFSS       STATUS,ZEROBIT  ;IS NUMBER=1?
          GOTO        BEGIN           ;No

          CALL        SCAN            ;Get 2nd number
          SUBWF       NUM2,W
          BTFSS       STATUS,ZEROBIT  ;IS NUMBER=3?
          GOTO        BEGIN           ;No

          CALL        SCAN            ;Get 3rd number.
          SUBWF       NUM3,W
          BTFSS       STATUS,ZEROBIT  ;IS NUMBER=7?
          GOTO        BEGIN           ;No
          BSF         PORTA,0         ;Turn on LED, 137 entered

TURN_OFF  CALL        SCAN            ;Get 1st number again
          SUBWF       NUM1,W
          BTFSS       STATUS,ZEROBIT  ;IS NUMBER=1?
          GOTO        TURN_OFF        ;No

          CALL        SCAN            ;Get 2nd number
          SUBWF       NUM2,W
          BTFSS       STATUS,ZEROBIT  ;IS NUMBER=3?
          GOTO        TURN_OFF        ;No

          CALL        SCAN            ;Get 3rd number.
          SUBWF       NUM3,W
          BTFSS       STATUS,ZEROBIT  ;IS NUMBER=7?
          GOTO        TURN_OFF        ;No
```

```
        BCF         PORTA,0         ;Turn off LED.
        GOTO        BEGIN
END
```

12.1.4 How Does the Program Work?

The ports are configured as in the previous code KEYPAD.ASM. The KEYS3.ASM program looks for the first key press and then it compares the number pressed with the required number stored in a user file called NUM1. It then looks for the second key to be pressed. But because the microcontroller is so quick, the first number could be stored and the program looks for the second number, but our finger is still pressing the first number.

12.1.5 Anti-Bounce Routine

Also when a mechanical key is pressed or released, it does not make or break cleanly; it bounces around. If the micro is allowed to, it is fast enough to see these bounces as key presses, so we must slow it down, as follows:

- We look first of all for the switch to be pressed.
- Then wait 0.1 seconds for the switch to stop bouncing.
- We then wait for the switch to be released.
- We then wait 0.1 seconds for the bouncing to stop before continuing.

The switch has then been pressed and released, indicating one action. The 0.1-second delay is written in the Header as DELAYP1.

12.1.6 Scan Routine

The scan routine used in KEYS3.ASM is written into the subroutine. When called, it waits for a key to be pressed and then returns with the number just pressed in W. It can be copied and used as a subroutine in any program using a keypad.

- The scan routine checks for key presses as in the previous example KEYPAD.ASM, Column1 checks for the numbers 1, 4, 7 and 11 being pressed in turn.
- If the 1 is not pressed, then the routine goes on to check for a 4.
- If the 1 is pressed, then the routine waits 0.1 second for the bouncing to stop.
- The program then waits for the key to be released.
- Waits again 0.1 seconds for the bouncing to stop.
- Then returns with a value of 1 in W.

Code for CHECK1:

```
CHECK1      BTFSC       PORTB,3      ;Is B3 Clear? Pressed?
            GOTO        CHECK4       ;No
            CALL        DELAYP1      ;Antibounce delay, B3 clear
CHECK1A     BTFSS       PORTB,3      ;Is B3 Set? Released?
            GOTO        CHECK1A      ;No
            CALL        DELAYP1      ;Antibounce delay, B3 Set
            RETLW       .1           ;Return with 1 in W.
```

If numbers 4, 7 or 11 are pressed the routine will return with the corresponding value in W. If no numbers in column1 are pressed then the scan routine continues on to column2 and column3. If no keys are pressed then the routine loops back to the start of the scan routine to continue checking.

12.1.7 Storing the Code

The code—i.e., 137—is stored in the files NUM1, NUM2, NUM3 with the following code:

```
MOVLW       1            ;First digit
MOVWF       NUM1
MOVLW       3            ;Second digit
MOVWF       NUM2
MOVLW       7            ;Third digit
MOVWF       NUM3
```

12.1.8 Checking for the Correct Code

We first of all CALL SCAN to collect the first digit, which returns with the number pressed in W. We then subtract the value of W from the first digit of our code stored in NUM1 with:

```
SUBWF NUM1,W.
```

This means SUBtract W from the File NUM1. The (,W) stores the result of the subtraction in W. Without (,W) the result would have been stored in NUM1 and the value changed!

We then check to see if NUM1 and W are equal—i.e., a correct match. In this case the zerobit in the status register would be set, indicating the result NUM1 − W=zero. This is done with:

```
BTFSS STATUS,ZEROBIT
```

We skip and carry on if it is set, i.e., a match. If it isn't, we return to BEGIN to scan again. With a correct first press, we then carry on checking for a second and if correct a third press to match the correct code. When the correct code is pressed we turn on our LED with:

```
B0SF PORTA,0
```

We then run through a similar sequence and wait for the code to turn off the LED.

Notice that if you enter an incorrect digi, you return to BEGIN or TURN_OFF. If you forget what key you have pressed, then press an incorrect one and start again.

You could, of course, modify this program by adding a fourth digit to the program and then turn on the LED, in which case you use another user file called NUM4. You could, of course, use a different code for switching off the output. You could also beep a buzzer for half a second to give yourself an audible feedback that you have pressed a button. As an extra security measure you could wait for a couple of seconds if an incorrect key had been pressed, or wait for 2 minutes if three wrong numbers had been entered.

The keypad routine opens up many different circuit applications. The SCAN routine can be copied and then pasted into any program using the keypad. Then when you CALL SCAN the program will return with the number pressed in W for you to do with it as you wish.

This page intentionally left blank

CHAPTER 13

Program Examples

13.1 Counting Events

Counting is a useful feature for any control circuit. We may wish to count the number of times a door has opened or closed, or count a number of pulses from a rotating disc. If we count cars into a car park we would increment a file count every time a car entered, using the instruction INCF COUNT. If we needed to know how many cars were in the car park, we would of course have to reduce the count by one every time a car left. We would do this by DECF COUNT. To clear the user file COUNT to start, we would CLRF COUNT. In this way the file count would store the number of cars in the car park.

Let's look at an application.

Design a circuit that will count 10 presses of a switch and then turn an LED on and reset when the next ten presses are started. The hardware is that of Fig. 11.1 with A0 as the switch input and B0 as the output to the LED.

There are two ways to count, UP and DOWN. We usually count up and know automatically when we have reached 10. A computer, however, knows when it reaches a count of 10 by subtracting the count from 10. If the answer is zero, then bingo. A simpler way, however, is to start at 10 and count down to zero—after 10 events we will have reached zero without doing a subtraction. Zero for the microcontroller is a useful number.

The initial flowchart for this problem is shown in Fig. 13.1. To ensure that the LED is OFF after the switch is pressed for the eleventh time, put in TURN OFF LED after the switch is pressed, as shown in Fig. 13.2.

Note: The switch will bounce and the micro is fast enough to count these bounces, thinking that the switch has been pressed several times. A 0.1-second delay is inserted after each switch operation to allow time for the bounces to stop.

The final flowchart is shown in Fig. 13.2.

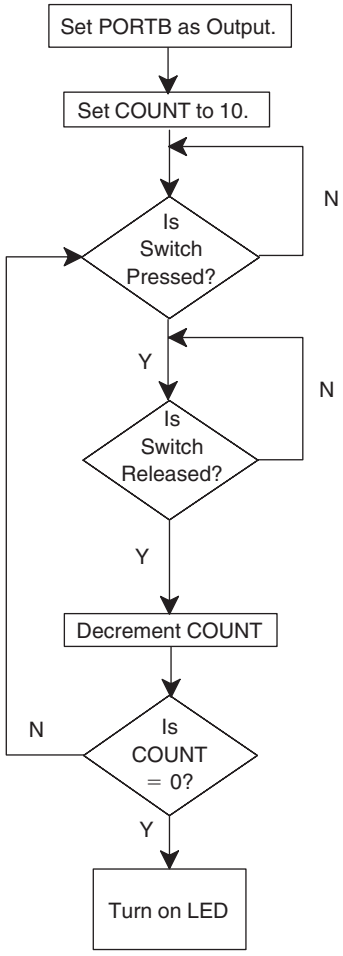


Figure 13.1: Initial Counting Flowchart

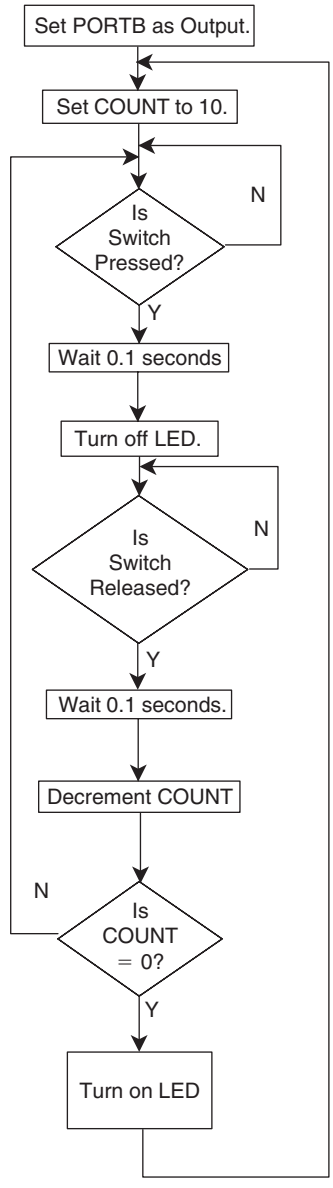


Figure 13.2: Final Counting Flowchart

13.1.1 The Program for the Counting Circuit

```
;COUNT84.ASM using the 16F84 with a 32 kHz crystal

;EQUATES SECTION

TMR0      EQU     1          ;means TMR0 is file 1.
STATUS    EQU     3          ;means STATUS is file 3.
PORTA     EQU     5          ;means PORTA is file 5.
PORTB     EQU     6          ;means PORTB is file 6.
TRISA     EQU     85H        ;TRISA (the PORTA I/O selection) is
                             ;file 85H
TRISB     EQU     86H        ;TRISB (the PORTB I/O selection) is
                             ;file 86H
OPTION_R  EQU     81H        ;the OPTION register is file 81H
ZEROBIT   EQU     2          ;means ZEROBIT is bit 2.
COUNT     EQU     0CH        ;COUNT is file 0C, a register to count
                             ;events.

;*********************************************************
LIST    P=16F84     ;we are using the 16F84.
ORG     0           ;the start address in memory is 0
GOTO    START       ;goto start!

;*********************************************************
;Configuration Bits

__CONFIG H'3FF0'    ;selects LP oscillator, WDT off, PUT on,
                    ;Code Protection disabled.

;*********************************************************

;SUBROUTINE SECTION.

;3/32 second delay.
DELAY   CLRF        TMR0                ;START TMR0.
LOOPA   MOVF        TMR0,W              ;READ TMR0 INTO W.
        SUBLW       .3                  ;TIME - 3
        BTFSS       STATUS, ZEROBIT     ;Check TIME-W=0
        GOTO        LOOPA               ;Time is not=3.
        RETLW       0                   ;Time is 3, return.

;*********************************************************

;CONFIGURATION SECTION

START    BSF         STATUS,5        ;Turns to Bank1.

         MOVLW       B'00011111'     ;5bits of PORTA are I/P
         MOVWF       TRISA
```

```
        MOVLW       B'00000000'
        MOVWF       TRISB           ;PORTB is OUTPUT

        MOVLW       B'00000111'     ;Prescaler is /256
        MOVWF       OPTION_R        ;TIMER is 1/32 secs.

        BCF         STATUS.5        ;Return to Bank0.
        CLRF        PORTA           ;Clears PortA.
        CLRF        PORTB           ;Clears PortB.

;*********************************************************
;Program starts now.
BEGIN   MOVLW       .10
        MOVWF       COUNT           ;Put 10 into COUNT.
PRESS   BTFSC       PORTA,0         ;Check switch is pressed
        GOTO        PRESS
        CALL        DELAY           ;Wait for 3/32 seconds.
        BCF         PORTB,0         ;TURN OFF LED.
RELEASE BTFSS       PORTA,0         ;Check switch is released.
        GOTO        RELEASE
        CALL        DELAY           ;WAIT for 3/32 seconds.
        DECFSZ      COUNT           ;Dec COUNT skip if 0.
        GOTO        PRESS           ;Wait for another press.
        BSF         PORTB,0         ;Turn on LED.
        GOTO        BEGIN           ;Restart
END
```

13.1.2 How Does it Work?

The file COUNT is first loaded with the count, 10, with:

```
MOVLW       .10
MOVWF       COUNT          ;Put 10 into COUNT.
```

We then wait for the switch to be pressed, by PORTA,0 going low:

```
PRESS   BTFSC      PORTA,0   ;Check switch is pressed
        GOTO       PRESS
```

Anti-bounce is next:

```
CALL        DELAY          ;Wait for 3/32 seconds.
```

Turn off the LED on B0:

```
BCF         PORTB,0
```

Wait for switch to be released

```
RELEASE  BTFSS      PORTA,0    ;Check switch is released.
         GOTO       RELEASE
```

Anti-bounce:

```
CALL    DELAY    ;Wait for 3/32 seconds.
```

Decrement the file COUNT; if zero turn on LED and return to begin. If not zero continue pressing the switch.

```
DECFSZ      COUNT        ;Dec COUNT skip if 0.
GOTO        PRESS        ;Wait for another press.
BSF         PORTB,0      ;Turn on LED.
GOTO        BEGIN        ;Restart
```

This may appear to be a lot of programming to count presses of a switch, but once saved as a subroutine, it can be reused in any other programs.

13.2 Look-Up Table

A look-up table is used to change data from one form to another, such as pounds to kilograms, 8ºC to 8ºF, inches to centimeters, etc. The explanation of the operation of a look-up table is best understood by way of an example.

13.3 7-Segment Display

Design a circuit that will count and display, on a 7-segment display, the number of times a button is pressed, up to 10. The circuit diagram for this is shown in Fig. 13.3.

The flowchart for the 7-segment display driver is shown in Fig. 13.4.

This is a basic solution that has a few omissions:

- The switch bounces when pressed.
- Clear the count at the start.
- The micro counts in binary, but we require a 7-segment decimal display. So we need to convert the binary count to drive the relevant segments on the display.
- When the switch is released it bounces.

The amended flowchart is shown in Fig. 13.5.

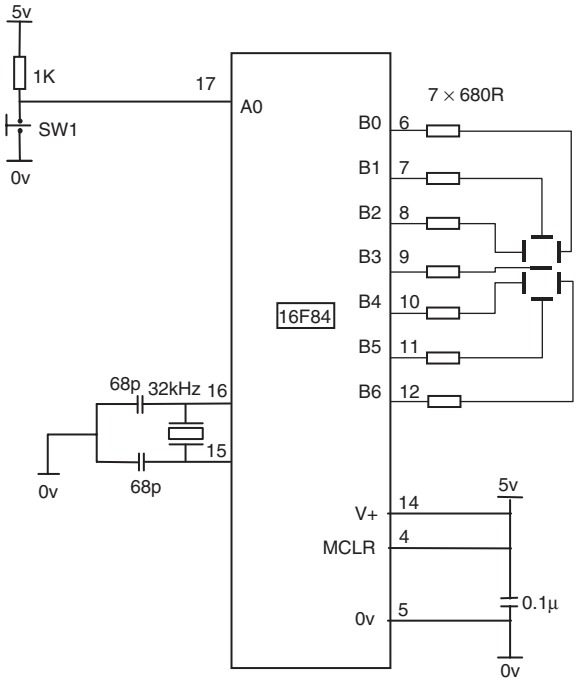


Figure 13.3: Circuit Diagram of the 7-Segment Display Driver

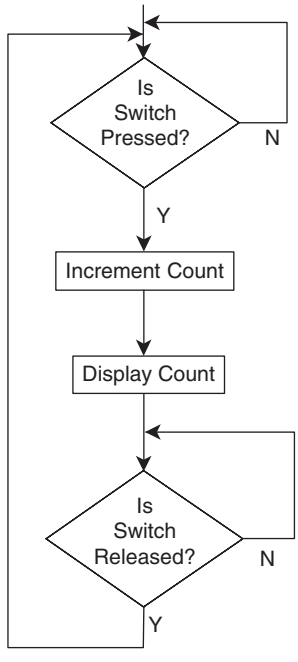


Figure 13.4: Initial Flowchart for the 7-Segment Driver

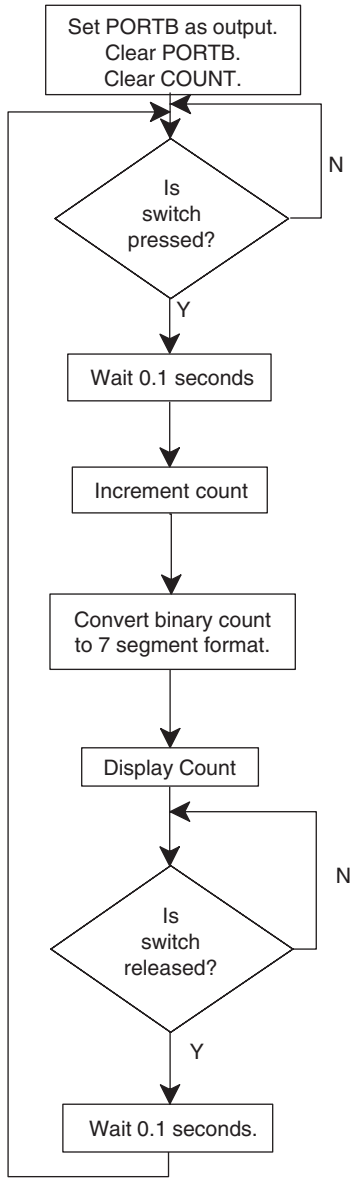


Figure 13.5: Amended Flowchart for 7-Segment Display

The flowchart is missing just one thing! What happens when the count reaches 10! The counter needs resetting (it would count up to 255 before resetting). The final flowchart is shown in Fig. 13.6.

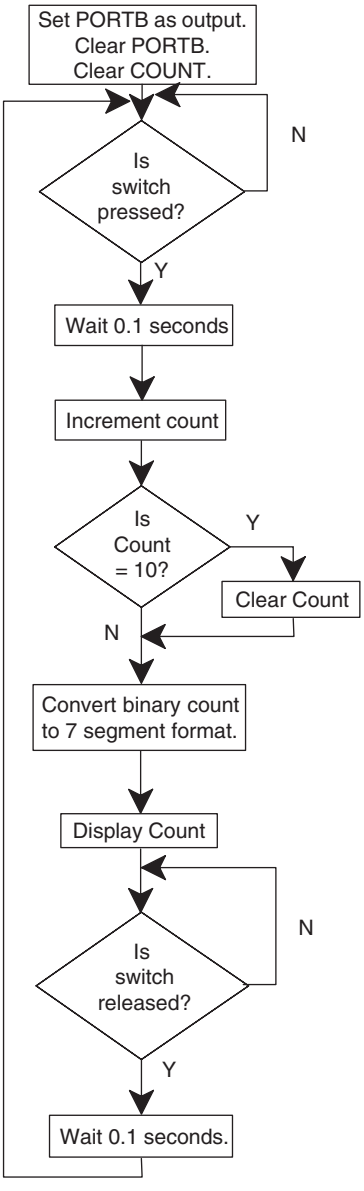


Figure 13.6: Final Flowchart for 7-Segment Display

Now about this look up table.

Table 13.1 shows the configuration of PORTB to drive the 7-segment display. (Refer also to Fig. 13.3).

Table 13.1: Binary Code to Drive 7-Segment Display

Number	PortB							
	B7	B6	B5	B4	B3	B2	B1	B0
0	0	1	1	1	0	1	1	1
1	0	1	0	0	0	0	0	1
2	0	0	1	1	1	0	1	1
3	0	1	1	0	1	0	1	1
4	0	1	0	0	1	1	0	1
5	0	1	1	0	1	1	1	0
6	0	1	1	1	1	1	0	0
7	0	1	0	0	0	0	1	1
8	0	1	1	1	1	1	1	1
9	0	1	0	0	1	1	1	1

The look-up table for this is:

```
CONVERT     ADDWF       PC
            RETLW       B'01110111'    ;0
            RETLW       B'01000001'    ;1
            RETLW       B'00111011'    ;2
            RETLW       B'01101011'    ;3
            RETLW       B'01001101'    ;4
            RETLW       B'01101110'    ;5
            RETLW       B'01111100'    ;6
            RETLW       B'01000011'    ;7
            RETLW       B'01111111'    ;8
            RETLW       B'01001111'    ;9
```

13.3.1 How Does the Look-Up Table Work?

Suppose we need to display a 0. We move 0 into W and CALL the look-up table; here it is called CONVERT. The first line says ADD W to the Program Count. Since W = 0 then go to the next line of the program, which will return with the 7-segment value 0.

Suppose we need to display a 6. Move 6 into W and CALL CONVERT. The first line says ADD W to the Program Count. Since W contains 6, then go to the next line of the program and move down 6 more lines and return with the code for 6, etc.

Just one more thing: To check that a count has reached 10, subtract 10 from the count if the answer is 0, bingo!

The program listing for the complete program is:

```
;DISPLAY.ASM

;EQUATES SECTION


PC         EQU    2       ;means PC is file 2.
TMR0       EQU    1       ;means TMR0 is file 1.
STATUS     EQU    3       ;means STATUS is file 3.
PORTA      EQU    5       ;means PORTA is file 5.
PORTB      EQU    6       ;means PORTB is file 6.
TRISA      EQU    85H     ;TRISA (the PORTA I/O selection) is file 85H
TRISB      EQU    86H     ;TRISB (the PORTB I/O selection) is file 86H
OPTION_R   EQU    81H     ;the OPTION register is file 81H
ZEROBIT    EQU    2       ;means ZEROBIT is bit 2.
COUNT      EQU    0CH     ;COUNT is file 0C, a register to count events.

;*********************************************************

LIST     P=16F84      ;we are using the 16F84.
ORG      0            ;the start address in memory is 0
GOTO     START        ;goto start!

;*********************************************************
;Configuration Bits
__CONFIG H'3FF0'     ;selects LP oscillator, WDT off, PUT on,
                     ;Code Protection disabled.

;*********************************************************
;SUBROUTINE SECTION.

;3/32 second delay.
DELAY     CLRF      TMR0              ;START TMR0.
LOOPA     MOVF      TMR0,W            ;READ TMR0 INTO W.
          SUBLW     .3                ;TIME - 3
          BTFSS     STATUS, ZEROBIT   ;Check TIME-W=0
          GOTO      LOOPA             ;Time is not=3.
          RETLW     0                 ;Time is 3, return.

CONVERT   ADDWF     PC
          RETLW     B'01110111'       ;0
          RETLW     B'01000001'       ;1
          RETLW     B'00111011'       ;2
          RETLW     B'01101011'       ;3
          RETLW     B'01001101'       ;4
          RETLW     B'01101110'       ;5
          RETLW     B'01111100'       ;6
```

```
         RETLW     B'01000011'       ;7
         RETLW     B'01111111'       ;8
         RETLW     B'01001111'       ;9
;*********************************************************
;CONFIGURATION SECTION
START    BSF       STATUS,5          ;Turns to Bank1.
         MOVLW     B'00011111'       ;5bits of PORTA are I/P
         MOVWF     TRISA
         MOVLW     B'00000000'
         MOVWF     TRISB             ;PORTB is OUTPUT
         MOVLW     B'00000111'       ;Prescaler is /256
         MOVWF     OPTION_R          ;TIMER is 1/32 secs.
         BCF       STATUS,5          ;Return to Bank0.
         CLRF      PORTA             ;Clears PortA.
         CLRF      PORTB             ;Clears PortB.
;*********************************************************
;Program starts now.
         CLRF      COUNT             ;Set COUNT to 0.
PRESS    BTFSC     PORTA,0           ;Test for switch press.
         GOTO      PRESS             ;Not pressed.
         CALL      DELAY             ;Antibounce wait 0.1sec.
         INCF      COUNT             ;Add 1 to COUNT.
         MOVF      COUNT,W           ;Move COUNT to W.
         SUBLW     .10               ;COUNT-10, W is altered.
         BTFSC     STATUS,ZEROBIT    ;Is COUNT - 10=0?
         CLRF      COUNT             ;Count=10 Make Count=0
         MOVF      COUNT,W           ;Put Count in W again.
         CALL      CONVERT           ;Count is not 10, carry on.
         MOVWF     PORTB             ;Output number to display.
RELEASE  BTFSS     PORTA,0           ;Is switch released?
         GOTO      RELEASE           ;Not released.
         CALL      DELAY             ;Antibounce wait 0.1sec.
         GOTO      PRESS             ;Look for another press.
END
```

13.3.2 How Does the Program Work?

The file count is cleared (to zero) and we wait for the switch to be pressed.

```
         CLRF      COUNT         ;Set COUNT to 0.
PRESS    BTFSC     PORTA,0       ;Test for switch press.
         GOTO      PRESS         ;Not pressed.
```

Wait for 0.1 seconds, handle anti-bounce.

```
CALL              DELAY
```

Add 1 to COUNT and check to see if it equals 10:

```
INCF     COUNT              ;Add 1 to COUNT.
MOVF     COUNT,W            ;Move COUNT to W.
SUBLW    .10                ;COUNT-10, W is altered.
BTFSC    STATUS,ZEROBIT     ;Is COUNT - 10=0?
```

If COUNT is 10, clear it to 0 and output the count as 0. If the COUNT is not 10 then output the count.

```
CLRF     COUNT         ;Count = 10 Make Count=0
MOVF     COUNT,W       ;Put Count in W again.
CALL     CONVERT       ;Count is not 10, carry on.
MOVWF    PORTB         ;Output number to display.
```

Wait for the switch to be released and debounce. Then return to monitor the presses.

```
RELEASE    BTFSS          PORTA,0      ;Is switch released?
           GOTO           RELEASE      ;Not released.
           CALL           DELAY        ;Antibounce wait 0.1sec.
           GOTO           PRESS        ;Look for another press.
```

13.3.3 Test Your Understanding

Modify the program to count up to 6 and reset.

Modify the program to count up to F in HEX and reset.

A look-up table to change ºC to ºF is shown below, called DEGREE:

```
DEGREE    ADDWF     PC          ;ADD W to Program Count.
          RETLW     .32         ;0°C=32°F
          RETLW     .34         ;1°C=34°F
          RETLW     .36         ;2°C=36°F
          RETLW     .37         ;3°C=37°F
          RETLW     .39         ;4°C=39°F
          RETLW     .41         ;5°C=41°F
          RETLW     .43         ;6°C=43°F
          RETLW     .45         ;7°C=45°F
          RETLW     .46         ;8°C=46°F
          RETLW     .48         ;9°C=48°F
          RETLW     .50         ;10°C=50°F
          RETLW     .52         ;11°C=52°F
          RETLW     .54         ;12°C=54°F
```

```
        RETLW     .55         ;13°C=55°F
        RETLW     .57         ;14°C=57°F
        RETLW     .59         ;15°C=59°F
        RETLW     .61         ;16°C=61°F
        RETLW     .63         ;17°C=63°F
        RETLW     .64         ;18°C=64°F
        RETLW     .66         ;19°C=66°F
        RETLW     .68         ;20°C=68°F
        RETLW     .70         ;21°C=70°F
        RETLW     .72         ;22°C=72°F
        RETLW     .73         ;23°C=73°F
        RETLW     .75         ;24°C=75°F
        RETLW     .77         ;25°C=77°F
        RETLW     .79         ;26°C=79°F
        RETLW     .81         ;27°C=81°F
        RETLW     .82         ;28°C=82°F
        RETLW     .84         ;29°C=84°F
        RETLW     .86         ;30°C=86°F
```

Another application of the use of the look-up table is a solution for a previous example. the "Control Application – A Hot Air Blower" introduced in Chapter 11. In this example when PORTA was read, the data was treated as a binary number, but we could just as easily treat the data as a decimal number.

```
i.e., A2 A1 A0=000 or 0
              =001 or 1
              =010 or 2
              =011 or 3
              =100 or 4
              =101 or 5
              =110 or 6
              =111 or 7
```

The look-up table for this would be:

```
CONVERT     ADDWF      PC
            RETLW      B'00000010'   ;0 on PORTA turns on B1
            RETLW      B'00000001'   ;1 on PORTA turns on B0
            RETLW      B'00000011'   ;2 on PORTA turns on B1,B0
            RETLW      B'00000001'   ;3 on PORTA turns on B0
            RETLW      B'00000000'   ;4 on PORTA turns off B1,B0
            RETLW      B'00000001'   ;5 on PORTA turns on B0
            RETLW      B'00000000'   ;6 on PORTA turns off B1,B0
            RETLW      B'00000010'   ;7 on PORTA turns on B1
```

The complete program listing for the program DISPLAY2 would be:

```
;DISPLAY2.ASM

;EQUATES SECTION

PC          EQU      2          ;Program Counter is file 2.
TMR0        EQU      1          ;means TMR0 is file 1.
STATUS      EQU      3          ;means STATUS is file 3.
PORTA       EQU      5          ;means PORTA is file 5.
PORTB       EQU      6          ;means PORTB is file 6.
TRISA       EQU      85H        ;TRISA (the PORTA I/O selection) is
                                ;file 85H
TRISB       EQU      86H        ;TRISB (the PORTB I/O selection) is
                                ;file 86H
OPTION_R    EQU      81H        ;the OPTION register is file 81H
ZEROBIT     EQU      2          ;means ZEROBIT is bit 2.
COUNT       EQU      0CH        ;COUNT is file 0C, a register to count
                                ;events.
;*********************************************************
LIST     P=16F84     ;we are using the 16F84.
ORG      0           ;the start address in memory is 0
GOTO     START       ;goto start!

;*********************************************************
;Configuration Bits

__CONFIG H'3FF0'     ;selects LP oscillator, WDT off, PUT on,
                     ;Code Protection disabled.

;*****************************************************
;SUBROUTINE SECTION.

CONVERT     ADDWF       PC
            RETLW       B'00000010'    ;0 on PORTA turns on B1
            RETLW       B'00000001'    ;1 on PORTA turns on B0
            RETLW       B'00000011'    ;2 on PORTA turns on B1,B0
            RETLW       B'00000001'    ;3 on PORTA turns on B0
            RETLW       B'00000000'    ;4 on PORTA turns off B1,B0
            RETLW       B'00000001'    ;5 on PORTA turns on B0
            RETLW       B'00000000'    ;6 on PORTA turns off B1,B0
            RETLW       B'00000010'    ;7 on PORTA turns on B1

;*****************************************************

;CONFIGURATION SECTION

START    BSF        STATUS,5        ;Turns to Bank1.

         MOVLW      B'00011111'     ;5bits of PORTA are I/P
         MOVWF      TRISA
```

```
        MOVLW     B'00000000'
        MOVWF     TRISB           ;PORTB is OUTPUT

        MOVLW     B'00000111'     ;Prescaler is /256
        MOVWF     OPTION_R        ;TIMER is 1/32 secs.

        BCF       STATUS,5        ;Return to Bank0.
        CLRF      PORTA           ;Clears PortA.
        CLRF      PORTB           ;Clears PortB.

;*********************************************************
;Program starts now.
BEGIN   MOVF      PORTA,W     ;Read PORTA into W
        CALL      CONVERT     ;Obtain O/Ps from I/Ps.
        MOVWF     PORTB       ;switch on O/Ps
        GOTO       BEGIN      ;repeat
END
```

13.3.4 How Does the Program Work?

The program first of all reads the value of PORTA into the working register, W:

```
MOVF   PORTA,W
```

The CONVERT routine is called, which returns with the correct setting of the outputs in W. If the value of PORTA was 3, then the look-up table would return with 00000001 in W to turn on B0 and turn off B1:

```
CALL    CONVERT  ;Obtain O/Ps from I/Ps.
MOVWF   PORTB    ;switch on O/Ps
```

The program then returns to check the setting of PORTA again.

13.4 Numbers Larger than 255

The PIC microcontrollers are 8-bit devices. This means that they can easily count up to 255 using one memory location, but to count higher, then more than one memory location has to be used for the count. Consider counting a switch press up to 1000 and then turn on an LED to show this count has been achieved. The circuit for this is shown in Fig. 13.7.

To count up to 1000 in decimal, 03E8 in hex, files COUNTB and COUNTA will store the count (a count of 65535 is then possible). COUNTB will count up to 03H. Then when COUNTA has reached E8H, LED1 will light indicating the count of 1000 has been reached.

The flowchart for this 1000 count is shown in Fig. 13.8.

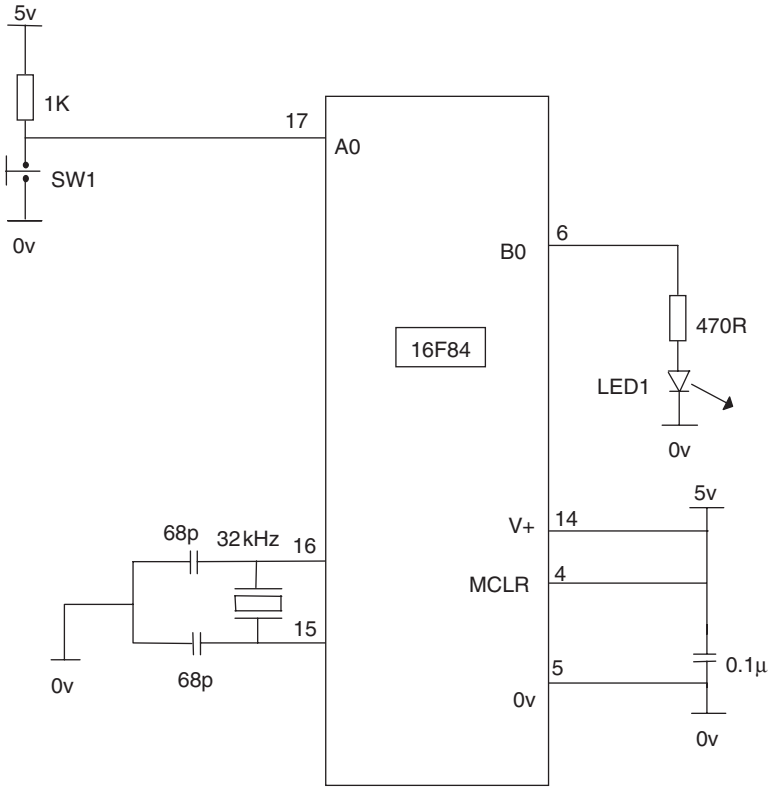


Figure 13.7: Circuit for 1000 Count

13.4.1 Flowchart Explanation

The program is waiting for SW1 to be pressed. When it is, there is a delay of 0.1 seconds to allow the switch bounce to stop. The program then looks for the switch to be released and waits 0.1 seconds for the bounce to stop. Then 1 is added to COUNTA and a check is made to see if the count has overflowed—i.e., reached 256. (Note that 255 is the maximum it will hold; when it reaches 256 it will reset to zero just like a two-digit counter would reset to zero going from 99 to 100.) If COUNTA has overflowed then we increment COUNTB. A check is made to see if COUNTB has reached 03H; if not we return to keep counting. If COUNTB has reached 03H then we count presses until COUNTA reaches E8H. The count in decimal is then 1000 and the LED is lit.

Any count can be attained by altering the values that COUNTB and COUNTA are allowed to count up to. To count up to 5000 in decimal, which is 1388H, ask if COUNTB = 13H, and then count until COUNTA has reached 88H.

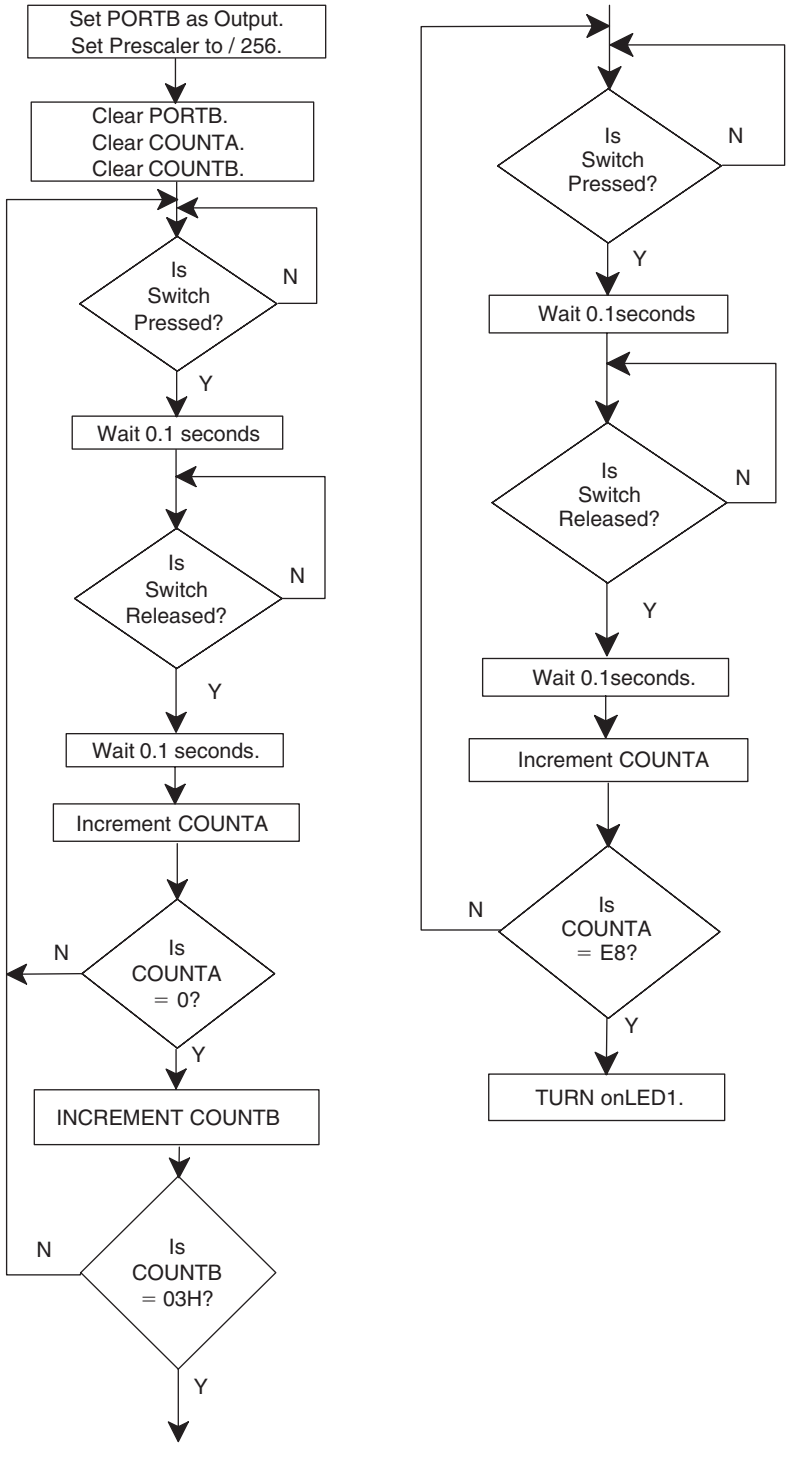


Figure 13.8: Count of 1000 Flowchart

13.4.2 The Program Listing

```
;CNT1000.ASM

;EQUATES SECTION

TMR0       EQU   1      ;means TMR0 is file 1.
STATUS     EQU   3      ;means STATUS is file 3.
PORTA      EQU   5      ;means PORTA is file 5.
PORTB      EQU   6      ;means PORTB is file 6.
TRISA      EQU   85H    ;TRISA (the PORTA
                        ;I/O selection) is file 85H
TRISB      EQU   86H    ;TRISB (the PORTB I/O selection) is file 86H
OPTION_R   EQU   81H    ;the OPTION register is file 81H
ZEROBIT    EQU   2      ;means ZEROBIT is bit 2.
COUNTA     EQU   0CH    ;USER RAM LOCATION.
COUNTB     EQU   0DH    ;USER RAM LOCATION.

;*********************************************************
LIST     P=16F84    ;we are using the 16F84.
ORG      0          ;the start address in memory is 0
GOTO     START      ;goto start!

;*********************************************************
;Configuration Bits

__CONFIG H'3FF0'   ;selects LP oscillator, WDT off, PUT on,
                   ;Code Protection disabled.
;*****************************************************
;SUBROUTINE SECTION.

;3/32 second delay.
DELAY    CLRF      TMR0             ;START TMR0.
LOOPA    MOVF      TMR0,W           ;READ TMR0 INTO W.
         SUBLW     .3               ;TIME - 3
         BTFSS     STATUS,ZEROBIT   ;Check TIME-W=0
         GOTO      LOOPA            ;Time is not=3.
         RETLW     0                ;Time is 3, return.

;*********************************************************

;CONFIGURATION SECTION

START    BSF       STATUS,5         ;Turns to Bank1.

         MOVLW     B'00011111'      ;5bits of PORTA are I/P
         MOVWF     TRISA
```

```
        MOVLW     B'00000000'
        MOVWF     TRISB             ;PORTB is OUTPUT

        MOVLW     B'00000111'       ;Prescaler is /256
        MOVWF     OPTION_R          ;TIMER is 1/32 secs.

        BCF       STATUS,5          ;Return to Bank0.
        CLRF      PORTA             ;Clears PortA.
        CLRF      PORTB             ;Clears PortB.

;*********************************************************
;Program starts now.

        CLRF      COUNTA
        CLRF      COUNTB

PRESS   BTFSC     PORTA,0           ;Check switch pressed
        GOTO      PRESS
        CALL      DELAY             ;Wait for 3/32 seconds.
RELEASE BTFSS     PORTA,0           ;Check switch is released.
        GOTO      RELEASE
        CALL      DELAY             ;Wait for 3/32 seconds.
        INCFSZ    COUNTA            ;Inc. COUNT skip if 0.
        GOTO      PRESS
        INCF      COUNTB
        MOVLW     03H               ;Put 03H in W.            *
        SUBWF     COUNTB,W          ;COUNTB -W (i.e. 03)
        BTFSS     STATUS,ZEROBIT    ;IS COUNTB=03H
        GOTO      PRESS             ;No

PRESS1  BTFSC     PORTA,0           ;Check switch pressed.
        GOTO      PRESS1
        CALL      DELAY             ;Wait for 3/32 seconds.
RELEASE1BTFSS     PORTA,0           ;Check switch released.
        GOTO      RELEASE1
        CALL      DELAY             ;Wait for 3/32 seconds.
        INCF      COUNTA
        MOVLW     0E8H              ;Put E8 in W.              *
        SUBWF     COUNTA            ;COUNTA - E8.
        BTFSS     STATUS,ZEROBIT    ;COUNTA=E8?
        GOTO      PRESS1            ;No.
        BSF       PORTB,0           ;Yes, turn on LED1.
STOP    GOTO      STOP              ;stop here

END
```

13.4.3 How Does the Program Work?

The two files used for counting are cleared.

```
CLRF    COUNTA
CLRF    COUNTB
```

As we have done previously, we wait for the switch to be pressed and released and to stop bouncing:

```
PRESS     BTFSC      PORTA,0     ;Check switch pressed
          GOTO       PRESS
          CALL       DELAY       ;Wait for 3/32 seconds.

RELEASE   BTFSS      PORTA,0     ;Check switch is released.
          GOTO       RELEASE
          CALL       DELAY       ;Wait for 3/32 seconds.
```

We add 1 to file COUNTA and check to see if it is zero. If it isn't then continue monitoring presses. (The file would be zero when we add 1 to the 8 bit number 1111 1111; it overflows to 0000 0000.)

```
INCFSZ    COUNTA      ;Inc. COUNT skip if 0.
GOTO      PRESS
```

If the file COUNTA has overflowed, then we add 1 to the file COUNTB, just like you would do with two columns of numbers. We then need to know if COUNTB has reached 03H. If COUNTB is not 03H, then we return to PRESS and continue monitoring the presses.

```
INCF      COUNTB
MOVLW     03H                 ;Put 03H in W.
SUBWF     COUNTB,W            ;COUNTB - W (i.e. 03)
BTFSS     STATUS,ZEROBIT      ;IS COUNTB=03H?
GOTO      PRESS               ;No
```

Once COUNTB has reached 03H we need only wait until COUNTA reaches 0E8H and we would have counted up to 03E8H, 5000 in decimal. Then we turn on the LED.

```
PRESS1    BTFSC    PORTA,0          ;Check switch pressed.
          GOTO     PRESS1
          CALL     DELAY            ;Wait for 3/32 seconds.
RELEASE1  BTFSS    PORTA,0          ;Check switch released.
          GOTO     RELEASE1
          CALL     DELAY            ;Wait for 3/32 seconds.
          INCF     COUNTA
          MOVLW    0E8H             ;Put E8 in W.
          SUBWF    COUNTA           ;COUNTA – E8.
          BTFSS    STATUS,ZEROBIT   ;COUNTA=E8?
```

```
          GOTO     PRESS1          ;No.
          BSF      PORTB,0         ;Yes, turn on LED1.
STOP      GOTO     STOP            ;stop here
```

This listing can be used as a subroutine in your program to count up to any number up to 65535 (or more if you use a COUNTC file). Just alter COUNTB and COUNTA values to whatever values you wish, in the two places marked * in the program.

Question: How would you count up to 20,000?

Answer: (Have you tried it first?). 20,000 = 4E20H so COUNTB would count up to 4EH and COUNTA would then count to 20H.

Question: How would you count to 100,000?

Answer: 100,000 = 0186A0H, so you would use a third file COUNTC to count to 01H, COUNTB would count to 86H and COUNTA would count to A0H.

Programming can be made a lot simpler by keeping a library of subroutines. Here is another....

13.5 Long Time Intervals

A more frequent use of a large count is to count TMR0 pulses to generate long time intervals. We have previously seen in the section on delay that we can slow the internal timer clock down to 1/32 seconds. Counting a maximum of 255 of these gives a time of 255 1/32 = 8 seconds.

Suppose we want to turn on an LED for 5 minutes when a switch is pressed:

5 minutes = 300 seconds = 300 × 32 (1/32 seconds), a TMR0 count of 9600. This is 2580 in hex. The circuit is the same as Fig. 13.7 for the 1000-count circuit, and the flowchart is shown in Fig. 13.9.

13.5.1 Explanation of the Flowchart

1. Wait until the switch is pressed; the LED is then turned on.
2. TMR0 is cleared to start the timing interval.
3. TMR0 is moved into W (read) to catch the first count.
4. Then wait for TMR0 to return to zero (the count will be 256), 100 in hex.
5. COUNTA is then incremented and steps 3 and 4 repeated until COUNTA reaches 25H.
6. Wait until TMR0 has reached 80H.
7. The count has reached 2580H, 9600 in decimal. 5 minutes have elapsed and the LED is turned off.

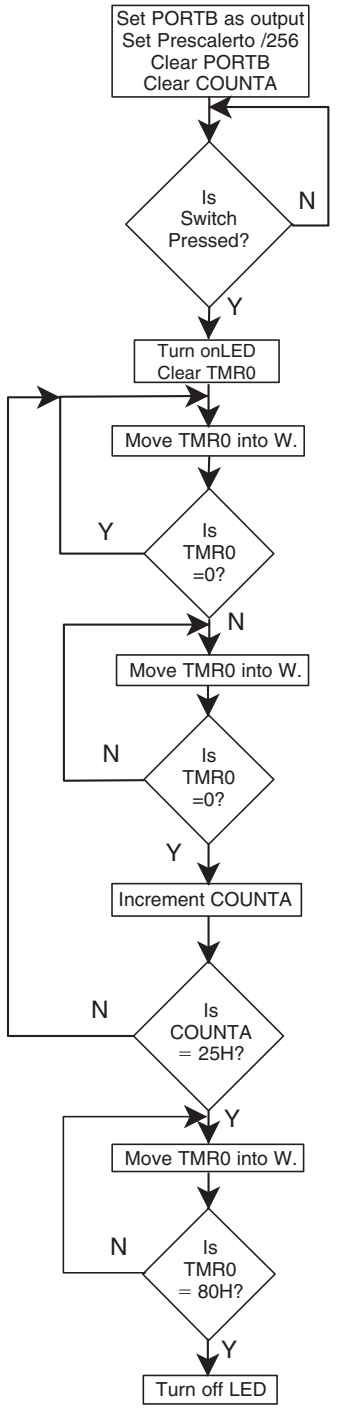


Figure 13.9: Flowchart for the 5 Minute Delay

13.5.2 Program Listing for 5 Minute Delay

```
;LONGDLY.ASM

;EQUATES SECTION

TMR0      EQU   1      ;means TMR0 is file 1.
STATUS    EQU   3      ;means STATUS is file 3.
PORTA     EQU   5      ;means PORTA is file 5.
PORTB     EQU   6      ;means PORTB is file 6.
TRISA     EQU   85H    ;TRISA (the PORTA I/O selection) is file 85H
TRISB     EQU   86H    ;TRISB (the PORTB I/O selection) is file 86H
OPTION_R  EQU   81H    ;the OPTION register is file 81H
ZEROBIT   EQU   2      ;means ZEROBIT is bit 2.
COUNTA    EQU   0CH    ;COUNT is file 0C, a register to count events.

;*********************************************************
LIST    P=16F84    ;we are using the 16F84.
ORG     0          ;the start address in memory is 0
GOTO    START      ;goto start!

;*********************************************************
;Configuration Bits
__CONFIG H'3FF0'   ;selects LP oscillator, WDT off, PUT on,
                   ;Code Protection disabled.

;*********************************************************
;CONFIGURATION SECTION

START   BSF       STATUS,5      ;Turns to Bank1.
        MOVLW     B'00011111'   ;5bits of PORTA are I/P
        MOVWF     TRISA

        MOVLW     B'00000000'
        MOVWF     TRISB         ;PORTB is OUTPUT

        MOVLW     B'00000111'   ;Prescaler is /256
        MOVWF     OPTION_R      ;TIMER is 1/32 secs.

        BCF       STATUS,5      ;Return to Bank0.
        CLRF      PORTA         ;Clears PortA.
        CLRF      PORTB         ;Clears PortB.
;*********************************************************
;Program starts now.

        CLRF      COUNTA
PRESS   BTFSC     PORTA,0           ;Check switch pressed.
        GOTO      PRESS             ;No
```

```
        BSF     PORTB,0          ;Yes, turn on LED
        CLRF    TMR0             ;Start TMR0.
WAIT0   MOVF    TMR0,W           ;Move TMR0 into W
        BTFSC   STATUS,ZEROBIT   ;Is TMR0=0.
        GOTO    WAIT0            ;Yes
WAIT1   MOVF    TMR0,W           ;No, move TMR0 into W.
        BTFSS   STATUS,ZEROBIT
        GOTO    WAIT1            ;Wait for TMR0 to overflow
        INCF    COUNTA           ;Increment COUNTA
        MOVLW   25H
        SUBWF   COUNTA,W         ;COUNTA - 25H
        BTFSS   STATUS,ZEROBIT   ;Is COUNTA=25H
        GOTO    WAIT0            ;COUNTA<25H
WAIT2   MOVF    TMR0,W           ;COUNTA=25H
        MOVLW   80H
        SUBWF   TMR0,W           ;TMR0 - 80H
        BTFSS   STATUS,ZEROBIT   ;Is TMR0=80H
        GOTO    WAIT2            ;TMR0<80H
        BCF     PORTB,0          ;TMR0=80H, turn off LED
END
```

The explanation of this program operation is similar to that of the count to 1000, done earlier in this chapter. This listing can be used as a subroutine and times up to 65535 × 1/32 seconds (34 minutes) can be obtained.

13.6 One Hour Delay

Another and probably a simpler way of obtaining a delay of 1 hour is:

- Write a delay of 5 seconds.
- CALL it 6 times, giving a delay of 30 seconds.
- Put this in a loop to repeat 120 times; 120 × 30 seconds = 1 hour.

The code for the 1-hour subroutine will look like this:

```
ONEHOUR     MOVLW    .120      ;put 120 in W
            MOVWF    COUNT     ;load COUNT with 120
LOOP        CALL     DELAY5    ;Wait 5 seconds
            CALL     DELAY5    ;Wait 5 seconds
            CALL     DELAY5    ;Wait 5 seconds
            CALL     DELAY5    ;Wait 5 seconds
            CALL     DELAY5    ;Wait 5 seconds
            CALL     DELAY5    ;Wait 5 seconds
            DECFSZ   COUNT     ;Subtract 1 from COUNT
```

```
            GOTO      LOOP      ;Count is not zero.
            RETLW     0         ;RETURN to program.
```

13.6.1 The Program for the One-Hour Delay

```
;ONEHOUR.ASM for 16F84. This sets PORTA as an INPUT (NB 1
;                       means input) and PORTB as an OUTPUT
;                       (NB 0 means output). The OPTION
;                       register is set to /256 to give timing pulses
;                       of 1/32 of a second.
;                       1hour and 5 second delays are
;                       included in the subroutine section.

;*********************************************************
;EQUATES SECTION

TMR0      EQU   1       ;means TMR0 is file 1.
STATUS    EQU   3       ;means STATUS is file 3.
PORTA     EQU   5       ;means PORTA is file 5.
PORTB     EQU   6       ;means PORTB is file 6.
TRISA     EQU   85H     ;TRISA (the PORTA I/O selection) is file 85H
TRISB     EQU   86H     ;TRISB (the PORTB I/O selection) is file 86H
OPTION_R  EQU   81H     ;the OPTION register is file 81H
ZEROBIT   EQU   2       ;means ZEROBIT is bit 2.
COUNT     EQU   0CH     ;COUNT is file 0C, a register to count events.

;*********************************************************
LIST    P=16F84     ;we are using the 16F84.
ORG     0           ;the start address in memory is 0
GOTO    START       ;goto start!

;*********************************************************
;Configuration Bits
__CONFIG H'3FF0'    ;selects LP oscillator, WDT off, PUT on,
                    ;Code Protection disabled.

;*********************************************************
;SUBROUTINE SECTION.

;1 hour delay.
ONEHOUR   MOVLW     .120            ;put 120 in W

          MOVWF     COUNT           ;load COUNT with 120
LOOP      CALL      DELAY5          ;Wait 5 seconds
          CALL      DELAY5          ;Wait 5 seconds
```

```
          CALL      DELAY5          ;Wait 5 seconds
          CALL      DELAY5          ;Wait 5 seconds
          CALL      DELAY5          ;Wait 5 seconds
          CALL      DELAY5          ;Wait 5 seconds
          DECFSZ    COUNT           ;Subtract 1 from COUNT
          GOTO      LOOP            ;Count is not zero.
          RETLW     0               ;RETURN to program.

;5 second delay.
DELAY5    CLRF      TMR0            ;START TMR0.
LOOPB     MOVF      TMR0,W          ;READ TMR0 INTO W.
          SUBLW     .160            ;TIME - 160
          BTFSS     STATUS,ZEROBIT  ;Check TIME-W=0
          GOTO      LOOPB           ;Time is not=160.
          RETLW     0               ;Time is 160, return.

;*********************************************************
;CONFIGURATION SECTION

START     BSF       STATUS,5        ;Turns to Bank1.

          MOVLW     B'00011111'     ;5bits of PORTA are I/P
          MOVWF     TRISA

          MOVLW     B'00000000'
          MOVWF     TRISB           ;PORTB is OUTPUT

          MOVLW     B'00000111'     ;Prescaler is /256
          MOVWF     OPTION_R        ;TIMER is 1/32 secs.
          BCF       STATUS,5        ;Return to Bank0.
          CLRF      PORTA           ;Clears PortA.
          CLRF      PORTB           ;Clears PortB.

;*********************************************************
;Program starts now.
          BSF       PORTB,0         ;Turn on B0
          CALL      ONEHOUR         ;Wait 1 Hour.
          BCF       PORTB,0         ;Turn off B0.
STOP      GOTO      STOP            ;STOP!

END
```

SECTION III

Programming PIC Microcontrollers Using PicBasic

This page intentionally left blank

CHAPTER 14

PicBasic and PicBasic Pro Programming

BASIC is one of the oldest and one of the easiest programming languages to learn. You should be able to learn and program in BASIC in less than an hour. In this chapter, we shall be looking at the principles of programming PIC microcontrollers using the PicBasic and PicBasic Pro languages. Both these languages are very similar to the standard BASIC language but they have some modified and some additional instructions specifically for microcontroller programming.

Both PicBasic and PicBasic Pro languages have been developed by the microEngineering Labs Inc. PicBasic is a lower-cost, simpler language than PicBasic Pro and it is aimed at students and hobbyists. PicBasic Pro is more expensive, aimed at professionals, and includes additional commands for more advanced instructions.

Table 14.1 gives a list of the comparison of PicBasic and PicBasic Pro languages. Before we proceed to the chapter on PIC applications and projects, we shall be looking at how we can program the PIC microcontrollers using these languages.

14.1 PicBasic Language

In this section, we shall be looking at the variable types and the commands of the PicBasic language. A detailed description of all the commands can be found in the PicBasic Compiler manual, available from the web site www.melabs.com, or a printed copy can be obtained from the microEngineering Labs Inc.

14.1.1 PicBasic Variables

Variables are used to store temporary data in a program. These variables are stored in the general-purpose area of the RAM memory of a microcontroller.

Variables in PicBasic can be bytes (8 bits), or words (16 bits). Byte variables are named B0, B1, B3, etc., and word variables are named W0, W1, W2, etc. Word variables are made up of two bytes. For example, W0 uses the same memory space as bytes B0 and B1. Similarly, W1 word variable is made up of bytes B2 and B3, and so on. We can access the bit positions of variables B0 and B1 using predefined names Bit0, Bit1,…,Bit15. For example, the least

Table 14.1: Comparison of PicBasic and PicBasic Pro

PicBasic	PicBasic Pro
Low-cost ($99.95)	Higher cost ($249.95)
Limited to first 2 K of program space	No program space limit
Interrupt service routine in assembly language	Interrupt service routine can be in assembly language or in PicBasic Pro
Peek and Poke used to access registers	Registers can be accessed directly by specifying their names
Some commands can be used only for PORTB, PORTC, or GPIO	Commands can be used for all ports
Clock speed 4 MHz	Any clock speed up to 40 MHz
Most 14-bit Pic microcontrollers supported	All PIC microcontrollers, including 12-bit ones are supported
More code space in memory	5–10% less code space in memory
More difficult to learn and less powerful	Easier to learn and more powerful
No LCD commands	Special LCD control commands (LCDOUT, LCDIN)
No hardware serial communication commands	Special hardware serial communications commands (HSERIN, HSEROUT)
No PWM commands	Special PWM commands for the microcontrollers that have built-in PWM circuit (HPWM)
No Select-Case command	Select-Case command for multi-way selection
No program memory read–write commands	Commands to read and write program memory locations (READCODE, WRITECODE)
No One-wire device interface	One-wire device interface commands (OWIN, OWOUT)
No USB commands	USB commands for microcontrollers that have built-in USB circuits (USBIN, USBOUT)
No X-10 remote control commands	X-10 remote control commands (XIN, XOUT)
No A/D commands	A/D commands for microcontrollers that have built-in A/D converters (ADCIN)

significant bit of B0 is labelled Bit0, the second bit Bit1, and the most significant bit as Bit7. Similarly, the least significant bit of B1 can be named as Bit8, and the most significant bit of B1 as Bit15.

Variables are stored in the RAM memory of a PIC microcontroller where B0 is the first RAM location, B1 is the second RAM location, and so on. The size of the RAM memory depends on the type of PIC microcontroller used and Table 14.2 gives a list of the variable names for various microcontrollers. For example, if we are using a PIC16F84-type microcontroller, we can define 52 variables from B0 to B51, and the highest variable name must not exceed B51. Note that you can only access RAM locations up to the available RAM. For example, if you

Table 14.2: PicBasic Variable Names

Microcontroller	Variables (Bytes)	Variables (Words)
PIC16C61	B0–B21	W0–W10
PIC16C71	B0–B21	W0–W10
PIC16C710	B0–B21	W0–W10
PIC16F83	B0–B21	W0–W10
PIC16C84	B0–B21	W0–W10
PIC16F83	B0–B21	W0–W10
PIC12F629	B0–B47	W0–W23
PIC12F675	B0–B47	W0–W23
PIC16F630	B0–B47	W0–W23
PIC16F676	B0–B47	W0–W23
PIC16C711	B0–B51	W0–W25
PIC16F84	B0–B51	W0–W25
PIC16C554	B0–B63	W0–W31
PIC16C556	B0–B63	W0–W31
PIC16C620	B0–B63	W0–W31
PIC16C621	B0–B63	W0–W31
PIC 12C67X	B0–B79	W0–W39
PIC14C000	B0–B79	W0–W39
PIC16C558	B0–B79	W0–W39
PIC16C558	B0–B79	W0–W39
PIC16C622	B0–B79	W0–W39
PIC16C62	B0–B79	W0–W39
PIC16C63	B0–B79	W0–W39
PIC16C64	B0–B79	W0–W39
PIC16C65	B0–B79	W0–W39
PIC16C72	B0–B79	W0–W39
PIC16C73A	B0–B79	W0–W39
PIC16C74A	B0–B79	W0–W39

try to access a RAM location that does not exist, the compiler does not generate an error and your program may not work as expected.

The relationships between the byte, word, and bit variables are given in Table 14.3. For example, word W2 is made up of bytes B4 and B5. You will see additional predefined variables in Table 14.3, named Port, Dirs, and Pins. Pins refers to the PORTB hardware, Dirs refers to the port data direction register for PORTB, i.e. TRISB and a 0 sets its associated Pin

Table 14.3: Relationship Between Byte, Word, and Bit Variables

Word Variable	Byte Variable	Bit Variable
W0	B0	Bit7, Bit6,... Bit0
	B1	Bit15, Bit14,... Bit8
W1	B2	
	B3	
W2	B4	
	B5	
W3	B6	
	B7	
	...	
	...	
W39	B78	
	B79	
Port	Pins	Pin7, Pin6,... Pin0
	Dirs	Dir7, Dir6,... Dir0

to an input, and a Dirs of 1 sets its associated Pin to an output. Port is a word variable that combines Pins and Dirs. The individual pins of a port can be accessed by the variable names Pin0, Pin1,…,Pin7.

14.1.1.1 Symbols

In order to make programs more readable, we can assign meaningful names to variables, instead of using B0, B1, etc. The PicBasic statement *symbol* is used for this purpose. For example, we can assign variable name count to location B0 with the instruction:

```
Symbol count = B0
```

Symbols must be declared at the top of a program. Symbols can also be used to assign constants to names. For example, the following statement assigns the decimal value 20 to the name *total*. Note that this statement does not occupy any location in the microcontroller RAM memory. The number is simply represented with a name.

```
Symbol total = 20
```

Command names in PicBasic are case insensitive and can be written in upper case, lower case, or with a mixture of the two. Thus, all the variables below are the same:

```
TOTAL
Total
toTal
```

14.1.1.2 Comments

Comments are useful in programs to describe the operation performed in a line or in a block of lines. A comment starts with either the keyword REM or the single quote character (‘). All the characters following a comment character are ignored. Examples of comments are:

```
REM     This is a simple test program
LOW0    ‘Clear Pin 0 to 0
HIGH1   REM Set Pin 1 to 1
```

14.1.1.3 Numeric Values

In PicBasic, numeric values can be specified in three ways: decimal, binary, and hexadecimal. Decimal values are the default and require no prefix. Binary values are specified using the prefix “%” followed by the number. Hexadecimal values are specified using the prefix “$” followed by the number. Some examples are:

```
REM A has the same value in all the following three statements
A = 10
A = %00001010
A = $0A
```

14.1.1.4 ASCII Values

Character constants can be converted into their ASCII values by enclosing them in double quotes. Only one character must be specified. For example,

```
“A”    ‘ASCII value of decimal 65
“1”    ‘ASCII value of decimal 49
```

14.1.1.5 String Constants

Although PicBasic does not provide string-handling functions, we can define strings of characters by enclosing them in double quotes. For example,

```
“COMPUTER”
```

The above string is treated as a string of ASCII characters with values “C”, “O”, “M”, “P”, “U”, “T”, “E”, “R”.

14.1.1.6 Line Labels

In PicBasic programs, we often want to jump to different parts of a program, or to jump to a subroutine. A line in PicBasic is referred by a line label. A line label can be a valid identifier (a valid name in PicBasic), followed by a colon character (:). For example,

```
LOOP:
```

14.1.1.7 Multi-Statement Lines

It is possible to use more than one statement on a line to make the program more readable. A colon (:) character should be used to separate more than one statement in a line. The size of the code does not change when more than one statement is written on the same line. For example, consider the following statements:

```
B0 = 3
B1 = 5
B2 = 8
```

The above statements can all be written on the same line as

```
B0 = 3 : B1 = 5 : B2 = 8
```

14.1.2 PicBasic Mathematical and Logical Operations

PicBasic supports a number of mathematical and logical functions that make calculations easy in programs. The operations are performed on integer numbers only with 16-bit precision and there is no floating-point number format. Also, all math operations are performed strictly from left to right. The operators supported are

+	addition
−	subtraction
*	multiplication
**	most significant bit (MSB) of multiplication
/	division
//	remainder in a division
MIN	limit to minimum value
MAX	limit to maximum value
&	bitwise AND
|	bitwise OR
^	bitwise XOR
&/	bitwise AND NOT
|/	bitwise OR NOT
^/	bitwise XOR NOT

Multiplication is done on 16 × 16 bit numbers, resulting in a 32-bit result. The "*" operator returns the lower 16 bits of the 32-bit result. Similarly, the "**" operator returns the upper 16 bits of the result. For example,

```
W2 = W1 * W0        'Multiply W1 with W0. The lower 16 bits of the result
                    'are placed in W2
```

```
or,

W2 = W1 ** W0  'Multiply W1 with W0. The upper 16-bits of the result
               'are placed in W2

or,

W2 = W1 * 100  'Multiply W1 with 100. Place the lower 16-bits of the
               'result in W2. Note that this is the multiplication
               'found in most programming languages
```

Similarly with division,

```
W2 = W1 / W0   'Divide W1 by W0. The result is placed in W2

or,

W2 = W1 // W0  'Divide W1 by W0. The remainder is placed in W2
```

MIN is used to limit the result to the minimum value defined. For example,

```
B1 = B0 MIN 100
```

sets B1 to the smaller of B0 and 100, i.e. B1 cannot be greater than 100. Similarly, MAX is used to limit the result to the maximum value defined. For example,

```
B1 = B0 MAX 100
```

sets B1 to the larger of B0 and 100; i.e., B1 will be between 100 and 255.

Bitwise logical operations operate on the entire byte and these operations can be used to extract bits from bytes or to set and clear bits of a byte. For example, to extract the least significant bit of B0 we can write

```
B0 = B0 & %00000001
```

Similarly, to set bit 2 of B1 to be 1 we can write

```
B1 = B1 | %00000100
```

To store the upper four bits of B2 in B1 we can write

```
B1 = B2 & %11110000
```

14.1.3 PicBasic Program Flow Control Commands

Program flow control commands are important in every programming language since they enable the programmer to make a decision and change the flow of the program based on this decision. PicBasic language supports the following program flow control commands:

```
BRANCH
BUTTON
CALL
FOR…NEXT
GOSUB...RETURN
GOTO
IF...THEN
```

We shall now see what the functions of these commands are and how to use them in programs.

BRANCH

```
BRANCH offset, (Label0, Label1,…)
```

When this command is executed, the program will jump to the program *label* based on the value of *offset*. *Offset* is actually a program value and if *offset* is zero, the program jumps to the first label; if *offset* is one, the program jumps to the second label; and so on.

Example:

```
BRANCH B2, (Lbl1, Lbl2, Lbl3)   'If B2=0 then goto Lbl1
                                'If B2=1 then goto Lbl2
                                'If B2=2 then goto Lbl3
```

BUTTON

```
BUTTON Pin, Down, Delay, Rate, Var, Action, Label
```

This command is used to check the status of a switch. The command operates in a loop and continuously samples the pin, debouncing it and comparing the number of iteration performed with the switch closed. The parameters are

Pin Pin number (0 to 7). PORTB pins only

Down State of pin when button is pressed (0 or 1)

Delay Delay before auto-repeat begins (0 to 255). If 0, no debounce or auto-repeat is performed. If 255, only debounce, but no auto-repeat is performed

Rate Auto-repeat rate (0 to 255)

Var Byte variable used for delay/repeat countdown. Should be initialized to 0 before use

Action State of pin to perform goto (0 if not pressed, 1 if pressed)

Label Program execution continues at this label if *Action* is true

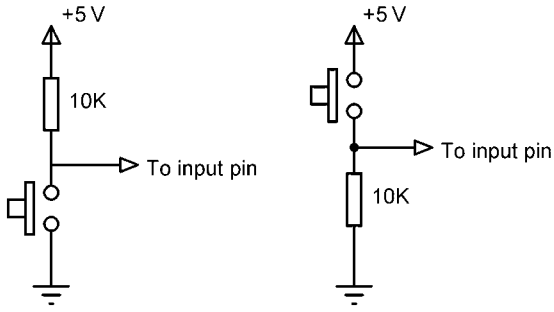


Figure 14.1: Switches That Can Be Used for the Button Command

Figure 14.1 shows the two types of switches that can be used with this command.

For example, the following command checks for a switch pressed on pin 2 (of PORTB) and jumps to Loop if it is not pressed (this command assumes that the port pin will be logic 0 when the switch is pressed; i.e., the figure on the left in Fig. 14.1):

```
BUTTON 2, 0, 255, 0, B0, 0, Loop
```

The following command checks for a switch pressed on pin 2 as above, but jumps to Loop if the switch is pressed:

```
BUTTON 2, 0, 255, 0, B0, 1, Loop
```

CALL

```
CALL Label
```

This command executes the assembly language subroutine named *Label*. For example, the command calls to assembly language routine with the name *calculate*.

```
CALL calculate
```

FOR...NEXT

```
FOR index=Start TO End (STEP (-) Inc)
      (body)
NEXT index
```

This command is used to perform iterations in a program. *Index* is a program variable which holds the initial value of the iteration count *Start*. *End* is the final value of the iteration count. *STEP* is the value by which the *index* is incremented at each iteration. If no *STEP* is specified,

the *index* is incremented by 1. The iteration repeats until *index End* and then execution continues with the next instruction following the *NEXT*. *Index* can be a byte (0 to 255), or a word (0 to 65535).

In the following example, the two statements enclosed within the FOR…NEXT are executed 10 times.

```
FOR  B0 = 1 TO 10
     B1 = B1 + 1
     B2 = B2 + 1
NEXT B0
```

or in the following example, the *index* is incremented by 2 in each iteration.

```
FOR  B0 = 1 TO 100 STEP 2
     B1 = B0 + 2
NEXT B0
```

GOSUB...RETURN

```
GOSUB Label
```

This program calls a subroutine starting at *Label*. It is like a GOTO command, but here the program returns when the RETURN statement is reached, and continues with the instruction after the GOSUB. The RETURN statement has no parameters. A subroutine has the following characteristics:

- A label to identify the starting point of the subroutine
- Body of the subroutine where the required operation is performed
- RETURN statement to exit the subroutine and return to the main calling program

Subroutines can be nested in PicBasic where a subroutine can call to other subroutines. The nesting should be restricted to no more than four levels deep. In the following example, the subroutine labelled *INC* increments variable B1 by one and then returns to the main program. On return to the main program, the statement B2 = B1 is executed.

```
   B0 = 0
   B1= 1
   GOSUB INC    'Jump to subroutine INC
   B2 = B1      'Subroutine returns here
   ........
   ........

INC:            'Start of the subroutine
   B1 = B1+ 1   'Body of the subroutine
   RETURN       'End of the subroutine
```

GOTO

```
GOTO Label
```

This command causes the program execution to jump to the statement beginning at *Label*. For example,

```
   GOTO Loop
   ......
   ......
   ......
Loop:
```

IF...THEN

```
IF Comp (AND / OR Comp) THEN Label
```

This statement is used to perform comparisons (*Comp*) in a program. If the result of the comparison is true then the program jumps to the statement at Label, otherwise execution resumes with the statement following IF…THEN.

A comparison can relate to a variable, to a constant, or to other variables. All comparisons are unsigned and the following comparison operators can be used:

<	less than
<=	less than or equal
=	equal
<>	not equal
>=	greater than or equal
>	greater than

Additionally, logical operators AND and OR can be used in a comparison operation. For example,

```
     IF B0 > 10 THEN CALC      'Jump to CALC if B0 > 10
     ..............................
     ..............................
CALC:
```

Another example is given below. In this example, if B2 is greater than 40 and at the same time B3 is less than 20 then the program jumps to the statement at label EXT. Otherwise, execution continues with the statement after the IF…THEN.

```
   IF B2 > 40 AND B3 < 20 THEN EXT
   ..................
   ..................
EXT:
```

It is important to be careful that only a *Label* is used after the THEN statement.

14.1.4 Other PicBasic Commands

We shall now briefly look at the remaining PicBasic commands in alphabetical order which are useful during the program development. More details about these commands can be obtained from the PicBasic manual.

EEPROM

```
EEPROM Location, (constant, constant,…., constant)
```

This command stores constants in consecutive bytes in on-chip EEPROM memory. The command only works with the PIC microcontrollers that have EEPROM, such as the PIC16F84, PIC16F877, etc. Location is optional, and if omitted the first EEPROM location is assumed. Constants can be numeric constants or string constants. Strings are stored as consecutive bytes of ASCII values. An example is given below.

```
EEPROM 3, (5, 2, 8)          'Store 5 in location 3,
                             '2 in location 4, and 8 in
                             'location 5
```

END

```
END
```

Stops execution and enters low power mode. The command has no parameters.

HIGH

```
HIGH Pin
```

Makes the specified pin an output pin and sets it to logic 1. *Pin* only applies to PORTB pins and it can take values from 0 to 7. In the following example, bit 1 of PORTB is configured as an output pin and is set to logic 1:

```
HIGH 1
```

I2CIN

```
I2CIN Control, Address, Var, (,Var)
```

This command is used to read data from serial EEPROMs with a 2-wire I^2C interface. A list of some compatible devices is given in Table 14.4. The lower 7 bits of the *Control* byte contain a 4-bit control code, followed by the chip select or additional address information, depending on the device used. As shown in Table 14.4, the 4-bit control code for EEPROMs is "1010". The high-order bit (MSB) of the *Control* byte is a flag indicating whether the *Address* is to be sent as 8 bits or 16 bits. If the flag is low, the *Address* is sent as 8 bits, and if it is high, the *Address* is sent as 16 bits. *(,Var)* shown in the command list is used only for 1-bit information. The I^2C

data and clock lines are predefined in the PicBasic library as bit 0 of PORTA (RA0) and bit 1 of PORTA (RA1), respectively.

Table 14.4: Some I²C Compatible EEPROMs

Device	Capacity	Control	Address size
24LC01B	128 bytes	01010xxx	8 bits
24LC02B	256 bytes	01010xxx	8 bits
24LC04B	512 bytes	01010xxb	8 bits
24LC08B	1 K bytes	01010xbb	8 bits
24LC16B	2 K bytes	01010bbb	8 bits
24LC32B	4 K bytes	11010ddd	16 bits
24LC65B	8 K bytes	11010ddd	16 bits
bbb = block select bits (each block is 256 bytes) ddd = device select bits xxx = don't care			

For example, when communicating with a 24LC02B EEPROM, the required *Address* is 8 bits, the control code is "1010" and chip select or additional address information is not required and can be assumed to be 0. The required *Control* byte is then "01010000".

Figure 14.2 shows how the 24LC02B (or any other serial EEPROM) can be connected to a PIC microcontroller. In this example, a PIC16F84 is used and pin RA0 and RA1 are connected

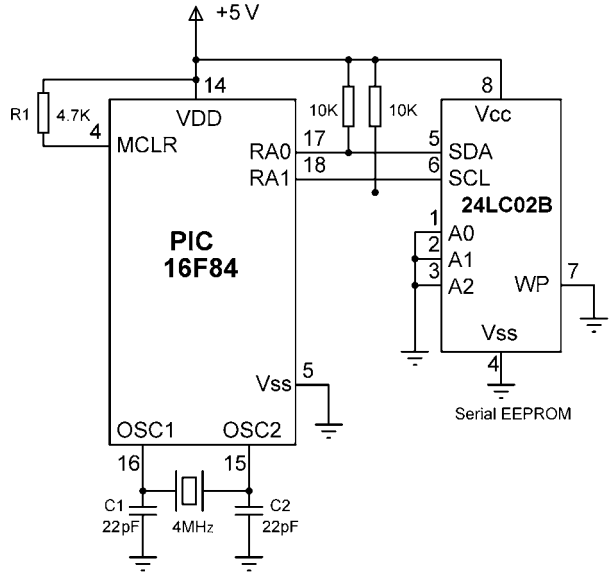


Figure 14.2: I²C Connections to a PIC Microcontroller

to the data and clock pins of the EEPROM, respectively. These are the only connections required to communicate with an I^2C-compatible device. As shown in the figure, the I^2C lines should be connected to Vdd (5 V supply) with 4.7 K resistors.

In the following example, a data byte is read from address 20 of the serial EEPROM and stored in variable B1. Note that the Control byte is set to "01010000", Address is assigned variable B0 and value 20 stored in it, and the byte read from the EEPROM is stored in data register B1.

```
Symbol con = %01010000
Symbol addr = B0
       addr = 20                ‘Set address to 20
       I2CIN con, addr, B1      ‘Read from address 20 to B1
```

I2COUT

`I2COUT` *`Control, Address, Value (,Value)`*

This command is used to send data to an I^2C compatible device such as a serial EEPROM described in command I2CIN. The *(,Value)* in the command is used for 16-bit information.

When writing data to an EEPROM, it is necessary to wait about 10 ms (device dependent) for the write operation to complete before attempting to write again. In the example given below, data byte 10 is written to address 30, and also data byte in variable B5 is written to address 31 of an EEPROM.

```
Symbol con = %01010000
Symbol addr = B0
       addr = 30                    ‘Set address to 30
       I2COUT con, addr, (10)       ‘Write byte 10 to address 30
       PAUSE 10                     ‘Wait 10ms
       addr 31                      ‘Set address to 31
       I2COUT con, addr, (B5)       ‘Write byte in B5 to address 31
       PAUSE 10                     ‘Wait 10 ms
```

INPUT

`INPUT` *`Pin`*

This makes the specified PORTB pin an input. *Pin* is from 0 to 7. For example,

```
INPUT 2                             ‘Make RB2 an input pin
```

LOOKDOWN

`LOOKDOWN` *`Search, (Constant, Constant,……), Var`*

This command provides a look-up table. It looks down a list of *Constants* and compares each one with the *Search* value. If a match is found, the position of the match is stored in *Var*. Note that the first *Constant* is assumed to be at position 0. The *Constant* list can be numeric or string constants. In the following example, if we assume that variable B0 has value 5 then variable B1 will contain 3, which is the position of 5 in the table:

```
LOOKDOWN B0, (0, 8, 9, 5, 12, 0, 1), B1
```

LOOKUP

```
LOOKUP Index, (Constant, Constant,…), Var
```

This command is used to retrieve values from a table. When *Index* is 0, *Var* is loaded with the first *Constant;* when *Index* is 1, *Var* is loaded with the second *Constant* and so on. In the following example, if we assume that variable B0 has value 3, variable B1 will be loaded with 8 which is the third element in the table starting from 0:

```
LOOKUP B0, (0, 9, 0, 8, 12, 32), B1
```

LOW

```
LOW Pin
```

This command makes the specified pin an output pin and clears it to logic 0. *Pin* only applies to PORTB pins and it can take values from 0 to 7. In the following example, bit 2 of PORTB is configured as an output pin and is cleared to logic 0:

```
LOW 2
```

NAP

```
NAP Period
```

The NAP command places the PIC microcontroller in low-power mode for a while to save power in battery applications. The Period is a variable from 0 to 7 and the approximate delay is given in Table 14.5.

In the following example the microcontroller is put into low power mode for just over 1 s:

```
NAP 6
```

OUTPUT

```
OUTPUT Pin
```

This command makes the specified pin of PORTB an output pin. Pin can take values from 0 to 7. In the following example, bit 2 of PORTB (RB2) is made an output pin:

```
OUTPUT 2
```

Table 14.5: Delay in NAP Command

Period	Delay (s, approx)
0	18×10^{-3}
1	36×10^{-3}
2	72×10^{-3}
3	144×10^{-3}
4	288×10^{-3}
5	576×10^{-3}
6	1.152
7	2.304

PAUSE

```
PAUSE Period
```

This is one of the commonly used commands to delay a program by a specified amount. Period is in milliseconds and can range from 1 to 65,535 ms (i.e., just over one minute). PAUSE does not put the microcontroller into low-power mode. In the following example, the program is delayed by 1 s:

```
PAUSE 1000
```

PEEK

```
PEEK Address, Var
```

This command is used to read the value of a RAM register at the specified Address and then put the value into variable Var. The PEEK command can be used to access all registers of the PIC microcontroller including the Port registers, A/D converter registers, etc.

In the following example, the 8-bit value of PORTB is read and stored in variable B0:

```
Symbol PORTB=6   'PORTB register address
PEEK PORTB, B0   'Read PORTB into B0
```

POKE

```
POKE Address, Var
```

This command is used to send data to a RAM register at the specified Address. The POKE command can be used to send data to all accessible registers of the PIC microcontroller, including the PORT registers, PORT direction registers, A/D converter registers, etc.

In the following example, TRISB is cleared to 0 so that all PORTB pins are outputs. The hexadecimal value 24 is sent to PORTB.

```
Symbol TRISB=$86    'TRISB register address
Smbol PORTB=6       'PORTB register address

POKE TRISB, 0       'Clear TRISB
POKE PORTB, $24     'Send $24 to PORTB
```

POT

```
POT Pin, Scale, Var
```

This command could be useful to read an analog voltage if the microcontroller has no built-in A/D converter. *Pin* is a PORTB pin and can take a value between 0 and 7. For this command to work, a resistor and a capacitor are serially connected to a port pin as shown in Fig. 14.3. When a voltage is applied to a resistor–capacitor circuit, the voltage across the capacitor rises exponentially as the capacitor is charged through the resistor. The charge time is dependent on the value of the resistor and the capacitor.

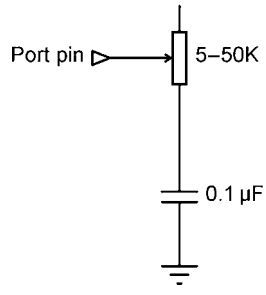


Figure 14.3: Resistor and Capacitor Connected to an I/O Pin

When the POT command is used, the capacitor is initially discharged by the I/O pin by placing the pin in output mode. After that, the I/O port is changed to an input port and starts timing the voltage across the capacitor until the voltage reaches the threshold value of the I/O pin. When this happens, the calculated charge time is converted into a number between 0 and 255 and is stored in *Var*. The *Scale* value should be set experimentally. To do this, set the device to maximum resistance and set the *Scale* to 255. The value returned in *Var* will be the proper scale value for the chosen components. An example is given below where the resistor–capacitor is connected to pin 1 of PORTB, the *Scale* value is set to 255 and the output value is stored in B0.

```
POT 1, 255, B0
```

PULSIN

```
PULSIN Pin, State, Var
```

The PULSIN command measures the pulse width of any signal connected to a PORTB pin. With a 4-MHz crystal or resonator, the pulse width will be measured in 10 μs units. If *State* is 0, the width of a low pulse is measured; if *Scale* is 1, the width if a high pulse is measured. The measured value in 10 μs units is stored in variable *Var*. *Var* can be a byte or a word. If a word is used, it can take values 1 to 65,535, i.e. the minimum pulse width that can be measured is 10 μs and the maximum is 655,350 μs. If a byte is used, the range of the measurement is 10 to 2550 μs.

PULSOUT

```
PULSOUT Pin, Period
```

This command generates a pulse on a PORTB pin (*Pin* can be 0 to 7) of specified *Period* in 10 μs units. The *Period* is a word and thus pulses of up to 655,350 μs can be generated. The specified pin is automatically made an output pin.

For example, to generate a 500-μs pulse on pin 1 of PORTB, we need the command

```
PULSOUT 1, 50
```

PWM

```
PWM Pin, Duty, Cycle
```

This command outputs a pulse-width-modulated (PWM) signal on the specified PORTB pin (*Pin* can be 0 to 7). The *Duty* is the pulse duty-cycle and can range from 0 to 255. 0 corresponds to a 0% duty-cycle, and 255 corresponds to a 100% duty-cycle. The generated PWM pulse is repeated *Cycle* times. The specified port *Pin* is made an output just before the command is executed and reverts to an input after the pulse is generated.

In the following example, a 200-cycle PWM signal is generated on bit 0 of PORTB with a duty-cycle of 50%:

```
PWM 0, 127, 200
```

Another use of this command is to generate an analog signal by sending the output to a resistor–capacitor circuit as shown in Fig. 14.4. In this circuit, the voltage across the capacitor will vary depending on the *Duty* and the *Cycle* of the pulses.

RANDOM

```
RANDOM Var
```

This command generates a random number and stores in word variable *Var*. For example, to generate a random number and store in W1 use the command:

```
RANDOM W1
```

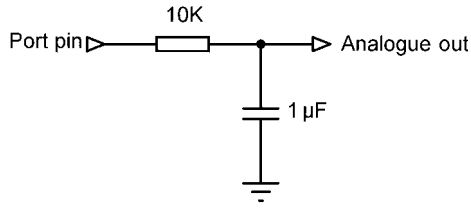


Figure 14.4: Using PWM Signal for D/A Conversion

READ

```
READ Address, Var
```

This command is used to read a byte from the specified *Address* of the built-in EEPROM memory. The byte read is stored in variable *Var*. This command can only be used with PIC microcontrollers that have built-in EEPROM memory (such as PIC16F84, or PIC16F877).

In the following example, the byte at address 10 of EEPROM is read and stored in variable B1:

```
READ 10, B1    'Read byte at address 10
               'and store in B1
```

REVERSE

```
REVERSE Pin
```

This command reverses the mode of a PORTB pin (*Pin* can be from 0 to 7). If the pin is an input, it is made an output. Similarly, if the pin is an output, it is made an input.

In the following example, bit 2 of PORTB is first made an output pin, then changed to an input pin:

```
OUTPUT 2     'RB2 is output pin
REVERSE 2    'RB' is an input pin
```

SERIN

```
SERIN Pin, Mode, (Qual, Qual,, ), Item, Item,
```

This command is used to receive RS232 serial asynchronous data on a PORTB pin (pin is between 0 and 7) using 8-bit data, no parity bit, and one stop bit. As shown in Table 14.6, *Mode* defines the baud rate and whether or not the pin data is inverted. For example, if Mode is N9600, the data is inverted and the selected baud rate is 9600.

The RS232 signal levels are 12 V and level converter circuits (such as MAX232) are normally used to convert the RS232 signal levels to TTL and the TTL levels back to RS232 levels. The I/O specifications of PIC microcontrollers allow RS232 signals to be directly connected to a port pin. As shown in Fig. 14.5, a resistor is all that is needed to receive RS232-compatible signals on a pin. When used in this mode, the data is to be inverted (i.e., use the "N" versions of the mode signals in Table 5.6)

Table 14.6: Selecting the Baud Rate with *Mode*

Symbol	Value	Baud Rate	Mode
T2400	0	2400	True
T1200	1	1200	True
T9600	2	9600	True
T300	3	300	True
N2400	4	2400	Inverted
N1200	5	1200	Inverted
N9600	6	9600	Inverted
N300	7	300	Inverted

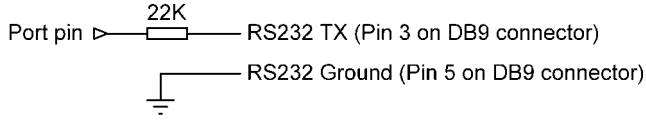


Figure 14.5: Connecting a RS232 Signal to a Port Pin

A number of qualifiers, enclosed in brackets, can be used with the SERIN command such that these bytes must be received before receiving the data items. Once the qualifiers are satisfied, SERIN receives the serial data and stores in *Item*s. The *Item* variable may be preceded by the hash character ("#"). This will convert the decimal number received into ASCII equivalent and store it in *Item*.

In the following example, pin 1 of PORTB (RB1) is defined as the serial I/O pin and the port pin is connected to the RS232 serial line using a resistor. The baud rate is assumed to be 4800. The microcontroller waits until the character "X" is received from the line and then stores the next byte in variable B0:

```
SERIN 1, N4800, ("X"), B0
```

SEROUT

```
SEROUT Pin, Mode, (Item, Item,…)
```

This command is similar to the SERIN command but is used to send RS232 asynchronous serial data to a pin of PORTB (*Pin* can be between 0 and 7). As before, *Mode* is used to set the communications baud rate. In addition to the standard inverted and non-inverted modes, it is also possible to set Open-Drain and Open-Collector modes where a pull-up resistor will be required at the output of the pin. Table 14.7 gives a list of the available *Modes*.

Data byte Item is sent to the specified port pin in serial format. The Item can be a string constant or a numeric value. A string constant consists of characters and each character of the string is sent out. For example, the string "COMPUTER" is sent out as 8 individual characters.

Table 14.7: Selecting the Baud Rate with *Mode*

Symbol	Value	Baud Rate	Mode
T2400	0	2400	True
T1200	1	1200	True
T9600	2	9600	True
T300	3	300	True
N2400	4	2400	Inverted
N1200	5	1200	Inverted
N9600	6	9600	Inverted
N300	7	300	Inverted
OT2400	8	2400	Open Drain
OT1200	9	1200	Open Drain
OT9600	10	9600	Open Drain
OT300	11	300	Open Drain
ON2400	12	2400	Open Source
ON1200	13	1200	Open Source
ON9600	14	9600	Open Source
ON300	15	300	Open Source

A numeric value will send the corresponding ASCII character. For example, 13 is the carriage-return character, 65 is character "A" and so on. A numeric value can be preceded by the hash character "#" and this will send out the ASCII representation of its decimal value. For example, #345 will be sent as "3", "4", and "5".

In the following example, it is assumed that pin 1 of PORTB (RB1) is used as the serial I/O pin and it is configured for 4800 baud. ASCII value of variable B0 is sent out from this pin, followed by a carriage-return.

```
SEROUT 1, N4800, (#B0, 13)
```

SLEEP

```
SLEEP Period
```

The SLEEP command is used to put the microcontroller in low-power mode and stops the micro-controller running for the specified *Period*. The *Period* is a word and can range from 1 to 65,535 and represents increments of 2.3 s. For example, a value of 1 will make the microcontroller sleep for 2.3 s, a value of 2 will make the microcontroller sleep for 4.6 s and so on. The maximum value of 65,535 makes the microcontroller sleep just over 18 h.

In the following example, the microcontroller sleeps for 23 s:

```
SLEEP 10
```

SOUND

```
SOUND Pin, (Note, Duration, Note, Duration,…)
```

This command is used to generate sound on a specified PORTB pin of the microcontroller (*Pins* are between 0 and 7). Note can take values from 0 to 255 and these values do not correspond to the musical notes. A 0 represents silence. Values from 1 to 127 are tones (1 is lower frequency than 127), and values from 128 to 255 are white noise (128 is lower frequency than 255). The sound continues for a length of time specified by Duration. Duration is measured in milliseconds and it can take values between 0 and 255. The SOUND command produces TTL level square waves and it is possible to connect a speaker to the output pin as shown in Fig. 14.6.

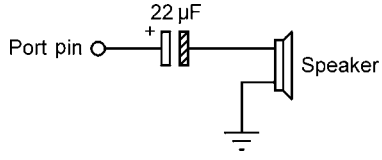


Figure 14.6: Connecting a Speaker for the SOUND Command

In the following example, a sound with note 20 and duration 100 ms is sent to pin 0 of PORTB. Then, another sound with note 23 and duration 200 ms is sent out from the same port pin.

```
SOUND 0, (20, 100, 23, 200)
```

TOGGLE

```
TOGGLE Pin
```

This command makes the specified *Pin* an output pin and inverts the state of this pin (*Pin* can take values from 0 to 7).

In the following example, bit 0 of PORTB (RB0) is first made low, and then changed to high using the TOGGLE command:

```
LOW 0
TOGGLE 0
```

WRITE

```
WRITE Address, Value
```

The WRITE command writes the Value byte to the specified EEPROM address. This command is only valid for the PIC microcontrollers which have built-in EEPROM memories.

In the following example byte in variable B0 is written to EEPROM address 2:

```
WRITE 2, B0
```

14.1.5 Recommended PicBasic Program Structure

There are many different ways in which a PicBasic program can be written. It is important to note that a program should be written in such a way that it is easily maintainable by other people.

This is specially important if you work in a firm and others may have to upgrade or maintain your program. The following steps should be followed to develop a maintainable program:

- Use a header in your programs. This header should briefly describe the function of the program. In addition, the author of the program, the program creation date, program file name, and any program modifications should be described in the header.
- Use comments in your programs to describe what you are trying to do. The comments can be used at the beginning of a piece of code, or after every statement.
- Use symbols as much as possible in your programs. Symbols make your programs more readable.

The author recommends that you use a template similar to the one given in Fig. 14.7 when developing PicBasic programs. As you can see in this figure, the header includes a brief description of the program, name of the author, the date, and the filename of the program. Comments are used in every line of the program to clarify the actions of the program.

14.2 PicBasic Pro Language

PicBasic Pro is a full-featured compiler and is for serious or professional PIC programmers. PicBasic Pro has many additional commands compared to the standard PicBasic compiler. In addition, the variables, constants and symbols are treated differently in PicBasic Pro. In this section, we shall only be looking at the commands specific to the PicBasic Pro language, and which have not been described in Section 14.1. Also, various features of the PicBasic Pro language are described in this section.

14.2.1 PicBasic Pro Variables

Variables in PicBasic Pro are stored in the general purpose RAM registers and are declared using the VAR keyword. Each variable has a name and a variable type. A variable type can be a bit, a byte, or a word. Some example variable declarations are

```
Total      VAR      word
Count      VAR      byte
Flag       VAR      bit
```

The VAR keyword can also be used to create an alias for a variable (i.e., another name). In the following example, *Sum* is another name for *Total*:

```
Sum        VAR      Total
```

The individual bits of a variable can be accessed by writing the variable name, followed by a dot "." character, and then the bit number (0 to 15), or the keyword BIT followed by the bit number (e.g., BIT0 to BIT15). The following are examples of accessing bit 0 of variable Total:

```
Total.0
Total.BIT0
```

```
Ô ***************************************************************************
‘
‘                          LED FLASHING PROGRAM
‘                          ====================
‘
‘ This program flashes and LED connected to port RB0 of PORTB. The
‘ Led is flashed with 1 second intervals.
‘
‘ Author:          Dogan Ibrahim
‘ Date:            September, 2005
‘ File:            LED.PBC

‘ Modifications
‘ ==========
‘
‘ ***************************************************************************
‘
‘ SYMBOLS
‘
Symbol LED = 0                  ‘ Define RB0 as LED
Symbol TRISB = $86              ‘ TRISBaddress
Symbol PORTB = $06              ‘ PORTB address
‘
‘ START OF MAIN PROGRAM
‘
        POKE TRISB, 0           ‘ Set PORTB pins as outputs
AGAIN:
        HIGH LED                ‘ Turn ONLED
        PAUSE 1000              ‘ Wait 1 second
        LOW LED                 ‘ Turn OFFLED
        PAUSE 1000              ‘ Wait 1 second
        GOTO AGAIN              ‘ Repeat
        END                     ‘ End of program
```

Figure 14.7: Recommended PicBasic Program Template

Arrays of variables can be created in PicBasic Pro by writing the name of the array, followed by the keyword VAR, and then the type and the size of the array. For example, a byte array called *Sum* with 10 elements of type *byte* can be declared as

```
Sum     VAR   byte[10]
```

In the above example, the first element of the array is *Sum[0]*, and the last element is *Sum[9]*. Arrays have a size-limit in PicBasic Pro.

- Maximum size of a bit array is 256
- Maximum size of a byte array is 96 (microcontroller-dependent)
- Maximum size of a word array is 48 (microcontroller-dependent)

14.2.2 Constants

Constants in the PicBasic Pro language are declared using the CON keyword. A constant value cannot be changed in a program.

In the following example, *Maxim* is declared as 10 and its value cannot be changed in the program:

```
Maxim     CON    10
```

14.2.3 Comments

Comments in PicBasic Pro are declared the same as in the PicBasic language—i.e., using the REM keyword or a single quote at the beginning of a line.

14.2.4 Multi-Statement Lines

Multi-statement lines are created as in PicBasic, by separating each statement with a colon ":" character.

14.2.5 INCLUDE

Other PicBasic Pro source files can be included in a program as in PicBasic language.

14.2.6 DEFINE

This command defines various compiler options, such as the clock oscillator frequency, pin number, etc.

14.2.7 Line Extension

When writing long programs, it may be necessary to continue part of a statement on a new line. A line can be extended by typing the line extension character "_" as the last character in the line to be continued. For example,

```
Item1, Item2, Item3, Item4, _
Item5, Item6
```

14.2.8 Accessing Ports and Other Registers in PicBasic Pro

PIC microcontroller ports or any other registers can easily be accessed by simply writing the name of the port or the register and using the equate "=" character. For example,

```
A = PORTA
UP = PORTB & $F0
PORTB = $2F
INTCON = $0F
```

The bits of a port or a register can be accessed by simply writing the name of the port or the register, followed by a dot "." character and the PORTBit to be accessed. For example,

```
L = PORTB.1           'Read bit 1 of PORTB and load into L
L = PORTB.BIT1        'Read bit 1 of PORTB and load into L
K = STATUS.0          'Read bit 0 of STATUS register and load into K
```

In most of the PicBasic I/O commands, *Pin* is used to define a pin of PORTB where *Pin* can take a value from 0 to 7 corresponding to PORTB pins. In a similar manner, PicBasic allows the use of numbers 0 to 15 to access port I/O pins. When only a number is used to access a port pin, the port and the pin number accessed depends on the package size of the microcontroller used. Table 14.8 shows the Pin definitions for 8 to 40 pin PIC microcontrollers.

Table 14.8: Port I/O *Pin* Definitions

PIC Micro Size	Pin 0–7	Pin 8–15
8 pin	GPIO	–
18 pin	PORTB	PORTA
24 pin (except 14,000)	PORTB	PORTC
28 pin	PORTC	PORTD
40 pin	PORTB	PORTC

For example, assuming we are using an 18-pin PIC microcontroller, the PicBasic command:

```
SOUND 3, 10,100
```

generates a sound with note 10 and duration 100 ms from bit 3 of PORTB (i.e., RB3). In PicBasic Pro, we can use the same statement, or we can write

```
SOUND PORTB.3, 10, 100
```

If we wish to generate the sound from bit 0 of PORTA, in PicBasic Pro we can write

```
SOUND PORTA.0, 10, 100
```

or

```
SOUND 8, 10, 100
```

There is no way of generating a sound from PORTA using the PicBasic language.

The direction of a port is determined by loading the corresponding TRIS register. For an output pin, a 0 is loaded into the corresponding TRIS register, and for an input pin a 1 is loaded into the corresponding TRIS register. In PicBasic Pro, the TRIS register can be accessed directly like any other register. For example, to configure all PORTB pins as outputs and then send the hexadecimal value $FF to PORTB we can write

```
TRISB = 0
PORTB = $FF
```

14.2.9 Arithmetic Operators

PicBasic Pro supports more arithmetic operators than PicBasic. Table 14.9 lists all the arithmetic operators supported by PicBasic Pro. In this section, we shall be looking only at these additional operators.

Table 14.9: PicBasic Pro Arithmetic Operators

Arithmetic Operator	Description
+ − * /	Add, subtract, multiply, divide
**	Top 16 bits of multiplication
*/	Middle 16 bits of multiplication
//	Remainder
<< >>	Shift left, shift right
ABS	Absolute value
COS	Cosine
DCD	Decode
DIG	Digit
MAX MIN	Maximum, minimum
NCD	Encode
REV	Reverse bits
SIN	Sine
SQR	Square root
& | ^ ~	Bitwise AND, OR, EXOR, NOT
&/ |/ ^/	Bitwise NAND, NOR, INOR

Shift

The shift operators "<<" and ">>" are used to shift a value left or right, respectively, 0 to 15 times. Zeroes are placed to the shifted positions. Shifting left is same as multiplying the number by 2, and shifting right is same as dividing the number by 2.

In the following first example, variable *Cnt* is shifted left twice. In the second example, variable *Sum* is shifted right three times.

```
Cnt = cnt << 2   'Shift left Cnt by 2 places
Sum = Sum >> 3   'Shift right Sum by 3 places
```

ABS

Operator ABS returns the absolute value of a number. In the following example, the absolute value of variable *p* is returned:

```
p = ABS p          'Return the absolute value
```

COS

Returns the cosine of a number. The result is in 2's complement format in the range 127 to 127. The number must be in radians in the range 0 to 255. In the following example, the cosine of 8 radians is returned:

```
Angle = COS 8        'Return the cosine of 8
```

DCD

This operator is used to set a bit of a byte or a word to 1. All other bits are set to 0. For example, to set bit 4 of a byte we can write

```
B4 = DCD 4           'Set bit 4 of variable B4
```

where variable B4 will take the binary value %00010000

DIG

This operator returns a digit of a number. The number can be up to 4 digits with the rightmost digit being digit 0. For example, if variable *Sum* is equal to 678, the first digit (number 7) can be extracted as

```
Sum = 678            'Sum 678
P = Sum DIG 1        'P = 7
```

NCD

The NCD operator is used to find the highest bit number set in a number. The bit numbers can range from 1 to 16. A zero is returned if no bit is set. In the following example, variable P 6 since the highest bit set in the number is the sixth bit (starting from 1).

```
P = NCD %00101011    'Highest bit set is 6
```

SIN

This operator is similar to the COS operator and it returns the sine of a number. The number must be expressed in radians and it must be between 0 and 255. For example, to find the sine of 10 radians, use

```
P = SIN 10
```

SQR

This operator returns the square root of a number. The result is an integer number. For example, to find the square root of variable *Total*, use

```
N = SQR Total        'Find square root of Total
```

14.2.10 PicBasic Pro Commands

PicBasic Pro has over 80 commands. Some commands are similar to the PicBasic commands with minor changes. For example, the range of the Pin variable is from 0 to 15, instead of 0 to 7. It is the author's recommendation that you use the port name, followed by a dot and the bit number when you wish to access a port pin. This makes your programs much more readable and easier to maintain.

In this section, we shall only look at the commonly used commands specific to the PicBasic Pro language. Further information about these or any other commands can be obtained from the PicBasic Pro user manual.

ADCIN

```
ADCIN Channel, Var
```

This command is used to read the on-chip A/D converter. This is not a very useful command and we shall see in the projects section how to read data from the A/D channel of a PIC microcontroller.

BRANCHL

```
BRANCHL Index, (Label, Label,......)
```

The BRANCH command used in the PicBasic language causes a limited range of branch (usually 1 K). The BRANCHL command can be used to create longer jumps in the program memory. The BRANCHL command is slower than the BRANCH command and generates more assembly code.

CLEAR

```
CLEAR
```

This command clears (zeroes) all the RAM registers in each bank.

CLEARWDT

```
CLEARWDT
```

If the watchdog timer is enabled, it can time out and reset the program to the beginning (address 0). The CLEARWDT command is used to reset the watchdog timer so that it does not time out.

COUNT

```
COUNT Pin, Period, Var
```

This command is used to count the number of pulses that occur on *Pin* during the *Period* and stores the result in *Var*. *Pin* can take values 0 to 15 but the "*Portname.number*" format is recommended (e.g., PORTB.0).

The highest frequency that can be counted with a 4-MHz crystal clock is 25 kHz, and 125 kHz when a 20-MHz clock is used. In the following example, the number of pulses on bit 0 of PORTB are counted in 100 ms and stored in variable *Cnt*:

```
COUNT PORTB.0, 100, Cnt
```

DATA

```
DATA @Location, Constant, Constant,.......
```

This command stores constants in the on-chip EEPROM memory during the programming of the device (not when the program is run). The command can only be used with the PIC microcontrollers that have on-chip EEPROMs. *Location* denotes the starting address of the EEPROM and if omitted, address 0 is assumed.

The following example shows how the numbers 5, 10, 15, and 20 can be stored in EEPROM starting from address 6:

```
DATA @6, 5, 10, 15, 20
```

DTMFOUT

```
DTMFOUT Pin, Onms, Offms, [Tone, Tone,.......]
```

This command produces Touch Tones normally available in keyboards and mobile phones. *Pin* can take a value between 0 and 15 (or *Portname.number*) and the specified pin is made an output. *Onms* is the duration of each tone in milliseconds, and *Offms* is the number of milliseconds pause between each tone. If the *Onms* or the *Offms* are not specified, they default to 200 ms and 50 ms, respectively.

A *Tone* can take a value between 0 and 15. Tones 0 to 9 are the same as on a telephone keypad. *Tone* 10 is the * key, *Tone* 11 is the # key, and *Tones* 12–15 are the extended keys A to D. The sound generated by the DTMFOUT should be smoothed using resistor–capacitor filters. It is recommended to use a high clock rate (e.g., 20 MHz) to get a smooth signal after the filtering.

In the following example, the DTMF tones for numbers 886 are sent from bit 0 of PORTB with the default duration and pause:

```
DTMFOUT PORTB.0, [8, 8, 6]
```

FREQOUT

```
FREQOUT Pin, Onms, Frequency1, [,Frequency2]
```

This command generates a signal with one or two different frequencies on the specified *Pin* for *Onms* milliseconds. *Pin* is automatically made an output and it can be 0 to 15 or a *Portname.number*. The generated signal is a square wave and filtering may be required to obtain a smooth signal.

In the following example, a 1-kHz signal is generated on port 0 of PORTB for 3 s:

```
FREQOUT PORTB.0, 3000, 1000
```

HPWM

```
HPWM Channel, Dutycycle, Frequency
```

Some PIC microcontrollers have one or more built-in circuits to generate pulse width–modulated square-wave signals (PWM). For example, PIC16F877 has two PWM *Channel*s. Channel 1 is known as CCP1 (also PORTC.2) and Channel 2 is known as CCP2 (also PORTC.1).

Dutycycle can vary from 0 to 255 which corresponds to 0% (low all the time) to 100% (high all the time), respectively. A value of 127 gives 50% duty cycle. The highest *Frequency* is 32,767 Hz, and on microcontrollers with two channels, the *Frequency* must be the same on both channels.

The PWM signal is output from the specified pin continuously in the background while the program executes other instructions.

In the following example, a 1-kHz, 50% duty cycle PWM signal is generated from Channel 1 (CCP1) of a PIC16F877 type microcontroller:

```
HPWM 1, 127, 1000
```

HSERIN

HSERIN2

These commands are only available on microcontrollers that have built-in serial port devices such as an USART. The use of these commands is complicated and more details can be obtained from the PicBasic Pro user manual.

HSEROUT

HSEROUT2

These commands are only used on microcontrollers that have built-in serial port devices such as an USART. The commands are used to send out serial asynchronous data from the

microcontroller with the required format. The use of these commands is complicated and more details can be obtained from the PicBasic Pro user manual.

IF..THEN..ELSE

These commands are similar to the PicBasic IF..THEN command but the PicBasic Pro language provides more flexibility when one or more comparisons are made. These commands can be used in the following formats:

Format 1:
```
IF Comparison [AND/OR Comparison…] THEN Label
```

Format 2:
```
IF Comparison [AND/OR Comparison…] THEN Statement….
```

Format 3:
```
IF Comparison [AND/OR ComparisonÖ] THEN
Statement….
ELSE
    Statement
ENDIF
```

Some examples for the use of this command are given below:

Conditional statement:
```
IF PORTB.0=0 THEN Led=1
```

Conditional jump:
```
IF (PORTB.0=0) AND (PORTB.1=1) THEN Loop
```

Multiple statements:
```
IF Cnt<10 THEN A=A+1:B=B+1
```

Multiple statements:
```
IF SUM<10 THEN
      Cnt=Cnt + 1
      Tot=Tot + 1
ENDIF
```

IF..THEN..ELSE

```
IF Total=100 THEN
     Flag=1
ELSE
     Flag=0
ENDIF
```

PAUSEUS

```
PAUSEUS Period
```

This command pauses the program for Period microseconds. Period is a word in the range 1 to 65,535. Thus, the maximum delay is 65.535 μs. PAUSEUS command assumes that we are using a 4-MHz clock. The minimum delay that can be generated with PAUSEUS using a 4-MHz clock is 24 μs.

REPEAT..UNTIL

```
REPEAT
      Statement….
UNTIL Condition
```

This command is used to create loops in programs. The statements between the REPEAT and UNTIL are executed until the specified *Condition* is true.

In the following example, the statements between REPEAT and UNTIL are executed 10 times:

```
k = 0
REPEAT
      Sum = Sum + 1
      Cnt = Sum
      k = k + 1

UNTIL k < 10
```

SELECT..CASE

```
SELECT CASE Var
      CASE Expr1 [,Expr…]
          Statement…
      CASE Expr2 [,Expr…]
          Statement…
       [CASE ELSE
          Statement…]
END SELECT
```

This command is used instead of using multiple IF..THEN commands. The variable *Var* is compared with different values (or ranges of values) and an action is taken based on its value. If *Var* does not match any of the conditions, then the statements after the CASE ELSE are executed. The IS keyword is used after CASE to specify a comparison other than equal to.

In the following example, if x is 1, B is set to 100. If x is 2, B is set to 6. If x is 3 or 4, B is set to 50. If x is greater than 120, B is set to 1. If x is none of these, then B is set to 0.

```
  SELECT CASE x
       CASE 1
            B=100
       CASE 2
            B=6
       CASE 3, 4
            B = 50
       CASE IS > 120
            B = 1
       CASE ELSE
            B = 0
END SELECT
```

SHIFTIN

`SHIFTIN` *`Datapin, Clockpin, Mode, [Var{\bits}, Var{\bits},…]`*

The SHIFTIN command is used to read data one bit at a time as clock is sent out to the sending device. The received data is stored in variables *Var*. *Datapin* is either from 0 to 15 or a *Portname. number* and specifies the pin number which is to receive the data. *bits* optionally specify the number of bits to shift in and if omitted, 8 bits are assumed. *Clockpin* is either 0 to 15 or a *portname. number* and specifies the pin number where the clock is sent out. *Mode* has a value between 0 and 7 and it specifies the mode of the clock operation as shown in Table 14.10. For *Mode*s between 0 and 3, the clock output is normally low and goes high to clock in a bit, then returns low. For *Mode*s between 4 and 7, the clock output is normally high and goes low to clock in a bit, then returns high.

Table 14.10: SHIFTIN Command Clock *Modes*

Mode No.	Operation
0	Shift in MSB first. Read before sending clock. Clock normally low
1	Shift in LSB first. Read before sending clock. Clock normally low
2	Shift in MSB first. Read after sending clock. Clock normally low
3	Shift in LSB first. Read after sending clock. Clock normally low
4	Shift in MSB first. Read before sending clock. Clock normally high
5	Shift in LSB first. Read before sending clock. Clock normally high
6	Shift in MSB first. Read after sending clock. Clock normally high
7	Shift in LSB first. Read after sending clock. Clock normally high

In the following example, data bits are received into bit 0 of PORTB and stored, LSB first, followed by 8 data bits in variable B1. *Mode* 0 is used here and the clock is sent out from bit 1 of PORTB.

```
SHIFTIN PORTB.0, PORTB.1, 0, [B1\8]
```

SHIFTOUT

```
SHIFTOUT Datapin, Clockpin, Mode, [Var{\bits}, Var{\bits},...]
```

This command is similar to SHIFTIN, but here, data bits are sent out one bit at a time. *Datapin* can be 0 to 15 or a *Portname.number*. *bits* optionally specify the number of bits to be shifted out and if omitted, 8 bits are assumed. *Mode* specifies which bit will be sent out first. If *Mode* is 0, the LSB is sent out first followed by other data bits. If *Mode* is 1, the MSB is sent out first followed by other data bits.

In the following example, the contents of variable B1 are sent out as 8 bits, LSB first, from bit 0 of PORTB. Bit 1 of PORTB is used as the clock pin.

```
SHIFTOUT PORTB.0, PORTB.1, 0, B1
```

SWAP

```
SWAP Var, Var
```

This command is used to swap the contents of two variables. It can be used with bit, byte, and word variables.

In the following examples, values of variables B1 and B2 are exchanged:

```
SWAP B1, B2
```

WHILE..WEND

```
WHILE condition
      Statement…
WEND
```

This is another command used to create loops in programs. The statements between the WHILE and WEND are repeated while the *Condition* is true.

In the following example, the statements between WHILE and WEND are repeated 10 times:

```
k = 0
WHILE k < 10
   Sum = Sum + 1
   B0 = B0 + 2
   k = k + 1
   WEND
```

14.3 Liquid Crystal Display (LCD) Interface and Commands

In many microcontroller-based applications, it is required to display a message or the value of a variable. For example, in a temperature-control application, it may be required to display the value of the temperature dynamically. Basically, three types of displays can be used in practice.

These are video displays, 7-segment LED displays, and LCD displays. Standard video displays require complex interfaces and their cost is relatively high and their operation is not covered in this book. 7-segment LED displays are made up of LEDs. Although the 7-segment LEDs are bright, their disadvantage is the high power consumption which makes them unsuitable in many battery-operated portable applications.

LCDs are alphanumeric displays which are frequently used in microcontroller-based applications. Some of the advantages of LCDs are their low cost and low power consumption. LCDs are ideal in low-power, battery-operated portable applications. These displays come in different shapes and sizes. Some LCDs have 40 or more characters with several rows. Some more advanced LCDs can be programmed to display graphics images. Some modules, such as the ones used in games, offer color displays while some others may incorporate back lighting so that they can be viewed in dimly lit conditions. In this section, we shall be looking at how we can interface the standard LCDs to a PIC microcontroller and what commands are available to use the LCDs.

There are basically two types of LCDs as far as the interface technique is concerned: parallel LCDs and serial LCDs. Parallel LCDs are connected to the microcontroller I/O ports using 4 or 8 data wires and data is transferred from the microcontroller to the LCD in parallel form. Serial LCDs are connected to the microcontroller using only one data line and data is transferred to the LCD using the standard RS232 asynchronous data communication protocols. Serial LCDs are easier to use but they usually cost more than the parallel ones. Serial LCDs also have the advantage that only one wire is required to interface them to a microcontroller, thus saving the I/O pins. In this section, we shall be looking at the interface and programming of both types of LCDs.

14.3.1 Parallel LCDs

Figure 14.8 shows a typical parallel LCD. The programming of a parallel LCD is usually a complex task and requires a good understanding of the internal operation of the LCDs,

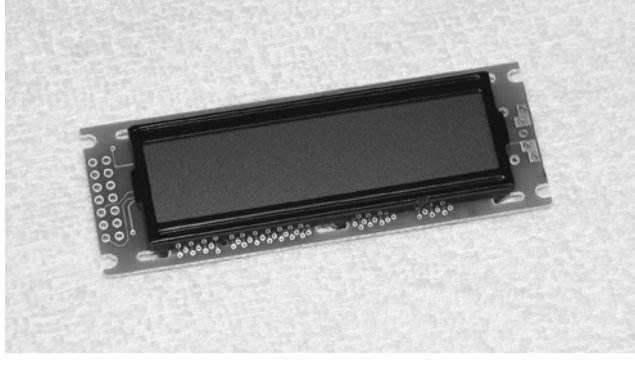


Figure 14.8: A Typical Parallel LCD

including the timing requirements. Fortunately, the PicBasic Pro language provides special commands for displaying data on HD44780 or compatible LCDs. All the user has to do is connect the LCD to the appropriate I/O ports and then use these special commands to simply send data to the LCD. The standard PicBasic language does not provide any special commands for programming the parallel LCDs.

14.3.1.1 HD44780 LCD Module

HD44780 is one of the most popular LCD modules used in the industry and also by hobbyists. This module is monochrome and comes in different shapes and sizes. Modules with line lengths of 8, 16, 20, 24, 32, and 40 characters can be selected. Depending on the model chosen, 1, 2, or 4 display rows can be selected. The display has a 14-pin connector for interfacing to a microcontroller. Table 14.11 shows the pin configuration of the LCD. A description of the pin functions is given below.

- V_{SS} is the 0 V or ground. V_{DD} pin should be connected to the positive supply. Although the manufacturers specify a 5 V supply, the module can be operated with as low as 3 V or as high as 6 V.
- Pin 3 is named as VEE and this is the contrast control pin. This pin is used to adjust the contrast of the LCD and it should be connected to a variable voltage supply. A potentiometer is usually connected between the power supply lines with its wiper arm connected to this pin so that the contrast can be adjusted. This pin can be connected to ground if contrast adjustment is not needed.

Table 14.11: Pin Configuration of HD44780 LCD

Pin No	Name	Function
1	V_{SS}	Ground
2	V_{DD}	Positive supply
3	V_{EE}	Contrast
4	RS	Register select
5	R/W	Read/write
6	E	Enable
7	D0	Data bit 0
8	D1	Data bit 1
9	D2	Data bit 2
10	D3	Data bit 3
11	D4	Data bit 4
12	D5	Data bit 5
13	D6	Data bit 6
14	D7	Data bit 7

- Pin 4 is the Register Select (RS) and when this pin is LOW, data transferred to the display is treated as commands. When RS is HIGH, character data can be transferred to and from the module.
- Pin 5 is the read/write (R/W) pin. This pin is pulled LOW in order to write commands or character data to the LCD module. When this pin is HIGH, character data or status information cannot be read from the module. This pin is usually connected to ground, i.e., the LCD is put into write mode.
- Pin 6 is the Enable (E) pin which is used to initiate the transfer of commands or data between the LCD module and the microcontroller. When writing to the display, data is transferred only on the HIGH to LOW transition of this pin. When reading from the display, data becomes available after the LOW to HIGH transition of the enable pin and this data remains valid as long as the enable pin is HIGH.
- Pins 7 to 14 are the eight data bus lines (D0 to D7). Data can be transferred between the microcontroller and the LCD module using either an 8-bit interface, or a 4-bit interface. In the latter case, only the upper four data lines (D4 to D7) are used and the data is transferred as two 4-bit nibbles. This mode has the advantage that fewer I/O lines are required to communicate with the LCD.

14.3.1.2 Connecting the LCD to the Microcontroller

PicBasic Pro compiler by default assumes that the LCD is connected to specific pins of the micro-controller unless told otherwise. It assumes the following connections:

LCD	Microcontroller
D4	RA0
D5	RA1
D6	RA2
D7	RA3
E	RB3
RS	RA4

Figure 14.9 shows the circuit diagram with the default connections between the LCD and the microcontroller. In addition to the above connections, the R/W pin of the LCD is not used and is connected to the ground. The contrast adjustment is done by connecting a potentiometer to VEE. Notice that port pin RA4 is connected to the 5 V supply with a resistor. This is because this pin is open-drain output and should be pulled HIGH with a resistor.

When the above connections are made between the microcontroller and the LCD, we can simply use the LCDOUT command to send data to the LCD module. Note that the connections between the microcontroller and the LCD can be changed using a set of DEFINE commands to assign the LCD pins to the PIC microcontroller.

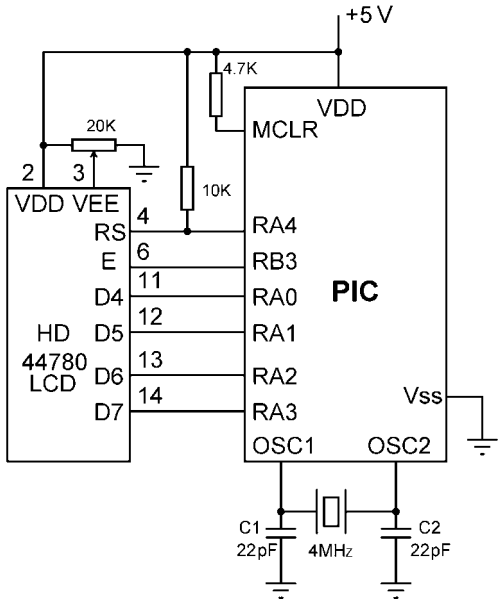


Figure 14.9: Default LCD Connections to a PIC Microcontroller

In the following example, PORTB pins 0 to 4 are used for LCD data (i.e., RB0 connected to D4, RB5 connected to D5, etc.), bit 4 of PORTB is connected to the RS pin of the LCD, bit 5 of PORTB is connected to the E pin of the LCD, the LCD is set for 4-bits of operation, and the LCD is assumed to have two rows.

```
DEFINE LCD_DREG        PORTB     ' Set LCD data port to PORTB
DEFINE LCD_DBIT        0         ' Set data starting bit to 0
DEFINE LCD_RSREG       PORTB     ' Set RS register port to PORTB
DEFINE LCD_RSBIT       4         ' Set RS register bit to 4
DEFINE LCD_EREG        PORTB     ' Set E register port
DEFINE LCD_EBIT        5         ' Set E register bit to 5
DEFINE LCD_BITS        4         ' Set 4 bit operation
DEFINE LCD_LINES       2         ' Set number of LCD rows
```

The format of the LCDOUT command is

```
LCDOUT Item, Item,...
```

where Item can be a command or data. A command is used to clear the display, home the cursor, move the cursor to left or right, etc. It is important that a program should wait for at least half a second before sending the first command to the LCD. This is because it can take quite a while before the LCD initializes itself.

Table 14.12 gives a list of the available commands. All commands must be preceded by the hexadecimal number $FE. For example, to clear the display we have to issue the command

```
LCDOUT $FE, 1
```

Table 14.12: LCD Commands

Command	Operation
$FE, 1	Clear display
$FE, 2	Home cursor
$FE, $0C	Cursor off
$FE, $0E	Underline cursor on
$FE, $0F	Blinking cursor on
$FE, $10	Move cursor left by one position
$FE, $14	Move cursor right by one position
$FE, $80	Move cursor to the beginning of first row
$FE, $C0	Move cursor to the beginning of second row
$FE, $94	Move cursor to the beginning of third row
$FE, $D4	Move cursor to the beginning of fourth row

Similarly, to move the cursor left by one position we have to issue the command

```
LCDOUT $FE, $10
```

Also, to move the cursor to the 5th position in the first row, we have to use the command

```
LCDOUT $FE, $80 + 5
```

Data is sent to the LCD using the LCDOUT command. The character set of the LCD is given in Table 14.13. A string can be sent to the LD by enclosing it in double-quotes. For example, the following command displays the string HELLO at the current cursor position:

```
LCDOUT "HELLO"
```

If a hash sign (#) precedes a variable (or if the characters DEC precede a variable), the ASCII representation for each digit is sent to the LCD. For example, if the variable B1 = 208, then the command

```
LCDOUT #B1
```

or

```
LCDOUT DEC B1
```

displays the characters "2", "0", and "8" on the LCD.

Table 14.13: LCD Character Table

If character BIN precedes a variable, the ASCII representation of its binary value is sent to the LCD. For example, if the variable B1 = 9, then the command

```
LCDOUT BIN B1
```

displays the characters '1001' on the LCD.

A numeric value preceded by HEX will send the ASCII representation of its hexadecimal value to the LCD. For example, if B0 = 255, then the command

```
LCDOUT HEX B0
```

will display "FF" on the LCD. It is also possible to send repeated characters to the LCD. In the following example, the characters "AAAAA" are sent to the LCD:

```
LCDOUT REP "A"\5
```

Example 14.1

A 2-row parallel LCD is connected to a PIC microcontroller as shown in Fig. 14.9. Write a PicBasic Pro program to display the string "PIC ROW 1" and "PIC ROW 2" in row 1 and row 2 of the LCD, respectively.

Solution 14.1

The required program is

```
PAUSE 1000              ' Wait 1 second for initialization
LCDOUT $FE ,1           ' Clear the LCD

LCDOUT "PIC ROW 1"      ' Display message in row 2
LCDOUT $FE, $C0         ' Move cursor to row 2
LCDOUT "PIC ROW 2"      ' Display message in row 2
```

14.3.2 Serial LCDs

A serial LCD is connected to a microcontroller using only one data line. Both PicBasic and PicBasic Pro languages can be used to send data to serial LCDs using the SEROUT command.

A popular serial LCD is the ILM-216 (see Fig. 14.10). This is a 16-pin, 2-row by 16-character LCD manufactured by Scott Edwards Electronics Inc. The device can operate with a baud rate of 2400 or 9600. In addition to the normal display functions, inputs for four push-button switches and also an output to drive a buzzer are included on the LCD module. The module incorporates an EEPROM memory and a backlight which are programmable.

Table 14.14 shows the pin configuration of this LCD. Pins 1 and 2 are the ground and the 5 V supply connections, respectively. Pin 3 is the serial input pin. Either RS232 voltage levels or standard TTL level signals can be connected to this pin. Similarly, pin 4 is the serial output pin and TTL logic levels (inverted) can be connected to this pin. Pin 5 is the buzzer out pin where a small buzzer (up to 25 mA) can be connected to this pin and the buzzer can be controlled with the software. Pins 6 to 8 are the option pins. Pin 7 is used to configure the device. Pin 8 is used to select a baud rate and when this pin is connected to pin 6, the device operates at 9600 baud. Leaving pin 8 unconnected configures the device to operate at 2400

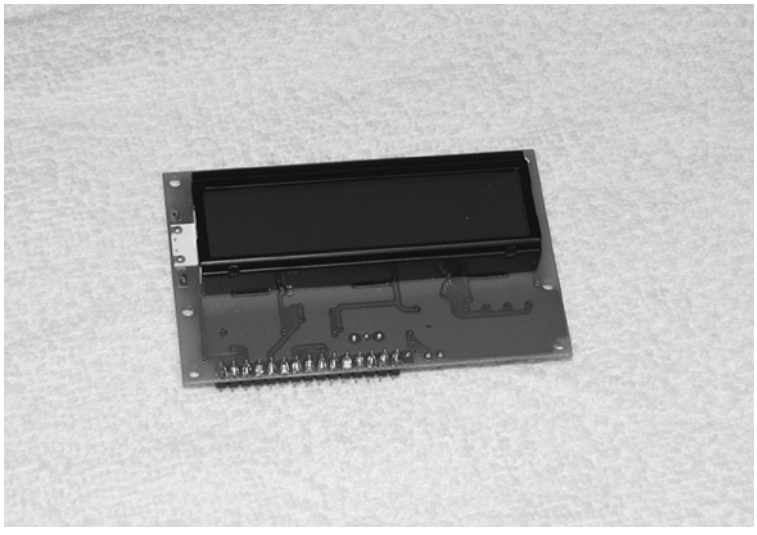


Figure 14.10: ILM-216 Serial LCD

Table 14.14: Pin Configuration of ILM-216

Pin No	Function
1	Ground
2	+5V
3	Serial in
4	Serial out
5	Bell
6	Ground
7	Config/test
8	9600 baud
9	Switch 1
10	Switch 1 ground
11	Switch 2
12	Switch 2 ground
13	Switch 3
14	Switch 3 ground
15	Switch 4
16	Switch 4 ground

baud. Pins 9 to 16 are four push-button switch inputs. The state of these pins can be read from the software.

The ILM-216 can be connected to a microcontroller using the following minimum pins:

```
Pin 1 ground
Pin 2 5V supply
Pin 3 to microcontroller serial output
Pin 4 to microcontroller serial input (if it is required to
      read the state of push-button switches on the LCD module)
```

The default factory configuration of the ILM-216 is 2400 baud, 8 data bits, no parity, and 1 stop bit. Table 14.15 gives a list of the control codes of ILM-216. These codes are summarized below.

Table 14.15: ILM-216 LCD Control Codes

Function	ASCII Code
Null	0
Cursor home	1
Hide cursor	4
Show underline cursor	5
Show blinking cursor	6
Bell	7
Backspace	8
Horizontal tab	9
Smart line feed	10
Vertical tab	11
Clear screen	12
Carriage return	13
Backlight on	14
Backlight off	15
Cursor position	16
Format right-aligned text	18
Escape codes	27

Null: These characters are ignored by the LCD

Cursor home: Moves the cursor to the first character position of the first row

Hide cursor: Hides the cursor so that it is not visible

Show underlined cursor: Shows a non-blinking underlined cursor at the current position

Show blinking cursor: Shows a blinking cursor at the current position

Bell: sends pulses to a buzzer connected to pin 5 of the LCD

Backspace: Moves the cursor back by one space and erases the character in that position

Smart line feed: Moves the cursor down by one line

Vertical tab: Moves the cursor up by one line

Clear screen: Clears the LCD screen

Carriage return: Moves the cursor to the first position on the next row

Backlight on: Turns on the LED backlight

Backlight off: Turns off the LED backlight

Position cursor: Accepts a number from 0 to 31 and moves the cursor to that position where 0 is the first character of the first row and 31 is the last character of the second row. Number 64 should be added to the required cursor position in order to get the actual displayed cursor position. For example, position 80 corresponds to the first character position in the second row (64 + 16 = 80).

Right align text: Accepts a number from 2 to 9 representing the width of an area on the screen in which right-aligned text is to be displayed.

Escape sequences: Escape codes enable the user to define a custom character, to transfer data from the EEPROM, and to read the state of the four push-button switch positions on the LCD module.

Example 14.2

An ILM-216 model serial LCD is connected to bit 0 of PORTB of a PIC microcontroller as shown in Fig. 14.11. Write a PicBasic Pro program to clear the LCD screen and then to display the string 'PIC LCD' in row 1 of the LCD. Wait 1 s for the initialization of the LCD.

Solution 14.2

The required program is given below. The PicBasic command SEROUT is used to send data to the serial LCD.

```
PAUSE 1000              'Wait 1 s for initialization
SEROUT PORTB.0, N2400 (12, "PIC LCD")
```

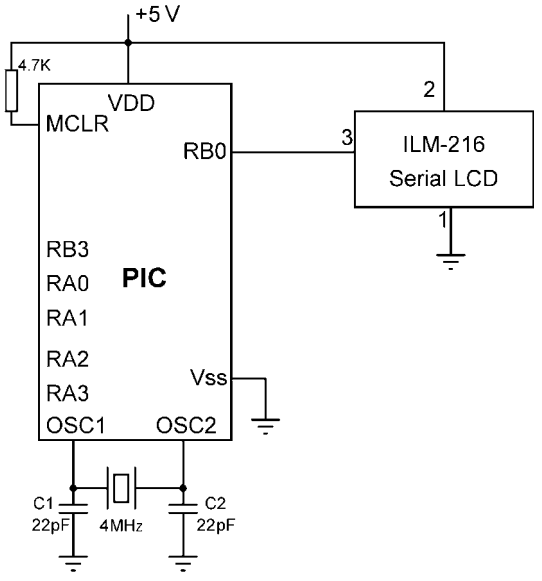


Figure 14.11: Connecting ILM-216 Model LCD to a PIC Microcontroller

14.4 Interrupts

As discussed in previous chapters, interrupts are very useful in many microcontroller applications. An interrupt, as the name suggests, interrupts the normal execution of a program and jumps to a designated address in the program memory called the Interrupt Service Routine (ISR) where a short program is executed. At the end of this program, control is returned to the main program and execution continues from the point it was interrupted.

Interrupts are asynchronous events and it is not known when they may occur. There are basically two types of interrupts: external interrupts and internal interrupts. External interrupts may occur when an external event occurs. For example, when an external signal changes its state. Internal interrupts are usually in the form of timer interrupts and an interrupt may be generated when the timer overflows.

When an interrupt occurs, the PIC microcontroller saves the address of the next instruction on stack and jumps to the ISR which is at address 4 of the program memory. When interrupts are expected from multiple sources, the program should check at the beginning of the ISR to determine the actual source of the interrupt.

PicBasic Pro allows the use of interrupts in programs. The command

```
ON INTERRUPT GOTO Label
```

declares *Label* as the starting point of the ISR. Further interrupts should be disabled by the DISABLE command just before entering the ISR. Also, further interrupts should be enabled

by the ENABLE command after the end of the ISR. The last statement in the ISR should be the RESUME statement which terminates the ISR and returns to the main program.

The structure of the main program and the ISR are as follows:

Main Program

```
ON INTERRUPT GOTO Mylabel
.........................
.........................
.........................
```

Interrupt Service Routine

```
DISABLE
Mylabel:
.........................
.........................
.........................
RESUME
ENABLE
```

14.5 Recommended PicBasic Pro Program Structure

A PicBasic Pro program can be written in many different formats. The author recommends that you use a template similar to the one given in Fig. 14.12 when developing PicBasic Pro programs. As you can see in this figure, the header includes a brief description of the program including the author name, the date, and filename of the program. Comments are used in every line of the program to clarify the actions of the program.

14.6 Using Stepping Motors

Stepping motors are widely used in many microcontroller-based projects where motion is required. This section describes the basic operation of these motors and also shows how they can be used in microcontroller-based projects with PicBasic and PicBasic Pro languages.

Stepping motors are electro-mechanical devices that convert electrical pulses into discrete mechanical movements. A conventional motor has a free running shaft and rotates continuously as long as power is applied to the motor. The shaft of a stepping motor rotates in discrete steps when electrical pulses are applied to it in the correct sequence. The speed of the rotation is related to the time between the input pulses and the length of rotation is directly related to the number of pulses applied. Basically, the motor rotates by an angle defined as the "stepping angle" each time a pulse is applied to the motor. For example, if the stepping angle of a stepping motor is specified as 10º, then each time a pulse is applied the motor will rotate by an angle of 10º and 36 pulses will be required to make a complete 360º rotation.

```
' ************************************************************************
'
'                          LED FLASHING PROGRAM
'                          ====================

'
' This program flashes and LED connected to port RB0 of PORTB.
' The Led is flashed with 1 second intervals.

' Author:       Dogan Ibrahim
' Date:         September, 2005
' File:         LED.PBP
'

' Modifications
' ==========

'
' ************************************************************************

'
' DEFINITIONS

  LED VAR PORTB.0               ' Define RB0 as LED

'
' START OF MAIN PROGRAM

        TRISB = 0               ' Set PORT B pins as outputs

AGAIN:
        LED =1                  ' Turn ON LED
        PAUSE 1000              ' Wait 1 second

        LED = 0                 ' Turn OFF LED
        PAUSE 1000              ' Wait 1 second

        GOTO AGAIN              ' Repeat

        END                     ' End of program
```

Figure 14.12: Recommended PicBasic Pro Program Template

Stepping motors have the following advantages over the conventional motors:

- Motor shaft position can be controlled very accurately using pulses and in open-loop mode.
- Stepping motors can be operated at very low speeds.
- Stepping motors are very reliable since there are no brushes and, as a result, these motors have very long operational lives.

- Stepping motors have full torque at standstill.
- The speed of stepping motors can be controlled easily and accurately.

There are basically two types of stepping motors: unipolar and bipolar. Unipolar motors are easy to control where two windings with common points are used, and a simple 1-of-*n* counter circuit can be used to generate the required stepping sequence. A driver transistor can be used for each winding. One of the most commonly used drive methods is 1 phase full step, also known as the "wave drive", where the motor windings are energized one at a time as shown in Table 14.16. The motor can be driven by using a MOSFET power transistor for each coil winding, as shown in Fig. 14.13. Unipolar motors can also be driven by using integrated circuits, such as the UCN5804B. This chip operates with voltages between 6 and 30 V. It contains a CMOS logic section for the sequencing logic and a high-voltage output section to directly drive a unipolar stepping motor. As shown in Fig. 14.14, the motor is connected directly to the chip and the chip generates the correct sequence of signals to drive the motor.

Table 14.16: One-Phase Full-Step Drive

Step	A	B	C	D
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

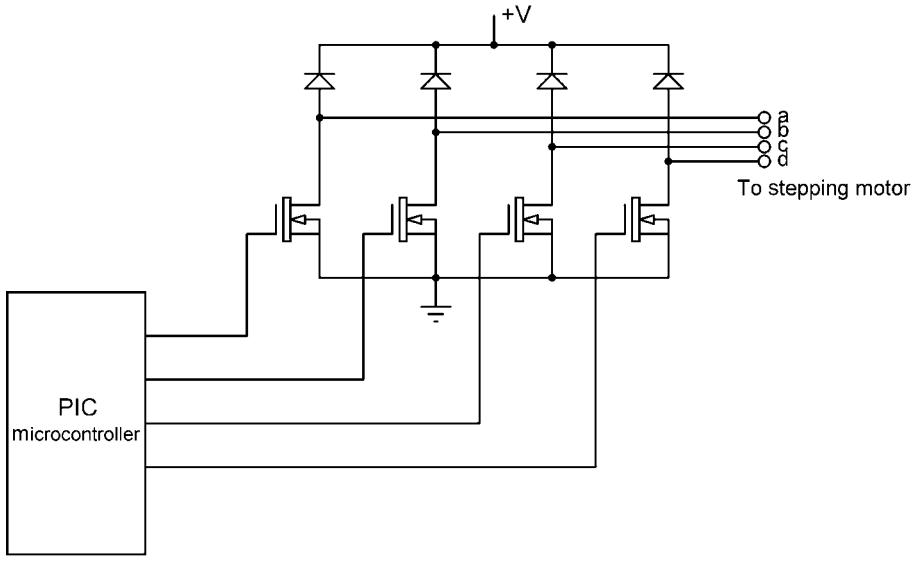


Figure 14.13: Driving a Unipolar Stepping Motor

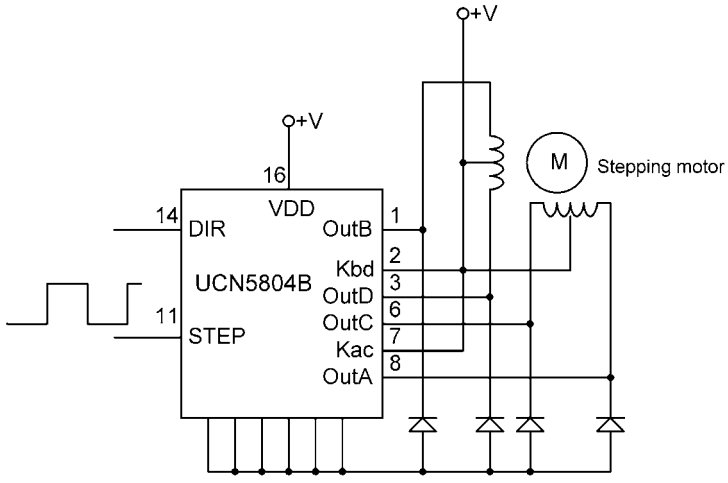


Figure 14.14: Controlling a Unipolar Motor Using a UCN5804B

The DIR input controls the direction of rotation. The motor is rotated by one step each time a pulse is applied to the STEP input.

Bipolar motors generally produce higher torques, but more complex circuits are required to control these motors. The control of bipolar stepping motors is beyond the scope of this book.

Figure 14.15 shows a typical small stepping motor.

Figure 14.15: A Typical Stepping Motor

14.7 Using Servomotors

Servomotors are generally used in radio-controlled toys, such as airplanes, boats, or robots. A servomotor consists of a DC motor with a series of gears attached to it. An internal potentiometer is used with feedback to control the movement of the motor. Normally, the output shaft is limited to 180º of rotation, but it is possible to modify a servomotor so that continuous rotation is obtained.

A servomotor is controlled with a PWM signal. In a modified servomotor, a pulse with a width of 2 ms rotates the motor clockwise at full speed. Similarly, a pulse with a width

Figure 14.16: A Typical Servomotor

of 1 ms rotates the motor anti-clockwise at full speed. Sending a pulse with a width of 1.5 ms stops the motor.

A servomotor requires only three wires to operate: V, ground, and the signal wire where the pulse is applied.

Figure 14.16 shows a typical small servomotor.

This page intentionally left blank

CHAPTER 15

Simple PIC Projects

In this chapter we will put PicBasic to work with one of the more popular PICs, the 16F876. It is a great device to play with because of its flash memory. "Flash" memory means it can be programmed over and over again without having to erase it under ultraviolet light. The 16F876 PIC also has in-circuit programming capability that allows you to reprogram the flash memory without removing it from the circuit.

For this book, I will be using a *bootloader* to program the PIC. A bootloader is a small section of code preprogrammed into the PIC that allows you to download your program via a serial port. There are various versions of a bootloader available, but I'm using MELOADER from microEngineering Labs since they are also the company that produces the PicBasic compilers.

You will see both a PBC and PBPro version for each project. The PBPro requires a special directive line to work with the bootloader. If you program with a standard programmer, then that line can be eliminated.

Let's get started.

15.1 Project #1—Flashing an LED

We'll begin with the easy project of flashing an LED on and off. While this seems simple, even experienced PIC developers often start with this just to make sure things are working properly. Sometimes, within a complex program, I will flash an LED on an unused I/O pin just to give visual feedback that the program is running.

In this example, we'll flash an LED connected to port B pin 0 (RB0). The PIC I/O can individually sink or source 25 milliamps (mA) of current. That is more than enough to drive an LED directly. The software will light the LED for one second, then shut it off for one second, and then loop around to do it again. While this program is simple, getting the LED to flash on and off verifies that you successfully wrote the program in PicBasic, compiled/assembled it, programmed the PIC, and correctly built the PIC circuit. That is a big first step, and why you'll find this first project very rewarding. Figure 15.1 shows the completed circuit board for this project.

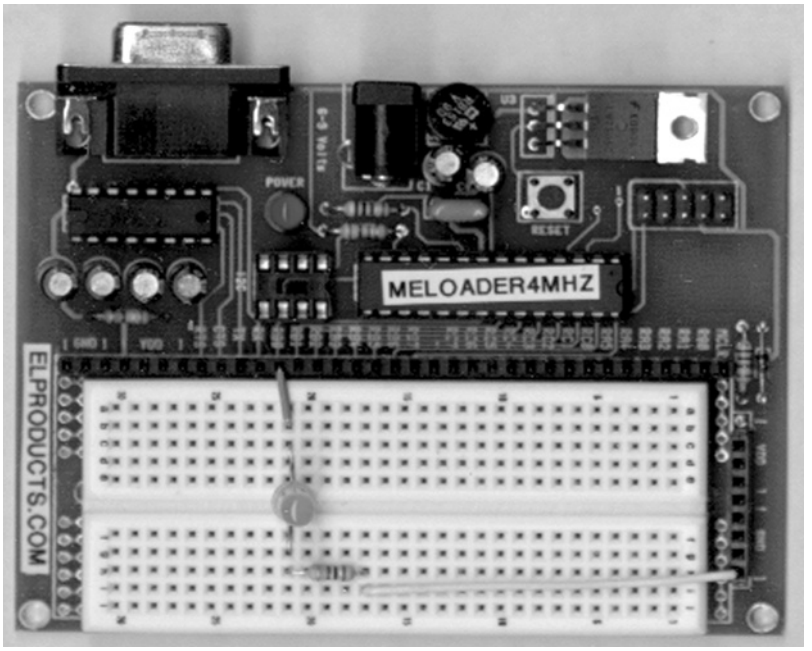


Figure 15.1: View of the Completed Circuit Board for the LED Flasher

Figure 15.2 shows the schematic for this project; it will become the main building block for many of your PicBasic designs. It contains a 4-MHz resonator connected to the PIC OSC1 and OSC2 pins. It also has a 1k pull-up resistor from the MCLR pin to Vdd voltage of 5 volts. These are the only connections a PIC needs to run besides power and ground. That makes getting projects going much easier.

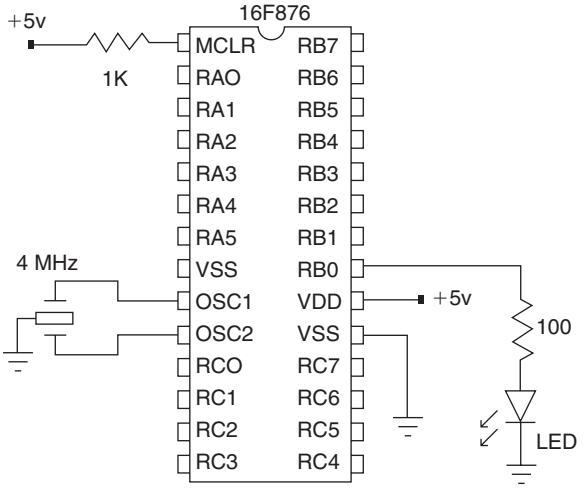


Figure 15.2: Schematic Diagram of the LED Flasher Circuit

15.1.1 PBC Code

The PBC code for this project is written to simply turn the B0 pin of Port B on and off at 1-second intervals. The first part of the program sets up the symbol LED to represent the Port B pin. This isn't necessary, but makes it easier to read than just putting a "0" everywhere the symbol LED is.

Next, Port B is set for which pins are inputs and which are outputs. Notice how PBC uses the DIRS directive and outputs are a "1" and inputs a "0". Finally, we enter the main code section. We use the `High` and `Low` commands to control Port B pin 0. The Pause command is used to create the 1-second delay.

The main program runs in a continuous loop using the `GOTO` command to jump it back to the `main:` label.

```
' ——[ Title ]———————————————————————
'
' File..... proj1PBC.bas
' Format.... PicBasic
' Purpose... PIC16F876 flash LED

' Author.... Chuck Hellebuyck
' Started... June 16, 1999
' Updated...

' ——[ Program Description ]———————————————
' This is a simple program written to flash an LED by turning it
' on for one second and off for one second. The LED should be
' connected to portB pin 0 (16F876 pin 21) with the Cathode to
' ground and the anode at the PIC. A 100 ohm resistor is used in
' series with the LED to limit the current.

'——[ PBC Code ]———————————————————————

Symbol LED = 0       'Rename pin 0 of portb (PIC 16F84 pin 6)
                     'to LED

    DIRS = %00000001 'Setup port b as RB7-RB1 inputs, RB0 as
                     'output

main:                'Label for beginning of main loop

High LED             'Set pin 0 of portb high (5 volts) which turns
                     '  the LED on

    Pause 1000       'Pause 1000 milliseconds (1 second) with
                     'LED on

    Low LED          'Set pin 0 of portb low (0 volts) which
                     'turns the LED off
```

```
Pause 1000          'Pause for 1 second with LED off

goto main           'Jump to the main label and do it all again

END                 'This line is not needed but its safe to
                    'put it here just in case the program gets
                    'lost.
```

15.1.2 PBPro Code

The code for the PBPro compiler is written to simply turn the B0 pin of Port B on and off at one-second intervals in a similar manner to the PBC version.

The first part of the program sets up the symbol LED to represent the Port B pin. This isn't necessary, but makes it easier to read than just putting a "0" everywhere the symbol LED is. Next, Port B is set for which pins are inputs and which are outputs. PBPro uses the TRIS directive and "1" is an input while "0" is an output. (This is a major difference between PBC and PBPro programs.) Finally, we enter the main code section. We use the `High` and `Low` commands to control Port B pin 0. The Pause command is used to create the 1-second delay.

The main program runs in a continuous loop using the `GOTO` command to jump it back to the `main:` label.

```
'———[ Title ]———————————————————————
'
' File..... proj1PRO.bas
' Format.... PicBasic Pro
' Purpose... PIC16F876 flash LED
' Author.... Chuck Hellebuyck
' Started... June 16, 2001
' Updated...

' ———[ Program Description ]———————————————
' This is a simple program written to flash an LED by turning it
' on for one second and off for one second. The LED should be
' connected to port B pin 0 (16F876 pin 21) with the Cathode to
' ground and the anode at the PIC. A 100 ohm resistor is used in
' series with the LED to limit the current.

'———[ PBPro Code ]———————————————————————

Define LOADER_USED 1        'Only required if bootloader used to
                             '  program PIC

symbol LED = 0               'Rename pin 0 of portb (PIC 16F876 pin 21)
                             'to LED
```

```
       TRISB = %11111110   'Setup port b as RB7-RB1 inputs, RB0 as
                           'output

main:                      'Label for beginning of main loop

       High LED            'Set pin 0 of portb high (5 volts) which
                           'turns the LED on

       Pause 1000          'Pause 1000 milliseconds (1 second) with
                           'LED on

       Low LED             'Set pin 0 of portb low (0 volts) which
                           'turns the LED off

       Pause 1000          'Pause for 1 second with LED off

       goto main           'Jump to the main label and do it all
                           'again and again

       END                 'This line is not needed but its safe to
                           'put it here just in case the program gets
                           'lost.
```

15.1.3 Final Thoughts

There are several variations to this same program that will work; you may have a totally different format in mind. What I wanted to do with this was show the basics from which any program can be built. If you want to try other variations of this project, you could modify the values for "pause" to make the LED flash faster or slower. You could also add a second LED and alternate the on and off modes so the LEDs appear to flash back and forth. You could also just build the next project!

15.2 Project #2—Scrolling LEDs

This project expands on the previous project by lighting eight LEDs in a scrolling, "back and forth" motion. The entire Port B I/O is used to drive the eight LEDs. This is a good project that demonstrates how to control all eight LEDs with a single looping routine. The routine uses a `FOR-NEXT` loop to make it happen. The completed circuit is shown in Fig. 15.3 and the schematic diagram is in Fig. 15.4.

15.2.1 PBC Code

The PBC code for this project is very similar to that for the previous project, but expands on it by adding a `FOR-NEXT` loop to the code. It uses the same Port B to control the LEDs, but this time uses all eight ports of Port B to control eight LEDs.

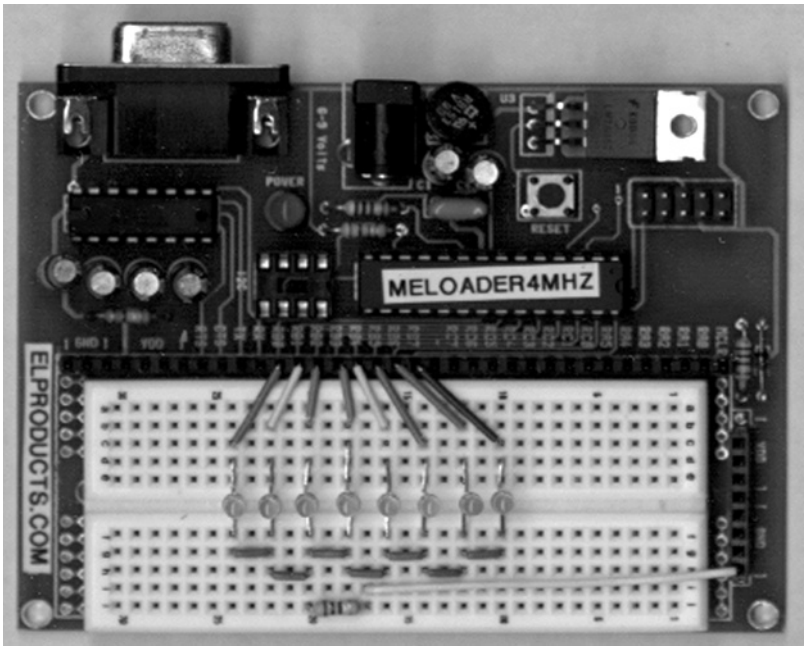


Figure 15.3: View of the Completed Circuit Board for "Scrolling LEDs" Project

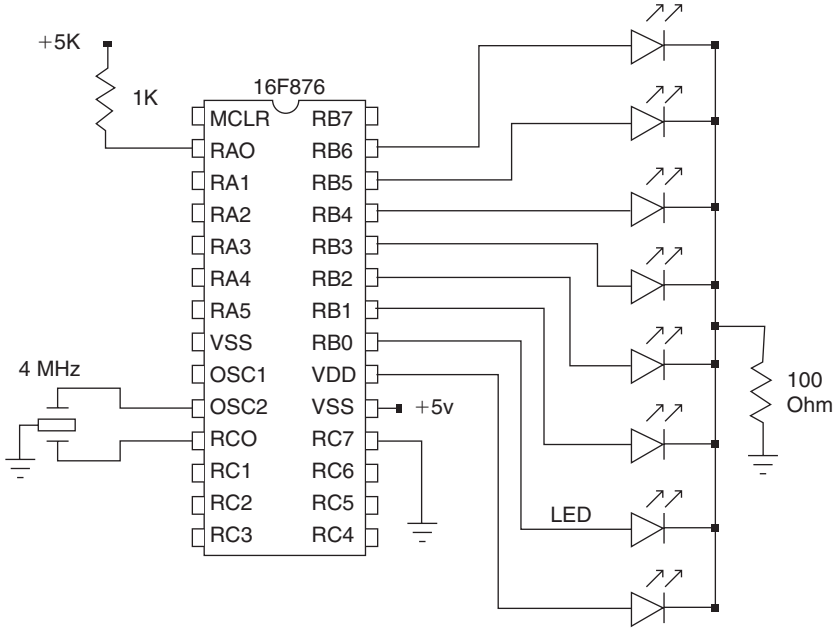


Figure 15.4: Schematic Diagram of the "Scrolling LEDSs" Circuit

In the PBC code, the symbol LED is established to represent the variable B0 rather than just a port pin. Through this variable LED, we will use the `FOR-NEXT` loop to increment through each Port B pin. Then we make all of Port B outputs with the `DIRS` directive and also add the `PINS` directive to establish all of Port B pins as low. This makes sure all the LEDs are off to start.

Next we enter the main loop of code. Within this main loop are two sub-loops created by separate `FOR-NEXT` loops. The first `FOR-NEXT` loop operates on the LED variable by increasing the value by one each time through the loop. This continues until LED is equal to 7, and then the second `FOR-NEXT` loop is entered.

The second `FOR-NEXT` loop actually initializes the LED variable to 7 and then decrements it by 1 on every loop. This continues until the LED variable is equal to 0. The LEDs light up in the opposite direction to the first `FOR-NEXT` loop. The program then has a `GOTO` statement to route the code back to the `main:` label to do it all again. The result is a scrolling light that moves back and forth.

This program shows how a small section of code within a `FOR-NEXT` loop can be used over and over to achieve different output results. The same effect could be achieved with a whole bunch of `High` and `Low` commands repeated for each LED. The effect would be the same, but the amount of program memory would be about five times larger!

```
' ———[ Title ]———————————————————————
'
' File..... proj2PBC.bas
' Format.... PicBasic
' Purpose... PIC16F876 scroll eight LEDs
' Author.... Chuck Hellebuyck
' Started... June 25, 1999
' Updated...

' ———[ Program Description ]———————————————
' This program will scroll a string of LEDs in a back and forth
' motion. Each LED is turned on for 1 second and then turned off.
' The next LED in line is turned on immediately after the previous
' LED is turned off. This continues for all eight LEDs and then
' the direction is reversed. This creates a back and forth motion
' of light. All the LEDs are connected to port B which makes the
' code easier to implement.A single command can control all the
' LEDs at once by setting or clearing the bit associated with each
' LED.

'———[ PBC Code ]———————————————————————

symbol LED = B0 'Rename variable B0 as LED
```

```
      pins = %00000000    'Initiate all port B pins to low

      dirs = %11111111    'Setup port b as all outputs

main:                     'Label for beginning of main loop

' *********** Light LEDs in right direction ************

      for led = 0 to 7    'Loop through all LEDs

      high led            'Set each pin of portb high (5 volts)
                          '  which turns the LED on

      pause 1000          'Pause 1000 milliseconds (1 second) with
                          'LED on

      low led             'Set each pin of portb low (0 volts) which
                          'turns the LED off

      next                'Continue until all 7 have been lit once

' *********** Light LEDs in left direction **************

      for led = 7 to 0    'Loop through all LEDs
      step -1             'backwards

      high led            'Set each pin of portb high (5 volts)
                          'which turns the LED on

      pause 1000          'Pause 1000 milliseconds (1 second) with
                          'LED on

      low led             'Set each pin of portb low (0 volts) which
                          'turns the LED off

      next                'Continue until all 7 have been lit once

' ********** Loop Back to Beginning ***************

      goto main           'Jump to the main label and do it all
                          'again

      END                 'This line is not needed but its safe to
                          'put it here just in case the program gets
                          'lost.
```

15.2.2 PBPro Code

This PBPro code is also similar to that for the first project with an added FOR-NEXT loop. It uses the same Port B to control the LEDs, but this time uses all eight ports of Port B to control eight LEDs.

The B0 is not a predefined byte variable in PBPro as it is in PBC. Instead, we create a byte variable called "LED" using the VAR directive. Through this variable LED, we will use the FOR-NEXT loop to increment through each Port B pin. Next, we make all of Port B outputs with the TRISB directive and also add the Port B directive to establish all of Port B pins as low. This makes sure all the LEDs are off to start. Both of these directives act on the PIC register of the same name. This is more efficient than the way PBC handles this function.

Then we enter the main loop of code. Within this main loop are two sub-loops created by separate FOR-NEXT loops. The first FOR-NEXT loop operates on the LED variable by increasing the value by one each time through the loop. This continues until LED is equal to 7, and then the second FOR-NEXT loop is entered.

The PBPro format allows us to work directly on the Port B register to change individual bits. "PortB.0" is the portname.number format and the line "PortB.0" represents pin 0 of Port B. We can add an equal sign after it and make it equal to 1 or 0. In this program, we take that a step further and add the "LED" variable to the end of it, "PortB.0[LED]". What that does is shift the bit to operate on from 0 to the LED value. For example, if LED equaled 5 then the "PortB.0[LED] = 1 would make the fifth bit of Port B high and turn on that LED.

The second FOR-NEXT loop actually initializes the LED variable to 7 and then decrements it by 1 on every loop. This continues until the LED variable is equal to 0. The PortB.0[LED] line then lights up the LEDs in the opposite direction from the first FOR-NEXT loop. The program then has a GOTO statement to route the code back to the main: label to do it all again.

This program shows how a small section of code within a FOR-NEXT loop can be used over and over to achieve different output results. The same effect could be accomplished with several PortB.0 = 1 and PortB.1 = 1, etc. commands repeated for each LED. As was the case with PBC code, the effect would be the same but the amount of program memory it would take would be about five times larger.

```
' ——[ Title ]——————————————————————
'
' File..... proj2PRO.bas
' Format.... PicBasic Pro
' Purpose... PIC16F876 scroll eight LEDs
' Author.... Chuck Hellebuyck
```

```
' Started... June 25, 1999
' Updated...

' ——[ Program Description ]————————————————
' This program will scroll a string of LEDs in a back and forth
' motion. Each LED is turned on for 1 second and then turned off.
' The next LED in line is turned on immediately after the previous
' LED is turned off. This continues for all eight LEDs and then
' the direction is reversed. This creates a back and forth motion
' of light. All the LEDs are connected to port B which makes the
' code easier to implement. A single command can control all the
' LEDs at once by setting or clearing the bit associated with each
' LED.

'——[ PBPro Code ]————————————————————————

Define LOADER_USED 1          'Only required if bootloader used to
                               'program
'PIC

   LED var Byte                'LED variable setup as byte

     PortB = %00000000         'Initiate all port B pins to low

     Trisb = %00000000         'Setup port b as all outputs

main:                          'Label for beginning of main loop

' *********** Light LEDs in right direction ****************

     for led = 0 to 7          'Loop through all LEDs

     portB.0[LED] = 1          'Set each pin of portb high (5 volts)
                               'which turns the LED on

     pause 1000                'Pause 1000 milliseconds (1 second) with
                               'LED on

     portb.0[LED] = 0          'Set each pin of portb low (0 volts) which
                               'turns the LED off

     next                      'Continue until all 7 have been lit once
```

```
' ************ Light LEDs in left direction ******************

     for led = 7 to 0 step -1        'Loop through all LEDs
                                     'backwards

     portb.0[led] = 1          'Set pin 0 of portb high (5 volts) which
                               'turns the LED on

     pause 1000                'Pause 1000 milliseconds (1 second) with
                               'LED on

     portb.0[led] = 0          'Set pin 0 of portb low (0 volts) which
                               'turns the LED off

     next                      'Continue until all 7 have been lit once

     goto main                 'Jump to the main label and do it all
                               'again and again

     END                       'This line is not needed but its safe to
                               'put it here just in case the program
                               'gets lost.
```

15.2.3 Final Thoughts

You can modify the timing to make the LEDs scroll faster or slower by changing the values for the "pause" command. With some trial and error, you could even get it to operate like the front of that car on the *Night Rider* TV show!

15.3 Project #3—Driving a 7-Segment LED Display

In this project, we drive a 7-segment display like those used in digital clocks and in old calculators. One of the biggest thrills for me, in my early days of fooling with electronics, was when I drove a 7-segment display to indicate 0 to 9 using four discrete integrated circuits. That was a long time ago! This project reduces those four chips to a single PIC.

Driving a 7-segment LED display is really the same as driving seven separate LEDs. Each segment of a 7-segment LED is an individual LED, but they have their cathodes (common cathode) or anodes (common anode) tied together at one pin. We'll use Port B of the PIC 16F876 to drive each segment individually of a "common cathode" 7-segment LED. The program will count from 9 to 0, and then light an LED on the eighth port of Port B that is not used by the 7-segment display. This program demonstrates the use of the `Lookup` command to set the proper Port B pins high to form the numbers 0 through 9. Figure 15.5 shows the completed circuit board for this project and Fig. 15.6 gives the schematic diagram.

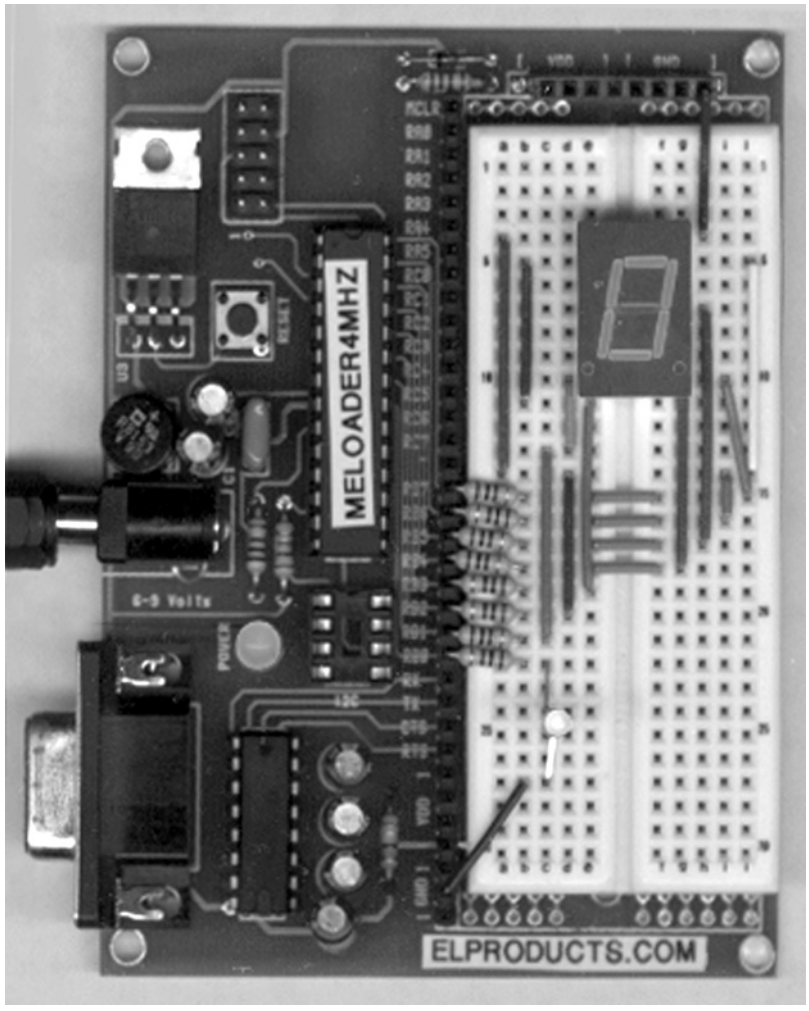


Figure 15.5: View of the Completed 7-Segment LED Display Driving Circuit

The circuitry uses a common cathode 7-segment display. The display pin numbers are not designated because I could not guarantee that the pin-out would match your LED display. You must check the pin-out data sheet for the display you use. The resistors for all the LED segments and the stand-alone LED are 100 ohms. The rest of the circuit is the standard PIC connections that were used in the previous projects.

15.3.1 PBC Code

The PBC version of the project code starts as usual with the variables defined. A general purpose "x" variable is used as the count variable. A second variable "numb1" is used to store the segment arrangements that form the individual numbers.

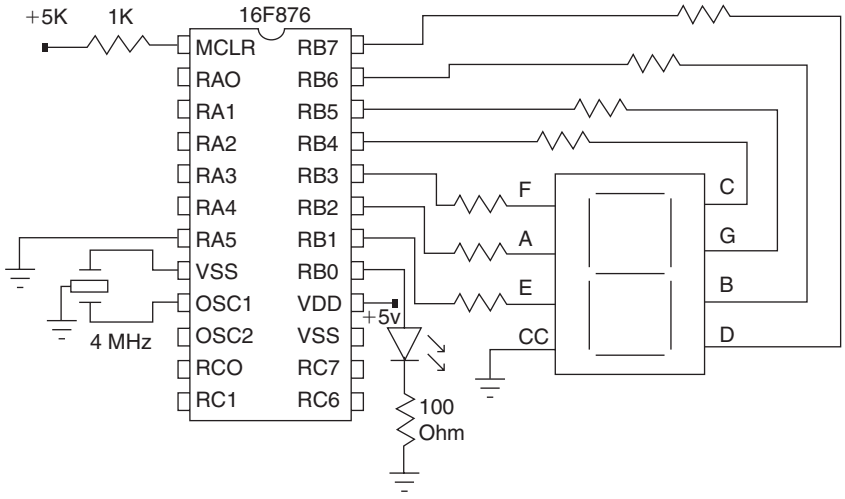


Figure 15.6: Schematic Diagram of the LED Display Driver

Next the I/O is set up and all outputs reset to off. Because PBC defaults to working with Port B, the `DIRS` directive is used. The program then begins the main loop at label `loop:`. This loop is really formed by the `FOR-NEXT` commands. Within the `FOR-NEXT` loop, the variable "x" is decremented from its initial value of 9 to the lowest value of 0. Within that loop is also a command we haven't used in previous projects, the `GOSUB` command.

The `GOSUB` command forces the PIC to put the main `loop` on hold and run a second loop of commands. Those commands are at label `convrt`. The `convrt` loop contains another command we have not used, `LOOKUP`. This `convrt` loop takes the value of variable "x" and, through the `LOOKUP` command, stores the proper Port B I/O set-up in variable "numb1". The `LOOKUP` command constants, contained in the parentheses, are the Port B I/O arrangements required to display the number contained in variable "x" on the LED display.

The `convrt` loop returns back to the main loop using the `RETURN` command. It returns the PIC back to the command that follows the `GOSUB` command. The command line after the `GOSUB` command is where Port B actual drives the LED display with the value in "Numb1". It's done with the simple `PINS` directive. All this continues until variable "x" equals zero. Then the program leaves the `FOR-NEXT` loop and moves to the `light` label. Within that section, the program turns on the standalone LED in the same way Project #1 did earlier. After that, the program loops up to `loop` again and starts the whole countdown over.

```
' ——[ Title ]————————————————————————
'
' File..... proj3PBC.bas
' Format.... PicBasic
```

```
' Purpose... PIC16F876 drives 7-segment LED
' Author.... Chuck Hellebuyck
' Started... August 1,1999
' Updated...

' ---[ Program Description ]-------------------------
'This program drives a common cathode 7-segment LED display to
'countdown from 9 to 0 and then lights a separate LED for one
'second to signify the end of the countdown. The program then
'loops around and counts down again.

'---[ PBC Code ]------------------------------

symbol        x = b0           ' Establish variable X

symbol        numb1 = b1       ' Establish variable numb1

init:

pins = %00000000               'Set all port B pins low
dirs = %11111111               'Set all port B as outputs

loop:
       for x = 9 to 0 step -1  'Countdown from 9 to 0

       gosub convrt            'Go to conversion routine

       pins = numb1            'Set proper I/O pins per convert
                               'routine

       pause 1000              'Keep display the same for 1 second

       next                    'Next number in countdown

light:
       high 0                  'Countdown reached 0 light LED

       pause 1000              'Keep LED lit for 1 second

       low 0                   'LED off

       goto loop               'Do it all again

'* Convert decimal number to proper segment alignment for LED
'display *
```

```
convrt:

lookup x,($DE,$50,$E6,$F4,$78,$BC,$BE,$54,$FE,$FC),numb1 'Match
                                                    'segments to value of x

return                    ' Return to the line after the gosub

       end                         'Add this in case program gets lost.
```

15.3.2 PBPro Code

The PBPro code starts as usual with the variables defined. A general-purpose "x" variable is used as the count variable. A second variable, "numb1," is used to store the arrangement of segments that form the individual numbers.

Next the I/O is set up and all outputs reset to off at the `init` label. The program then begins the main loop at label `loop`. This loop is really formed by the `FOR-NEXT` commands. Within the `FOR-NEXT` loop is where the variable "x" is decremented from its initial value of 9 to the lowest value of 0. As with the PBC code, within that loop is also a command we haven't used in previous projects, the `GOSUB` command.

The `GOSUB` command forces the PIC to put the main `loop` on hold and runs a second loop of commands. Those commands are at label `convrt`. The `convrt` loop contains another command we have not used, `Lookup`. This `convrt` loop takes the value of variable "x" and, through the `Lookup` command, stores the proper Port B I/O set-up in variable "numb1". The `Lookup` command constants, contained in the brackets, are the Port B I/O arrangements required to display the number contained in variable "x" on the LED display.

Notice how the `LOOKUP` command in PBPro uses brackets around the selection list, while PBC earlier used parentheses. This minor code difference will produce an error if you try to convert a PBC program to PBPro.

The `convrt` loop returns back to the main `loop` using the `RETURN` command. It returns the PIC back to the command that follows the `GOSUB` command. The command line after the `GOSUB` command is where Port B actual drives the LED display with the value in "Numb1". It's done by directly modifying the Port B register.

All this continues until variable "x" equals zero. Then the program leaves the `FOR-NEXT` loop and moves to the `light` label. Within that section, the program turns on the stand-alone LED in the same way Project #1 did earlier. After that, the program loops up to `loop` again and starts the whole countdown over.

```
' ——[ Title ]————————————————————————
'
' File..... proj3PRO.bas
```

```
' Format.... PicBasic Pro
' Purpose... PIC16F876 drives 7-segment LED
' Author.... Chuck Hellebuyck
' Started... June 17, 2001
' Updated...

' ---[ Program Description ]-------------------------
'This program drives a common cathode 7-segment LED display to
'countdown from 9 to 0 and then lights a separate LED for one
'second to signify the end of the countdown. The program then
'loops around and counts down again.

Define LOADER_USED 1    'Only required if bootloader used to
                        'program PIC

x var byte         ' General purpose variable

numb1 var byte     ' variable to store the 7-segment I/O setup

init:

portb = %00000000      'Set all port B pins low

trisB = %00000000      'Set all port B as outputs

loop:
     for x = 9 to 0 step -1    'Countdown from 9 to 0

     gosub convrt              'Go to conversion routine

     portb = numb1             'Set proper I/O pins per convert
                               'routine

     pause 1000                'Keep display the same for 1 second

     next                      'Next number in countdown

light:
     high 0                    'Countdown reached 0 light LED

     pause 1000                'Keep LED lit for 1 second

     end

     low 0                     'LED off

     goto loop                 'Do it all again
```

```
'* Convert decimal number to proper segment alignment for LED
'display *

convrt:
lookup x,[$DE,$50,$E6,$F4,$78,$BC,$BE,$54,$FE,$FC],numb1 'Match
                                                    'segments to value of x

     return

     end           'Add this in case program gets lost.
```

15.3.3 Final Thoughts

You could modify the count direction to count up instead of counting down just by changing the `FOR-NEXT` loop values from "9 to 0" to "0 to 9" and remove the step 1. You could also replace the stand-alone LED with a low current buzzer that will beep when the count is finished.

This page intentionally left blank

CHAPTER 16

Moving On with the 16F876

Now we'll discuss additional interesting projects that use more of the PIC's resources. We will continue to work with the 16F876 flash PIC and will be accessing additional I/O directly using PBPro and indirectly using the `PEEK` and `POKE` commands in PBC. After that, we use some of the special I/O of the 16F876 that includes an analog-to-digital (A/D) converter shared with the Port A digital I/O pins. With PBC, the A/D ports can be accessed with a few fairly simple commands and PBPro allows A/D access with a single command. As a final project, we'll drive a servomotor. These are popular with hobbyists and are really quite easy to control with both PBC and PBPro.

16.1 Project #4—Accessing Port A I/O

Most of the PBC commands work directly on the pins of Port B, but what if you want to use another port in the PIC to do something? The answer is the `POKE` and `PEEK` commands. Through these commands you can change the state of the I/O pin from high to low or low to high or make a pin an input and read it for a high level or low level.

Because we are controlling I/O indirectly through `PEEK` and `POKE`, you have to know something about the inner workings of a PIC I/O structure. Every I/O port has two registers associated with it; a direction register called the TRIS (TRISA for port A) register and a data register called by the port name (i.e., PORTA) register.

When the PIC is first powered up, all I/O is put into a high impedance input mode. To make a pin within the port an output requires you to clear the bit associated with it in the port's TRIS register.

Bit 0 in the TRISA register determines the direction for the RA0 pin of PORTA. If the TRISA bit is a "0," then the pin is set to an output. If the TRISA bit is set to a "1," then the pin is set to an input. Therefore, when the PIC is first started up all the TRIS pins are set to "1" meaning all inputs. You can change any single bit to an output or change all of them to outputs.

In this project we will set PORTA pin 0 to an input to read the state of a switch. We will set PORTA pins 1 and 2 to outputs to drive LEDs. If pin 0 is high (switch open), we will light the

LED on pin 1. If pin 0 is low (switch closed), then we will light the LED on pin 2. Seems easy enough, right?

(Note: Port A direction bits are opposite the direction bits PBC uses in the DIRS control of Port B. That's because PBC tried to maintain compatibility with the Basic Stamp. PBC inverts the DIRS to the proper Port B TRISB settings.)

PBPro is much simpler to use than PBC because it can operate directly on Port A the same way it operates on Port B in previous chapter projects. PBPro can access the TRISA register directly with a one-line command (TRISA = %00001111). Although the PBPro command set includes the `PEEK` and `POKE` commands, the compiler manufacturer does not recommend using them.

Once you look at the PBPro code and the PBC code described for this project, you'll understand how PBPro improves on the PBC structure and thus makes programming PICs in Basic easier.

The completed circuit is shown in Fig. 16.1 and its schematic is in Fig. 16.2. As you can see, the circuit is fairly simple. It contains a standard 4-MHz resonator connected to the PIC OSC1 and OSC2 pins. It also has a 1k pull-up resistor from the MCLR pin to the Vdd voltage of 5 volts. In addition, there are the connections required for the I/O control of Port A.

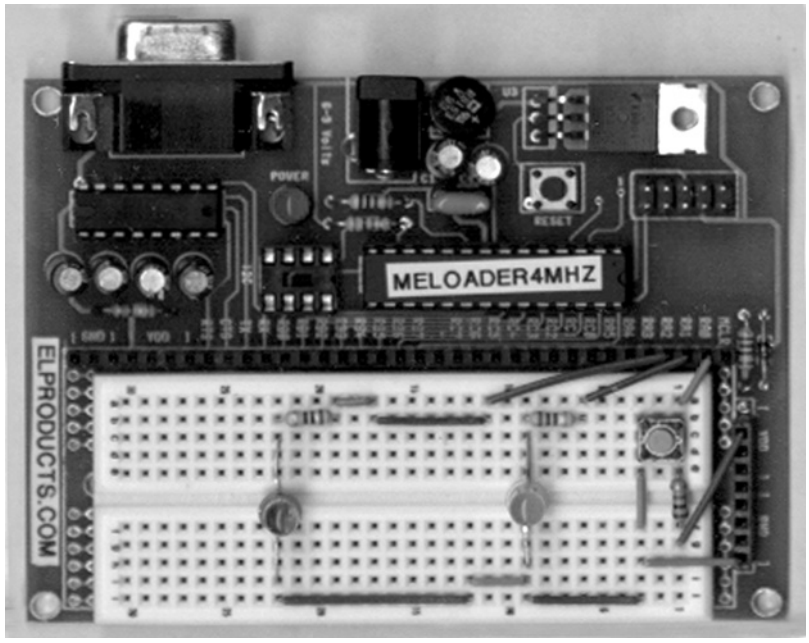


Figure 16.1: Completed Circuit for the Accessing Port A I/O

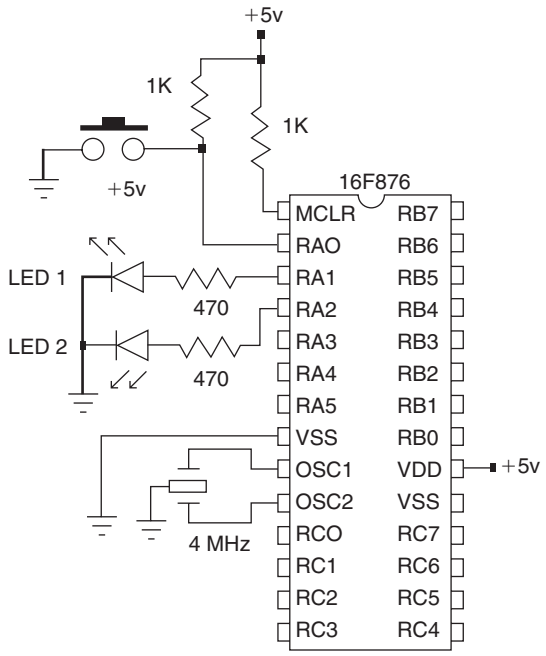


Figure 16.2: Schematic for Circuit Shown in Figure 16.1

Notice that a connection to Port A doesn't look any different than the previous project's connections to Port B. The only difference between Port A and Port B in PBC is the way they are accessed in software. PBPro treats them the same.

The PIC has very powerful I/O circuitry that allows it to drive LEDs directly. A series resistor is required to limit the current. LED1 and LED2 are driven this way through 470-ohm resistors; changing them will change the LED's brightness. Just don't exceed 20 mA to prevent damage to the PIC.

The pushbutton switch is connected directly to Port A pin 0 (RA0). A 1k resistor "pulled-up" to 5 volts is also connected to it. The pushbutton switch is a normally open type that means the switch is open circuit when it's not being pressed. The 1k resistor sets the RA0 threshold to 5V, or high, when the switch is not pressed.

When the switch is pressed, the signal at RA0 goes to ground, indicating a low level. Make sure you use the correct type of switch when you build this or the circuit will not work properly with the software.

16.1.1 PBC Code

The circuit's PBC software, as listed below, first establishes the symbols Port A, TrisA, and ADCON1 by setting them equal to the memory location where they reside in the PIC. This

makes understanding the PBC code much easier if you ever have to refer back to this code at a later date. After you use a PIC for a while, you will find you remember these location addresses but it's still much easier to read the port description.

If you're wondering where those memory location values came from, then look no further than the PIC data sheets or data book. Within those data sheets is a *memory map*. The memory map shows the numeric location of every register in the PIC.

The next set of instructions actually work on those locations using the `POKE` command. The digital direction of Port A needs to be set up by modifying the TrisA register. Notice how each port pin is set to a 0 or a 1 using the "%" binary directive. This makes it easy to tell which pin does what. The left-most digit is port pin 7 and the right-most is port pin 0.

The digital state of each pin is also established by modifying the Port A register in the same way the TrisA register was set up. Making a pin a 0 sets that pin low, a 1 makes it high.

In the line below, pin 1 of Port A is set high while the rest of the port is set low. This presets the LED connected to RA1 pin to the "on" or lit state.

```
poke PortA,     %00000010          ' Set PortA RA1 high to turn
                                   ' on LED1
```

Now I have to throw a little curve in here because I chose to use the PIC16F876. This PIC has an A/D shared with Port A. At power-up, the A/D has control over Port A. For that reason, the command to `POKE` a value of 6 into the ADCON1 register is necessary to set Port A up as a digital I/O port. Reading the data sheet about the A/D register will reveal this, but it's not instantly obvious.

The next section of code at label `Main` is the main loop. It first uses the `PEEK` command to read all the pins of Port A as a byte and store it in RAM byte predefined variable B0. Even though Port A pin RA1 is set to an output, it will read that pin as the state defined by the Port A register. RA2 will be read the same way. The rest of the pins of Port A are read as the voltage applied to those pins.

Since we only care about the state of RA0 where the switch is connected, the next line uses the `&` directive or `AND` directive to perform a logical AND on the byte that was read into RAM byte B0. By comparing it to %00000001, we are essentially erasing all the bits to 0, except for the last bit that is the state of the switch. If the switch is open, this bit will be high. If the switch is closed, this bit will be low. Therefore, if the switch is open B0 will equal one, and if the switch is closed B0 will equal zero.

The next line of code tests for that. By using the `IF-THEN` command we test for a zero value of B0. If B0 equals zero (switch closed), then we want the program to jump to label `LED2`. If B0 does not equal zero, then the program just jumps to the next command.

The next command repeats what was done at the top of the program and "pokes" LED1 on by setting that bit in the PORTA register. After the program completes that, the next command simply jumps the program back to the label `Main` to start the process all over again. On the other hand, if the value of B0 was indeed 0, then at label `LED2` we use the same `POKE` command but this time to set Port A pin RA2 to a 1 and RA1 to a 0 which turns LED1 off and turns LED2 on to indicate the switch was pressed.

Immediately after that, the program jumps back to `Main` and tests the switch again. If the switch is not still being pressed, the program will turn off LED2 and turn LED1 back on with the `PEEK` command.

To keep the LED2 on, you have to keep your finger pressing the switch closed. As soon as you lift your finger off the switch, the LEDs should change back to LED1 on and LED2 off.

```
' ——-[ Title ]————————————————————-
'
' File...... proj4PBC.bas
' Format... PicBasic
' Purpose... Accessing PortA using Peek and Poke
' Author.... Chuck Hellebuyck
' Started... May 1, 2000
' Updated...

' ——-[ Program Description ]———————————————-
' This program demonstrates how to access PortA using Peek and
' Poke in PicBasic.

Symbol      PortA = 5                ' PortA address
Symbol      TrisA = $85              ' PortA data direction
                                     ' register
Symbol      ADCON1 = $9F         ' A/D control register (PortA
                                 ' secondary control)

Init:
           poke TrisA, %00000001      ' Set PortA RA4-RA1 to
                                      ' output, RA0 to input

           poke PortA, %00000010      ' Set PortA RA1 high to
                                      ' turn on LED1

           poke ADCON1 = 6            ' Set PortA to digital
                                      ' I/O

Main:
           ' *** Test the switch state ***
           peek PortA, B0         ' Read all PortA states and
                                  ' store in B0
           B0 = B0 & %00000001      ' Clear all bits in B0 except
                                    ' bit 0
```

```
            if B0 = 0 then led2       ' If switch is closed then
                                      ' jump to
                                      ' the LED2 on routine

            poke PortA, %00000010     ' Turn LED1 on, LED 2 off

            goto Main                 ' Jump to the top of the main
                                      ' loop

LED2:
            '*** Turn LED2 on ***
            poke porta, %00000100   ' LED2 on and LED1 off

            goto Main               ' Jump to the top of the main loop
```

16.1.2 PBPro Code

The PBPro software, as listed below, first goes to work on the special function registers TRISA, PORTA, and ADCON1. PBPro makes it easier on you to access these registers because they don't require you to know the register address in the PIC. PBPro already has that built in.

The work done on those registers is to preset them to defined values by directly equaling them to a value. The digital direction of Port A needs to be set up by modifying the TrisA register. Notice how each port pin is set to a 0 or a 1 using the % binary directive. This makes it easy to tell which pin does what. The left-most digit is port pin 7 and the right-most is port pin 0.

The digital state of each pin is also established by modifying the Port A register in the same way the TrisA register was set up. Making a pin a 0 sets that pin low, a 1 makes it high. In the code below, pin 1 of Port A is set high while the rest of the port is set low. This presets the LED connected to RA1 pin to the on or "lit" state.

Now I have to throw a little curve in here because I chose to use the PIC16F876. This version of the PIC has an A/D shared with Port A. At power-up, the A/D has control over Port A. For that reason, the command to preset a value into the ADCON1 register is necessary to set Port A up as a digital I/O port. Reading the data sheet about the A/D register will reveal this but it's not instantly obvious.

The next section of code at label `Main` is the main loop. The first thing we do is go to work on the pin connected to the switch, RA0. PBPro allows us to use the `IFTHEN` command to test that pin directly using the portname.pinnumber convention. If the state of RA0 is high or a "1" then the switch is open and we should proceed to the next command. The next two commands work directly on Port A register to set RA1 on and RA2 off, thus turning LED1 on and LED2 off. After that, the program loops back to test the switch again. If the switch is closed when we were at the `IF-THEN` command, then RA0 will be low or a "0" and the command then directs the program to jump to label `LED2`.

At LED2 we operate directly on the Port A register to set and clear the LED pins to their proper state. In this case we set LED1 off and LED2 on. After that, the program loops back to `Main` to start all over again testing the switch. To keep LED2 on, you have to keep your finger pressing the switch closed. As soon as you lift your finger off the switch, the LEDs should change back.

If you compare the PBC program to the PBPro program, it becomes clear that not having to `PEEK` and `POKE` makes the PBPro version much easier to use and understand.

```
' ——-[ Title ]————————————————————-
'
' File...... proj4PRO.bas
' Format.... PicBasic Pro
' Purpose... Using Porta on PIC16F876
' Author.... Chuck Hellebuyck
' Started... June 1, 2000
' Updated...

' ——-[ Program Description ]———————————————-
'This program demonstrates how to control PortA with PicBasic Pro.
'Peek and Poke commands are not required because PicBasic Pro has
'control over all registers including I/O registers.
'direct This program will do the same function as proj4PBC.bas but
'in Pro format. PortA RA1 and RA2 will drive LEDs. The RA0 port
'will be an input and read the state of a momentary push button
'switch to determine which LED to light. If switch is pressed LED2
'will light. If switch is not pressed then LED1 will light.

Define LOADER_USED 1      'Only required if bootloader used to
                          'program PIC

Init:
       adcon1 = 6                     ' Set all PortA to digital
                                      'I/O
       trisa = %00000001              ' set PortA RA4-RA1 to
                                      'outputs, RA0 input
       porta = %00000010              ' Set PortA RA1 high to turn
                                      'on LED1

Main:
         ' *** Test the switch state ***
         if portA.0 = 0 then led2     'If switch is pressed then
                                      'jump to LED2 routine

         portA.1 = 1          ' Turn LED1 on
         portA.2 = 0          ' Turn LED2 off
         goto Main            ' Jump to the top of the main loop
```

```
LED2:
            '*** Turn LED2 on
            porta.2 = 1           'LED2 on
      porta.1 = 0           ' LED1 off
            goto Main               ' Jump to the top of the main
                                    ' loop
```

16.1.3 Final Thoughts

It is obviously more complicated to access Port A than Port B in PicBasic. But after you set up variables for the TRISA register and Port A register locations, the program becomes easier to understand. `POKE` and `PEEK` are very useful commands since they allow your PBC program to access any register on the PIC. As you get more familiar with PICs, you will be able to access internal timers, analog-to-digital conversion, internal registers, and many other features available on the various PICs.

The PBPro compiler once again makes programming the PIC a little shorter and a little easier. Although these programs were simple, it was interesting to see how you could modify their function just by pressing a switch. You can build on this basic arrangement to add more switches and functions to control more than just LEDs. We'll touch on some of those in later chapters. But for our next project, let's skip the step where we changed the ADCON1 register to digital I/O and use Port A as an A/D converter.

16.2 Project #5—Analog-to-Digital Conversion

This project uses one of the most useful features of the PIC16F876, the analog-todigital (A/D) converter. Almost everything in the real world is not digital but instead analog. To control something in the real world, or to understand something in the PIC, we have to convert that real-world analog data into the digital form the PIC understands. That is done with an A/D converter. For example, if you have to read a temperature or light levels, you will need both a sensor to convert the measurement into a variable voltage and an A/D converter to change the resulting voltage into a digital value.

In this example, our sensor will be a variable resistor called a *potentiometer* (POT). As we turn the POT's shaft, we want to read the variable resistance from that POT and light some LEDs to show how much we turned it. This could be compared to the volume adjustment you make on a stereo. As you turn the knob for volume, the sound from the stereo gets louder. That's because it is reading the resistance of the POT connected to the knob you turned to adjust the amplifier's output.

This project will use a POT connected to Port A pin RA2. We'll control five LEDs using Port B. The program will have thresholds of A/D values associated with each LED so, as we turn the POT, the LEDs will light in order just like a bar meter on a stereo.

Fortunately, we don't need to know too much about the operation of the A/D circuit; the software just assumes the circuit works, and it does. A/D circuits come in different forms but they all do the same thing—convert an analog voltage into a digital voltage. An A/D register's digital output will have a *resolution* to it. That means it can output an 8-bit digital value, 10-bit digital value, or larger if required. The PIC 16F876 has a 10-bit resolution A/D register, but can also operate as an 8-bit. We will use it as an 8-bit since it's a little easier to understand. Eight bits fit into one byte, and that's much easier to manipulate in code.

Figure 16.3 shows the completed circuit and the schematic is given in Fig. 16.4. The LEDs are connected to Port B with 100-ohm series resistors. Note that the same basic connections are used—OSC1, OSC2, MCLR, 5 volts, and ground—that were used in the previous project.

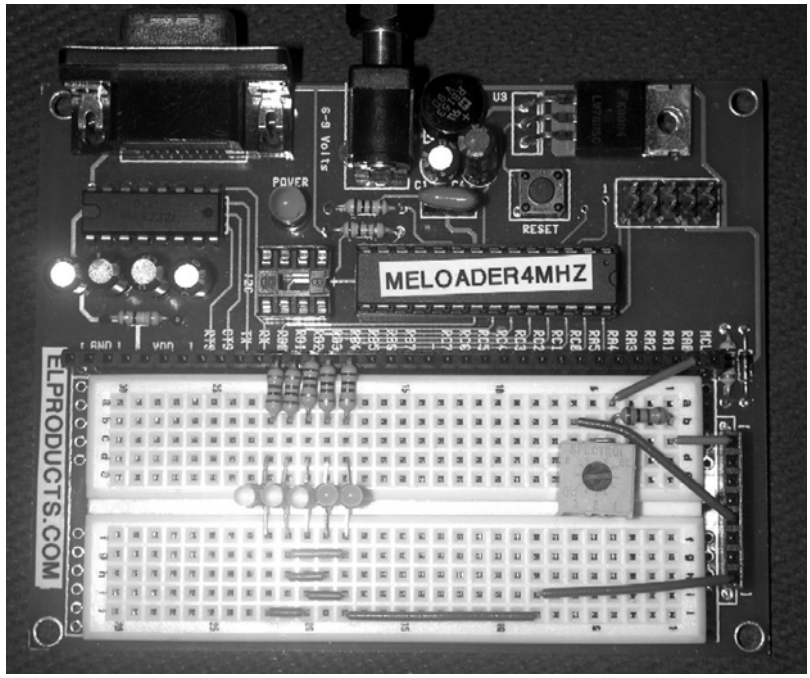


Figure 16.3: Completed Circuit for the A/D Conversion Project

Looking at Fig. 16.4, notice we add the potentiometer to Port A RA2. The 15k pull-up resistor is needed to supply power to the POT. You can vary the values of the POT and pull-up resistor and still get similar results. By adjusting the POT, we are changing the voltage at RA2 through the resistor divider formed between the 15k resistor and POT.

The A/D port cannot handle voltages above 5 volts. Therefore if you need to measure larger voltages, you either have to step it down using resistors or build a voltage conversion circuit using an op amp IC. (But that's a subject for another book and another author!)

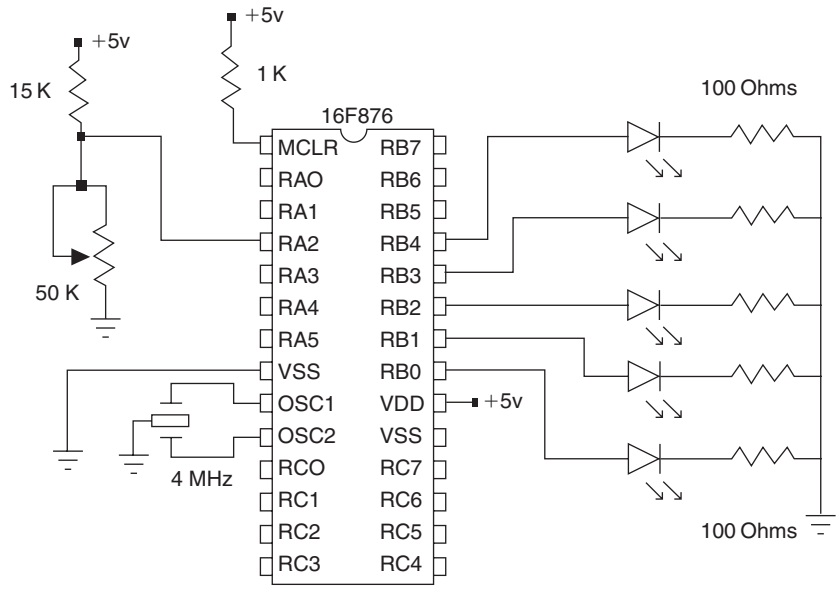


Figure 16.4: Schematic for the Circuit Shown in Figure 16.2

16.2.1 PBC Code

In the PBC program for this circuit, the first part of the PBC code establishes names for the register locations. ADCON0 and ADCON1 are special function registers for controlling the A/D register and ADRESH and ADRESL are where the result of the A/D conversion is stored. If you operate in 10-bit mode, both ADRESH and ADRESL are used to hold the 10-bit result. If you use 8-bit resolution, then you only need one byte to store the result so that register is ADRESH. These register values are once again found in the PIC16F876 data sheet.

After the A/D special function registers are established, the code then initializes the Port B pins at the `Init` label. Using the `PINS` directive and the `DIRS` directive, Port B is set as all outputs and all pins set to zero. Then the program enters the main code at label `Start`.

First step at `Start` is to set Port A to operate as an A/D register. The `POKE` command is used to adjust the TrisA register to all inputs. The `POKE` command is then used to set ADCON1 to hex value $02. This sets all Port A pins connected to the A/D converter to operate as A/D input pins rather than digital I/O pins. This adjustment of ADCON1 also clears the ADFM bit, described in Chapter Four, to set the A/D output to 8-bit mode. This will put the full result in the ADREH register as a byte.

Next we set the A/D converter operating mode. This step selects which A/D port to actually read. We `POKE` the ADCON0 register to control this. If we were reading more than one POT, we would have to do this step over again for that second POT connected to a separate A/D port.

We set ADCON0 to %11010001. Starting from the left, the two most significant bits select the RC internal oscillator as the clock for the A/D circuitry. (Unless you are into extremely accurate A/D measurements, this is the choice to use.) The next three bits are "010" and they select channel 2 or pin RA2 as the A/D port pin to read. Finally, the last bit is a "1" and it turns the A/D converter on. In fact, it starts the A/D conversion.

Since the A/D conversion is started, we have to check when it is completed. We do that at label `loop`. We test it by first using `PEEK` to copy the whole ADCON0 register into predefined RAM byte B0. Then in the next command line we use and `IFTHEN` statement to test the Go/Done bit by using the predefined "Bit2" name associated with RAM byte B0. That's the advantage of using B0 as the register because we have easy access to the bits. If that bit2 value is 1, the conversion is not complete so we loop back to `PEEK` ADCON0 again. If the bit2 value is 0, then we know the A/D conversion is complete and the result is stored in the ADRESH register. We then use the `PEEK` command to read ADRESH and store the result in predefined RAM byte B3.

Now that we have the A/D result from the POT, we need to determine how many LEDs to light up. We do that with a series of `IF-THEN` statements. We test the value four times to see if it is greater than a preset value. This program tests it for a value of 25, 75, 125, and 175. If the value of B3 is less than the predefined value, the next command uses the `Pins` directive to set the correct number of LEDs. If the value of B3 is larger than the predefined value, the `IF-THEN` command jumps the program to the next `IF-THEN` test of B3. If all the `IF-THEN` commands are smaller than the value of B3, the final step simply sets all the LEDs on because the value is greater than all the tested values.

After the LEDs are set, we pause 100 milliseconds to let the LEDs glow. Then the program loops back up to `Loop` to get another reading off the POT. The 100-millisecond delay can be eliminated or reduced if you want to read the POT a little faster.

To understand how the voltage at the POT is compared to the set values in the `IF-THEN` section of code, let's look at the math involved. The A/D converter defaults to the 5-volt Vref as the reference voltage used internally by the A/D converter. It takes the ratio of the voltage at RA2 and the Vref voltage of 5 volts and multiplies it by 255. The result of that calculation is then stored in the ADRESH register. For example, if the voltage at RA2 is 2.30 volts, then the result would be:

(2.30 / 5) * 255 = 117.30.

In our code we turn on the first three LEDs when the A/D result is less than 125.

```
tst3:
       if B0 > 125 then tst4       'If A/D value is between 75 and 125
       pins = %00000111                  ' then light LED0 - LED2
       goto cont                         'continue with the program
```

Therefore the fourth LED will light when the voltage at RA2 increases above the 125 value, which is just about 2.45 volts.

To me this is a great example of how programming in PicBasic is so powerful. You are using simple PEEK and POKE commands to control a high-powered micro-controller the same way someone programming in assembly code would do. But it's so much easier to read and understand. You will see that the A/D converter is very accurate and not too difficult to set up. That's why I don't recommend the POT command because using a PIC with A/D gives more accurate and consistent results.

```
' ——-[ Title ]————————————————————-
'
' File...... proj5pbc.BAS
' Purpose... POT -> 16F876 -> LEDs
' Author.... Chuck Hellebuyck
' Started... May 19, 2001
' Updated...

' ——-[ Program Description ]———————————————-
'
' This Program uses the 16F876 to read a potentiometer (POT) and
' drive LEDs in a bar graph mode as the POT is turned.
'
' RA2        pot connection
' RB4        LED4
' RB3        LED3
' RB2        LED2
' RB1        LED1
' RB0        LED0

' ——-[ Revision History ]———————————————
'
'

' ——-[ Constants ]————————————————————-
'

' A/D Variables and symbols
'
Symbol   ADCON0 = $1F                  ' A/D Configuration
Register 0
Symbol   ADRESH = $1E                  ' A/D Result for 8-bit
mode
Symbol  ADRESL = $9E
Symbol   ADCON1 = $9F                    ' A/D Configuration
Register 1
Symbol   TRISA  = $85                    ' PortA Direction
register
```

```
' ——-[ Variables ]————————————————————-
'
' B0 and B3 are used but predefined in PBC therefore no symbol
' required

' ——-[ Initialization ]————————————————

'
Init:
       pins = $0000                          ' all outputs off to
                                             ' start
       Dirs = %11111111                      ' All of Port B is
                                             ' outputs

' ——-[ Main Code ]————————————————————-
'

'************ A/D Read ************************
'
' PEEK and POKE Commands
'
' Access 16F876 A/D using Peek and Poke

Start:
       poke TRISA, $FF                            'Set PortA to Inputs
       poke ADCON1, $02                ' Set PortA 0-5 to analog
                                       '   inputs, and also
                                           ' Sets result to left
                                           ' justified 8-bit mode
       poke ADCON0, %11010001          ' Set A/D to RC Osc, Channel
                                       ' 2, A/D converter On

loop:

       Peek ADCON0,B0
              Bit2 = 1
              Poke ADCON0,B0                 ' Set ADCON0-Bit2 high
                                             ' to start conversion

test:
       Pause 5
              Peek ADCON0,B0
              If Bit2 = 1 Then test            ' Wait for low on bit-2
                                               ' of ADCON0, conversion
                                               ' finished

         Peek ADRESH,B3                        ' Move HIGH byte of
                                               ' result B3 variable
```

```
'********** Drive LEDs ************************

LEDtst1:
       if B3 > 25 then tst2          'If A/D value is less than 25
       pins = %00000001              ' then light LED0 only
       goto cont                     'continue with the program

tst2:
       if B3 > 75 then tst3          'If A/D value is between 25
                                     ' and 75
       pins = %00000011              ' then light LED0 & LED1
       goto cont                     'continue with the program

tst3:
       if B3 > 125 then tst4         'If A/D value is between 75
                                     ' and 125
       pins = %00000111              ' then light LED0 - LED2
       goto cont                     'continue with the program

tst4:
       if B3 > 175 then tst5         'If A/D value is between 125
                                     ' and 175
       pins = %00001111              ' then light LED0 - LED3
       goto cont                     'continue with the program

tst5:
       pins = %00011111              'A/D value is greater than 175
                                     ' so
                                     ' light all the LEDs 0-4

cont:
       Pause 100                     'wait 1 second
       goto loop

       end
```

16.2.2 PBPro Code

And now for a real demonstration of the simplicity the PBPro compiler offers, we'll use the `ADCIN` command to perform the same function that took several steps in PBC. We'll read the same A/D port RA2 and light the same LEDs based on the same threshold values, but we'll do it in about half the code space PBC required.

We start off with the `DEFINE` statements required by PBPro. The same `Loader_Used` define statement is at the top because I am using a bootloader to program the PIC. Next are a series of `DEFINE` statements dedicated to the ADCIN command. These make it simple to set the output result to eight bits, the clock source to RC, and add a sample time that sets when we check the status of the A/D conversion.

Next, at the `init` label the program establishes a byte variable called "adval." This is where the A/D result will be stored. After that, we set Port B to be all outputs and initialize all LEDs to off by setting all the Port B pins to 0.

After that we enter the `main` code section. We first set Port A to all inputs by modifying the TrisA register. Then we work on the ADCON1 register to make all inputs of Port A work with the A/D register rather than as digital I/O. Then the `ADCIN` command is issued. Within this command we define which A/D port to read (2) and where to put the result (adval). The testing for completion of the A/D conversion is all done by the `ADCIN`command. After the conversion is complete, we can go right to work on the result and test it against the `IF-THEN` statement thresholds the same way the PBC version did.

At each step, we compare "adval" to the predefined values. We light the LEDs by working directly on the Port B register. We light the LEDs according to the value of "adval." If "adval" is less than a predefined value in the `IF-THEN` statement, then the next command is the Port B manipulation. If none of the values in the `IF-THEN` statements are larger than "adval," then the Port B register is changed to light all five LEDs. After this, the program pauses for 100 milliseconds and then jumps back to the `Loop` label to test the A/D register again.

```
' ——-[ Title ]————————————————————
'
' File...... proj5pro.BAS
' Purpose... POT -> 16F876 -> LEDs
' Author.... Chuck Hellebuyck
' Started... May 19, 2001
' Updated...

' ——-[ Program Description ]———————————————————
'
' This Program uses the 16F876 to read a potentiometer (POT) and
' drive LEDs in a bar graph mode as the POT is turned.
'
' RA2       pot connection
' RB4       LED4
' RB3       LED3
' RB2       LED2
' RB1       LED1
' RB0       LED0

' ——-[ Revision History ]———————————————————
'
'
```

```
' ——-[ Constants/Defines ]————————————————-
'
Define LOADER_USED 1        'Only required if bootloader used to
                            'program PIC

' Define ADCIN parameters
Define ADC_BITS        8                ' Set number of bits in result
Define ADC_CLOCK       3                ' Set clock source (3=rc)
Define ADC_SAMPLEUS    50               ' Set sampling time in uS

' ——-[ Variables ]————————————————-
'
adval   var       byte                  ' Create adval to store result

' ——-[ Initialization ]————————————————
'
Init:
      PortB = $00                       ' all outputs off to start
      TrisB = %00000000                 ' All of Port B is outputs

' ——-[ Main Code ]————————————————
'
      TRISA = %11111111                         ' Set PORTA to all input
      ADCON1 = %00000010                ' Set PORTA analog

loop:
      ADCIN 2, adval                    ' Read channel 0 to adval

'********** Drive LEDs *************************

LEDtst1:
      if adval > 25 then tst2                   'If A/D value is less
                                                'than 25
      portb = %00000001                 ' then light LED0 only
      goto cont                         'continue with the program

tst2:
      if adval > 75 then tst3                   'If A/D value is
                                                'between 25 and 75
      portb = %00000011                 ' then light LED0 & LED1
      goto cont                         ' continue with the program

tst3:
      if adval > 125 then tst4          ' If A/D value is between 75
                                        ' and 125
      portb = %00000111                 ' then light LED0 - LED2
      goto cont                         ' continue with the program

tst4:
      if adval > 175 then tst5          'If A/D value is between 125
                                        'and 175
```

```
      portb = %00001111              ' then light LED0 - LED3
      goto cont                      ' continue with the program

tst5:
      portb = %00011111              ' A/D value is greater than 175
                                     ' so
                                     ' light all the LEDs 0-4

cont:
      Pause 100                      ' wait 1 second
      goto loop

     end
```

16.2.3 Final Thoughts

You can use this concept whenever you want to interface digital circuitry to the real analog world. Reading sensors is probably the most common application but not the only one. The code for this project can easily be turned into a subroutine for more complex programs. You can even modify it to read more than one sensor connected to each of the Port A A/D pins. Don't be discouraged if you have difficulty at first understanding the programs and what they're doing; they'll probably be tough to initially understand because we covered so much in this project.

But now let's change direction again and drive something other than an LED. How about driving a servomotor? The next project does just that.

16.3 Project 6—Driving a Servomotor

If you've ever built radio control airplanes or robots, you are probably familiar with *servomotors*. Inside a servomotor is a DC motor with a series of gears attached. The gears drive the output shaft and also control an internal potentiometer. The potentiometer feeds back the output shaft position to the internal control electronics that control the DC motor. The output shaft is limited to 180 degrees of rotation, but some people rework the internals to make the servomotor turn a continuous 360 degrees. The servomotor is controlled by a pulse-width modulated (PWM) signal. The signal has to be between one and two milliseconds. A 1-millisecond wide pulse moves the shaft all the way to the left, and a 2-millisecond wide pulse moves it all the way to the right. Any pulse width in between moves the shaft between the end points in a linear rotation. A 1.5-millisecond pulse would put the shaft at the halfway point.

This project is quite simple. It first moves the shaft to the center position, and then rotates the shaft back and forth between the end points. It's simple, but quite fun to play with. The finished project is shown in Figs 16.5 and 16.6 shows the schematic diagram.

The servomotor only requires three wires: 5 volts, ground, and the signal wire that is connected to RB2. Be sure to use a good power supply. The servomotor draws a lot more power than any of the previous projects in this book. If you are using a regulator to produce

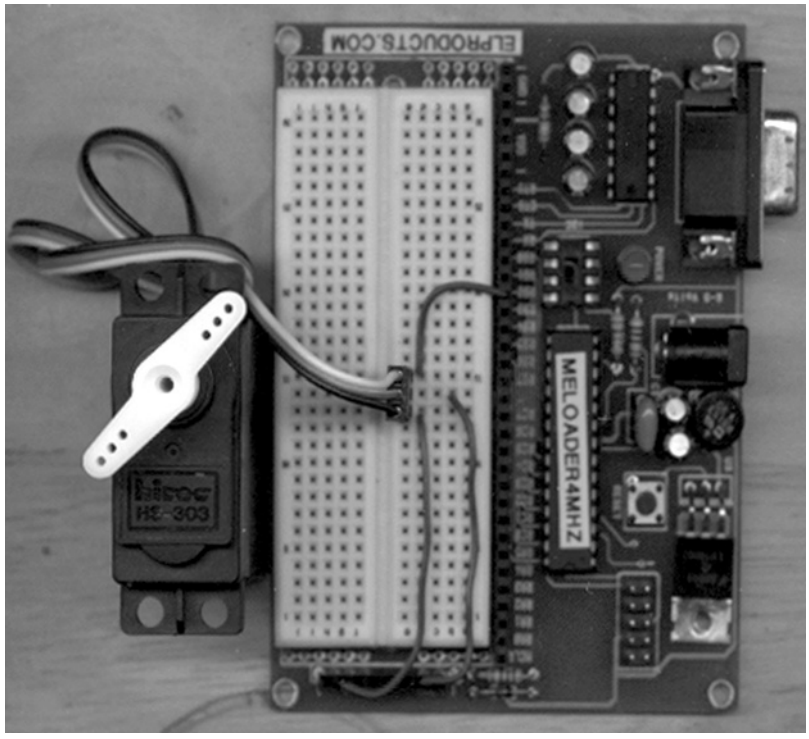


Figure 16.5: Completed Servomotor Control Circuit

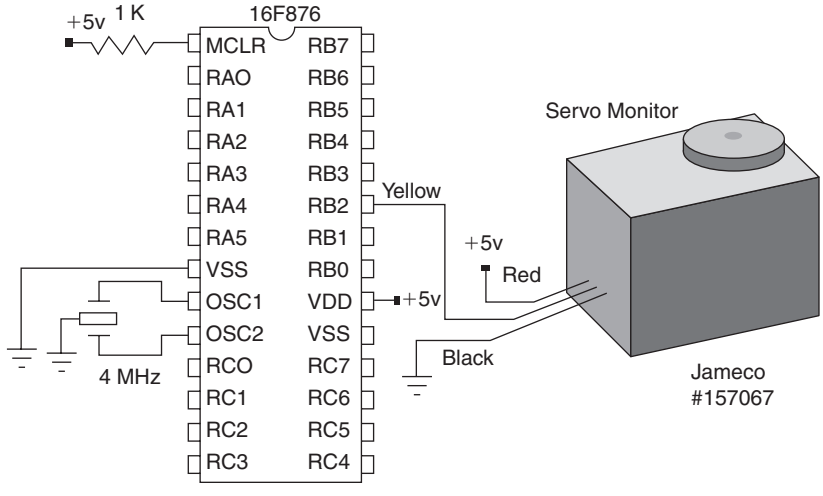


Figure 16.6: Schematic for the Circuit Shown in Figure 16.5

the 5 volts, I suggest you use at least a TO-220 package with proper heat sinking. Many people use separate power sources for the servomotor and the PIC; just make sure your grounds are all connected if you use that method. The rest of the circuit is the standard PIC connections that were used in the previous projects.

16.3.1 PBC Code

There is really not a lot to explain about the PBC code for this project. It's quite simple. The key command in controlling a servo is the `PULSOUT` command. (Why they didn't just spell the full word "pulse" in that command is beyond me! I get more syntax errors from spelling it `PULSEOUT` than any other stupid mistake.) The `PULSOUT` command requires the PIC pin used to output the signal and the *period* constant to tell how long to send the pulse. That is the format of the command covered earlier in the book.

The PBC code does not require any variables to be established since we are using the pre-defined B0 and B2 byte variables. There is nothing to initialize either. Therefore we jump right into the main code loop. We begin at the `Center:` label. This block of code centers the shaft of the servomotor. I have the PIC send the `PULSOUT` signal 100 times by using a `FOR-NEXT` loop using variable B2. I also have a 20-millisecond pause to allow the servomotor to react. Servomotors require a constant signal to maintain proper position. They have tremendous resistance to movement if you have any loads on the shaft, but they will not hold forever and therefore you should send the position signal often. The 20-millisecond delay is the recommended delay between commands for many servomotors.

The PBC `PULSOUT` command has a 10-microsecond resolution. The period constant that is used in the center block of code needs to result in a 1.5-millisecond pulse. Therefore the actual value used in the command is 150 (150 * 10 microseconds = 1500 microseconds, or 1.5 milliseconds).

I deliberately send this command 100 times because I found that gave me enough time to pull the linkage arm off the servo and position it back on the motor shaft at center while the motor was being driven to center.

Once the `Center:` loop is complete, the program moves into the `servo:` label block. Here is where the PIC 16F876 is set to drive the servomotor back and forth between the full counter-clockwise range of positions and the full clockwise range of positions. We do this with two `FOR-NEXT` loops. The first `FOR-NEXT` loop increments variable B0 by 1, starting with 100 and ending with 200. These are the end points for the servomotor. When the `servo:` label is approached in the code, you will know it because the servo will drive immediately from the center position to most counter-clockwise position. Then the servomotor will slowly step the servo shaft to the full clockwise position. Then the next `FOR-NEXT` loop increments the

servomotor in the opposite direction by changing the B0 variable from 200 to 100 in −1 steps. Notice how the `FOR-NEXT` command has the `Step -1` statement following it. This is required to make the `FOR-NEXT` loop count down instead of incrementing B0.

After that second `FOR-NEXT` is complete, we use a `GOTO` statement to jump back up to the `servo:` label and do it all again, thus creating a back and forth movement of the servomotor shaft.

The servomotor linkage could be tied to anything you could think of. Ever see one of those Christmas displays where the elf is pounding a hammer? A servomotor could be controlling that with code similar to this program.

```
' ——-[ Title ]————————————————————-
'
' File..... proj6PBC.bas
' Purpose... PIC 16F876 -> Servo
' Author.... Chuck Hellebuyck
' Started... 15 January 2000
' Updated...

' ——-[ Program Description ]———————————————-
'
'This is a simple program to drive a servo. It initially sets the
'servo to the halfway point of its movement for a short period of
'drives the servo to its full counterclockwise position and then
'to its full
'clockwise position. The back and forth movement is
'repeated continously in a loop.
'
'Connections:
'PIC-PIN              Servo
'RB2                  Control wire (yellow)
'Vdd - 5V             Power Wire (red)
'Vss - Ground         Ground wire (black)
'MCLR - 5V (thru 1k)

' ——-[ Constants/Defines ]———————————————-
'

' ——-[ Variables ]————————————————-
'

' ——-[ Initialization ]————————————————
'

' ——-[ Main Loop ]————————————————
```

```
Center:
       For b2 = 1 to 100      'Send center signal 50 times
       pulsout 2, 150         '150 * 10usec = 1.5 msec
       pause 20               'wait 20 msec
       next                   'if 50 are complete move on

servo:
'———-Clockwise Direction———————————
       for b0 = 100 to 200      'Move from left to right
       pulsout 2,b0                'send position signal
       pause 20                 'wait 20 msec
       next                     'if all positions complete move on
'———-Counter Clockwise Direction———————
       for b0 = 200 to 100 step -1   'move right to left
       pulsout   2,b0                     'send position signal
       pause 20                         'wait 20 msec
       next                             'if all positions complete
move on

       goto servo               'loop to servo label and do it again
```

16.3.2 PBPro Code

For this project, the PBC code and PBPro code are almost identical. In fact, the PBPro code is a few lines longer. (I'll repeat much of what I said in the PBC section above in case you're only reading the PBPro sections.)

This code has to first establish the variables. B0 and B2 are not pre-defined. I could have put one of the `DEFINE` options at the top called:

```
Include "bs1defs.bas"
```

This would establish all the B and W variables the Stamp and PBC use but I don't recommend it. You should always define your variables with the VARdirective. This is a good habit and allows you to name the variables anything you want. That's what I do in the variables section of the code listing. B0 and B2 are established as byte variables. I also have the `DEFINE` statement for the bootloader added which PBPro requires (PBC never requires this).

After that the code is identical to the PBC version. We use the same `PULSOUT` command, which requires the PIC pin used to output the signal and the *period* constant to tell how long to send the pulse.

After establishing the variables we jump right into the `main` code loop. We begin at the `Center:` label. This block of code centers the shaft of the servomotor. I have the PIC send the `PULSOUT` signal 100 times by using a `FOR-NEXT` loop using variable B2. I also have a 20-millisecond pause to allow the servomotor to react. Servomotors require a constant signal

to maintain proper position. They have tremendous resistance to movement if you have any loads on the shaft, but they will not hold forever. Therefore you should send the position signal often. The 20-millisecond delay is the recommended delay between commands for many servomotors.

The PBPro `PULSOUT` command has a 10-microsecond resolution with a 4-MHz crystal/resonator running the PIC. The period constant, which is part of the `PULSOUT` command, used in the center block of code needs to result in a 1.5-millisecond pulse. Therefore, the actual value used in the command is 150 (150 * 10 microseconds = 1500 microseconds, or 1.5 milliseconds). I deliberately send this command 100 times because I found that gave me enough time to pull the linkage arm off the servomotor and position it back on the motor shaft at center while the servomotor was being driven to center.

Once the `Center:` loop is complete, the program moves into the servo: label block. Here is where the PIC 16F876 is set to drive the servomotor back and forth between the full counter-clockwise range of position and the full clockwise range of position. We do this with two `FOR-NEXT` loops.

The first `FOR-NEXT` loop increments variable B0 by 1 starting with 100 and ending with 200. These are the end points for the servomotor. When the servo: label is approached in the code, you will know it because the servomotor will drive immediately from the center position to most counter-clockwise position. Then the servomotor will slowly step the servomotor shaft to the full clockwise position. Then the next `FOR-NEXT` loop increments the servomotor the opposite way by changing the B0 variable from 200 to 100 in -1 steps. Notice how the `FOR-NEXT` command has the Step -1 statement following it. This is required to make the `FOR-NEXT` loop count down instead of incrementing B0.

After that second `FOR-NEXT` is complete we use a `GOTO` statement to jump back up to the `servo:` label and do it all again thus creating a back and forth movement of the servomotor shaft.

```
' ——-[ Title ]————————————————————
'
' File..... proj6PRO.bas
' Purpose... PIC 16F876 -> Servo
' Author.... Chuck Hellebuyck
' Started... 15 January 2000
' Updated...

' ——-[ Program Description ]———————————————————
'
'This is a simple program to drive a servo. It initially sets the
'servo to the halfway point of its movement for a short period of
```

```
'time and then drives the servo to its full counterclockwise
'position and then to its full clockwise position. The back and
'forth movement is repeated continously in a loop.
'
'Connections:
'PIC-PIN               Servo
'RB2                   Control wire (yellow)
'Vdd - 5V              Power Wire (red)
'Vss - Ground          Ground wire (black)
'MCLR - 5V (thru 1k)

' ——-[ Constants/Defines ]————————————————-
'
Define LOADER_USED 1      'Only required if bootloader used to
                          'program PIC

' ——-[ Variables ]————————————————-
'
B2    var      byte           ' Generic Byte
B0  var byte          ' Generic Byte to store Servo position

' ——-[ Initialization ]————————————————
'

' ——-[ Code ]————————————————-

Center:
       For b2 = 1 to 100      'Send center signal 50 times
       pulsout 2, 150         '150 * 10usec = 1.5 msec
       pause 20               'wait 20 msec
       next                   'if 50 are complete move on

servo:
'———-Clockwise Direction————————
       for b0 = 100 to 200    'Move from left to right
       pulsout 2,b0           'send position signal
       pause 20               'wait 20 msec
       next                   'if all positions complete move on
'———-Counter Clockwise Direction————
       for b0 = 200 to 100 step -1    'move right to left
       pulsout 2,b0                   'send position signal
       pause 20                       'wait 20 msec
       next                           'if all positions complete
move on

       goto servo             'loop to servo label and do it again
```

16.3.3 Final Thoughts

You can easily modify the loops to position the servomotor based on a switch input or even use an A/D input to control the servo. The potentiometer circuit in Project #5 could be combined with this code to make a servomotor that follows the movement of the potentiometer. You could use this to control something from a distance with just a few wires connected between the POT and PIC16F876 circuit and the servomotor somewhere else. You could even have the servomotor controlling something inside a climate-controlled chamber while you control the servomotor from outside the chamber.

CHAPTER 17

Communication

In this chapter, we'll use the power of PicBasic to communicate with a PC serial port using RS232 format. We'll also use PicBasic to drive a parallel LCD module (commonly used to display information). The third project of this chapter will combine the first two projects into one by creating a serial LCD module.

Any of these are good learning projects because many of your PicBasic projects will require some way to communicate information—data, time, etc.—to other humans while the PIC is running. It can even be used to display variable data so you can monitor if your program is running correctly.

There's a lot of material in this chapter, so let's get moving!

17.1 Project # 7—Driving an LCD Module

One of the first projects I attempted when I started with PicBasic was to drive an LCD module. LCD modules come in various configurations, but 99% of them use the same interface chip, the Hitachi 44870 LCD character driver. This project is really quite simple but forms the basis for all LCD projects you may build in the future. In this project, we will drive a 2 × 16 LCD module and display the phrase output by so many simple computer programs: "Hello World". This project will show how the PBC and PBPro compilers differ while performing the same task and show the basic structure for controlling an LCD.

The schematic for this project, shown in Fig. 17.1, is simple. The completed project is shown in Fig. 17.2. Note the standard MCLR pull-up resistor and 4-MHz resonator connected to OSC1 and OSC2. The easiest I/O to control with PBC is Port B, so we use that here to control the LCD. The LCD can be controlled in an 8-bit mode or a 4-bit mode, which requires eight I/O or four I/O, respectively. Most people want to save I/O so they use the 4-bit mode. We do the same thing here. Port B is connected to the DB4-DB7 of the LCD module. Through these connections all control characters are sent.

The I/O ports have external pull-up resistors to guarantee a logic high level. The PIC Port B has internal pull-up resistors that you can set, but that will cause confusion for the beginning programmer. Therefore, I went with the external resistors.

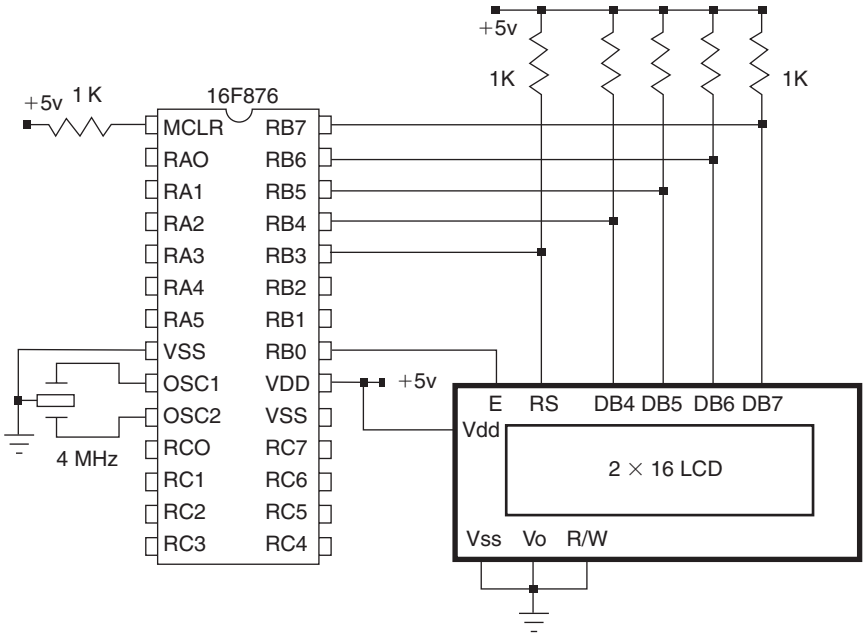


Figure 17.1: Schematic Diagram for the LCD Module Project

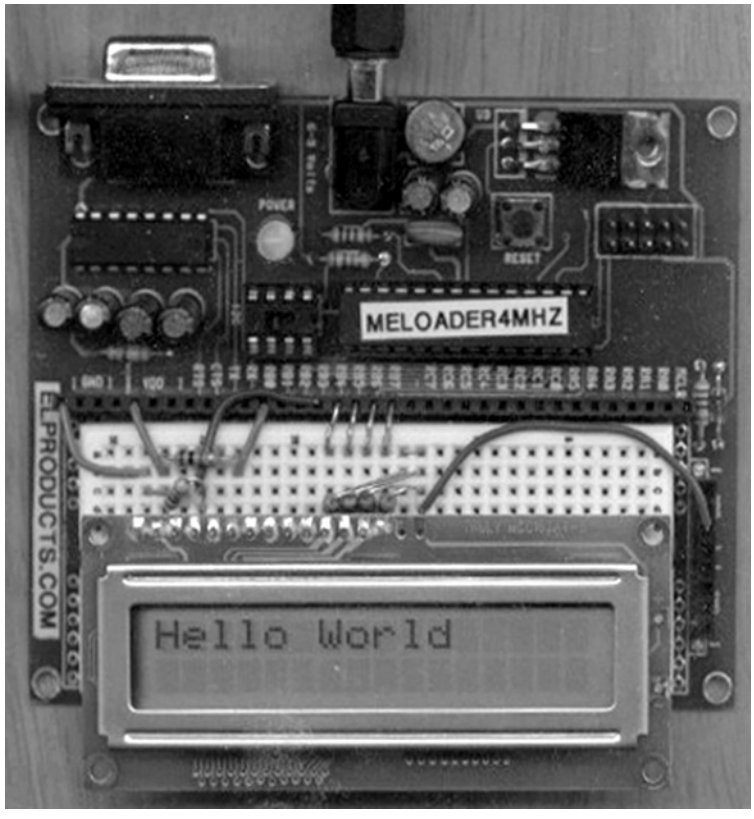


Figure 17.2: View of the Complete LCD Module Circuit Board

The LCD has the R/W pin grounded, which limits it to write-only mode, which is all we plan to do anyway. The Vo pin controls the contrast level of the LCD. We ground it for simplicity, and that sets the LCD to maximum contrast. The RS pin is connected to Port B pin 3. It's used to tell the LCD if a character or LCD command is coming from the PIC. The software section will explain this better. Finally +5 volts and ground are connected to the LCD. The LCD pins are not numbered since LCDs come in different pin layouts. Check the data sheet for your LCD to verify the proper pin numbers.

17.1.1 PBC Code

The PBC code has several important steps. First, several variables and constants for the LCD setup are established. Some of them don't get used in this program, but I set them up so you can expand the program without having to add a bunch of new ones. Next, the LCD is initialized through a whole list of commands. The LCD data sheet will explain several steps to set up the LCD. Those steps are spelled out below.

First, we follow the LCD setup process by sending the proper command three times and then all the commands to establish the LCD setup that matches your LCD. The `PULSOUT` command controls the E, or enable, line of the LCD. The LCD automatically initializes in 8-bit mode. Sending the same command three times and then pulsing the E line for the LCD to recognize it converts the LCD to 4-bit mode.

After that, the LCD is set up with several steps. Notice how almost every other command is a `GOSUB` to the `LCDCMD` subroutine. This subroutine controls the RS line of the LCD. That makes the LCD electronics read the information as a command to control the LCD rather than a character to be displayed on the LCD. It does that by setting the RS pin low prior to jumping to the `WRLCD` subroutine. When the `WRLCD` subroutine is done, it returns to the `LCDCMD` subroutine that sets the RS bit back to high and returns to the section of main code that jumped to the `LCDCMD` subroutine. All these set-up commands are LCD commands that establish the number of rows on the LCD, whether the cursor is on or off, and other minor features.

After the LCD set-up section comes the `main` loop of the code. It starts off by sending commands to clear the LCD. Then it begins to send each character of the phrase "Hello World". Each letter has to be sent separately to the LCD with the same `WRLCD` subroutine mentioned in the LCD setup section above. Because we are writing characters and not commands, we skip the `LCDCMD` subroutine and go right to the `WRLCD` subroutine. The first step is to store the character in the variable "char" and then jump to the `WRLCD` subroutine.

The `WRLCD` subroutine is really the heart of the program. Let's look at it in some detail.

```
pins = pins & %00001000  ' output high nibble
```

This line takes all the PORTB pins and does a logical "and" with the binary value %00001000. What that does is reset every bit to 0 except the fourth bit, which is the RS bit. If that bit is a 1 it stays a 1; if it's a 0 it stays a 0. Therefore, it's left untouched because it can sometimes be set by the `LCDCMD` subroutine that indicates `WRLCD` is sending a command rather than a character to display.

```
b3 = char & %11110000     'store high nibble of char in B3
```

This next line takes the "char" variable and logically "ANDs" it with binary %11110000. Once again we are clearing the lower four bits but leaving the upper four alone. We do this because we need to break the 8-bit "char" byte into two nibbles so we can send it to the LCD. (Remember we are communicating to the LCD using the 4-bit mode to save I/O.) Notice how we operate on "char" but store the result in b3. This leaves "char" unchanged.

```
pins = pins|b3            'combine RS signal with char
```

This line combines the PORTB pins with the four unchanged "char" bits to set PORTB with the proper bit states. We do this with the logical "OR" directive. Because of the way we preset the pins and b3 variable, this command just joins the lower four bits of "pins" with the upper four bits of b3 to produce the desired PORTB state.

```
pause 1                   'wait for data setup
```

We pause briefly to let the data stabilize.

```
PULSOUT E, 100            ' strobe the enable line
```

Now we pulse the E line of the LCD so the LCD module knows to accept the data at its data lines. The pulse is a 1-millisecond wide positive pulse.

```
b3 = char * 16            'Shift low nibble to high nibble
```

This next line is tricky. Since we just sent the upper four bits of the "char" variable, we now have to send the lower four bits. In order to do that, we have to shift the lower four bits to the upper four bits' position. Multiplying any byte by 16 shifts the bits over four places. If we wanted to go the opposite way, we would have divided by 16. Again notice we leave "char" unchanged and store the result in b3.

```
pins = pins & %00001000   'combine RS signal with char
```

Now we reset PORTB I/O back to zeros, except for the RS bit, just like we did at the beginning of this subroutine.

```
pins = pins|b3            ' output low nibble
```

Once again we combine the shifted lower four bits, now in b3, with the lower four bits of PORTB pins data register using the logical "OR" directive.

```
pause 1                   'wait for data setup
```

We pause briefly again to let the data setup.

```
PULSOUT E, 100      'strobe the enable line
```

We pulse the E line so the LCD reads the second set of four bits. The LCD now has the full "char" byte and displays the character on the LCD.

```
RETURN
```

As the final subroutine step, we return back to the area of the program that called the `WRLCD` subroutine. Actually we return to the command just after the `GOSUB WRLCD` command line that sent us here.

That's really all there is to this program. After each character of "Hello World" is sent to the LCD, we just pause for a second and then loop back to do it again.

```
' ——[ Title ]———————————————————————————————-
'
' File..... Proj7PBC.BAS
' Purpose... PIC -> LCD (4-bit interface) using 16F876 and 2x16
' LCD
' Author.... Chuck Hellebuyck
' Started... January 20, 2002
' Updated...

' ——-[ Program Description ]————————————————-
'
'
' PIC16F876 to LCD Port predefined connections:
'
' PIC    LCD               Other Connections
' ——     ——-               ———————————-
' B4            LCD.11
' B5     LCD.12
' B6     LCD.13
' B7     LCD.14
' B3     LCD.4
' B0     LCD.6
' OSC1              Resonator - 4 mhz
' OSC2              Resonator - 4 Mhz
' MCLR              Vdd via 1k resistor
' Vdd               5v
' Vss               Gnd

' ——-[ Revision History ]———————————————————
'
'
```

```
' ——-[ Constants ]————————————————-
'
' LCD control pins
'
symbol E =  0     ' LCD enable pin (1 = enabled)
symbol RS = 3     ' Register Select (1 = char)

' LCD control characters
'
symbol ClrLCD = $01           ' clear the LCD
symbol CrsrHm = $02         ' move cursor to home position
symbol Row2   = $C0           ' 2nd row position of LCD
symbol CrsrLf = $10         ' move cursor left
symbol CrsrRt = $14         ' move cursor right
symbol DispLf = $18         ' shift displayed chars left
symbol DispRt = $1C         ' shift displayed chars right
symbol Digit  = $30         ' Character column code for LCD
' ——-[ Variables ]————————————
'
symbol x = B0             ' General purpose variable
symbol char = B1          ' char sent to LCD
symbol loop1= B2          ' loop counter


' ——-[ Initialization ]————————————————
'
Init:  pins = $0000       ' all outputs off to start

      Dirs = %11111111    ' LCD pins
      PAUSE 215           ' pause for LCD setup

' Initialize the LCD (Hitatchi HD44780 controller)
'
I_LCD:
      pins = %00110000    'set to 8 bit operation
      PULSOUT E,100               'SEND DATA 3 TIMES
      PAUSE 10
      PULSOUT E,100               'SEND DATA 3 TIMES
      PAUSE 10
      PULSOUT E,100               'SEND DATA 3 TIMES
      PAUSE 10
      PINS = %00100000    'SET TO 4 BIT OPERATION
      pause 1
      PULSOUT E,100               'SEND DATA 3 TIMES
      HIGH RS
      CHAR = %00101000    '4 BIT, 2 LINES, 5X7 CHARACTER
      GOSUB LCDCMD
```

```
      CHAR = 8              'Display, cursor and blink off
      GOSUB LCDCMD
      CHAR = 6              'Shift display right
      GOSUB LCDCMD
      CHAR = 1              'clear display and return home
      GOSUB LCDCMD
      CHAR = 15             'display, cursor and blink on
      GOSUB LCDCMD


' ---[ Main Loop ]---------------------------------
'


Start:char = clrlcd       ' Clear LCD
gosub lcdcmd                      ' and position cursor at home
char = CRSRHM                        ' Issue cursor home command
gosub lcdcmd                          ' send cursor home on LCD


'***** Send "Hello World" to first line of LCD ******
       char = "H"                 ' Send "Hello World" one
                                  ' letter
       gosub wrlcd                ' at a time to the LCD
       char = "e"
       gosub wrlcd
       char = "l"
       gosub wrlcd
       char = "l"
       gosub wrlcd
       char = "o"
       gosub wrlcd
       char = " "
       gosub wrlcd
       char = "W"
       gosub wrlcd
       char = "o"
       gosub wrlcd
       char = "r"
       gosub wrlcd
       char = "l"
       gosub wrlcd
       char = "d"
       gosub wrlcd
       Pause 1000                 'Pause long enough to see it

       goto start
```

```
' Send command byte to LCD Subroutine
'
LCDcmd:
       LOW RS                       ' RS low = command
       GOSUB WrLCD                  ' send the byte
       HIGH RS                      ' return to character mode
       RETURN

' Write ASCII char to LCD Subroutine
'
WrLCD:
    pins = pins & %00001000         ' output high nibble
       b3 = char & %11110000        'store high nibble of char in B3
       pins = pins|b3               'combine RS signal with char
       pause 1                      'wait for data setup
       PULSOUT E, 100               ' strobe the enable line
       b3 = char * 16               'Shift low nibble to high nibble
       pins = pins & %00001000      'combine RS signal with char
       pins = pins|b3               ' output low nibble
       pause 1                      'wait for data setup
       PULSOUT E, 100               'strobe the enable line
       RETURN
```

17.1.2 PBPro Code

The PBPro version of the code demonstrates one of PBPro's major advantages over PBC. All of that set-up routine we did in the PBC example, and all of the `LCDCMD` and `WRLCD` subroutine stuff, is done within the PBPro command `LCDOUT`. In fact, we still have control over the LCD set-up, but we do it with `DEFINE` statements rather than a series of `GOSUB` commands.

The first part of the program establishes all the `DEFINE` statements to tell PBPro what port to use for the data port, RS line, and E line. Each define indicates which pin(s) of the port are for communication. We also use `DEFINE` to communicate the 4-bit mode setup and the number of LCD lines. Finally we even have a `DEFINE` to control the time between commands being sent and time delay for data set-up. Some LCDs are picky, so PBPro allows you to adjust the timing of its `LCDOUT` command to work with various LCDs. The `DEFINE` statements I used here should work with most LCDs since I really slowed things down.

```
lcdout $fe, 1          ' Clear LCD
```

The `main` loop is very small but does the same thing as the longer PBC program we previously examined. The `LCDOUT` command line above is sending a command to the LCD to clear the screen. The `LCDOUT` command is sending it as a command signal because of the "$fe" in front of the code "1" for clearing the LCD. All the toggling of the RS line is done by

the LCDOUT command. All you have to remember is to put the "$fe" in front of the code so PBPro knows you meant to send a command.

```
LCDOUT "Hello World"     ' Send "Hello World" to the LCD
```

The next line sends the characters within the quote marks, "Hello World". The LCDOUT command allows you to put the whole phrase between quotes and then it sends it, character by character, to the LCD. The "$fe" is left off because these are characters to display and not command codes. After this line, we pause for 1 second and loop around to do it again.

See how much easier PBPro is to use? The LCDOUT command is priceless in my opinion and is one of the reasons PBPro is worth the extra money it costs. Even though the actual compiled code is not a lot smaller than the PBC code, the listing is much smaller and easier to read.

```
' ——-[ Title ]———————————————————————-
'
' File...... proj7pro.BAS
' Purpose... PIC -> LCD (4-bit interface) using 16F876 and 2x16
' LCD
' Author.... Chuck Hellebuyck
' Started... November 19, 1999
' Updated...


' ——-[ Program Description ]———————————————————-
'
'
' PIC16F876 to LCD Port predefined connections:
'
' PIC          LCD       Other Connections
' ———          ———-             ——————————————-
' B4        LCD.11
' B5        LCD.12
' B6        LCD.13
' B7        LCD.14
' B3        LCD.4
' B0        LCD.6
' OSC1                   Resonator - 4 mhz
' OSC2                   Resonator - 4 Mhz
' MCLR           Vdd via 1k resistor
' Vdd                    5v
' Vss                    Gnd


' ——-[ Revision History ]———————————————————
'
'
```

```
' ——[ Includes / Defines ]————————————————————
'
Define LOADER_USED 1            'Only required if bootloader used to
                                'program PIC

DEFINE LCD_DREG      PORTB        'Define PIC port used for LCD Data
                                  'lines
DEFINE LCD_DBIT      4                 'Define first pin of portb
                                       'connected to LCD DB4
DEFINE LCD_RSREG PORTB            'Define PIC port used for RS line of
                                  'LCD
DEFINE LCD_RSBIT 3                'Define Portb pin used for RS
                                  'connection


DEFINE LCD_EREG      PORTB     'Define PIC prot used for E line of LCD
DEFINE LCD_EBIT      0               'Define PortB pin used for E
                                     'connection
DEFINE LCD_BITS      4               'Define the 4 bit communication
                                     'mode to LCD
DEFINE LCD_LINES 2             'Define using a 2 line LCD
DEFINE LCD_COMMANDUS 2000      'Define delay between sending LCD
                               ' commands
DEFINE LCD_DATAUS 50           'Define delay time between data sent.

' ——-[ Constants ]———————————————————-
'
' ——-[ Variables ]————————————————————-
'
' ——-[ Initialization ]———————————————————-
'
' ——-[ Main Code ]————————————————————-
'


Start:
       lcdout $fe, 1                    ' Clear LCD
       lcdout $fe, 2                    ' Position cursor at home

'***** Send "Hello World" to first line of LCD ******

    LCDOUT "Hello World" ' Send "Hello World" to the LCD
    Pause 1000                  ' Pause for 1 second to see it

    Goto Start                          ' Loop back and do it all
                                        ' again
```

17.1.3 Final Thoughts

As this project illustrates, PBC can do a lot, but PBPro can do more. Imagine if you had several messages you wanted to display on the LCD. With PBC, you would have to spell out each character with a separate command. You could set up a lookup table and send it from a loop. That would save code space, but PBPro would just require a single `LCDOUT` command for each displayed message.

These two programs can be easily modified to fit into any PicBasic program you write to have an LCD display as part of the project.

17.2 Project #8—Serial Communication

This project requires a terminal program running on your PC. The editor software I'm using is called "Codestudio" and has a built-in terminal window. Many of the windows interface software programs available as shareware on the web for using PBC and PBPro have a terminal window. In this project, we will communicate with the PIC16F876 using the serial port of the PC and display info in that terminal window. This whole project is built around the `SERIN` and `SEROUT` commands. Those commands can be used on any PIC pin.

The 16F876 has a dedicated serial buffer built in on pins C6 and C7. We won't use those here since I wanted to demonstrate the versatility of the `SERIN` and `SEROUT` commands. We will use a serial buffer chip that shifts the 0–5 volts PIC signal to the −12 v to +12 v signal the PC serial port likes to see. Some PCs will read the 0–5 volt signal, but it's safer to buffer your circuit from the PC serial port.

This project will send a menu of commands to the PC to be displayed in the terminal window. You will choose from that menu and send back your selection from the terminal window, through the serial port, and back to the PIC circuit. Based on which selection you make at the PC, the PIC will respond with a message or change the state of an LED. From this you will easily see how we could control a PIC-based module directly from a PC serial port.

Figure 17.3 shows the schematic for this project while Fig. 17.4 shows the completed circuit. The standard resonator, MCLR, and power/ground connections are present as in previous projects. Added are the connections to the RS232 level shifter chip and DB9 connector that will hook to a PC "straight-thru" serial cable. We will use that standard cable rather than a null-modem cable. The level shifter chip is available from various sources and all have the same pin-out. The circuit also has an LED connected to Port B pin 0. We use this as a visual indicator that is controlled from the PC. By choosing the proper selection from the menu, you can control the state of the LED.

The PBC code starts off initializing the LED to the off state. (This shouldn't take much explanation by now!) The main part of the program at label `Menu` creates the menu you will

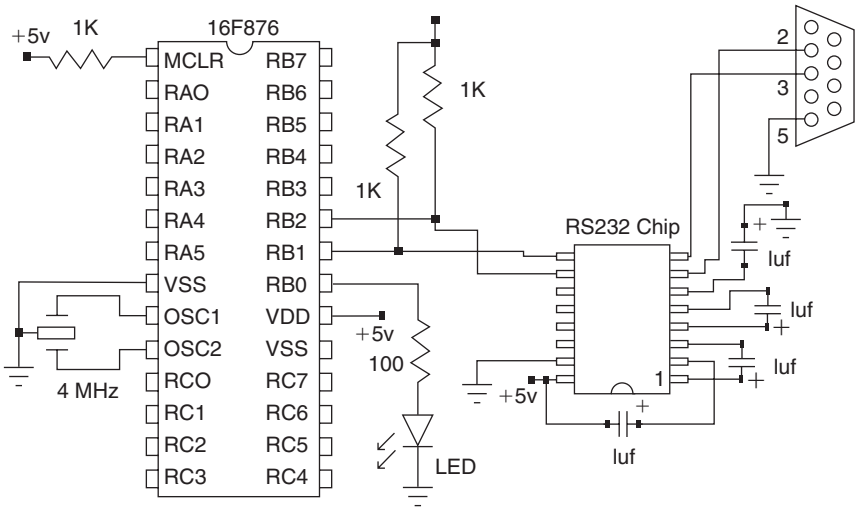


Figure 17.3: Schematic Diagram for the Serial Communications Project

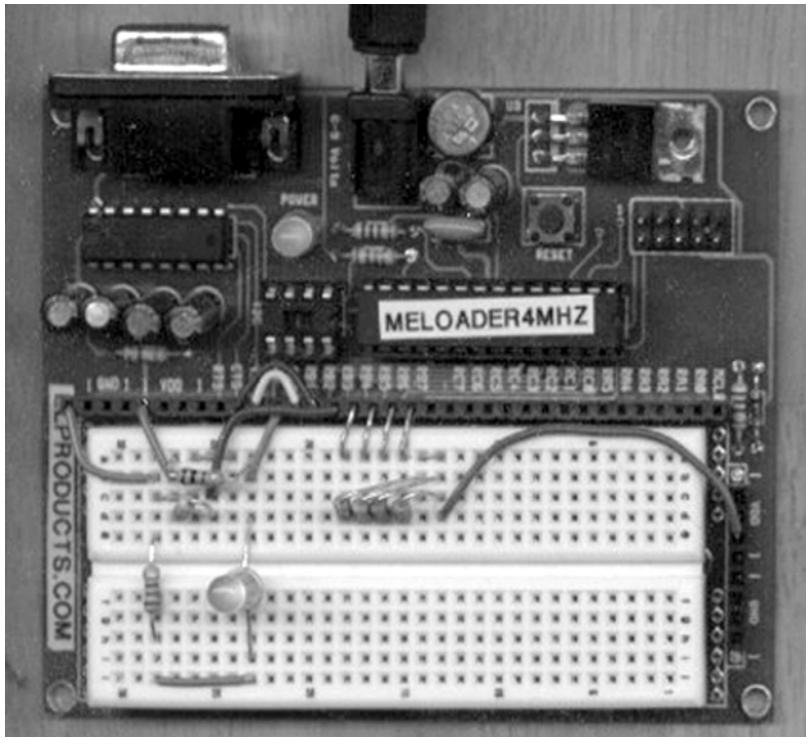


Figure 17.4: Completed Circuit for the Serial Communications Project

see on the PC screen terminal window. It does this by sending each line of the menu serially to the terminal program using the SEROUT command. The program uses 2400 baud true mode to communicate. The terminal program has to be set at the same baud rate. We have to use the T2400 true mode directive so the RS232 level shifter chip sees the proper signal levels.

The information between quotes in the SEROUT command lines are sent as ASCII bytes. The PC should recognize the characters and display the word "Menu" as the first line. For each number displayed as part of the menu, we place a "#" symbol in front of the actual number so the ASCII equivalent is sent instead of just the number. You see, if we just sent the number "1" the ASCII character associated with the value one would be displayed. Instead we want the ASCII character associated with hex $31, which is the ASCII character "1". By putting a "#" in front of the number, PBC will send a $31 instead of a $01.

Each SEROUT command line ends with sending the 10 and 13 characters. These are the ASCII codes for line feed and carriage return. These make each line of SEROUT sent information display as a separate line on the PC. It might help you to look up the ASCII character set to understand all the characters and their code. You can find that in Appendix B.

The next section at label receive is where the PIC waits for the menu choice to be sent by the PC. The choice is sent as an ASCII value. To convert that back into a decimal number we can use, we subtract $30 (30 hex). All ASCII numbers are offset by $30 (0 = $30, 1 = $31, etc). Once we have the menu selection as a numeric value, we can use that to operate on the user's choice. We use that numeric value to branch to one of four locations. They are labeled Zero, One, Two, and Three. At each label we have a different function; some are simply a single line that sends back a "Hello" or "Goodbye".

At label One and label Two, we simply send back serial "Hello" or "Goodbye" and then return to the menu routine to redisplay the menu choices. At label Three, we have the PIC control a LED. Each time you enter choice three at the PC, the LED connected to the PIC Port B pin 0 reverses its state from on to off or off to on. We can tell what state it was in previously by the bit flag "LED." If it's a 0, then we know the LED is off, so go to the routine that turns the LED on. If it's a 1, then we go to the routine that turns the LED off. We also send a message with the state of the LED using the SEROUT command. After that we return to the menu routine to display the menu choices again. That's really all there is to it.

```
' ——-[ Title ]———————————————————-
'
' File...... proj8pbc.BAS
' Purpose... PIC -> PC serial port using 16F876
' Author.... Chuck Hellebuyck
' Started... November 9, 2001
' Updated...
```

```
' ——-[ Program Description ]———————————————
'
'
' PIC16F876 hardware connections
' PIC          External
' ——           ————-
' RB1          Max232(RX)
' RB2          Max232(TX)
' RB0          LED
' MCLR         5v
' Vdd          5v
' Vss          gnd
' Gnd          DB9-pin 5 (gnd)

' ——-[ Revision History ]———————————————
'
'

' ——-[ Constants ]———————————————-
'

' ——-[ Variables ]———————————————-
'
symbol RX = B2              ' Receive byte
symbol LED = bit0           ' LED status bit

' ——-[ Initialization ]———————————————
'
Init:
LED = 0                             'Initialize LED flag to zero
low 0                       'Initialize LED to off

' ——-[ Main Code ]———————————————-
'
Menu:
' ****** [Menu setup on PC screen] ************************
      serout 2, T2400, ("menu", 10, 13)                 'Display menu
                                                        'on PC screen
      serout 2, T2400, (#1, ") ", "send hello", 10, 13)
      serout 2, T2400, (#2, ") ", "send goodbye", 10, 13)
      serout 2, T2400, (#3, ") ", "toggle LED", 10, 13)
```

```
Receive:
' ***** [Receive the menu selection from PC] ***************
        serin 1, T2400, RX       'Receive menu number
        RX = RX - $30                      'Convert ASCII number to
                                           ' decimal
        If RX > 3 then error               'Test for good value
        Branch RX, (zero, one, two, three)       'redirect to menu
                                                 'selection code
Error:
        serout 2, T2400, ("error", 10, 13, "Try again", 10, 13)
        goto menu

Zero:
' ***** [ Code for zero value ] ****************************
        goto menu                 'Return to menu, zero is not a
                                  'valid selection

One:
' ***** [Code for selection 1] *************************
        serout 2, T2400, ("Hello",13,10,13)    'Send "Hello" back
                                               'to PC
        goto menu                           'Return to main menu
                                            'routine

Two:
'***** [Code for selection 2] ************************
        serout 2, T2400, ("Goodbye",13,10,13)  'Send "Goodbye" to
                                               'PC
        goto menu                 'Return to Menu routine

Three:
'***** [Code for selection 3] **********************
        if LED = 1 then off         'If LED bit =1 then goto
                                    'off
        high 0                      'Turn LED on
        led = 1                                 'Set LED bit to 1
        serout 2, T2400, ("LED ON",13,10,13)  'Send LED status to
                                               'PC
        goto menu                   'Return to main menu

Off:
        low 0                      'Turn LED off
        led = 0                                  'Clear LED bit to 0
        serout 2, T2400, ("LED OFF",13,10,13)  'Send LED status to
                                               'PC
        goto menu                   'Return to main menu

Goto menu
```

17.2.1 PBPro Code

The PBPro code starts off initializing the LED to the off state. PBPro also requires a `DEFINE` to establish the `SEROUT` mode definitions. PBPro doesn't automatically recognize the T2400 mode in the `SEROUT` command without the MODEDEFS.BAS `INCLUDE`. PBPro also has to set up the TRISB and PORTB registers so the proper state of the pins is established.

The main part of the program at label `Menu` sets up the menu you will see on the PC screen terminal window. It does this by sending each line of the menu serially to the terminal program using the `SEROUT` command. The program uses 2400-baud true mode to communicate. The terminal program has to be set up at the same baud rate. We have to use the T2400 true mode directive so the RS232 level shifter chip sees the proper signal levels.

The information between quotes in the `SEROUT` command lines is sent as ASCII bytes. The PC should recognize the characters and display the word "Menu" as the first line. For each number displayed as part of the menu, we place a "#" symbol in front of the actual number so the ASCII equivalent is sent instead of just the number. If we just sent the number "1" the ASCII character associated with the value one would be displayed. Instead we want the ASCII character associated with hex $31, which is the ASCII character "1". By putting a "#" in front of the number, PBPro will send a $31 instead of a $01.

Each `SEROUT` command line ends with sending the 10 and 13 characters. These are the ASCII codes for line feed and carriage return. These make each line of `SEROUT` send information and display as a separate line on the PC. Again, it might help you to look up the ASCII character set to understand all the characters and their code. You can find it in Appendix B.

The next section label `receive` is where the PIC waits for the menu choice to be sent by the PC. The choice is sent as an ASCII value. To convert that back into a decimal number we can use, we subtract $30 (30 hex). All numbers are offset by $30 (0 = $30, 1 = $31, etc). Once we have the menu selection as a numeric value, we can use that to operate on the user's choice. We use that numeric value to branch to one of four locations. They are labeled `Zero`, `One`, `Two`, and `Three`. At each label, we have a different function. Some are simply a single line that sends back a "Hello" or "Goodbye".

At label `One` and label `Two` we simply send back serial "Hello" or "Goodbye" and then return to the menu routine to redisplay the menu choices. At label `Three`, we have the PIC control an LED. Each time you enter choice three at the PC, the LED connected to the PIC Port B pin 0 reverses its state from on to off, or off to on. We can tell what state it was in previously by the bit flag "LED." If it's a 0, then we know the LED is off so go to the routine that turns the LED on. If it's a 1, then we go to the routine that turns the LED off. We also send a message with the state of the LED using the `SEROUT` command. After that we return to the menu routine to display the menu choices again.

```
' ——-[ Title ]———————————————————————-
'
' File..... proj8pro.BAS
' Purpose... PIC -> PC serial port using 16F876
' Author.... Chuck Hellebuyck
' Started... November 9, 2001
' Updated...

' ——-[ Program Description ]——————————————————-
'
'
' PIC16F876 hardware connections
' PIC      External
' ——     ——————-
' RB1      Max232(RX)
' RB2      Max232(TX)
' RB0      LED
' MCLR     5v
' Vdd      5v
' Vss      gnd
' Gnd      DB9-pin 5 (gnd)

' ——-[ Revision History ]———————————————————
'
'
' ——-[ Includes/Defines ]———————————————————-
'
include "modedefs.bas"     'include serout defines
define loader_used 1            'Used for bootloader only

' ——-[ Constants ]——————————————————-
'

' ——-[ Variables ]——————————————————-
'
RX var byte          ' Receive byte
LED var bit          ' LED status bit

' ——-[ Initialization ]———————————————————
'
Init:
TRISB = %00000010    'All port b output except pin 1 (RX) is
                     'input
PORTB = %00000000    'Initialize PortB to all zeros and LED
                     'to off

LED = 0              'Initialize LED flag to 0
```

```
' ——-[ Main Code ]————————————————————-
'
Menu:
' ****** [Menu setup on PC screen] ************************
        serout 2, T2400, ["menu", 10, 13]            'Display menu
                                                     'on PC screen
        serout 2, T2400, [#1, ") ", "send hello", 10, 13]
        serout 2, T2400, [#2, ") ", "send goodbye", 10, 13]
        serout 2, T2400, [#3, ") ", "toggle LED", 10, 13]

Receive:
' ***** [Receive the menu selection from PC] ***************
              serin 1, T2400, RX               'Receive menu
                                               'number
        RX = RX - $30                          'Convert ASCII
                                               'number to decimal
              If RX > 3 then Error                     ' Test for
                                                       'good value
        Branch RX, [zero, one, two, three]     'redirect to menu
                                               'selection code

Error:
        serout 2, T2400, ["error", 10, 13, "Try again", 10, 13]
        goto menu

Zero:
'***** [ Code for zero value ] ****************************
       goto menu                'Return to menu, zero is not a
                                'valid selection

One:
'***** [Code for selection 1] ************************
        serout 2, T2400, ["Hello", 10, 13]     'Send "Hello" back
                                               'to PC
        goto menu                         'Return to main menu
                                          'routine

Two:
'***** [Code for selection 2] ************************
        serout 2, T2400, ["Goodbye", 10, 13]   'Send "Goodbye" to
                                               'PC
        goto menu                         'Return to Menu routine

Three:
'***** [Code for selection 3] *********************
        if LED = 1 then LEDoff                 'If LED bit =1 then
                                               'goto off
```

```
        portb.0 = 1                          'Turn LED on
        led = 1                                 'Set LED bit to 1
        serout 2, T2400, ["LED ON", 10, 13]     'Send LED status to
                                                'PC
        goto menu                             'Return to main menu

LEDOff:
       portb.0 = 0                            'Turn LED off
       led = 0                                   'Clear LED bit to 0
       serout 2, T2400, ["LED OFF", 10, 13]      'Send LED status to
                                                 'PC
       goto menu                             'Return to main menu

Goto menu
```

17.2.2 Final Thoughts

As you can see, the PBC and PBPro programs are very similar for this project. You can easily expand the menu to include more choices. You can also have the menu choices do a lot more than light a LED or send back a message; for example, you could have a series of control circuits tied to the PIC pins and control them through this same setup. Or how about a robot arm in a "cold chamber"? You could have that robot arm controlled by a PIC and that PIC controlled through a single serial connection to a PC in a warm lab. Interesting?

17.3 Project #9—Driving an LCD with a Single Serial Connection

This project uses the same hardware connections as Projects #7 and #8, but the software is unique. The idea is to receive information through the serial port and then display the information on the LCD. This allows a serial port from any PC or another PIC to drive the LCD with a single serial connection.

The software works on simple principles. The PIC waits for three bytes of data. The first is the "row" byte. It indicates on which row the information should be displayed.

The second byte is the "location" byte. It indicates at which position (column) within the row the information should start. The third and final byte is the character code of the information to be displayed. This would be the code for a letter ("a") or number ("1"). This is defined by the display character generator, but is typically an ASCII value similar to what we did in Project #8 to set up the PC menu.

The third byte has an alter ego, though; it can also be used to control the LCD via custom code commands. To enter this command control mode, we set the "row" byte equal to 0. When the module receives the 0 value for the "row" byte, then the module knows that the character code is a command code and not a character to be displayed. A separate action routine will occur based on that command code. Clearing the whole display would be one such command code.

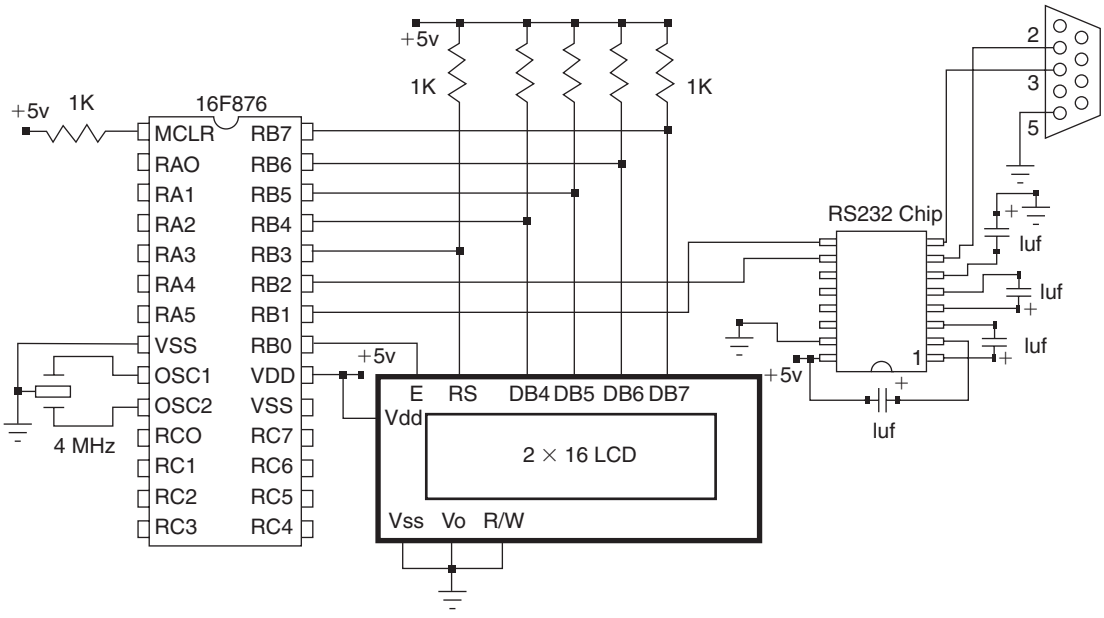


Figure 17.5: Schematic Diagram for Driving a LCD Through a Serial Connection

The schematic for this project is shown in Fig. 17.5; it combines the schematics of Projects #7 and #8 into one schematic. It's not too different from those projects, so there is not much to explain here. We use the same resonator and MCLR pull-up resistor as every other project. The serial connection is the same as Project #8 and the LCD connection is the same as Project #7. The completed circuit is shown in Fig. 17.6.

17.3.1 PBC code

While the circuit for this project may be simple, the PBCcode for it may initially look confusing. Let's break it down.

The labels `Init` and `I_LCD` initialize the LCD the same way Project #7 did. We use the 2 × 16 module because those are very common LCD modules. You can reconfigure the code to work with any LCD size by first modifying the initialization section.

The next label, `start`, is where this project code really starts. The main line is the `SERIN` line that waits for the row, location, and value bytes. The program will sit here forever if it doesn't receive any information. When information is received, it first tests the row byte and location byte to see if "Row" is not 0 and "Location" is 0, indicating the cursor should not be moved from its existing position and to write the character in the value byte at that location. It does that by jumping to the `Display` label.

If the location byte is not 0, independent of the value of "Row," then the next command that is run which is a `BRANCH` instruction. The `BRANCH` instruction jumps the program to the proper

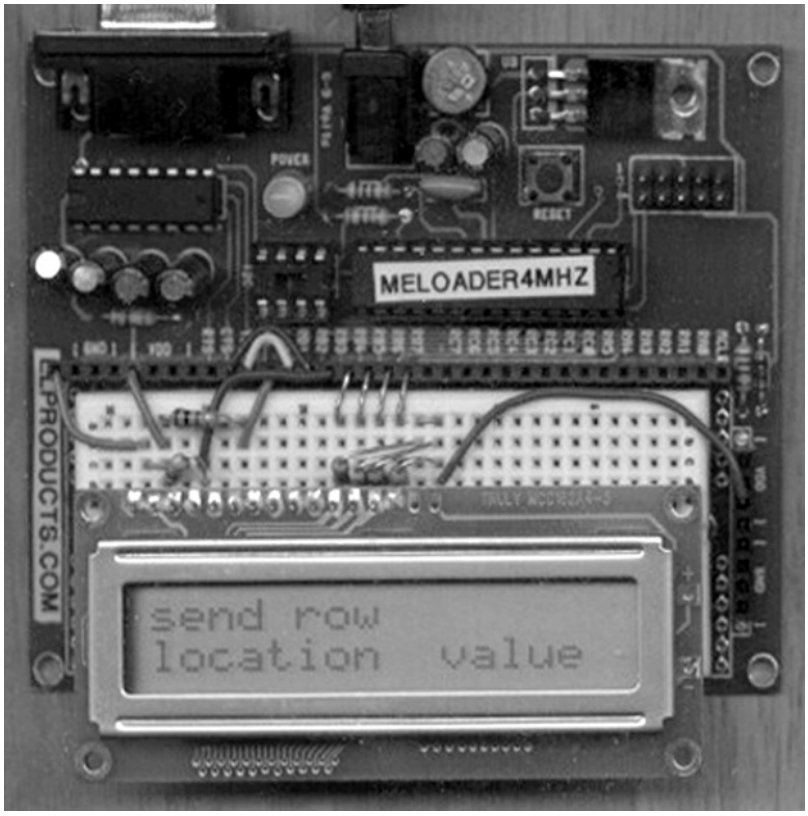


Figure 17.6: Finished Circuit Board for the Circuit in Figure 17.5

label based on the value of the row byte. If the row byte is 0, then the program jumps to the label `Command`. The code at the `Command` label will send a LCD command based on the value byte received (I'll explain this in more detail later).

If "Row" does not equal 0, then the `BRANCH` command redirects the program to the `R1` or `R2` labels. Let's try to follow that path. At labels `R1` and `R2`, the program converts the location byte received into the corresponding LCD code to position the LCD cursor at the proper row and column using the `LOOKUP` command. We have to subtract 1 from the location because the `LOOKUP` command starts at 0 instead of 1.

After we have the proper position code from the `LOOKUP` command, the program then jumps to the `LCDcmd` subroutine to send the special LCD position code to the LCD and move the cursor on the LCD. When the subroutine is done, the program jumps back to the command after the `GOSUB LCDcmd` line. That command is a jump to the `Display` label.

At the `Display` label, the program first stores the serial-received "Value" byte in the "Char" variable. Then the program jumps to the subroutine `WrLCD` to send that byte to the LCD character generator that actually displays the character on the LCD. The program then returns

back to the Display label routine, which then sends the program back to the top to receive a new set of information at the Start label. (See how we first positioned the cursor based on the row and location bytes and then sent the character code to be displayed at that position?)

If the "Row" byte is 0, the BRANCH command under the Start label redirects the program to the Command label. We don't stop at the R1 and R2 labels to get a LCD command code because the "Value" byte received serially should have the LCD command byte in it. All we have to do is convert that value byte to the proper LCD command code based on the table above the Command label. First we take the "Value" byte and convert it into a decimal number by subtracting hex $30. Then we use the LOOKUP command to change the "Value" byte into the proper LCD command code. That command code is stored in the "Char" variable.

The next command line jumps the program to the LCDcmd subroutine where the command code is sent to the LCD. After that is complete we return from the subroutine and then jump back to the Start label to receive more data.

That's really all this program does. It hopefully was clear to you, as this program jumps around a lot.

```
' ——-[ Title ]———————————————————————
'
' File..... proj9pbc.BAS
' Purpose... Serial -> PIC16F876 -> LCD (4-bit interface)
' Author.... Chuck Hellebuyck
' Started... January 20,2002
' Updated...

' ——-[ Program Description ]———————————————————
'
'
' PIC16F876 Port Hardware connections:
'
' PIC         LCD                Other Connections
' ———         ——-          ———————————————-
' B4          LCD.11
' B5          LCD.12
' B6          LCD.13
' B7          LCD.14
' B3          LCD.4
' B0          LCD.6
' OSC1                    Resonator - 4 mhz
' OSC2                    Resonator - 4 Mhz
' MCLR          Vdd via 1k resistor
' Vdd                     5v
' Vss                     Gnd
```

```
' B1                              Max232(RX)
' B2                              Max232(TX)

' ——-[ Revision History ]————————————————————
'
'
' ——-[ Constants ]———————————————————————
'
' LCD control pins
'
symbol E = 0                     ' LCD enable pin (1 = enabled)
symbol RS = 3                    ' Register Select (1 = char,
                                 ' 0 = command)

' LCD control characters
'
symbol ClrLCD = $01              ' clear the LCD
symbol CrsrHm = $02              ' move cursor to home position
symbol Row2 = $C0                ' 2nd row position of LCD
symbol Row3 = $94                ' 3rd row position of LCD
symbol Row4 = $D4                ' 4th row position of LCD
symbol CrsrLf = $10              ' move cursor left
symbol CrsrRt = $14              ' move cursor right
symbol DispLf = $18              ' shift displayed chars left
symbol DispRt = $1C              ' shift displayed chars right
symbol Digit = $30               ' Character column code for LCD
' ——-[ Variables ]———————————————————————
'
'B3 reserved for drive routine
symbol x = B0                    ' General purpose variable
symbol char = B1                 ' char sent to LCD
symbol loop1= B2                 ' loop counter
symbol ROW = b5                  ' LCD ROW value
symbol LOCATION = b6             ' Column position on the LCD
symbol VALUE = b7                ' Value is the ASCII Character to
                                 ' display
symbol temp2 = b8                ' unused
symbol temp3 = b9                ' unused
symbol temp4 = b10               ' unused
symbol temp1 = b11               ' unused


' ——-[ Initialization ]——————————————————————
'
Init:
       pins = $0000                        ' all outputs off to start
       Dirs = %11111101                 ' LCD pins
```

```
       PAUSE 215                                    ' pause for LCD
                                                    'setup
' Initialize the LCD (Hitatchi HD44780 controller)
'
I_LCD:


pins = %00110000             'set to 8 bit operation
PULSOUT E,100                'SEND DATA 3 TIMES
PAUSE 10
PULSOUT E,100               'SEND DATA 3 TIMES
PAUSE 10
PULSOUT E,100               'SEND DATA 3 TIMES
PAUSE 10
PINS = %00100000            'SET TO 4 BIT OPERATION
pause 1
PULSOUT E,100               'SEND DATA 3 TIMES
HIGH RS
CHAR = %00101000            '4 BIT, 2 LINES, 5X7 CHARACTER
GOSUB LCDCMD
CHAR = 8                    'Display, cursor and blink off
GOSUB LCDCMD
CHAR = 6                    'Shift display right
GOSUB LCDCMD
CHAR = 1                    'clear display and return home
GOSUB LCDCMD
CHAR = 15                   'display, cursor and blink on
GOSUB LCDCMD

' ——-[ Main Code ]————————————————————
'
'Display initial screen
'
'******* Main Program *****************************************


Start:
       SERIN 1,T2400,ROW,LOCATION,VALUE  'receive serial data
       Row = Row - $30                   ' Correct row to decimal
                                         ' number
       Location = Location - $30         ' Correct location to
                                         ' decimal number


'******* Decision and Branch Routine ****************************
' if ROW = 0 then its a command so jump to command
' if ROW does not = 0 and LOCATION = 0 then write the value where
```

```
' the cursor is. VALUE is the ASCII equivalent of the value sent
' per the LCD ASCII chart
      IF ROW <> 0 AND LOCATION = 0 THEN DISPLAY       'Test for
                                                      'zero value
      BRANCH ROW,(COMMAND,R1,R2)',R3,R4)              'Branch to
                                                      'proper row

'******* position cursor where new value will be written *********

R1:
      LOCATION = LOCATION - 1       'Correct location byte for zero
                                    'value
' Shrunk to fit in one line, lookup table for cursor position LCD
' code
LOOKUP LOCATION,($80,$81,$82,$83,$84,$85,$86,$87,$88,$89,$8A,$8B,
   $8C,$8D,$8E,$8F,$90,$91,$92,$93),char
       GOSUB LCDcmd                 'Send position command to LCD
       GOTO DISPLAY                 'Go to character display
                                    'routine

R2:
      LOCATION = LOCATION - 1       'Correct location byte for zero
                                    'value
'Shrunk to fit in one line, lookup table for cursor position LCD
'code
LOOKUP LOCATION,($C0,$C1,$C2,$C3,$C4,$C5,$C6,$C7,$C8,$C9,$CA,$CB,
   $CC,$CD,$CE,$CF,$D0,$D1,$D2,$D3),char
       GOSUB LCDcmd                 'Send position command to LCD
       GOTO DISPLAY                 ' Go to character display
                                    ' routine

'**** row 3 and row 4 setup for 4x16 LCD. These command commented
'out.
'R3:
'      LOCATION = LOCATION - 1
'LOOKUP LOCATION,($94,$95,$96,$97,$98,$99,$9A,$9B,$9C,$9D,$9E,$9F,
'$A0,$A1,$A2,$A3,$A4,$A5,$A6,$A7),char
'            GOSUB LCDcmd
'      GOTO DISPLAY
'
'R4:
'      LOCATION = LOCATION - 1
'LOOKUP LOCATION,($D4,$D5,$D6,$D7,$D8,$D9,$DA,$DB,$DC,$DD,$DE,$DF,
'$E0,$E1,$E2,$E3,$E4,$E5,$E6,$E7),char
'      GOSUB LCDcmd
'      GOTO DISPLAY
```

```
'****** convert value to be displayed for wrlcd routine
****************

DISPLAY:
      char = VALUE          ' Store received Value character in
                            ' char variable
      GOSUB WrLCD           ' Jump to routine that sends char to
                            ' LCD
      GOTO START            ' jump back to beginning of main loop

'***** ROW=0, Therefore run a command per list below
*******************

'0    clear LCD and move to position 1 of row 1
'1    shift cursor left
'2    Display is off, Cursor is off, Cursor Blink is off
'3    Display is on, Cursor is off, Cursor Blink is off
'4    Display is on, Cursor is on, Cursor Blink is off
'5    Display is on, Cursor is off, Cursor Blink is on
'6    Display is on, Cursor is on, Cursor Blink is on
'7    shift display right
'8    shift display left
'9    shift cursor right

COMMAND:
       value = value - $30     'Correct value to decimal for
                               'command byte only
       '*** Convert byte received to LCD command code
       LOOKUP VALUE,($01,$10,$08,$0C,$0E,$0D,$0F,$1C,$18,$14),char
       GOSUB LCDcmd          'Jump to routine that sends command to
                             'LCD
       GOTO START            'Jump to beginning of main loop

' ****** Send command byte to LCD Routine *************
'LCDcmd:
       LOW RS                          ' RS low = command
       GOSUB WrLCD              ' send the command byte
       HIGH RS                         ' return to character mode
       RETURN                   ' Return to caller of this
                                ' subroutine

'******* Write ASCII char to LCD **********
WrLCD:
       pins = pins & %00001000     ' Set LCD data lines to zero,
                                   ' leave RS bit alone
```

```
b3 = char & %11110000       'store high nibble of char in B3
pins = pins|b3                    'Output high nibble and RS
                                  'signal
pause 1                     'Wait for data setup
PULSOUT E, 100              'Strobe the enable line
b3 = char * 16                    'Shift low nibble to high
                                  'nibble
pins = pins & %00001000     'Set LCD data lines to zero,
                            'leave RS bit alone
    pins = pins|b3                          'Output low
                                            'nibble and RS
                                            'signal
    pause 1                       'Wait for data setup
PULSOUT E, 100              'Strobe the enable line
RETURN                   ' Return to where this subroutine was
                         ' called

END
```

17.3.2 PBPro Code

The PBPro code also may initially look confusing. The first part of the program establishes all the DEFINE statements to tell PBPro which port to use for the data port, RS line, and E line. Each DEFINE then directs which pin(s) of the port are for communication. We even use the DEFINE statements to communicate the 4-bit mode and the number of LCD lines. Finally, we even have a DEFINE to control the time between commands being sent and time delay for data set-up. Some LCDs are picky so PBPro allows you to adjust the timing of its LCDOUT command to work with various LCDs. The DEFINE statements I used here should work with most LCDs, since I really slowed things down.

Next we establish the variables "Row," "Location," "Value,"and "Char"which will be used throughout the PBPro code. After that, the Init label sets up PORTB for proper data direction using the TRIS directive. This is followed by a direct control of PORTB to set the state of each PORTB pin. Finally, we use the LCDOUT command to display an initial message that says "Serial LCD".

The label start is where this project code really starts. The main line is the SERIN line that waits for the row, location, and value bytes. The program will sit here forever if it doesn't receive any information. When information is received it first tests the "Row" byte and "Location" byte to see if "Row" is not 0 and "Location" is 0 by using an IF-THEN statement. If these variables are at that state, it indicates don't move the cursor from its existing position and write the character in the "Value" byte at that location. It does that by jumping to the Display label.

If the "Location" byte is not 0, independent of the "Row" byte value, then the next command run is a BRANCH instruction. The BRANCH instruction jumps the program to the proper label based

on the value of the "Row" byte. If the "Row" byte is 0, then it jumps to the Command label. The code at the Command label will send a LCD command based on the "Value" byte received.

If "Row" does not equal 0, then the BRANCH command redirects the program to the Row1 or Row2 labels. At label Row1 and Row2, the program converts the "Location" byte received into the proper LCD code to position the LCD cursor at the proper row and column using the LOOKUP command. We have to subtract 1 from the "Location" byte because the LOOKUP command starts at 0 instead of 1.

After we have the proper position code from the LOOKUP command, the program then jumps to the LCDcmd subroutine to send the special LCD position code to the LCD and move the cursor on the LCD. When the subroutine is done, the program jumps back to the command after the GOSUB LCDcmd line. That command is a jump to the Display label.

At the Display label, the program first stores the serially received "Value" byte and stores a copy of it into the "Char" variable. Then the program jumps to the subroutine WrLCDto send that byte to the LCD character generator that actually displays the character on the LCD. The program then returns to the routine at the Display label, which sends the program back to the top to receive a new set of information at the Start label. Note how we first positioned the cursor based on the "Row" and "Location" bytes and then sent the character code ("Value" byte) to be displayed at that position.

As mentioned above, if the "Row" byte is zero, the BRANCH command under the Start label redirects the program to the Command label. We don't stop at the Row1 and Row2 labels to get a LCD command code because the "Value" byte received serially should have the LCD command byte in it. All we have to do is convert that "Value" byte to the proper LCD command code based on the table above the Command label. We take the "Value" byte and convert it into a decimal number by subtracting hex $30. Then we use the LOOKUP command to change the "Value" byte into the proper LCD command code. That command code is stored in the "Char" variable.

The next command line jumps the program to the LCDcmd subroutine where the command code is sent to the LCD. After that is complete, we return from the subroutine and then jump back to the Start label to receive more data.

```
' ——-[ Title ]————————————————————————-
'
' File..... proj9pro.BAS
' Purpose... Serial -> PIC16F876 -> LCD (4-bit interface)
' Author.... Chuck Hellebuyck
' Started... January 22 2002
' Updated...

' ——-[ Program Description ]———————————————————-
'
'
```

```
' PIC16F876 Port Hardware connections:
'
' PIC             LCD               Other Connections
' ———             ——-               ————————
' B4         LCD.11
' B5         LCD.12
' B6         LCD.13
' B7         LCD.14
' B3         LCD.4
' B0         LCD.6
' OSC1                           Resonator - 4 mhz
' OSC2                           Resonator - 4 Mhz
' MCLR          Vdd via 1k resistor
' Vdd                            5v
' Vss                            Gnd
' B1                                          Max232(RX)
' B2                                          Max232(TX)


' ——-[ Revision History ]————————————————
'
'
'——-[DEFINES]————————————————————

include "modedefs.bas"          'include serout defines
Define LOADER_USED 1           'Only required if bootloader used to
                               'program PIC


DEFINE LCD_DREG PORTB           'Define PIC port used for LCD Data
                                'lines
DEFINE LCD_DBIT       4                 'Define first pin of portb
                                        'connected to LCD DB4
DEFINE LCD_RSREG PORTB          'Define PIC port used for RS line of
                                'LCD
DEFINE LCD_RSBIT 3              'Define Portb pin used for RS
                                'connection
DEFINE LCD_EREG       PORTB     'Define PIC prot used for E line of LCD
DEFINE LCD_EBIT       0                 'Define PortB pin used for E
                                        'connection
DEFINE LCD_BITS       4                 'Define the 4 bit communication
                                        'mode to LCD
DEFINE LCD_LINES 2             'Define using a 2 line LCD
DEFINE LCD_COMMANDUS 2000      'Define delay between sending LCD
                               'commands
DEFINE LCD_DATAUS 50                    'Define delay time between data
                                        'sent.
```

```
' ——-[ Constants ]———————————————————-
'
' ——-[ Variables ]———————————————————-
'
ROW var byte          ' LCD ROW value
LOCATION var byte     ' Column position on the LCD
VALUE var byte        ' Value is the ASCII Character to display
char var byte                 ' Temporary storage of character code


' ——-[ Initialization ]————————————————
'
Init:
      TRISB = $0000                  ' all outputs off to start
      portb = %11111101            ' LCD pins
      LCDOUT "Serial LCD"      ' Display project name on LCD
      pause 1000                           ' Delay 1 second

' ——-[ Main Code ]———————————————————
'
'Display initial screen
'
'******* Main Program ********************************************

Start:
       SERIN 1,T2400,ROW,LOCATION,VALUE  'Receive serial data
            Row = Row - $30                    'Correct Row to
                                               'decimal number
       Location = Location - $30         'Correct Location to
                                         'decimal number

'******* Decision and Branch Routine *****************************
' if ROW = 0 then its a command so jump to command
' if ROW does not = 0 and LOCATION = 0 then write the value where
' the cursor is. VALUE is the ASCII equivalent of the value sent
' per the LCD ASCII chart

IF ROW <> 0 AND LOCATION = 0 THEN DISPLAY       'Test for zero
                                                'value
BRANCH ROW,[COMMAND,Row1,Row2]',Row3,Row4]      'Branch to proper row

'******* position cursor where new value will be written *********
Row1:
     LOCATION = LOCATION - 1    'Correct location for zero value
'**** Shrunk line to fit, convert location byte to char byte for
                                  'LCD command
```

```
LOOKUP LOCATION,[$80,$81,$82,$83,$84,$85,$86,$87,$88,$89,$8A,$8B,
   $8C,$8D,$8E,$8F,$90,$91,$92,$93],char
       GOSUB LCDcmd                'Jump to LCD command routine
       GOTO DISPLAY                ' Jump to LCD display routine

Row2:
     LOCATION = LOCATION - 1    'Correct location for zero value
'**** Shrunk line to fit, convert location byte to char byte for
LCD command
LOOKUP LOCATION,[$C0,$C1,$C2,$C3,$C4,$C5,$C6,$C7,$C8,$C9,$CA,$CB,
   $CC,$CD,$CE,$CF,$D0,$D1,$D2,$D3],char
       GOSUB LCDcmd                'Jump to LCD command routine
       GOTO DISPLAY                'Jump to LCD command routine
'
'**** These commented out lines are for converting this program to
                        ' 4x16 LCDs

'Row3:
'      LOCATION = LOCATION - 1
'LOOKUP LOCATION,[$94,$95,$96,$97,$98,$99,$9A,$9B,$9C,$9D,$9E,$9F,
   '$A0,$A1,$A2,$A3,$A4,$A5,$A6,$A7],char
'      GOSUB LCDcmd
'      GOTO DISPLAY
'

'Row4:
'     LOCATION = LOCATION - 1
'LOOKUP LOCATION,[$D4,$D5,$D6,$D7,$D8,$D9,$DA,$DB,$DC,$DD,$DE,$DF,
'$E0,$E1,$E2,$E3,$E4,$E5,$E6,$E7],char
'      GOSUB LCDcmd
'      GOTO DISPLAY


'****** convert value to be displayed for wrlcd routine
****************

DISPLAY:
       char = VALUE      'Store Value byte into char variable
       GOSUB WrLCD       'Jump to routine that sends display
                         'characters to LCD
       GOTO START        'Jump back to beginning of main loop

'***** ROW=0, Therefore run a command per list below
******************

'0     clear LCD and move to position 1 of row 1
'1     shift cursor left
```

```
'2     Display is off, Cursor is off, Cursor Blink is off
'3     Display is on, Cursor is off, Cursor Blink is off
'4     Display is on, Cursor is on, Cursor Blink is off
'5     Display is on, Cursor is off, Cursor Blink is on
'6     Display is on, Cursor is on, Cursor Blink is on
'7     shift display right
'8     shift display left
'9     shift cursor right

COMMAND:
       value = value - $30     'Convert value to decimal number for
                               'command only
'**** Convert Value byte to LCD code byte and store in char ***
       LOOKUP VALUE,[$01,$10,$08,$0C,$0E,$0D,$0F,$1C,$18,$14],char
       GOSUB LCDcmd 'Jump to routine that sends LCD commands
       GOTO START 'Jump to beginning of program

' *** Send command byte to LCD Subroutine
'
LCDcmd:
       LCDOUT $FE, char                ' Send command to LCD
       RETURN                 'Return to where this routine was
                              'called

' *** Send ASCII character, to be displayed, to LCD
'
WrLCD:
       LCDOUT char             ' Send char to LCD
       RETURN                  ' Return to where this routine was
                               ' called

       END
```

17.3.3 Final Thoughts

This isn't the best serial LCD example I could have written, but it is unique in that each position of the LCD can be accessed with a single serial command based on a simple row and column (location) value. This project can be expanded in numerous ways. Just playing around with the way the program receives data could be a major change. Possibly making the SERIN command line part of a loop until a certain character is received would allow multiple characters to be received at one time before putting them on the LCD screen. Use your imagination!

Hopefully you've learned how to control a LCD module and how to communicate using the SERIN and SEROUT commands. These are some of the most common functions your PicBasic programs will perform and also brings PICs into the real world. People ask me all the time

what I do with the programming tools I use and sell at my website. Once I show them a PC screen responding to actions at the PIC circuit, it seems to answer their question. When the LCD displays messages to them, they seem to understand this as well. Soon they are making comments about building alarm systems and sprinkler systems, or even ideas for robots. It's the human connection they were looking for and the LCD or PC is just that. Adding an LCD and serial connection to any PIC project makes it look more professional and a lot more complicated than it actually is. But PicBasic makes it easy!

This page intentionally left blank

SECTION IV

Programming PIC Microcontrollers Using MBasic

This page intentionally left blank

CHAPTER 18

MBasic Compiler and Development Boards

18.1 The Compiler Package

18.1.1 A Note on Compiler Versions

By the time this book is published, Basic Micro will have released an updated MBasic compiler (version 5.3.0.0) and rationalized its compiler family, dropping its "standard" version compiler, making the former "professional" version its flagship PIC compiler. (If you are still using version 5.2., check with Basic Micro for upgrade information. Owners of MBasic Professional version 5.2 qualify for a free upgrade, while MBasic Standard owners qualify for a reduced price upgrade to MBasic-Professional.) In addition, Basic Micro has a made available a free version of its MBasic Professional compiler, MBasic876 on the CD-ROM associated with this book. MBasic876 is a complete, 100% functional version of MBasic Professional, limited to working only with the 16F876 and 16F876A devices.

All programs in this book were originally developed and tested with MBasic Professional, version 5.2.1.1 and have been verified with a prerelease version of 5.3.0.0. However, bug-fixes and other "tweaking" to the official release version 5.3.0.0 may occur that introduce minor incompatibilities between the code in this book and Basic Micro's ultimately released compiler. The CD-ROM associated with this book provides both 5.2.1.1-compliant and 5.3.0.0-compliant source code.

Unless specifically noted, this book assumes you are using MBasic or MBasic876, version 5.3.0.0. The printed program listings are for version 5.3.0.0.

18.1.2 MBasic Compiler

As used in this book, Basic Micro's MBasic compiler comprises three main elements:

1. MBasic Compiler Software—From version 5.3.0.0 onward, Basic Micro offers one version of its MBasic compiler, the "Professional" version. MBasic runs under Microsoft's Windows operating system in any version from Windows 95 to Windows XP. The computer requires an RS-232 port for connection to the ISP-PRO programmer board. A second RS-232 port, although not essential, is useful to capture any serial information

from the program you are developing. If your computer does not have a second serial port, but does have a USB port, you may wish to add one using an inexpensive USB-to-serial converter.

2. ISP-PRO Programmer—MBasic, after the assembly stage completes, generates Microchip-compatible standard HEX code file that must be loaded into the PIC. Basic Micro offers a programmer, the ISPPRO, well integrated with the MBasic compiler that automatically loads HEX code file. A major plus of Basic Micro's ISP-PRO is real-time debugging through its "in-circuit debugging" or ICD capability. Although it would be possible to substitute a third-party programmer for the ISP-PRO, losing both seamless integration with the compiler and ICD ability more than offsets any cost savings. The ISP PRO communicates with the computer running MBasic via an RS-232 cable, and with the PIC to be programmed through a 6-wire RJ11 telephone-type cable for Basic Micro's development and prototype boards, or a 10-pin standardized header for other boards.
3. Development Board—Basic Micro offers plug board style development boards and solder-in prototype boards for 8- and 18-pin and 28- and 40-pin PICs. The experiments in this book assume the user has Basic Micro's development boards. These boards have an RJ11 connector for the ISP-PRO connection and an uncommitted RS-232 port that may be used by the PIC for communications to the outside world.

Note on serial ports: The single largest source of trouble reported in calls to Basic Micro's help line concerns unreliable serial port connections with laptop computers. The built-in serial port on many laptop computers cannot reliably operate at 115.2 kb/s, the default speed at

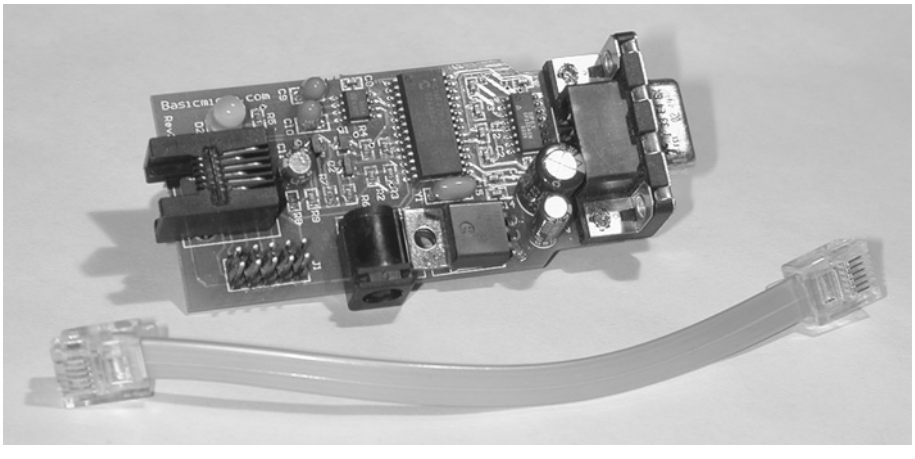


Figure 18.1: ISP-PRO and RJ-11 Jumper Cable

which the PC-to-ISP-PRO communications link operates. In those cases, Basic Micro suggests using an inexpensive add-on USB-to-serial adapter to substitute for the built-in serial port and recommends Bafo Technologies' BF-180 USB-to-serial adapter. A slightly more expensive alternative that I have had reliable results with is Belkin's F5U109, sold as a "USB PDA Adapter," but which is, in fact, a straight USB-to-serial adapter. Many other USB-to-serial adapters likely will provide reliable results.

In addition to the development and prototype boards, the ISPPRO is compatible with Basic Micro's Universal Adapter. The Universal Adapter, however, does not contain an oscillator or the other circuitry needed to actually run a PIC program, and is intended for programming only.

18.2 BASIC and Its Essentials

This book is not intended to teach BASIC programming from the ground up. There are many good "BASIC programming for the beginner" books and we assume the reader has at least passing familiarity with program control statements, mathematic procedures and variable assignment and structure. It also assumes the reader has installed the MBasic compiler (either the full version or MBasic876, version 5.3.0.0 as of the date of writing) and has familiarized himself with the first 80 pages or so in the MBasic User's Guide. Incidentally, because MBasic is, in some respects, a return to the early days of micro computer language implementation, I've found 20-year old reference documents for IBM's Personal Computer BASIC beneficial in refreshing my memory on some of the finer points of BASIC syntax or procedure and of considerably more help than modern books detailing, for example, Visual Basic. A visit to your local used bookstore may turn up useful reference material. I've provided the names of a few of my favorite long-out-of-print BASIC references in this chapter's reference section.

As a guide to finding the appropriate procedure, Table 18.1 groups MBasic's commands into a logical classification.

Table 18.1: Taxonomy of MBasic Functions and Procedures

Group	Procedure	Group	Procedure

Program Flow	`Repeat/Until`
	`While/WEND`
	`Do/While`
	`For/Next`
	`If/Then/Else/EndIf`
	`GoTo`
	`GoSub/Return`
	`Branch`
		Hardware Related	`ADIN`
			`ADIN16`
			`Count`
			`HPWM`
			`SetCapture`
			`GetCapture`
			`SetCompare`

(Continued)

Table 18.1: Continued

Group	Procedure	Group	Procedure

Pin Related	`Button`
	`Low`
	`PulsIn`
	`PulsOut`
	`RCTime`
	`Reverse`
	`Toggle`
	`SetPullups`
	`INxx`
	`Outxx`
	`Dirxx`
		Miscellaneous	`DeBug`
			`End`
			`Let`
			`Nap`
			`Sleep`
			`Stop`

EEPROM	`Data`
	`Read`
	`ReadDM`
	`Write`
	`WriteDM`
		Variables	`Clear`
			`Swap`

I/O	`I2Cin`
	`I2Cout`
	`Owin`
	`Owout`
	`SerDetect`
	`SerIn`
	`SerOut`
	`ShiftIn`
	`ShiftOut`
	`HSerIn`
	`HSerOut`
		Sound and Sound Related	`DTMFOut`
			`DTMFOut2`
			`FreqOut`
			`PWM`
			`Sound`
			`Sound2`

LCD	`LCDWrite`
	`LCDRead`
	`LCDInit`
		Data Table	`LookDown`
			`LookUp`

Timing	`Pause`
	`PauseUs`
	`PauseClk`
		Memory Related	`Peek`
			`Poke`

Random Generator	`Random`	Program Memory	`ReadPM` `WritePM`

On Reset	`OnPOR`
	`OnBOR`
	`OnMOR`
		Explicit External Device Support	`Servo`
			`SPMotor`
			`Xin`
			`Xout`

(Continued)

Table 18.1: Continued

Group	Procedure	Group	Procedure
Interrupts	Enable Disable OnInterrupt SetExtInt SetTmr0 SetTmr1 SetTmr2 IsrASM GetTimer1	Assembler	ASM {} ISRASM
Command Modifiers	Dec Hex Bin Str Sdec Shex Sbin Ihex Ibin ISHex ISBin REP Real WaitStr Wait Skip	Math Operators and Functions	+ - * LowMult HighMult FractionalMult / // ABS SIN COS DCD SQR BIN2BCD BCD2BIN Max Min Dig Rev
Bitwise Operators	! & | ^ >> <<	Comparison Operators	= <> < > <= >=
Logical Operators	And Or Xor Not Not Not And Or Xor	Floating Point Conversion	ToInt ToFloat FloatTable

18.3 Development Boards

Basic Micro offers two breadboard style development boards; models 0818 for 8- and 18-pin DIP PICs (Fig. 18.2), and the 2840 for 28- and 40-pin DIP PICs (Fig. 18.3). Both boards have a small solderless plug-in area for additional components and are fully assembled with surface-mount components. Sockets are installed for the PICs. An expanded development board, is under development.

Additionally, Basic Micro offers corresponding semi-permanent prototype boards, models 08/18, Fig. 18.4 and 28/40, Fig. 18.5 differing from the development boards in that additional

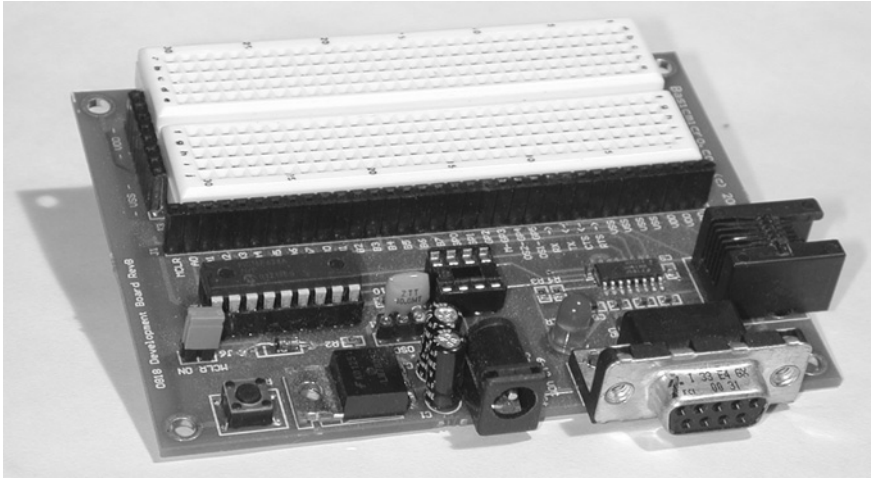


Figure 18.2: Basic Micro's 0818 Development Board

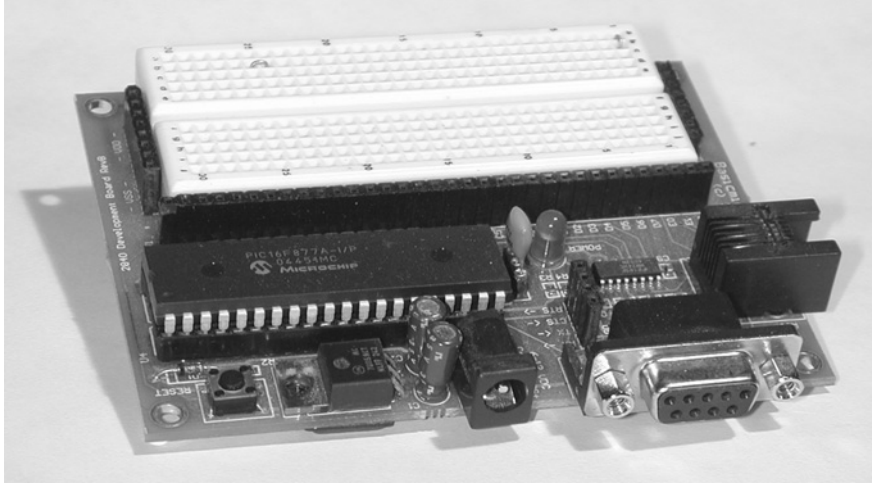


Figure 18.3: Basic Micro's 2840 Development Board

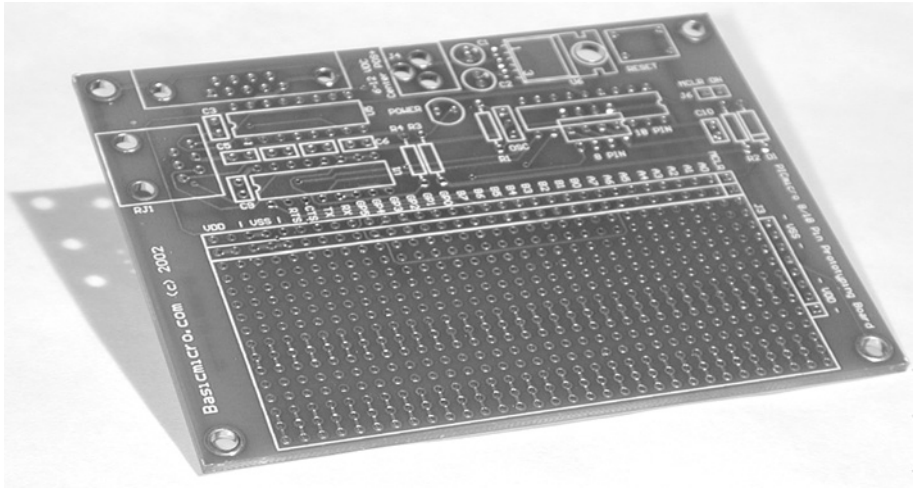


Figure 18.4: Basic Micro's 08/18 Prototype Board

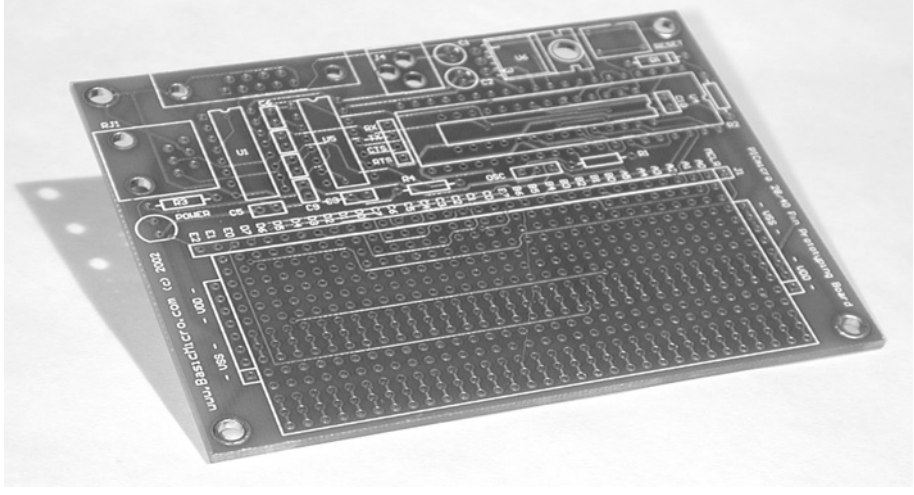


Figure 18.5: Basic Micro's 28/40 Prototype Board

components are to be soldered in rather than plugged into a solderless breadboard. These are sold as bare boards, but Basic Micro also offers an inexpensive complete parts kit. The prototype boards use through-hole components.

All four boards permit in-circuit programming—that is, the PIC may be programmed without removing it from your board, or disconnecting its pins from whatever you may have connected to them. Figure 18.6, a simplified block diagram of the 28/40 prototype board, shows how this is possible. Three of the pins required for programming, RB4, RB6 and RB7, are switched through a 74HC4053 analog multiplexer/demultiplexer between their normal connection to

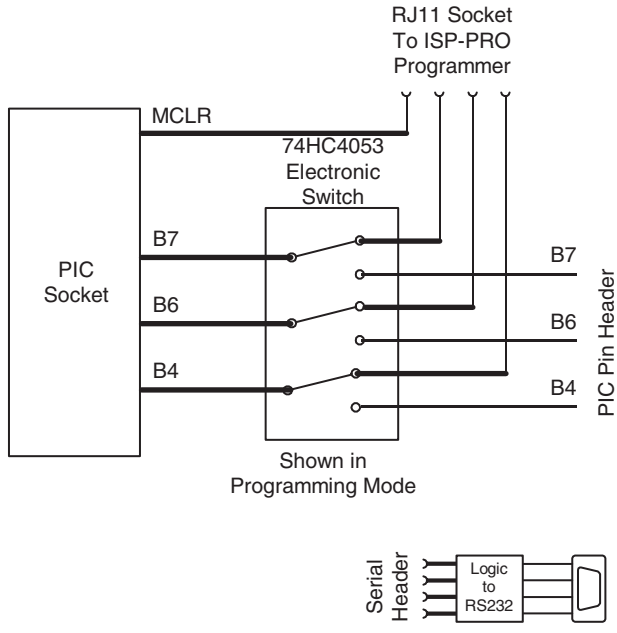


Figure 18.6: Simplified Block Diagram of 28/40

the PIC pin header and the RJ11 socket that connects to the ISP-PRO programmer. For our purpose, the 74HC4053 can be regarded as an electronic three-pole double throw switch, controlled by the ISP-PRO. The MCLR (master clear) pin is the fourth connection required for programming and is directly connected to the RJ11 programming socket.

The 0818 and 08/18 boards follow a similar design, but with extra configuration jumpers necessitated by the multiple functions Microchip assigned to certain pins of PICs produced in 8 and 18-pin packages. The 0818 and 08/18 data sheets should be consulted before programming these small PICs.

All four boards bring the various PIC pins to logically labeled headers; for example, A0, A1, so you don't have to continually cross-reference physical pin numbers with their logical assignments.

In working with Basic Micro's development boards and ISP-PRO programmer, watch out for the following:

- These are sold as bare boards, with unprotected traces on the bottom. Don't put them down on conductive surfaces or the board may be damaged and watch for stray wires or component leads as well. (I watched my ISP-PRO board be dragged by its serial cable across the metal edge on the table and looked on helplessly as sparks flew. Needless to say, the ISP-PRO didn't work after that.) It helps to add small stick-on rubber feet to the bottom of all boards.

- It is possible to damage the 74HC4053 electronic switch, as it is rated at a maximum switched current of 25 mA. The most likely damage scenario comes from forcing the PIC to sink excessive current. Additionally, unlike a mechanical relay, the 74HC4053 introduces approximately 80 to 100 ohms of series resistance.

Another difficulty beginners often have is confusing V_{DD} and V_{SS} when wiring circuits. V_{SS} is ground in Basic Micro's development boards. V_{DD} is the supply voltage and is +5 volts in the development boards. Thus a schematic reference to +5 V is the same as V_{DD} and a reference to ground corresponds to V_{SS}. (This terminology comes from V_{DD} as the "drain" voltage and VSS as the "source" voltage for a field effect transistor, the basic building block of PICs.)

18.4 Programming Style

Every program printed in the text is also provided as a file in the accompanying CD-ROM with two versions supplied—the originally developed programs compatible with MBasic version 5.2.1.1. and a revised version compatible with 5.3.0.0. There may be differences between the printed program and the CD-ROM for several reasons:

- The CD-ROM is quicker to update and may have a later or corrected version of the text program.
- Page width and overall length restrictions make it necessary to limit the comments and blank spaces used on the printed page. The data files have no similar restriction and hence may have additional comments and may be formatted for greater readability. (Although not documented in the User's Guide, MBasic uses the vertical bar " | " as a continuation symbol, thus allowing one logical line of BASIC code to be split over two or more physical lines.)

18.4.1 Standard Program Layout

As an aid in readability and maintainability, I like to follow a standard layout when programming, as exemplified in the following template:

```
;Program Sample.Bas - File name
;Version 1.00
;14 September 2003 - original version
;
;Constants
;-----------------
Define constants here

;Variables
;-----------------
Declare variables here
```

```
;Initialization
;--------------
Initialization code here - this code is executed only once

Main
;---------
Main code segment here
If something GoSub Sub1
If something else GoSub Sub2
If something totally different happens GoSub Sub3
GoTo Main ;if appropriate to have a continuous loop

Sub1
;-----
      Subroutine code here
Return

Sub2
;-----
      Subroutine code here
Return

Sub3
;-----
      Subroutine code here
      Includes GoSub SubSub1

Return

SubSub1
;------
      Subroutine code goes here
Return

End
```

The structure is logical; first define constants and variables and conduct any necessary initialization, then write the main program segment. I use subroutines as necessary, with the goal of keeping the main segment short and to the point. MBasic has no restriction on the number of subroutines that may be nested, so one subroutine may call another.

With version 5.3.0.0, MBasic introduced user-defined functions, albeit with global, not local, variables. This feature was added too late to be used in the programming examples.

18.4.2 Constants, Variable and Subroutine Names

Intelligently selecting constant, variable and names aids program readability. MBasic permits variables and constants to have names up to 1024 characters long, should you so wish. (MBasic has several hundred reserved words and you shouldn't use these names. In most

cases you will receive a warning or error message if you try to redefine a reserved word.) All names are case insensitive, so we can use capitalization to improve readability without worrying about consistency. The naming conventions I've developed include:

- Index or counting variables, for example, control variables used in For/Next loops, start with letters in the range i…n. (Yes, this is a holdover from the early days of FORTRAN when integer variable names had to start with one of these letters.) Keep these names short, particularly if they will be used frequently. In many cases, the single letters i, j…n are perfectly suitable.
- Names should reflect contents or activities, without being overly long. Suppose we use the A/D converter to read a voltage followed by a subroutine call to average this reading with the last 15 readings—that is, a moving average of 16 readings and that the individual reading isn't elsewhere used. Since the voltage value being read will be discarded after the averaging process, we may use `TempVolt`, so the A/D read statement would be: `ADIN AN0, CLK, ADSETUP, TempVolt`. I like to name variables that have limited scope with a two-part name, starting with Temp. To make the name more readable, use upper and lower case, thus `TempVolt` is easier to read than `tempvolt` or `TEMPVOLT`. Or, insert an underscore as a separator; `Temp_Volt`. (Permissible separator characters are _,@,$,%,? and `.) Subroutines should be named according to the activity performed, in this example, `TakeAverage` is an appropriate name, or even `TakeAverageVolt`. I would reserve the name `AvgVolt` for the variable holding the average of the last 16 readings, thus keeping the suffix name `Volt` and changing the prefix to describe the type of voltage parameter in the variable. Short, concise and informative variable names can often be constructed of the form "adjective–noun" while subroutine names are often of the form "verb-nou n" where the verb describes the action and the noun describes the subject of the action.
- Although very long names, such as `AverageTheLast16VoltageReadings` may seem descriptive at first, they actually hinder, rather than assist comprehension if they are used with abandon.

These are guidelines, not hard and fast rules, and even my observance isn't 100%, but following a logical consistent naming approach will pay dividends in the long run in terms of fewer errors and improved readability.

18.5 Building the Circuits and Standard Assumptions

In addition to the MBasic compiler (either the full or MBasic876 free versions), an ISP-PRO programmer and a 2840 development board, and any associated parts required for specific projects, you should have access to:

- A second larger plug board to hold overflow circuitry.
- Assorted jumper wires. You can purchase a kit of jumper wires, or make your own from a short length of scrap 25-pair or 50-pair telephone cable.

- A second adjustable regulated power supply, preferably one that has two independent outputs so that positive and negative voltages may be provided.
- A digital multimeter.
- Your ability to troubleshoot, experiment and verify operation of circuits will be greatly enhanced if you have a triggered oscilloscope.

Instead of buying resistors and capacitors one or two at a time, consider buying assortment kits. Possible suppliers are identified in Appendix A. Almost all resistors used with a PIC will be 10 K or less in value and are ¼ watt dissipation rating, so even a limited assortment of values will be quite beneficial.

Since we use prebuild development boards, certain things are omitted in the schematics provided in this book.

- Basic Micro's prototype and development boards include bypass capacitors on the V_{DD} supply headers. Hence, they will not normally be illustrated in schematic diagrams. However, if parts of the test circuitry is build on an second plug board, or if you are designing a printed circuit board to hold your design, good design practice says you should liberally bypass V_{DD}.
- The development boards are designed for ceramic resonator frequency determining elements and are shipped by Basic Micro with a 10 MHz resonator. The circuits and software in this book use a 20 MHz resonator. In many cases, there will be no difference in performance, but for maximum compatibility with the programs in this book, use a 20 MHz resonator. Resonators cost under $1 and are available from suppliers.

18.5.1 Choice of PIC

I've used a 16F877A to develop the circuit in this book, but none of the programs depend upon the "A" version's added features, so a 16F877 will work equally well. Except for those few circuits that are input/output pin constrained, you may substitute a 16F876 or '876A in any design and use Basic Micro's free MBasic876 compiler for almost every program we develop.

18.6 Pins, Ports and Input/Output

Since every useful program must read from or write to a PIC's input/output pins, let's summarize how MBasic handles pins and ports. It can be confusing because some pins have triple or quadruple or even more duties and because MBasic provides several ways to address any given pin. And, the word pin itself has dual usage, as it refers to the physical packaging (an 8-pin DIP package for example) and to those physical pins that may be used for various purposes. To simplify our discussion we will limit ourselves to PICs that are supported by MBasic and plug into either the 0818 or 2840 development boards.

PICs communicate with external circuitry through intermediary "ports." Ports are treated internally by the PIC's CPU, and by MBasic, as byte (8-bit) variables with each bit corresponding to a particular pin. For example, the most significant bit in PortB's byte value corresponds to pin RB7, while the least significant bit corresponds to RB0. (In some PICs, not all bits of each port variable have physical pin assignments.)

Letters from A…E identify ports, except in DIP8 packaged PICs which have only one port, called GPIO (general purpose input/output). Thus, we have GPIO, PortA, PortB…PortE as predeclared variables in MBasic. (Port identifiers are written without a space, for example, PortA, not Port A.) Of course, not all PICs physically support all of these ports, and in some cases not all eight bits of a port have associated pins. For example, the PIC16F876 has PortA (but only bits 0…5 are mapped to pins), PortB and PortC. MBasic's configuration files, fortunately, ensure that only legitimate port variables are predeclared for the particular PIC being programmed.

Figure 18.7 illustrates, as an example, Port B and its pin assignments. Each general-purpose I/O pin is identified with a consistent naming convention. For example, RB0 is PortB, bit 0. The "R" in RB stands for "register," which is synonymous with "file" or "variable." MBasic also predefines constants associated with each of these pins, so we have constants B0, B1…B7 available to us. MBasic gives us a second way to reference pins through a sequential numbering system, for example, A0 = 0, A1 = 1 up through E7 = 39. Finally, to provide backwards compatibility with the Basic Stamp™, MBasic includes the dedicated functions `INxx`, `OUTxx` and `DIRxx` where xx is the bit, nibble, byte or word identifier as reflected in Table 18.2.

MBasic permits us to reference a port or a pin as an address or as a variable. As an address, the port or pin is an argument to certain functions. As a variable, the value of the port (either in reading or writing) can be used like any other variable. There are also the dedicated functions identified in Table 18.2 that operate on specific ports or pins without an explicit port or pin reference,

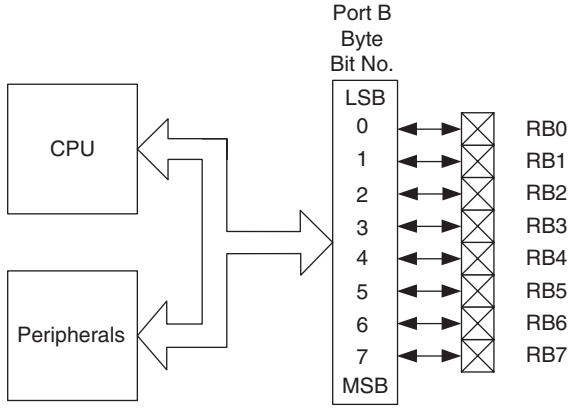


Figure 18.7: PortB to Pin Assignments

Table 18.2: Port and Bit I/O Variables, Constants and Dedicated Functions

	Variables		Constants		Dedicated Functions			
Port at a Time	Nibble at a Time	Bit at a Time						
					INS/OUTS/DIRS			
Port Variable	Nibble	Port.Bit	Pin Constant	Pin Value	Bit	Nibble	Byte	Word
PortA	PortA.Nib0 or PortA.LowNib	PortA.Bit0 PortA.Bit1 PortA.Bit2 PortA.Bit3	RA0 RA1 RA2 RA3	0 1 2 3	IN0/OUT0/DIR0 IN1/OUT1/DIR1 IN2/OUT2/DIR2 IN3/OUT3/DIR3	INA/OUTA/DIRA	INL/OUTL/DIRL	INS/OUTS/DIRS
	PortA.Nib1 or PortA.HighNib	PortA.Bit4 PortA.Bit5 PortA.Bit6 PortA.Bit7	RA4 RA5 RA6 RA7	4 5 6 7	IN4/OUT4/DIR4 IN5/OUT5/DIR5 IN6/OUT6/DIR6 IN7/OUT7/DIR7	INB/OUTB/DIRB		
PortB	PortB.Nib0 or PortB.LowNib	PortB.Bit0 PortB.Bit1 PortB.Bit2 PortB.Bit3	RB0 RB1 RB2 RB3	8 9 10 11	IN8/OUT8/DIR8 IN9/OUT9/DIR9 IN10/OUT10/DIR10 IN11/OUT11/DIR11	INC/OUTC/DIRC	INHOUTH/DIRH	
	PortB.Nib1 or PortB.HighNib	PortB.Bit4 PortB.Bit5 PortB.Bit6 PortB.Bit7	RB4 RB5 RB6 RB7	12 13 14 15	IN12/OUT12/DIR12 IN13/OUT13/DIR13 IN14/OUT14/DIR14 IN15/OUT15/DIR15	IND/OUTD/DIRD		
PortC	PortC.Nib0 or PortC.LowNib	PortC.Bit0 PortC.Bit1 PortC.Bit2 PortC.Bit3	RC0 RC1 RC2 RC3	16 17 18 19	INxx: Read status, whether in input or output mode OUTxx: Write value DIRxx: Set direction 1=input, 0=output			
	PortC Nib1 or PortC.HighNib	PortC.Bit4 PortC.Bit5 PortC.Bit6 PortC.Bit7	RC4 RC5 RC6 RC7	20 21 22 23				

(Continued)

Table 18.2: Continued

Variables			Constants		Dedicated Functions			
Port at a Time	Nibble at a Time	Bit at a Time						
					INS/OUTS/DIRS			
Port Variable	Nibble	Port.Bit	Pin Constant	Pin Value	Bit	Nibble	Byte	Word
PortD	PortD. Nib0 or PortD. LowNib	PortD.Bit0 PortD.Bit1 PortD.Bit2 PortD.Bit3	RD0 RD1 RD2 RD3	24 25 26 27	xx as appropriate for bit, nibble, byte or word Note: DIRxx command is reversed from Basic Stamp			
	PortD. Nib1 or PortD. HighNib	PortD.Bit4 PortD.Bit5 PortD.Bit6 PortD.Bit7	RD4 RD5 RD6 RD7	28 29 30 31				
PortE.	PortE.Nib0 or PortE. LowNib	PortE.Bit0 PortE.Bit1 PortE.Bit2 PortE.Bit3	RE0 RE1 RE2 RE3	32 33 34 35				
	PortE.Nib1 or PortE. LowNib	PortE. Bit4 PortE.Bit5 PortE.Bit6 PortE.Bit7	RE4 RE5 RE6 RE7	36 37 38 39				

such as IN0. We must remember that MBasic automatically initializes all I/O pins as inputs and that before reading from or writing to a port or a pin we must follow some simple rules:

First, set the direction of the port or pin to be either an input or output;
Second, read the port or pin if an input, or write to the port or pin if an output;

or,

Read from or write to a port or pin with a procedure that automatically sets the direction.

18.6.1 Output Mode

Let's see how many different ways we can assign a pin to an output and to make its value 0. We'll use pin RB0 as our example.

```
;Direct Pin Addressing
;--------------------
Output B0             ; First make it an output. B0 is a constant
PortB.Bit0 = 0        ; PortB.Bit0 is a variable
```

```
Dir8 = 0              ; Special purpose function, DIR8 is for pin B0
Out8 = 0              ; Likewise for Out8

Low B0                ; Low function automatically makes it an output
                      ; no need to separately make it into an output
Output 8              ; B0 is an alias for 8 so can use 8 directly
PortB.Bit0 = 0        ; Make the variable assignment

Low 8                 ; B0 is an alias for 8 so can use 8 directly
                      ; LOW switches to output mode and outputs 0

TRISB.Bit0 = 0        ; TRISB variable controls PortB I/O direction,
                      ; 0=output & 1=input.
PortB.Bit0 = 0        ; PortB.Bit0 is a variable

;Byte at a time addressing to deal with multiple pins
;in one instruction
;-------------------------------------------------
TRISB = %00000000     ; Sets all 8 pins to 0, i.e., output
PortB = %00000000     ; Assign all 8 bits (pins) to 0.

DIRH = %00000000      ; Make all 8 Pins in PortB output
OUTH = %00000000      ; Set all 8 bits (pins) to 0
```

18.6.2 Input Mode

To make RB0 an input and read its value, we have the choice of a similar set of options:

```
;Direct Pin Addressing
; Assume we have already declared:
;    BitVar  Var    Bit
;    ByteVar Var    Byte
; to hold the value being read
;--------------------
Input B0              ; First make it an input. B0 is a constant
BitVar = PortB.Bit0   ; PortB.Bit0 is a variable

Dir8 = 1              ; Special purpose function, DIR8 is for pin B0
BitVar = In8          ; Likewise for IN8

Input 8               ; B0 is an alias for 8 so can use 8 directly
BitVar = PortB.Bit0   ; Make the variable assignment

TRISB.Bit0 = 1        ; TRISB variable controls PortB I/O
                      ; direction,0=output & 1=input.
BitVar = PortB.Bit0   ; PortB.Bit0 is a variable

;Byte at a time addressing to deal with multiple pins
;in one instruction
;-------------------------------------------------
```

```
TRISB = %11111111       ; Sets all 8 pins to 1, i.e., input
ByteVar = PortB         ; Read all 8 bits (pins) into ByteVar.

DIRH = %11111111        ; Make all 8 Pins in PortB input
ByteVar = INH           ; Read all 8 bits (pins) into ByteVar
```

18.6.3 Pin Variables vs. Addresses

One common error by beginners is confusing pin *variables* with pin *addresses*. The functions `Output`, `Low` and `Input` require a pin *address* as their argument. The pin address may be one of MBasic's predefined constants, for example, `B0`, or its equivalent numerical value, `8`. The pin address may also be the value of a variable, such as:

```
For I = B0 to B7        ; I goes from 8 (B0) to 15 (B7)
        Low I           ; Makes B0 low, then B1 through B7 sequentially
Next
```

In the `Low I` statement, `Low` operates on the value of `I`, which it interprets as the address of a pin. When `I` is `8`, for example, `Low` operates on pin RB0. Thus, the above code fragment is identical with:

```
For I = 8 to 15
        Low I
Next
```

Pin *variables* are used to read the value of a pin or of a port and to write to a pin or port via an *assignment* (the "=" sign). Thus we have `ByteVar = PortB`, or `PortB = $FF`. We also may use `PortB` like any other byte variable, such as `x=2*PortB`.

If we try to read the *value* of pin B0 as an input with the statement `BitVar = B0`, the compiler will produce no error, but `BitVar` will not hold the desired result. Rather, this statement is identical with `BitVar = 8`. If testing for a pin value condition in a loop statement, it's important that the variable construct be used.

```
;To test Pin B0
;--------------
If PortB.Bit0 = 1 Then
; execute code goes here
EndIf
;The following code will compile but won't work
;since it's the same as writing If (8 = 1), which is always false
If B0 = 1 Then
;execute code goes here
EndIf
```

Finally it's possible to *read* from a pin or port that is set for output in whole or in part, and to write to a pin or port that is set for *input* in whole or in part. No error message will be generated.

If you are experiencing strange or unstable results reading or writing to pins or ports, check to ensure the correct direction is set and that you are correctly using pin variables and pin constants.

PICs equipped with analog-to-digital converters apply the designators AN0, AN1… to pins that also have an analog function. Thus, a 16F876 pin name RA0/AN0 indicates that the pin has three possible uses: digital output, digital input (RA0) and analog (AN0) input.

18.6.4 Run Time vs. Program Time Pin Assignments

All the pin assignments we have discussed to this point are run time alterable, i.e., their status may be altered by the program on the fly. In one part of your program a pin may be an input and later in the program the same pin may be an output. However, in some PICs—most often those in the 8 and 18-pin packages—certain pin configurations may only be established at program time, a task usually accomplished via an option dialog box in MBasic before compiling your code. (This permits Microchip to make their smaller package devices more flexible, but at the cost of confusion to beginning programmers.) Then, depending upon the program time configuration, further run time changes may be possible. Program time pin setup is highly device specific and reference to the data sheet for your specific PIC will be beneficial.

We'll explore the difference between run time and program time alterable pins in the context of the 12F629, which has 3 pins that must be configured at program time:

12F629 Example of Pins Configured at Program Time		
Pin Name	**Program Time Configuration**	**Run Time Configuration**
GP3/MCLR/Vpp	GP3 (general-purpose I/O)	GP3: Input GP3: Output
	MCLR/Vpp (master clear/Vprogram)	None
GP4/T1G/OSC2/CLKOUT	GP4 (general-purpose I/O)	GP4: Input GP4: Output T1G (timer 1 gate)
	OSC2 (second resonator connection)	None
	CLKOUT (clock out)	None
GP5/T1CKI/OSC1/CLKIN	GP5 (general-purpose I/O)	GP5: Input GP5: Ouput T1CKI (Timer 1 clock in)
	CLKIN (external clock input)	None
	OSC1 (first resonator connection)	None

If the GP3/MCLR/Vpp pin is defined at program time to be a general-purpose I/O pin, it may be used for input or output exactly as we have earlier discussed, and changed from input to output under program control. However, if at program time it is defined as MCLR, it is

unavailable for any other purpose. This selection is accomplished with the MCLR check box found in MBasic's Configuration dialog box, as shown in Fig. 18.8.

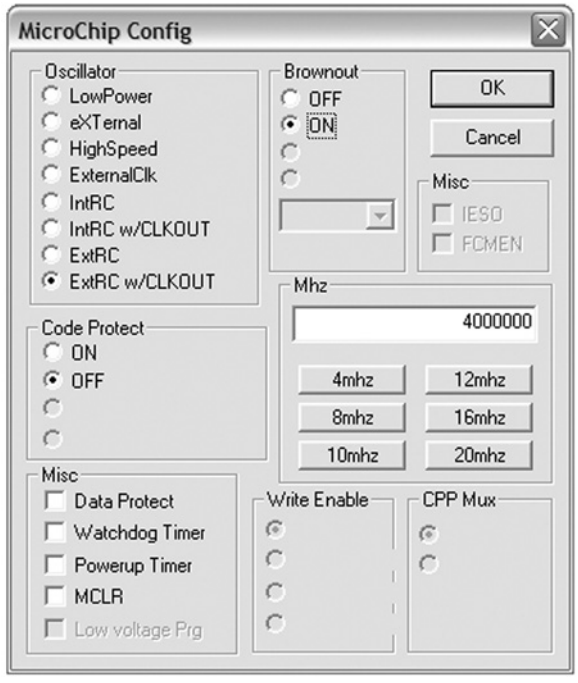


Figure 18.8: Program Time Pin Options for 12F629

The two oscillator pins also must be defined at program time, but are linked. If you plan to use an external resonator or crystal, the OSC1 and OSC2 pin configuration must be active. If you plan to use an external clock source, then the CLKIN option must be active. If you wish to use the internal RC oscillator, then the CLKOUT pin may either be GP5 or OSCOUT. If an external RC oscillator is used, the RC network must connect to the OSC1 pin. Table 18.3 shows how these options interact and how they are selected in the MBasic configuration box of Fig. 18.8. (The MBasic configuration options correspond to the first column in Table 18.3.)

Table 18.3: Configuration Dialog for 12F629—Oscillator Configuration

Configuration Dialog Box	Oscillator Configuration	GP4/T1G/OSC2/CLKOUT Function	GP5/T1CKI/OSC1/CLKIN Function
LowPower	LP	OSC2—crystal connection	OSC1—crystal connection
External	XT	OSC2—crystal connection	OSC1—crystal connection
HighSpeed	HS	OSC2—crystal connection	OSC2—crystal connection
ExternalClk	EC	GP4 (general-purpose I/O)	CLKIN (external clock input)

(Continued)

Table 18.3: Continued

Configuration Dialog Box	Oscillator Configuration	GP4/T1G/OSC2/CLKOUT Function	GP5/T1CKI/OSC1/CLKIN Function
IntRC on GP4	INTOSC	GP4 (general-purpose I/O)	GP5 (general-purpose I/O)
IntRC on ClkOut	INTOSC	CLKOUT (clock waveform output)	GP5 (general-purpose I/O)
ExtRC on GP4	RC	GP4 (general-purpose I/O)	CLKIN-RC circuit on pin
ExtRC on ClkOut	RC	CLKOUT (clock waveform output)	CLKIN-RC circuit on pin

Further complicating an already complex matter, an external clock source may be used in the LP, XT and HS modes by feeding it into OSC1, in which case, OSC2 is unused. The LP, XT and HS modes set internal parameters in the oscillator section of the 12F629 and establish the maximum resonator or crystal frequency and associated capacitor values. Section 9 of Microchip's PIC12F629/675 Data Sheet should be consulted for specifics.

18.6.5 LVP Programming Pin Selection

One compile time feature shared by 16F876/877 chips (including the "A" versions), along with many other flash program memory PICs, is the low voltage programming (LVP) option. Historically, flash memory has required application of a programming voltage two or three times that of the normal operating voltage, typically 12V for a PIC operating with V_{DD} of 5V, known as high voltage programming (HVP). Newer PICs, such as the 16F876/877/876A/877A may be optionally programmed in LVP mode, using only +5V. Whether a PIC that supports LVP actually has LVP enabled is determined by a configuration bit, the value of which is keyed to the LVP checkbox in MBasic's configuration setup dialog box seen in Fig. 18.8. If the chip does not support LVP, as is true in Fig. 18.8, the LVP check box is grayed out.

To ensure maximum flexibility when programming both older and newer model PICs, Basic Micro's ISPPRO and its 0818 and 2840 Development Boards use high voltage programming and for the programs in this book you should not select LVP mode in MBasic's programming options. Microchip enables LVP as part of the manufacturing process, so when programming a new PIC for the first time you will find it necessary to clear the LVP selection box in MBasic's configuration menu, if that model PIC has LVP functionality.

Basic Micro's ISP-PRO does not support LVP and programs only using HVP mode. But, since a PIC with LVP enabled is still programmable via HVP, you can, nonetheless, select LVP and program 16F876/877/876A/877A chips with MBasic and the ISP-PRO. However, if you do so, pin RB3 becomes the LVP control pin and is no longer available as a general-purpose I/O pin. (The specific pin used for LVP control varies; for example, a 16F628 uses RB4.) Not every LVP-capable model PIC behaves so nicely and you may find some model devices refuse to program if you inadvertently leave the LVP option selected. I've even seen different samples of the same model PIC behave differently with the LVP option selected. In this case, clearing the LVP check box, followed by several cycles of MBasic's "erase" function usually restores programmability.

18.6.6 Weak Pull-Up

One last remark and we may leave this overly long discussion of pins. Many PICs have built-in "weak" pull-up resistors for Port B, usable when set to be an input. We'll deal with floating input gates and the need for pull-up resistors in Chapter 4, but MBasic's procedure for controlling Port B's internal pull ups is `SetPullUps <mode>` where mode is one of two pre-defined constants, `Pu_Off` or `Pu_On` for de-activating or activating, respectively, internal pull-up resistors. Pull-up resistors for all eight pins of Port B are activated or deactivated by this command, not individual pins. For Port B pins that are set to be outputs, `SetPullUps` has no effect.

18.7 Pseudo-Code and Planning the Program

In describing an algorithm or even a complete program, we will often use a mixture of English and MBasic statements called "pseudo-code." Pseudo-code is a useful tool when developing an idea before writing a line of true code or when explaining how a particular procedure or function or even an entire program works. To distinguish it from MBasic or assembler, pseudo-code will appear in bold, italic Courier typeface.

Let's illustrate the benefit of pseudo-code with a simple example. Suppose we wish to illuminate an LED for 1 second whenever a button is pushed. We'll assume that the button is connected to the RB0 pin on a PIC, that pressing the button takes RB0 low and that the LED is illuminated by taking pin RB1 high. After any button push, the program repeats and waits for the next push. (We'll ignore button debounce, multiple button presses and a few other real-world concerns.) A pseudo-code version of our program is:

```
Initialization Routine
Initialize button pin to be an input
Initialize LED pin to be an output
Set LED pin to not illuminate LED

Main Program Loop Start
        Read button pin and determine state
                If button not pressed do nothing
                If button pin is pressed, set LED pin to illuminate
                If button has been pressed wait 1 second
                Turn LED Off after 1 second
Go back to Main Program Loop and test for next press
```

This pseudo-code example offers an easy to understand statement of the program structure and once we are satisfied that its logic matches our desired operation, we can easily transform it into an MBasic program, with each line of pseudo-code expanding into one or two lines of real code:

```
;Initialization
;-------------
Input B0    ; Initialize button pin to be an input
```

```
Output B1   ; Initialize LED pin to be an output
Low B1      ; Set LED pin to not illuminate LED

Main
;-----
      If PortB.Bit0 = 0 Then ; Read pin and determine state
            High B1          ; If pressed, LED illuminated
            Pause 1000       ; If button pressed wait 1 second
            Low B1           ; Turn LED Off after 1 second
      EndIf                  ; If button not pressed do nothing
                             ; Go back to Main Program Loop and
GoTo Main                    ; test for next press

End
```

For all but the simplest programs, I start with a high level pseudo-code definition of overall program flow—perhaps aided by a simple flow chart—but at a high level of abstraction. The first pseudo-code draft concentrates on the high level program flow and logic, with subroutines, initialization, input/output concerns being a line or two of pseudo-code. It shows the desired input and output data, even if it is something as simple as "read user's keypad press." I may even write and debug MBasic code implementing the high level design, substituting dummy sub-routines for the detailed ones, and hard-coding user inputs.

Once the high-level program flow is functioning, I write pseudo-code for each main subroutine, followed by an MBasic realization. Each subroutine is written and debugged before starting on the next, to the extent permitted by the program logic. Where one subroutine depends on the next, follow a top down approach.

If you have to boil down writing good code into one rule, it would be:

Think first, code later.

If we are permitted a few more rules, they would be:

- Define the problem, including the "goes intos" and "comes out ofs," that is, the information to go into the program, such as switch readings, sensor readings and the conditions that cause those readings to be generated, and the desired output, such as logic level pin changes, analog outputs through a digital-to-analog converter, and the actions these outputs cause, such as turning on a motor or activating a solenoid.
- Document the problem and the solution and keep the documentation up to date as you develop answers, even if you are programming for personal satisfaction, or education and not with the though of developing a commercial product.
- Think first, code later—the single most important thing for a programmer to do is to resist the siren call of writing code and instead study, understand and plan what to do.

Coding is often the simplest task and can be almost mechanical, if the problem is first properly understood and defined.

- Program modularly, proceeding from the top down, an essential philosophy if you are to efficiently produce readable, stable code. It's possible to program from the bottom up, starting with details and working towards a general structure that fits together the details, but it's never the right way to proceed. And if you do manage to make it work, a change in details can upset the general structure you've cobbled together. Top-down programming is much more tolerant to the inevitable changes that occur as a project progresses. And, write code in modular subroutines, not as one large omnibus program. Code development, maintainability and debugging are all immensely aided by coding in subroutines. As seen in many examples in this book, our modular programming takes the form of simple programs to test concepts, hardware and code, but which are then combined into more complex constructions.

18.8 Inside the Compiler

The process of compiling an MBasic program and having the resultant code programmed into the PIC is illustrated in Figs 18.9 and 18.10. The process is transparent to users, as Basic

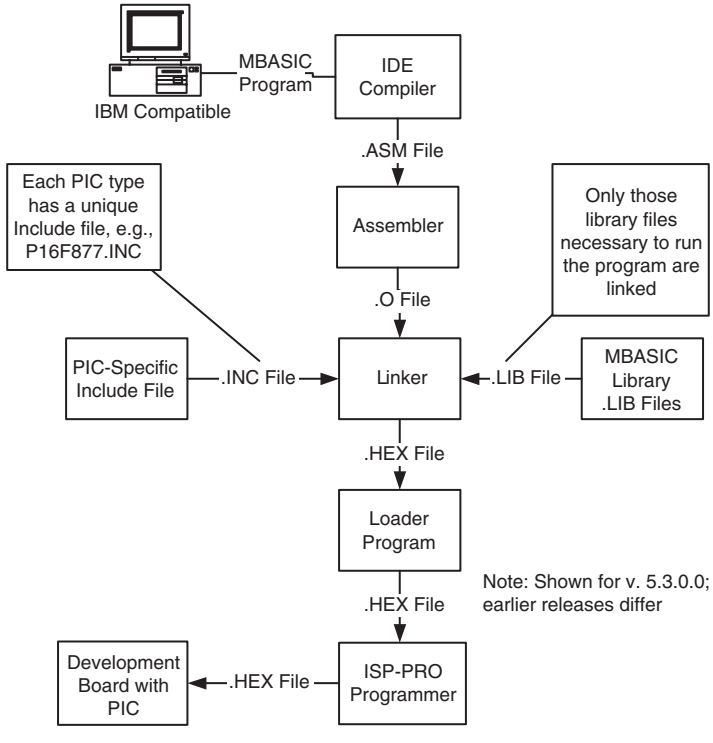


Figure 18.9: Flow During Program Compilation

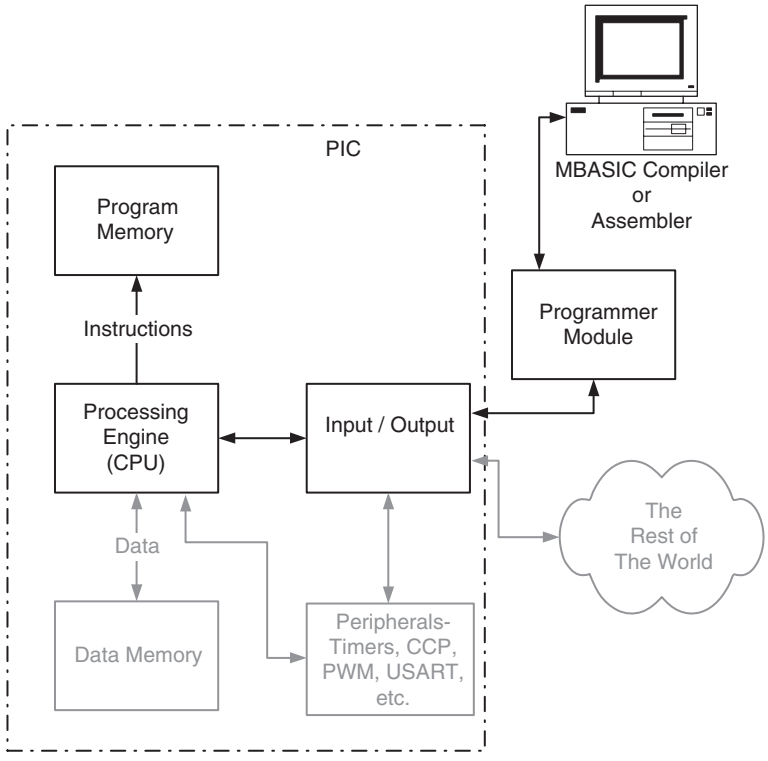


Figure 18.10: PIC to Compiler Flow

Micro's integrated development environment automatically invokes the assembler, the linker and the loader program and deletes any unnecessary intermediate files. We will find reason, however, to examine the intermediate assembler (.ASM) file when we examine how assembler programming meshes with MBasic.

18.8.1 Compiler vs. Interpreter

We've referred to the MBasic software as a "compiler." Since the software runs under Windows on an Intel-compatible PC, and the target is a PIC it should be called a "cross-compiler," as the term "compiler" is usually understood to mean that the target processor is the same as the processor upon which the compiler executes.

A more fundamental question, though, is MBasic a compiled language or an interpreted language? In the purest case, a compiler takes the high level language source code and translates all of it at once to machine code. A pure interpreter, in contrast, translates the high level language statements one at a time. All else being equal, a pure interpreter runs the same source code significantly slower than a pure compiler. However, the interpreter yields a much more compact result, occupying a fraction of the code space in the target machine required for its

compiled equivalent. There are, in the real world, few pure compilers and few pure interpreters, and MBasic falls on the continuum between the two extremes, albeit closer to the interpreter end than the compiler end, a decision necessitated by the small code and variable size in PICs. If you examine the intermediate assembler file output from a simple program, you will find the MBasic statements are converted to tokens and ordered according to Reverse Polish Notation, which will be familiar to those having used an older Hewlett-Packard handheld calculator.

Let's look at a very simple example to illustrate how MBasic converts a statement into RPN tokens. Here's a simple mathematical expression in MBasic:

```
Ax       var       byte
Bx       var       byte
Cx       var       byte
Dx       var       byte
Ex       var       byte
Fx       var       byte
Gx       var       byte

Gx = Ax+(Bx—Cx/Dx)—Ex*Fx
```

The resulting assembler output, extracted from the ASM file, after compiling under MBasic version 5.2.1.1 (removing nonessential lines) from the MBasic is:

```
retlw _VARP32
retlw HIGH 0848h ;AX
retlw LOW 0848h ;AX
retlw _VARP32
retlw HIGH 0849h ;BX
retlw LOW 0849h ;BX
retlw _VARP32
retlw HIGH 084ah ;CX
retlw LOW 084ah ;CX
retlw _VAR
retlw HIGH 084bh ;DX
retlw LOW 084bh ;DX
retlw _DIV
retlw _SUB
retlw _ADD
retlw _PUSH32
retlw _VARP32
retlw HIGH 084ch ;EX
retlw LOW 084ch ;EX
retlw _VAR
retlw HIGH 084dh ;FX
retlw LOW 084dh ;FX
retlw _MULL
retlw _SUB
```

```
retlw _PUSH32
retlw _ADDRESS
retlw HIGH 084eh ;GX
retlw LOW 084eh ;Gx
retlw _LET
```

We won't spend a lot of time on this topic and will save for later discussion several important concerns, but let's go through it quickly. `Retlw` statements (return literal in the w register) return a literal value to the calling function; for example, `retlw` 1 returns the value 1 when it is called. (If you compile this statement under version 5.3.0.0, the `retlw` statements are replaced by chip-specific macro calls, but the code structure remains the same.)

The statement group:

```
retlw _VARP32
retlw HIGH 0848h ;AX
retlw LOW 0848h ;AX
```

can be understood to accomplish placing the value of the MBasic variable `Ax` into a form usable by MBasic's mathematical library.

The statement `retlw _DIV` instructs MBasic's mathematical library to execute the division operator on the two values immediately preceding it.

Based on this understanding, we can simplify the meaning of the assembler code:

```
AX
BX
CX
DX
DIVIDE
SUBTRACT
ADD
EX
FX
MULTIPLY
SUBTRACT
Gx
LET (=)
```

Parentheses may help show how this is evaluated, and it will be immediately clear if you have used an HP calculator with RPN:

```
AX ADD (BX Subtract(CX DX Divide)) Subtract ((EX FX Multiply)) (GX =)
```

Or, `Gx = Ax+(Bx-Cx/Dx)-Ex*Fx`

References

The following two references are long out of print, but may turn up from time to time in used bookstores.

[18.1] *BASIC Reference*, Personal Computer Hardware Reference Library, Third Ed., (May 1984). Any of IBM's BASIC handbooks from the late 1970's to mid '80s are worth acquiring as they document, in IBM's exceedingly thorough fashion, the nuts and bolts of BASIC programming. Obviously some of the information is highly specific to PCs of the era, but a surprisingly high amount of information is directly applicable to MBasic.

[18.2] Nagin, Paul and Ledgard, Henry, *Basic With Style—Programming Proverbs, Hayden Book Company*, Rochelle Park, NJ (1978). This thin volume is packed with more good advice on programming than most books three times its size. It isn't as much a "how to code" book as it is a "how to think about coding" book, and is thus that much more valuable, as "how to code" books are plentiful.

[18.3] Brown, P.J., *Writing Interactive Compilers and Interpreters*, John Wiley & Sons, New York, NY (1979).

[18.4] Aho, A.V., et al, *Compilers Principles, Techniques and Tools*, Addison-Wesley Publishing Co., Reading, MA (1986).

[18.5] *PIC12629/675 Data Sheet*, Microchip Technology, Inc., Document No. DS41190C (2003).

[18.6] *PIC16F87X Data Sheet*, Microchip Technology, Inc., Document No. DS30292C (2001).

This page intentionally left blank

CHAPTER 19

The Basics—Output

A great deal of your work with PICs will involve turning things on and off. The action may be as simple as illuminating an LED to show program status, or as complex as sequencing multiple motors. You may accomplish these actions with the PIC's input/output pins, unaided, or with external electronic or electromechanical devices. The action may require "sourcing" or "sinking" current or voltage. A high state pin sources current into an external load, while a low state output pin receives or sinks current from an external load. In this chapter, we will review a few elementary electronics principles and learn how to use them to allow PICs to control external devices.

This chapter deals with the electronic characteristics of PIC pins as output devices.

Diagrams and discussions in this book assume positive or classical current flow, in which current flows from positive to negative, as shown in Fig. 19.1. Traditional circuit equations, as well as the arrow symbol for diodes and transistors follow this convention as well.

Before building any of the sample circuits please download and read the relevant data sheets from the device manufacturer's Internet website. To keep this chapter at a manageable length, I've had to gloss over many subtleties in the specifications and application of these devices, hitting only the highlights. Careful advance study of data sheets and any associated application notes will reduce the time spent designing and debugging your designs.

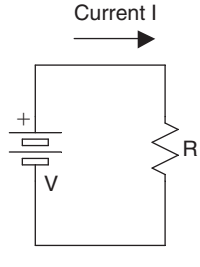


Figure 19.1: Conventional Current Flow

19.1 Pin Architectures

At first glance, Microchip's simplified schematic of the I/O pins may seem confusing. Chapter 3 of the 16F87x Reference, for example, requires ten figures to illustrate the internals of I/O pin construction. At the beginning and intermediate stages of programming with MBasic and concentrating only on the output mode, though, we can simplify things further, reducing the essentials to those of Fig. 19.2. In the 16F87x series, and in other mid-range PICs, when in output mode, pins are connected to a classical complementary metal oxide semiconductor (CMOS) configuration. In some cases, such as for RA0…RA3, Microchip's documents show the CMOS transistors directly; in others, such as RB0…RB3, they are not shown but are imbedded in a logic gate symbol.

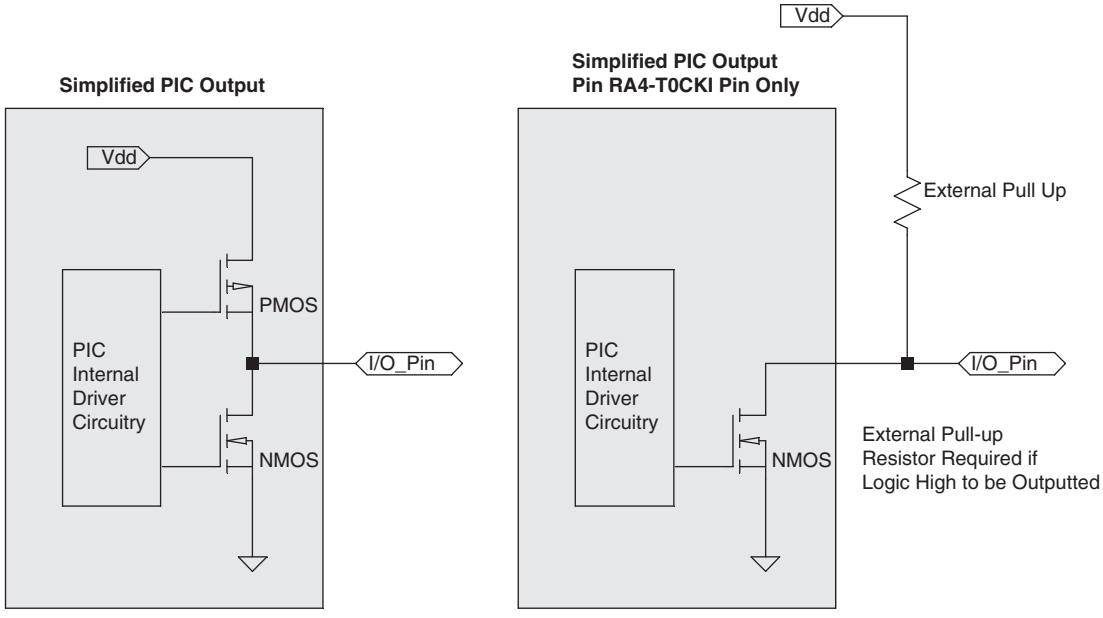


Figure 19.2: Simplified Output Pin

For many purposes, we can regard the PMOS and NMOS transistors of Fig. 19.2 as simply switched resistors; they are either very high resistances, amounting to almost open circuits, or a low value resistor, as illustrated in Fig. 19.3. When the output is low, the pin appears to be a low value resistor, approximately 25 ohms. When the output is high, the pin appears to be the V_{DD} source connected through a resistor of about 85 ohms, as long as the sourced current doesn't exceed 15 mA or so.

When using Basic Micro's development or prototype boards, the 74HC4053 multiplexer needed to permit in-circuit programming adds approximately 50–100 ohms series resistance to pins RB4, RB6 and RB7. In many cases this additional resistance can be ignored.

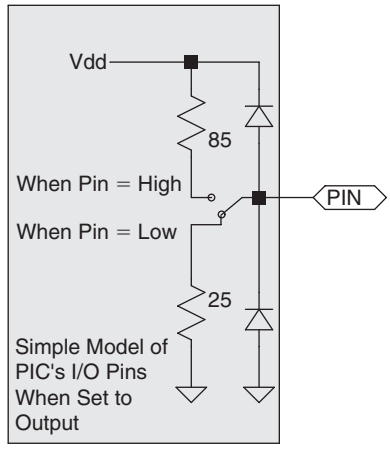


Figure 19.3: For Many Analyses, the Output Pins Appear to be Simple Resistors

One pin, RA4, is different; it is configured as an open drain MOSFET. When set to low, it performs identically with the other pin architectures. However, when set to high, there is no internal connection with V_{DD} and hence it will not directly source voltage. If it's necessary to use RA4 as a sourcing output pin, you can add an external "pull-up" resistor, typically in the range of 470 ohms–4.7 K ohms. The sourced current then comes from the pull-up resistor. Unlike all other pins that cannot exceed V_{DD}, RA4's open drain is rated to 12 volts.

When either sourcing or sinking current, the safe operating limits of the PIC must be observed. The following maximum safe parameters apply to the 16F87x series, and the Electrical Characteristics section of Microchip's data sheet for your target PIC should be consulted. Exceeding these limits may cause damage to the device, or reduce its reliability.

Absolute Maximum Ratings for 16F87x PICs				
Symbol	**Characteristic**	**Maximum Value**	**Units**	**Conditions**
V_{OD}	Open drain high voltage	14	V	Applies to pin RA4 only
	Voltage on any pin with respect to Vss	−0.3 V to V_{DD}+0.3 V	V	
	Total chip supply current into V_{DD} supply pin	250	mA	
	Total chip current out of V_{SS} pin	300	mA	
I_{OK}	Output clamp current (Vf < 0 or Vf > V_{DD})	±20	mA	
	Maximum output current sunk by any I/O pin	25	mA	

Absolute Maximum Ratings for 16F87x PICs				
Symbol	**Characteristic**	**Maximum Value**	**Units**	**Conditions**
	Maximum output current sourced by any I/O pin	25	mA	
	Maximum current sunk by PortA, PortB and PortE, combined	200	mA	PortD and PortE are not implemented on 16F873/876 devices
	Maximum current sourced by PortA, PortB and PortE, combined	200	mA	PortD and PortE are not implemented on 16F873/876 devices
	Maximum current sunk by PortC and PortD, combined	200	mA	PortD and PortE are not implemented on 16F873/876 devices
	Maximum current sourced by PortC and PortD, combined	200	mA	PortD and PortE are not implemented on 16F873/876 devices

Before starting our circuit discussion, let's review these maximum ratings.

Open drain high voltage—RA4 is unique and omits the internal PMOS transistor connection to V_{DD}. V_{OD} is maximum safe voltage that may be applied to RA4.

Voltage on any pin with respect to V_{SS}—In the normal circuit, V_{SS} will be at ground potential. Your circuit should be designed so that when in output mode, pins will not be taken more than 0.3 V negative with respect to ground nor more than 0.3 V above the PIC's positive supply voltage, V_{DD}. Should these voltages be significantly exceeded, the protective diodes shown in Fig. 19.2 will start to conduct, potentially causing the pin or chip maximum current limit to be exceeded, unless otherwise current limited.

Total chip supply current into V_{DD} supply pin—In addition to sourcing current limits on individual pins, this parameter establishes a global maximum available current for the entire PIC. It is, with negligible error, the sum of all pin sourcing currents.

Total chip current out of V_{SS} pin—In addition to sinking current limits on individual pins, this parameter establishes a global maximum for the entire PIC. It is, with negligible error, the sum of all pin sinking currents.

Output clamp current ($Vf < 0$ or $Vf > V_{DD}$)—If a pin is taken above V_{DD} or below ground, it must be current limited, commonly with a series resistor, so that the output clamp current is not exceeded.

Maximum output current sunk by any I/O pin—The maximum safe sinking current when a pin is low. Sinking current is not internally limited and is governed by the external circuit parameters.

Maximum output current sourced by any I/O pin—The maximum current that may be safely sourced by a high pin. Internal circuitry limits sourcing current to approximately 25–30 mA so it is safe, but not good design practice, to operate an output pin into a short circuit.

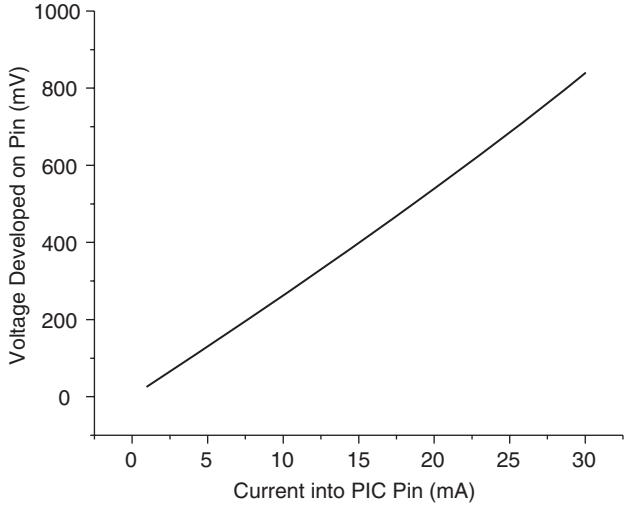
Maximum current sunk by PortA, PortB and PortE, combined—Another composite limit, applying to sinking current by all Ports A, B and E pins combined.

Maximum current sourced by PortA, PortB and PortE, combined—Another composite limit, applying to sourcing current by all Ports A,B and E pins combined.

Maximum current sunk by PortC and PortD, combined—Another composite limit, applying to sinking current by all Ports C and D pins combined.

Maximum current sourced by PortC and PortD, combined—Another composite limit, applying to sourcing current by all Ports C and D pins combined.

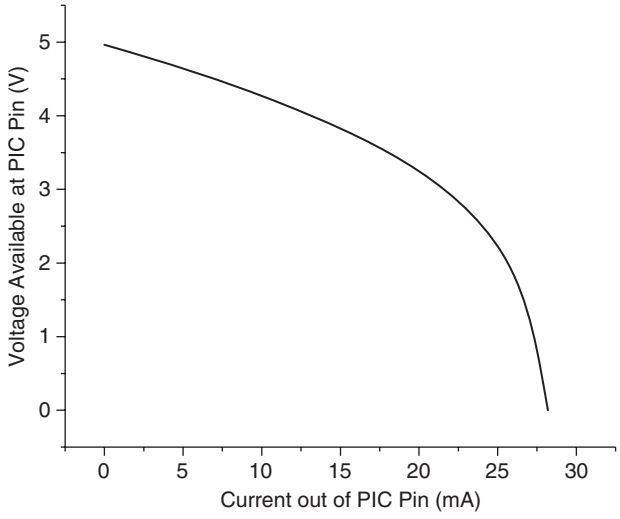
A high output will source between 25 and 30 mA into a short circuit indefinitely, but when sinking current, the maximum safe current rating must be observed. Figures 19.4 and 19.5 illustrate the typical voltage versus current relationship for both sourcing and sinking current. Also remember that when using Basic Micro's 2840 Development Board, pins RB4, RB6 and RB7 are switched through the 74HC4053 multiplexer which has a 25 mA maximum current limit.



Figures 19.4: Typical E vs. I for Sinking Current

One final bit of terminology and we'll be onto circuitry. Figure 19.6 shows three possible switching configurations. For clarity, the drawing shows a mechanical switch. We, of course, will use a variety of electronic substitutes.

Low side switching—The switch is between the load and ground. When closed, both sides of the switch are at ground potential.



Figures 19.5: Typical E vs. I for Sourcing Current

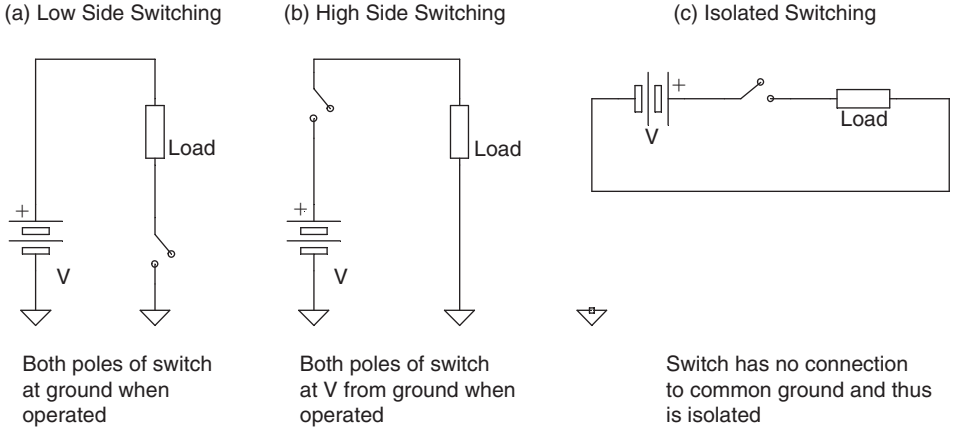


Figure 19.6: Possible Switching Configurations

High side switching—The switch is between the voltage being switched and the load. When closed, both sides of the switch are at the switching voltage.

Isolated switching—There is no common connection between the circuit being switched and the controlling PIC. Many devices suitable for isolated switching also work for low side or high side switching.

19.2 LED Indicators

In learning how to program a PC in a high level language, the traditional first program writes "Hello World" to the screen. Since PICs don't have a screen, the first MBasic program

traditionally blinks an LED. We'll do that idea one better, building up to four states with one LED and one PIC pin. But, first we'll start with two LEDs and two pins as shown in Fig. 19.7.

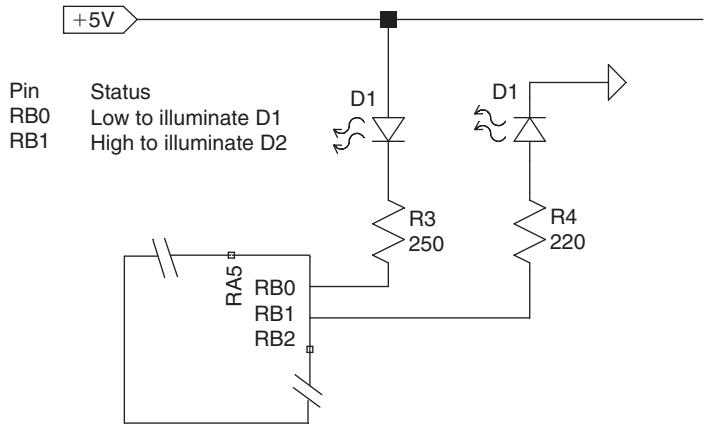


Figure 19.7: LED Connections

```
i       var     byte
For i = B0 to B1     ;LEDs are on B0...B1
   Output I    ;so we make them outputs
Next

Main
        For i = B0 to B1     ;some will illuminate with a low
          Low i
        Next
        Pause 1000
        For i = B0 to B1     ;some will illuminate with a high
                High i
        Next
        Pause 1000
GoTo Main

End
```

Program 19.1

The code is straightforward; after declaring our index variable `i`, we set pins RB0…RB1 to be outputs with the `Output` procedure inside a `For...Next` loop. The `Output(i)` procedure takes the pin address as its argument, with `i` ranging from `B0` to `B1`, predefined in MBasic as the numerical addresses of pins RB0…RB1. We then set these pins to alternate between low and high, with 1 second (1,000 milliseconds) in each state using MBasic's `High` and `Low` procedures inside two `For...Next` loops, each followed by a `Pause(1000)` procedure. An endless loop (`Main...GoTo Main`) causes the alternating high/low steps to be repeated.

D1 illuminated when RB0 low—When RB0 goes low, current from the +5 V supply goes through series combination of LED D1, resistor R3 and the internal resistance of RB0. LEDs may be regarded as a device that have approximately a constant voltage drop for typical operating currents in the range from 1 mA to tens of mA. Figure 19.8 illustrates, for current levels between 1 and 50 mA, the LED's voltage drop is between 1.7 and 2.2 V. With only a small error, we may regard the LED as a constant voltage device, with about a 2 V drop. (There's a slight difference in voltage drop for different output colors, but for almost all red, green and yellow LEDs, we may calculate the current limiting resistors assuming a 2 volts drop.)

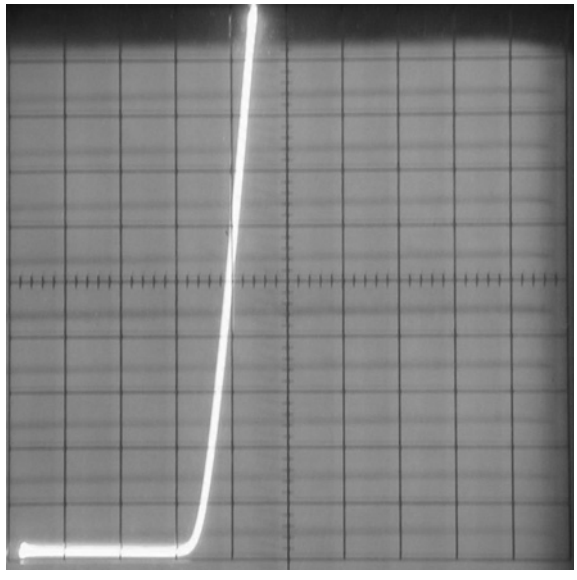


Figure 19.8: E/I Curve Trace of Red LED. Horiz: 0.5 V/div Vert: 5 mA/div

We may now solve the current loop equation for the circuit involving D1, remembering that a low pin is functionally equivalent to a 25 ohm resistor:

$$5V = 2V + 250I + 25I$$

rearranging

$$5V - 2V = 250I + 25I \quad \text{so} \quad 3V = 275I$$

or

$$I = \frac{3}{275} = 10.9 \text{ mA}$$

where I is the current through the LED and series resistor.

More often, we wish to calculate the series current limiting resistor needed for a particular LED current *I* (in mA) where the LED is on when the PIC driving pin is low:

$$R_3 = \frac{3000}{I_{mA}} - 25$$

We fudged a bit by assuming the voltage drop across D1 is constant regardless of current, but these simple equations will be within 10% of a more detailed calculation, more than accurate enough for determining the current through an LED indicator.

D2 Illuminated when RB1 high—When RB1 goes high, current from the V_{DD} (the +5 V supply in Basic Micro's development boards) goes through the series combination of LED D2, resistor R4 and the internal resistance of RB1. This is only a slight rearrangement of our earlier analysis of D1, with the internal equivalent resistance of the high pin being 85 ohms. Hence,

$$5V = 2V + 220I + 85I$$

rearranging

$$5V - 2V = 220I + 85I \quad \text{so} \quad 3V = 305I$$

or

$$I = \frac{3}{305} = 9.8 \text{ mA}$$

where *I* is the current through the LED and series resistor.

Or, to calculate the series current limiting resistor where the LED is on when the PIC driving pin is high (in mA):

$$R_4 = \frac{3000}{I_{mA}} - 85$$

In addition to the constant voltage drop fudge, this analysis assumes a high pin is modeled accurately as an 85 ohm resistor in series with V_{DD}. As Figure 19.5 shows, this assumption starts to fail as the sourced current exceeds 15 mA and the plot of I versus E diverges from a straight line.

Two LED's on one pin—We can connect two LEDs to one pin using the circuits we just developed as shown in Fig. 19.9. The current for each LED is calculated using the same equations for individual pin connections.

Four states from one pin—Using the connection of Fig. 19.10, it's possible for one pin to produce four states in a 2-pin dual LED. (Most dual LEDs have two pins, but some dual LEDs have three pins permitting the circuit of Fig. 19.9 to be used.) Fairchild's MV5491A two-pin dual LED is configured as a red and green LED in anti-parallel whereby current flow in one direction provides red light while the opposite direction provides green light.

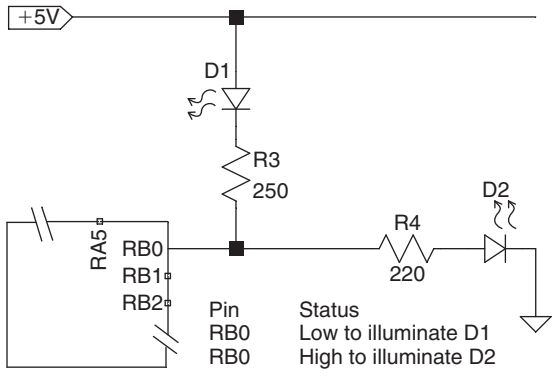


Figure 19.9: Two LEDs on One Pin

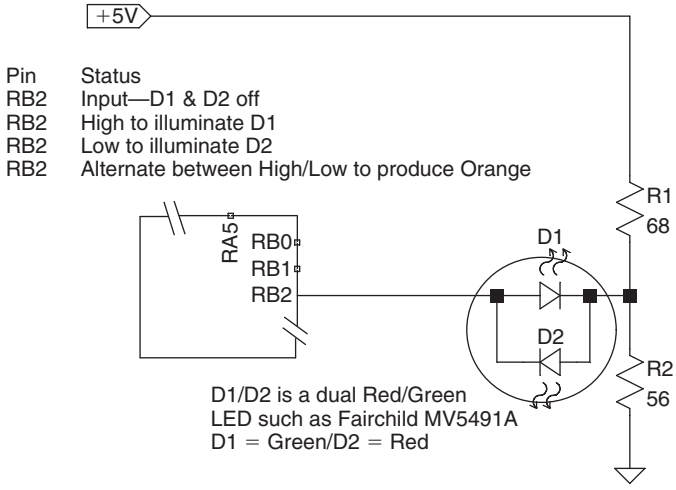


Figure 19.10: One Pin, Four States

In the circuit of Fig. 19.10 when RB2 is high, current flows from RB2 through D1 and R2. When RB2 is low, current flows from the +5 V supply through R1, D2 and is sunk at RB2. The suggested resistors yield 6.9 mA current for the green LED (D1) and 8.6 mA for the red LED (D2).

It's possible to get a third color out of this design as well. By rapidly switching between the red and green LEDs, the eye perceives orange. The following code fragment will accomplish this, switching at approximately 100 Hz.

```
Main
      High B2
      Pause 5
      Low B2
      Pause 5
GoTo Main
```

Finally, if a fourth condition, LED off, is desired, switch RB2 to input. As an input, RB2 is essentially an open circuit, and neither D1 nor D2 will be illuminated. This trick will not work with the configuration of Fig. 19.9, as both diodes will illuminate in that state.

Program 19.2 exercises all four states of Fig. 19.10's dual LED.

```
;Four states from one dual color LED and one PIC Pin
;Assumes bi-color LED on RB2
;With voltage divider circuit

i       Var          Byte
Main
        High B2         ;Green
        Pause 1000
        Low B2                  ;Red
        Pause 1000

        For i = 0 to 255        ;Orange
                High B2
                Pause 5
                Low B2
                Pause 5

        Next

        Input B2                ;no illumination
        Pause 1000

GoTo Main

End
```

Program 19.2

Program 19.2 first illuminates the green LED for 1 second followed by red for 1 second, followed by 2.5 seconds of orange when both the red and green diodes are sequentially active for 5 ms. Finally, the diode is dark for 1 second.

19.3 Switching Inductive Loads

Stepper motors and relays are common inductive loads switched by PICs. Consider the circuit shown in Fig. 19.11 that controls a small Omron G2RL-24 relay.

From introductory circuit theory, we know that when current flows through an inductor, energy is stored in its magnetic field. When the circuit is switched off, the stored energy must go "somewhere." What happens, of course, is the collapsing magnetic field causes a voltage spike—hundreds of volts even in a the small G2RL-24 relay—at the collector of Q1. In the absence of protective circuitry, Q1 will temporarily break down when the spike exceeds its V_{CEO} rating (40 V for a 2N4401) and the stored energy is dissipated in Q1. Even if Q1 isn't

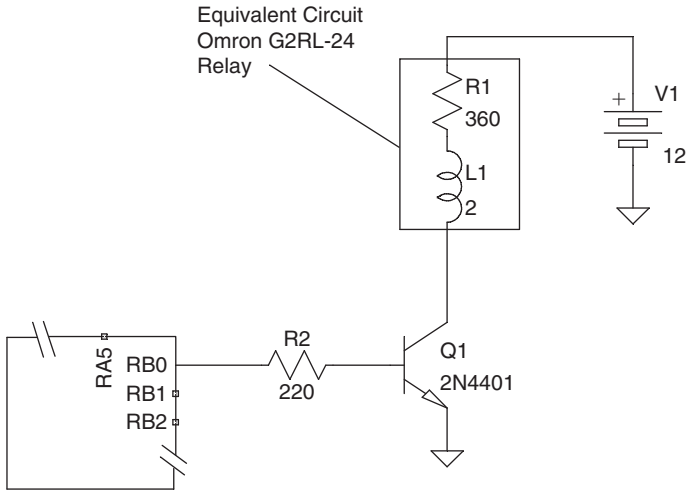


Figure 19.11: Switching an Inductive Load

damaged by the repeated over-voltage breakdown, good design practice says that we should limit the over voltage to safe limits. Fortunately, as shown in Fig. 19.12 it's easy to add a protective diode. D1 is called a "clamping diode" because it clamps the voltage spike. You may also see it referred to as a "snubbing diode" or "snubber."

When the magnetic field collapses as Q1 is switched off, the induced voltage causes diode D1 to conduct, and the stored energy is dissipated in the internal resistance of the inductor, R1 in Fig. 19.12, and an optional external resistor, R3.

As with many things in electronics, there is a tradeoff here. The current resulting from the magnetic field decay doesn't drop to zero instantaneously, but rather as a function of the total

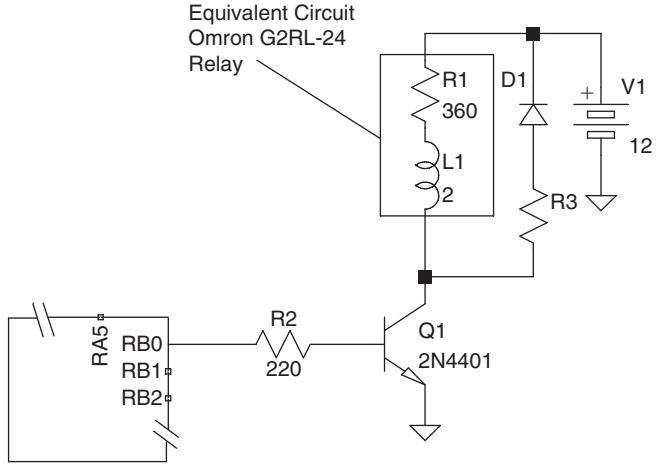


Figure 19.12: D1 and R3 are Added Protective Components

resistance in the D1-R3-R1 circuit. The faster we make the current drop to zero (smaller the series resistance) the higher the voltage spike. Conversely, if we limit the voltage spike to its minimum level by setting R3 at zero ohms, we find the longest time for the induced current to decay. Figure 19.13 illustrates how the peak voltage spike and current decay times interact for the circuit of Fig. 19.12. If you are familiar with elementary calculus, this relationship is obvious since the voltage E across an inductor of value L Henries is proportional to the time rate of change of the instantaneous current i through the inductor:

$$E = L\frac{di}{dt}$$

A faster decay (greater *di*/*dt*) means more induced voltage and vice versa. If we are concerned with the relay release time, we want the current to decay below the release current as quickly as possible. This suggests a higher series resistor, perhaps with a Q1 possessing a higher V_{CEO} to accept the resulting higher voltage spike. Perhaps more of a concern exists with when driving a stepper motor. We wish the magnetic field to collapse as quickly as possible when current is removed from a winding, particularly if we are interested in running the motor near its maximum steps per second rating.

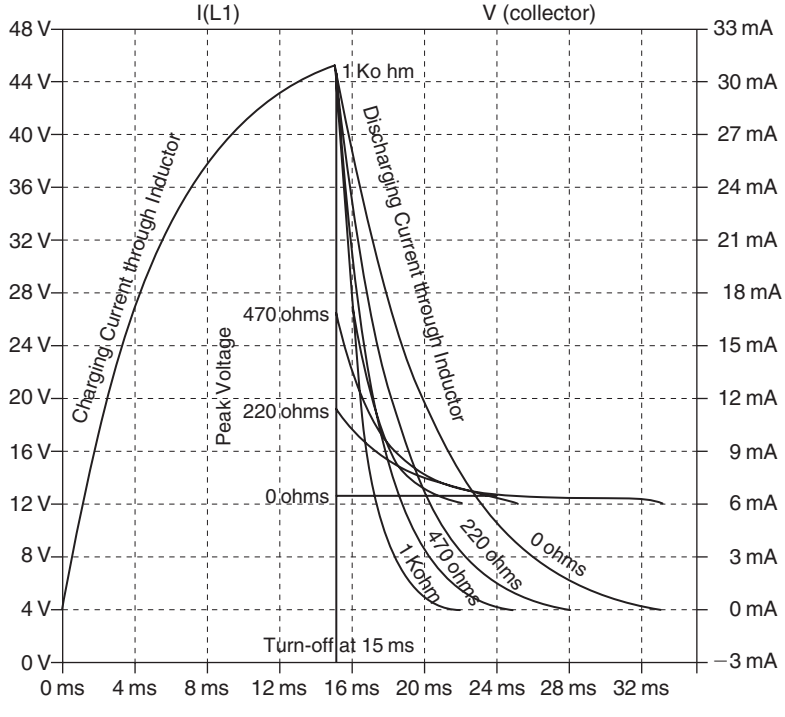


Figure 19.13: Shutdown at 15 ms, Current and Voltage vs. Clamping Circuit Resistance

For critical applications, and particularly for stepper motors, the clamping diode should be a fast switching device, such as a Schottky diode. Alternatively, it is possible to use a zener diode, set to avalanche upon turn off. By delaying the onset of current flow until the zener diode avalanches, significantly faster decay is possible. (The zener is in series with D1, polarized so that D1 prevents forward current flow through the zener.) We'll look at stepper motor driving circuits in detail in a later chapter.

It is possible to calculate the inductive spike level and decay time analytically, but it's much easier to use a SPICE circuit simulation program such as Linear Technology's LTSpice.[Ref 4]

The remainder of this chapter won't mention inductive spike protection, unless it is appropriate because of device characteristics.

19.4 Low Side Switching

19.4.1 Small NPN Switch

Figure 19.14 depicts a simple low side switch. When RB0 is high, Q1 is forward biased into conduction and current flows through the load. Let's work through a few design concerns with this simple circuit. We will assume the load is a 100 ohm resistor and V is 12 volts. Hence, the current being switched is 120 mA. We'll treat the current through this circuit as a constant and ignore the voltage drop across the switching device to simplify our calculations. Since our switch circuits will operate with a voltage drop of well under 0.5 V, our simplifications will not introduce appreciable error.

Voltage rating—When RB0 is low, Q1 appears as an open circuit and thus has the full load supply voltage V across it. In a transistor data sheet, the maximum voltage that may safely be applied in this mode is V_{CEO} or maximum collector to emitter voltage, base open. Our

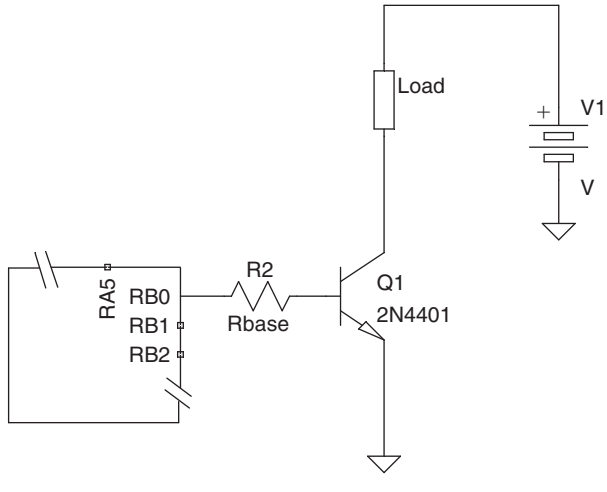


Figure 19.14: 2N4401 NPN Low Side Switch

particular device, a 2N4401 is rated for 40 V V_{CEO}. It should be safe to use it up to about 25 volts, applying a reasonable safety margin to the rated value. Our 12 V switching example will be well within Q1's ratings.

Leakage current—When the 2N4401 is cut off—that is, the base voltage is less than about 0.4 V, some leakage current, I_{CEX}, will still flow through the device's collector. I_{CEX} is rated not to exceed 0.1 μA in the 2N4401, a negligible value in the context of our circuits.

Saturation voltage—When the base drive is sufficient to saturate a bipolar transistor, the voltage drop between the collector and emitter is approximately a constant, referred to in data sheets as $V_{CE(SAT)}$, 0.4 V for a 2N4401 at current levels near 100 mA.

Collector current and device power dissipation—The 2N4401 has a maximum continuous collector current rating of 600 mA, and a maximum power dissipation rating of 625 mW at room temperature (25°C). The saturated collector voltage $V_{CE(SAT)}$ is 400 mV at 150 mA and 750 mV at 500 mA. The thermal resistance junction to ambient $R_{\theta JA}$ is 200ºC/watt and the maximum junction operating temperature is +150°C.

We'll assume adequate base drive to saturate Q1, hence we expect the collector voltage at 120 mA to be 400 mV or less. We'll also assume Q1 is to be on continuously—continuously in this context means long enough for thermal equilibrium to be reached, a matter of a few seconds for a 2N4401 size device. Hence:

The device dissipation will be 120 mA × 400 mV, or 48 mW.

The junction temperature rise over ambient will thus be 200°C/watt × 0.048 watts, or 9.6°C. Assuming the ambient air temperature is 120°F (49°C), the maximum junction temperature will thus be 48° + 9.6° = 57.6°C.

To determine case temperature, we use the thermal resistance junction to case $R_{\theta JC}$ specification, 83.3ºC/ watt. The case will thus be at 83.3ºC/watt × 0.048 watts, or 4ºC above ambient temperature. Our design thus is well within the safe operating parameters of the 2N4401.

If the 2N4401 is being cycled off and on at a rapid rate, the duty cycle will enter into certain of these ratings. For example, suppose the 2N4401 is driving a multiplexed LED display, on for 2 ms and off for 8 ms, for a duty cycle of 0.20. The average power dissipation of Q1 will thus be 20% of the peak power and the permissible peak power dissipation limit may be as much as five times the continuous value. Of course, for this averaging effect to work, the on time must be short compared with the time it takes for the device to reach thermal equilibrium.

Base current drive—As a rough approximation, we may regard Q1 as a current operated switch—that is, for every milliampere of current we wish to be sunk by Q1's collector, we must inject into the base a certain current level. (This is a highly simplified approximation of semiconductor operation, but adequate for our purpose.) The ratio of collector current to base current is known as h_{FE} or "DC current gain." The DC current gain varies from device type to device type, is not well controlled from example to example of the same device type

and, finally, varies with current even for a particular transistor. 2N4401 devices, for example, have an h_{FE} that varies from 20 to 300, depending on the collector current.

If we are not concerned with switching time, or power minimization, the simplest design approach is to assume the worst case h_{FE} and design accordingly. To sink 120 mA, for example, since the minimum specified h_{FE} at 100 mA is 100, the target base current should be 1.2 mA. However, we note that at both 500 mA and 10 mA collector currents, the minimum h_{FE} drops to 40. Hence, as a matter of perhaps excessive caution, and to ensure Q1 is driven well into saturation, we will design for h_{FE} of 40, representing 3 mA base current.

The base to emitter junction voltage, V_{BE}, for a 2N4401 is specified at 750 mV for a base current of 15 mA and collector current of 150 mA, so we will use this value in calculating the base resistor, R2. (Since the base to emitter junction is modeled as a forward biased silicon diode, 700 mV is a commonly used rough estimate for the base to emitter voltage over a wide range of base currents for all silicon bipolar junction transistors.) R2's value (neglecting the PIC's approximately 85 ohm series resistance when sourcing current) is thus:

$$R2 = \frac{5V - 0.750\,V}{0.003A} = 1.4\text{ K ohm}$$

Switching Speed—We've alluded to Q1's switching speed concerns several times in our design. If we are switching an LED, or relay or stepper motor, these problems are unlikely to concern us. However, there are times where it is critical to switch a load as fast as possible. Figure 19.15 shows what happens when very short switching intervals are used in the circuit of Fig. 19.14. RB0 emits a fast rise and fall 200 ns wide pulse and Q1 turns on with less than 20 ns delay. However, when RB0 goes low, Q1 exhibits nearly 500 ns turn off delay. The turn-off delay results from the "stored charge" effect, where excess minority carrier charge is stored in the base region of the transistor junction structure and must be removed before the transistor turns off and collector current ceases. We assume that anyone desirous of switching speeds in the sub-microsecond range knows about stored charges and the mitigating techniques to deal with the problem.

One final point should be noted with the bipolar transistor design of Fig. 19.14—the voltage V being switched is immaterial. Of course, Q1 must be rated to withstand the voltage, but with a suitable transistor, the circuit of Fig. 19.14 could switch 500 volts as easily as it switches 5 volts. The current required to saturate or cut off Q1 is not affected by the voltage it switches.

19.4.2 Small N-Channel MOSFET Switch

At the risk of considerable oversimplification, Q1 in Fig. 19.14 may be thought of as a current controlled switch; current injected into the base causes the collector to be pulled close to ground potential. There is a similar voltage controlled switch, the MOSFET, whereby voltage

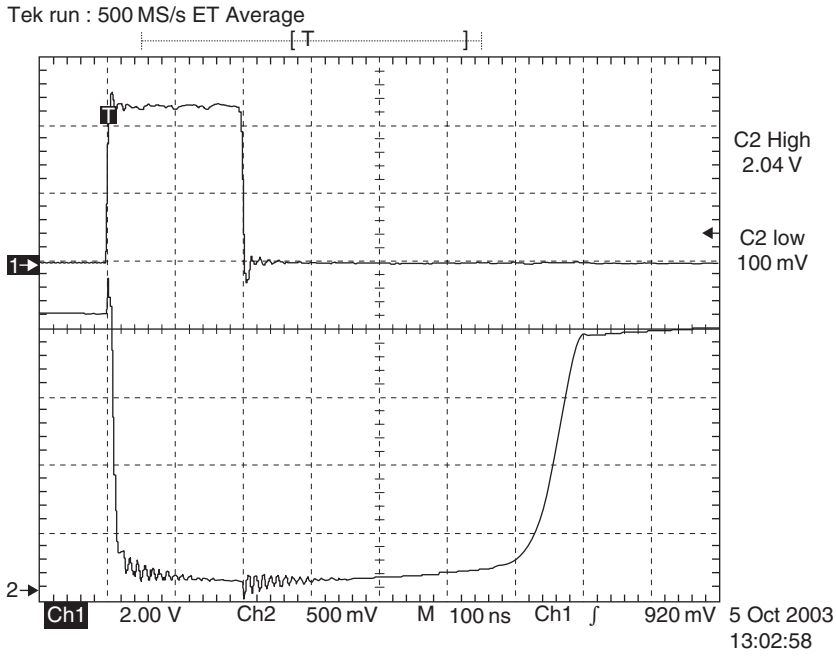


Figure 19.15: 2N4401 Switching Time Ibase = 6 mA, Ic = 40 mA; Ch1: PIC Pin to Base Drive; Ch2: 2N4401 Collector

applied to the device's gate causes the drain voltage to be pulled close to the source, or ground potential in a low side switch.

Figure 19.16 is the MOSFET counterpart of Fig. 19.14. A 2N7000 MOSFET compares favorably with the 2N4401 in the maximum permissible voltage, with a 60 V rating. However,

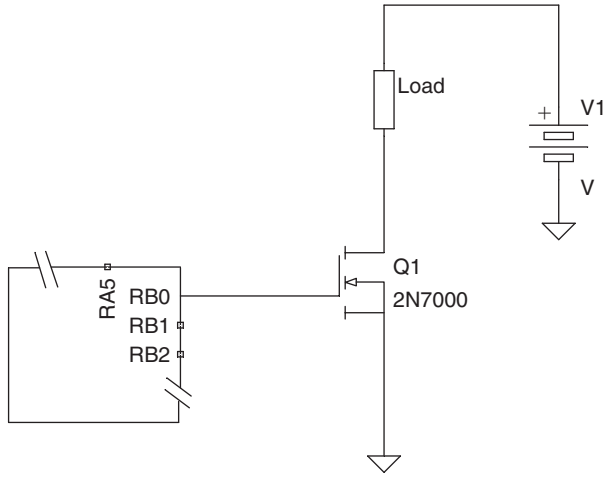


Figure 19.16: 2N7000 NPN Low Side Switch

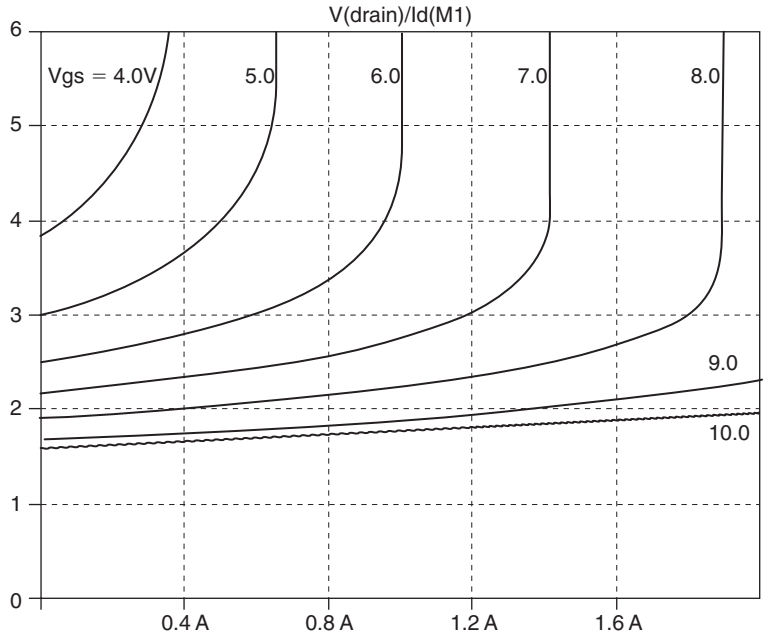


Figure 19.17: 2N7000 Predicted On-resistance Variation with Gate Voltage and Drain Current

the 2N7000 is rated at 200 mA maximum continuous current and 500 mA maximum pulsed current with a total device maximum dissipation of 400 mW. Let's look at the areas of difference between the MOSFET and NPN bipolar transistor.

When saturated, a MOSFET acts like a low value resistor between the drain and source, referred to $R_{DS(ON)}$ with the corresponding voltage between the drain and source determined by the product of the drain current I_D and $R_{DS(ON)}$. Recall that in the 2N4401, the corresponding voltage $V_{CE(SAT)}$ is approximately a constant value over a wide current range.

The relationship between $R_{DS(ON)}$ and the gate voltage is, as illustrated in Fig. 19.17, complex. The point to be taken away from Fig. 19.17 is that since we can drive Q1's gate only to +5 v with a high on an output pin, we exit the saturation region with only modest drain current.

Let's run through the same 120 mA sink design we did for the 2N4401. We've already determined that the 2N7000 meets our open circuit voltage requirements and that 120 mA is less than the maximum permissible continuous drain current. With a gate drive of +5 V and 120 mA drain current, Fig. 19.17 shows $R_{DS(ON)}$ will be about 3.2 ohms. Since V_{DS} is the IR drop across $R_{DS(ON)}$, we may calculate it as 0.120 A × 3.2 ohms, or 0.38 volts, very similar to our 2N4401 NPN bipolar transistor design.

The power dissipated in Q1 equals $I_D \times V_{DS}$, or 0.38 volts × 0.120 mA or 46 mW, almost identical in value with the 48 mW we found for the 2N4401 bipolar switch and well within the device ratings. The 2N7000's thermal resistance junction to ambient $R_{\theta JA}$ is 312.5ºC/watt and

the maximum junction operating temperature is +150ºC. The temperature rise at the case will thus be 312.5ºC/watt × 0.046 watt, or 14.4ºC. Assuming the ambient air temperature is 120ºF (49ºC), the maximum junction temperature will thus be 48ºC + 14.4ºC = 62.4ºC, all quite acceptable values.

If we were to repeat this series of calculations for, say a 400 mA load, we will find $R_{DS(ON)}$ is 3.6 ohms, V_{DS} is 1.44 V and Q1's dissipation is 576 mW, well over the maximum permissible value for a 2N7000. The problem is that 5 V is inadequate gate voltage to fully turn the MOSFET at 400 mA.

When looking at nanosecond switching with a 2N4401, we found significant turn-off problems due to stored charge. As Fig. 19.18 shows, both the turn on and turn off times for a 2N7000 are quite respectable. But, a close examination of the leading edge of the PIC output foreshadows the main difficulty of driving MOSFETs, gate charge. The small plateau or kink in the rise time of the PIC output reflects the fact that the gate of a MOSFET behaves like a nonlinear capacitive load to its driving circuitry. Accordingly, the switching time performance is defined by the ability of the driving PIC to change the gate voltage. The 2N7000's gate, assuming a 5 V gate drive and 120 mA load, looks approximately like a 200 pF capacitor. We will examine MOSFET gate driving problems when we look at a higher power device, the IRF510. For now, we simply note that a PIC is capable of directly driving a 2N7000 to quite respectable switching speeds.

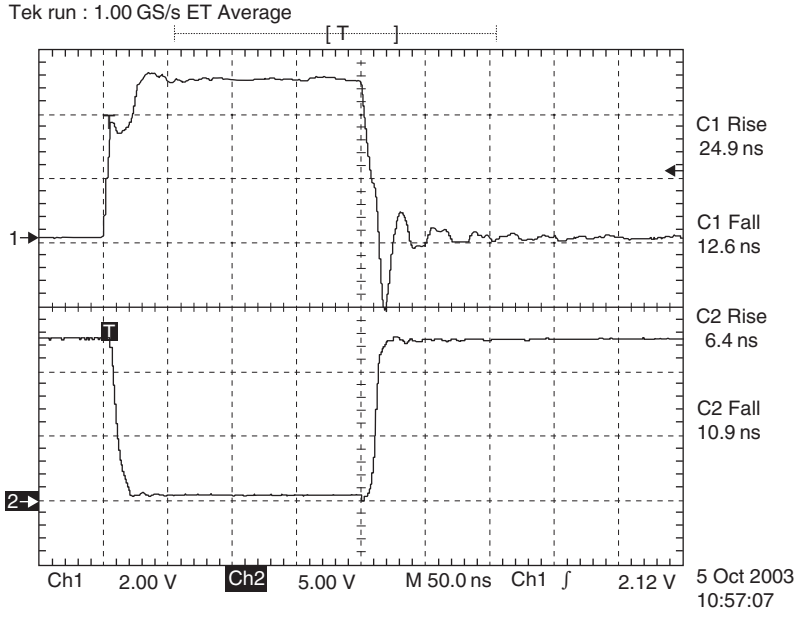


Figure 19.18: 2N7000 Driven by PIC Turn-on/Turn-off Speed Ch1: PIC Output; Ch2: 2N7000 Drain

19.4.3 High Power Bipolar Low Side Switching

Both the 2N4401 and 2N7000 are low power devices, good for continuous currents of a few tenths of an ampere at most. Suppose we wish to switch an ampere or two with a bipolar transistor. In most instances, we will not be able to build a high power switch by simply replacing the 2N4401 with a high power device, as the 25 mA or so maximum current output of a high PIC pin isn't enough base drive to reliably saturate a simple bipolar power transistor.

Consider Fig. 19.19. The TIP31's maximum current rating is 3.0 A, but at this level the minimum guaranteed h_{FE} is only 10, so 300 mA base drive would be required for saturation. Even at 1.5 amperes collector current the PIC is short of achieving full saturation with the maximum possible PIC output current as its base drive. (This, of course, would require h_{FE} to be at least 60.) Figure 19.20 shows the V_{CE} is 920 mV, instead of the expected 300 mV shown in the TIP31's data sheet for 1.5A collector current.

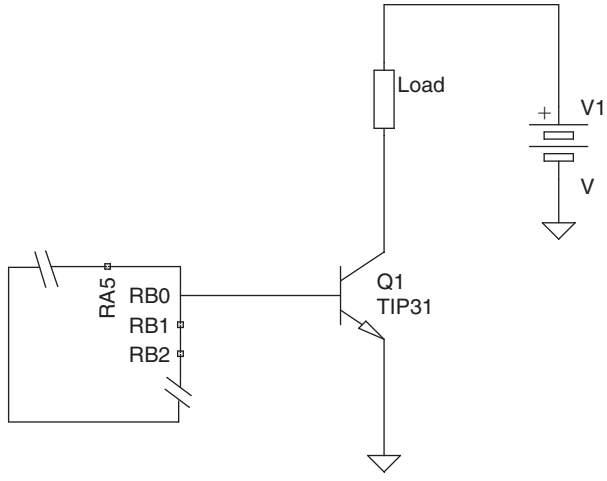


Figure 19.19: High Current Switching with TIP31

A Darlington transistor solves our base drive problem. As reflected in Fig. 19.21, a Darlington transistor uses one transistor as a emitter follower current booster to drive the base of the second transistor. The Darlington configuration may use two separate devices or, as in the case of the TIP120, the driver transistor may be integrated onto the same die as the power transistor. The composite h_{FE} of the Darlington pair is approximately the product of the h_{FE} of the two transistors. Since the driver transistor doesn't handle high currents, its h_{FE} may be very large, thus ensuring a composite h_{FE} in the thousands. For a collector current of 1A, the TIP120's h_{FE} is approximately 3300, permitting collector saturation with a base current of as little as 300 μA.

So far, so good. The TIP120 is rated at 60 V V_{CE}, 5A maximum collector current I_C and 65 watts dissipation, with an appropriate heat sink. The price to be paid for the extra h_{FE}, though, is found in the saturation voltage, $V_{CE(SAT)}$. For an I_C of 3A, $V_{CE(SAT)}$ is 2.0 V while at 5A, $V_{CE(SAT)}$ is

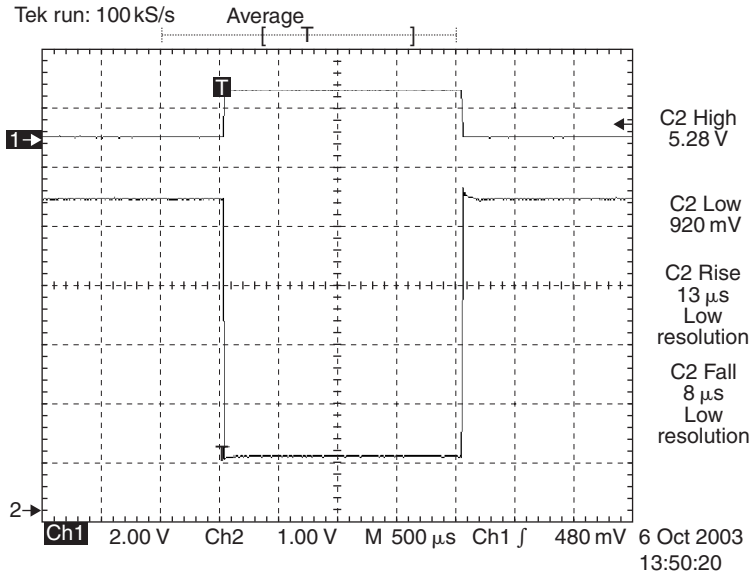


Figure 19.20: TIP31 at 1.5A IC Ch1: Base; Ch2: Collector

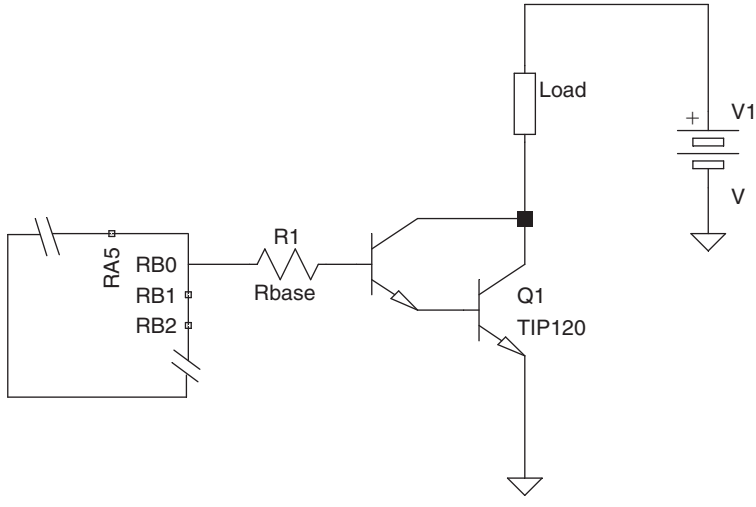


Figure 19.21: Switching with TIP120 Darlington

4.0 V. A quick glance in Fig. 19.21 reveals why $V_{CE(SAT)}$ is so poor. The driver transistor relies upon $V_{CE(SAT)}$ as its source of current to supply base drive to the output transistor.

Recall that the base of the output transistor's base must be at about 0.7 V above its emitter before base current flows. And, the driver transistor itself introduces another 0.4 V or so minimum voltage drop even if it is saturated. Hence, if $V_{CE(SAT)}$ drops below 1.1 V, the driver transistor no longer can supply base current to the output stage.

Another area we expect to see a performance problem with the Darlington is turn off time as power transistors are slower than the small switching devices we have looked at so far. Figure 19.22 confirms our pessimism, as the TIP120's turn off time is nearly two microseconds. However, we should remember that for many applications, 2 μs turn off time is more than adequate.

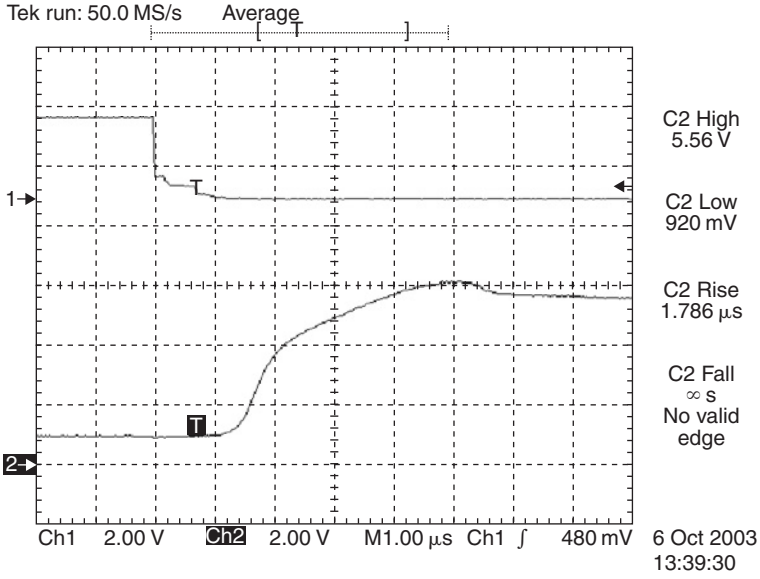


Figure 19.22: TIP120 Turn off Time Ch1: PIC Output; Ch2: TIP120 Collector

Calculating the base series resistor Rbase, and the other parameters follow the approach used for the 2N4401 and won't be repeated. However, attention must be paid to proper heat sink selection for power devices. Although the TIP120 is rated at 65 watts dissipation, that value presumes an adequate heat sink. Absent a heatsink, its rating is only 2 watts. Operating without a heatsink, in continuous operation, a TIP120 should not sink more than 1A or so—and even that is pushing it.

19.4.4 High Power MOSFET Low Side Switching

19.4.4.1 IRF510 Switch

Just as there are high power relatives of the 2N4401, the 2N7000 has many big brothers as well. We'll quickly examine one of the original power MOSFETs but then turn our attention to a new power MOSFET device that solves many of the shortfalls of earlier devices.

The IRF510 is one of the earliest inexpensive power MOSFETs and is still in production. It's rated at 100 V V_{DS}, maximum I_D of 5.6 A and a power dissipation of 43 watts, assuming proper heat sinking. It is supplied in a TO220 package. $R_{DS(ON)}$ is under 1 ohm at room temperature.

(Modern devices are in the tens of milliohm range, but we'll stick with the IRF510 to illustrate the point on gate voltage concerns.)

When we substitute the IRF510 for the 2N7000, as shown in Fig. 19.23 again we are limited to 5 V gate drive. If we consult the $R_{DS(ON)}$ versus gate voltage and current plot of Fig. 19.24, we see that for 5 V V_{GS} and 1A I_D, we expect $R_{DS(ON)}$ to be about 0.5 ohm. But, suppose we wish to sink 3.0A of current representing, say a 12 volt supply and a 4 ohm load, well within the IRF510's ratings. As Fig. 19.24 shows, the minimum V_{GS} to obtain saturation at 3A exceeds

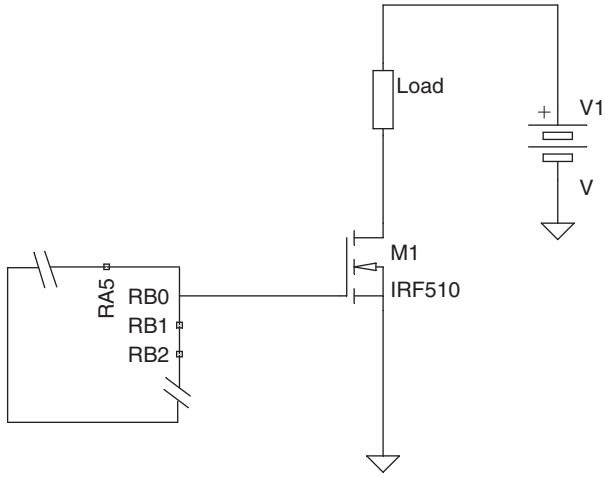


Figure 19.23: Low Side Switching with IRF510

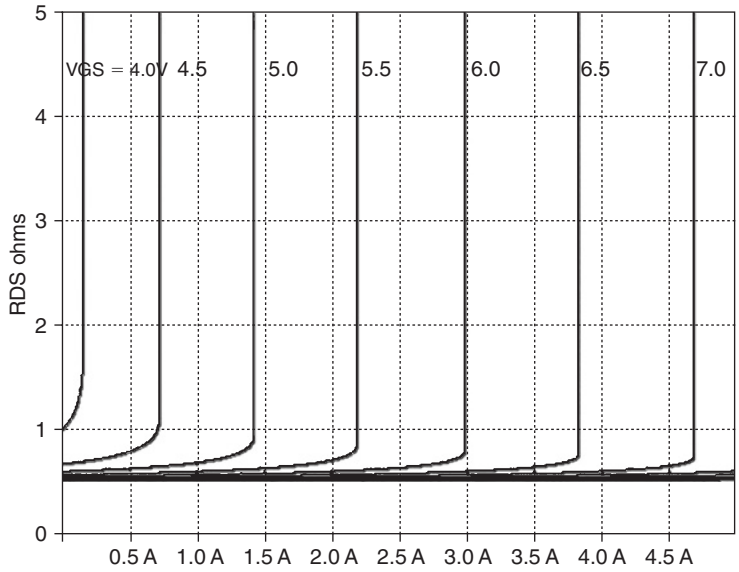


Figure 19.24: IRF510 Predicted On-resistance Variation with Gate Voltage and Drain Current

5 V. If all we have to drive the gate is the +5 V high from our PIC, $R_{DS(ON)}$ will be many ohms indicating the IRF510 is out of saturation and operating in the linear region. In fact, under these conditions with $V_{GS} = 5$ V we will find V_{DS} approximately 6.2 V and ID 1.4A. The IRF510 will dissipate 8.6 watts and its R_{DS} is 4.4 ohms. If we expected the IRF510 to act as a saturated switch, with an $R_{DS(ON)}$ of a few tenths of an ohm, we will be in for a surprise. And, if our design didn't use a heatsink because it wasn't necessary with a low $R_{DS(ON)}$, we'll have an even greater surprise as the IRF510 destroys itself from overheating.

Finally, rounding out the difficulties in driving an IRF510 directly from a PIC, we see the expected turn-on problem. Figure 19.25 shows nearly 750 μs turn-on delay.

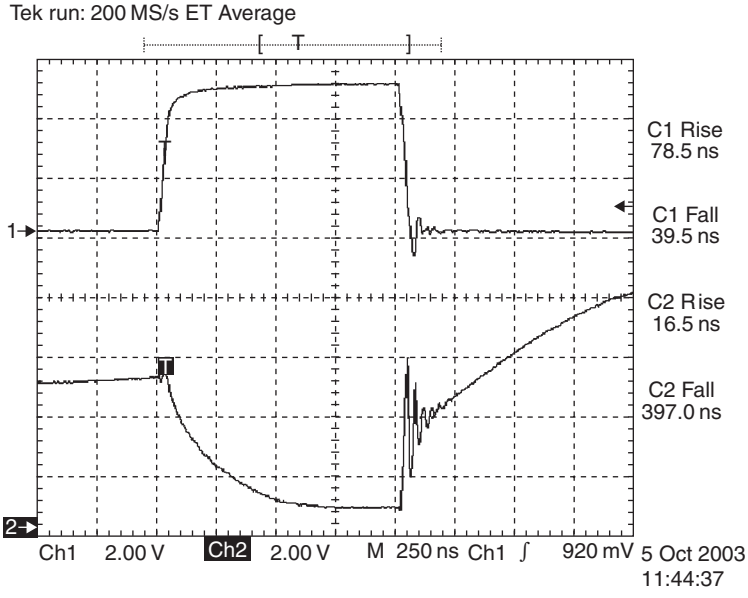


Figure 19.25: PIC Output; Ch2: IRF510 Drain

19.4.4.2 IPS021 Switch

Many suppliers have addressed the shortcomings seen when we examined the IRF510. We'll look at one particular device, International Rectifier's IPS021 intelligent power MOSFET switch which is rated at 50 V V_{DS} and has several interesting features:

- Logic level input with built-in voltage multiplier and level converter to ensure saturation of the switching MOSFET;
- $R_{DS(ON)}$ 0.125 ohms or less at room temperature and 5 V input;
- Over-temperature protected with automatic shutdown;
- Over-current protected, at 5.5 A nominal;
- Built-in snubbing protection for inductive loads.

The IPS021 is supplied in a TO-220 package and has the same pin connection as the IRF510. In almost all cases it can be directly substituted for an IRF510 with little or no change in circuit design or physical layout. As shown in Fig. 19.26, it is connected in the same fashion. International Rectifier recommends a series resistor of 500 ohms to 5 K ohms between the driving pin and the input pin of the IPS021, although it may not be necessary in all cases.

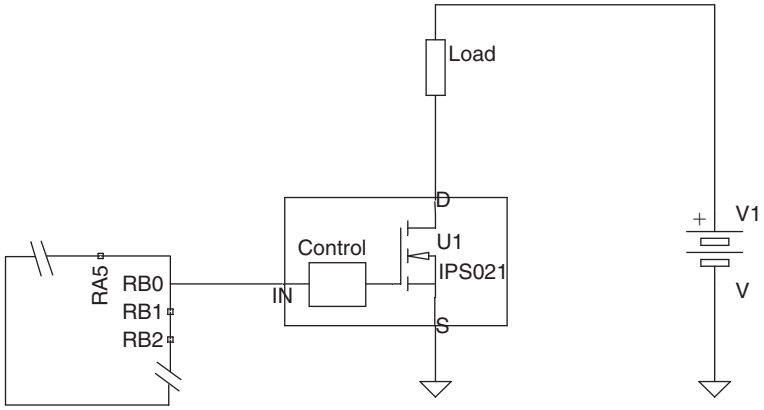


Figure 19.26: Low Side Switching with IPS021 Intelligent Power Switch

What's not to like about the IPS021? The main issue is switching speed, with turn-on and turn-off speeds in the several microsecond range. For many purposes, where a few microseconds of turn-on or turn-off time is immaterial, the IPS021 is a good choice. Figure 19.27 shows excellent performance in switching 1.5A in the circuit of Fig. 19.26. The series resistance of 240 milliohms computed from Fig. 19.27 includes wiring and plugboard components as well as the IPS021's $R_{DS(ON)}$.

19.4.5 High Side Switching

19.4.5.1 Small PNP Switch

Figure 19.28 depicts a simple high side switch. When RB0 is low, Q1 is forward biased into conduction and current flows through the load. Let's work through a few design concerns with this simple circuit. We will assume the load is a 47 ohm resistor and V is 2 volts. Hence, the current being switched is approximately 45 mA.

We calculate the base resistor and power dissipation exactly as for a low side switch, but we must be careful to select the correct voltage sources. The 2N4403 h_{FE} at -2 V V_{CE} and -150 mA I_C is specified at a minimum of 100. We will wish the bias current to be at least 450 μA. (In a PNP transistor, the collector is negative with respect to the emitter; hence the sign of the voltages and currents are reversed from the NPN case.) When conducting, the base bias source (RB0) is at 0 volts and the emitter is at *V* volts (2.0 V in our design example) positive. Hence in our example the voltage to drive the base current is 2.0 V. We will use the standard

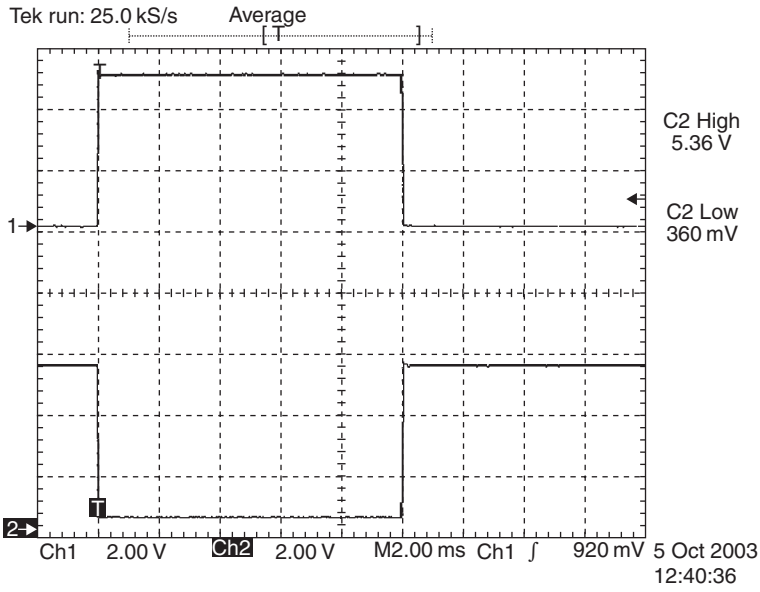


Figure 19.27: IPS021 Switching 1.5A Ch1: PIC Output; Ch2: IPS021 Drain

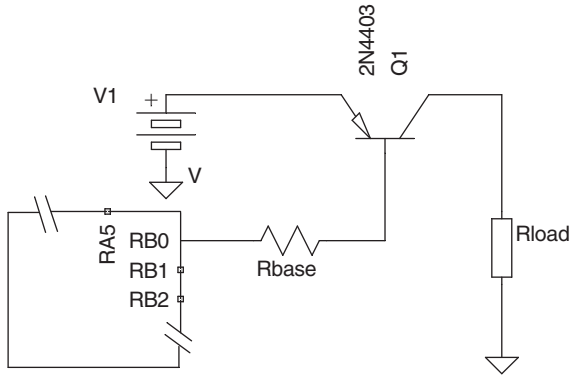


Figure 19.28: 2N4403PNP High Side Switch

700 mV assumption for Q1's base-emitter junction voltage drop and we'll ignore RB0's 26 ohm equivalent output resistance when low. Hence, the net voltage across Rbase is $-2.0\,V + 0.7\,V$ or $-1.3\,V$. To obtain 450 μA current flow, Rbase should be $1.3\,V/450 \times 10^{-6}$ or 2.9 Kohm. If we wish to ensure saturation under varying temperatures and component tolerances, we will increase the base current to, say 750 μA (Rbase calculated as 1.7 Kohm) and use the nearest standard 5% resistor value, 1.8 Kohm. Figure 19.29 shows our design in operation.

In low-side NPN transistor switching design, the voltage being switched was immaterial, as long as the transistor was within its ratings. When RB0 is at ground, the NPN transistor is reversed biased and hence cut off; both the emitter and collector are at the same potential, ground so there is no voltage difference between the base and emitter. Looking at the PNP high side circuit when

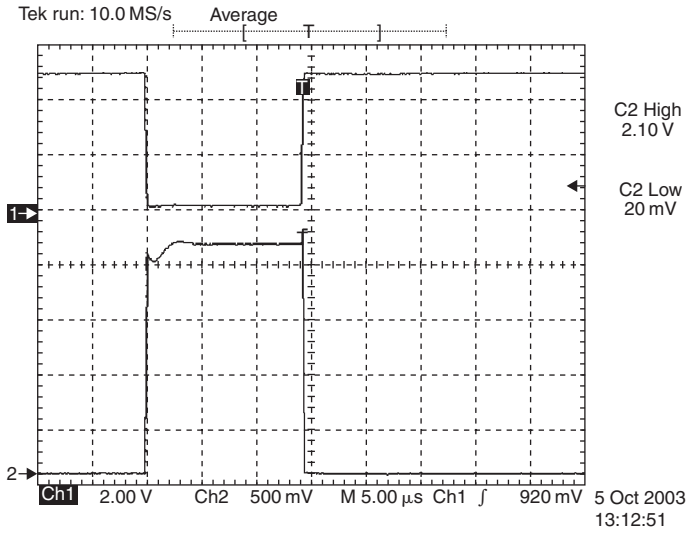


Figure 19.29: 2N4403 PNP High Side Switch Ch1: PIC Output; Ch2: 2N4403 Collector

RB0 is high and we wish Q1 to be cut off shows a different story, however. Suppose we wish to high side switch 12 V and RB0 is high, at 5 V. The end of Rbase connected to Q1's base is one base-emitter diode drop lower at +11.3 V while the end connected to RB0 is at +5 V. Hence current will flow out Q1's base-emitter junction; Q1 becomes forward biased and will conduct. This is the central problem in high side switching with a PNP transistor or a positive MOSFET (PMOS); we must have available control voltages at least equal to the voltage to be switched in order to turn off the device.

To control a high side switch exceeding the PIC's V_{DD}, we may modify the circuit of Fig. 19.28 by adding an 2N4401 NPN transistor to control the base of Q1, as shown in Fig. 19.30.

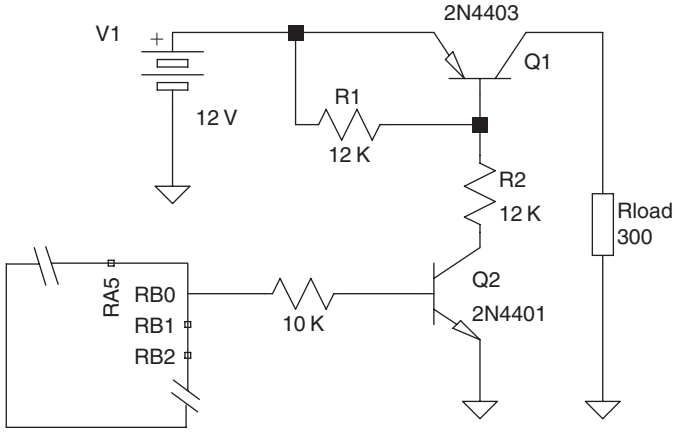


Figure 19.30: Use Both PNP and NPN Transistors to Implement High Side Switch

When RB0 is high, Q2 is forward biased and its collector is essentially at ground ($V_{CE(SAT)}$ is around 400 mV). Q1 is then biased into conduction by current flowing through R2 and Q2. When RB0 is low, Q2 is cut off and may be regarded as an open circuit. Hence, Q1's base is pulled to 12 V by R1, making V_{CE} zero, cutting Q1 off. The values given in Fig. 19.30 provide −0.9 mA base drive, more than adequate to saturate Q1 for the load current. Note that adding the 2N4401 driver inverts the control sense compared with the circuit of Fig. 19.28.

19.4.6 High Power High Side Switching

Rather than duplicate the low side switching discussion but with high power PNP transistors, or PMOS devices, we will immediately jump to an integrated high side MOSFET switch, International Rectifier's IPS511. The IPS511 is a member of IR's intelligent power switch family, but with some features not included in the IPS021 we earlier examined. (If you wish to experiment with a high side version of the earlier low side switching designs, a TIP125 is the PNP complement of the TIP120 Darlington and a TIP32 is the PNP complement of the TIP31. Finally, a BS250P is a good p-channel analog of the 2N7000 and an IRF9510 is the p-channel complement of the IRF510.)

If you construct high side switches with p-channel devices, you will soon discover they exhibit limited operating range when the gate voltage swing is limited to the 0…5 V available from a direct connection to a PIC's output pin. To reliably turn off an IRF9510 with a 5 V gate swing, for example, requires it to be switching less than approximately 8 volts, while to ensure the device is saturated requires it to be switching at least 4.5 volts. In the high side switching configuration, V_{GS} equals the voltage being switched minus either 0 V (PIC at low) or 5 V (PIC at high). Since V_{GS} must be at least −4.5 V to to bring $R_{DS(ON)}$ to reasonable levels (turn-on; PIC output is low so V_G is 0 V) the minimum voltage to be switched (V_S) must be 4.5 V. Conversely, to ensure turn off, V_{GS} must be at least −3 V when the PIC output is high. When high, V_G is +5 V, so V_S can't exceed +8 V. Hence, an IRF9510 high side switch with its gate driven directly by a PIC's output pin is of limited practical use. An auxiliary n-channel gate control device will be necessary in most cases, such as the 2N7000 shown in Fig. 19.31. To avoid exceeding the IRF9510's V_{GS} limit, the maximum voltage being switched must not exceed 20 V.

The IPS511 is a much cleaner alternative and offers many useful features, including:

- Logic level input with built-in voltage multiplier and level converter to ensure saturation of the switching MOSFET;
- $R_{DS(ON)}$ 0.135 ohms or less at room temperature and 5 V input;
- Over-temperature protected with automatic shutdown;
- Over-current protected, at 5.0A nominal;

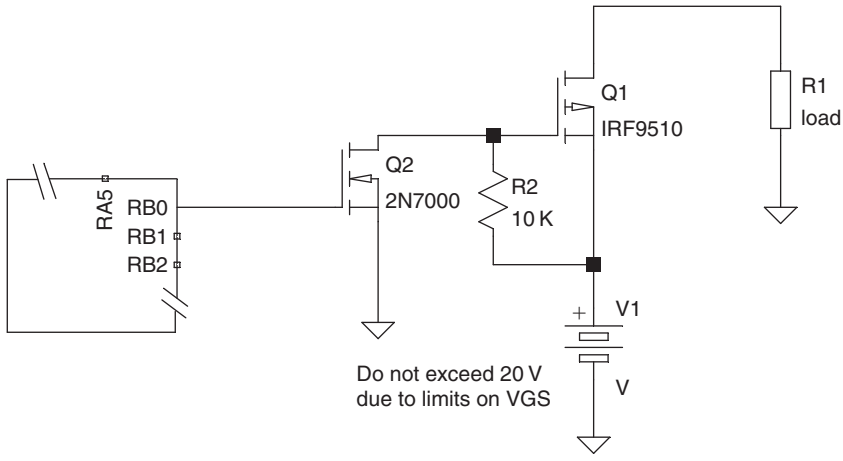


Figure 19.31: Suggested 2N7000/IRF9510 High Side Switch

- Built-in snubbing protection for inductive loads;
- Status feedback reporting of normal operation, open load, over current and over temperature.

The IPS511 is available in a 5-pin TO220 package and is rated at 50 V V_{DS}. Figure 19.32 shows a typical connection. As with IR's IPS021 low side intelligent switch, the IPS511 provides crisp, low drop switching as seen in Fig. 19.33. Rise and fall times for the IPS511 are well under 100 μs.

Unlike the IPS021, the IPS511 has a status pin which, when read in conjunction with the In and Out pins on the IPS511, provides useful information. In Fig. 19.32, the output voltage is sensed through a voltage divider consisting of R3 and R4, assuming, as is likely the case, the IPS511 is switching greater than 5 V. Applying Ohm's law and summing the voltage drops, if we wish the voltage at RB2 to be 5 V, we can state the relationship between R3, R4 and V_{OUT} as:

$$R_3 = \frac{V_{OUT} R_4}{5} - R_4$$

Since no significant current flows into a PIC input pin, we may select R4 as some convenient value, say 10 K. V_{OUT} will be only a few tenths of a volt different from V1, so we may use V1 instead of V_{OUT} in our calculation. Assume V1 is 12 V. We calculate R3 as:

$$R_3 = \frac{12 \times 10{,}000}{5} - 10{,}000 = 14{,}000$$

The nearest standard 5% value is 15 K.

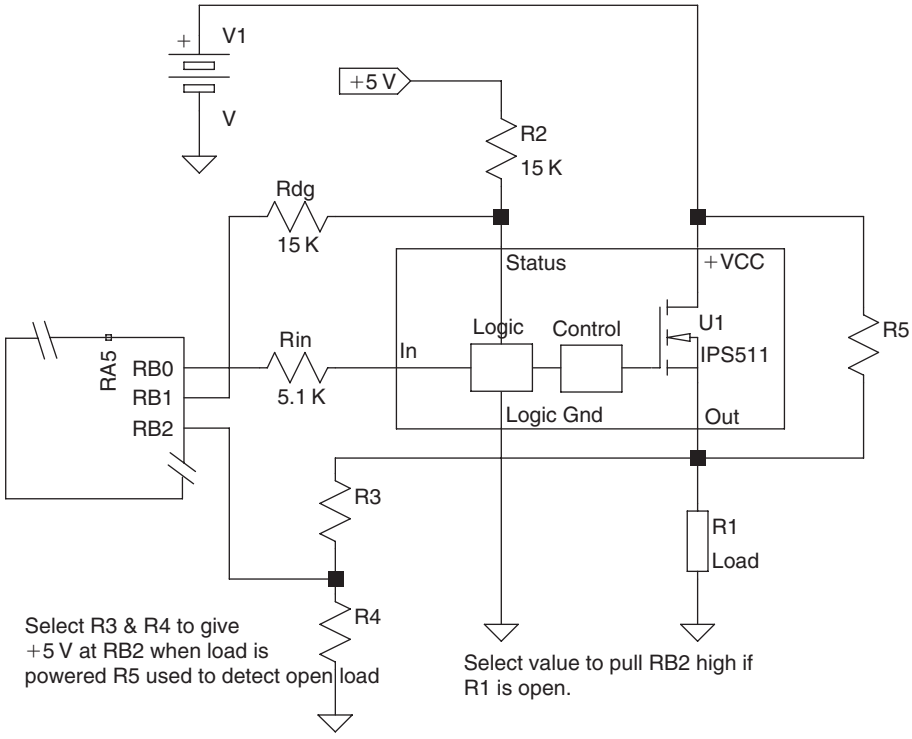


Figure 19.32: High Side Switching with IPS511 Intelligent Power Switch

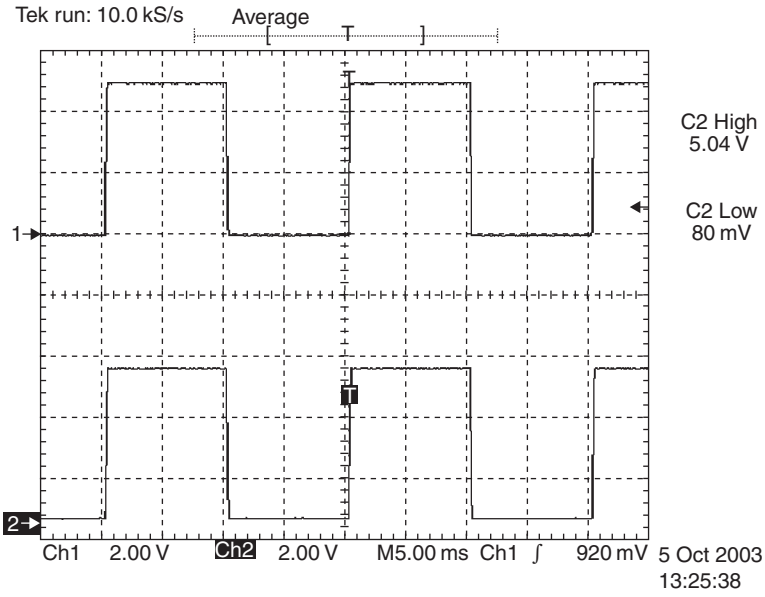


Figure 19.33: IPS511 Switching 50 mA Ch1: PIC Output; Ch2: IPS511 Drain

If we wish to detect an open load condition, we must also install the pull-up resistor R5. R5's value depends, in part, upon the load resistance and the acceptability of the resulting parasitic current through the load when the IPS511 is off as well as the values of R3 and R4. Let's assume Rload is 12 ohms. If we set R5 to 1200 ohms, 100 times Rload, the parasitic current will be not more than 1% of the normal on current and the voltage across the load will be less than 1% of V1. For many purposes, this will be more than acceptable. If the load should become an open circuit, then when the IPS511 is off the voltage at RB2 will then be derived through a voltage divider where the top resistor consists of R3 in series with R5. If we can keep R5 to approximately 10% of R3, we will still maintain a good high condition on RB2. R3 is 15 K, so as long as R5 is at least 1500 ohms, we have met this objective. Hence, we may pick R5 as 1300 ohms, a standard 5% value between our minimum desired value of 1200 ohms and our upper objective of 1500 ohms. We can quickly check the voltage at RB2 under the four possible conditions:

Voltage at RB2 Under Conditions of Load Status			
In State	**Load**	**RB2 Voltage**	**RB2 Status**
H	Normal	4.8	H
H	Open	4.8	H
L	Normal	0.1	L
L	Open	4.6	H

As we will learn later, when an input, the maximum voltage level a PIC is guaranteed to be read as a low is 0.8 V, while the minimum voltage that will be read as a high is 2.0 V on a PortB pin, assuming V_{DD} is in the range 4.5 V…V_{DD}…5.5 V. Hence, RB2 will be read as low only under the condition at the third line in the table.

To determine the status of the IPS511 we would implement the following pseudo-code:

```
;Possible routine to read status of IPS511
;Assumes connection to PIC as at Fig 19.32
;
Status      var   PortB.Bit1
Load        var   PortB.Bit2

Input B1
Input B2
;
Main
      High RB0    ;turn on the IPS511 and apply power to load
      Pause 2     ;Delay to ensure IPS511 is fully on

      ;now check the status when the IPS511 should be on
      If (Status=1) AND (Load=1) Then GoSub NormalOn
      If (Status=0) AND (Load=0)Then GoSub Overload
```

```
      Low RB0     ;turn the IPS511 off
      Pause 2     ;now check the status when off
      If (Status=0) AND (Load=0) Then GoSub NormalOff
      If (Status=1) AND (Load=1) Then GoSub OpenLoad

NormalOn ;Subroutine if all is OK at the load when on
;------
      Code to be executed if result is OK
Return

NormalOff ;Subroutine if all is OK at load when off
;-------
      Code to be executed if result is OK
Return

OpenLoad ;Subroutine to be executed if load is open
;------
      Code to be executed if the load is open
Return

OverLoad ;Subroutine to be executed if excessive current
;----------   or over temperature
      Code to be executed for over current condition
      Or over temperature condition. Determine the difference
      Between the two by checking for cycling or steady state
      Low on Load
      If Load is cycling-problem is over temperature
Return
```

Separating certain faults, such as over temperature and over current may require repeated polling of the status and output pins to determine whether the fault periodically clears itself (as the device cools down and the thermal trip resets) or remains static. Additionally, if the onset of current limiting must be detected it may be necessary to alter R3, R4 and R5 to cause abnormally low, but not zero, voltage across the load to read as a low on RB2.

19.5 Isolated Switching

Although we will discuss a variety of devices under the isolated switching category, of course these may also be used for low side or high side switching.

19.5.1 Relay Switching

Relays predate electronics, as they were developed to extend the range of manual Morse telegraph systems in the mid-1800s. Nonetheless, we should not quickly discard the relay solution to switching. Relays are available in a rich variety of contact configurations, contact material, power rating, voltage rating and coil rating.

Good things about relays

- Resistant to damage from overloads and polarity reversal
- Excellent isolation between switched load and controlling circuitry
- Wide range of ratings
- Can switch AC, DC, video, RF, low level audio, etc. by proper selection of device.

Not so good things about relays

- Limited life, although many relays are rated for tens of millions of operations
- Noise
- Speed of operation and release, usually in the millisecond range.
- Size may be an issue
- Contact bounce
- Requires power to hold relay operated, unless it is a latching relay.

We'll look at three relays:

Make/Model	Coil Rating	Contact Configuration	Contact Rating	Comments
Standex JG102-12-1	12 V/24 mA	SPST (1A)	48 V/1A	Very high speed reed relay
Omron G5V-2-H1	12 V/12.5 mA	DPDT (2C)	125 VAC/0.5A 30 VDC/2A	Low signal relay with bifurcated contacts; ultra sensitive
Omron G2RL-24	12 V/33.3 mA	DPDT (2C)	250 VAC/8A 30 VDC/8A	General-purpose power relay

To test the contact closure and operate/release time, we'll use the circuit of Fig. 19.34. Figure 19.34 should be familiar; a 2N7000 low side switch drives the relay coil. As we earlier determined, the current and voltage required to operate the relays under test is well within the limits for a 2N7000. To sense contact closure and bounce, we use a 5 V source in series with a 56 ohm resistor to pass approximately 90 mA through the contacts. In order to make measurements, V2's negative terminal is connected to ground; but in order to emphasize the ability of relays to switch isolated circuits, Fig. 19.34 shows the load in its most general "floating" form.

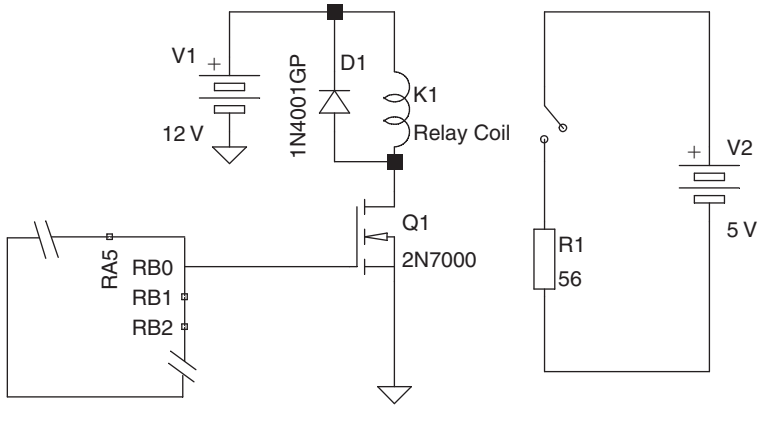


Figure 19.34: Relay Test Circuit

Since a relay coil is an inductive load, we must use a snubbing diode to avoid damaging the 2N7000. To illustrate the voltage spike even a small inductive load generates, compare Figs 19.35 and 19.36. Without a snubbing diode, the inductive spike exceeds 76 V, at which point the 2N7000 breaks down. Adding a 1N4001 snubbing diode reduces the spike to 12.7 V—that is, 0.7 V above the 12 V relay supply voltage.

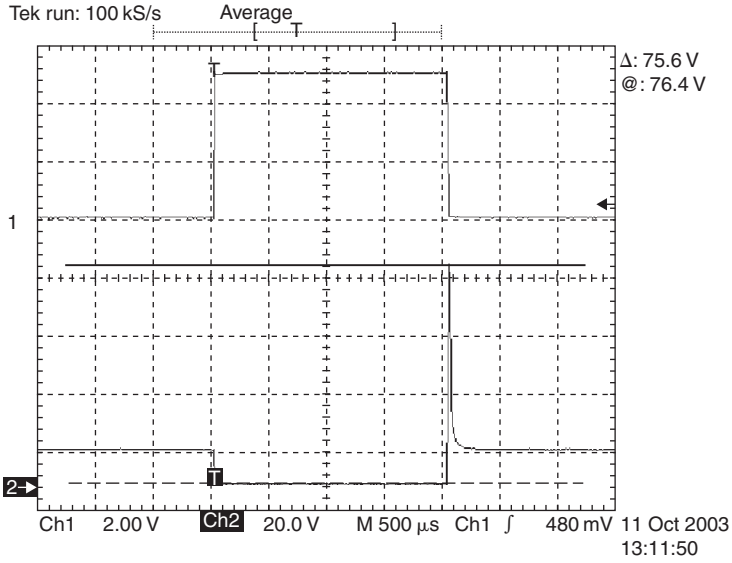


Figure 19.35: JG102-12-1 Relay Without Snubber Ch1: PIC Output; Ch2: 2N7000 Drain

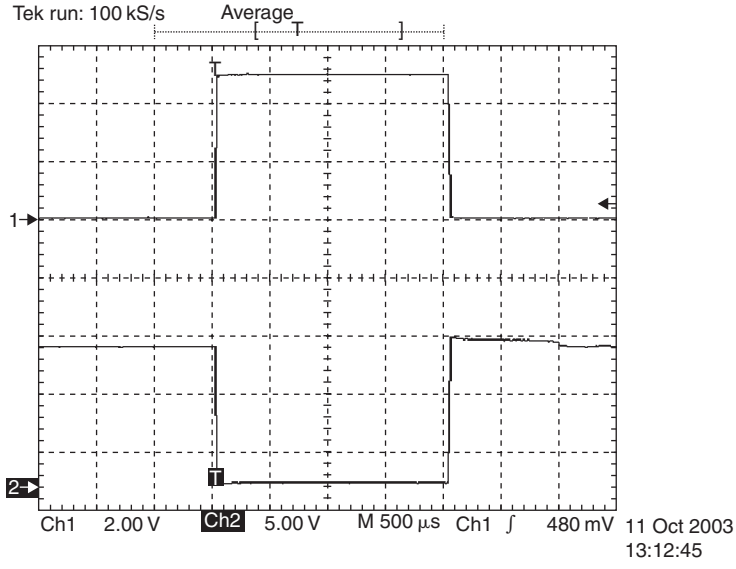


Figure 19.36: JG102-12-1 Relay with Snubber Ch1: PIC Output; Ch2: 2N7000 Drain

Finally, although the operating and release times for these relays is long compared with pure electronic switches, for many applications a few milliseconds delay between PIC output and relay pull-in is of little consequence.

19.5.1.1 Standex JG102-12-1

Reed relays are well known for high-speed operation, and the JG102-12-1 (since replaced by model JG10012-1) meets our expectations. Figures 19.37–19.39 show operating times for this device. The JG10212-1 relay turns on in 250 μs, but an additional 150 μs is necessary for contact bounce to cease. Turn-off time is also approximately 250 μs. Since the turn-on and turn-off times are approximately equal, the overall relay high time is very close to the PIC control pin high time, although delayed by approximately 400 μs. (The terms "operate time" or "pull-in time" and "release time" are usually used when discussing relay speeds, instead of turn-on and turn-off. However, to illustrate the commonality with transistor switching, we'll use the turn-on and turn-off terminology as well.)

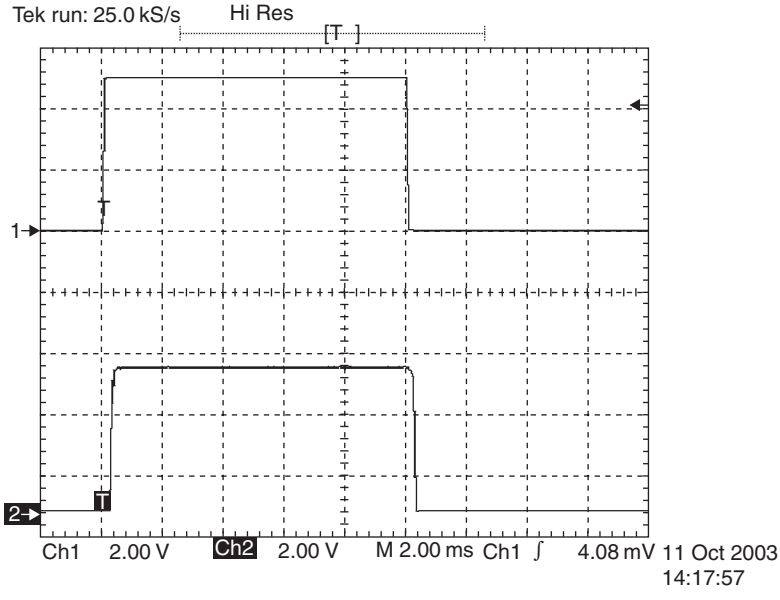


Figure 19.37: JG102-12-1 Reed Relay Ch1: PIC Output; Ch2: Relay Contact

Reed relays are commonly used in telecommunications equipment to switch voice, data and high frequency signals and are not often employed for switching power circuits.

19.5.1.2 Omron G5V-2-H1

The G5V-2-H1 is a member of Omron's telecommunications family, optimized for low level signals and is used for purposes similar to those of reed relays as well as for low level power

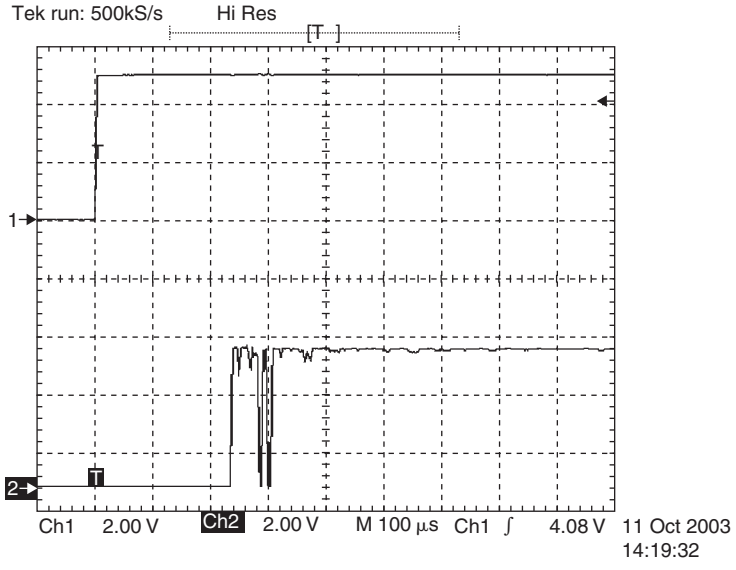


Figure 19.38: JG102-12-1 Reed Relay Turn-on Delay and Contact Bounce Ch1: PIC Output; Ch2: Relay Contact

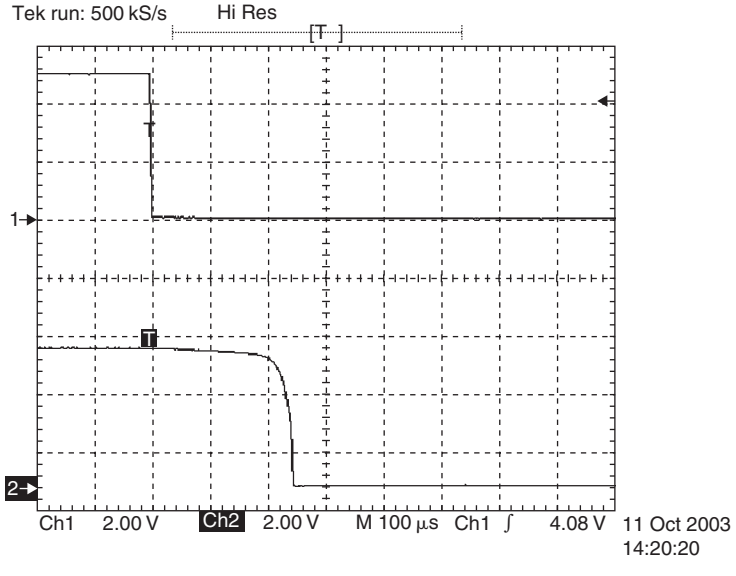


Figure 19.39: JG102-12-1 Reed Relay Turn-off Delay Ch1: PIC Output; Ch2: Relay Contact

switching. The G2V is of conventional relay construction, but with bifurcated cross-point gold plated silver contacts. Even though the contacts are of precious metal, Omron quotes a minimum contact load of 10 µA and 10 mV DC. At lower levels, oxides and contaminants may prevent reliable operation.

Although constructed with bifurcated cross-point contacts, contact bounce is still apparent with the G5V-2-H1, as reflected in Fig. 19.40. Operate and release times are approximately 4 ms.

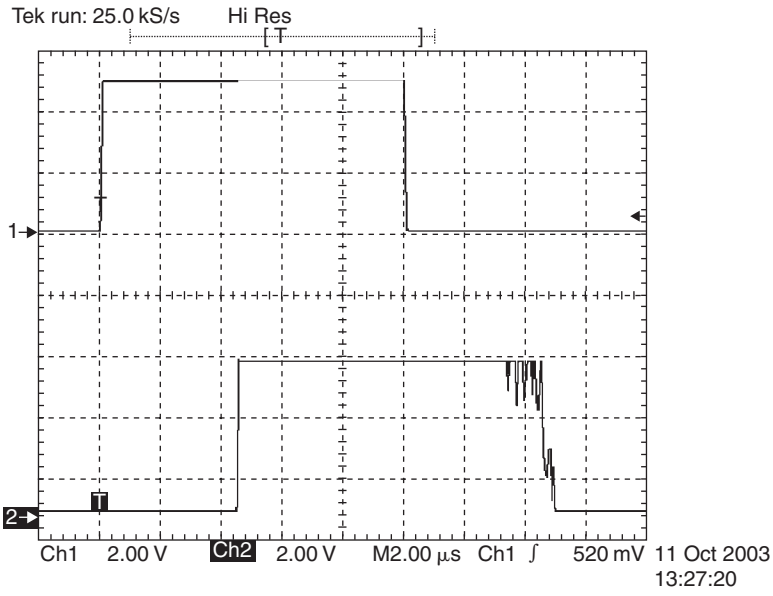


Figure 19.40: G5V-2-H1 Relay Ch1: PIC Output; Ch2: Relay Contact

19.5.1.3 Omron G2RL-24

The G2RL-24 is a power relay, suitable for switching AC and DC up to 8 A. As might be expected, relays designed to switch higher currents are more substantially constructed and hence take more time to operate and release. Figure 19.41 shows the G2RL-24 requires nearly 8 ms to operate and dampen contact bounce. Release time is shorter, only 4 ms. Since release is shorter than operate, the original 10 ms PIC output only results in 6 ms of useful relay closure.

19.5.2 4N25 Optical Isolated NPN Switch

Optical couplers or "optoisolators" consist of an LED packaged with a photodiode or a phototransistor. When illuminated, light from the LED saturates the receptor and it conducts. There is no electrical connection between the LED input and photo device output, so the two circuit halves are independent and may be at a potential difference of hundreds or even thousands of volts.

Optoisolator are available in a wide range of operating speeds and configurations. We'll first look at a low power optoisolator, the venerable 4N25 device, followed by a modern high power optically coupled MOSFET, the PS710A-1A-1.

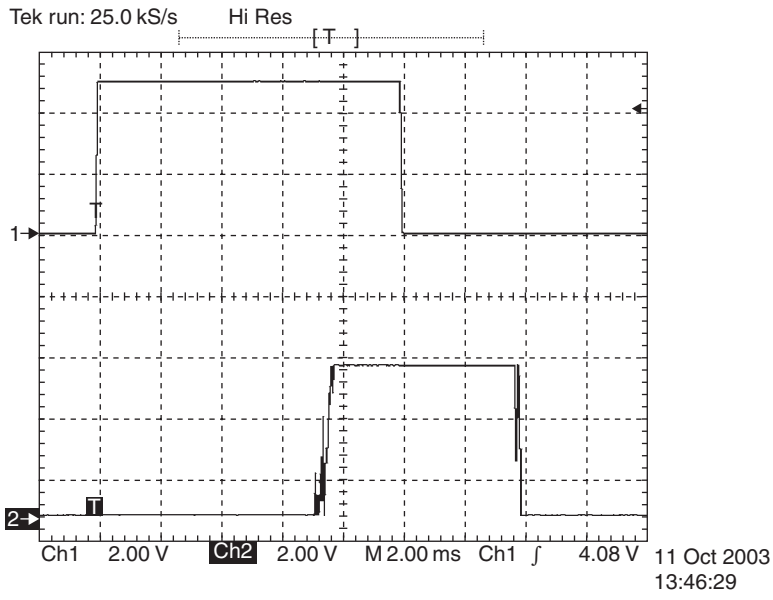


Figure 19.41: G2RL-24 Relay Ch1: PIC Output; Ch2: Relay Contact

Figure 19.42 shows how simple it is to connect an optoisolator to a PIC. R1 is selected to achieve the desired LED on-current using the methodology developed earlier in this chapter. The 220 ohm resistor is intended to provide approximately 10 mA LED on-current. The 4N25's output transistor is configured as a low side switch in the example.

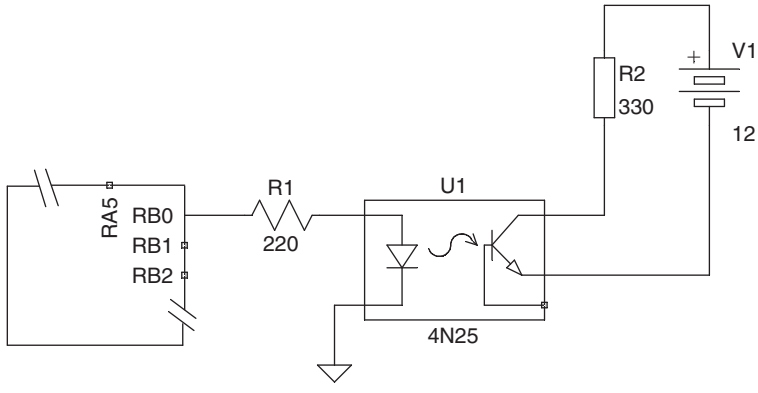


Figure 19.42: 4N25 Optoisolator Connection

With a maximum current rating of only 150 mA, the 4N25 is not intended to switch large currents so it will often be used as the first stage in a multistage switching arrangement, such as that shown in Fig. 19.43. In this design, the 4N25 operates as an emitter follower. When the LED is illuminated via RB0 going high, current flows through the 4N25's output transistor and R3,

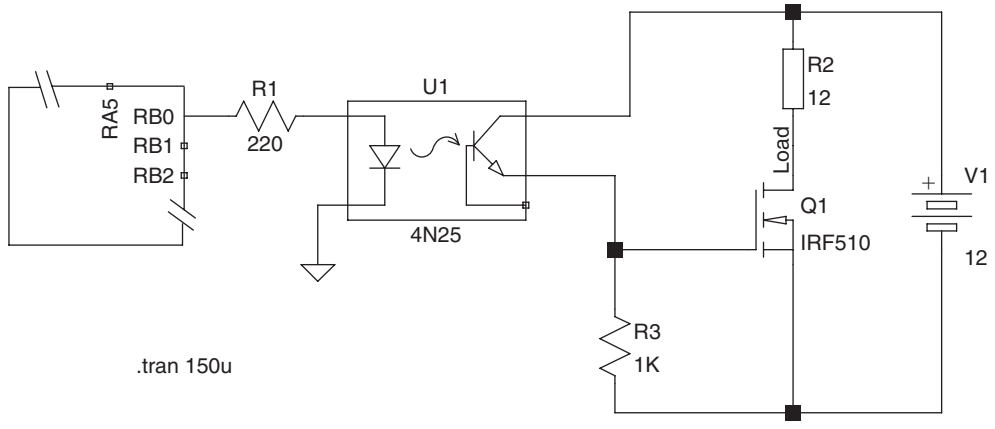


Figure 19.43: 4N25 with IRF510

taking the gate of Q1 into conduction. Current then flows through the load and Q1. When the LED is dark, the 4N25's output transistor is cut off and Q1's V_{GS} is close to zero and Q1 is cut off. Of course, any of the more modern IPS products may be substituted for the IRF510.

Low-power optoisolators are also often used to isolate data or signal circuits from the PIC, and an optical coupled RS-232 circuit permits isolating a PIC from the associated computer or controlled devices.

19.5.3 PS710A-1A AC/DC Optically Isolated MOSFET

NEC's PS710A-1A is a high power MOSFET optoisolator. Unlike the 4N25, the PS710A is a power device, capable of switching loads up to 1.8 A at 60 V, and its series MOSFET design will switch both AC and DC. Each MOSFET has an $R_{DS(ON)}$ of 0.1 ohm and, for DC switching, may be paralleled handle 3.6A. The PS701A-1A is connected as illustrated in Fig. 19.44 for AC or DC switching. Other configurations are possible for DC switching and you should consult the data sheet for additional information.

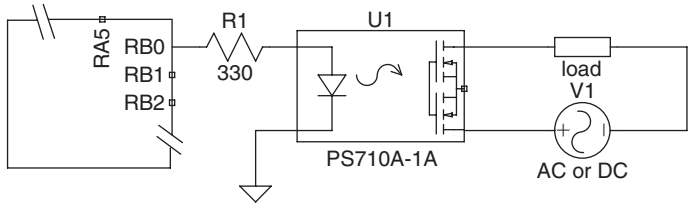


Figure 19.44: AC/DC Isolated Switching with NEC's PS710A-1A

Figure 19.45 shows the results for the circuit of Fig. 19.44, switching 1 A at 5 V. Figure 19.45 confirms the data sheet's 1 ms typical turn-on time and 50 μs typical turn-off time.

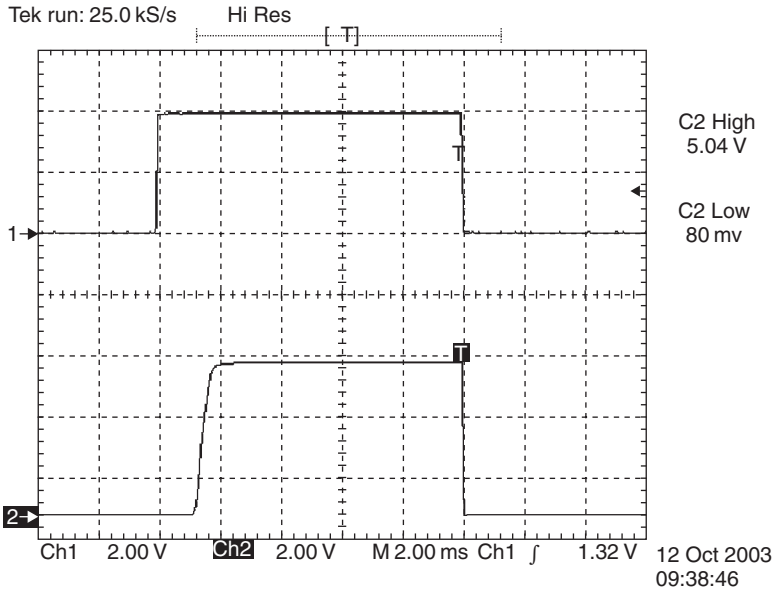


Figure 19.45: Switching with PS710A-1A Ch1: PIC Output; Ch2: Load

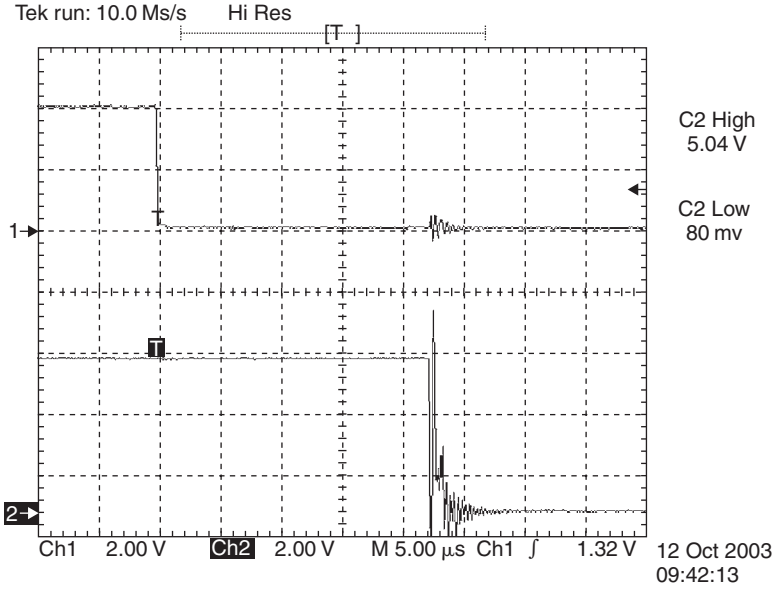


Figure 19.46: Undesired Oscillation in Plug Board Layout of PS710A-1A Ch1: PIC Output; Ch2: Load

In switching high currents in microsecond times, undesired transients and oscillations are often seen, particularly when using a plug-in board and lab power supplies with long leads. When the design is transferred to a printed circuit board with wide power traces and integrated power

distribution filtering the problem is often solved. In some cases additional filtering and bypassing will be necessary. Figure 19.46 shows an expanded view of the PS710A-1A's turn-off interval illustrates the type of extraneous transients sometimes seen in a plug board layout. Don't be surprised to see similar unwanted oscillations in some of your layouts. Even in a breadboard layout, you can usually stop or at least reduce these extraneous signals by better attention to lead dress and grounding jumper locations, combined with additional power lead bypass capacitors.

19.6 Fast Switching—Sound from a PIC

So far in this chapter, our emphasis has been on relatively slow switching. But, if we switch a loudspeaker off and on at an audio rate, we can produce sound, perhaps to be used as an alert tone, or a beep to confirm an action or status. (We will need fast switching to control a DC motor's speed through pulse width modulation, and to control stepper motors, both topics dealt with in later chapters.)

We can generate a sound either through a self-contained sounder, such as the Sonalert® products introduced by Mallory, or through the PIC producing the audio signal itself. A Sonalert may be driven by a PIC using any of the techniques you learned earlier in this chapter. Later chapters explore in some detail the advantages and disadvantages of various ways to generate audio signals using MBasic. Here, however, we will just look at two simple interfaces and one of the many audio output procedures available in MBasic.

We'll assume you don't intend to produce ear splitting, high fidelity output from a PIC. Rather, you are interested in beeps and other alerting tones. In some cases, it may be possible to obtain adequate volume levels by driving the speaker directly from a PIC, as illustrated in Fig. 19.47. When thinking of a speaker, low impedance designs most often come to mind, with 3.2, 4 and 8 ohm devices being common. I've generally been disappointed with the volume levels when a low impedance speaker is directly connected to a PIC. Indeed, a series resistor, R1 in Fig. 19.47, of 50 ohms or so is necessary to produce useful sound output.

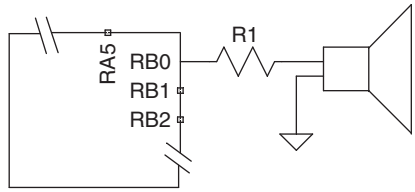


Figure 19.47: Driving a Speaker Directly from a PIC

Remember, however, that a high output pin may be thought of as a 5 V source in series with approximately 85 ohms. Working into, for example, a 3.2 ohm speaker with a 50 ohm series resistor, approximately 99.8% of the theoretical maximum output power of the pin will be lost

and not transferred to the speaker. This still may produce an acceptable volume level. If the speaker has higher impedance, say at least 32 ohms, a much greater proportion of the available power will produce useful sound. If necessary, a simple series resistor, shown as R1 in Fig. 19.47, can serve as a volume control.

If you've examined a loudspeaker you know the typical construction consists of a paper cone that moves in or out in response to current through the voice coil. Our simple connection of Fig. 19.47 moves the cone only in one direction, either in or out, depending on which speaker connection you ground and which you connect to the PIC's pin. The unidirectional motion throws away one half the potential sound level. Depending on your desired sound level and speaker rating, this may or may not be important. Figure 19.48 shows we are able to develop a peak current of 27 mA through a 3.2 ohm speaker. This particular speaker yielded a weak sound with 27 mA current. We may calculate the power delivered to the speaker by recalling that the RMS power a square wave is equal to the one-half the peak power. (The RMS of the on period is equal to the peak; but since half the cycle is off, the RMS reduces by one half.) Hence, the RMS power delivered to the speaker is approximately 1.2 mW. (This is based upon the speaker's nominal 3.2 ohm impedance. Measurements of the particular speaker I tested showed its true impedance at 1000 Hz is 3.09 ohms, representing 2.95 ohms resistance in series with 149 μH inductance.)

Let's look at a higher power driver for a low impedance speaker. Since we are not overly concerned with the sound quality—the PIC sound procedure we use outputs a square wave,

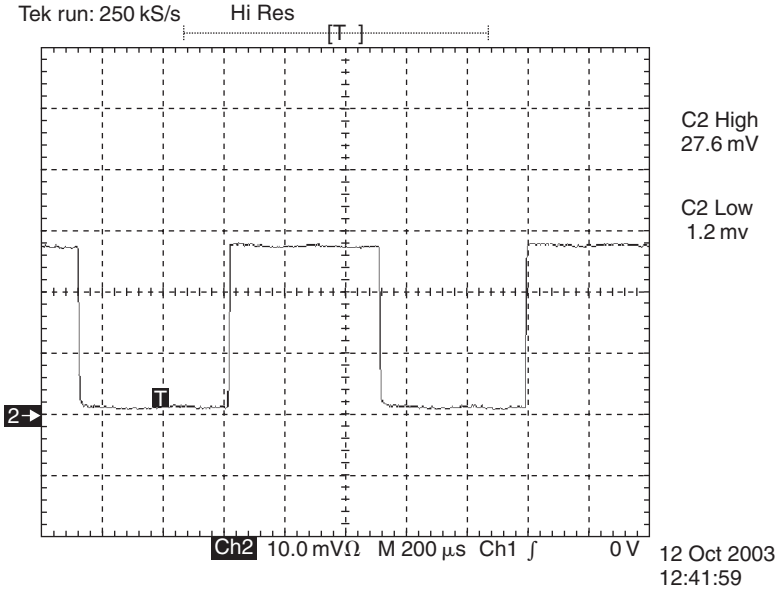


Figure 19.48: Direct Drive of 3.2 ohm Speaker with PIC and 56 ohm Series Resistor Ch2: Speaker Current (mA)

after all—we will use a 2N4401 emitter follower to drive the 3.2 ohm speaker, using the circuit shown in Fig. 19.49. And, to permit the speaker's voice coil to have both in and out excursions, we use C1 to block the DC component.

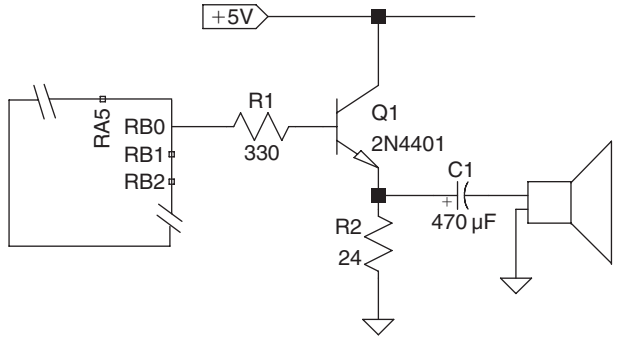


Figure 19.49: 2N4401 Emitter Follower Speaker Drive

Figure 19.50 shows the resulting current thorough the speaker. The RMS power delivered to the speaker is now approximately 45 mW, yielding nearly 16 dB more sound output, a very noticeable improvement over the direct drive connection.

Program 19.3 uses MBasic's `sound` procedure to output a 1000 Hz square wave for 1,000 milliseconds on RB0. The tone output is repeated endlessly through the `GoTo Main` loop.

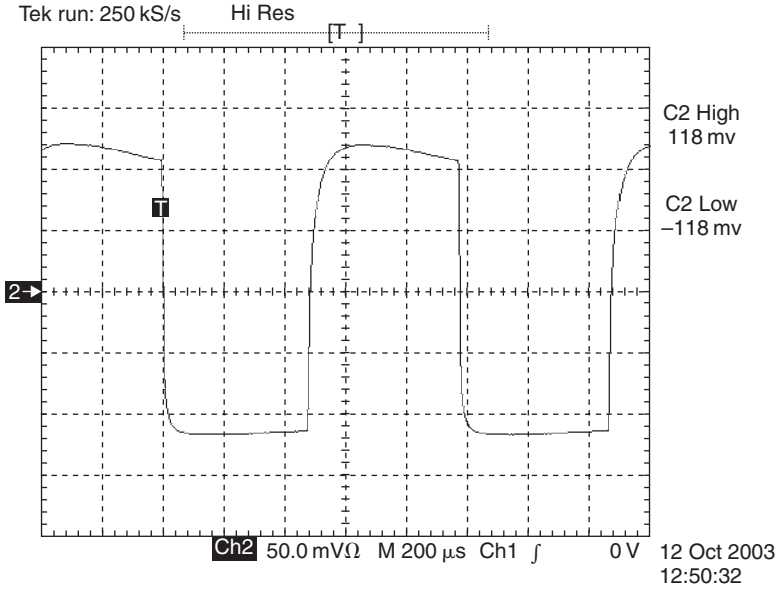


Figure 19.50: 2N4401 Follower Drive of 3.2 Ohm Speaker with PIC and 56 Ohm Series Resistor Ch2: Speaker Current (mA)

```
;Program 19.03
Main
   ;burst of 1000 Hz for 1 second
        w/ endless loop
   Sound B0,[1000000\1000]
GoTo Main

End
```

Program 19.3

References

[19.1] Horowitz, Paul and Hill, Winfield, *The Art of Electronics, 2nd. Ed.*, (1989). If you have only one book on electronics in your library, this should be it. A long-awaited 3rd edition is rumored to be in the works, but that shouldn't discourage you from purchasing the 2nd edition.

[19.2] American Radio Relay League, *ARRL Handbook for Radio Communications* 2003 ed., American Radio Relay League (2003). Although aimed at radio amateurs, the ARRL Handbook provides good entry-level coverage of basic analog and digital electronics, test equipment and construction practices. The ARRL updates its handbook every year, so purchase the most recent version available.

[19.3] Ludeman, Robert R., *Introduction to Electronic Devices and Circuits*, Saunders College Publishing (1990). Written as an introductory text for community college electronic technician students, it's a good summary of basic solid-state electronics without requiring advanced mathematics.

[19.4] Linear Technology Corp. makes available a free Microsoft Windows-based SPICE simulator and schematic capture software, "LTSpice/SwitcherCAD III." Although aimed as a design tool supporting LTC's products, the software is not limited to LTC devices. It may be downloaded at http://www.linear-tech. com/software/. In addition, add-on device libraries and explanatory material for LTSpice are available in the associated Yahoo user group http://groups.yahoo.com/group/LTspice/ in the files folder. The schematics and simulations in this book use LTSpice.

[19.5] Barkhordarian, Vrej, *Power MOSFET Basics*, International Rectifier Corp. Technical Note (undated).

[19.6] International Rectifier Corp., *The Do's and Don'ts of Using MOS-Gated Transistors*, AN-936 (v. Int). (Undated).

[19.7] International Rectifier Corp., *Current Ratings of Power Semiconductors*, AN-949 (v. Int), (Undated).

[19.8] International Rectifier Corp., *Selecting and Designing in The Right Schottky*, AN-968, (Undated).

[19.9] Omron Electronics, Inc., *Relay User's Guide* (1990). Available for free download at Omron's reference center http://oeiwcsnts1.omron.com/pdfcatal.nsf. From this page, select Relays. From the relays page select Manual.

Data sheets for the devices used in this chapter are available for downloading at the following URL addresses:
2N4401: http://www.fairchildsemi.com/ds/2N/2N4401.pdf
2N4403: http://www.fairchildsemi.com/ds/2N/2N4403.pdf
PS710A-1A: http://www.csd-nec.com/opto/english/pdf/PN10268EJ01V0DS.pdf
2N7000: http://www.fairchildsemi.com/ds/2N/2N7000.pdf
4N25: http://www.fairchildsemi.com/ds/4N/4N25.pdf
TIP31: http://www.fairchildsemi.com/ds/TI/TIP31.pdf
TIP120: http://www.fairchildsemi.com/ds/TI/TIP120.pdf
MV5491A Dual LED: http://www.fairchildsemi.com/ds/MV/MV5094A.pdf
IRF510: Go to International Rectifier's home page http://www.irf.com/ and enter IRF510 in the search box.
IRF9510: Go to International Rectifier's home page http://www.irf.com/ and enter IRF9510 in the search box.
IPS021: Go to International Rectifier's home page http://www.irf.com/ and enter IPS021 in the search box.
IPS511: Go to International Rectifier's home page http://www.irf.com/ and enter IPS511 in the search box.
G5V Relay: Go to Omron's home page for US relay products http://oeiweb.omron.com/ and enter G5V-2-H1 in the search box.
G2RL-24 Relay: Go to Omron's home page for US relay products http://oeiweb.omron.com/ and enter G2RL-24 in the search box.
Standex JG100 Relay: http://www.standexelectronics.com/serjg.htm

This page intentionally left blank

CHAPTER 20

The Basics—Digital Input

After Chapter 19's examination of the output mode, we'll now turn to PIC pins used as digital input devices.

Digital signal levels are either a logical low (0) or a logical high (1)—what could be simpler? As with the rest of this book, we assume V_{DD} is 5 volts and V_{SS} is 0 volts. In PIC logic, a 0 volt input corresponds to a logical low. Likewise, a 5V input is a logical high. But, suppose our input is 1.7V. Is it a logical low or is it a logical high? Does the answer to this question depend on our choice of an input pin? And, does it depend on the voltage at the input pin earlier in time? We'll find out in this chapter.

20.1 Introduction

First, let's define a few terms, as illustrated in Fig. 20.1:

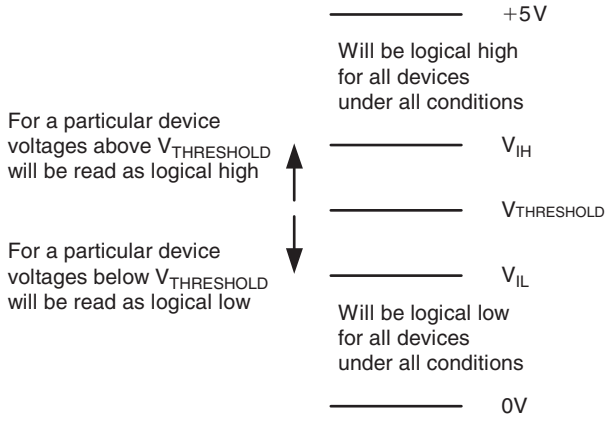


Figure 20.1: Input Level Relationships

V_{IL}—The maximum voltage on an input pin that will be read as a logical low.
V_{IH}—The minimum voltage on an input pin that will be read as a logical high.
Undefined region—The undefined region is the voltage level between V_{IL} and V_{IH}. Input voltages in the undefined region may be read as a low or as a high, as the PIC's input

circuitry may produce one result or the other. (Obviously the voltage will be read as either a low or a high; it's just that we have no assurance which one it will be.)

Threshold voltage—The input voltage that separates a low from a high; the threshold voltage, V_T, minus a small increment is read as a low while the threshold voltage plus a small increment is read as a high. The threshold voltage differs from logic family to logic family and somewhat between different chips of the same type. The differences between V_T and V_{IL} and V_{IH} are design margins accounting for device-to-device process tolerances, temperature effects and the like. In an ideal logic device, V_{IL} equals V_{IH} and there is no undefined region.

Microchip has chosen to build PICs with varying input designs and associated varying V_{IL} and V_{IH} values. We will not consider a few special purpose pins, such as those associated with the oscillator, in this chapter. Even so, the 16F87x series PICs have three input pin variations:

TTL level—Of the many logic families introduced in the early days of digital integrated circuits, transistor-transistor logic (TTL) was by far the most successful with TTL and its descendants still used today. Port A and Port B input pins mimic TTL logic input levels, except for RA4, which has a Schmitt trigger input. (Of course, TTL style Schmitt trigger inputs exist; but since they are not found in the PICs we consider, we won't further consider them.)

Schmitt trigger inputs—Almost all other input pins are of Schmitt trigger design. A Schmitt trigger has different transition voltages, depending on whether the input signal is changing from high to low or low to high, as illustrated in Fig. 20.2.

Special Schmitt trigger inputs—Pins RC3 and RC4 are software selectable Schmitt trigger or SMBus configuration. (SMBus is a protocol developed by Intel for data exchange between

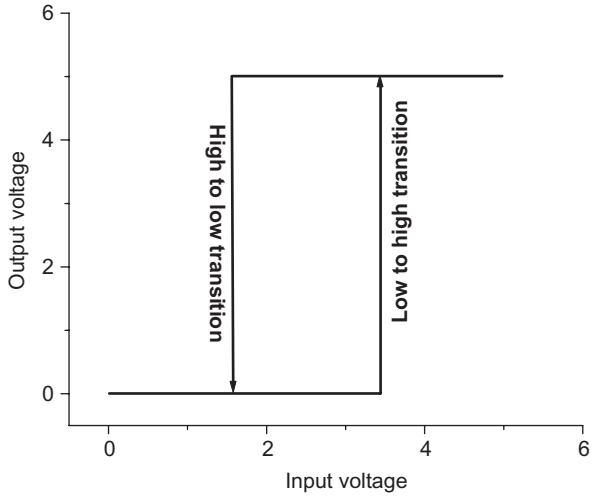


Figure 20.2: Schmitt Trigger Input

integrated circuits. We will not further discuss SMBus communications in this book.) When RC3 and RC4 are used as normal, general purpose, input pins, their parameters differ slightly from those of the other Schmitt trigger input pins.

Input Logic Level Specifications: 16F87x with $4.5\,V < V_{DD} < 5.5\,V$		
Input Type	V_{IL} (max)	V_{IH} (min)
TTL	0.8 V	2.0 V
Schmitt	0.2 V_{DD}	0.8 V_{DD}
RC3/RC4 Schmitt	0.3 V_{DD}	0.7 V_{DD}

Fig. 20.3 illustrates the differences between the three input types. To see how these input designs differ in practice, we can apply a time-varying 0…5 V signal to an input pin and plot the input voltage against the output value using an oscilloscope. If we have a stand-alone logic gate, this is a straightforward exercise. Fig. 20.4, for example, shows the input versus output result for a SN7400 TTL quad NAND gate. (To obtain a noninverting output, the test configuration places two NAND gates in series, and to yield the full 5 V logic high, a 2.2 Kohm pull-up resistor was added to the output gate.) We see a clear, narrow transition, with $V_{TRANSITION}$ around 1.6 V, fully consistent with Microchip's V_{IL} and V_{IH} parameters for TTL mimicking input pins.

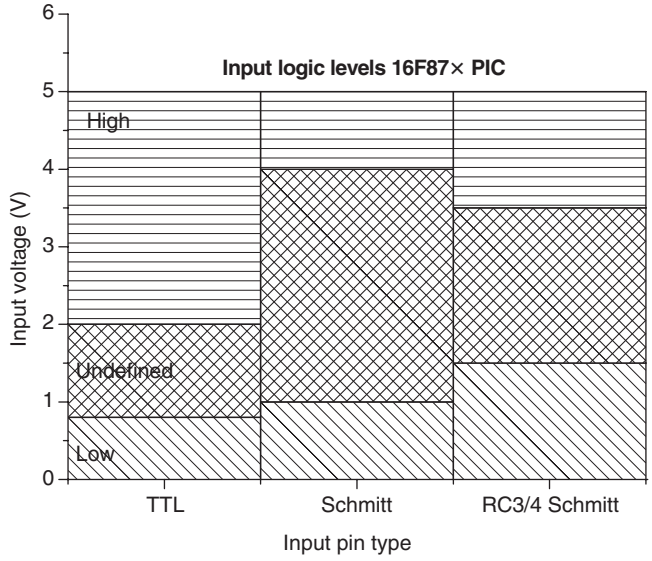


Figure 20.3: Input Logic Level Comparison

If we wish to obtain a similar plot with a 16F877, however, we must somehow determine the logical state—high or low—of the input pin corresponding to the input voltage to. The

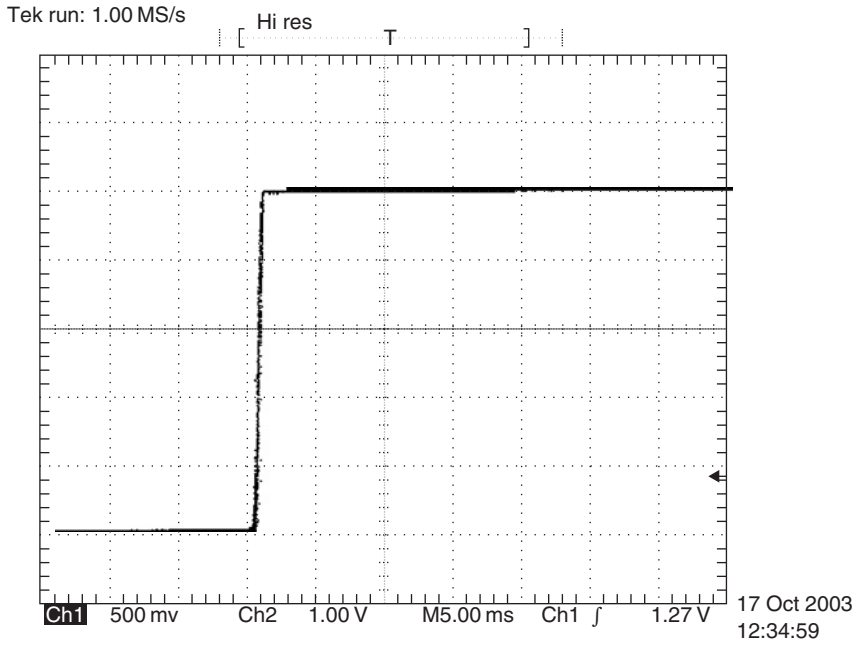


Figure 20.4: SN7400 TTL Logic Levels Ch1: X-axis (Gate Input); Ch2: Y-axis (Gate Output)

easiest way to do this is simply echo the input pin's value to an output pin. In pseudo-code the algorithm is:

```
ReadPin:
Read Input Pin - If Input Pin <> 1 Make Output Pin = 0
Read Input Pin - If Input Pin <> 0 Make Output Pin = 1
Goto ReadPin
```

It turns out that to be useful, the process of reading the input pin and making the output pin must be done more quickly than possible using MBasic, we'll add some high speed assembler into the mix. The reason for the unusual pseudo-code structure (the not-equal operator) will become clear when we look at the real code.

```
;Program 20.01

Input B0
Output B1

Main

ASM
{
ReadIn
        btfss   PortB,0         ;If RB0=1 then skip setting it to 0
        bcf     PortB,1         ;make RB1=0
```

```
        btfsc  PortB,0        ;If RB0=0 then skip setting it to 1
        bsf    PortB,1        ;make Rb1=1
        GoTo   ReadIn         ;Repeat the loop
}

GoTo Main
End
```

Program 20.1

Don't worry if you don't understand the assembler portions of Program 20.1, as we will learn more about mixing assembler and MBasic later. However, the actual program tracks the pseudo-code. We first make RB0 an input and RB1 an output using MBasic. Then, the main program is an endless loop.

The `btfss PortB,0` statement reads the 0'th bit of Port B (RB0) and if it is "set," i.e., if it reads high, the immediately following statement is skipped. If it is low, the statement immediately following is executed. (The mnemonic is Bit Test File, Skip if Set, or `btfss`).

```
btfss       PortB,0       ;If RB0=1 then skip setting it to 0
bcf         PortB,1       ;make RB1=0
```

The above code reads RB0 and branches, depending on its value. If RB0 is set—that is, RB0 = 1, then the next statement `(bcf PortB,1)` is skipped. Conversely, if RB0=0, then `bcf PortB,1` is executed. The `bcf`—or bit clear file—operator clears or sets to zero a bit, in this case, RB1. These two statements, therefore, read RB0 and set RB1 to zero if RB1 is zero.

```
btfsc       PortB,0       ;If RB0=0 then skip setting it to 1
bsf         PortB,1       ;make Rb1=1
```

The next two statements perform the inverse operation. RB0 is read a second time and tested—but this time for the clear state—using the `btfsc` operation. (The `btfsc` operator works just like `btfss`, except the next instruction is skipped if the tested bit is clear.) If RB0 is high, then the operation `bsf PortB,1` is performed, thereby setting RB1.

Execution continues with the `GoTo ReadIn`, which loops back to reading RB0.

To terminate this program, power must be removed from the PIC, or another program written into its memory.

Program 20.1 isn't the fastest way to transfer an input pin level to an output pin, but we'll look at more efficient techniques later. It also uses multiple read actions, thereby creating the possibility of mishandling if the input changes value between the two read operations.

With a 20-MHz clock, this program has a 1.2 μs operating cycle. (The SN7400 gate, performing these tasks in hardware, requires less than 10 ns, 120 times faster than the PIC's software.) Hence, when we examine the input/output relationship with the same set up we

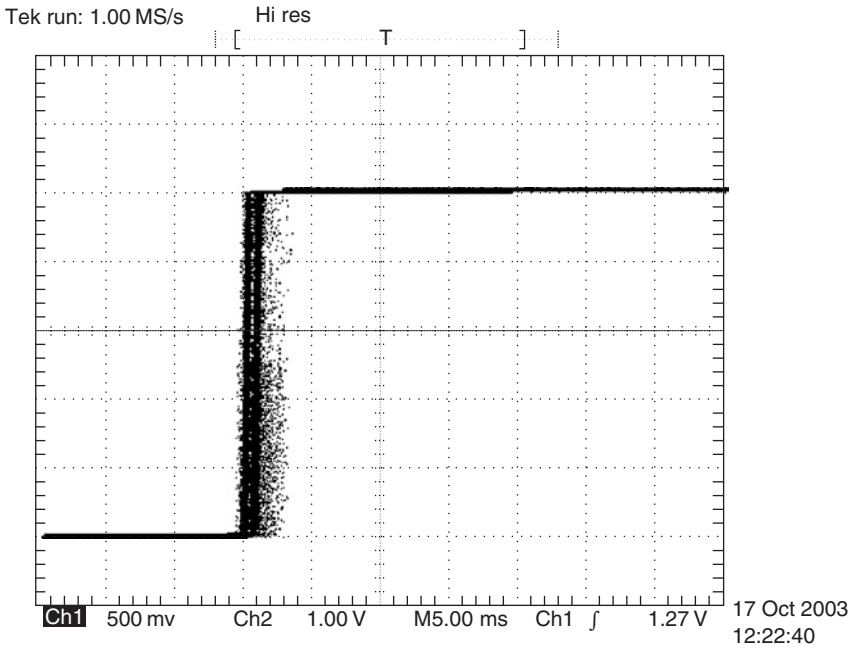


Figure 20.5: 16F876 TTL Logic Levels Ch1: X-axis (RBO Input); Ch2: Y-axis (RB1 Output)

used for the SN7400 gate, we will expect to see horizontal smearing, where the output lags the input by this delay. See Fig. 20.5. We see $V_{THRESHOLD}$ is approximately 1.5 V, quite close to the value measured for the SN7400 true TTL gate.

Modifying Program 20.1 to accept RC0 as the input and running the same input/output sweep, we see in Fig. 20.6 the hysteresis of the Schmitt trigger. The low-to-high transition occurs at approximately 3.1 V, while the high-tolow transition occurs at approximately 1.8 V. The beauty of separate high-low and low-high transition levels is noise rejection. Suppose the input signal has noise riding on it, perhaps induced from high-speed logic chips on the same board. Once a transition from one state to the other has occurred, it takes a 1.3 V noise excursion to cause a reverse transition. As we shall see later, this hysteresis adds greatly to noise rejection.

We understand Microchip's decision to use TTL mimicking inputs as the product of backward compatibility with earlier PICs and access to the huge base of TTL devices. But, why has Microchip designed the rest of its PIC inputs with Schmitt trigger inputs instead of normal CMOS inputs, such as that seen in Fig. 20.7 for a CD4001BE quad NOR gate? After all, PICs are CMOS devices so it makes sense to give them standard CMOS characteristics, right?

If Microchip thought its PICs would communicate only with other integrated circuits installed on the same board, it likely would have adopted standard CMOS input levels. But, PICs often must communicate with the real world, through sensors and switches, operating in a much less

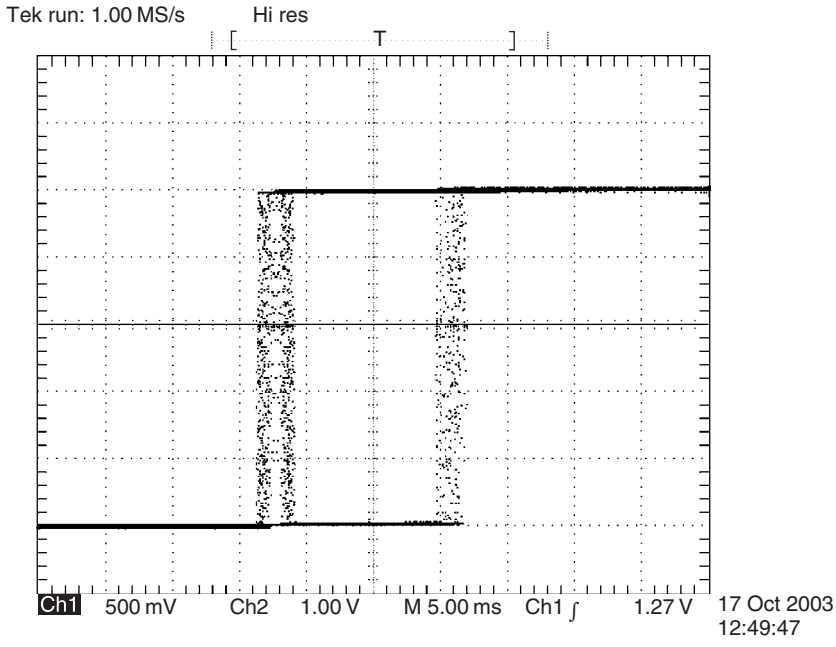


Figure 20.6: 16F876 Schmitt Trigger Logic Levels Ch1: X-axis (RC0 Input); Ch2: Y-axis (RB1 Output)

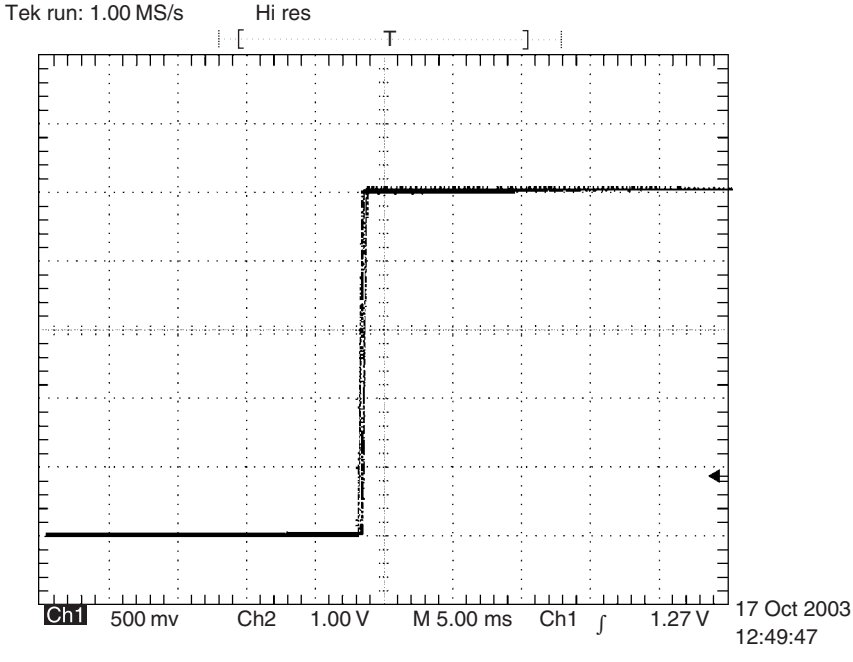


Figure 20.7: CD4001BE Quad 2-Input NOR CMOS Logic Levels Ch1: X-axis (Input); Ch2: Y-axis (Output)

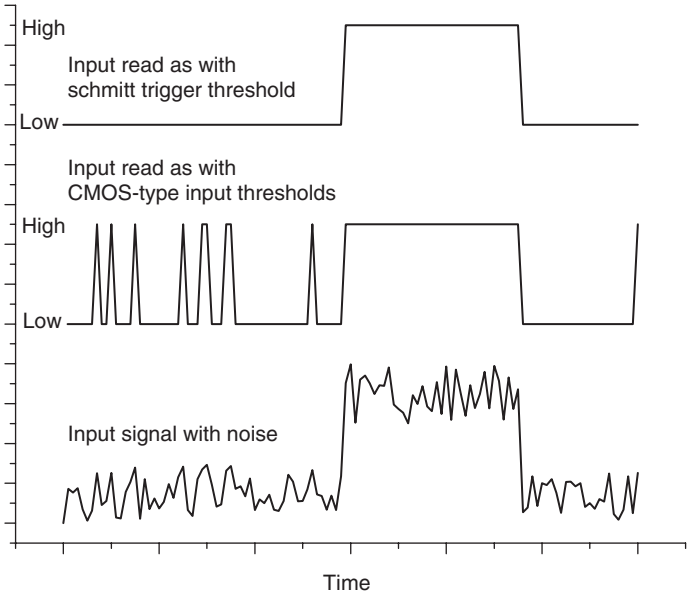


Figure 20.8: Noisy Input Read Differently by Standard CMOS and Schmitt Inputs

benign environment where the noise rejecting properties of the Schmitt trigger come to the fore. Figure 20.8 illustrates the ability of a Schmitt trigger input to correctly read an input signal in the presence of large noise voltages.

In order to ensure that levels are correctly read, regardless of the input type, we should aim for logic 0 input levels not exceeding 0.8 V and logic high levels of at least 4.0 V. If we meet these objectives, all input port types will correctly read the input levels. If these levels cannot be achieved, it will be necessary to further investigate the design to ensure reliable data transfer.

Regardless of their type, PIC input pins represent a high input impedance to the outside world. As reflected in Fig. 20.9, the only current that flows in or out of an input pin is due to leakage and does not exceed 1 μA. (In the 16F876 family, pin RA4's leakage current may be up to 5 μA.) For low impedance circuits we may safely consider an input pin as an open circuit, with zero current flow in or out of the pin. A high impedance input pin carries with it the possibility of static damage, as even a small static charge produces high voltages into a high impedance pin. Although the clamping diodes to V_{DD} and V_{SS} help prevent damage, it's still a good idea to follow anti-static precautions when handling PICs. It also means that input pins should be protected when exposed to outside world voltage and currents. We will briefly cover some of these real world issues in this and later chapters.

The exception to this assumption relates to the software enabled internal pull-up resistors on pins RB0…RB7. If the weak pull-up feature on Port B is enabled via MBasic's procedure `SetPullUps`, the input pins are connected to V_{DD} through an internal 20 Kohm resistor.

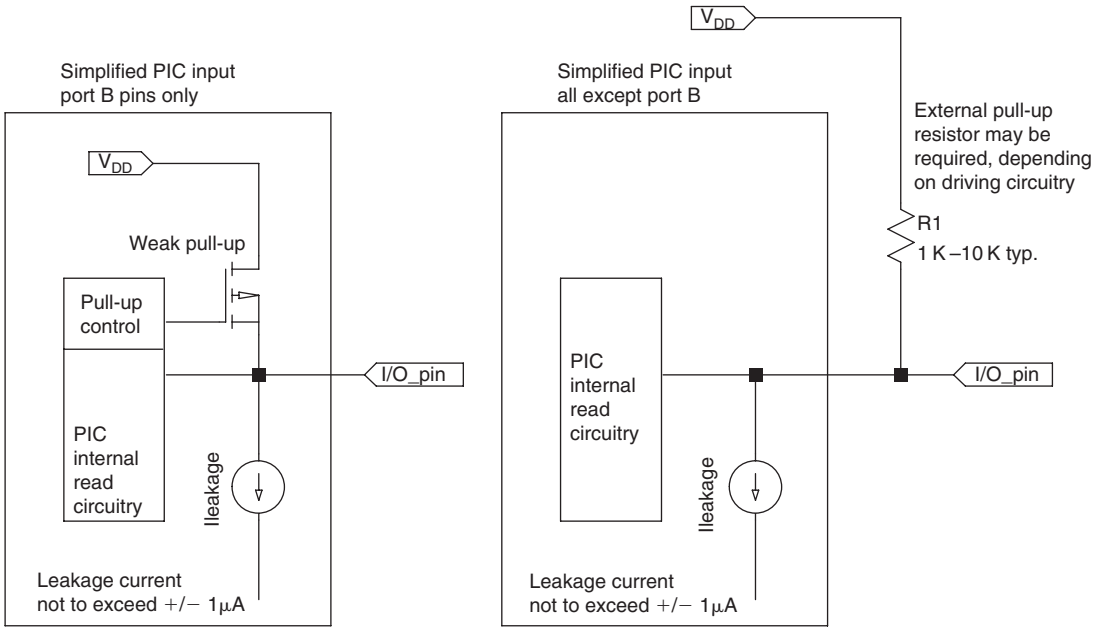
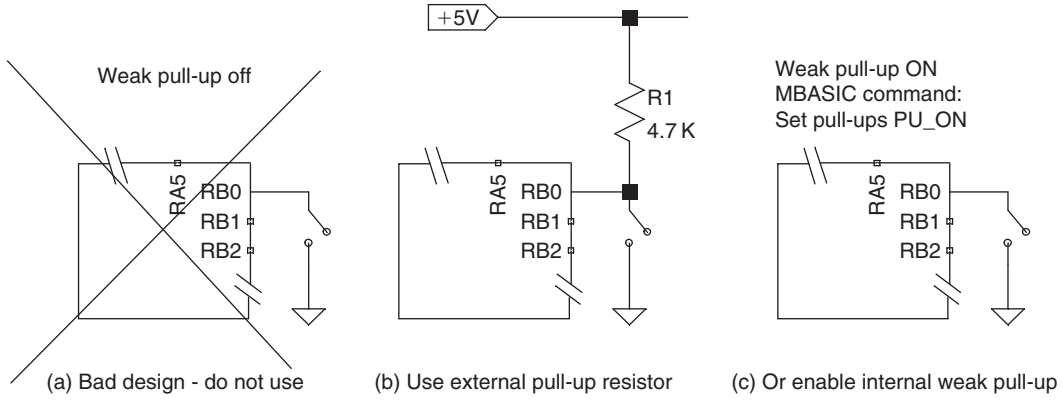


Figure 20.9: Simplified Model of Input Pins

If enabled through `the SetPullUps PU_On` or disabled through `SetPullUps PU_Off` procedure, the action applies to all PortB pins. In this case, the input pin sources 250 μA.

We've referred to pull-up resistors without explaining why and where they are used. Suppose we wish to read a switch and determine if it is open or closed. If we simply connect the switch to an input pin, as in Fig. 20.10a, we cannot be assured of RB0's status when the switch is open. Certainly, if the switch is closed, RB0 will be at ground potential and will be read



Figures 20.10a,b,c: Pull-up Required to Read Switch

as a logical low. When open, however, the voltage at RB0 results from the PIC's internal random leakage current, plus whatever stray signal that outside circuitry may induce in the connections between the switch and RB0. The result may be a logical high, or low, and we are without any assurance that the switch's position will be correctly or consistently read.

If instead, we connect RB0 to V_{DD}, either through the internal pull-up source (Fig. 20.10c) or through an external pull-up resistor (Fig. 20.10b), when the switch is open RB0 will be "pulled up" to V_{DD} and the switch position will correctly be read as a logical high.

20.2 Switch Bounce and Sealing Current

Beyond assuring that an open switch is correctly read, a pull-up resistor provides the switch with enough current to reliably operate. In addition to the mechanical wiping action of switch contacts, a small DC current greatly assists in maintaining reliable conduction between moving switch contacts. And, if the switch is connected through wiring with mechanical splices and crimp or screw pressure terminals, the current helps clean oxidizing film from the mating conductors. From telephone terminology, we often refer to this as a "sealing" current or a "wetting" current. Hence switch circuits are often referred to as wet—carrying a current well in excess of that necessary to pull an input pin high or low—or dry—carrying only a minimal signal current. For most mechanical switches, unless we are trying to save power such as in a battery powered system, select the pull-up resistor to supply 5 to 20 mA current flow through the switch when closed. For 5 V systems, use a pull-up resistor of 1 Kohm to 250 ohms. Finally, the smaller the pull-up resistor value, the less stray voltage will be induced into the wiring connecting the PIC with the switch.

When a mechanical switch operates, the same "contact bounce" phenomena we observed earlier with relay contacts is seen. In most switches, the contact separation operation ("break") is relatively clean, but the contact closure ("make") exhibits multiple bounce events. Fig. 20.11 shows nearly 1.5 ms is required to reach a steady state closed condition in the microswitch SPDT switch being tested.

Why should we be worried about switch bounce? The answer is that in many cases, we don't care. Bounce isn't a concern, for example, where a baud rate switch is read only at start-up, or where a limit switch senses the position of a part and activates a software stop sequence. In the first case, the switch isn't operated while the program executes and in the second multiple activations of stop sequence isn't a concern. Suppose, however, the switch counts the number of times an action is performed. Clearly we want one switch operation to increment the count once, not once for each of a dozen or so bounces. To prevent switch bounce from repeatedly triggering an operation, we must debounce the switch.

We can debounce a switch either with external electronics, or in software.

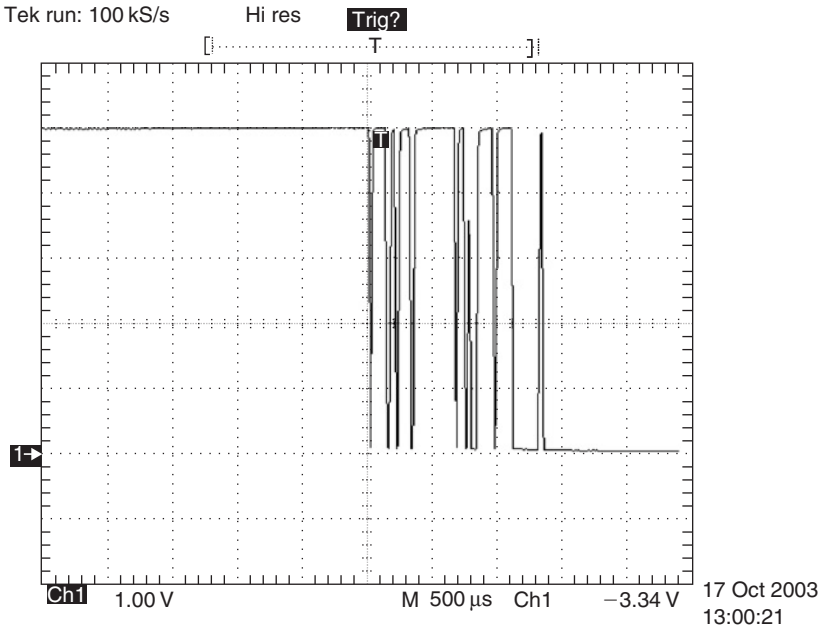


Figure 20.11: Contact Bounce Upon Closure of Microswitch

20.3 Hardware Debouncing

Although specialized integrated circuit "debouncers," such as Fairchild's FM809 microprocessor supervisor devices, exist we'll look at a simple circuit described in a recent issue of *EDN Magazine*, as shown in Fig. 20.12 [Ref. 1].

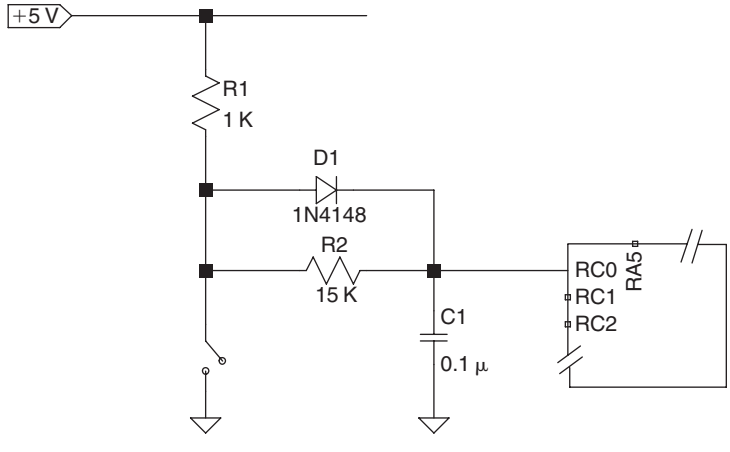


Figure 20.12: Hardware Debounce

When the switch is open, C1 charges to V_{DD} and RC0 is read as high. When the switch makes, C1 is discharged through R2 (D1 is reversed biased and may be neglected) and will be read

as low when the voltage across C1 drops below high-to-low transition voltage, approximately 1.8 V. If the time constant of R2-C1 is long compared with the individual bounce intervals, the decay will be smooth and only one transition through $V_{THRESHOLD}$ will occur. But, even if spikes of several hundred millivolts occur at RC0 the output will stay low, as in order to change its read state, RC0 must see the low-to-high transition voltage of approximately 3.1 V.

When the switch is closed, the input pin is connected to ground through R2. The leakage current from the PIC input pin is rated not to exceed 1 μA, so in the worst case with R2 at 15 Kohm, the input pin will be at 0.015 V, well within the logical low range.

When the switch is opened after closure, C1 charges through R1 and R2 in parallel with D1. Until the voltage across C1 reaches 0.7 V from V_{DD}, C1 charges mostly through R1 and D1. Hence, the charge cycle is significantly shorter than the discharge cycle. This design assumes that the switch has bounce problems only on make and therefore little or no anti-bounce effect is required on break.

The relationship between C1, R2 and the desired debounce time T_B is given by:

$$R2C1 = -\frac{T_B}{\text{Ln}\dfrac{V_{THRESHOLD}}{V_{DD}}}$$

If we design for a Schmitt input where $V_{THRESHOLD}$ for a high to low transition is approximately 1.8 V, and if we make C1 0.1 μF, a convenient value, we may simplify this equation and solve for R2 in terms of T_B:

$R2 \approx 10T_B$

R2 is in Kohm, and TB is in milliseconds. If the desired debounce time is 1.5 ms, R2 should be 15 Kohm. Figure 20.13 shows how well this simple circuit removes the bounce from the same switch shown in Fig. 20.11.

To study the effect of the debounce circuit, we simply make RB0 equal to RC0 and repeat the test in an endless loop.

```
;Program 20.02
;Program to echo read of C0 to B0
;Variables
Temp       Var       Byte

Input      C0
Output     B0

Main
            PortB.Bit0 = PortC.Bit0
GoTo Main
End
```

Program 20.2

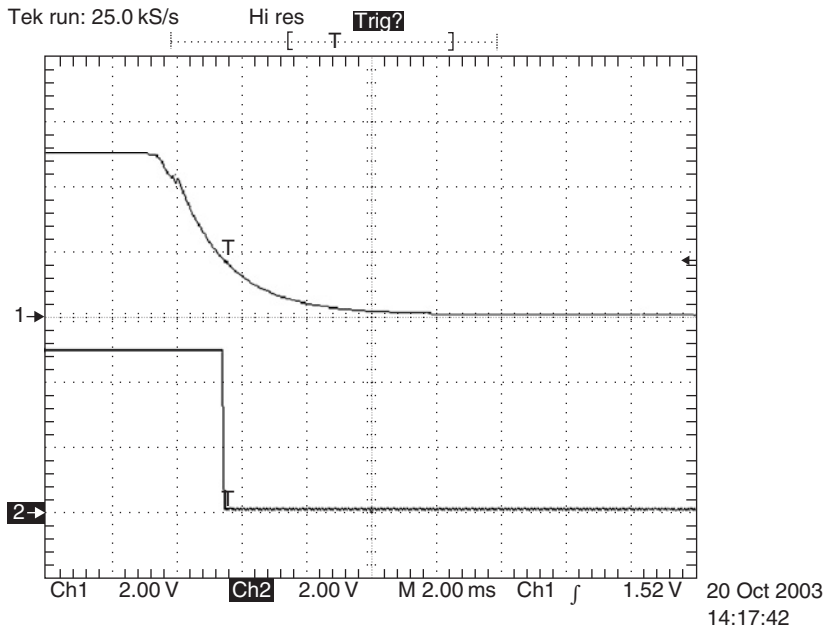


Figure 20.13: External Debounce Circuit Operation Ch1: RC0 PIC Input; Ch2: RB0 PIC Output

The key statement in the program is `PortB.Bit0 = PortC.Bit0` where the assignment operator forces a read of RC0 and a subsequent write of the resulting value to RB0. This read/write sequence is repeated every time the loop executes. Program 20.2 runs quite a bit slower than Program 20.1, with the switch pin RB0 being read once every 68 μs, compared with the once every 1.2 μs in the assembler code.

20.4 Software Debouncing

We may also debounce the switch in software. In the simplest case, we simply pause program execution to allow the switch contacts to stabilize.

```
;Program 20.03
;Variables
Temp      Var      Byte

Input     B0
Output    B1

Main
    PortB.Bit1 = PortB.Bit0
    Pause 2

GoTo Main
End
```

Program 20.3

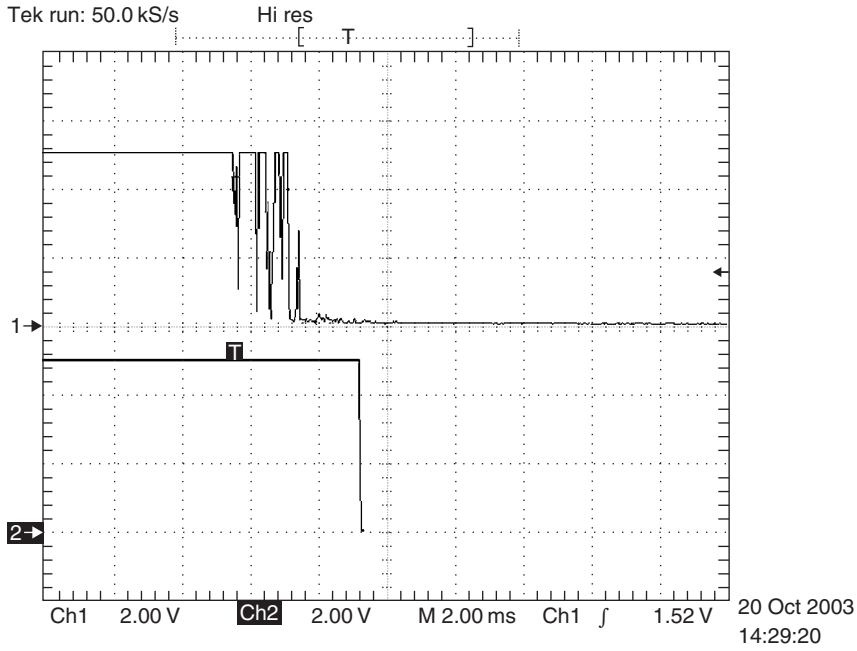


Figure 20.14: Debounce via Pause2 Ch1: RB0 PIC Input; Ch2: RB1 PIC Output

Figure 20.14 shows the effect of pausing 2 m after reading the input pin. The actual reading of RB0 in the statement `PortB.Bitl = PortB.Bit0` still occurs very quickly; in 0.2 μs for a 20-MHz clocked PIC, even though the entire statement requires approximately 14 μs to execute. However, these quick reads are now spaced approximately every 2 ms apart. Hence, if the read happens to detect a high during the bounce period, the output stays high until at least the next read period. By that time, the switch will have had time to stabilize and the next read will correctly be a low. If, on the other hand, RB0 reads a low during a bounce, the output drops to low immediately. The next read, 2 ms later, will also read a low, since the switch will have by then stabilized. In either case the switch is successfully debounced.

MBasic also provides a built-in switch debounce procedure `Button`, invoked as:

```
Button pin, downstate, delay, rate, bytevariable, targetstate, address
```

Pin—is the address of the pin to be read and may be either a pre-defined constant, such as `B0`, or a variable set to the pin's address.

Downstate—`Button` tests for a specific input, either high or low. `Downstate` is a constant or variable (either 0 or 1) that defines which condition represents the switch being operated. If the switch operation causes a low to be applied to `pin`, `Downstate` should be 0; if operation causes a high, `Downstate` should be 1.

Delay—a constant or variable (0…255) that controls how long `Button` waits before starting auto-repeat. Auto-repeat means that `Button` acts as if the switch is repeatedly cycled, similar to the repeated letters obtained when you hold a key down on your computer keyboard. If `Delay` is set to 0, debounce and auto-repeat are disabled; if `Delay` is set to 255, debounce is enabled, auto-repeat is disabled. Delay is in units of the time `button` is called. For example, if `button` is used in a loop that requires 2 ms to execute, and if `Delay` is set to 10, the net delay is 20 ms.

Rate—is a constant or variable (0…255) that determines the time between auto-repeats. `Rate` is also in units of time `button` is called.

ByteVariable—is a byte variable used by `Button` for workspace. You must declare this variable.

TargetState—is a constant or variable (0 or 1) that specifies which state `pin` must be in to cause program execution to branch to `Address`.

Address—A label to which execution will jump upon `pin = TargetState`.

Program 20.4 illustrates how `Button` may be used.

```
;Program 20.04
;Variables
Temp    Var        Byte

Input   B0
Output  B1

Main
        High B1
        Button B0, 0, 255, 100, Temp, 1, UpButton
        GoTo Main

UpButton:
        Low B1
        Pause 10
GoTo Main

End
```

Program 20.4

Program 20.4 reads the status of a switch connected as shown in Fig. 20.15. Program 20.4 makes an output pin, B1, track the debounced input pin B0. We first set B1 high and test to see if B0 is low via the `Button` debounce procedure. If the switch is open, B0 remains high, and program execution resumes after `Button` with the `GoTo Main` statement which loops back to setting B1 high and reading B0. If, however, the switch is closed and `Button` reads

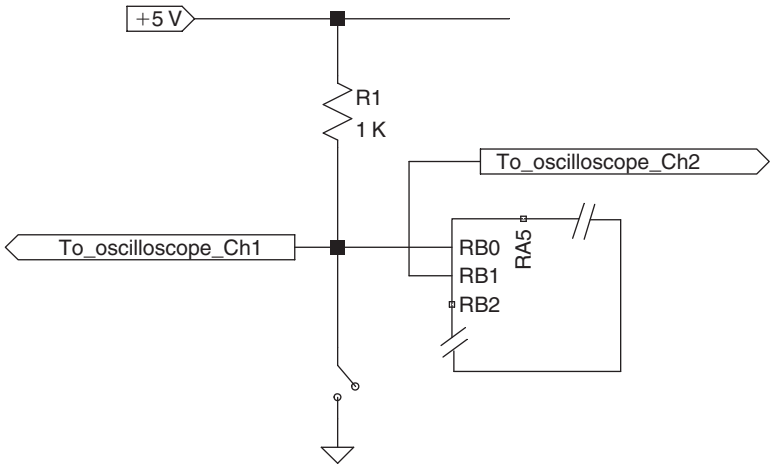


Figure 20.15: Switch Connection

a low on B0, program execution branches to `UpButton`. `UpButton` is a dummy procedure that simply makes B1 low for 10 ms, after which execution resumes with setting B1 high and reading B0 through the `GoTo Main` statement. If you wish the button press to do something useful, the appropriate code would be at the procedure `UpButton`.

Figure 20.16 shows the output of Program 20.4 where `Button` is set for automatic repeat operation with `Delay 250` and `Rate 250`. Since the program loop reading `Button` takes 140 μs to execute, it will start repeating after 250 × 140 μs, or 35 ms. It will continue to repeat

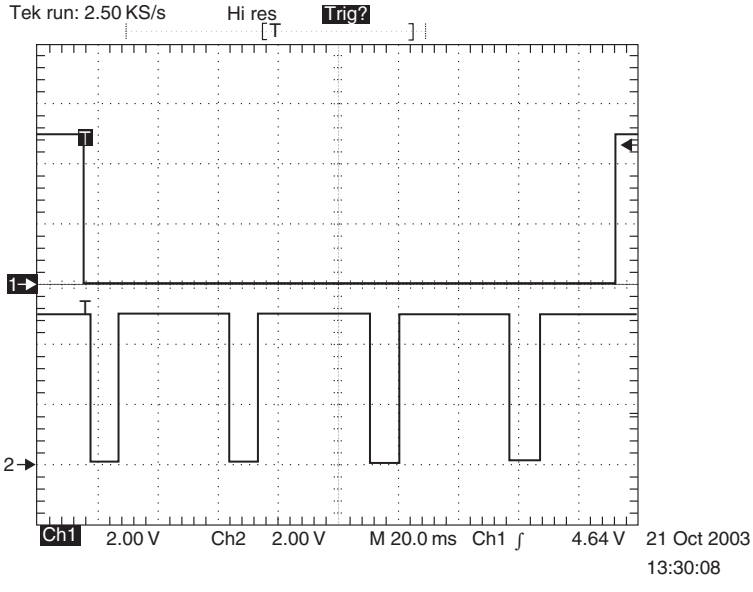


Figure 20.16: Button with Delay = 250, Rate = 250 Ch1: RB0 PIC Input; Ch2: RB1 PIC Output

every 250 × 140 μs or 35 ms. Upon activation (either initially or following each auto-repeat) program execution branches to a simple routine that drops the output low for 10 ms.

In many cases, we need debounce only, with one operation for every button push. Figure 20.17 shows the effectiveness of `Button` at debouncing.

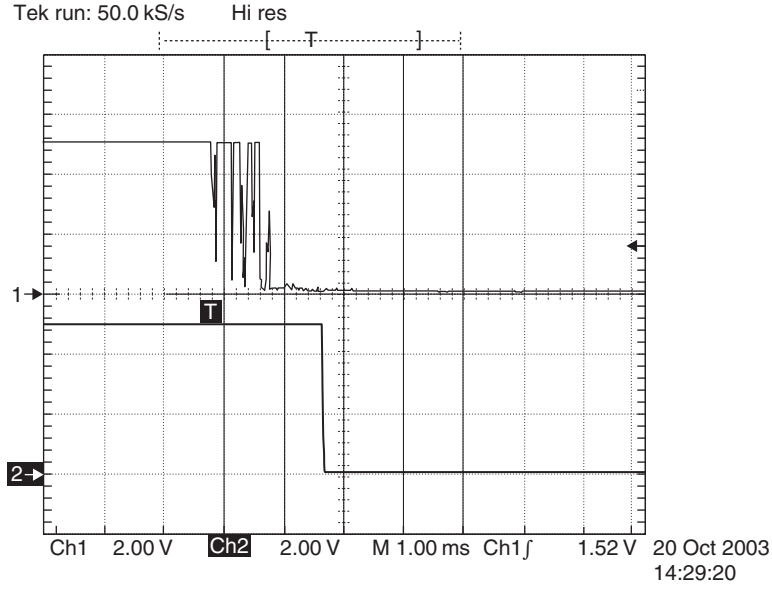


Figure 20.17: Button with Delay = 255, rate = 100 Ch1: RB0 PIC input; Ch2: RB1 PIC Output

Figure 20.18 shows three possible switch connections. SW1 and SW2 correspond to State 0 and State 1 designs in MBasic's User's Guide button discussion. When SW1 is closed, RB0 goes low while when SW2 is closed, RB1 goes high. If SW2 is installed on the same printed circuit board as the PIC, the connection shown in Fig. 20.18 is acceptable. However, if the switch is remote from the PIC, perhaps even connected by a lengthy cable, the design associated with SW2 requires unprotected +5 V to be run to SW2, thus exposing it to possible problems. If a normally closed switch is available—perhaps the NC contacts of a SPDT switch—the alternate configuration of SW3 is better. In the worst case should either side of SW3 be inadvertently grounded no damage will occur.

20.5 Isolated Switching

Just as with switching isolated loads, there are occasions when you may wish to read a switch closure that cannot have a common ground with your PIC circuit. Or, you may have a long run of cable between the switch and the PIC and wish to ensure against stray induced voltages or ground currents.

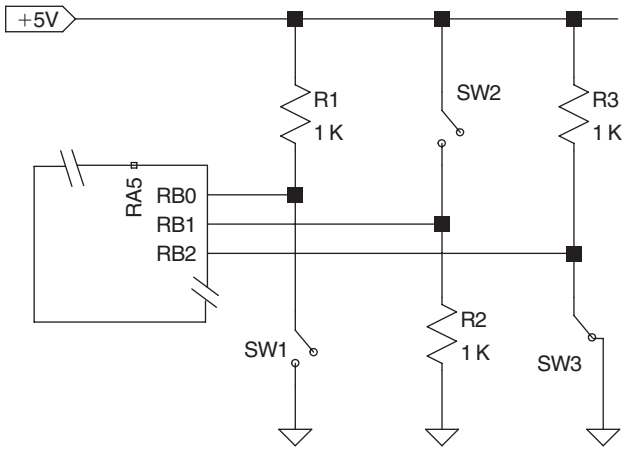


Figure 20.18: Alternative Switching Connections

We can apply several techniques developed in Chapter 19 isolating PIC inputs from switches. Figure 20.19 implements two simple approaches. RB0 is switched through a relay, while RB1 is switched through an optical coupler. Both RB0 and RB1 need debouncing treatment.

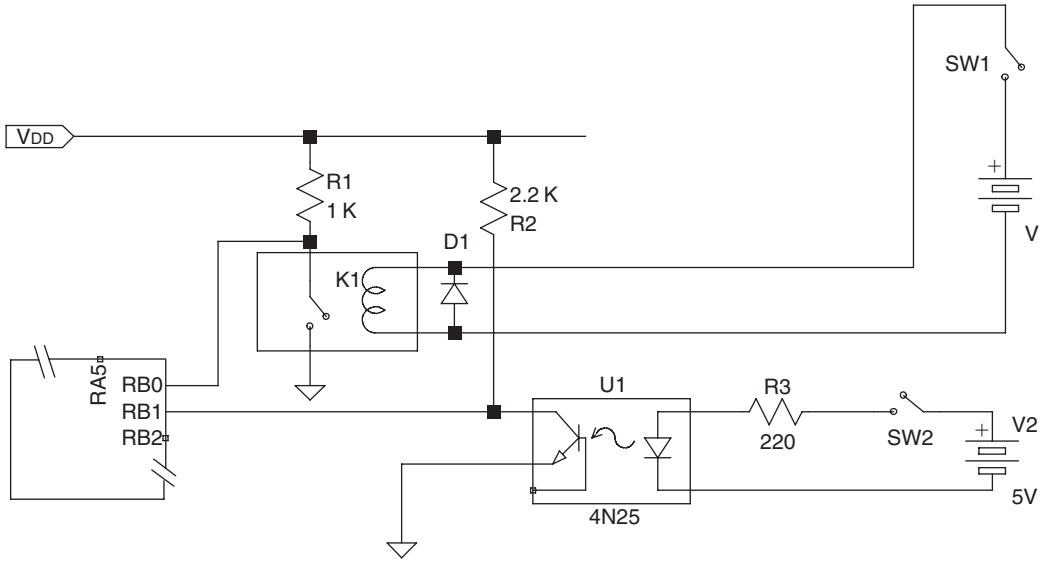


Figure 20.19: Isolating Remote Switches from a PIC

SW1 makes or breaks current through relay K1, thereby operating the contacts connected to RB0. When SW1 is open, RB0 is high; when SW1 is closed, RB0 is low. R1 is selected to permit a reasonable wetting current, 5 mA in the example, to flow through K1's contacts. D1 is a snubbing diode to reduce arcing when SW1 is opened. For small relays operated from low voltage sources, D1 may be safely omitted.

SW2 makes or breaks current through the LED half of optoisolator U1. When illuminated, U1's LED saturates the phototransistor and pulls its collector—and RB1—low. When not illuminated, U1's phototransistor is off and RB1 is taken high R2. R3 should be selected to ensure the phototransistor is fully saturated when the LED is illuminated, with 15 to 20 mA being a typical current value for a 4N25 optoisolator. R2 should be selected to ensure the phototransistor's collector voltage is under 0.5 V when the LED is illuminated. For a 4N25, we can accomplish this by a collector current of 1 mA or so. The component values in Fig. 20.19 result in an LED-on voltage of 0.15 V at the phototransistor's collector.

20.6 Reading a Keypad

So far, we've looked at isolated switches, with one pin for each switch. We'll look at one type of multiple switch assembly—the keypad—in this chapter.

Keypads, such as those found on telephones and calculators, are almost always matrix switches, as illustrated in Fig. 20.20. Pressing a button establishes a connection between its column and row terminals. For example, pressing the "5" button in Fig. 20.20 connects column 2 with row 2. Many inexpensive keypads, including the Velleman 16KEY model we'll use in our experiments, are constructed with a conductive elastomer design rather than physical make/break switches. These switches typically have an on resistance of 100–200 ohms, compared with the milliohm range resistance found in mechanical switches.

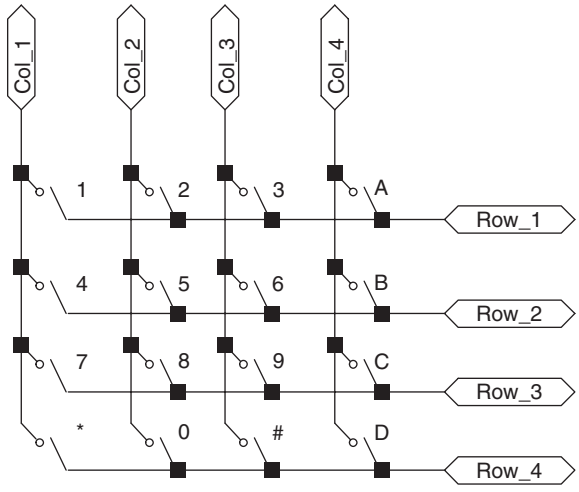


Figure 20.20: Typical 16-Position Keypad Switch

Let's see how we might go about reading the keyboard. Suppose we connect it as shown in Fig. 20.21. Then, we sequentially make pins B4…B7 output a high and after each high, we read pins B0…B3 to see if any are high. If so, we can determine the row number and column number and from that identify which button has been pressed. This process is illustrated in Fig. 20.22.

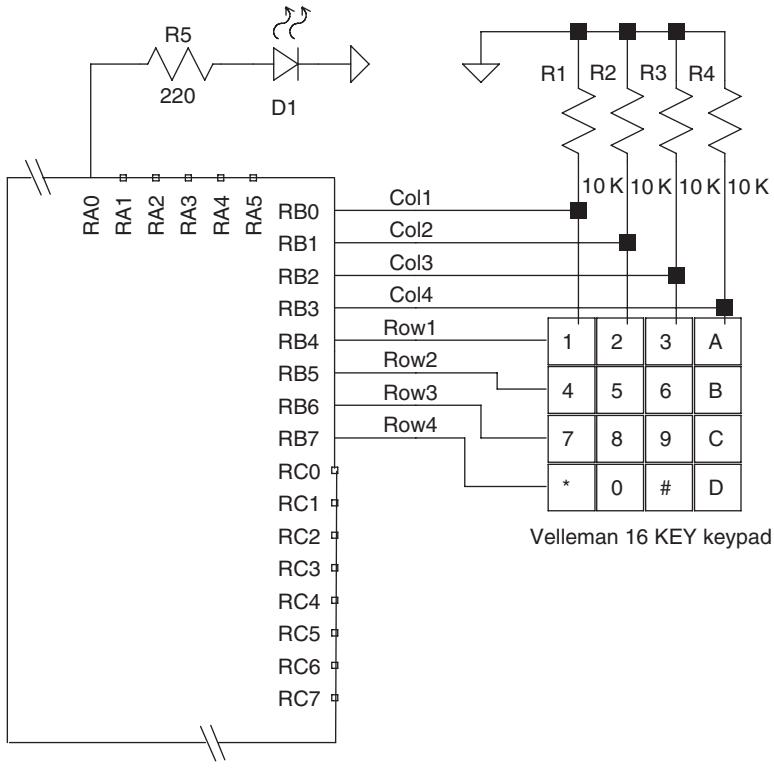


Figure 20.21: Keyboard Connection to PIC

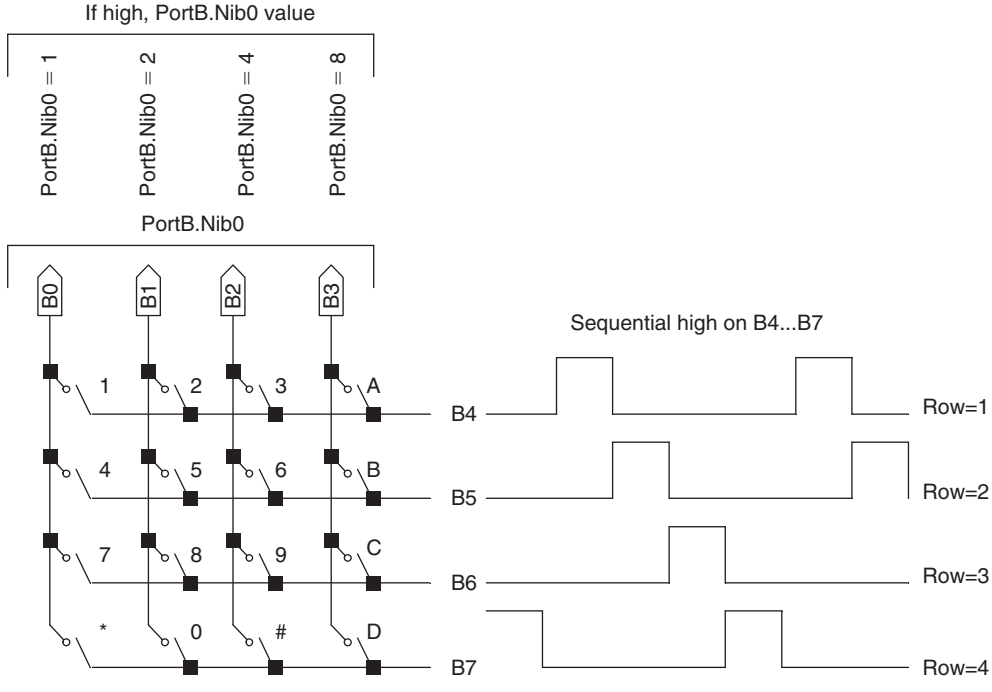


Figure 20.22: Reading the Keypad

R1…R4 are "pull-down" resistors and ensure that input pins RB0…RB3 do not drift upwards towards a logical high from internal leakage currents or possible leakage across the keypad's open contacts. R5 limits the current through LED D1 to approximately 10 mA when RA0 is high.

In pseudo-code, we would:

```
ScanKeyboard
      For Row = B4 to B7
              High Row
              Pause 10
              Column = PortB.Nib0
              If Column <> 0 then key must have been pressed
              So go to a subroutine to deal with the press
              If not, then keep scanning the row
      Next Row
GoTo ScanKeyboard
```

We can determine the row number directly since `RowNumber = Row-B4+1`. `Column`, on the other hand, is read as 1,2,4 or 8. We have several ways to calculate the `ColumnNumber` from `Column`, but we'll use a simple one now. We define an auxiliary array, `TempArray` and assign `TempArray(1)=1, TempArray(2)=2, TempArray(4)=3` and `TempArray(8)=4`. Then, `ColumnNumber = TempArray(Column)`.

Program 20.5 reads the keypad and then flashes an LED connected to RA0 in the sequence <flash row number> pause <flash column number>.

```
;Program to test keypad reading
;
;the keypad output pulses a LED with the row/column value. LED is
;powered by the port and goes to ground. For example pressing the
;"8" key pulses the LED in a 3 - pause - 2 pattern.

Column      Var         Byte     ;Counter for For/Next loops
Row         Var         Byte     ;Counter for For/Next loops
Temp        Var         Byte     ;holds the column binary value
LED         Con         A0       ;Have an LED hanging off A0
RIndex      Var         Byte     ;Holds the Row Value 1..4
CIndex      Var         Byte     ;Holds the Column value 1..4
i           Var         Byte     ;Counter for various For/Next loops
TempArray   Var         Nib(9)   ;Use for conversion

;
;
;Initialize
;==========
Column = 0
```

```
Row = 0
Temp = 0

Output LED   ;set up the LED pin
Low LED      ;we want the LED off

For Column = B0 to B3
    Input Column            ;Set these for input
Next ;Next Column

For Row = B4 to B7
    Output Row          ;Set these for output
    Low Row             ;we will pulse a high, start them out as lows
Next ;Next Row

;holds the actual column number. 1,2,4 & 8 are only legal values
;so other elements of the array can have random values
TempArray(1) = 1
TempArray(2) = 2
TempArray(4) = 3
TempArray(8) = 4
;
;
;Main Here we read the keypad. Put a 1 sequentially on the rows
;==== and see which column has the high.
;Since the called subroutine takes a long time to run and is only
;triggered once, no additional debounce is required. This may not
;be the case for other called subroutines.

Loop:
   For Row = B4 to B7                   ;Scan the rows
       High Row                         ;pulse a 1 across each row
       Pause 10                         ;wait a bit
       Temp = PortB.Nib0                ;read all 4 columns at once
       Low Row                          ;restore the low
       If Temp > 0 Then GoSub LED_On                    ;button pushed
   Next ;Row
GoTo Loop ;check for more keypresses

;Execute upon keypress --at the moment it just flashes an LED
;-----------------------------------------------------------------
LED_On:
      ;pulse LED number of row times
      Rindex = Row - B4 + 1 ;the row number, from 1..4

      For i = 1 to RIndex ;now flash the LED
          High LED
          Pause 150
```

Program 20.5: Continued

```
            Low LED
            Pause 150
        Next          ;Next i

        ;now we convert column value (held in Temp) to column number
        ;Column value is 1,2,4,8 corresponding to Column 1,2,3 or 4.

        CIndex = TempArray(Temp) ;conversion via array indexing

        Pause 500      ;pause to permit the user to distinguish rows
                       ;from columns when watching the LED

        ;same approach to flash the column number using the LED
        For i = 1 to CIndex
            High LED
            Pause 150
            Low LED
            Pause 150
            Next     ;Next i
Return ;LED_On

End
```

Program 20.5: Continued

The central portion of the program is the key scan loop, which implements the pseudo-code almost directly:

```
Loop:
   For Row = B4 to B7        ;Scan the rows
       High Row              ;pulse a 1 across each row
       Pause 10              ;wait a bit
       Temp = PortB.Nib0     ;read all 4 columns at once
       Low Row               ;restore the low
       If Temp > 0 Then GoSub LED_On ;button pushed
   Next ;Row
GoTo Loop ;check for more keypresses
```

The subroutine `LED_On` is a dummy routine that simply flashes an LED to show the row and column numbers of the button being pressed. You may wish to use this keypad routine as one building block in a more complex and useful program.

Reference

[20.1] Mancini, Ron, "Examining Switch-Debounce Circuits," EDN, p. 22, Feb. 21, 2002.

This page intentionally left blank

CHAPTER 21

Introductory Stepper Motors

Stepper motors, as the name implies, rotate in discrete steps. Most conventional motors are continuous; if we make a mark on the shaft of a conventional motor and if we were able to precisely control the motor's excitation, we could make the shaft move to any angle. 123.456 degrees from the starting mark is just as achievable as 321.765 degrees. Of course, practical considerations make this degree of precision unlikely in a real motor. A stepper motor's shaft, in contrast, is moveable to only certain, pre-defined angles. A 48-step stepper motor, for example, may be positioned only in increments of 7.5 degrees (360º/48). Hence, we may command the shaft to go to 7.5º (one step) or 15º (two steps) but not to 8.432º. (Later in this chapter, we'll see ways to step the shaft rotation one-half or a smaller fraction of the motor's normal step size, so the difference between conventional and stepper motors blurs.)

Why would we want a motor that only moves in steps? Suppose we wish to move an ink jet printer's print head across the paper, and that we must position the print head with an accuracy of 0.001″. We'll assume the print head is attached to a nonslip, no stretch toothed belt and that the belt is driven through a toothed pulley system attached to a positioning motor. Let's attach a 200-step stepper motor to the pulley, through 5:1 step down gears, and pick the pulley size so that 1,000 steps of the motor moves the printing head 1.000 inch. Each motor step therefore corresponds to 0.001″ and we may position the print head to 4.567″ by initializing the print head at the start position and then advancing the motor by 4,567 steps. This "open loop" solution is much cheaper than a "closed loop" design that continuously monitors the position of the print head and stops the advance when the target position is reached.

We'll concentrate on stepper motor fundamentals in this chapter.

21.1 Stepper Motor Basics

21.1.1 Introduction

Let's start by considering the pluses and minuses of stepper motors

Stepper Advantages	Stepper Disadvantages
• Precise control of position-one pulse advances one step, permitting open loop control. • Full torque from zero RPM. • Step accuracy typically 5% of step size, but errors are non-cumulative. • No brushes or other current carrying moving parts; lifetime is therefore limited only by the bearing life. • Easy to interface with microcontrollers. • The motor is self-locking if the windings are powered while not rotating; even if unpowered, most designs have appreciable residual torque.	• Limited speed. • Certain step rates may mechanically resonate with the motor causing loss of torque and undesired vibration. • Large steppers are not readily available. Most steppers are in the 0.0001 HP to 0.05 HP range. • Torque decreases as rotational speed increases; if the motor stalls, position location is lost.

21.1.2 Operation

Let's see how a stepper motor works. We'll consider a highly simplified motor that doesn't match real motor designs but provides a useful mental model of how a stepper functions. Figure 21.1 illustrates our simple motor model. At the center is a bar magnet, free to rotate, surrounded by four electromagnets, spaced 90 degrees to each other. The electromagnets are wound over soft iron poles. In motor terminology, the bar magnet is the rotor and the surrounding electromagnets form the stator. In Fig. 21.1, the motor is unpowered and the rotor has automatically rotated to the position of minimum magnetic energy; that is, the permanent magnet rotor positions itself so that its flux has the shortest air path and the longest iron path. If you try to rotate the rotor by twisting the shaft, you will note resistance; you have to supply the energy required to break the magnetic attraction between the stator and the nearest pole. You should feel the same resistance, followed by the motor snapping into a new stable position,

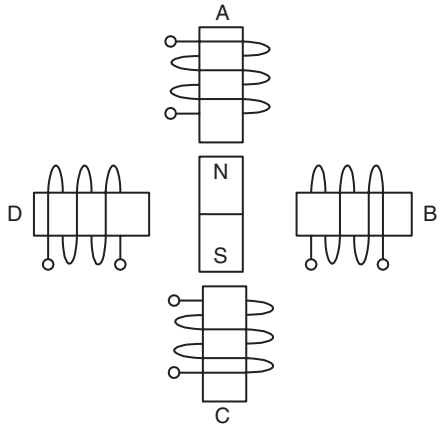


Figure 21.1: Simple Stepper Model: Unpowered

if you twist the shaft of a real stepper motor. (One uncommon type of stepper, the variable reluctance motor, doesn't exhibit this behavior, as we'll note later.) In an unpowered motor the torque required to break the rotor free from its rest position is called the detent torque.

Now, let's energize winding A, as shown in Fig. 21.2. We'll set the polarity of the current through A so that the inward facing pole is a magnetic south pole, which attracts the rotor's north pole. As long as winding A is energized, the rotor is held in place. The external torque necessary to override the magnetic attraction and move the rotor is known as the holding torque or static torque. For most motors the holding torque when operated at rated current is about 10 times the detent torque.

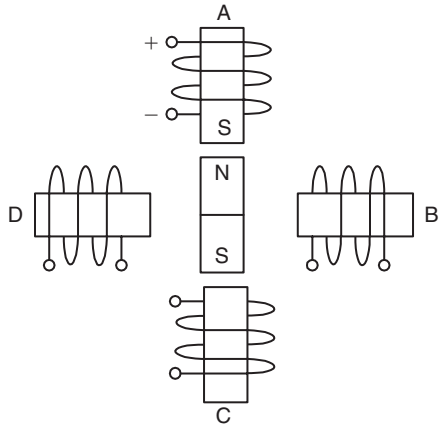


Figure 21.2: Winding a Energized

What happens if we energize the windings in sequence? Suppose we apply current to windings A, B and C, in that order, with the polarity so that the inside of each winding is a magnetic south pole. Figure 21.3 shows the result; the rotor rotates clockwise, as its north pole is sequentially attracted by the south poles temporarily created by energizing A, B and C. Of course, we may keep this up and energize D, then A, B, C, D… so long as we desire the rotor to continue stepping clockwise. Should we wish to stop the motor, we may either de-energize all windings or, if we need additional rest torque, we may keep one winding energized. If we wish to rotate the rotor counter-clockwise, we energize the windings in the reverse order: D, C, B, A.

As we'll see later, there are several variations on the order of energizing windings, including energizing more than one winding at once.

21.1.3 Unipolar and Bipolar

The windings in our stepper may be internally connected in several configurations. Two, however, two are of interest, the unipolar and bipolar connection. We'll start with the bipolar motor, sometimes called a two-phase stepper motor.

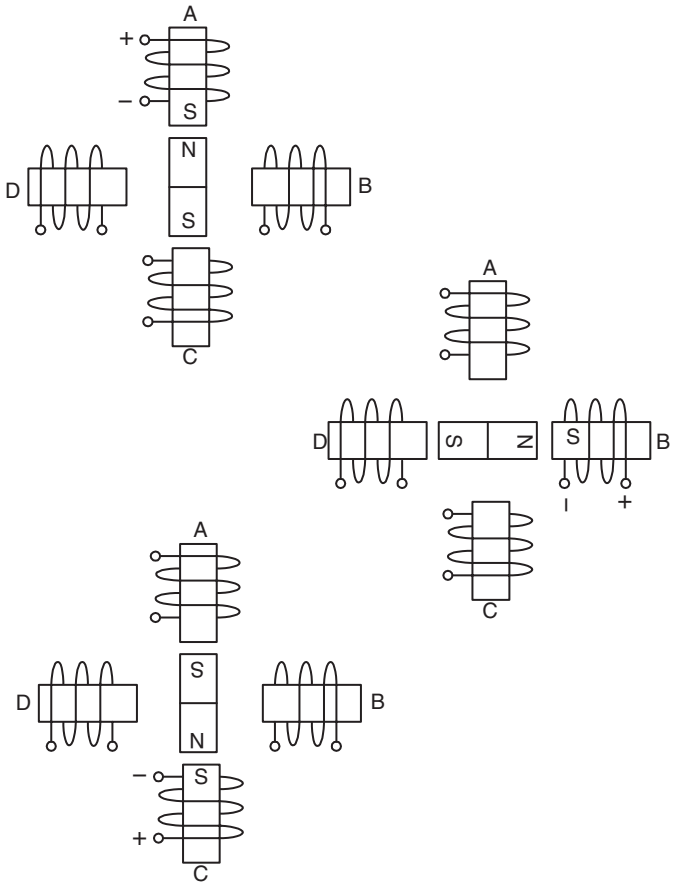


Figure 21.3: Sequential Current Flow in A, B and C Causes Rotation

Figure 21.4 shows our simple model motor connected in the bipolar configuration. The motor has four terminals accessible to the user, 1 through 4. The upper diagram shows our motor in the starting position, with current flowing through both windings A and C. Note that windings A and C are wound so as to produce opposite field polarity; when current flows in the direction of the arrows, winding A presents a south pole to the rotor while winding C presents a north pole. This polarity is represented by terminal 1 being positive with respect to terminal 3. Unlike our earlier examination, the rotor is thus held in place by two energized windings, not one. Suppose we then de-energize windings A and C, energize windings B and D to rotate the rotor 90 degrees clockwise, de-energize B and D and then re-energize windings A and C. We now desire the magnetic polarity to match that of the lower illustration in Fig. 21.4; the magnetic polarity of windings A and C are reversed from the upper illustration. We accomplish this magnetic polarity reversal by reversing the direction of current flow through windings A and C; we make terminal 3 positive with respect to terminal 1. Let's see how the polarity changes for one complete clockwise rotation cycle.

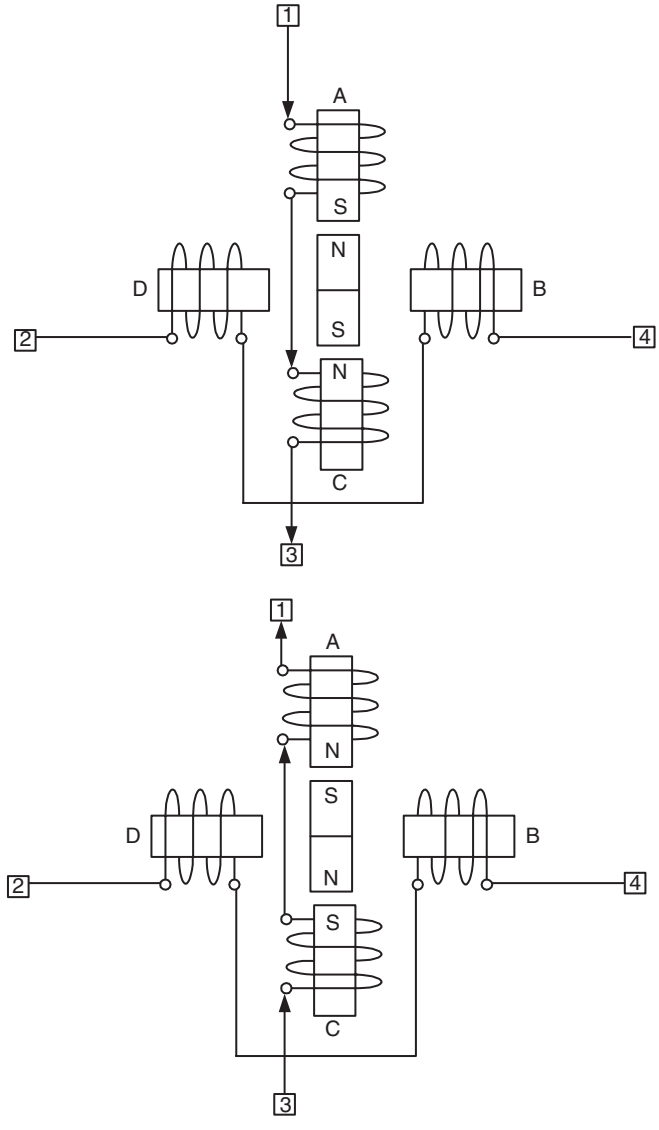


Figure 21.4: Bipolar Configuration of Simple Stepper

Step	Terminal 1	Terminal 3	Terminal 2	Terminal 4
1	+	−	None	None
2	None	None	+	−
3	−	+	None	None
4	None	None	-	+

The reason we term this connection "bipolar" is that the current polarity in the windings reverses—that is, it has two possible polarities, depending on the step. We'll consider how to accomplish reversing the winding current flow when examining drive circuits.

Figure 21.5 shows our simple stepper connected as a unipolar motor. (Unipolar motors are also known as four phase stepper motors.) The windings A-C and B-D remain in series, but the center taps, X and Y, respectively are also available, thus giving us six connections to the windings. (In some unipolar motors, both center taps are connected together and only five wires are brought out, as shown by the dashed connection line between terminals X and Y in Fig. 21.5). In the normal mode of operation, the center tap is always connected to the positive supply and we cause current to flow in the windings by connecting their free ends to the negative return, ground in most designs.

In the upper illustration in Fig. 21.5, winding A is energized by placing positive voltage on terminal X and grounding terminal 1. From this is the starting point we'll assume that winding A is then been de-energized, winding B energized to pull the rotor 90 degrees clockwise and then winding B is de-energized and now winding C is energized by connecting terminal 3 to ground, as shown in the lower illustration in Fig. 21.5. Current flow through the windings is always in the same direction; hence the name "unipolar" for this connection. Note that since windings AC and B-D are in series just as in our bipolar configuration, we may take this unipolar motor and connect it to a bipolar drive circuit (making no connections to the center taps X and Y) and it will work. (This is true for real unipolar motors, not just for our simple model).

We can summarize the benefits and drawbacks of unipolar and bipolar connections as:

Configuration	Advantages	Disadvantages
Unipolar	• Simplest drive circuit.	• Less efficient use of motor windings.
Bipolar	• Efficient use of motor windings. • Greater torque than for same size unipolar motor.	• Requires special drive circuitry; most commonly an H-bridge arrangement.

21.1.4 Types of Stepper Motors

Figure 21.6 shows four typical stepper motors. The smaller two motors at the left of the figure are known as tin can or can stack or permanent magnet motors, while the two larger motors are hybrid constructed.

Tin can motors are inexpensive, constructed from a pressed or stamped case and with a smooth permanent magnet rotor magnetized with alternating north and south poles. Usually tin can motors have relatively coarse step sizes, with 24 and 48 steps/rev (15 and 7.5 degrees/step) being typical values. Tin can motors use sleeve bearings and are typically found in

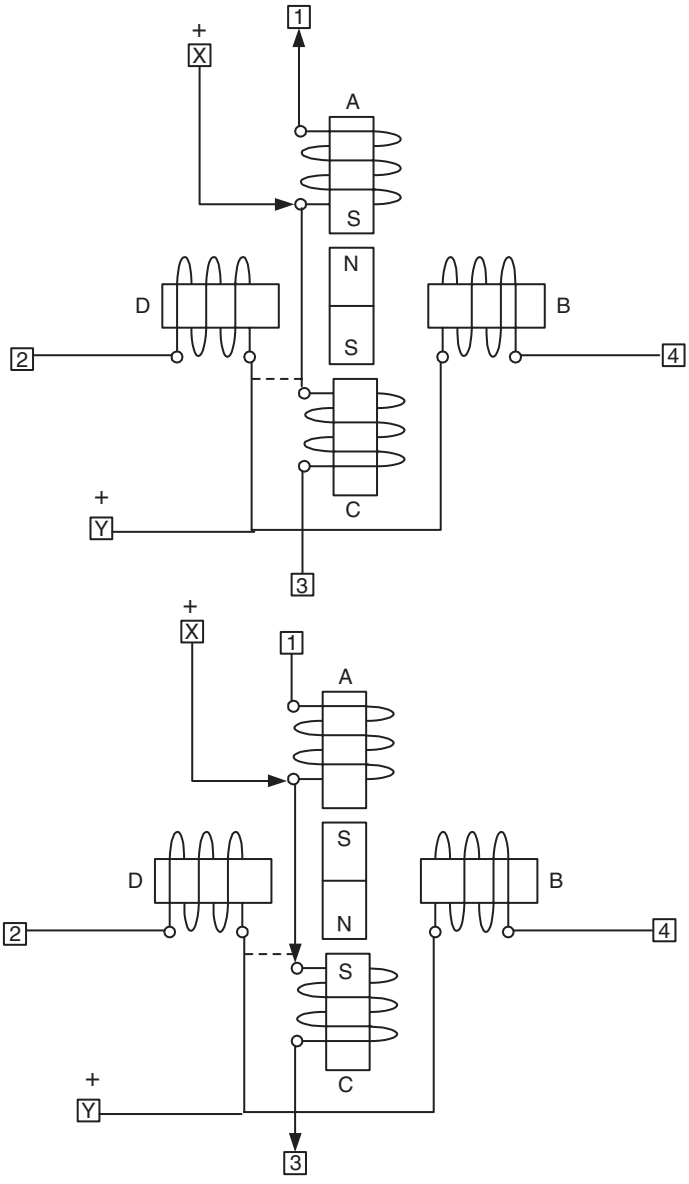


Figure 21.5: Unipolar Configuration of Simple Stepper

inexpensive electronic products, such as ink jet printers and fax machines. Most manufacturers use the case diameter (in millimeters) and number of steps as part of the model number. For example, the smallest motor in Fig. 21.6 is a Nippon Pulse Motor model PF35-48L4 stepper. The case diameter is 35 mm (about 1-3/8″) and it has 48 steps per revolution. The L4 suffix indicates the coil voltage (nonstandard) and rotor magnet type (Neodymium). Other

manufacturers use different identifiers, of course, but case diameter and number of steps/revolution are commonly incorporated into the model identification.

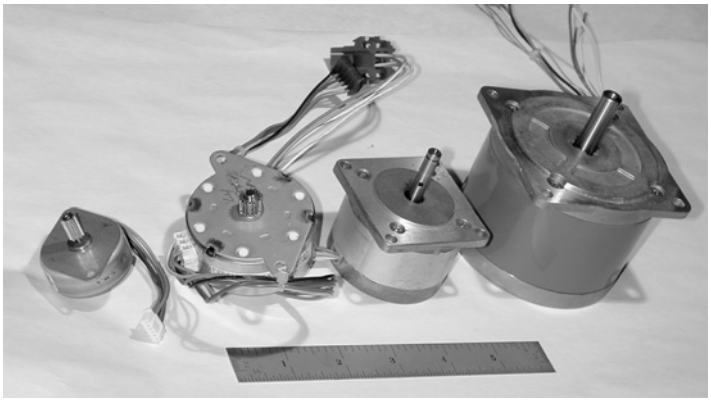


Figure 21.6: Typical Stepper Motors

The two larger motors in Fig. 21.6 are of hybrid construction. Figures 21.7 through 21.9 show a partially disassembled hybrid motor. (I don't recommend disassembling a stepper motor unless absolutely essential, as some high performance rotors will be partially demagnetized if the rotor is removed from the motor case.) Figure 21.8 shows the toothed permanent magnet rotor. The rotor is constructed of two toothed segments, with one segment offset by one-half tooth width from the other, thereby effectively halving the step size.

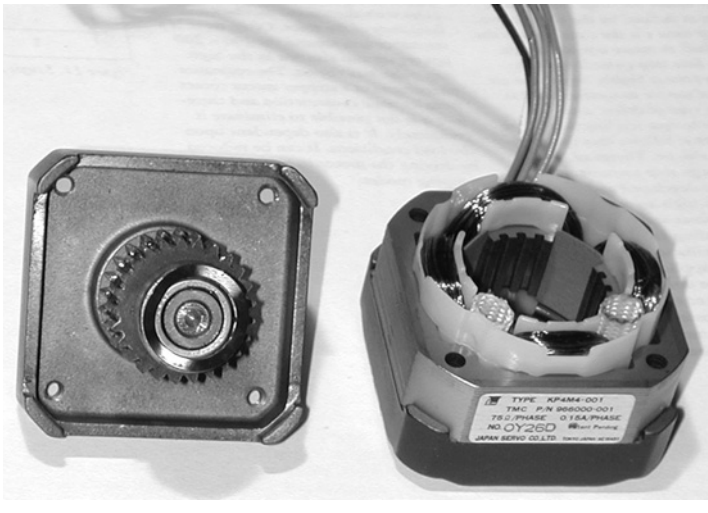


Figure 21.7: Hybrid Motor Disassembly

Figure 21.8: Hybrid Motor Toothed Rotor

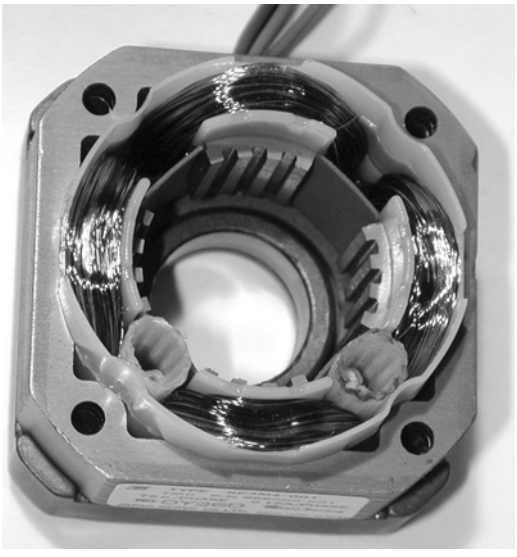


Figure 21.9: Hybrid Motor Interrupted Toothed Stator

(The objects at the end of the rotor shaft are a ball bearing and a Belleville washer.) Figure 21.9 shows the stator, which has several noteworthy features. First, the four windings are clearly visible, just like our mental motor model. However, the poles are segmented, with each pole having four projecting pieces. (In motor terminology, these are *salient* poles). It

isn't necessary—or even desirable—for the poles to be continuous around the inner periphery of the motor; the rotor is continuous, which is sufficient.

Hybrid motors are often manufactured in industry-standard case sizes, as defined by the National Electrical Manufacturers Association (NEMA). A motor manufactured by Company X in NEMA style 34 is mechanically interchangeable with another NEMA 34 case size motor manufactured by Company Y. The largest motor in Fig. 21.6 is a NEMA 34 motor, while the one next to it is a NEMA 23 motor. Of course, the electrical and performance specifications of two motors with identical NEMA case sizes are not necessarily (or even usually) the same.

Hybrid motors are more expensive than tin can motors and feature higher quality construction, such as ball bearings instead of sleeve bearings, and cast or machined cases instead of pressed case. Additionally, the toothed construction permits much finer steps, with 180 and 200 step/rev being common values.

A third type of motor is the variable reluctance, resembling the hybrid in construction, but with a nonpermanent magnet toothed rotor. Variable reluctance motors are relatively uncommon and will not be further discussed.

There are other much less common stepper motor types, such as three-phase unipolar. These require specialized driving circuits and are beyond the scope of this text.

21.1.5 Identifying Stepper Motors

To identify a stepper motor that has no nameplate or for which a data sheet is not available, we may use the following steps.

1. Turn the motor shaft by hand. You should feel the detents; if you feel no detents the motor is not a stepper of the type considered in this chapter. Turn the motor shaft one complete revolution and count the number of detents you encounter. This gives the steps/rev value for the motor. Common step/rev values are 24, 48, 72, 100, 180, 200, 400 and 800, although the later two values are relatively unusual. Your count should be close to a multiple of 10 or 12. If you count 197, it almost certainly means you missed a count here or there and have a 200 step/rev motor.
2. How many wires or a connection does the motor have?
   a. Four—you likely have a bipolar motor.
   b. Five or six—you likely have a unipolar motor.
3. With an ohmmeter, identify the wire colors or terminal numbers corresponding to your windings and label them as in Fig. 21.10 (bipolar) or Fig. 21.11 (unipolar). Make a note of your resistance measurements. The resistance of windings A and B should measure within 5% or so of each other. Likewise, in a unipolar motor, the resistance from the common center tap to each winding end should be approximately equal

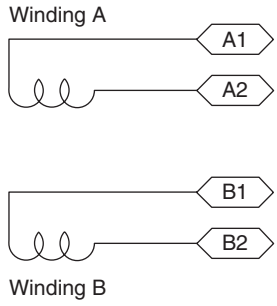


Figure 21.10: Winding Labels for Bipolar Stepper Motor

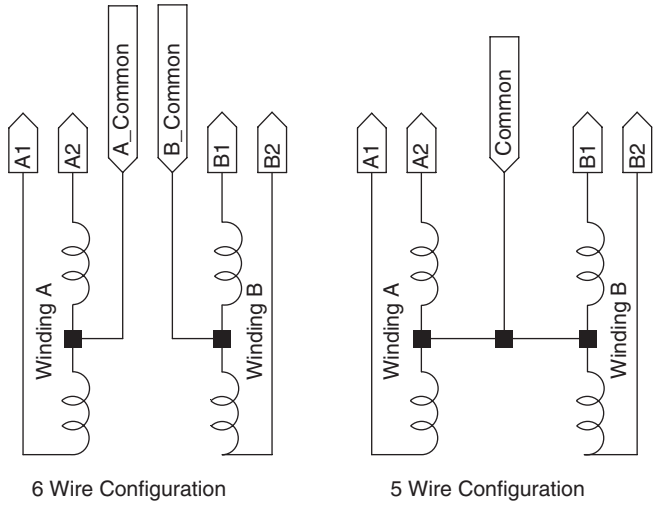


Figure 21.11: Winding Labels for Unipolar Stepper Motor

and the resistance across each complete winding (A1 to A2 and B1 to B2) should be approximately equal and twice the value from the common to each end.

4. If you have access to an inductance bridge, measure the inductance of the windings. If you don't have a bridge, you may safely skip this step.

5. Now we will attempt to "guestimate" the motor's voltage and current ratings. This step is necessary only if your motor doesn't have a nameplate or part number providing this information or if you can't find a data sheet for the motor. There is no magic way to accomplish reverse engineering the motor's rating with complete accuracy, but we can come close enough for experimentation purposes. Measure the physical size of the unknown motor and determine the construction type. Is it a tin can or a hybrid motor? Next, search the manufacturer's catalogs, either paper copies or on the internet, until you find a motor with the same physical size, construction type connection type

(bipolar or unipolar) and number of steps. See if you can find a motor with similar coil resistance and (if you have measured it, coil inductance). If you can't find a match, then calculate the power dissipation (in watts) for several motors of the same case size as your motor using the formula $P = I^2R$, where P is the power in watts, I is the motor's current rating in amperes and R is the motor's winding resistance in ohms, with both I and R from the catalog values. Your calculated P will likely differ among the matching motors, so calculate an average value. Then, using the average power dissipation for the physically similar motors and your measured resistance value, calculate the resulting I for your motor, using the formula $I = \sqrt{\frac{P}{R}}$.

6. Now that we have determined the motor's rated current, *I*, and the measured *R*, calculate the motor's nominal operating voltage *V* from Ohm's law, $V = IR$.

7. Note any other important parameters from the closest matching data sheet, such as the maximum speed in steps/second or the maximum torque.

21.1.6 Reading a Stepper Specification Sheet

Let's look at a typical specification sheet for an inexpensive tin can motor, a Nippon Pulse Motor model PF35-48. I've reproduced the data sheet parameters below.

Parameters	Units	PF35-48			
Drive Mode		Unipolar		Bipolar	
Excitation Mode		Full-step (2-2 ex)			
Step Angle	°	7.5			
Step Angle Tolerance	%	±5			
Steps per Revolution		48			
Voltage	V	12	5	12	5
Winding Resistance	ohm/Ø	90	16	100	17
Winding Inductance	mH/Ø	48	8.9	124	19
Holding Torque	mN · m	20	20	25	25
Rotor Inertia	$kg \cdot m^2$	4.5×10^{-7}			
Starting Pulse Rate, Max	pps	500			
Slewing Pulse Rate, Max	pps	530			
Ambient Temp. Range, Operating	°C	−10 ~ +50			
Temperature rise	K	55			
Mass	g	80			

What does each line mean?

Drive mode—The PF35-48 is available in either a bipolar or a unipolar configuration.

Excitation mode—As we will see later, a stepper may be operated in several modes, and certain parameters, such as torque and step angle, are different for different modes. The data sheet's statement "full-step (2-2 ex)" means that the performance data is based upon full step operation, with both coils energized. If this sentence doesn't mean much to you right now, put a star in the margin and come back to it after reading the rest of this chapter.

Step angle—The angle in degrees through which the shaft rotates when it advances one step while in full step mode. The value 7.5º corresponds to 48 steps/rev.

Step angle tolerance—The tolerance, as applied to the step angle, that is, the angle the motor shaft advances in one full step is 7.5º ± 5%. It's important to remember this tolerance applies on a step-by-step basis and is not cumulative. After 48 steps, the motor will return to its original starting point with an accuracy of ±5% × 7.5º or ±0.375º. After 48000 steps (1000 complete revolutions), the motor will be at its original starting angle ±0.375 degrees. The noncumulative error performance of a stepper is the key to its ability to perform precision operations. If the error were cumulative, after being commanded to perform 48000 steps, or 1,000 revolutions, the shaft angle would be unknown within ±50 revolutions, quite a difference from the actual ±0.375 degrees!

Steps per revolution—The number of full steps required to return the motor shaft to its starting angle. Since there are 360º in one revolution, the step angle and steps per revolution are related by the formula: $\theta = \frac{360}{N}$ where θ is the step angle and N is the number of steps per revolution.

Voltage and winding resistance—We'll consider these two parameters at the same time. You may recall from high school physics that the magnetic field of an electromagnet is proportional to the current in the windings multiplied by the number of turns (ampere-turns) and that the attractive force between two magnets is proportional to their magnetic fields. Hence, for a fixed number of turns, the shaft torque in the stepper motor is proportional to the current through the windings. If we double the current, we double the torque. And, we know from elementary circuit theory that resistive power dissipation is proportional to the square of the current; $P = I^2R$. If we double the current, the power dissipated in the motor goes up fourfold.

The motor designer must balance these two effects against each other; to make the motor more powerful for its size, the designer wishes to maximize the current. However, more current causes more internal heating and if the motor temperature exceeds a certain level the winding insulation may break down and the motor will fail. Alternatively, to increase the stator's

magnetic field, the designer may decide to use smaller diameter wire, which allows more turns in a given space (increasing ampere turns), but the smaller diameter wire has greater resistance which means we must apply higher voltage to the stator coil to obtain the desired current. The trend is to lower voltage power supplies, and the motor manufacturers try to meet their customers' needs with lower voltage motor designs.

The motor's rated voltage and resistance allow us to calculate the nominal winding current using Ohm's law: I = V/R. As we will see when we look at driver circuits, usually we drive the motor through a quasi-constant current arrangement.

In this case, the PF35-48 motor has two winding options; a 12 V winding with 100 ohms resistance and a 5 V winding with 17 ohms resistance. We can calculate the corresponding currents as 120 mA and 294 mA, respectively. Since the torque specifications are identical for both winding options, we may safely assume that the 12 V winding has close to 2.45 more turns than the 5 V version, thereby keeping the ampere turns— and torque—identical. A quick check confirms that the PF35-48 is designed for identical power dissipation for both 12 V and 5 V windings. The 12 V coil dissipates 1.44 watts at the rated voltage, while the 5 V coil dissipates 1.47 watts.

The PF35-48 motor I used in this chapter has an "L4" suffix, meaning it is a "special" voltage rating. Since we know the motor dissipation is 1.4 watts, and since I measured the coil resistance as 20 ohms, we may determine the motor's rated voltage is:

$$P = \frac{E^2}{R}; E = \sqrt{PR}$$
$$E = \sqrt{1.4 \times 20} = 5.3\text{V}$$

Likewise, we calculated the rated current:

$$P = EI; I = \frac{P}{E}$$
$$I = \frac{1.4}{5.3} = 265\,\text{mA}$$

where:

P is the power dissipation in watts;
I is the current in amperes;
V is the voltage in volts;
R is the resistance in ohms.

Finally, there is a "nondissipative" element of input power to the motor; the power that goes to perform mechanical work at the output shaft. We've neglected this, as for many stepper motor applications the mechanical work output is small compared with the resistive winding loss.

Winding inductance—As a consequence of producing the desired magnetic field, the motor windings are inductors. Their inductance limits the rate of rise of current through the winding when a winding is energized. From a performance prospective, we would like the winding current to immediately assume its final value when the voltage step applied. We'll look at this in more detail when we consider drive circuits, but let's examine the difference between the 5 V and 12 V versions of the PF35-48 motor.

In introductory electrical circuits class, we learn that if a voltage is applied across a series circuit of an inductor and a resistor, such as the windings of a stepper motor, the current starts at zero and increases according to an exponential function, with a limiting value determined by the series resistance. The current versus time relationship is:

$$i = I_f \left(1 - e^{\frac{Rt}{L}}\right)$$

where:
I_f is the steady-state current, in this case the applied voltage V divided by the winding's resistance R, or

$$I_f = \frac{V}{R}$$

R is the winding's resistance (plus any other resistance in the circuit) in ohms;
L is the winding's inductance in henries;
t is the time in seconds after the voltage is applied to the inductor;
e is the base for natural logarithms, 2.71828182…

We often can use a simpler calculation—the inductive time constant τ for circuit analysis:

$$\tau = \frac{L}{R}$$

where:
τ is the time constant, in seconds.

After the time τ , the current will have reached approximately 63% of its steady state value; after time 2τ it will have reached 86.4%, etc.

Figure 21.12 shows the result of a SPICE circuit simulation of the current through the 5 V and 12 V winding of a bipolar PF35-48 motor. It follows the negative exponential form of the equation shown above. Although the magnitudes of the two currents are different, both reach equal percentages of the maximum at almost the same time. This

can be seen either by examining Fig. 21.12, or by comparing the time constants of the two windings:

$$\tau_{5V} = \frac{19 \times 10^{-3}}{17} = 1.12\,\text{ms}$$

$$\tau_{12V} = \frac{124 \times 10^{-3}}{100} = 1.24\,\text{ms}$$

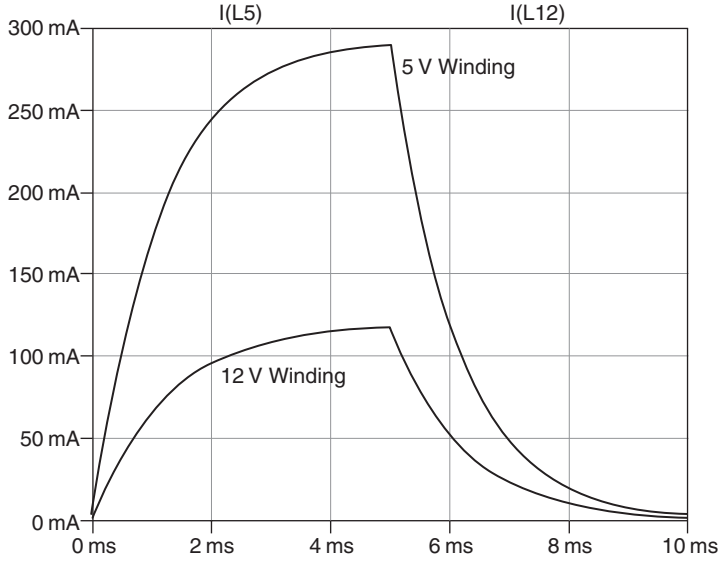


Figure 21.12: Simulated Current Through 5 V and 12 V Stepper Motor Windings, PF35-48 Motor

Since the two time constants are almost identical, the current through the two coils will follow an almost identical time relationship, expressed in terms of the percentage of final current versus time. Since the motor's torque is proportional to ampere turns, and since the 12 V motor has proportionally more turns than the 5 V motor, we conclude that regardless of whether the 5 V or 12 V windings are used, the PF35-48's torque versus time performance will be almost identical. That the 5 V and 12 V windings have almost identical time constants is, of course, not an accident, but rather a product of the designer's intention to produce two motors that have essentially identical performance, regardless of whether the 5 V or 12 V version is used.

Holding torque—is the torque that must be applied externally to rotate the shaft from its position if power is applied to the windings. It's stated in the metric unit milli-Newton-meters (mN-m), and may be converted to the corresponding Imperial unit ounce-inches by the relationship: 1 oz-in = 7.06 mN-m. If you work in Imperial units, don't forget the difference between force and weight; torque is expressed in force units, not weight units.

Rotor inertia—is the moment of rotational inertia of the motor's rotor. For our purpose, we'll just note that this value is a measure of the effort required to get the rotor moving and to stop it, once it is moving. If you know how to use the rotor inertia value in your dynamic performance calculations, you don't need me to summarize it here and if you don't know how to make those performance calculations, then a short summary won't do you any good.

Starting pulse rate, max and slewing pulse rate max—Both values are measures of how fast the motor may be stepped—from either a standing start, or once rotating. Let's see how fast we may run the motor, in terms of revolutions per minute, a term perhaps more common in everyday usage. We'll assume the motor starts from rest, so the maximum rate at which we may pulse it is 500 pps.

In one second, therefore, the number of revolutions is equal to the number of steps divided by the number of steps/revolution:

$$R_{\text{sec}} = \frac{Steps/sec}{Steps/rev} = \frac{500}{48} = 10.4\ rev/sec$$

To convert to revolutions per minute, multiply by 60:

$$RPM = \frac{60P}{N} = \frac{60 \times 500}{48} = 625$$

where

RPM is the speed in rev/min;
P is the number of pulses/sec;
N is the number of steps per revolution for the mode being used.

By small motor standards, this is a modest speed indeed. But the advantage of a stepper is not its high speed, but rather its precision and its ability to provide torque at slow speeds.

If you have to wring the last possible bit of speed from the motor, you can ramp the speed up from a standing start.

Ambient temperature range—operating and temperature rise—The motor is rated to operate in an ambient temperature range of –10ºC (+14ºF) to +50ºC (122ºF). Its temperature rise is 55ºK. A Kelvin degree is numerically equal to a Celsius degree, with the difference being the zero point—Celsius's zero is approximately the freezing point of water while 0ºK represents absolute zero—approximately −273ºC or −459ºF. Hence, if operated at the maximum permitted ambient temperature, the motor's temperature will not exceed 105ºC (222ºF).

Mass—The motor's weight.

21.1.7 Operation Modes

We've alluded to various drive methods earlier, so let's see what's involved in controlling a stepper.

We'll start with a unipolar motor, as it's the easiest to understand. We'll assume it's a 6-wire motor and that it's connected to a power supply set to the motor's rated voltage. Figure 21.13 shows four SPST switches that enable us to connect any of the motor's four windings to ground; in a real circuit, we would use, for example, one of the low side driver circuits we saw in Chapter 19. But to understand the principles, we'll just think of the low side driver circuit we finally settle upon a nothing more than a way to either isolate the motor winding or connect it to ground.

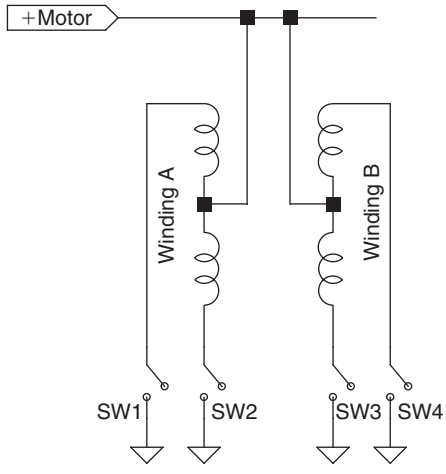


Figure 21.13: Unipolar Motor Connection

We'll return to our simplified motor to examine how we step the switches. We'll see three patterns produce useful results; wave, full step and half-step.

Wave—Figure 21.14 shows how we may advance the rotor by sequentially energizing coils A, B, C and D (note that I've shown the switches in winding order, not numerical switch order):

Step	S1 (Winding A)	S3 (Winding B)	S2 (Winding C)	S4 (Winding D)
1	Closed	Open	Open	Open
2	Open	Closed	Open	Open
3	Open	Open	Closed	Open
4	Open	Open	Open	Closed

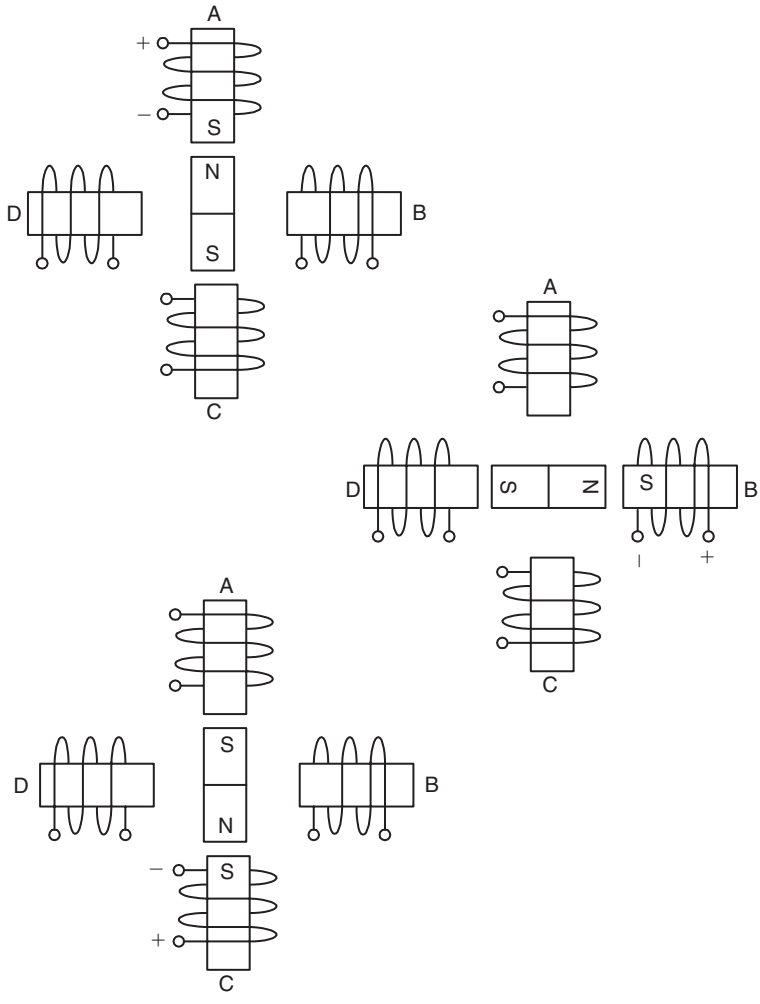


Figure 21.14: Wave Stepping

This mode is called "wave" because you can see a sequence of switch closures stepping down and across the table, causing a rotating magnetic wave within the motor. The stator poles are aligned with the rotor poles for each step.

Full step—In wave mode, only one winding is energized at any time. If we could simultaneously energize two windings, we could double the magnetic strength within the motor, thereby doubling the torque. Can we do this? Figure 21.15 shows how we might accomplish this. By simultaneously energizing windings A and B, for example, we create the equivalent

of (almost) a double-strength coil, halfway between A and B. Thus, our stepping pattern is AB, BC, CD, DA:

Step	S1 (Winding A)	S3 (Winding B)	S2 (Winding C)	S4 (Winding D)
1	Closed	Closed	Open	Open
2	Open	Closed	Closed	Open
3	Open	Open	Closed	Closed
4	Closed	Open	Open	Closed

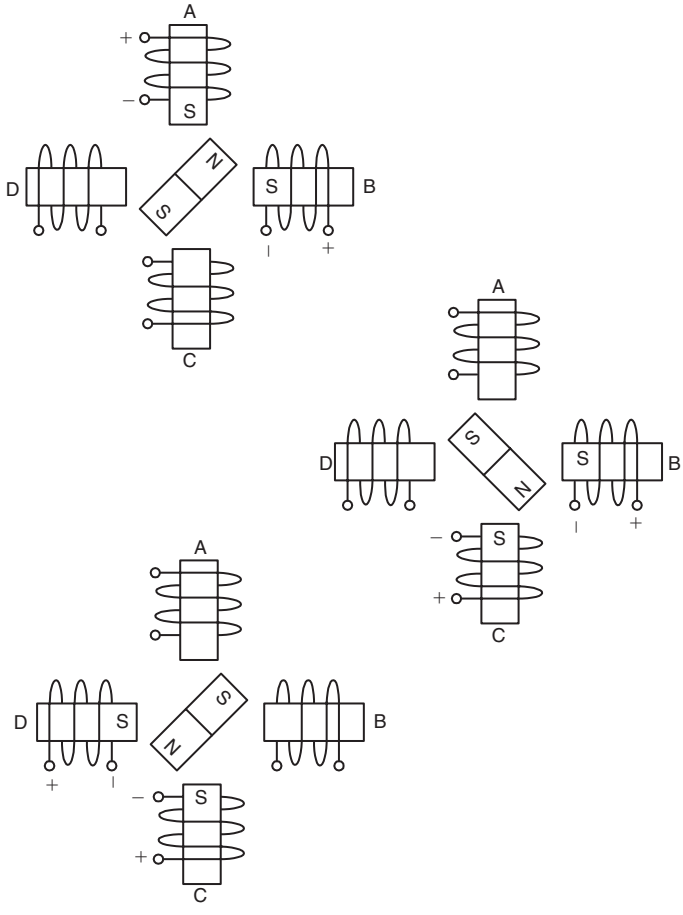


Figure 21.15: Full-Step Mode

In fact, energizing two windings at 90 degrees to each other doesn't quite double the torque compared with a single winding, but it's close. The rotor poles align half-way between adjacent energized stator poles.

Half-step—We've seen that wave mode makes the rotor step in alignment with the poles, and full wave mode causes the rotor to move to half-way between the poles. Suppose we combined and interleafed wave and full wave mode. We could then cause the rotor to move in increments of one-half the normal step size. Figure 21.16 shows how this works. Our excitation mode is A, AB, B, BC, and so on:

Step	S1 (Winding A)	S3 (Winding B)	S2 (Winding C)	S4 (Winding D)
1	Closed	Open	Open	Open
2	Closed	Closed	Open	Open
3	Open	Closed	Open	Open
4	Open	Closed	Closed	Open
5	Open	Open	Closed	Open
6	Open	Open	Closed	Closed
7	Open	Open	Open	Closed
8	Closed	Open	Open	Closed

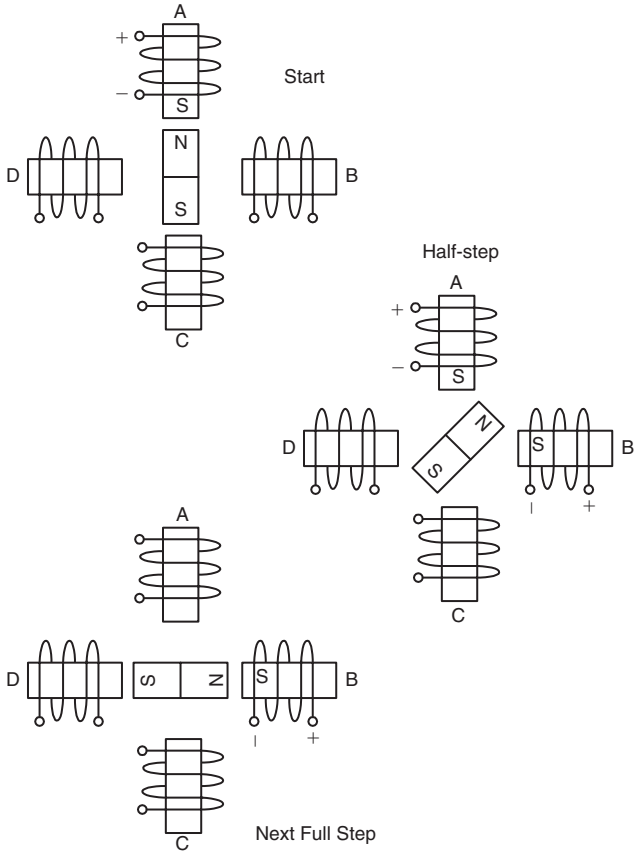


Figure 21.16: Half-Step Mode

If you carefully look at the half step excitation pattern, you see that the wave and full step patterns are interleafed. You should also see that half stepping produces nonuniform torque—for half the steps only one winding is energized while for the remaining half of the steps two windings are energized. This effectively limits half stepping operation to torque requirements that may be met by only one winding.

Although we've looked at the patterns for a unipolar motor, in fact the identical patterns work for a bipolar motor. Let's look at how we might switch a bipolar motor. Figure 21.17 shows a conceptual view of controlling a bipolar motor. For each winding pair we have three options: current flow in one direction, current flow in the opposite direction and unenergized. For convenient reference, we'll identify the current flow direction by a plus sign. A+/C for example, means the free end of winding A is positive and the free end of winding C is negative, Fig. 21.18 shows wave drive operations for the first three steps. The winding excitation pattern for one complete revolution is: A+/C, B+/D, C+/A, D+/B.

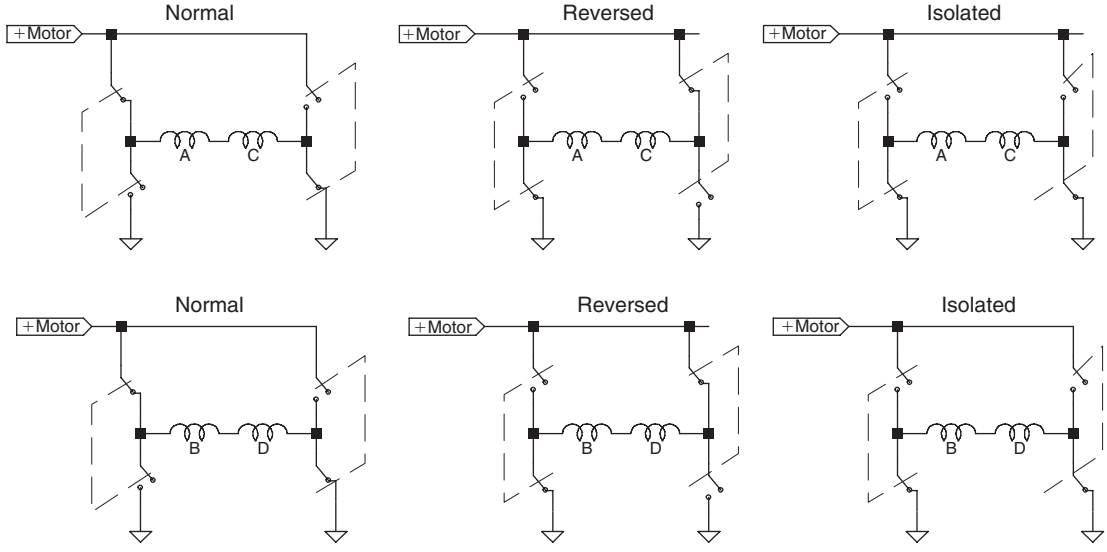


Figure 21.17: Bipolar Motor Connection

We'll use an electronic version of the switches shown in Fig. 21.17, in particular a SN754410 H- bridge. The H-bridge, as we'll see later in this chapter, has four inputs, corresponding to each end of the two windings. When the corresponding input is set at logic high (1), that winding end is connected to the positive motor supply; when an input is logic low, the associated winding end is connected to ground. Hence, A+/C corresponds to a logic pattern

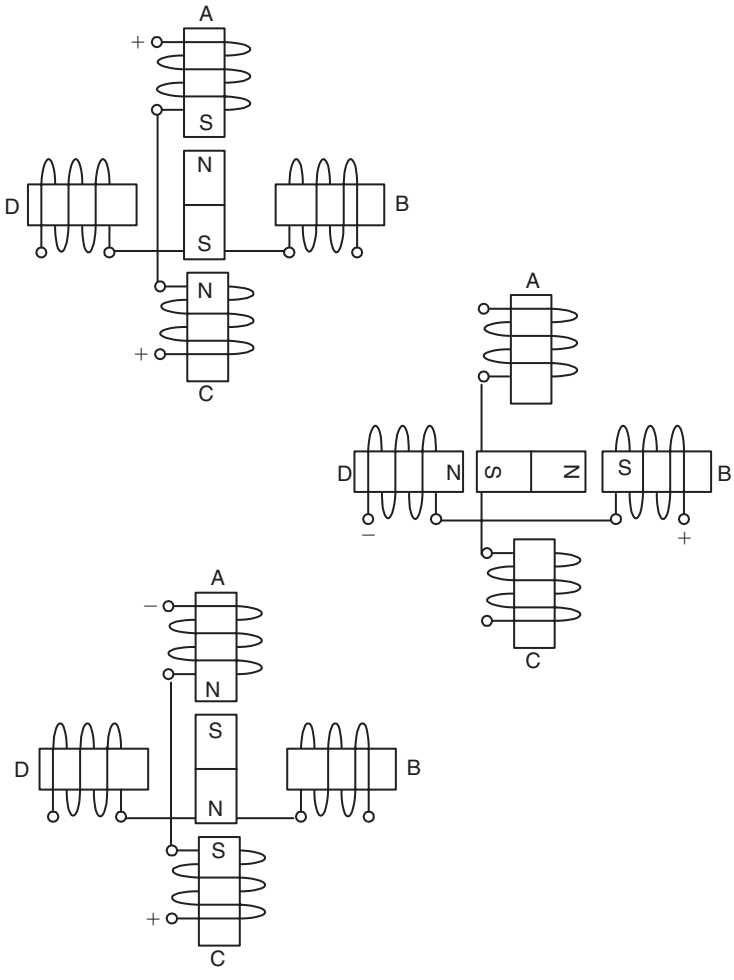


Figure 21.18: Bipolar Connection–Wave Mode

of 10, while A/C+ is 01. If AC is de-energized, the logic pattern is 00. Using this terminology, the wave drive pattern for our bipolar stepper is:

Step	S1 (Winding A)	S3 (Winding B)	S2 (Winding C)	S4 (Winding D)
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

Compare this with the drive pattern for wave drive of a unipolar motor. Substitute 1 for a closed switch and 0 for an open switch and you will see that the two patterns are identical.

Since the full step and half-step patterns are also identical with their unipolar counterparts, we won't repeat the drawings and patterns charts for bipolar motors.

Comparing Figs 21.14 and 21.18 should reveal an advantage of bipolar motors; in wave drive, a unipolar motor has only one winding energized, while the bipolar motor has two windings energized, thus approximately doubling the available torque. If we make the same comparison for half-step operation, we find that the unipolar motor has either one or two windings energized while the bipolar motor has two or four windings energized, again about doubling the available torque. With respect to full step drive, the torque difference is less pronounced, as both motor have two windings energized for each step. However, due to rotor and stator alignment issues, a bipolar motor typically provides 20% to 30% more torque in full step mode than an otherwise identical unipolar counterpart. Of course, the motor's power and temperature ratings must also be considered and may be a limiting factor.

21.2 Programs

Let's start with a simple program demonstrating MBasic's built-in support for stepper motors, via the `SpMotor` function. Figure 21.19 shows how to connect the stepper to the 16F877A. We use no special functions, so almost any PIC may be substituted for the '877A. The motor you use must be a unipolar stepper and it must draw less than 500 mA. I used the PF35-48 unipolar stepper we earlier analyzed. (The PF35-48 I used has 5 V windings.) Note carefully the winding connections in Fig. 21.19. Each pair of windings connects to alternating outputs, not to adjacent outputs. Set the motor supply voltage to the stepper's rated voltage. Don't use the built-in +5V supply in the 2840 Development Board to run a stepper—it's not safely up to the task.

As we learned in Chapter 19, the PIC's output circuitry is not suitable to directly connect to a stepper. Hence we use a ULN2003A Darlington transistor driver. Do not connect a stepper directly to the PIC; it may well destroy the PIC. The ULN2003A contains seven Darlington transistors, with built-in base current limiting resistors. A logical 1 on the ULN2003A's input saturates the output, thereby providing the functional equivalent of closing one of the four mechanical switches in Fig. 21.13.

As we learned in Chapter 19, when we switch an inductive load, the collapsing magnetic field causes a voltage spike that may damage the switching transistor. In addition, the coupled windings of a stepper motor act like transformer windings and can have induced in them a substantial voltage spike when other windings are turned on or off. To control the spike, we use the built-in diodes of the ULN2003A and an external zener diode. The circuit will work without the zener, but with slower current decay. Slower current decay may not be a problem if your stepper runs at slow step speeds, but to maximize speed add the zener. Figure 21.20 is a SPICE simulation analysis of the current release improvement a zener diode makes over using the ULN2003A's internal clamping diode. With a zener, the motor current decays to zero in about

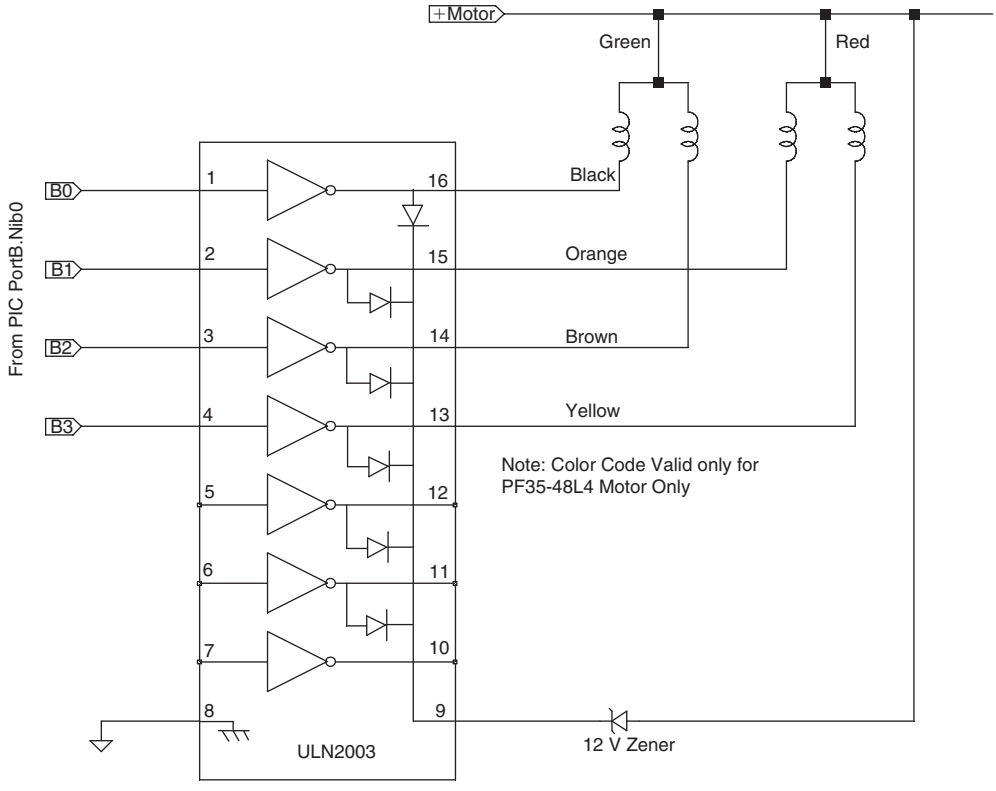


Figure 21.19: Unipolar Stepper Motor Connection for Program 21.1

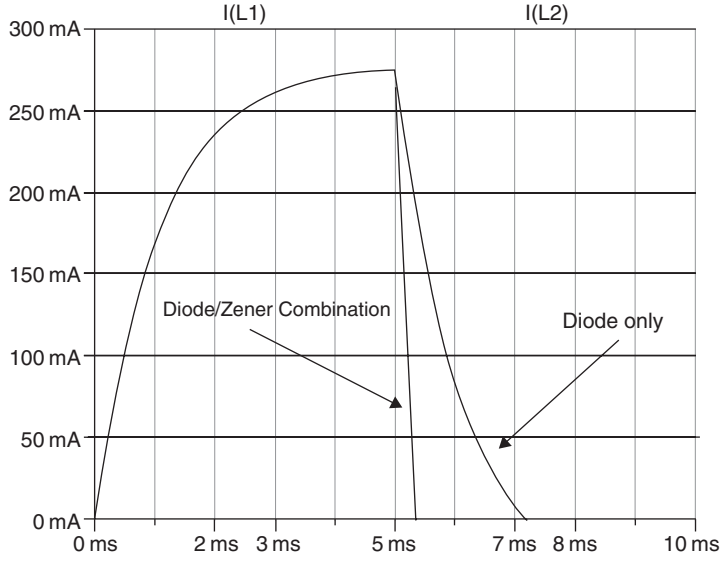


Figure 21.20: A Zener Diode Improves Current Release Time

400 μs, versus 2.3 ms for a simple diode clamp. Why do we want the current to quickly decay after a winding is de-energized? The decay current acts to hold the rotor in place and prevents it from being attracted to the newly energized winding. This may not be so important if the motor is stepped slowly, but as we increase the step speed, it becomes increasingly important to ensure the current decays quickly upon release. As we'll see later in this chapter, for fast stepping rates we must also improve the rate of current increase at turn-on.

```
;Program 21-01
;Demo SpMotor function with ULN2003A
;Bipolar motor

;Constants
;----------
MPin      Con    B0       ;starting pin for nibble
MDelay    Con    10000    ;usec/step
MStep     Con    1000     ;total steps

Main
      ;Rotate in one direction
      SpMotor MPin, MDelay, MStep
      Pause 1000
      ;Now reverse direction
      SpMotor MPin, MDelay, -MStep
      Pause 1000
GoTo Main
```

Program 21.1

Program 21.1 exercises MBasic's built-in stepper motor function, `SPMotor`, invoked with three arguments: `SPMotor Pin, Delay, Step.`

Pin—is a constant or variable that defines the first pin of four consecutive output pins used by `SPMotor`.

Delay—is a constant or variable defining the time between steps. Delay is in microseconds with a maximum 32-bit integer value.

Step—is a variable or constant and defines how many steps are to be taken, and the direction. Positive numbers step in one direction and negative numbers step in the reverse direction.

```
Mpin       Con     B0        ;starting pin for nibble
Mdelay     Con     10000     ;microseconds/step
Mstep      Con     1000      ;total steps
```

We start by defining all three arguments to `SPMotor`; for our test, we'll use pins B0…B3, step at 10 ms/step and take 1000 steps. (10 ms is 10000 μs.)

```
Main
      ;Rotate in one direction
      SpMotor MPin, MDelay, MStep
      Pause 1000
      ;Now reverse direction
      SpMotor MPin, MDelay, -MStep
      Pause 1000
GoTo Main
```

Our program steps the motor 1000 steps in one direction, pauses for one second and steps 1000 steps in the reverse direction. After another one-second pause, the cycle repeats endlessly.

Now is as good a time as any to discuss rotation direction. Although MBasic's User's Guide refers to "clockwise" and "counter-clockwise" rotation as being associated with positive and negative values of the step argument, in fact, the direction of rotation is also governed by the order in which the windings are connected to the driver. If the motor rotates in a direction opposite to the way you wish it to rotate, reverse any winding pair. For example, referring to Fig. 21.19, to reverse the motor's rotational direction, interchange the motor leads connected to pins 16 and 14 of the ULN2003A. Or, interchange the motor leads connected to pins 15 and 13. However, if you interchange both pairs of leads, you will restore the original rotational direction! Accordingly, should your motor rotate counterclockwise even though `SPMotor` has a positive value step argument, reverse any pair of winding leads to synchronize your motor's windings with `SPMotor`'s output pulse sequence.

Let's look at the waveforms associated with Program 21.1 and the circuit of Fig. 21.19. Figure 21.21 shows the voltage at pin 16 of the ULN2003A (Chan. 1) and the associated current (Chan. 2) through the motor winding. Since Channel 1 of the oscilloscope is connected to the collector of the drive transistor, current flow is associated with taking the collector low. Let's see what we may learn from Fig. 21.21.

First, we note that when saturated, the ULN2003A's collector does not drop much below 1 V, with the measured value being 960 mV. We learned the reason that a Darlington transistor has a relatively high V_{SAT} in Chapter 19, so this value is expected. However, since we are applying only 5 V to the motor windings and ULN2003A, the motor has only 4 V applied across its windings, which reduces the current and hence its available torque.

The current through the motor winding (Channel 2) shows a typical inductive current rise, but with two interesting aspects. Note the kink; current goes up rapidly, then reverses momentarily before resuming the rise. A similar effect is seen at the release. (Figs 21.21 and 22 were taken without the zener diode, so they show a relatively slow current release.) Also, the slope of the

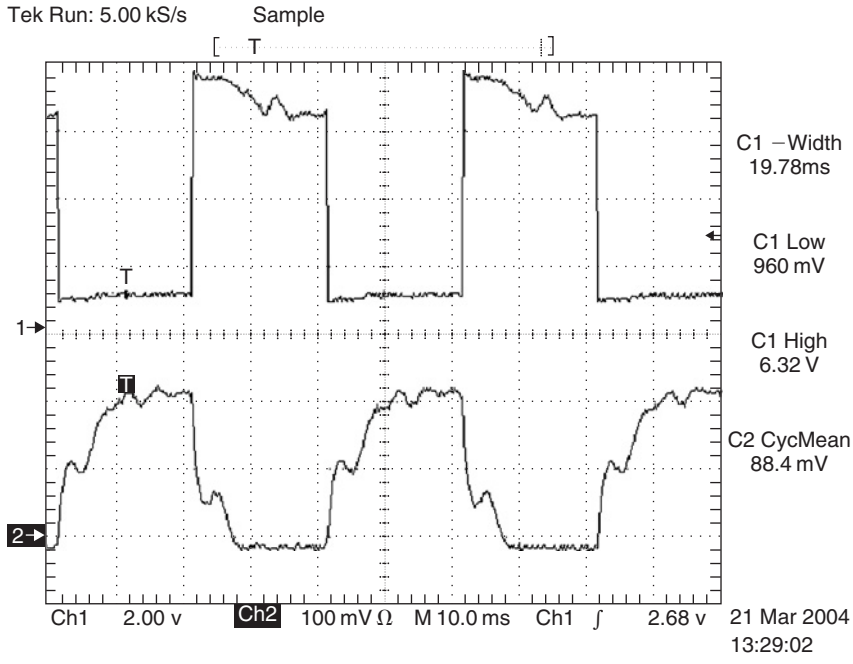


Figure 21.21: Waveforms of Program 21.1; Ch1: Pin 16 ULN2003A; Ch2: Current Through Motor Winding Connected to Pin 16

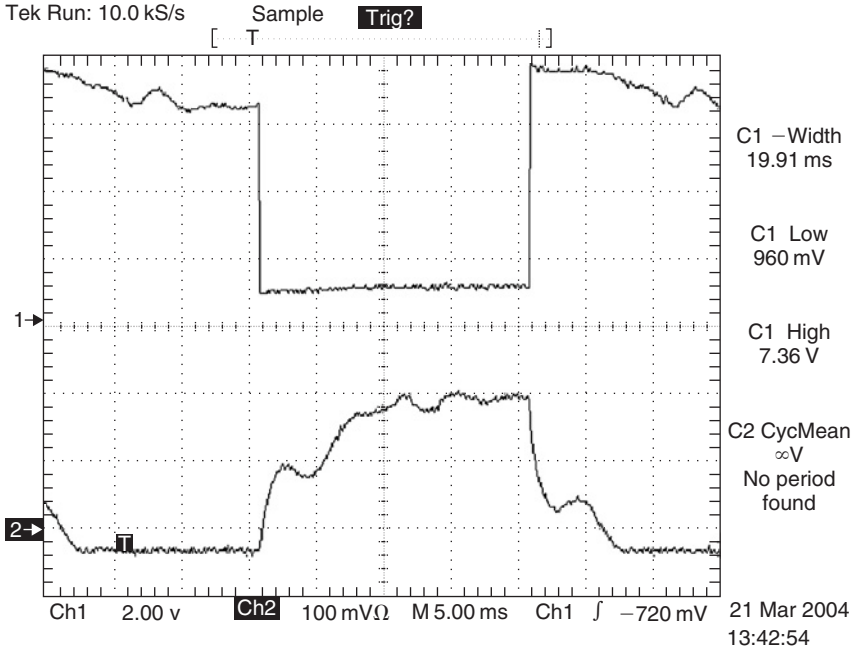


Figure 21.22: Expanded View of Current Waveform; Ch1: Pin 16 ULN2003A; Ch2: Current Through Motor Winding Connected to Pin 16

current rise is faster before the kink than afterward. Figure 21.22 provides an expanded view of the current waveform.

What causes the kink? There are two factors at work here. First, when the rotor moves, its magnetic field induces a current back into the windings, with the direction opposing the original current. Second, as the rotor moves, the internal geometry of the motor is different, giving rise to a change in inductance of the winding. From circuit theory, we know the relationship between a constant inductance L and the rate of change of current for a constant applied voltage V is:

$$V = L\frac{di}{dt}$$

Rearranging we find:

$$\frac{di}{dt} = \frac{V}{L}$$

This relationship may be interpreted as V/L being the slope of the curve of current versus time plot. If we substitute finite changes, i.e., $\frac{\Delta I}{\Delta T}$ for the infinitesimal $\frac{di}{dt}$, we determine the winding inductance before the kink as 58 mH and after the kink as 133 mH. These values should only be regarded as approximate, as there are several sources of error in this simplistic approach, not the least of which being L is not a constant but is rather changing with the motor shaft position.

Finally, you may ask why, since we set the parameter `Mdelay` at 10 ms, the current pulse measures 20 ms duration? To answer this question, consider the output pattern of a full step drive (to clarify the repetitive nature, I've include a fifth step that takes us back to the starting point):

Step (Start time)	B0	B2	B1	B3
1 (t = 0 ms)	1	1	0	0
2 (t = 10 ms)	0	1	1	0
3 (t = 20 ms)	0	0	1	1
4 (t = 30 ms)	1	0	0	1
5 (t = 40 ms)	1	1	0	0

A high on B0…B3 corresponds to a low at the Darlington output, and current flow. As we see, each winding is held low for the duration of two consecutive steps, or 20 ms in this case.

Let's see if we can improve the current rise time. Figure 21.23 shows one simple approach to speeding up the current rise. Although our motor is rated at 5 V, we'll use a series resistor and higher voltage to speed up the current rise. Why does a resistor improve rise time? As we learned earlier, the current rise time is proportional to L/R, where R is the total resistance in

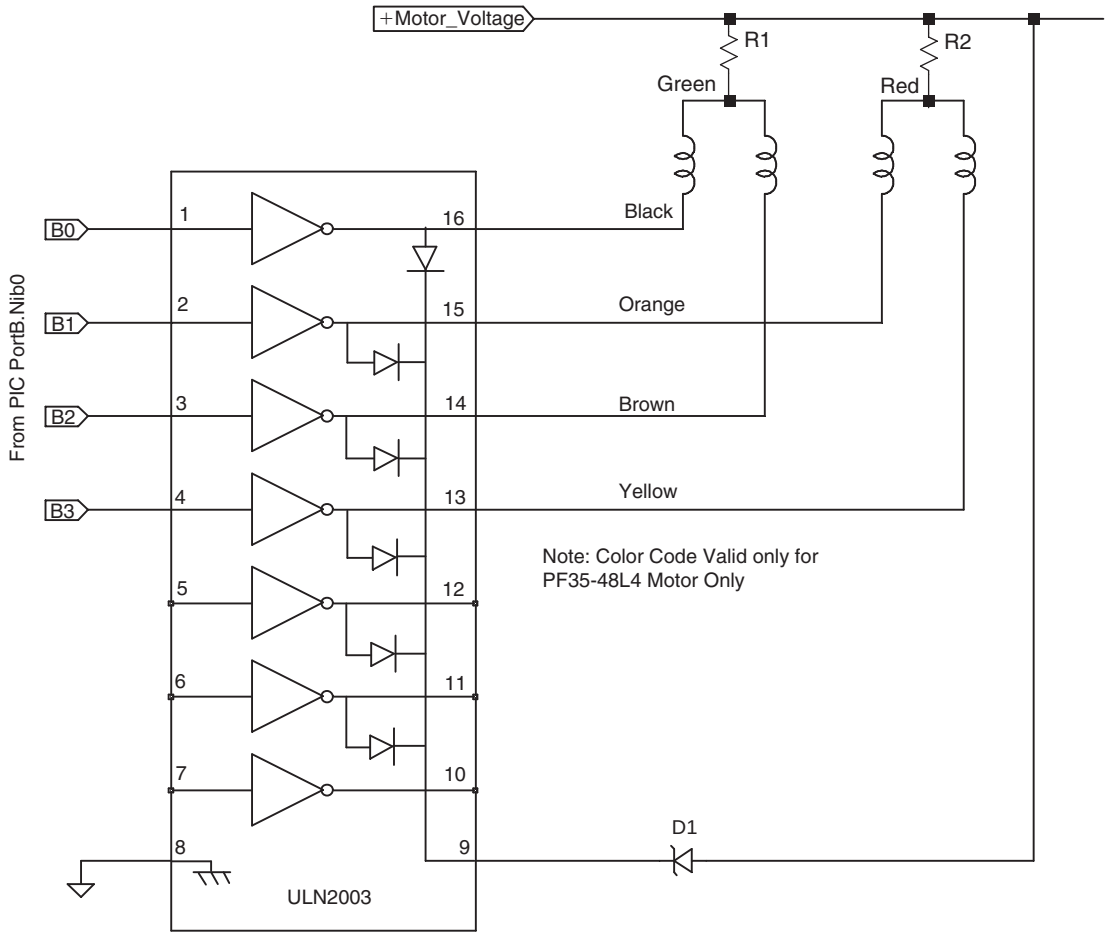


Figure 21.23: L/R Drive for Unipolar Motor

the series circuit. Since L is governed by the motor's construction, and hence not alterable by us, we may reduce the quotient L/R by increasing R. Alternatively, you may wish to think of this approach as using a higher voltage and a series resistor to approximate a constant current source. This approach is sometimes called an "L/R drive" design.

We can't of course, select a series resistance and voltage randomly. The resistance must be chosen to provide the rated motor current, based upon the total series resistance, and the desired drive voltage. Figure 21.24 shows the relevant resistances and voltage drops we must consider. We've previously measured the voltage drop V_{SAT} across Q1 at 1 V. And, I measured $R_{internal}$ at 20 ohms for my PF35-48 motor. We previously calculated the motor's rated current as 265 mA. The relationship between R1, V1 and I_{motor} is:

$$R_1 = \frac{V_1 - V_{SAT}}{I_{motor}} - R_{\text{Internal}}$$

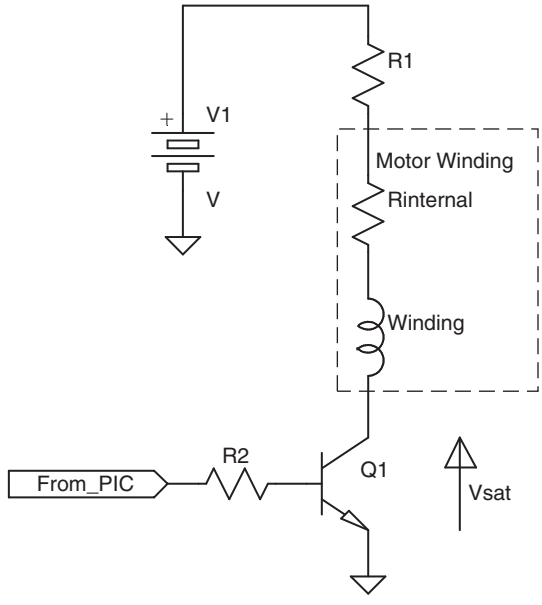


Figure 21.24: Calculating Motor Series Resistor Value

Suppose we wish to feed the motor from a 12 V supply. What is R1's value?

$$R_1 = \frac{12 - 1}{0.265} - 20 = 21.5\ \Omega$$

To illustrate the difference in rise time versus series resistance, I conducted tests with 20 ohms, 40 ohms and 50 ohms series external resistance. The test voltages I used differ slightly from theoretically exact values, but are within reasonable tolerances. I've also measured the inductance of a winding as 8.5 mH at 1000 Hz with a General Radio 1650A RLC bridge and we'll use this value for calculating L/R. (Yes, I know this value doesn't come close to the values we estimated earlier from the rise time data, but it closely matches the data sheet value.)

Parameter	R1 = 0	R1 = 20	R1 = 40	R1 = 50
Supply Voltage	5	11	16	18
L/R (μs)	425	212	142	121
Figure	Fig 21.21	Fig 21.25	Fig 21.26	Fig 21.27

Comparing the following four figures shows significant improvement in the initial current rise time. We also see diminishing returns setting in. The maximum step rate of the PF35-48 is 550 pps if we ramp up the speed from a standing start. If we use wave excitation, each motor current pulse would be 1.8 ms, as unlike full step excitation, the pulse length is not doubled. If

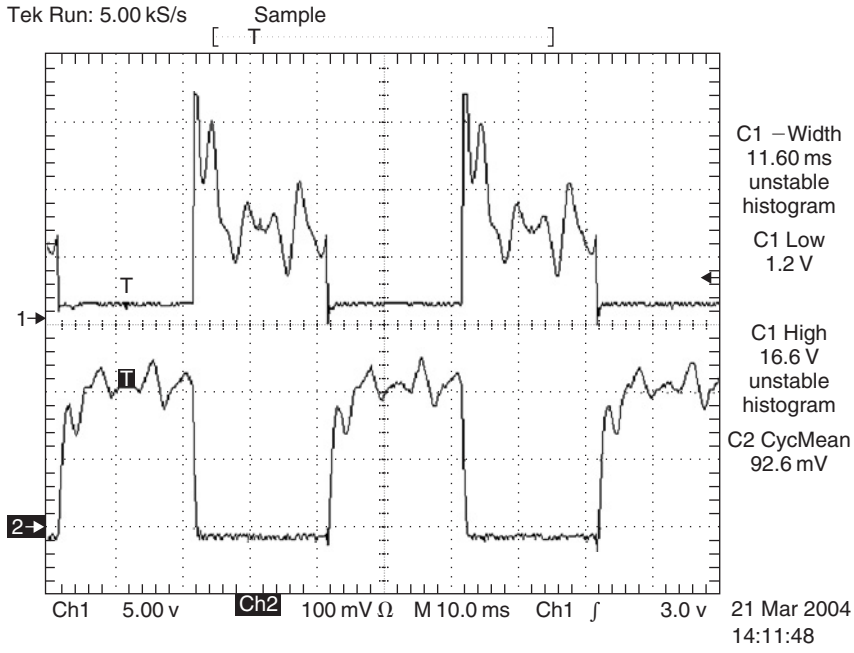


Figure 21.25: 20 Ohms Series Resistance and 11 V Supply Ch1: Pin 16 ULN2003A; Ch2: Current Through Motor Winding Connected to Pin 16

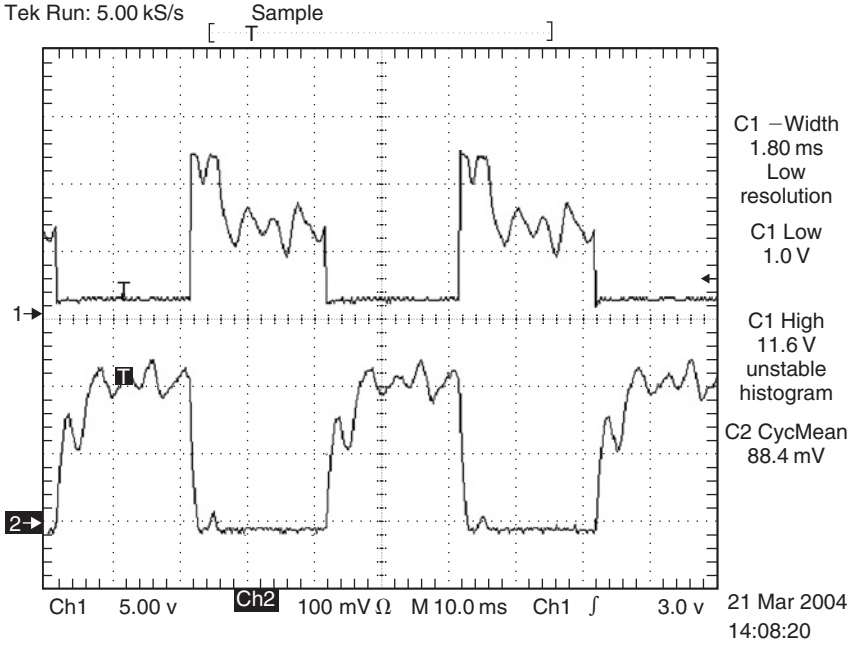


Figure 21.26: 40 Ohms Series Resistance and 16 V Supply Ch1: Pin 16 ULN2003A; Ch2: Current Through Motor Winding Connected to Pin 16

we desire the rise time to be roughly 10% of the pulse length, we see that the series resistance should be between 20 and 40 ohms. If we calculate it exactly, for a rise time of 180 μs, we determine that the total resistance must be at least 47 ohms, which requires 27 ohms external resistance plus the windings internal 20 ohms. The required supply voltage is then 13.5V, allowing 1 V for V_{SAT} drop in the ULN2003A. We should also check the resistor dissipation. We've designed the voltage and resistance value based on 265 mA, so we can calculate the resistor's dissipation by $P = I^2R$.

$$P = I^2R = 0.265^2 \times 27 = 1.9\,\text{W}$$

In full step operation, the average power dissipated will be one half this value. However, if we operate the motor in the locked position—that is, a winding powered up to lock the rotor in place with the motor's holding torque, R1 will dissipate the full 1.9 W. R1 should accordingly be a 27-ohm, 5 W resistor. When stopped, `SPMotor` retains the output value of the last step position, so two windings will be energized unless you explicitly drop all output pins low.

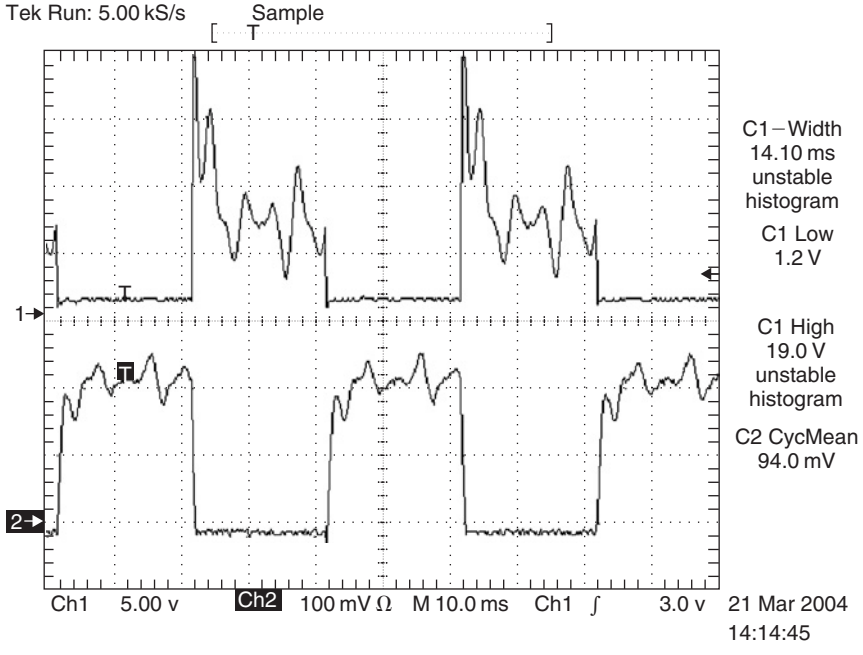


Figure 21.27: 50 Ohms Series Resistance and 18 V Supply Ch1: Pin 16 ULN2003A; Ch2: Current Through Motor Winding Connected to Pin 16

We will make one final measurement—how high a voltage spike might we expect without a clamping diode? If you wish to make this measurement, reconfigure your circuit to the arrangement shown in Fig. 21.28. We are intentionally operating the ULN2003A in an unsafe environment, so there is a risk that your ULN2003A will be destroyed. If you don't want to

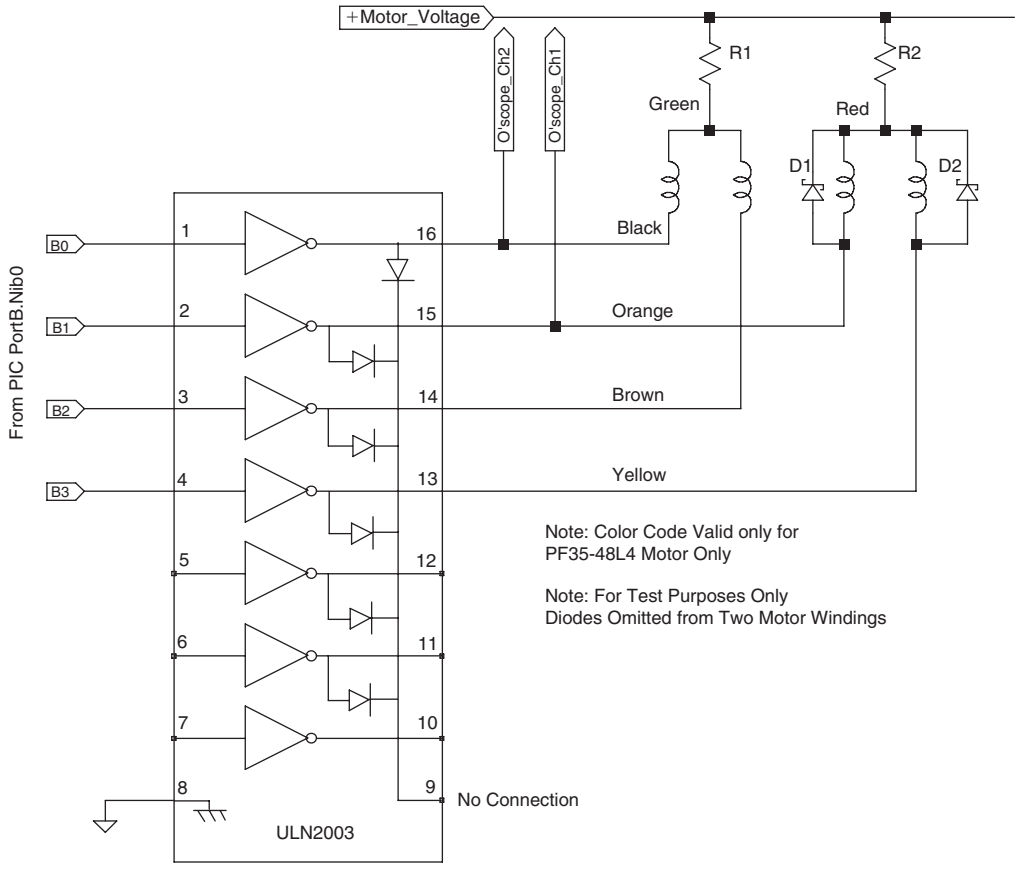


Figure 21.28: Configuration for Voltage Spike Test

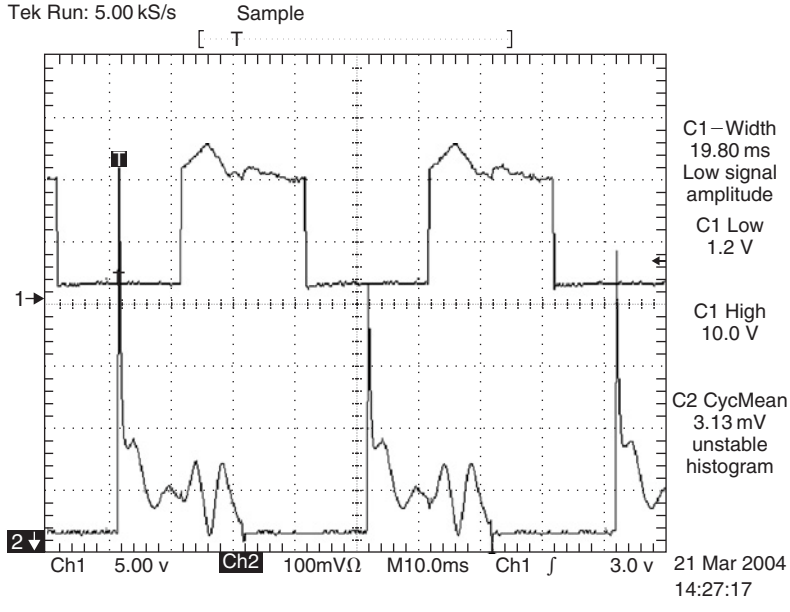


Figure 21.29: Spikes from Collapsing Field; Ch1: Clamped with Schottky Diode; Ch2: Unclamped

risk your ULN2003A, skip this test. Figure 21.29 shows that without a clamping diode, spikes reach close to 30 V. The clamping diodes I used are 1 A, 50 V Schottky devices.

Program 21.2

From version 5.3.0.0 onward, MBasic's `SPMotor` can be commanded to operate in either full step or half step modes. By adding the line `SPMOTOR_Half CON 0` to the program, we may force `SPMotor` to operate in half step mode. The configuration line `SPMOTOR_Half CON 0` is a compile-time option, i.e., you may not switch between half and full step operation while your program is running. Rather, `SPMotor` will either cause full step operation (the default) or half step operation if you include the command `SPMOTOR_Half CON`. (The associated CD-ROM contains an example of half-step operation of `SPMotor` in a second version of Program 21.1. Of course, this program requires MBasic 5.3.0.0 or later to correctly operate.)

To provide more flexibility, let's look at writing our own routines for both wave and half-step modes. While we're at it, we'll include the full step mode. Program 21.2 is compatible with any of the circuits we used with Program 21.1.

```
;Demo Full, Half and Wave with ULN2003A

;Constants
;----------
SeqLen       Con       8          ;how many patterns for repeat
MPin         Con       B0         ;starting pin for motor driver
MDelay       Con       5          ;milliseconds per step
                                  ;each MStep causes 8 motor cycles
MStep        Con       1000       ;number of cycle repeats to make

;Following hold the output patterns for each stated mode.
;Full and Wave are 4 steps long, so we repeat each to make
;all patterns 8 steps long.
HalfStep  ByteTable 8,12,4,6,2,3,1,9
FullStep  ByteTable 12,6,3,9,12,6,3,9
WaveStep  ByteTable 8,4,2,1,8,4,2,1

;Variables
;---------
i         Var       Word
j         Var       Word
MPort     Var       PortB.Nib0

;Initialization
;------------
For j = MPin to Mpin+3
        Output j
```

Program 21.2

```
Next
MPort = 0

Main
      ;First, full step drive
      For i = 0 to MStep-1
            For j = 0 to SeqLen-1
                  MPort = FullStep(j)
                  Pause MDelay
            Next ;j
      Next

      MPort = 0
      Pause 1000

      ;Next half step
      For i = 0 to MStep-1
            For j = 0 to SeqLen-1
                  MPort = HalfStep(j)
                  Pause MDelay
            Next ;j
      Next

      MPort = 0
      Pause 1000

      ;Lastly wave drive
      For i = 0 to MStep-1
            For j = 0 to SeqLen-1
                  MPort = WaveStep(j)
                  Pause MDelay
            Next ;j
      Next

      MPort = 0
      Pause 1000
GoTo Main
```

Program 21.2: Continued

The concept behind Program 21.2 is simple; we write the binary pattern corresponding to the desired motor step sequence to four consecutive pins on a port. Let's recap the stepping sequence for all three modes:

Mode	Step	Bit 3	Bit 2	Bit 1	Bit 0	Decimal
Full	1	1	1	0	0	12
	2	0	1	1	0	6
	3	0	0	1	1	3
	4	1	0	0	1	9

(Continued)

Mode	Step	Bit 3	Bit 2	Bit 1	Bit 0	Decimal
Half	1	1	0	0	0	8
	2	1	1	0	0	12
	3	0	1	0	0	4
	4	0	1	1	0	6
	5	0	0	1	0	2
	6	0	0	1	1	3
	7	0	0	0	1	1
	8	1	0	0	1	9
Wave	1	1	0	0	0	8
	2	0	1	0	0	4
	3	0	0	1	0	2
	4	0	0	0	1	1

I've added a column "decimal" to the table to show the numerical value of the four pattern bits. Perhaps the most convenient form to hold these pattern bits is a byte table. We can construct three byte tables, one for half-step, one for full step and one for wave drive.

```
HalfStep  ByteTable 8,12,4,6,2,3,1,9
FullStep  ByteTable 12,6,3,9,12,6,3,9
WaveStep  ByteTable 8,4,2,1,8,4,2,1
```

The half-step table entries follow the sequence directly. The full and wave byte tables, however, double up the values, so that we use eight entries to hold two complete four-step patterns. We do this to make the three byte tables identical length, as it lets us use the same code to output any of the three modes. (We'll see the benefit of this approach more in Program 21.3.)

```
;Constants
;----------
SeqLen      Con     8         ;how many patterns for repeat
MPin        Con     B0        ;starting pin for motor driver
MDelay      Con     5         ;milliseconds per step
                              ;each MStep causes 8 motor cycles
MStep       Con     1000      ;number of cycle repeats to make
```

We've used the same names as in Program 21.1 for the key constants. However, we've deviated from how `Mstep` is used. In MBasic's `SPMotor` function, `Mstep` defines how many full steps the motor makes. For simplicity, we use `Mstep` to define how many 8-step cycles Program 8.2 outputs. (We'll get back to a step-based approach in Program 21.3.) Thus, if we set `Mstep` to 1000, Program 21.2 outputs 1000*8, or 8000 motor steps.

```
MPort       Var         Port B.Nib0
```

We also define one variable, `Mport`, to use when addressing the output port. Since we only need four pins, we use a nibble. For consistency with Program 21.1, we use Port B's low nibble.

```
For j = MPin to Mpin+3
        Output j
Next
MPort = 0
```

We initialize the four output pins to be outputs.

```
Main
        ;First, full step drive
        For i = 0 to MStep-1
                For j = 0 to SeqLen-1
                      MPort = FullStep(j)
                      Pause MDelay
                Next ;j
        Next
```

To output full step, we loop through the byte array `FullStep`, sending each pattern in sequence. We repeat this `Mstep` times.

```
        MPort = 0
        Pause 1000
```

At the completion of outputting the full steps, we de-energize all windings and pause for one second.

```
        ;Next half step
        For i = 0 to MStep-1
                For j = 0 to SeqLen-1
                      MPort = HalfStep(j)
                      Pause MDelay
                Next ;j
        Next

        MPort = 0
        Pause 1000
```

We repeat the same sequence, except outputting the half step pattern.

```
        ;Lastly wave drive
        For i = 0 to MStep-1
                For j = 0 to SeqLen-1
                      MPort = WaveStep(j)
                      Pause Mdelay
        Next ;j
        Next
```

```
            MPort = 0
            Pause 1000
    GoTo Main
```

We conclude with wave drive, and when completed go back to `Main` and start the sequence over again.

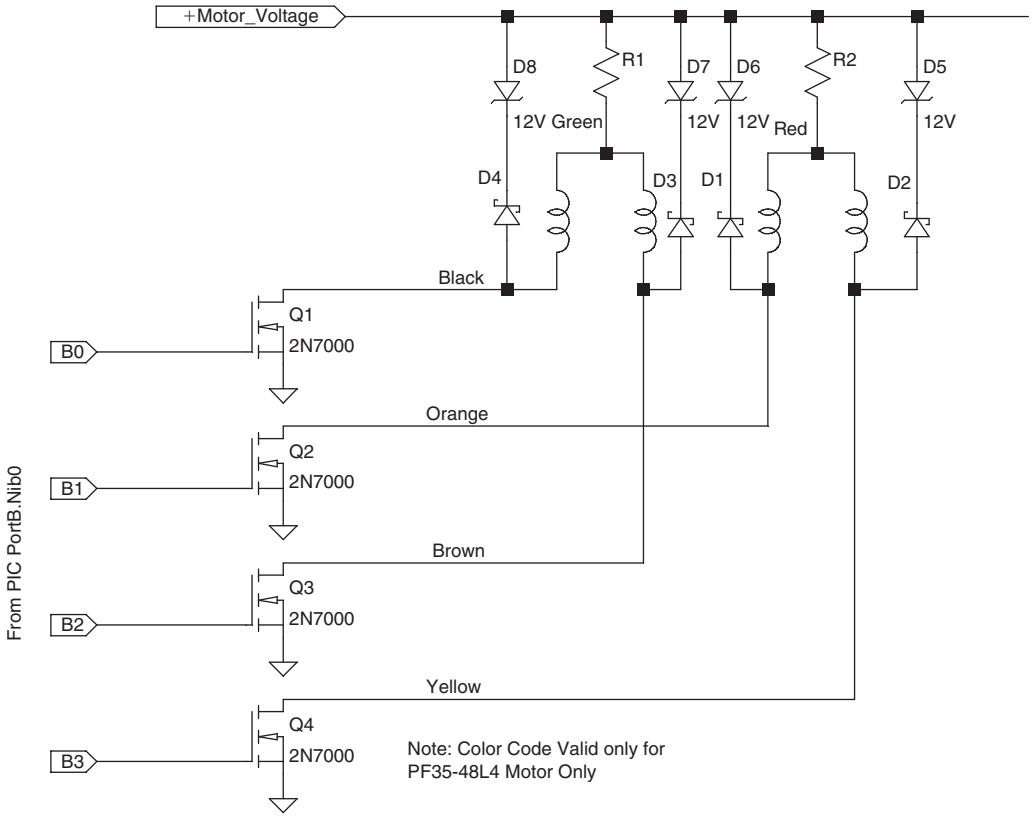


Figure 21.30: Unipolar Driver Using 2N7000s

After verifying Program 21.2 with one of the ULN2003A circuits we used with Program 21.1, let's try an alternative driver. In Chapter 19, we saw how efficient a small MOSFET, the 2N7000, is as a low side driver, so let's replace the ULN2003A with four 2N7000s. Figure 21.30 shows our circuit. I've shown four Schottky diodes as clamps, because I didn't have four suitable Zener diodes at hand. But, you can significantly improve the current release time by adding Zener diodes in reverse series with the four Schottky diodes. Incidentally, if your motor only has five windings, the connection arrangement shown in Fig. 21.31 may be used. If we restrict ourselves to full step or wave drive, we can use a simpler arrangement; use one series resistor for all windings (but only one mode for any resistor arrangement). This is possible because the same number of windings will always be on; one in the case of wave drive or two in

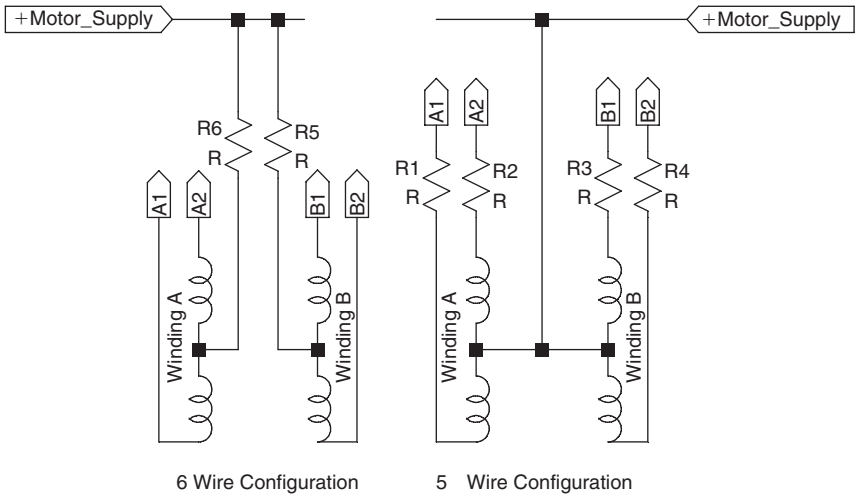


Figure 21.31: Connection Arrangement Differs for Five and Six Wire Unipolar Steppers

the case of full step drive. Thus, the circuit has a constant load and one current limiting resistor is adequate for each mode. (You can't use the same resistor for wave and for full step operation, of course.) However, if we operate half-step mode, the current alternates between one and two windings energized, so a single current limiting resistor is not acceptable. Figure 21.32 shows the overlap between two motor windings during full step operation.

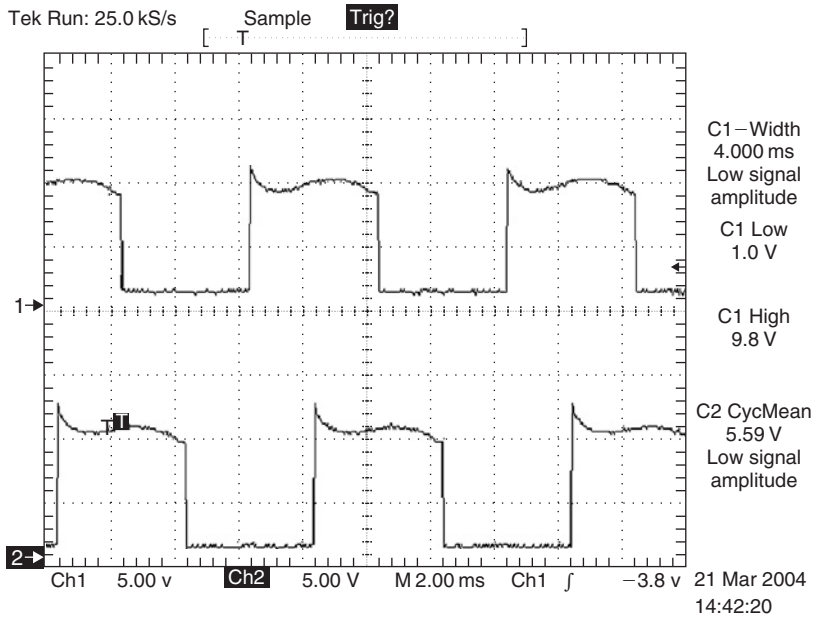


Figure 21.32: 2N7000 Driver Output During Full Step Operation; Ch1: Drain Voltage Q1; Ch2: Drain Voltage Q2

Program 21.3

We'll wrap up our excursion into the world of stepper motors with a program that demonstrates an improved approach to the technique introduced in Program 21.2. We'll encapsulate the nuts and bolts of outputting the stepper pattern in a subroutine. We'll also look at bipolar drivers and show how to connect a unipolar motor as a bipolar device. To verify the operation of Program 21.3, you may start with any of the circuits we've developed for Programs 21.1 and 21.2.

```
;Demo Full, Half and Wave with
;SN754410 unipolar and ULN2003A bipolar operation
;Quasi-Function subroutine

;Constants
;----------
MPin          Con     B0        ;starting pin for motor driver
MDelay        Con     5         ;milliseconds per step
                                ;unlike program 8-2, MStep is number of
                                ;steps
MStep         Con     1000      ;number of steps to make
                                ;can't be more than 65535.

SeqLen        Con     8         ;how many patterns for repeat
ModeFull      Con     0         ;mode constants
ModeWave      Con     8
ModeHalf      Con     16

DFwd          Con     1         ;direction constants
DRev          Con     -1        ;fwd and rev are arbitrary

;0..7 posn = full, 8...15=wave 16-23 = half
;for full & wave repeat for cycle length = 8
StepTable ByteTable |
12,6,3,9,12,6,3,9,8,4,2,1,8,4,2,1,8,12,4,6,2,3,1,9

;Variables
;---------
i             Var     Word          ;counts steps
j             Var     SByte         ;indexes patterns
MPort         Var     PortB.Nib0    ;output port pins
Direct        Var     SByte         ;direction
Mode          Var     SByte         ;mode type

;Initialization
;------------
;set motor port to output
For j = MPin to Mpin+3
              Output j
```

Program 21.3

```
Next
;all windings off
MPort = 0

Main
      ;Full step forward
      Mode = ModeFull
      Direct = DFwd
      GoSub BiPolar
      MPort = 0
      Pause 1000
      ;Full step reverse
      Direct = Drev
      GoSub BiPolar
      MPort = 0
      Pause 1000

      ;Half step forward
      Mode = ModeHalf
      Direct = DFwd
      GoSub BiPolar
      MPort = 0
      Pause 1000
      ;Half step reverse
      Direct = Drev
      GoSub BiPolar
      MPort = 0
      Pause 1000

      ;Wave step forward
      Mode = ModeWave
      Direct = DFwd
      GoSub BiPolar
      MPort = 0
      Pause 1000
      ;Wave step reverse
      Direct = Drev
      GoSub BiPolar
      MPort = 0
      Pause 1000

GoTo Main

;Called subroutine needs MPin, MStep and MDelay set
;Direct and Mode also must be set. Motor pattern in
;Byte Array StepTable()
;and needs i & j free variables
BiPolar
;------
```

Program 21.3: Continued

```
      j = Mode    ;starting point into table
      ;If reverse, then start at end of table entry
      If Direct = DRev Then
            j = j + SeqLen-1
      EndIf

      ;Step through the required number of steps
      For i = 0 to MStep-1
            MPort = StepTable(j)    ;output pattern
            Pause MDelay                  ;wait
            j = j + Direct                ;next pattern no.
            ;only have j < Mode if reverse stepping
            ;check and wrap back to starting point
            If (j < Mode) AND (Direct = DRev) Then
                        j = Mode + SeqLen -1
            EndIf
            ;Likewise j>= Mode+SeqLen if forward. Check for
            ;overrun and wrap back to starting point
            If (j >= (Mode+SeqLen)) AND (Direct = DFwd) Then
                        j = Mode
            EndIf
      Next
Return
```

Program 21.3: Continued

Program 21.3 uses the same constant names as in Programs 21.1 and 21.2 to define the starting pin, the delay (in milliseconds) per step and the number of steps to take. In order to more closely mimic the functionality of the function `SPMotor, Mstep` in Program 21.3 defines the number of steps, not the number of 8-step cycles as in Program 21.2.

```
MPin      Con    B0      ;starting pin for motor driver
MDelay    Con    5       ;milliseconds per step
                         ;unlike program 8-2, MStep is number
                         ;of steps
MStep     Con    1000    ;number of steps to make
```

Although I've made `Mdelay` and `Mstep` constants, you may wish to make them variables and compute or assign their values based on user inputs or calculated parameters. Since we use a word value counter to keep track of the number of steps, `Mstep` can't exceed 65535. If this is inadequate, change the counter variable i to be a type long.

It's also necessary to define a variable to address the output port nibble. I've named this variable `Mport`.

```
MPort     Var    PortB.Nib0    ;output port pins
```

If you use different pin connections, don't forget to change both `Mpin` and `Mport`.

```
ModeFull            Con      0        ;mode constants
ModeWave            Con      8
ModeHalf            Con      16

DFwd                Con      1        ;direction constants
DRev                Con      -1       ;fwd and rev are arbitrary

Direct      Var     SByte             ;direction
Mode                Var      SByte    ;mode type
```

We set the direction and operational mode using two variables, `Direct` and `Mode`. In order to make our program more intelligible, we define three mode constants, `ModeFull, ModeWave` and `ModeHalf`, representing full step, wave and half-step operation, respectively. We'll see the reason for choosing values of 0, 8 and 16 for these constants a bit later. Likewise, we define two directional constants, `DFwd` and `DRev` for forward and reverse. Your choice of motor winding connection, as we covered in connection with Program 21.2, determines whether forward is clockwise or counterclockwise.

```
StepTable ByteTable 12,6,3,9,12,6,3,9,8,4,2,1,8,4,2,1,8,12,4,6,2,3,1,9
```

Rather than keep three separate byte tables to hold the pattern for full step, wave and half step, I've combined them into one 24-element byte table. Elements 0…7 are the full step patterns, 8…15 are the wave patterns and 16…23 are the half-step patterns. As with Program 21.2, `StepTable` holds two complete cycles of full step and wave patterns, so that full, wave and half-step all have eight entries. We also now see why the constants `ModeFull, ModeWave` and `ModeHalf` are defined with the values 0, 8 and 16—these numbers are the starting point for indexing into `StepTable` for their respective modes.

```
;Initialization
;------------
;set motor port to output
For j = MPin to Mpin+3
        Output j
Next
;all windings off
MPort = 0
```

We initialize by setting output state the pins we have selected to control the motor and we de-energize all the windings. In the de-energized mode, the stepper has only its residual torque to hold the rotor against external torque, so you may instead wish to initialize the motor with one winding energized.

Let's now examine the subroutine `Bipolar`. I named this subroutine `Bipolar` because I originally intended to demonstrate Program 21.3 only with a bipolar motor, but later decided this is too limiting. The step pattern, if you use the correct driver and motor connections,

is identical for a unipolar and bipolar motor. Hence, if properly connected Program 21.3 functions identically with either motor type.

Subroutine `BiPolar` assumes we have defined and set values for `MPin, MStep, MDelay, Direct, Mode,` and that `Mport` is aliased with the port nibble to be used. In addition, it uses two variables `i`, type word, and `j`, type sbyte.

```
BiPolar
;------
      j = Mode    ;starting point into table
      ;If reverse, then start at end of table entry
      If Direct = DRev Then
            j = j + SeqLen-1
      EndIf
```

To reverse direction, we step through `StepTable` in reverse order, that is, emitting the values of `StepTable` in the sequence `StepTable(0)`, `StepTable(1)` …`StepTable(7)` is forward, while the sequence `StepTable(7)`, `StepTable(6)` …`StepTable(0)` is reverse. Hence, based upon the value of `Direct`, we start indexing into `StepTable` at 0,8 or 16 for forward, or 7, 15 or 23 for reverse. The constant `SeqLen` holds the number of elements in `StepTable`—8—for each mode. We enter the program control loop, therefore, with `j` properly set at the first element in the pattern we wish to send to the motor.

```
      ;Step through the required number of steps
      For i = 0 to MStep-1
            MPort = StepTable(j)    ;output pattern
            Pause MDelay            ;wait
            j = j + Direct          ;next pattern no.
```

We now emit `Mstep` number of steps, using a loop construction to control program execution. (Since we start with 0, to send `Msteps` steps, we must stop at `MStep-1`.) After the pattern for each step is emitted we update `j` with the statement `j = j + Direct. Direct` is either +1 (forward) or −1 (reverse) so `j` increments or decrements based upon the chosen direction of rotation,

```
      ;only have j < Mode if reverse stepping
      ;check and wrap back to starting point
      If (j < Mode) AND (Direct = DRev) Then
            j = Mode + SeqLen -1
      EndIf

      ;Likewise j>= Mode+SeqLen if forward. Check for
      ;overrun and wrap back to starting point
      If (j >= (Mode+SeqLen)) AND (Direct = DFwd) Then
            j = Mode
      EndIf
   Next
Return
```

We must limit j's value so that it wraps around when it attempts to go outside the range of values permitted for the selected mode. We do this with two IF...Then statements. If the direction is forward (Direct=DFwd) we check to see if j exceeds the maximum index, which is Mode + SeqLen. If this is the case, we reset the index by the statement j=Mode.

If the direction is reverse (Direct=Drev) we check for an under-run—that is, j is less than Mode. If this is the case, we reset the index by the statement j=Mode+SeqLen-1.

To show how subroutine BiPolar is called, we return to the main program loop.

```
;Full step forward
Mode = ModeFull
Direct = DFwd
GoSub BiPolar
MPort = 0
Pause 1000
;Full step reverse
Direct = DRev
GoSub BiPolar
MPort = 0
Pause 1000
```

We set the variables that define the mode, Mode, and direction, Direct, and then call BiPolar. Of course, if we made Mstep or Mdelay variables instead of constants, they also would require setting to appropriate values.

The remainder of Program 21.3 demonstrates the remaining modes and directions and requires no further analysis.

After you have verified that Program 21.3 is functioning with a unipolar motor and any of the circuits developed for Programs 21.1 and 21.2, we may now turn to a bipolar driver. Figure 21.33 shows a suitable circuit. Note carefully the input and output connections to the SN754410. I've shown connections for either a pure bipolar motor, or for connecting a 6-wire unipolar motor as a bipolar device. And, the circuit of Fig. 21.33 may be used with the Program 21.1 or 21.2, should you so desire. The H-bridge circuit permits MBasic's SP-Motor function to drive a bipolar motor, not just a unipolar motor as the User's Guide suggests.

Our circuit is based upon an SN754410 quad half-H bridge device. (By combining two half-H bridges, we get a full H-bridge.) Earlier, in Fig. 21.17, we saw a conceptual overview of an "H" bridge. By setting one control pin high and the other low, the SN754410 permits current flow through the winding in one direction. If the two control pins reverse state, current flows through the winding in the reverse direction. Pins 1 and 9 are "enable" pins that must be held at +5 V for their associated bridge drivers to function. The SN754410 has two V_{CC} connections; one for +5 V and the second to be connected to the motor supply. Don't reverse these connections or the "magic smoke" inside the SN754410 will escape and you then can throw what's left into the trash!

The SN754410's superior ratings make it preferable to the more popular L293. And, making it even better, the SN754410 is few cents cheaper than the L293. The SN754410 H-bridge is constructed with Darlington transistors in both the high side and low side switches. Hence, we anticipate approximately a 1 V drop in both the high and low side motor connections. In computing the current limiting resistors, don't forget to include both voltage drops. In our application, the increased voltage drop is not of major concern, but it might be for larger motors. Fortunately, a wide variety of H-bridge circuits are available, including those with less voltage drop and higher current ratings than the SN754410.

Figure 21.33 also shows eight Schottky voltage clamp diodes, based on the recommended design in the SN754410's data sheet. The SN754410 contains internal clamping diodes and for low power stepper motors, the internal diodes may be adequate. However, adding external clamping diodes at the motor itself is an excellent safety precaution, and should be followed. Schottky diodes have the dual advantage of being much faster than conventional silicon power diodes of the 1N400X family and also provide a lower forward voltage drop.

To demonstrate how Program 21.3 operates in the bipolar mode, I used the same unipolar PF35-48 motor that appears throughout this chapter. In this case, however, we operate with two windings in series, so the winding resistance is 40 ohms, not the 20 ohms seen for each individual winding.

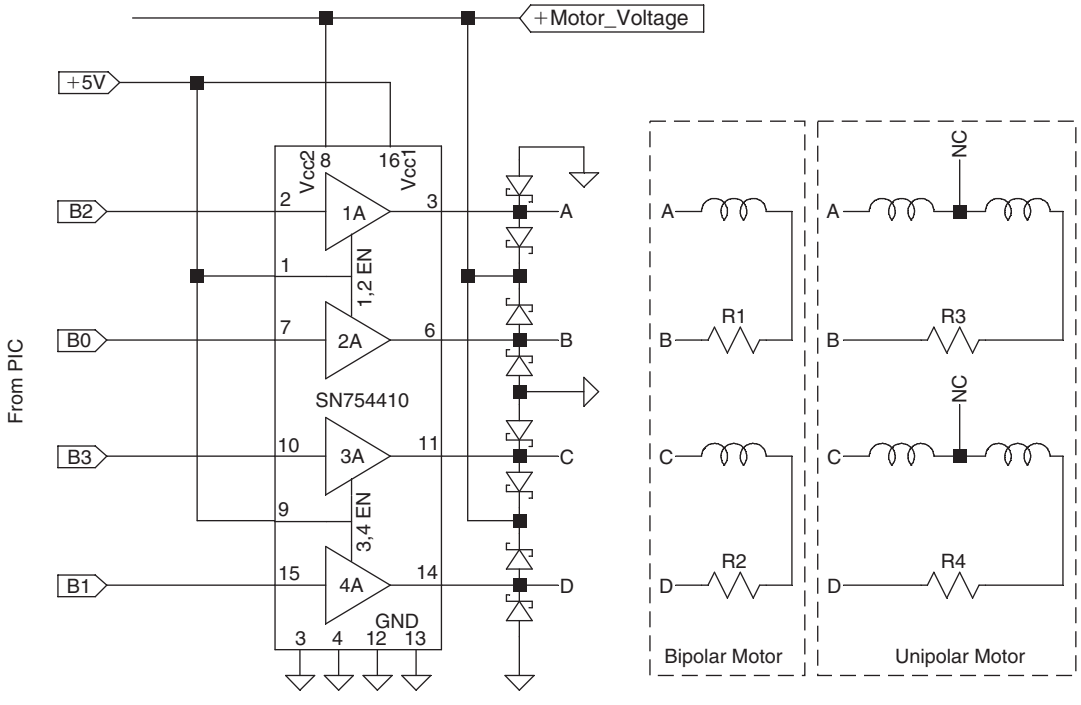


Figure 21.33: Bipolar Motor Driver

At what voltage and current should we operate the PF35-48? Since each winding in the unipolar mode is 5.4 V, we should operate the series connection at 10.8 V, thereby placing 5.4 V across each winding, corresponding to 265 mA, right? Well, maybe not. Let's go back to the power analysis we conducted earlier in this chapter. In the unipolar mode, full step excitation, the PF35-48 is rated at about 1.4 W dissipation in each of the two windings that are simultaneously energized, or about 2.8 W for the entire motor. In unipolar operation, there is no case in which all four windings are simultaneously energized. In full step bipolar operation, all four windings are energized. If we operate the motor at 10.8 V, therefore, the total power dissipation will be 5.6 W, twice the rated value. Instead, we have to reduce the current through the windings so that each pair of simultaneously energized windings dissipates 1.4 W.

We know that $P = I^2R$, so we can calculate the safe maximum current, based upon a total motor dissipation of 2.8 watts.

$$P = I^2R; I = \sqrt{\frac{P}{R}}$$

$$I = \sqrt{\frac{1.4}{40}} = 187 \text{ mA}$$

Since the total winding resistance is 40 ohms, this current corresponds to 7.5 V. If we allow 1 V for both sides of the H-bridge, we may operate the motor at 9.5 V.

If we wish to improve the current rise time by operating the motor at higher voltage and adding series resistance, we should use 187mA as the safe operating current for full step operation. This calculation is based upon full step operation; if we were to operate the motor in bipolar wave drive, only two windings are energized at any time, so we may use the full unipolar current rating and operate the motor at 10.8 V, plus the H-bridge drops. For half-step operation, the safe current will be between these two values, as the average number of windings energized is 3 (4 steps at 4 windings; 4 steps at 2 windings, for an average of 3 energized windings.)

When driving a unipolar motor in a bipolar circuit, you should be aware that the winding inductance also increases. The bipolar connected winding inductance is approximately four times the inductance of a single winding. The inductance doesn't simply double because the two windings are closely magnetically coupled and their mutual inductance must be taken into account. Since the series resistance only doubles, the L/R time constant of the bipolar connected winding is twice that of the unipolar connection.

Of course, you need not go through these calculations if the motor is of bipolar design, as its ratings will be based upon bipolar operation.

Let's see how the current flow looks when running Program 21.3 with a bipolar driver. Figure 21.34 shows the current flow through one winding in full step operation. The current pulses are

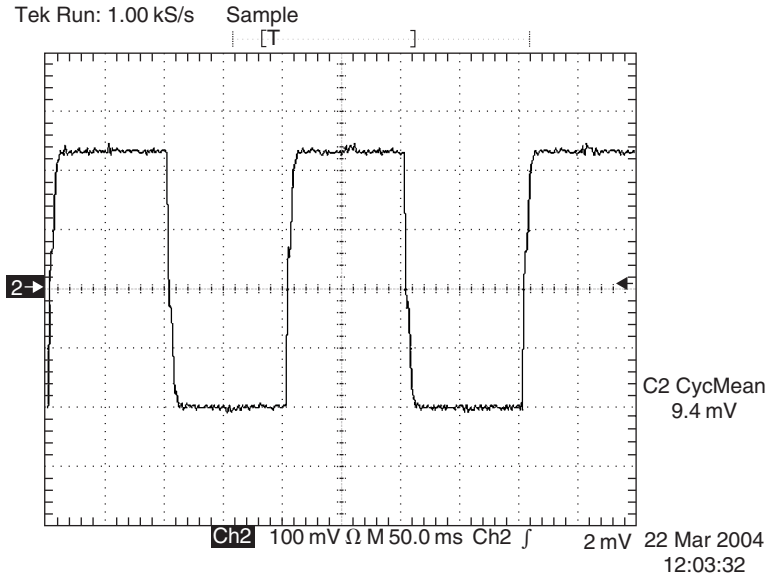


Figure 21.34: Bipolar Operation—Full Step Mode; Ch1: Off; Ch2: Motor Current

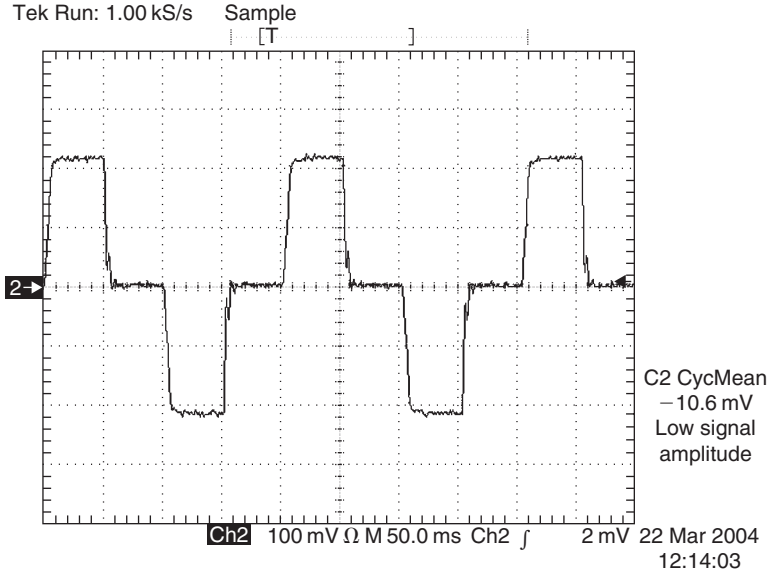


Figure 21.35: Bipolar Operation—Wave Mode; Ch1: Off; Ch2: Motor Current

approximately equal and show the expected alternating direction, +/−/+/−… Figure 21.35 illustrates wave operation. The winding current pattern is +/0/−/0/…, again as expected. Finally, Fig. 21.36 shows half-step operation. The current waveform looks unusual, doesn't it? But, it's exactly what we should expect. Let's look at a piece of the half-wave pattern table.

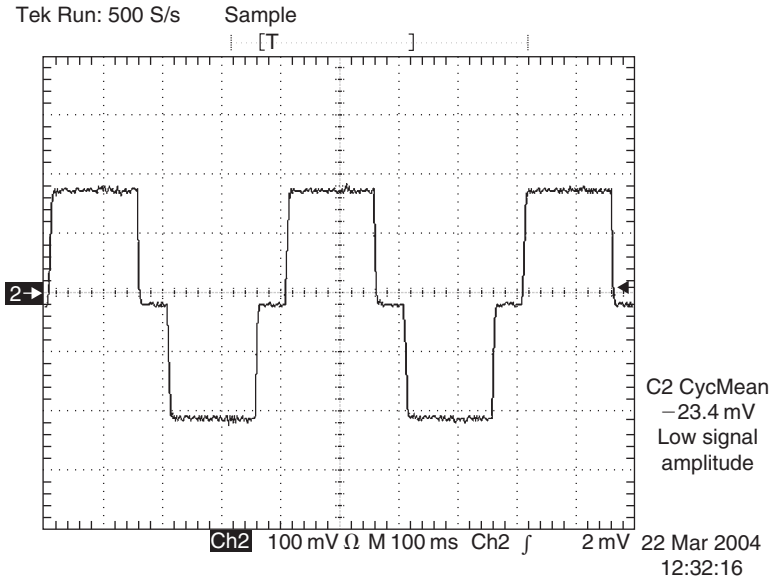


Figure 21.36: Bipolar Operation—Half-Step Mode; Ch1: Off; Ch2: Motor Current

The winding that is shown in Figs 21.34 through 21.36 is connected to the SN754410's pin 3. Hence, its current flow is controlled by Bits 0 and 2 of the output nibble, corresponding to the PIC's output pins B0 and B2 for the particular connection shown in Figure 21.33. (I've omitted Bits 3 and 1 from the table extract.) The SN754410 interprets input commands as 10 = current flow in one direction; 01 = current flow in opposite direction and 00 = no current flow. Based on this, I've added a current flow column, arbitrarily assigning the 10 direction as a plus.

Mode	Step	Bit 2	Bit 0	Current Flow
Half	1	0	0	None
	2	1	0	+
	3	1	0	+
	4	1	0	+
	5	0	0	None
	6	0	1	−
	7	0	1	−
	8	0	1	−

Thus, the current pattern we see should be 0/+/+/+/0/−/−/−, which is an alternating current flow, with a period of no current flow between the reversals. The quiescent period should be one-third the duration of the active current flow period. And, that's exactly what Fig. 21.36 shows.

References

[21.1] Thompson Airpax Mechatronics, *Product Selection and Engineering Guide*, (undated), available for downloading at http://www.thomsonindustries.com/PDFs/Catalogs_and_ Brochures/Airpax_Catalog_ Idx.pdf.

[21.2] Thompson Airpax Mechatronics, *Stepper Motor Handbook*, (undated), available for downloading at http://www.allegromicro.com/techpub2/compumot/a04a08.pdf.

[21.3] Texas Instruments, *SN754410 Quadruple Half-H Driver Data Sheet*, Doc. No. SLRS007B, (Nov. 1995).

[21.4] Telefonaktiebolaget LM Ericsson, *Industrial Circuits Data Book and Stepper Motor Control Handbook*; *Stepper Motor Basics*, (1995) available for download at http://library.solarbotics.net/pdflib/pdf/motorbas.pdf.

[21.5] Sax, H., *Application Note 235: Stepper Motor Driving*, SGS-Thomson Microelectronics (July 1988), available for downloading at http://library.solarbotics. net/pdflib/pdf/motorbas.pdf.

[21.6] Hopkins, Thomas L., *Application Note 460: Stepper Motor Driver Considerations Common Problems and Solutions*, SGS-Thomson Microelectronics (2003), available for downloading at http://www.st.com/ stonline/books/pdf/docs/1675.pdf.

[21.7] Condit, Reston, & Jones, Douglas W., *Stepping Motors Fundamentals*, AN907, Microchip Technology Inc., Doc. No. DS00907A (2004), available for downloading at http://www.microchip.com/download/ appnote/pic16/00907a.pdf.

[21.8] Condit, Reston, *Stepper Motor Control Using the PIC16F684*, AN906, Microchip Technology, Inc., Doc. No. DS00906A (2004), available for downloading at http://www.microchip.com/download/appnote/pic16/00906a.pdf.

[21.9] Yedamale, Padmaraja, *Stepper Motor Microstepping with PIC18C452*, AN822, Microchip Technology, Inc., Doc. No. DS00822A (2002), available for downloading at http://www.microchip.com/download/ appnote/pic16/00822a.pdf.

This page intentionally left blank

CHAPTER 22

Digital Temperature Sensors and Real-Time Clocks

We're going to look at two communications protocols, "1-wire" and "three-wire" (also known as SPI—Serial Peripheral Interface) for serial data exchange between a PIC and external sensors. We'll look at these protocols in the context of two specific devices, the DS18B20 12-bit temperature sensor and the DS1302 real-time clock. (By the way, don't confuse Dallas Semiconductor's DS18B20 sensor with its similar, cheaper, DS18S20 device. The DS18S20 is also a 1-wire temperature sensor, but with 9-bit resolution, yielding 0.5ºC steps.)

In order to keep our discussion of manageable length, we'll briefly mention that these devices are but two examples from the world of sensors. One form or another of electronic sensor can measure almost any physical parameter of interest, directly or indirectly. Historically, sensors used an analog change in an electrically measurable parameter—resistance, capacitance or voltage being the most common—to measure a change in an underlying parameter, such as temperature, humidity, pressure, acceleration, or light intensity. Since the sensor output is analog, the resulting value must be read with an analog instrument, or converted to a digital value with an analog to digital converter.

Pure analog sensors are still widely used, but have been augmented with a built-in digital conversion, so the output can be directly read by a microcontroller, even though the microcontroller doesn't have a built-in ADC. (Don't be fooled by the digital output, however, as in almost every case the underlying physical sensor process remains analog.)

22.1 DS18B20 Temperature Sensor

22.1.1 One-Wire Protocol

Before delving into the specific temperature sensor we'll use, let's explore its 1-wire protocol. (If you want to skip this discussion and jump right into the program code, go ahead, as MBasic does an excellent job of encapsulating the nuts and bolts of 1-wire protocol into the `OWIn` and `OWOut` procedures.)

Maxim Integrated Products' Dallas Semiconductor division developed the "1-wire" protocol as an economical way of exchanging information between microprocessors and ancillary

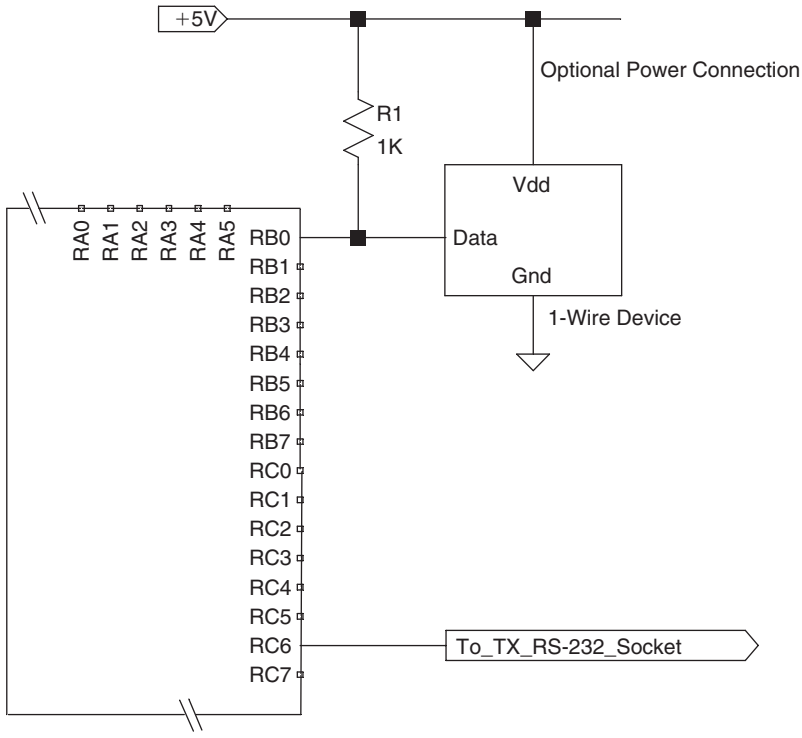


Figure 22.1: Typical 1-Wire Device Connection

chips. It's called a 1-wire network because it uses one wire (plus ground) to communicate with the master microcontroller. One-wire devices are slaves and communicate only when so commanded by the master. Data is transmitted serially over the data line at an equivalent 14 kb/s rate. Figure 22.1 shows how to connect 1-wire devices to a PIC. Dallas has a range of 1-wire devices that include memory, encryption and serial number generation, in addition to temperature sensing.

The 1-wire data protocol differs significantly from others we've seen, however. Although MBasic will hide the complexity for us, let's take a look at typical signaling in a 1-wire device as shown in Fig. 22.2.

Communications from the master PIC to slave 1-wire devices starts with the master outputting a 480 μs low reset pulse, following which the master switches to receive. The 1-wire device responds by taking the data line low sending a "presence" pulse, for approximately 150 μs. (We will use the presence pulse to detect whether or not a 1-wire device is connected.) Following the reset pulse, the master sends commands to the slave and the slave responds. Figure 22.3 shows the complete sequence; the reset/presence pulse, followed by a 1-byte interrogation request sent by the master, with an 8-byte reply by the slave device.

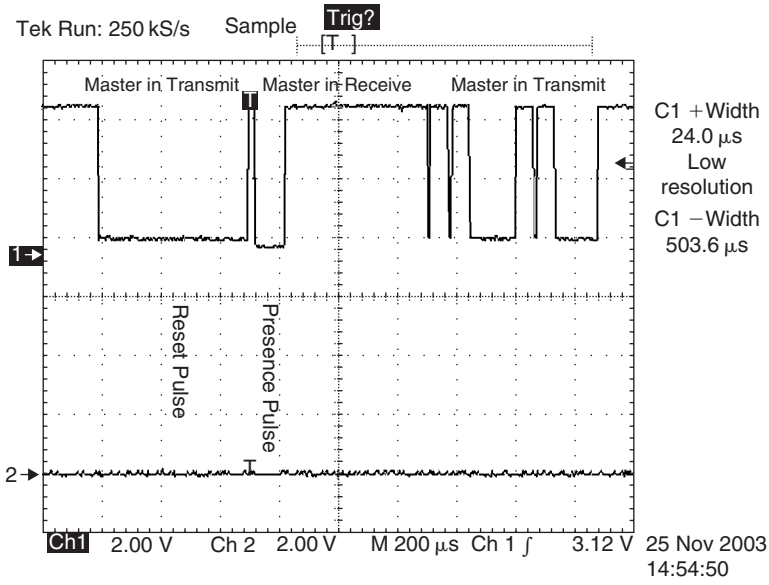


Figure 22.2: Reset and Presence Pulses in 1-Wire Protocol

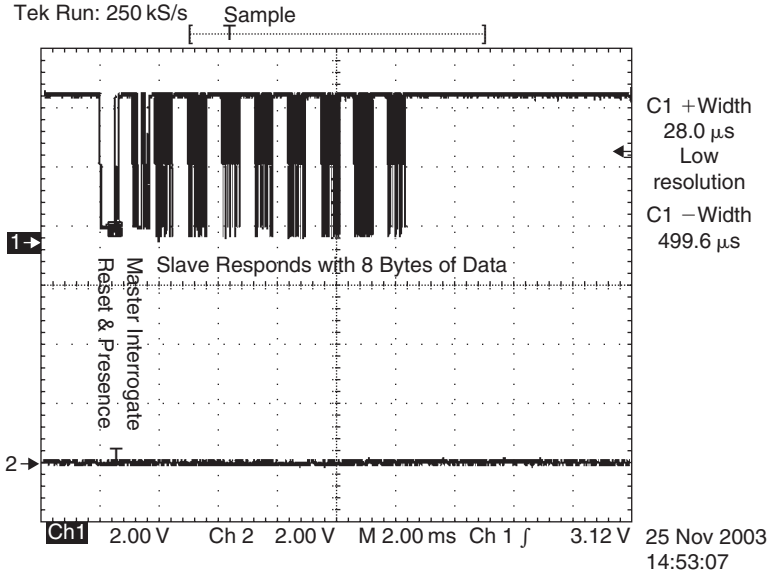


Figure 22.3: Reset—Interrogate—Response in 1-Wire Protocol

Let's take a closer look at how data bits are sent. Figure 22.4 shows the first byte transmitted by a DS18B20 responding to an interrogation request to send its device ID code. (The first byte is the family code, $28 (%00101000) in the case of the DS18B20, sent least significant bit first.) For each of the 8 bits to be received, the master switches to output mode, drops the

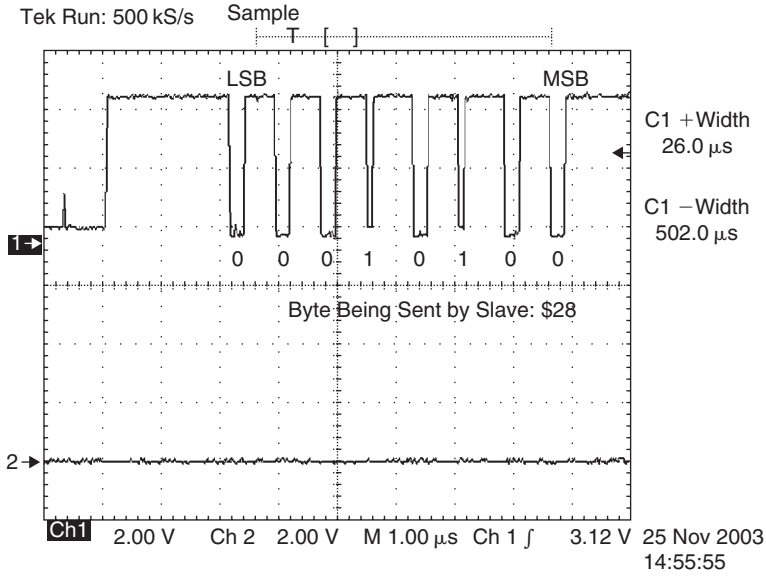


Figure 22.4: Typical Byte Transmission in 1-Wire Protocol

data line low briefly (at least 1 µs), releases the data line and switches to input mode. The data line is then pulled high by the pull-up resistor, R1 in Fig. 22.1. The slave 1-wire device then responds by either pulling the data line low (transmitting a 0) or leaving the data line high (transmitting a 1). The slave will maintain this status for 15 µs. The master must therefore read the data line status within 15 µs after it switches to input mode.

A master writes to the slave using a similar process, but with different timing. To write a 1 to the slave, the master pulls the bus low for 15 µs or less time and either releases the bus to go high (writes a 1 to the slave) or holds the bus low for a total time of 60 µs (writes a 0 to the slave).

Dallas refers to this technique as "time slot" coding—that is, the process of the master taking the data line low is seen as "issuing a time slot." The slave device then responds within the time slot with either a 0 or a 1, and the master then issues the next time slot (if any). When looking at the output on an oscilloscope, we see wide and narrow zero-level pulses, as reflected in Fig. 22.4. A narrow pulse results if a 1 follows the time slot issuance from the slave; a wide pulse results when a 0 follows it.

An integral part of the 1-wire protocol is the ability to obtain operating power for the chips from the data line, without a separate power supply line, sometimes referred to as "parasitic" powering. One-wire devices have an internal capacitor that charges from the data line and power it when sending a 0 V signal. (If we look at it in detail, however, we'll see that parasitic power has its drawbacks, so all of our circuits use external power connections.)

One last comment and we'll see some real code. Each individual 1-wire part produced has its own unique 64-bit serial number and may be addressed either individually through the serial number, or globally, without reference to its serial number.

22.1.2 Reading a One-Wire Devices' Serial Number

Program 22.1 reads the serial number of a one-wire device connected with the circuit shown in Fig. 22.1 and writes it to the serial output port. (When building the circuit don't forget the pull-up resistor!)

```
;Program 22.1
;Read 1-wire device w/ READROM $33 Command
;Write ROM output (Device ID, SN & CheckSum)
;to serial output.
;Only works with 1 device attached!

;Variables
Temp          Var            Byte
i             Var            Byte

;Constants
ReadRom Con          $33

EnableHSerial
SetHSerial H19200

Main
      ;Issue universal READROM Command
      OWOut B0,1,Main,[ReadRom]

      ;Response is 8 bytes
      ;Device Family         1 Byte
      ;SN                    6 Bytes
      ;Check Sum             1 Byte
      For i = 0 to 7
              OWIn B0,0,[Temp]
              HSerOut [Hex Temp," "]
      Next
      ;Write CR
      HSerOut [13]

      Pause 1000
GoTo Main
```

Program 22.1

MBasic's support for 1-wire devices is contained in two procedures; `OWOut` for writing to the 1-wire device and `OWIn` for reading from the 1-wire device. Our first action is to instruct the 1-wire device to report its serial number, through the function: `OWOut B0,1,Main,[ReadRom]`. `OWOut` is invoked with three options:

```
Owout Pin,Mode,{NCLabel},[{Mods} Exp]
```

`Pin`—is the pin to which the data line is connected. `Pin` can be either a constant or a variable.

`Mode`—defines the specifics of the data transfer. Permissible `mode` values are:

Mode	Send Reset?	Data Mode	Speed
0	No	Byte	Low
1	Before Data	Byte	Low
2	After Data	Byte	Low
3	Before and After Data	Byte	Low
4	No	Bit	Low
5	Before Data	Bit	Low

(Dallas Semiconductor is expanding its 1-wire product line to include new devices with bit rates of several hundred kb/s and has a proposed an even higher 1 Mb/s rate. MBasic currently supports only the initial 14 kb/s speed 1-wire protocol.)

`NCLabel`—an optional address label to which execution will jump if no 1-wire chip is detected.

`Mods`—MBasic's standard command modifiers, for example, `Str`, `Real`, and so on.

`Exp`—is a variable or constant holding the value to be sent.

Figure 22.1 shows `B0` as the data pin. Commands written to a 1-wire device require a leading reset pulse, we'll use mode 1. For simplicity, we'll omit device connection error checking.

We'll look at 1-wire commands in more detail in connection with later programs, but for now we'll note that $33 is the command value for "read ROM," that is, send the serial number of the device. We declare a constant, `ReadRom`, and set its value to $33. Hence, our command is:

```
OWOut B0,1,Main,[ReadRom]
```

We now read the response from the 1-wire device with `OWIn` and write the output to the serial port, one byte at a time:

```
For i = 0 to 7
        OWIn B0,0,[Temp]
        HSerOut [Hex Temp," "]
Next
```

`OWin` follows the same syntax as `OWOut`:

```
OWIn Pin,Mode,{NCLabel},[{Mods} Var]
```

`Pin`, `Mode` and `NCLabel` are identical with their corresponding elements in `OWOut`, and require no further description. Since we are reading a value, of course, it must be read into a variable, not a constant.

Dallas Semiconductor's 1-wire specifications define device serial numbers as 8 bytes (64 bits) long, configured as:

8-bit CRC Code		48-bit Serial Number		8-bit Family Code	
MSB	LSB	MSB	LSB	MSB	LSB

The family code for 18B20 digital thermometer chips is $28. The CRC (cyclic redundancy code) is an error-checking feature, so that, should we desire, we may verify that the 56 bits of the family code and serial number have been correctly received and were not corrupted. We'll not further consider how CRCs are calculated, as it's a topic well beyond the level of this introductory book.

When we run the program, we see the following repetitive output:

```
28 4C D4 3E 0 0 0 D6
28 4C D4 3E 0 0 0 D6
28 4C D4 3E 0 0 0 D6
...
```

The digits 4C D4 3E 0 0 0 D6 are, of course, dependent upon the particular DS18B20 chip. I plugged in a second DS18B20 chip and found its serial number:

```
28 FE DA 3E 0 0 0 C1
28 FE DA 3E 0 0 0 C1
28 FE DA 3E 0 0 0 C1
...
```

In the output, $28 is the family number and $D6 (or $C1 in the second example) is the CRC. The center six bytes represent the serial number of the chip. But, there's a difference between the result and the serial number specification isn't there? The definition has the CRC sent first and the family code sent last. Yet, Program 22.1 displays the family code first, and the CRC last. The explanation is that 1-wire devices store the least significant byte at the lower address and the most significant byte at the higher address. Bytes are transmitted and received from lowest address to highest address. Hence, the net effect is the send/receive byte order is reversed from the data sheet description. This is more confusing to describe than to use; when we wish to address a particular device, we just repeat the byte order we read with Program 22.1.

22.1.3 Reading the Temperature

Program 22.1 will confirm that your circuit is properly wired up and the DS18B20 is functioning. Program 22.2 shows how we can read the temperature. (Remember the | symbol is a line continuation.)

```
;Reads a Dallas 1-wire Temperature sensor
;DS18B20. Assumes only ONE device is connected to the bus,
;since we use global address mode.

;Variables
;----------------------
Temp            Var     Word    ;holds raw binary temperature output
PlusFlag        Var     Byte    ;plus/minus flag
Centigrade      Var     Float   ;Floating point C temp
Fahrenheit      Var     Float   ;Floating point F temp
TH              Var     Byte    ;Lower alarm value
TL              Var     Byte    ;Upper alarm value
ConfigReg       Var     Byte    ;Temperature resolution

;Initialization
Init
;--------------
      EnableHSerial
      SetHSerial H19200

      ;$4E Write to RAM; dummy $FF to TH & TL
      ;$7F to ctrl reg for 12-bit
      OWOut B0,1,Init,[$CC,$4E,$FF,$FF,$7F]
      OWOut B0,1,Init,[$CC,$48]
      OWOut B0,1,Init,[$CC,$B8]

      ;Read scratchpad memory
      OWOut B0,1,Init,[$CC,$BE]
      Pause 1000
      ;Read output and check configuration
      OWin B0,0,[Temp.Byte0,Temp.Byte1,TH,TL,ConfigReg]
      HSerOut ["Init OK R1: ",BIN ConfigReg.Bit6 ," R0: " |
      ,BIN ConfigReg.Bit5,13]

Main
      ;Cause temp conversion to start
      OWOut B0,1,Main,[$CC,$44]

Wait
      ;read output and loop until conversion is done.
      ;conversion finished if writeback of 1, 0 if not done
      ;Note this only works with external power
```

Program 22.2

```
      OWIn B0,0,[Temp]
      If Temp = 0 Then Wait
      OWOut B0,1,Main,[$CC,$BE]
      OWin B0,0,[Temp.Byte0,Temp.Byte1,TH,TL,ConfigReg]
      ;Read TH,TL and ConfigReg, but don't use
      ;Temperature data returned as 2's complement for below 0.
      ;Hence, <0 is detected by a 1 in the returned highest bit.
      PlusFlag = 1
      If Temp.HighBit = 1 Then      ;check for < 0C
         ;Subtract from 0 to read below 0 value
         Temp = $0000-Temp
         PlusFlag = 0
      EndIf

      ;Returned in Centigrade steps of 0.0625 Deg C.
      Centigrade = (ToFloat Temp) * 0.0625
      If PlusFlag = 0 Then          ;below zero, multiply by -1
            Centigrade = -Centigrade
      EndIf

      ;Standard conversion formula to get to F from C.
      Fahrenheit = (Centigrade * 1.8) + 32.0 ;deg F

      ;Write output. Note the \2 and \1 modifiers give number of
      ;places after decimal point
      HSerOut ["Temperature ",Real Fahrenheit \2," F " ,Real|
      Centigrade \1," C", 13]

      Pause 2000
GoTo Main
```

Program 22.2: Continued

Program 22.2 initializes the DS18B20 to 12-bit resolution mode, and then every 2 seconds reads the temperature and writes the value in Fahrenheit to the serial port.

Program 22.2 uses global addressing and will work only if there is one 1-wire device connected to the data bus.

After declaring the necessary variables, we initialize the DS18B20.

The DS18B20 has variable resolution, and can be programmed for 9, 10, 11 or 12-bit resolution. (The DS18S20 is a variant with only 9-bit resolution.) The trade-off for increased resolution is longer conversion time.

Resolution	Conversion Time	Resolution		R1	R0
9-bit	93.75 ms	0.5°C	0.900°F	0	0
10-bit	187.5 ms	0.25°C	0.450°F	0	1
11-bit	375 ms	0.125°C	0.225°F	1	0
12-bit	750 ms	0.0625°C	0.113°F	1	1

We'll use 12-bit resolution, the default configuration for DS18B20 devices. Two option bits, R1 and R0, in the configuration register control resolution, so we'll set both to 1 to establish 12-bit resolution. We should, of course, not confuse resolution and accuracy. Regardless of the resolution selected, the DS18B20's accuracy remains at ±0.5ºC over the range −10ºC through +85ºC.

Configuration Register							
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	R1	R0	1	1	1	1	1

The configuration word to command 12-bit resolution is thus `%01111111`, or `$7F`. How, then, to write `$7F` to the configuration register?

The DS18B20's memory is organized as nine bytes of RAM and three bytes EEPROM. The configuration register is byte no. 4:

Address	Scratchpad Memory Contents		EEPROM
0	Temperature LSB (read only)		
1	Temperature MSB (read only)		
2	TH Register	← →	TH
3	TL Register	← →	TL
4	Configuration Register	← →	Configuration
5	Reserved ($FF)		
6	Reserved ($0C)		
7	Reserved ($10)		
8	CRC		

Our ability to selectively write to the scratchpad and EEPROM is limited. In fact, we have but four memory commands, plus two function commands:

DS18B20 Command Values			
Command	**Description**	**Command Value**	**1-Wire Bus Activity**
Convert Temperature	Initiate temperature reading	$44	Transmit conversion status (only with external power)
Read Scratchpad	Read all 9 scratchpad locations	$BE	DS18B20 transmits up to 9 bytes to master
Write Scratchpad	Write all 3 scratchpad locations, TH, TL and Configuration Register	$4E	Master transmits TH, TL and Configuration to DS18B20

(Continued)

DS18B20 Command Values			
Command	**Description**	**Command Value**	**1-Wire Bus Activity**
Copy Scratchpad	Copies TH, TL and Configuration from the scratchpad to EEPROM	$48	None
Recall EEPROM	Copies TH, TL and Configuration from EEPROM to the scratchpad	$B8	DS18B20 transmits recall status to master
Read Power Supply	Informs master whether 18B20 is directly or parasitically powered	$B4	DS18B20 transmits power supply status to master

To write the resolution information to the configuration register and then to the EEPROM, we must write 3 bytes, TH, TL and the configuration byte. TH and TL are alarm trigger registers and store values for high (TH) and low (TL) temperature alarms. We shall not use either, so we will write a dummy value, $FF to both TH and TL.

One final bit of housekeeping and then we'll write our configuration bits. Before a 1-wire device may receive any commands, it must be addressed. We'll continue In Program 22.2 with global addressing, `$CC`. All 1-wire devices respond to a `$CC` address, so to avoid device conflict we may have only one 1-wire device connected to pin B0.

```
OWOut B0,1,Init,[$CC,$4E,$FF,$FF,$7F]
OWOut B0,1,Init,[$CC,$48]
OWOut B0,1,Init,[$CC,$B8]
OWOut B0,1,Init,[$CC,$BE]
Pause 1000
OWin B0,0,[Temp.Byte0,Temp.Byte1,TH,TL,ConfigReg]
HSerOut ["Init OK R1: ",BIN ConfigReg.Bit6 ," R0: " |
,BIN ConfigReg.Bit5,13]
```

The first four lines of code implement our strategy; we write `$FF,$FF,$7F` to the scratchpad with the `$4E` command. We then move scratchpad memory locations 2, 3 and 4 to EEPROM with a `$48` command. For our demonstration, we then copy the EEPROM back to scratchpad memory with a `$B8` command and, finally, read back the contents of the scratchpad with a `$BE` command. Note that in every case, we issue a reset and a global address (`$CC`) before the command byte.

We then read back the bytes sent after the `$BE` command with the `OWIn` procedure and write the values to the serial port. The response we see from the serial port should be:

```
Init OK R1: 1 R0: 1
```

Now, we initiate a temperature read.

```
Main
      ;Cause temp conversion to start
      OWOut B0,1,Main,[$CC,$44]
```

```
Wait
      ;read output and loop until conversion is done.
      ;conversion finished if writeback of 1, 0 if not done
      ;Note this only works with external power
      OWIn B0,0,[Temp]
      If Temp = 0 Then Wait
```

Temperature reads must start with the "convert temperature" `$44` command. Upon receipt of the $44, the DS18B20 starts the temperature reading process and upon completion stores the value in the scratchpad at locations 0 (least significant byte) and 1 (most significant byte). If externally powered, the DS18B20 will respond during the conversion process with a 0, changing to a 1 when completed. Hence, after initiating the convert temperature command, we keep reading the DS18B20 until it returns a 1, indicating temperature data is ready to be read.

We now are ready to read the temperature from the DS18B20's scratchpad memory. The data is returned in increasing address order, from 0 (least significant byte of temperature) to 4 (configuration register).

```
OWOut B0,1,Main,[$CC,$BE]
OWin B0,0,[Temp.Byte0,Temp.Byte1,TH,TL,ConfigReg]
;Read TH,TL and ConfigReg, but don't use
```

This is the same code we used when reading the configuration register during initialization. Now, however, we discard all read data, save for `Temp.Byte0` and `Temp.Byte1`, as these contain our temperature reading.

The temperature data is returned in the following format:

Temp.Byte0	**Bit 7**	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
Least Significant Byte	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
Temp.Byte1	**Bit 15**	**Bit 14**	**Bit 13**	**Bit 12**	**Bit 11**	**Bit 10**	**Bit 9**	**Bit 8**
Most Significant Byte	S	S	S	S	S	2^6	2^5	2^4

A 1 at bit 0, for example, represents 1 × 2–4 degrees C, or 0.0625ºC.

Since the range of temperatures reported by the DS18B20 extends below 0ºC, the data includes a sign bit, identified as "S" in the table. If the temperature is above 0, S is 0; if below 0ºC, S is 1. For temperatures below 0, we must subtract the reading from $0000 to obtain the number of degrees below 0ºC. (The data is stored as "a 16-bit sign-extended two's complement number." Don't let this scare you; it's simple to make it useful.)

Here's how values around 0ºC are reported by the DS18B20:

Temperature	Binary Output	Hexadecimal Output
+10.125°C	**0000 0000 1010 0010**	**$00A2**
+0.5°C	**0000 0000 0000 1000**	**$0008**
0°C	**0000 0000 0000 0000**	**$0000**
−0.5°C	**1111 1111 1111 1000**	**$FFF8**
−10.125°C	**1111 1111 0101 1110**	**$FF5E**

We read the two temperature bytes one at a time, storing them in the word variable `Temp`. After reading `Temp`, we check to see if the returned value is less than 0ºC, as indicated by a 1 in at bit 15:

```
PlusFlag = 1
       If Temp.HighBit = 1 Then ;check for < 0C
            ;Subtract from 0 to read below 0 value
            Temp = $0000-Temp
            PlusFlag = 0
       EndIf
```

If a 1 is found, then we remove the two's complement by subtracting the value from $0000. This gives the number of degrees below 0ºC. (If you're not convinced, subtract it by hand, or use Window's accessory calculator in scientific view. For −10.125ºC, $0000 – $FF5E = $00A2. Decimal ($A2) = 162. Each step is 0.0625ºC, so 162 * 0.0625 = 10.125.)

```
       ;Returned in Centigrade steps of 0.0625 Deg C.
       Centigrade = (ToFloat Temp) * 0.0625
       If PlusFlag = 0 Then          ;below zero, multiply by -1
              Centigrade = -Centigrade
       EndIf

       ;Standard conversion formula to get to F from C.
       Fahrenheit = (Centigrade * 1.8) + 32.0 ;deg F
```

We now convert the returned raw temperature word (after any necessary below zero correction) to a floating point Celsius (Centigrade) value by multiplying it by the step size, 2^{-4}, or 0.0625 degrees C/step. (2^{-4} is the same as $1/2^4$, or 1/16, which is 0.0625.) Since `Temp` is an integer, we first convert it to a floating-point value with the `ToFloat` operator. We then multiply by the step size. (If you run the DS18B20 at less than 12-bit resolution, the output jumps in larger intervals, but each step remains 0.0625ºC.)

If the returned value was less than 0ºC, PlusFlag has been set to 0, so we must multiply the temperature by −1, which is accomplished by prefixing Centigrade with a minus sign.

Finally, we convert the floating point Celsius temperature to Fahrenheit.

```
;Write output. Note the \2 and \1 modifiers give number of
;places after decimal point
HSerOut ["Temperature ",Real Fahrenheit \2," F " ,Real|
Centigrade \1," C", 13]
```

Now we write the temperature (in Celsius and Fahrenheit) to the serial port. Note our use of the undocumented \2 and \1 modifiers to display two and one digits following the decimal point. Here's a sample output from Program 22.2:

```
Init OK R1: 1 R0: 1
Temperature 74.63 F 23.6 C
Temperature 74.63 F 23.6 C
Temperature 74.63 F 23.6 C
Temperature 74.63 F 23.6 C
Temperature 73.62 F 23.1 C
```

22.2 Reading Multiple Sensors on the Same Bus

A primary advantage of the 1-wire bus is that we may place many sensors on it, and selectively read each one. Let's try something simple, one indoor temperature sensor and one outdoor sensor, both connected to B0, as shown in Fig. 22.5.

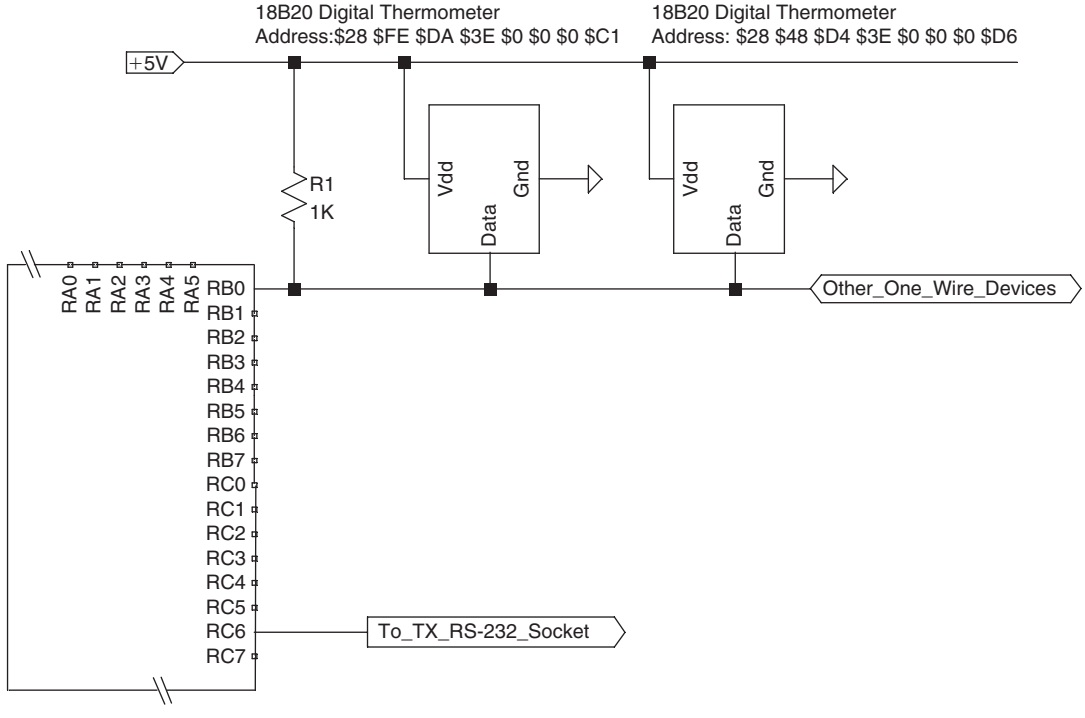


Figure 22.5: Multiple DS18B20 Temperature Sensors on One Bus

Figure 22.6: Temporary Outdoor Temperature Sensor Mounting

First, though, run Program 22.1 for each DS18B20 and note their serial numbers. The two sensors I had on hand identified themselves as:

```
28 4C D4 3E 0 0 0 D6
28 FE DA 3E 0 0 0 C1
```

I made a simple temporary outdoor sensor mount from a short length of PVC pipe and fittings, as shown in Fig. 22.6. The sensor is at the top of the "T" fitting and I soldered its three pins to three conductors in a short length of CAT5 data cable. I stuffed the cable opening of the "T" with a few plastic shipping pellets for weatherproofing and brought the free end of the cable into my workshop. (Reference [7] discusses cable selection and distance issues and should be consulted for cable runs more than a couple dozen feet.) The other sensor I mounted on the breadboard for an indoor comparison reading.

Program 22.3 selectively reads the indoor and outdoor sensors and writes their readings to the serial port.

```
;Program 22.3 Read 2 DS18B20 temperature
;sensors connected to the same bus.
;Write the two temperatures to serial output
;The sensors are addressed by SN

;Variables
;----------------------
Temp         Var     Word     ;holds raw binary temperature output
PlusFlag     Var     Byte     ;plus/minus flag
```

Program 22.3

```
Centigrade   Var     Float    ;Floating point C temp
Fahrenheit   Var     Float    ;Floating point F temp
TH           Var     Byte     ;Lower alarm value
TL           Var     Byte     ;Upper alarm value
ConfigReg    Var     Byte     ;Temperature resolution

; The full ROM SN of the two DS18B20 Sensor Chips
;Chip 1 $28 $FE $DA $3E $0 $0 $0 $C1
;Chip 2 $28 $48 $D4 $3E $0 $0 $0 $D6
;your sensors will have different serial numbers!

;Initialization
;----------------
      EnableHSerial
      SetHSerial H19200

;Initialize both sensors to 12-bit resolution.
;Since we have multiple sensors on same bus, we must use
;selective addressing $55 followed by 8-byte chip address
;Otherwise, routine is same as Program 22.2.
Init1
;----------------
      OWOut B0,1,Init1,[$55,$28,$FE,$DA, |
      $3E,$0,$0,$0,$C1,$4E,$FF,$FF,$7F]
      OWOut B0,1,Init1,[$55,$28,$FE,$DA,$3E,$0,$0,$0,$C1,$48]
      OWOut B0,1,Init1,[$55,$28,$FE,$DA,$3E,$0,$0,$0,$C1,$B8]
      OWOut B0,1,Init1,[$55,$28,$FE,$DA,$3E,$0,$0,$0,$C1,$BE]
      Pause 1000
      ;Read output and check configuration
      OWin B0,0,[Temp.Byte0,Temp.Byte1,TH,TL,ConfigReg]
      HSerOut ["Init 1 OK R1: ",BIN ConfigReg.Bit6 , |
      " R0: ",BIN ConfigReg.Bit5,13]

Init2
;----------------
      OWOut B0,1,Init2,[$55,$28,$48,$D4, |
      $3E,$0,$0,$0,$D6,$4E,$FF,$FF,$7F]
      OWOut B0,1,Init2,[$55,$28,$48,$D4,$3E,$0,$0,$0,$D6,$48]
      OWOut B0,1,Init2,[$55,$28,$48,$D4,$3E,$0,$0,$0,$D6,$B8]
      OWOut B0,1,Init2,[$55,$28,$48,$D4,$3E,$0,$0,$0,$D6,$BE]
      Pause 1000
      ;Read output and check configuration
      OWin B0,0,[Temp.Byte0,Temp.Byte1,TH,TL,ConfigReg]
            HSerOut ["Init 2 OK R1: ",BIN ConfigReg.Bit6 , |
      " R0 ",BIN ConfigReg.Bit5,13]
```

Program 22.3: Continued

```
Main
      ;Selective start of conversion on Chip No. 1
      ;Again we start with $55,ChipAddress
      OWOut B0,1,Main,[$55,$28,$FE,$DA,$3E,$0,$0,$0,$C1,$44]

Wait
      ;Wait for good data
      OWIn B0,0,[Temp]
      If Temp = 0 Then Wait
      ;Selective read of data on Chip No. 1
      OWOut B0,1,Main,[$55,$28,$FE,$DA,$3E,$0,$0,$0,$C1,$BE]
      OWin B0,0,[Temp.Byte0,Temp.Byte1]

      ;From raw data, compute Centigrade and Fahrenheit
      GoSub ComputeTemp

      HSerOut ["Sensor 1 "]
      ;Write the Degrees F and Degrees C to the serial port
      GoSub WriteTemp

;------------------------ Sensor No. 2 ----------------------
;Now we repeat but with the Sensor No. 2 Selective Address

Part2
      OWOut B0,1,Part2,[$55,$28,$48,$D4,$3E,$0,$0,$0,$D6,$44]

Wait2
      OWIn B0,0,[Temp]
      If Temp = 0 Then Wait2

      OWOut B0,1,Part2,[$55,$28,$48,$D4,$3E,$0,$0,$0,$D6,$BE]
      OWin B0,0,[Temp.Byte0,Temp.Byte1]

      GoSub ComputeTemp

      HSerOut [$9,"Sensor 2 "]
      GoSub WriteTemp
      HSerOut [13]

      Pause 1000
GoTo Main

ComputeTemp   ;Subroutine accepts raw data and outputs Deg F and Deg C
;------------

      ;Raw data is in 2's complement if less than 0C.
      ;Check high bit for < 0
```

Program 22.3: Continued

```
          PlusFlag = 1
          If Temp.HighBit = 1 Then    ;check for < 0C
             Temp = $0000-Temp    ;Just subtract to convert
             PlusFlag = 0
          EndIf
          Centigrade = (ToFloat Temp) * 0.0625 ;step 0.0625 deg C
          If PlusFlag = 0 Then                          ;below zero
                Centigrade = fneg Centigrade; If < 0 add minus sign
          EndIf
          ;Standard conversion from Centigrade to Fahrenheit
          Fahrenheit = (Centigrade * 1.8) + 32.0
                ;Centigrade to deg F
Return
   WriteTemp
   ;-------------
          ;Write generic output to serial
          ;\2 and \1 fix the number of digits after .
          ;that are displayed
          HSerOut [Real Fahrenheit \2," F ",Real Centigrade \1," C"]
   Return
```

Program 22.3: Continued

We'll concentrate on the changed parts of this program, as it largely duplicates what we learned in Program 22.2.

Since we have two DS18B20 sensors on the same bus, we must selectively address each one, instead of using the `$CC` universal address. We do this simply by replacing `$CC` with `$55`, followed by the 8-byte serial number of the device. Hence, to initialize the configuration register we have:

Universal address version:

```
OWOut B0,1,Init,[$CC,$4E,$FF,$FF,$7F]
```

Selective address version for DS18B20 with SN $28 $FE $DA $3E $00 $00 $00 $C1:

```
OWOut B0,1,Init1,[$55,$28,$FE,$DA,$3E,$0,$0,$0, |
$C1,$4E,$FF,$FF,$7F]
```

(Remember the | is the line continuation symbol.)

Since we have two devices to initialize, we repeat the selective initialization twice, once for each device.

The main program loop likewise selectively addresses one sensor, calls the subroutine `ComputeTemp` to convert the raw temperature word to Centigrade and Fahrenheit values, and then calls a second subroutine, `WriteTemp`, to write the Centigrade and Fahrenheit values to the serial output port. (The `$9` character is a horizontal tab, used to space the results across the terminal screen.)

The two subroutines, `ComputeTemp` and `WriteTemp` track the embedded code in Program 22.2.

Here's the output of Program 22.3, taken before I assembled the outdoor sensor and moved it outside:

```
Init 1 OK R1: 1 R0: 1
Init 2 OK R1: 1 R0: 1
Sensor 1 74.74 F23.7 C      Sensor 2 74.74 F 23.7 C
Sensor 1 74.74 F 23.7 C     Sensor 2 74.74 F 23.7 C
Sensor 1 73.84 F 23.2 C     Sensor 2 74.63 F 23.6 C
Sensor 1 74.97 F 23.8 C     Sensor 2 74.63 F 23.6 C
Sensor 1 80.26 F 26.8 C     Sensor 2 74.86 F 23.8 C  ← Finger on Sensor 1
Sensor 1 78.46 F 25.8 C     Sensor 2 74.86 F 23.8 C
Sensor 1 77.22 F 25.1 C     Sensor 2 79.02 F 26.1 C  ← Finger on Sensor 2
Sensor 1 76.32 F 24.6 C     Sensor 2 82.51 F 28.0 C
Sensor 1 75.76 F 24.3 C     Sensor 2 80.37 F 26.8 C
Sensor 1 75.64 F 24.2 C     Sensor 2 78.57 F 25.8 C
```

To see how fast the sensors react, I grasped each one between two fingers for a few seconds. (The measurement cycle is about 4 seconds per reading.) It took only one read cycle to see the temperature start to rise, and about five or six cycles to return to ambient.

22.3 DS1302 Real-Time Clock

Now that we've gotten the temperature out of the way, let's look at timekeeping. We could use our PIC as a clock, such as described in Reference [9]. But, we're going to unload the timekeeping function onto a dedicated special purpose chip from Dallas Semiconductor, the DS1302.

The DS1302 is a real time clock, requiring an external 32.678 KHz crystal. The DS1302 provides seconds, minutes, hours, day-of the-week, date, month and year information, including leap year adjustment up to 2100. It also supports battery or capacitor backup, and includes an integrated trickle charger. The DS1302 communicates with the PIC via a three-wire serial connection, supported by MBasic's `ShiftIn` and `ShiftOut` procedures.

We'll connect the DS1302 to our PIC using the circuit shown in Fig. 22.7. A couple points in Fig. 22.7 deserve amplification. First, we have chosen a very large capacitor, 0.47 F, to serve as our power supply backup. Once fully charged (it takes a little over 1 hour), the 0.47 F capacitor will keep the DS1302 running at least a day if the primary power is interrupted. Second, the DS1302's accuracy is dependent upon the accuracy of the 32.678 KHz crystal. The crystal's design capacitance should match the 6 pF specification of the DS1302. (If we desire precision timekeeping, there are many more things that will concern us, but those are beyond the scope of this chapter. See, for example, Reference [10].)

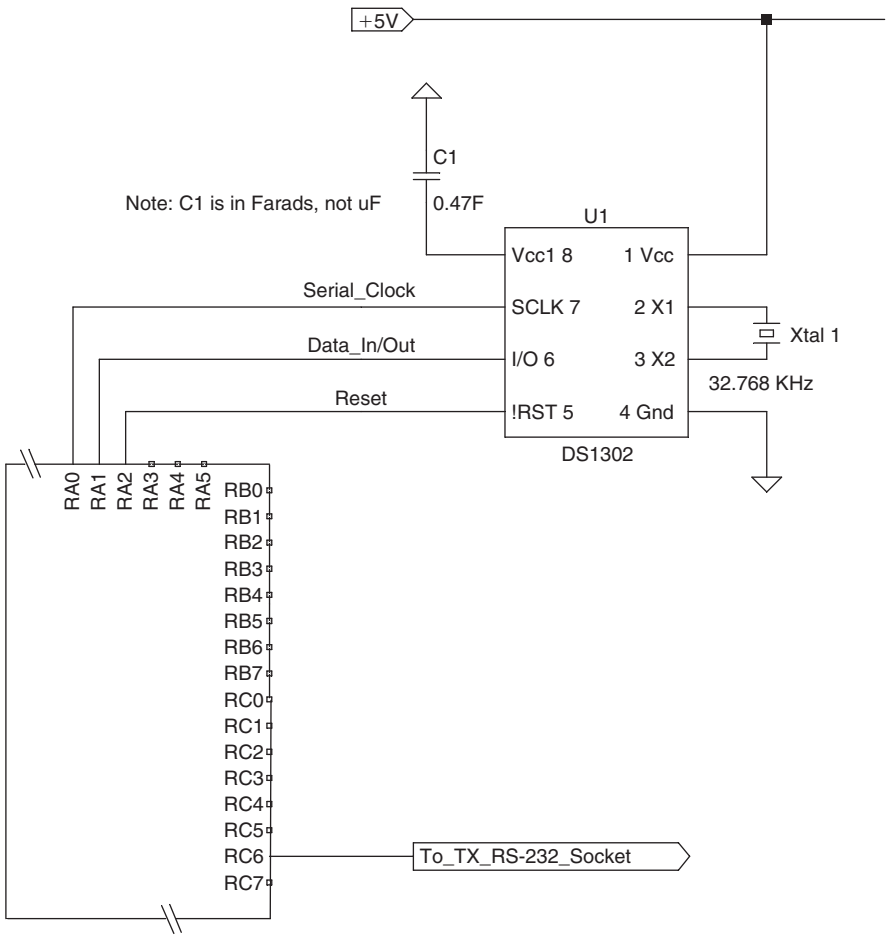


Figure 22.7: Connecting a DS1302 Real-Time Clock

The DS1302 has three connections to the PIC:

SCLK (Serial Clock Input)—The clock determines when data bits are received and valid. Data is transferred to the DS1302 or read from the DS1302 when the clock changes state. Data is received by the DS1302 during the rising edge. Data is transmitted by the DS1302 on the falling edge of the clock, but are read by the PIC master on the rising edge. All clock pulses are generated by the PIC; the DS1302 only receives clock pulses and cannot generate them.

I/O (Data)—The data line establishes whether a 1 or a 0 is read at the appropriate clock edge. The data line is bi-directional; the DS1302 may send to the PIC or the PIC may send to the DS1302.

RST (Reset)—The reset line serves two purposes. First, when taken high it turns on the DS1302's control logic. Second, when taken from high to low, it terminates data transfer

from the PIC to the DS1302. The PIC controls the reset line. Figure 22.8 shows a sample data exchange between a PIC and a DS1302. A total of 5 bytes are exchanged. Figure 22.9 more clearly illustrate the relationship between the clock and data lines.

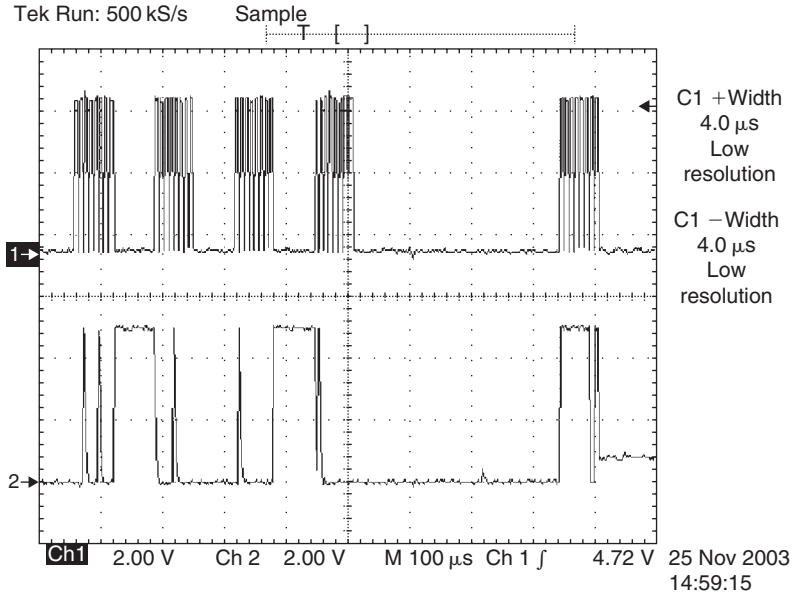


Figure 22.8: Data Transfer DS1302 Ch1: Clock; Ch2: Data

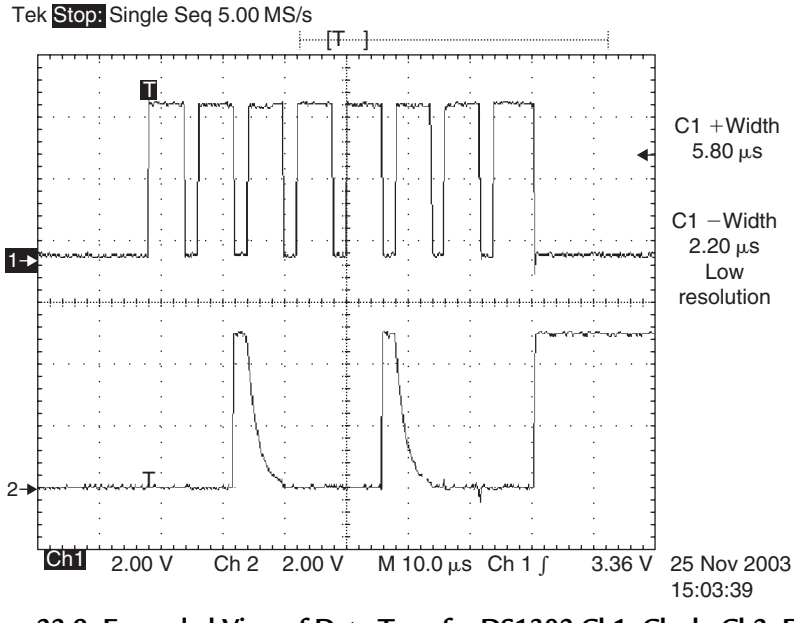


Figure 22.9: Expanded View of Data Transfer DS1302 Ch1: Clock, Ch2: Data

Fortunately, MBasic hides the details of data transfer to and from the DS1302.

Program 22.4 reads the DS1302 and writes the current time, once every second, to the serial port.

```
;Program 22.4. Reads a Dallas/Maxim DS1302
;Real time clock.
;Also writes the current time to the clock

;Variables
;---------------
RTCCmd          Var         Byte              ;Command byte
Temp            Var         Byte              ;Temporary variable
i               Var         Byte              ;counter
TimeData        Var         Byte(7)
OldSeconds      Var         Byte              ;last read seconds

;For convenience we alias the time array
Seconds         Var         TimeData(0)       ;Seconds
Minutes         Var         TimeData(1)       ;Minutes
Hours           Var         TimeData(2)       ;Hours
Date            Var         TimeData(3)       ;Day of Month
Month           Var         TimeData(4)       ;Month
Day             Var         TimeData(5)       ;Day of Week
Year            Var         TimeData(6)       ;Year

;Constants
;--------------             () are DIP8 numbers
Clk             Con         A0              ;Clock pin    (7)
Dta             Con         A1              ;Data pin     (6)
Reset           Con         A2              ;Reset pin    (5)

CtrlReg         Con         %00111

;Initial load of Sec   Min   Hrs   Date   MO    Day    YR
PreSet ByteTable $00,  $53,  $12,  $24,   $11,  $02,   $03
;These are in BCD, hence November (Month 11)
;is entered as $11, not 11.

;Command to select charger settings
Charger         Con         %01000
;1 diode, 2K charging resistance. See
;Figure 5 of DS1320 data sheet
Battery         Con         %10100101

;For Day and Month names, use the following. Since
;Sun=1 and Jan=1 we use nul at the beginning instead of
;subtracting 1.
DayName ByteTable "NulSunMonTueWedThuFriSatSun"
MoName ByteTable "NulJanFebMarAprMayJunJulAugSepOctNovDec"
```

Program 22.4

```
    ;Initialization
    ;--------------
          EnableHSerial
          SetHSerial H19200

          ;To set the clock we first disable write protection.
          ;Then send the clock set information, and then restore
          ;write protection.

          ;Disable DS1302 WriteProtection by 0 in Bit7
          Temp = $00
          RTCCmd = CtrlReg
          GoSub PreSetData

          Set the date and time from the Preset byte table
          For i = 0 to 6
                  Temp = Preset(i)
                  RTCCmd = i
                  GoSub PreSetData
          Next

          ;Since we have a Super Cap backup, we want charger ON
          Temp = Battery
          RTCCmd = Charger
          GoSub PreSetData

          ;DS1302 Write protection back on
          Temp = $80
          RTCCmd = CtrlReg
          GoSub PreSetData

    Loop
    ;---------
          ;Read the clock output. If seconds changed, write time
          GoSub ReadData
          ;Write in form:
          ;13:10:45 11/24/03 Mon 24-Nov-2003 to the serial port
          ;New seconds, so write output
          If OldSeconds <> Seconds Then
          HSerOut [Dec2 (BCD2BIN Hours)\2,":",Dec2 |
          (BCD2BIN Minutes)\2,":",Dec2 (BCD2BIN Seconds)\2]
          HSerOut [" ",Dec2 (BCD2BIN Month)\2,"/",Dec2 |
          (BCD2BIN Date)\2,"/",Dec2 (BCD2BIN Year)\2]
          HSerOut [" ",Str DayName(Day*3)\3," "]
          HSerOut [Dec2 (BCD2BIN Date)\2,"-",Str |
          MoName(3*BCD2BIN Month)\3,"-20",Dec2 (BCD2BIN Year)\2,13]
      EndIf
GoTo Loop
```

Program 22.4: Continued

```
PreSetData              ;Routine to upload data to the DS1302
;----------
      High Reset
      ;The %0\1,RTCCmd\5,%10\2 results in the equivelent of
      ;sending 10 RTCCMD 0. Each element is sent in element order
      ;but bit reversed, as required by the DS1320.
      ShiftOut Dta,Clk,LSBFIRST,[%0\1,RTCCmd\5,%10\2,Temp\8]
      Low Reset
Return

ReadData
;---------
      ;Reset pin must be high for read
      High Reset
      ;%10111111 is BURST SEND REGISTER command, so get 8 bytes
      ;the last byte is the control register, which we don't use
      ShiftOut Dta,Clk,LSBFirst,[%10111111\8]
      OldSeconds = Seconds
      ;Now actually read the burst send data.
      ShiftIn Dta,Clk,LSBPRE,[Seconds\8,Minutes\8,Hours\8, |
      Date\8,Month\8,Day\8,Year\8]
      Low Reset
Return
```

Program 22.4: Continued

We've set up a seven-element data structure to hold time and date information from the DS1302:

```
TimeData        Var          Byte(7)
;For convenience we alias the time array
Seconds         Var          TimeData(0)        ;Seconds
Minutes         Var          TimeData(1)        ;Minutes
Hours           Var          TimeData(2)        ;Hours
Date            Var          TimeData(3)        ;Day of Month
Month           Var          TimeData(4)        ;Month
Day             Var          TimeData(5)        ;Day of Week
Year            Var          TimeData(6)        ;Year
```

Clock and calendar data is held in the DS1302 in a 10-byte register structure:

Register Address	Contents
0	Seconds
1	Minutes
2	Hours
3	Date (Day of Month)
4	Month
5	Day of Week

Register Address	Contents
6	Year
7	Control Register
8	Trickle Charger Control
9	Clock Burst Register

In addition, the DS1302 has 31 bytes of general-purpose RAM, at memory addresses 0…31

The index values of data array `TimeData` align with the DS1302's memory addresses. We've also aliased "user friendly" functional names for each element of `TimeData`.

```
;Constants
;--------------          () are DIP8 numbers
Clk          Con          A0        ;Clock pin  (7)
Dta          Con          A1        ;Data pin   (6)
Reset        Con          A2        ;Reset pin  (5)
```

As usual, we define the three connection pins by declaring three custom constants, `Clk, Dta` and `Reset`.

```
;Initial load of Sec  Min  Hrs  Date MO   Day  YR
PreSet ByteTable $00, $53, $12, $24, $11, $02, $03
;These are in BCD, hence November (Month 11)
;is entered as $11, not 11.
```

At the first power-up—and whenever power is removed without the backup "super cap" in place—we must initialize the DS1302 with the current date, time and day of week. For clarity, we'll store the initialization inf ormation in a byte table.

The seven date, time and day of week values are stored in the DS1302 in binary coded decimal format. As a reminder, each nibble in a BCD byte is regarded as a separate digit, ranging from 0…9. Hence, the BCD representation of the number 28 (decimal) is $28 (decimal 40). The day of the week entry starts with Sunday = 1. For convenience in setting the clock, we'll store the byte table in the same order it will be sent to the DS1302—that is, in the DS1302's internal address order. Accordingly, 12:53:00, Monday, 24 November 2003 is stored in our byte table as `$00, $53, $12, $24, $11, $02, $03`.

```
;Command to select charger settings
Charger      Con         %01000
;1 diode, 2K charging resistance. See
;Figure 5 of DS1320 data sheet
Battery      Con         %10100101
```

Since we provide backup power, it's necessary to enable the internal charger circuitry in the DS1302, as the default status is disabled.

This is an appropriate time to look at how we exchange data with the DS1302. It's a straightforward two-step process:

- Send a command byte to the DS1302
- Send one or more bytes of data to the DS1302, or receive one or more bytes of data from the DS1302, if the command byte calls for reading data.

The DS1302's command byte structure is quite logical:

<table>
<tr><th>Bit 7</th><th>Bit 6</th><th>Bit 5</th><th>Bit 4</th><th>Bit 3</th><th>Bit 2</th><th>Bit 1</th><th>Bit 0</th></tr>
<tr><td>0 = Write Disabled</td><td>0 = Register</td><td colspan="5" rowspan="2">5-bit representation of Register or RAM Address</td><td>0 = Write</td></tr>
<tr><td>1 = Write Enable</td><td>1 = RAM</td><td>1 = Read</td></tr>
</table>

The 5-bit representations of the register or RAM address means that the leading three zeros are truncated from the normal 8-bit binary representation. For example, the normal byte-length binary representation of the digit 8 is %00001000. The 5-bit representation is %01000.

Bit 6 of the command byte determines whether we are addressing the DS1302's registers or its general-purpose RAM. (All of our communications will be with the registers, and we won't use the DS1302's RAM at all. Hence, Bit 6 will always be 0 in this chapter's programs.)

To set the value of the day of the month to the day-of-the-month register we send two bytes to the DS1302; first the command byte, followed by the value to be loaded into the DS1302. Suppose we are to set the day of the month to the 28th. The BCD representation of the 28th is $28. Let's determine the command byte:

- Bit 7 must be `%1`, since we are writing data to the DS1302
- Bit 6 must be `%0`, since we are writing data to a register, not RAM.
- Bits 5…1 are the 5-bit representation of the address of the day of the month register. The day of the month register is number 3, so the 5-bit representation is `%00011`.
- Bit 0 must be %0, since we are writing data to the DS1302.

Hence, our command byte is:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	0	0	0	0	1	1	0

This binary value corresponds to `$86`. Hence, to set the day-of-the-month to the 28th, we must send the two byte sequence `$86 $28` to the DS1302.

One more housekeeping detail before we set the clock. Bit 7 of the control register is a master write protect bit. It must be set to 1 before any data may be written to either the DS1302's registers or RAM.

DS1302 Control Register							
Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
0 = Master Write Enable 1 = Master Write Disable	0	0	0	0	0	0	0

The control register's address is 7, so its 5-bit binary address is `%00111`. We've predefined this value as a constant, `CtrlReg`, which we will use instead of `%00111`.

So, in pseudo-code, we'll use the following algorithm to set the clock values:

```
Take Reset pin high to permit writing
Set Bit 7 of the control register to 0 to enable write

Sequentially write the clock values:
For i = 0 (Seconds) to 6 (Years)
        Construct control byte
        Send control byte
        Send data byte from PreSet
Next i
Set Bit 7 of the control register to 1 protecting against
Further changes to the date and time.
Take Reset pin low to block writing
```

Here's the actual code:

```
;Disable DS1302 WriteProtection by 0 in Bit7
        Temp = $00
        RTCCmd = CtrlReg
        GoSub PreSetData

        Set the date and time from the Preset byte table
        For i = 0 to 6
                Temp = Preset(i)
                RTCCmd = i
                GoSub PreSetData
        Next

PreSetData              ;Routine to upload data to the DS1302
;----------
        High Reset
        ;The %0\1,RTCCmd\5,%10\2 results in the equivelent of
        ;sending 10 RTCCMD 0. Each element is sent in element
        ;order but bit reversed, as required by the DS1320.
        ShiftOut Dta,Clk,LSBFIRST,[%0\1,RTCCmd\5,%10\2,Temp\8]
        Low Reset
Return
```

It tracks the pseudo-code quite closely. First, the master write protect bit is cleared, and then the seven time/ date registers are set.

The only tricky part of this program fragment centers around:

```
ShiftOut Dta,Clk,LSBFIRST,[%0\1,RTCCmd\5,%10\2,Temp\8]
```

Let's look at `ShiftOut`. `ShiftOut` is invoked with three arguments:

Dta—A constant or variable that defines the data pin. (We declared `Dta` a constant equal to pin `A1`.)

Clk—A constant or variable that defines the clock pin. (We declared `Clk` a constant equal to pin `A0`.)

Mode—Mode is a value that defines the order in which the bits are sent, and the relationship between the clock (rise or fall) and data being valid, meaning is the data valid on clock rise or on clock fall? Beginning with version 5.3.0.0, MBasic permits `ShiftOut` to have one of 12 possible modes.

	Mode Constant	Bit Order	Clock/Data Relationship	Speed (Machine Cyles)	Speed (With 20 MHz clock)
High Speed (all new in 5.3.0.0)	**FASTMSBPRE**	MSB first	Data valid on leading edge	25	200 KHz (5 µs/bit)
	FASTLSBPRE	LSB first			
	FASTMSBPOST	MSB first	Data valid on falling edge		
	FASTLSPOST	LSB first			
Slow Speed (all new in 5.3.0.0)	**SLOWMSBPRE**	MSB first	Data valid on leading edge	100	50 KHz (20 µs/bit)
	SLOWLSBPRE	LSB first			
	SLOWMSBPOST	MSB first	Data valid on leading edge		
	SLOWLSBPOST	LSB first			
Normal Speed { } indicates backwards compatible for older versions	**MSBPRE {MSBFIRST}**	MSB first	Data valid on leading edge	50	100 KHz (10 µs/bit)
	LSBPRE {LSBFIRST}	LSB first			
	MSBPOST	MSB first	Data valid on leading edge		
	LSBPOST	LSB first			

The DS1302 requires medium speed, LSB-first bit order, and the bit sample before clock pulse speed. Hence, the mode we shall use is `LSBFIRST`, both for `ShiftIn` and `ShiftOut`.

Let's examine the information being sent, `[%0\1,RTCCmd\5,%10\2,Temp\8]`. We send four separate elements, `%0`, `RTCCmd`, `%10`, and `Temp`. Each element is sent in its order of

appearance (left to right), but with its bits sent LSB first. Only the number of bits specified by the `\x` command is sent. Suppose we're clearing the master write protection bit. In this case, `RTCCmd` is set to `CtrlReg (%00111)` and `Temp` is set to $00. The control byte is sent as the composite of the first three elements, sent in element order, but with bits reversed:

Send Order							
%0	%00111					%10	
1	2	3	4	5	6	7	8
0	1	1	1	0	0	0	1

In sending order, therefore, the first three elements are sent as `%01110001`. The DS1302 receives eight bits in that order, and reassembles it into the normal byte order as `%10001110`. Recalling how the command byte is constructed, we see that the byte reassembled by the DS1302 instructs it that the next byte is to be stored in the command register. Since the next byte sent, `Temp`, is $00, the net result of `ShiftOut Dta,Clk,LSBFIRST,[%0\1,RTCCmd\5,%10\2,Temp\8]` is to write a `%00000000` into the command register, thus activating the master write enable.

With this understanding, we can now see how the following code fragment works.

```
For i = 0 to 6
        Temp = Preset(i)
        RTCCmd = i
        GoSub PreSetData
Next
```

The command byte is assembled in the subroutine `PreSetData`, just as we've gone through. Hence, the `Seconds` value that is contained at `Preset(0)` is written to the Seconds register (address 0 in the DS1302), the Minutes value that is contained at `Preset(1)` is written to the Minutes register (address 1 in the DS1302) and so forth.

```
;SInce we have a Super Cap backup, we want charger ON
Temp = Battery
RTCCmd = Charger
GoSub PreSetData
```

We use the same approach to enable the DS1302's built-in backup power supply charging circuit. We've predefined `Charger` as `%01000`, or `8`, the address of the charger control register. The charger options byte has multiple options:

Bit 7	Bit 7	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Trickle Charger Control Bits (TCS)				Diode Control Bits		Resistor Control Bits	

The TCS bits determine whether the charger is active, or disabled. The default is disabled, and only the TCS bit pattern `%1010` enables the charger. Any other bit pattern in the TCS bits disables the charger.

To prevent the backup power from trying to run all the devices connected to the V_{CC} line, the DS1302 includes reverse current isolation diodes. Bits 3 and 2 determine whether one or two diodes will be in series. Two diodes increase the isolation, but add an extra 0.7 V drop to the charging voltage.

Permitted values for bits 3 and 2 are:

Bit 3	Bit 2	Diode Connection
0	0	Trickle charger disabled, independently of TCS bits
0	1	One series diode selected
1	0	Two series diodes selected
1	1	Trickle charger disabled, independently of TCS bits

Finally, Bits 1 and 0 determine the charging current, by selecting the series resistor value:

Bit 1	Bit 0	Resistor Connection
0	0	None
0	1	Typically 2 Kohm
1	0	Typically 4 Kohm
1	1	Typically 8 Kohm

We've selected TCS—enabled, one diode and 2 Kohm series resistance. Hence, our charger option byte is `%1010 01 01`, and we've defined the constant `Battery` accordingly:

```
Battery     Con     %10100101
```

To activate the charger we use the subroutine `PreSetData` to load the appropriate command byte (constructed in `PreSetData` from the 5-bit address `Charger`), followed by the `Battery` option byte.

A brief word on backup power is appropriate. At one time, our choice would have been limited to a choice of battery chemistry. We might have chosen from among a variety of primary (nonrechargeable) batteries, or secondary (rechargeable) batteries. But, recently supercapacitors—capacitors with values of 0.1 farad or more—have appeared on the market. For low current drain devices, a supercap has many advantages over a battery, most

importantly its much longer lifetime. The particular supercap we're using is a 0.47 F, 5 V device. It charges to essentially full capacity in about 4,000 seconds, or just over one hour, when we select a 2 K charging resistor and one diode options in the DS1302 charger option byte. (Supercaps are known for high levels of dielectric absorption, so measurable charging current may be observed for hundreds, or thousands of hours.) The DS1302 consumes 0.3 μA operating current and will continue to operate until 2.0 V. Hence, neglecting internal losses in the supercap, the 0.47 F backup capacitor will allow the DS1302 to continue operating for approximately 3.6 million seconds—about 41 days after power is removed. In practice, the supercap's leakage current may be as much as 10 μA or more, which limits the backup operation to around one day, still more than adequate to carry operation over during normal power outages. If we require extended operation without mains power, then we should consider battery backup, perhaps a long life lithium primary battery.

Since the timekeeping continues during power outages, we need only set the clock once and thereafter may comment out the clock setting parts of Program 22.4.

Now that we've set the clock, let's read it and write the current time to the serial port. We've bundled all the read code into the subroutine `ReadData`, so let's look at it first.

```
ReadData
;---------
       ;Reset pin must be high for read
       High Reset
       ;%10111111 is BURST SEND REGISTER command, so get 8 bytes
       ;the last byte is the control register, which we don't use
       ShiftOut Dta,Clk,LSBFirst,[%10111111\8]
       OldSeconds = Seconds
       ;Now actually read the burst send data.
       ShiftIn Dta,Clk,LSBPRE,[Seconds\8,Minutes\8,Hours\8, |
       Date\8,Month\8,Day\8,Year\8]
       Low Reset
Return
```

Large parts of this should look familiar.

Before doing anything with the DS1302, we must bring the reset pin high. Next, we send a command byte to the DS1302 using `ShiftOut`. The particular command byte we send is the "clock/calendar burst read" command, `%10111111`, and we don't have to construct it from a composite of three elements. The clock/calendar burst read command instructs the DS1302 to send the contents of the first eight registers in address order (starting with seconds and ending with the control register) sequentially, without further commands from the master PIC.

Following the clock/calendar burst read command, we read the first seven bytes with the `ShiftIn` procedure: `ShiftIn Dta,Clk,LSBPRE,[Seconds\8,Minutes\8,Hours\8,`

Date\8,Month\8,Day\8,Year\8]. (ShiftIn is the receive version of ShiftOut and requires the same arguments and mode selection, so we won't elaborate on it.)

We read the first 7 bytes into the array TimeData, using the alias variables Seconds, Minutes, and so on that we earlier declared. Instead of using the DS1302's clock/calendar burst read function, we could have individually read the first seven registers into the elements of TimeData(), using code much like that found in PreSetData, that is, send a command byte to read register 0, then read the value into TimeData(0), repeat for register 1, and so forth.

Now, let's look at how we call ReadData and what we do with the time information it returns.

```
Loop
;---------
        ;Read the clock output. If seconds changed, write time
        GoSub ReadData
        ;Write in form:
        ;13:10:45 11/24/03 Mon 24-Nov-2003 to the serial port
        ;New seconds, so write output
        If OldSeconds <> Seconds Then
        HSerOut [Dec2 (BCD2BIN Hours)\2,":",Dec2 |
        (BCD2BIN Minutes)\2,":",Dec2 (BCD2BIN Seconds)\2]
        HSerOut [" ",Dec2 (BCD2BIN Month)\2,"/",Dec2 |
        (BCD2BIN Date)\2,"/",Dec2 (BCD2BIN Year)\2]
        HSerOut [" ",Str DayName(Day*3)\3," "]
        HSerOut [Dec2 (BCD2BIN Date)\2,"-",Str |
        MoName(3*BCD2BIN Month)\3,"-20",Dec2 (BCD2BIN Year)\2,13]

    EndIf
GoTo Loop
```

Our main program loop calls ReadData to obtain the current time and date. We then compare the current value of Seconds with the value last time we called ReadData, held in OldSeconds. If the value is different, we write the current time to the serial port with a series of HserOut calls and update the value of OldSeconds.

The HserOut calls are mostly straightforward, but a few points are worthy of note. Since the raw data is in BCD form, we use MBasic's BCD2BIN function to convert to the standard binary form with which we are accustomed. To give a month name and day-of-week name output, we earlier defined byte tables DayName and MoName (month name). We index into these arrays with the day-of-the-week number, or the month number. Since both the month and day-of-week start with 1, we added the dummy string "Nul" to the beginning of both DayName and MoName. Alternatively, we could have started the byte tables with "Sun" and "Jan" and subtracted 1 from the day-of-week number and month number.

Following is a sample of Program 22.4's output:

```
13:20:56 11/24/03 Mon 24-Nov-2003
13:20:57 11/24/03 Mon 24-Nov-2003
13:20:58 11/24/03 Mon 24-Nov-2003
13:20:59 11/24/03 Mon 24-Nov-2003
13:21:00 11/24/03 Mon 24-Nov-2003
13:21:01 11/24/03 Mon 24-Nov-2003
13:21:02 11/24/03 Mon 24-Nov-2003
```

22.4 Combination Date, Time and Temperature

Let's merge our program to read two temperature sensors, and add a DS1302 real-time-clock to yield a time and temperature logging output. Program 22.5 reads two DS18B20 temperature sensors and a DS1302 clock chip and writes the time and temperature readings to the serial port. Figure 22.10 shows the circuit hookup. Depending on the length of the cable to the

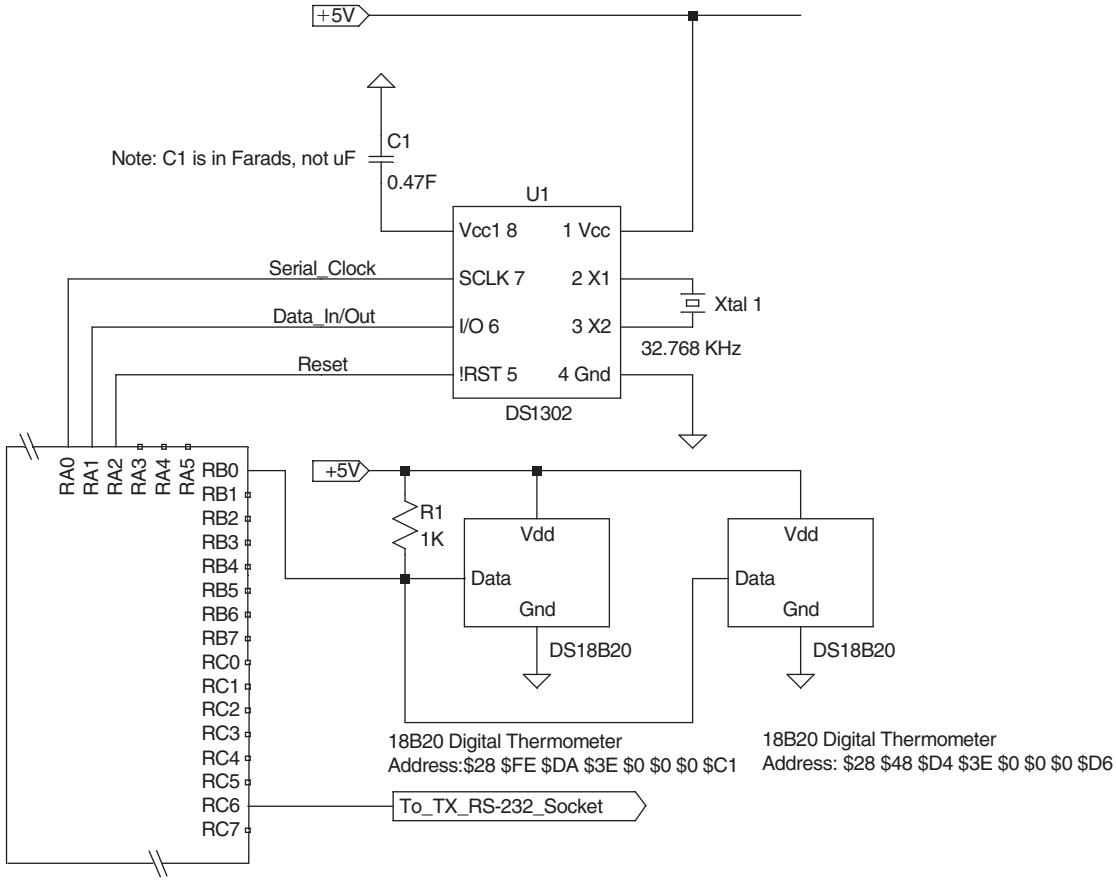


Figure 22.10: Circuit to Read Two Temperature Values and Time of Day

remote DS18B20, it may be necessary to decrease the value of R1, the pull-up resistor. It can be as low as 220 ohms, if required.

Program 22.5

Program 22.5 is very long and is largely a composite of earlier programs in this chapter. Rather than follow our normal structure of listing the program in full, followed by a section-by-section analysis, we'll skip the full listing. Program 22.5, like all other programs in this book, is available in an electronic copy in the associated CD-ROM. If you need a listing to follow while reading the analysis, you may print one from the program file.

Program 22.5 should, by now, require little analysis, as it simply merges programs we've exhaustively studied. Following is a sample of Program 22.5's output:

```
15:12:00 11/24/03 Sensor 1 73.62 F Sensor 2 65.41 F
15:13:00 11/24/03 Sensor 1 73.73 F Sensor 2 65.41 F
15:14:00 11/24/03 Sensor 1 73.62 F Sensor 2 65.52 F
15:15:00 11/24/03 Sensor 1 73.51 F Sensor 2 65.41 F
15:16:00 11/24/03 Sensor 1 73.39 F Sensor 2 65.41 F
15:17:00 11/24/03 Sensor 1 73.51 F Sensor 2 65.29 F
```

We've set the program to output a reading once every minute, but the output interval may be increased by changing the value of the constant `Interval`.

```
Interval      con    1      ;how often read data
```

Defining `Interval` as 5, for example, outputs a reading once every five minutes.

Figures 22.11 and 22.12 show the indoor and outdoor temperature readings collected with this program over 24 hours. The abrupt drop in outdoor temperature coincided with the arrival of a cold front and rainstorm. The variation in indoor temperature (the data was collected in my basement workshop) results from the forced air furnace cycling off and on.

22.4.1 Which Edge?

Serial clocked data is transferred on either the rising edge of the clock pulse, or the falling edge. By convention, clock idle is 0 V, so the leading edge is the transition from 0 to 1 and the falling edge is the transition from 1 to 0. Figure 22.13 shows both possibilities. Figures 22.14 and 22.15 show a bit with value 1 being transferred with these two options. The data being transferred is %10000000, MSB first.

Don't get confused when looking at Figs 22.14 and 22.15 because the data pulse is wider than the clock pulse. In Fig. 22.15, for example, the data is valid on the falling edge of the first clock pulse, but it's also valid on the rising edge of the second clock pulse. But, the data is only read once for each clock pulse; i.e., data is always read on the rising edge or the falling

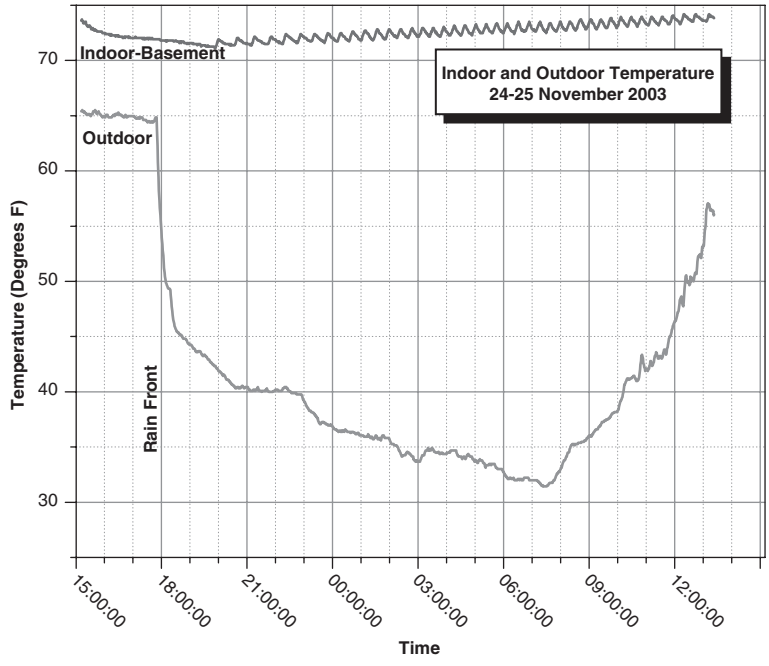


Figure 22.11: Indoor and Outdoor Temperature

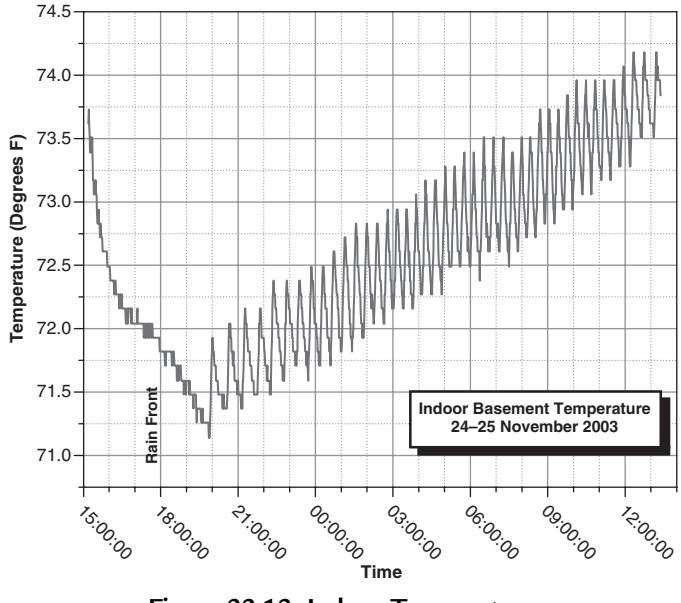


Figure 22.12: Indoor Temperature

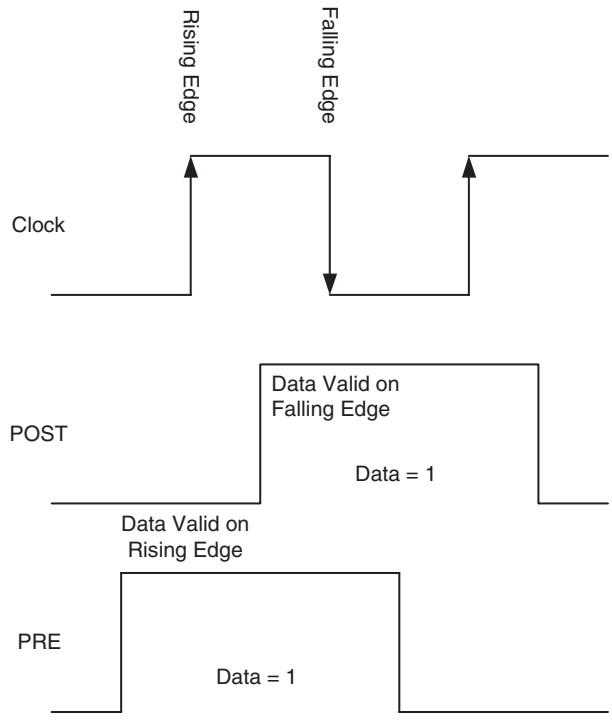


Figure 22.13: Rising and Falling Clock Edge Data Transer

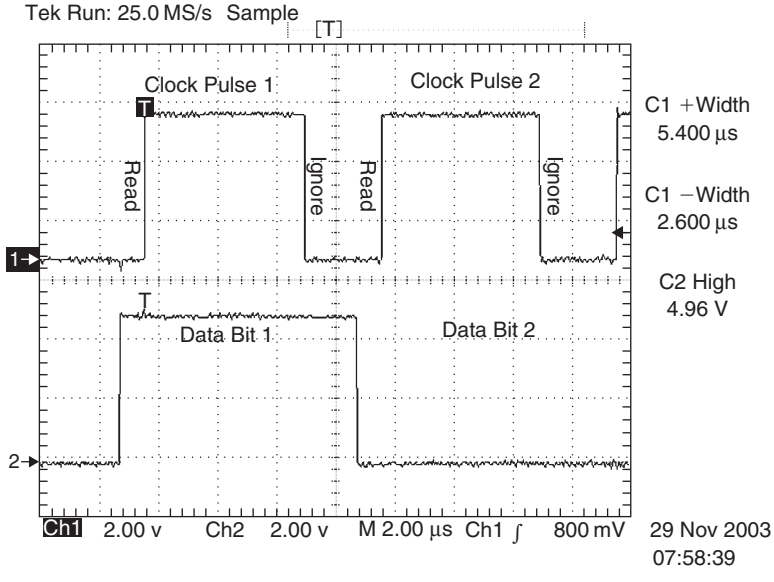


Figure 22.14: Data Valid on Rising Edge Ch1: Clock; Ch2: Data

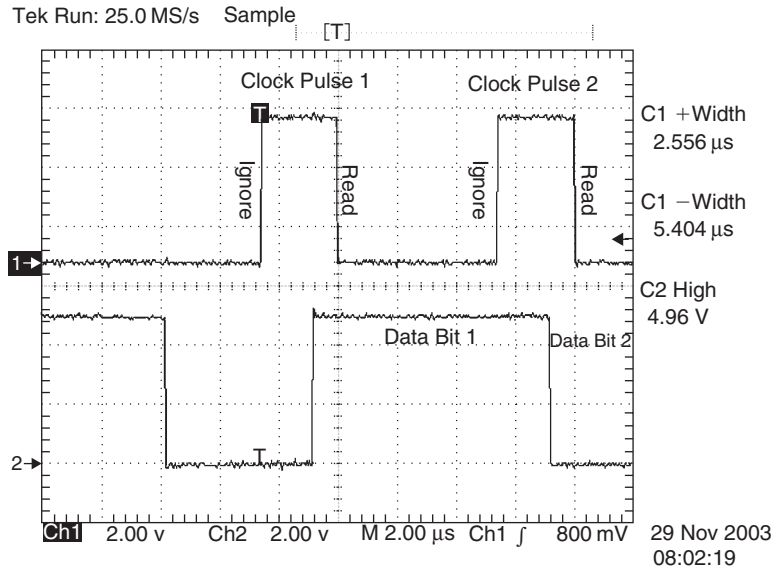


Figure 22.15: Data Valid on Falling Edge Ch1: Clock, Ch2: Data

edge, but not both. In Fig. 22.15, data is read on the falling edge of each clock pulse; hence the data line's status during the rising edge is immaterial.

22.4.2 Bit Order

We have two choices to send the individual bits that comprise a byte; from most significant (B7) to least significant (B0), or from least significant to most significant.

22.4.3 Reading the Data Sheet

Let's look at the DS1302 data sheet to see where we find the clock edge and bit order information.

Let's start with bit order, as that's the simpler of the two. It's hard to be clearer than the DS1302 data sheet:

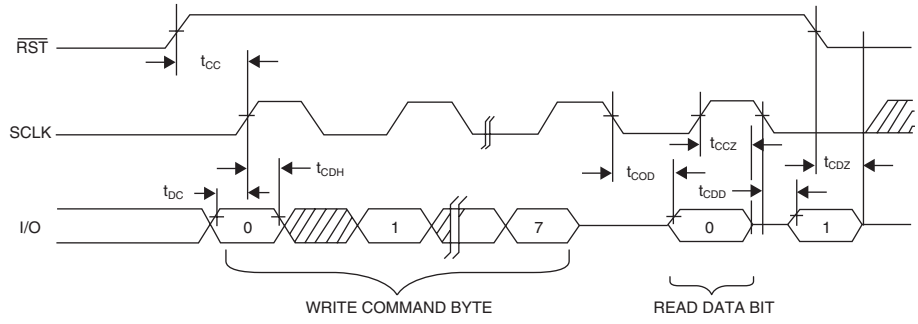
> **DATA INPUT**
> Following the eight SCLK cycles that input a write command byte, a data byte is input on the rising edge of the next eight SCLK cycles. Additional SCLK cycles are ignored should they inadvertently occur. Data is input starting with bit 0.
>
> **DATA OUTPUT**
> Following the eight SCLK cycles that input a read command byte, a data byte is output on the falling edge of the next eight SCLK cycles. Note that the first data bit to be transmitted occurs on the first falling edge after the last bit of the command byte is written. Additional

SCLK cycles retransmit the data bytes should they inadvertently occur so long as RST remains high. This operation permits continuous burst mode read capability. Also, the I/O pin is tri-stated upon each rising edge of SCLK. Data is output starting with bit 0.

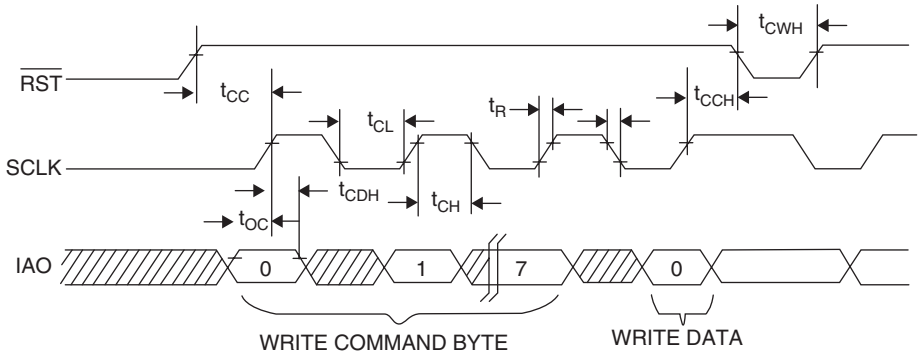
Since data is both input and output starting with bit 0, the bit order is LSB first.

To determine whether the data is to be read on the rising or falling clock edge, we examine the timing diagrams.

TIMING DIAGRAM: READ DATA TRANSFER Figure 5



TIMING DIAGRAM: WRITE DATA TRANSFER Figure 6



(Figures courtesy of Maxim Integrated Products, Inc. Reprinted with permission.)

With respect to the DS1302 reading data (PIC writing data with `ShiftOut`), the data bits are shown to be read by the DS1302 on the rising edge of the clock. Hence we will use LSBPRE mode with `ShiftOut`.

With respect to the DS1302 writing data (PIC reading data with `ShiftIn`) the data is shown to be written immediately after the clock's falling edge and hence must be read by the PIC on the rising clock edge. Accordingly, we will use LSBPRE mode with `ShiftIn`.

Figures 22.16 through 22.19 show the difference among the four order and clocking options. In each case, the byte being sent is %01000000.

Predefined Constant	Value	Meaning		Sample Illustration
		Bit Order	Clock / Data Relationship	
MSBPRE (MSBFIRST)	0	MSB first	Data valid on leading edge	Figure 22.16
LSBPRE (LSBFIRST)	1	LSB first	Data valid on leading edge	Figure 22.17
MSBPOST	2	MSB first	Data valid on falling edge	Figure 22.18
LSBPOST	3	LSB first	Data valid on falling edge	Figure 22.19

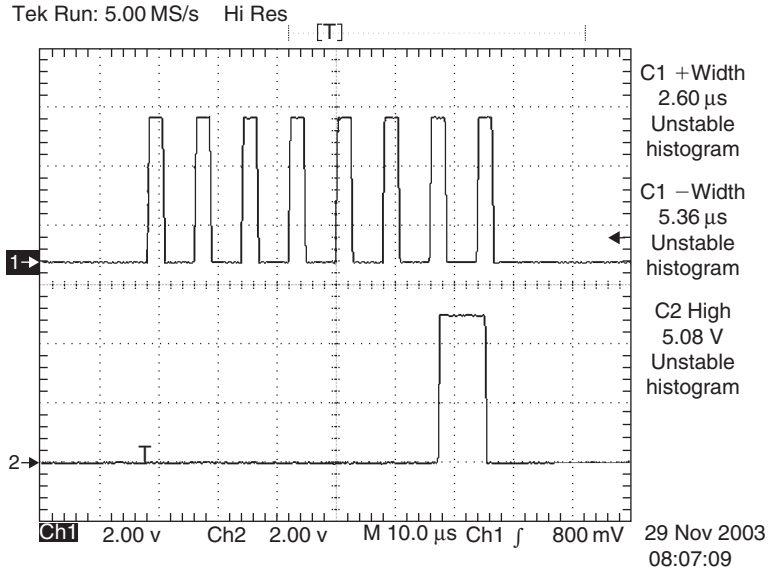


Figure 22.16: Mode: MSBPRE Ch1: Clock; Ch2: Data

22.5 Ideas for Modifications to Programs and Circuits

- The DS2401 is a 1-wire electronic serial number chip. Its sole function is to return a 64-bit unique serial number. How might this device be used? For example, it could be used as a security access device, or it could identify a calibrated printed circuit board. Or, it might be used to multiplex several switches on a single bus (each switch connects a different DS2401 to the bus. By reading the serial numbers, which switch is closed may be determined.) Try revising the circuits to read and use the unique serial number available from a dS2401.

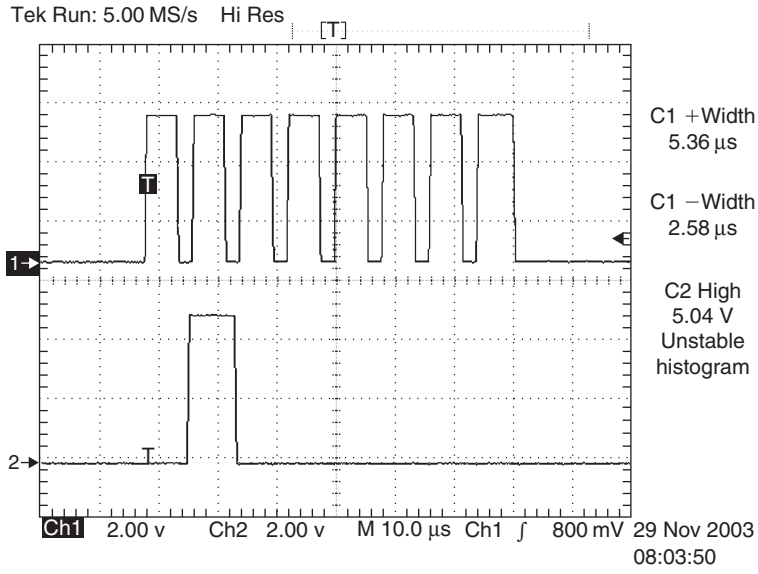


Figure 22.17: Mode: LSBPRE Ch1: Clock; Ch2: Data

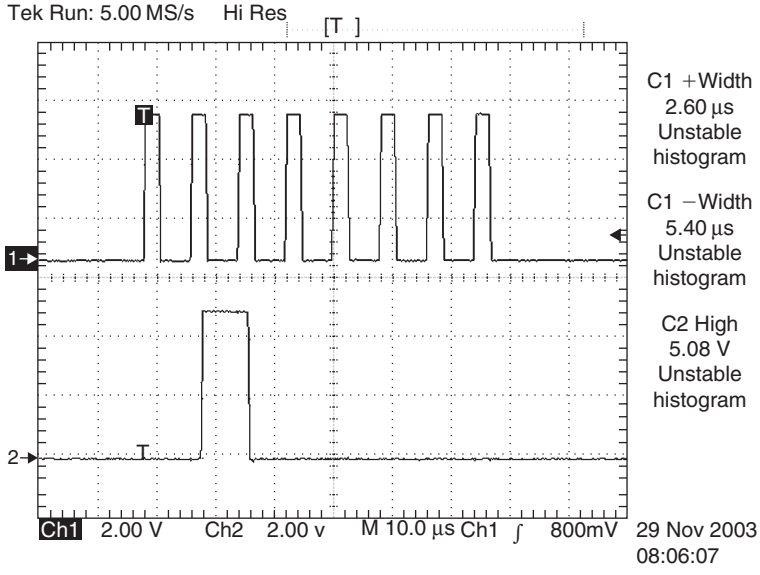


Figure 22.18: Mode: MSBPOST Ch1: Clock; Ch2: Data

- Program 22.5 is a start of a logging weather station. Add a Motorola MPXA6115AU absolute pressure sensor for barometric readings and a Humirel HM1500 humidity sensor to the circuit of Fig. 22.10. Add a multiline LCD to display indoor temperature, outdoor temperature, barometric pressure and humidity. (Both the HM1500 and MPXA6115AU sensors require reading an analog voltage.

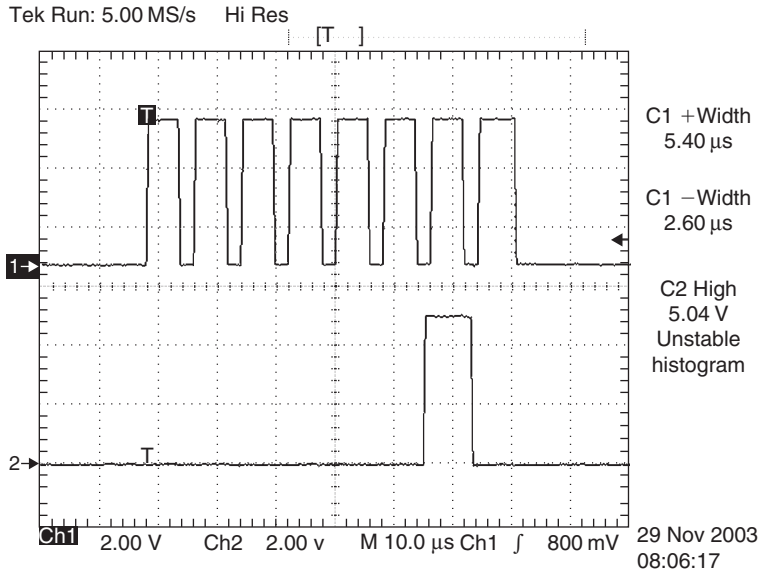


Figure 22.19: Mode: LSBPOST Ch1: Clock; Ch2: Data

- How long can the cable connecting the remote DS18B20 be before capacitive loading becomes a problem and temperature values can no longer be read? How does this vary with cable type (cable capacitance)? How does changing the pull-up resistor alter the distance performance?
- Dallas Semiconductor has an interesting line of 1-wire devices, packaged in stainless steel "buttons" called the IButton. IButton products have their own website, http://www.ibutton.com. Are any of the IButton devices useful for your projects? How would you go about interfacing them with a PIC?

Sidebar
ShiftOut/ShiftIn—Rising or Falling Clock and Bit Order

When transferring serial clocked data we must decide whether the data is to be valid on the rising edge of the clock, or the falling edge. We also must decide whether the data is to be sent is in order B7...B0 (most significant bit first) or B0...B7 (least significant bit first). MBasic provides four options when writing serial clocked data with **`ShiftOut`**, or when reading it with **`ShiftIn`**. Although we talk of these as "options" they in fact are mandatory for the particular device with which you intend to communicate. The Dallas Semiconductor DS1302 real-time clock chip, for example, requires the data to be valid on the rising edge and for data to be sent least significant byte first. Not all types of chips that communicate via serial clocked data will use these conventions. Accordingly, you must delve into the device's data sheet to determine the correct clock edge and the correct bit order sequence.

References

[22.1] Motorola, Inc., Sensor Device Data Book, Publication DL200/D, Rev. 5 (January 2003).

[22.2] Dallas Semiconductor Corp. division of Maxim Integrated Products, Inc., DS1302 Trickle Charge Timekeeping Chip, (September 29, 2001).

[22.3] Dallas Semiconductor Corp. division of Maxim Integrated Products, Inc., DS18B20 Programmable Resolution 1-Wire Digital Thermometer, (January 5, 2002).

[22.4] Dallas Semiconductor Corp. division of Maxim Integrated Products, Inc., DS2401 Silicon Serial Number, (February 21, 2002).

[22.5] Dallas Semiconductor Corp. division of Maxim Integrated Products, Inc., 1-Wire Communication with a Microchip PICmicro Microcontroller, (September 9, 2003).

[22.6] Awtrey, Dan, Transmitting Data and Power over a One-Wire Bus, Sensors, The Journal of Applied Sensing Technology (February 1997).

[22.7] Dallas Semiconductor Corp. division of Maxim Integrated Products, Inc., MicroLAN Design Guide—Tech Brief 1, (September 24, 2002).

[22.8] Dallas Semiconductor Corp. division of Maxim Integrated Products, Inc., Printed Circuit Board Identification Using 1-Wire Products, Application Note 178, (June 5, 2002).

[22.9] Richey, Rodger, Yet Another Clock Featuring the PIC16C924, AN649, Microchip Technology, Inc., Doc. No. DS00649A (1997).

[22.10] Dallas Semiconductor Corp. division of Maxim Integrated Products, Inc., Crystal Considerations for Dallas Real-Time Clocks Application Note 58, (December 9, 2002).

CHAPTER 23

Infrared Remote Controls

According to Internet lore, the average household has between five and seven remote controls. Before starting this chapter, I collected a box full to test—one Zenith, two Nakamichis, two Mitsubishis, one Samsung, one Toshiba, one Hitachi, and two Radio Shack universal remote controls—just by going through our house. Two belong to equipment long since scrapped as uneconomical to repair, but the rest are in more or less common use.

Although the first "cordless" TV remote controls used ultrasonic technology (and without electronics in the remote, no less, employing mechanically stroked tuning forks), when inexpensive LEDs became available, digital infrared transmission became the norm. (Low-power radio is used for garage door openers, automobile remote locks and the like.)

With an inexpensive IR receiver module connected to a PIC, it's easy to decode most remote controls. Once decoded, we then may use the techniques we've seen in other chapters to control a variety of devices. We might use a relay to turn a light off and on, or we might translate volume up/down buttons to adjust audio levels through a SPI-controlled MCP42010 digital potentiometer.

Finding definitive information on IR remote controls isn't easy; the consumer electronic manufacturers hold the information closely. I've pieced together this chapter from research published on the Internet by inspired hackers, supplemented with measurements of the assortment of remote controls available to me.

First, let's look at the lowest level of the transmission and reception—and at this level, all remotes work alike. When you push a button, the remote generates a particular digital control sequence and transmits it as multiple bursts of IR light. Each individual burst consists of a few dozen rapid on/off cycles, as illustrated in Fig. 23.1. An IR receiver detects these bursts and smoothes out the individual on/off carrier cycles, thereby reconstituting the original digital sequence, as shown in Fig. 23.2.

The frequency of each IR on/off cycle within a burst is the carrier frequency, and varies from manufacturer to manufacturer. The lowest carrier frequency commonly used is 32.75 kHz, and the highest is 56.8 kHz, but by far the most common carrier frequency is 38 kHz. IR receivers are frequency selective and to achieve reasonable range, you must match the receiver frequency to the remote's carrier frequency. (A few broadband receiver modules are available, such as Vishay Semiconductor's TSOP1100 model.) We'll talk more about IR receivers shortly.

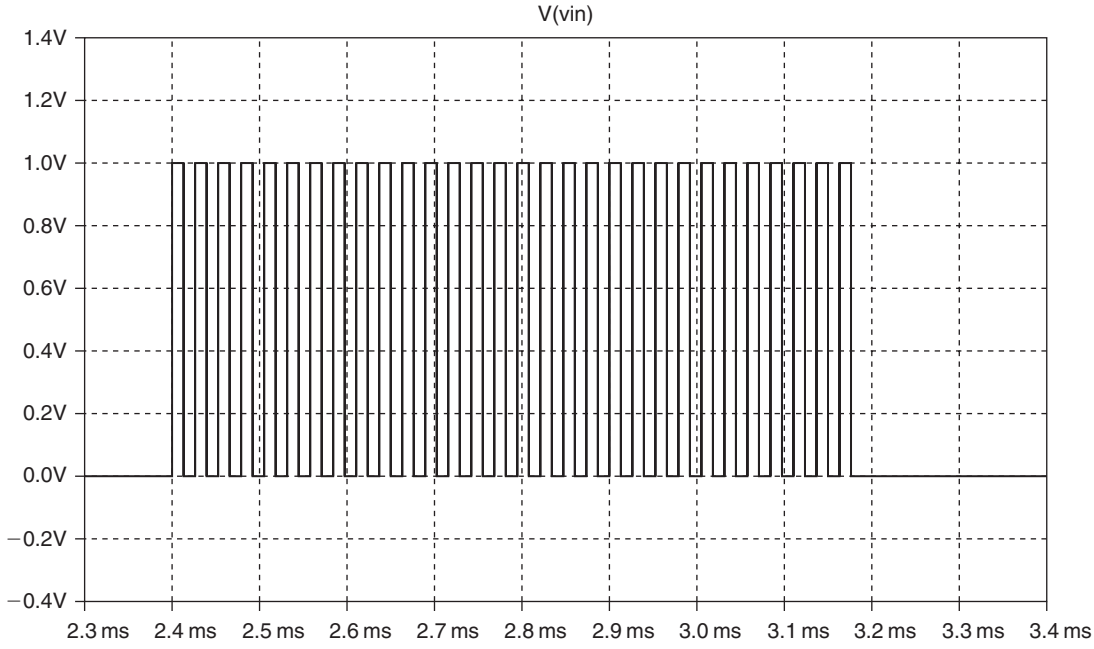


Figure 23.1: Simulated Drive to IR LED for One Burst—38 KHz Carrier

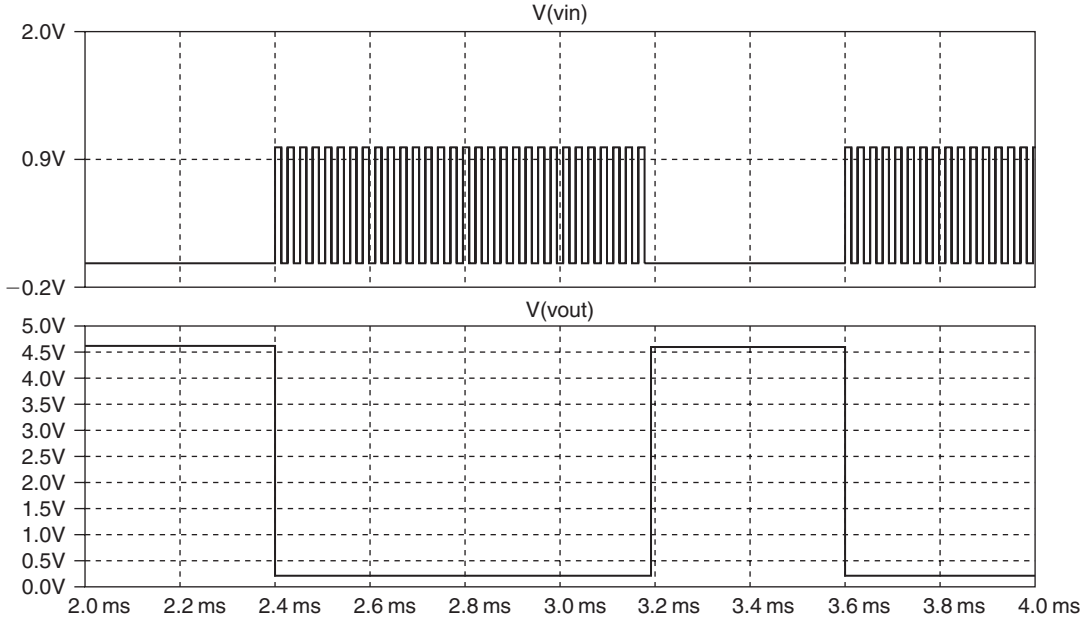


Figure 23.2: IR Receiver Smoothes Carrier Cycles into Clean Waveform; Upper Waveform: Raw IR; Lower Waveform: IR Receiver Output

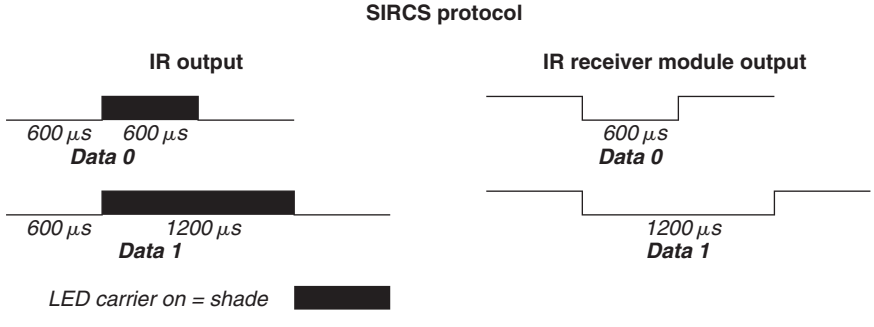
The IR receiver also translates between light level and PIC-compatible logic level—its output is either at logic 1 or logic 0. Receivers are designed so that when no IR light is detected, the receiver's output is logic 1; and is logic 0 when IR light is detected. This may seem backwards at first, but in fact we'll find it very convenient to forget about whether a light output causes a 1 or a 0 and instead concentrate on the receiver's logic output values.

23.1 Common Encoding Standards

How, then, are the receiver's 1s and 0s organized into useful information? Here we must delve into some detailed manufacturer-specific information. Three commonly used methods of encoding the 1s and 0s into useful information are:

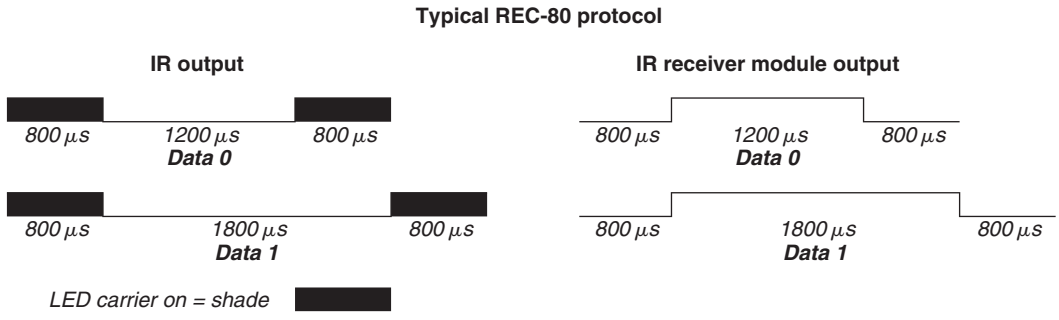
Philips RC-5 Code—is a 14-bit biphase (Manchester) encoded system, with a 36 kHz carrier. The documentation for this system is publicly available in a Philips Semiconductor Application Note. Because we're going to concentrate on other systems, we won't further discuss the RC-5 code. (Philips recently introduced RC-6, an expanded version of RC-5, to provide for additional data length.)

Sony—(Serial Infrared Remote Control System, SIRCS)—Originally a 12-bit code, but later expanded to 15-bit and 20-bit versions. The data is transmitted LSB first, using pulse width coding:



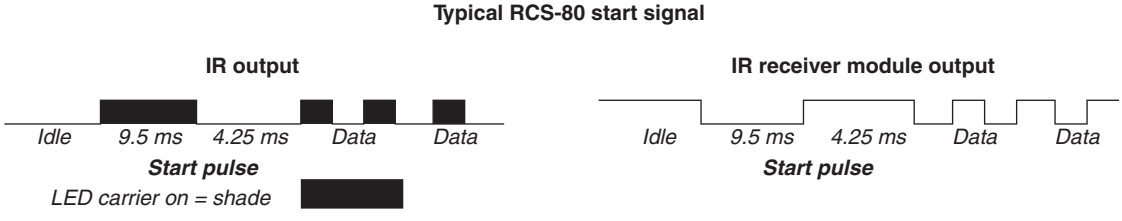
SIRCS uses a 40 kHz carrier frequency, and, as reflected above, 0 and 1 are distinguished by different carrier-on intervals. We won't further concern ourselves with SIRCS.

REC-80—Is a 32-bit "space width" modulated code, and is, in some sense, an inverse of SIRCS. In SIRCS, the carrier-off period is constant and the carrier-on period has two values, a narrow pulse for 0 and a wide pulse for 1. REC-80 has a constant carrier-on period and has two off-periods, a narrow pulse for 0 and a wide pulse for 1.



The sketch of the IR receiver output suggests our decoding strategy for REC-80 signals; measure the logic high pulse duration out of the receiver and classify it as wide or narrow. We'll see, however, that in practice the narrow pulses tend to be much shorter than 1200 μs and the wide pulses are somewhat shorter than 1800 μs, due, in part at least, to the IR receiver's response time. (Some sources suggest the data 0 "no-carrier" period is twice the "carrier-on" time, and the data 1 "no-carrier" period is three times the "carrier-on" time. None of the remote control signals I measured came close to this 1:2:3 ratio.)

A long start signal is sent before the data bits:



The normal REC-80 protocol has 32 data bits, in theory, organized as:

Typical REC-80 Data Organization Sending Order							
Byte 3		Byte 2		Byte 1		Byte 0	
Device Code		D.C. Check Byte		Function Code		F.C. Check Byte	
LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB

The check bytes are logical NOT copies of the associated data bytes. (The logical NOT operation converts 1s to 0s and 0s to 1s.) Again, however, you should be prepared for discrepancies between this typical organization and the way manufacturers actually implement their remote controls. Some, for example, might increase the possible number of device codes and function codes by dropping the check bytes, thereby yielding 16-bit device codes and function codes.

Most remote controls send the code for a key press several times, with the thought that at least one of the repeated messages will be correctly received. Repeated receptions of the same command may lead to a different error, however. For example, the power-on/power-off code is typically assigned to one "power" button. How then is the receiver to distinguish between receiving two correct copies of a code the user intends to be a single power-on command from a power-on followed by a power-off command? Some manufactures employ a "toggle bit" to distinguish between these two cases; sequential presses of a key will alternate between two codes differing in the toggle bit. Thus, two successive copies of a command received due to normal message repetition will be identical; two successive copies of a command received due to the user pressing the command twice will differ by the value of the toggle bit. None of the remote controls I worked with implemented toggle bits, however.

23.2 IR Receiver

Although it's possible to decode an IR control signal with nothing more than a photodiode or phototransistor, to work well these devices require significant ancillary circuitry. Rather than reinvent the mousetrap, we'll use an inexpensive IR receiver module. To give you an idea of the complexity of these devices, Fig. 23.3 provides a high-level block diagram of a typical IR receiver, a Vishay TSOP12xx series device.

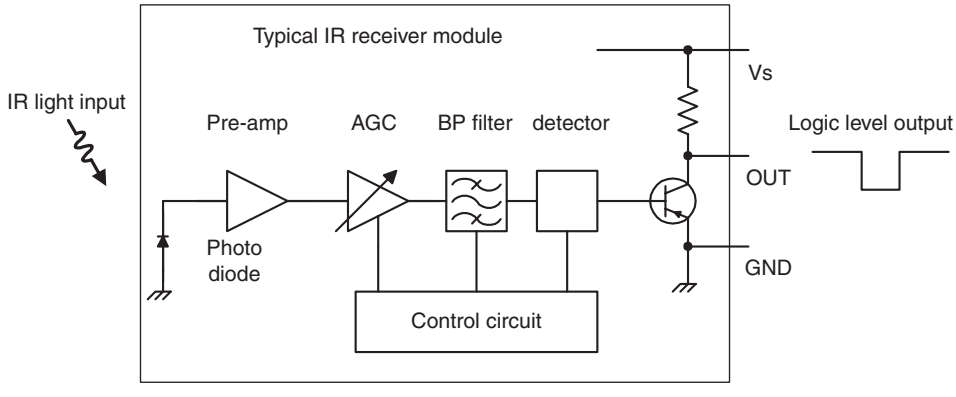


Figure 23.3: Block Diagram—Typical IR Receiver Module

Let's see how these individual modules work, at least at a high level. The photodiode is most often employed with a current-to-voltage pre-amplifier. Light photons hit the PN semiconductor junction in the photodiode and give rise to small currents by displacing electrons. The current-to-voltage pre-amplifier provides a voltage output proportional to the photoelectric current. To prevent the photodiode from being saturated by ambient light, some integrated receivers, such as the TSOP devices, include an optical filter to reject visible light but pass infrared.

The block labeled "AGC" represents the automatic gain control function within the module. The signal level received by the photodiode changes drastically as you orient the remote at differing angles to the receiving diode, or change distance between the remote control and the receiving diode. The AGC amplifier increases its gain when the input signal is weak and decreases its gain when the input signal is strong, thereby stabilizing its output at a level roughly independent of the received signal level.

After the AGC amplifier, the incoming carrier signal is bandpass filtered. This is a very important step in reducing interference from other light sources, such as room lighting, or stray sunlight. After all, the IR receiver must differentiate a weak IR emitter from a sea of background illumination, and this is the job of the bandpass filter. (Even though the optical filter helps, many light sources, such as sunlight, have strong infrared components.) By selectively filtering on the carrier frequency, the bandpass filter rejects other light sources, such as sunlight, that are either unmodulated, or fluorescent lighting, which is modulated with a 120 Hz carrier from the AC mains power. The bandpass filter is responsible for the need to match the receiver frequency to the carrier frequency of the remote control.

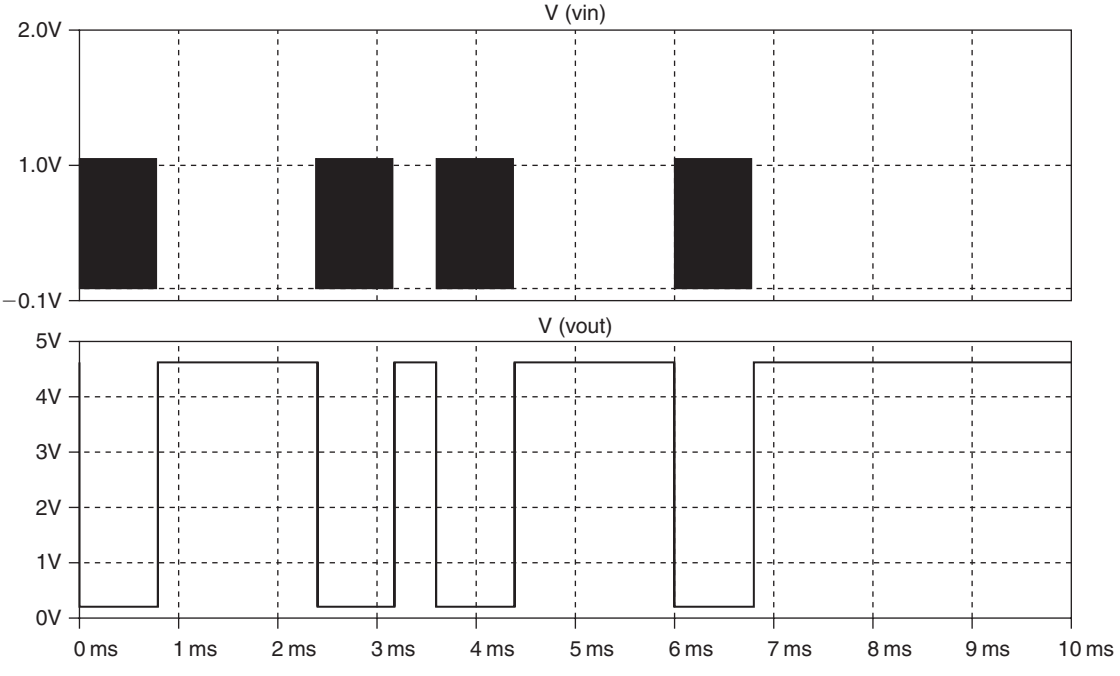


Figure 23.4: Input IR Pulses Yield Inverted Logic Level Output

The output of the bandpass filter is envelope detected and drives the logic level output stage. As illustrates, the convention is that IR receivers invert the sense of the received signal; no-IR causes a logical high and IR-received causes a logical low.

> *In the remainder of this chapter, when we refer to high and low, or positive pulses and negative pulses, it is with respect to the output of the IR receiver,* ***not the actual IR illumination of the remote control unit.***

In writing this chapter, I tested two Sharp Electronics GP1U5 receiver modules, and several Vishay TSOP12xx integrated receivers, with similar results. The GP1U5 modules have recently been replaced by GP1UD26/27/28 series devices, which are physically smaller but have similar performance characteristics. I've also seen data sheets leading me to believe that Sharp's GP1U5 modules have been cloned and still may be available if you search diligently. Vishay's TSOP series of IR receivers is available from many electronic parts suppliers, and cost around $1 in small quantities.

You should match the IR receiver's response frequency (the bandpass filter center frequency in Fig. 23.3) to the carrier frequency of your remote control, within 10% or so. Operating outside this range, such as decoding a 32.75 kHz signal with a 38 kHz receiver—or vice versa—significantly reduces range. If you don't know the carrier frequency of your remote control, it's easy enough to measure it with a phototransistor using the circuit shown in Fig. 23.5. The resulting waveform, when the QSE114 phototransistor is illuminated by a Mitsubishi remote control, is shown in Fig. 23.6. Measuring the time interval between peaks yields a calculated carrier frequency of 37.88 kHz, so the nominal carrier frequency is 38 kHz. You may wonder why the phototransistor output isn't the nice square wave we expect from theory. The QSE114 is an IR photoreceptor, with a rise and fall time of 8 μs, so we see one problem immediately; a 38 kHz square wave has an on-time of 13 μs, so we would expect to see a severely distorted output waveform to begin with. And we won't see even that speed from a QSE114 when operating with a simple resistive collector load. But, to measure the carrier frequency, all we need are consistent reference points on the waveform.

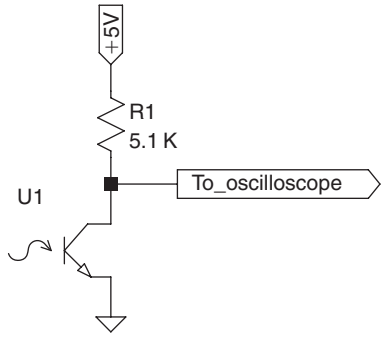


Figure 23.5: Simple Circuit for Measuring IR Carrier Frequency

One final note and then on to some code. Although the bandpass filter reduces stray light interference, it can't solve a second major problem—background lighting saturating the photodiode and the current-to-voltage preamplifier, since the photodiode is more or less

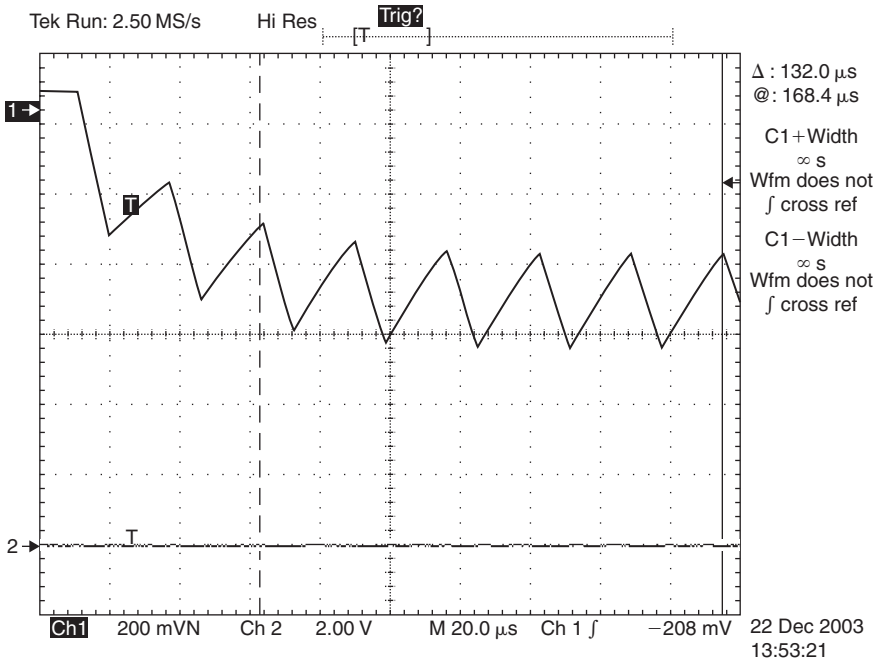


Figure 23.6: Measuring the Carrier Frequency of a Remote Control Ch1: Collector of QSE114 Phototransistor; Ch2: Unused

sensitive to visible as well as infrared light frequencies. You may significantly improve decoding range by adding a dark red plastic IR filter over the lens of the IR receiver. If you can't find the correct IR filter material, experiment, as some common plastics are opaque to visible light but transmit IR. A fully exposed piece of developed color negative film may also work as an IR filter. Vishay's TSOP-series integrated IR receivers have a built-in IR filter, which is another reason to prefer them to Sharp's GP-series modules.

23.3 Characterizing Wide/Narrow Pulse Intervals

Remember that we have decided to work only with REC80 protocol, and that high/low references are with respect to the IR receiver output. Connect the IR receiver module to the PIC as illustrated in Fig. 23.7. Let's look at the output of the IR receiver when it receives a remote control signal. Figure 23.8 shows idle, then a start pulse, followed by 32 wide or narrow data pulses, with a return to idle. We might measure the pulse width with the oscilloscope, by expanding the sweep speed, with the results illustrated in Fig. 23.9. But, since our ultimate objective involves reading the remote control via a PIC, let's use the PIC to measure the received pulse widths.

The start pulse we see in Fig. 23.8 consists of about 10 ms low (IR-on), followed by a 4.2 ms high (IR-off). Since we key on highs, we'll regard the start pulse as a nominal 4.2 ms high pulse preceding the actual data.

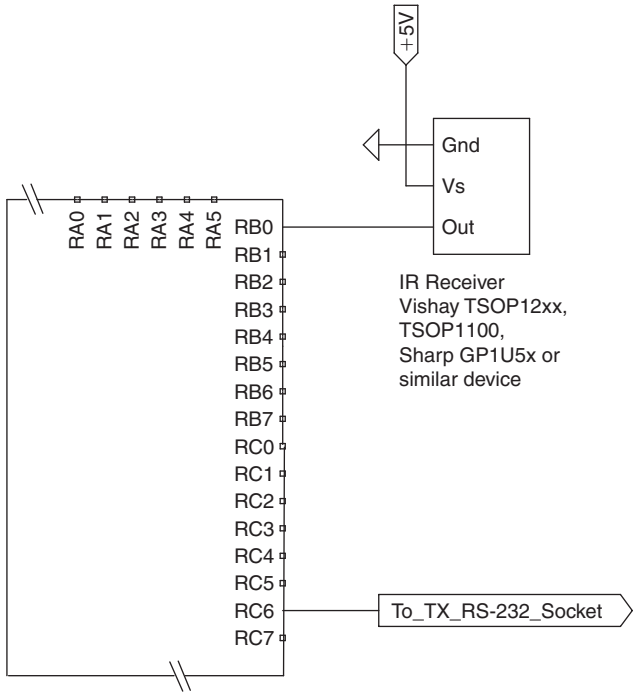


Figure 23.7: Connecting an IR Receiver Module to the PIC

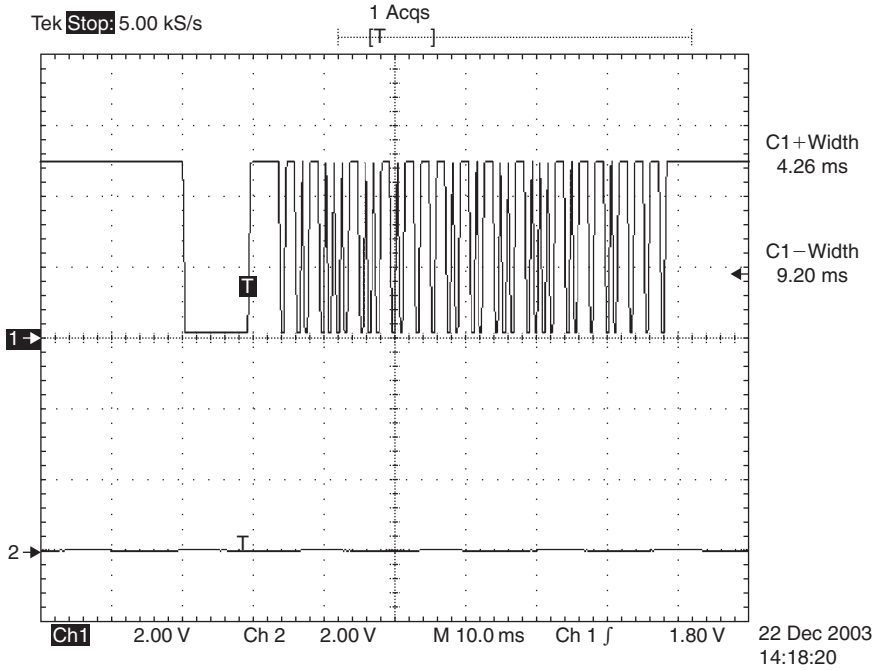


Figure 23.8: IR Receiver Output, REC-80 Signal; Ch1: IR Output Ch2: Unused

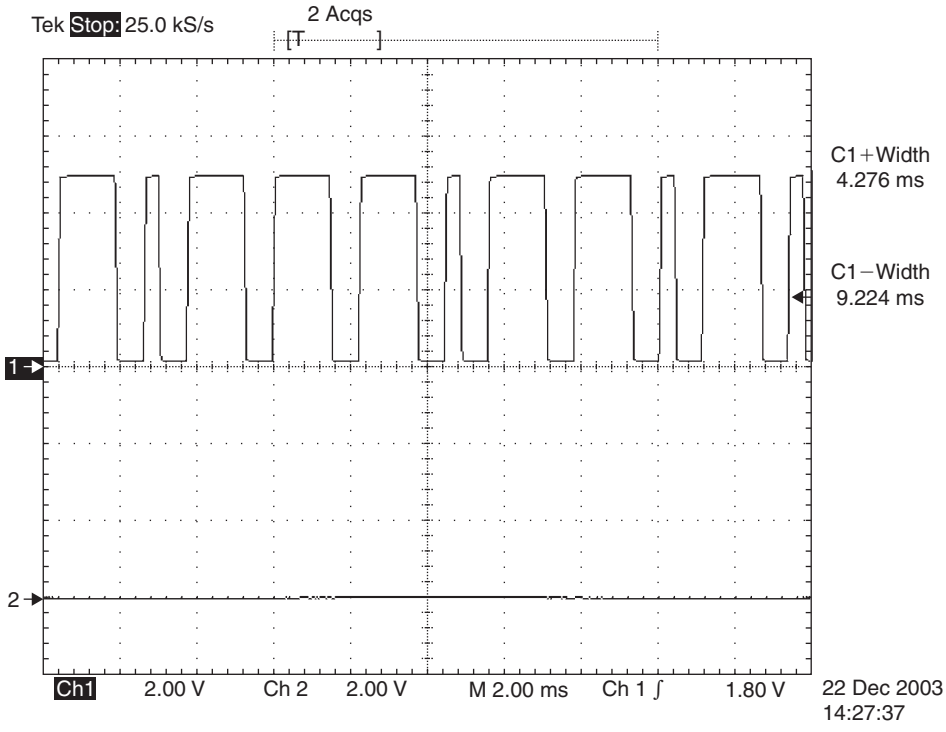


Figure 23.9: Expanded View of Data Pulses; Ch1: IR Output; Ch2: Unused

```
;Program 23.1
;IR decoder output ;to Pin B0.
;Measure positive pulse ;width and
;form histogram and dump to serial for analysis

;Variables
;----------
Width           Var       Word          ;Read pulse width
WidthArray      Var       Byte(100)     ;Data to be histogrammed
i               Var       Word          ;counter
j               Var       Byte          ;counter

;Initialization
;-------------
Clear
Input B0    ;IR Receiver to pin B0
EnableHSerial
SetHSerial H115200
```

Program 23.1

```
Main
;--------
      ;In idle condition receiver is high
      If PortB.Bit0 = 1 Then
            GoTo Main
      EndIf
      ;press keys quickly in front of receiver
      For i = 0 to 999 ;take 1K samples

            ;measure positive pulse width; actually no-IR-output
            ;but we are inverted, so it makes sense to
            ;consider it a positive going pulse
            ;with variable width
            ;PulsIn in units of 1.4us => 18 units = 25.2us
            PulsIn B0,1,Width
            ;save in bins of 25.2uSec
            j=Width / 18
            If j > 99 Then
                  j = 99
            EndIf
            ;increase counter
            WidthArray(j) = WidthArray(j)+1
      Next

      ;dump histogram counters for later analysis
      For j = 0 to 99
        HSerOut ["Width",9,Dec(j*252)/10,9,Dec WidthArray(j),13]
      Next
      Pause 100
End
```

Program 23.1: Continued

Our objective with Program 23.1 is to measure a selection of remote controls and see if we might identify the width threshold to distinguish a wide (logic 1) pulse from a narrow (logic 0) pulse. We do this by measuring 1000 pulses, generated when you rapidly press a random selection of remote control keys while the program is sampling data. The program then categorizes the measured widths into histogram-type width bins.

```
Main
;--------
      ;In idle condition receiver is high
      If PortB.Bit0 = 1 Then
            GoTo Main
      EndIf
```

We know that the IR receiver's idle condition is high, so we read `B0` until we detect a low, indicating data is starting to arrive. Actually, you'll see that there is sufficient random

interference from florescent lighting to cause periodic low conditions, so it's a good idea to load the program into the PIC, get the remote control in position and then reset the PIC as you begin to press random buttons. (We'll see how to reduce false starts in Program 23.2.)

```
For i = 0 to 999 ;take 1K samples
        PulsIn B0,1,Width
```

The key to reading the IR receiver is MBasic's `PulsIn` procedure. `PulsIn` measures the input pulse duration in unit steps. The unit step duration corresponds to seven instruction cycle durations, where one instruction cycle is $4/F_{osc}$.

Oscillator Frequency	Instruction Cycle Length	PulsIn Unit Step Size	Timeout Base Value
20 MHz	200 ns	1.4 μs	91.75 ms
16 MHz	250 ns	1.75 μs	114.7 ms
10 MHz	400 ns	2.8 μs	183.5 ms

`PulseIn` is invoked with several parameters:

```
PulseIn Pin, State {TimeoutLabel, TimeoutMultiple,} Var
```

Pin—A constant or variable that defines the pin to be measured. `PulsIn` automatically places the pin into input mode. Figure 23.7 shows we've connected the IR receiver to `B0`.

State—`PulseIn` can measure either the positive width or negative width, i.e., the width of the logic 1 signal or the width of the logic 0 signal. `State` is a variable or constant that defines which width is to be measured. If `State = 1`, we start measuring when `B0` goes from 0 to 1 and stop measuring when `B0` goes from 1 to 0, i.e., we measure the positive width. Conversely, if `State = 0`, we measure the width of the 0 level interval. Since we wish to measure the positive width, we set `State = 1`. (Don't get confused; the variable period in REC-80 is the time between LED carrier bursts; but since the IR receiver is inverting, by measuring positive intervals, we are, in fact, measuring the interval between IR light bursts.)

{Timeout Label, TimeoutMultiple}—`PulseIn` checks for the transition defined by `State`, either 0→1 or 1→0. If no such transition has been received after 65,535 unit step intervals, `PulsIn` times out, and execution passes to the next MBasic statement. Optionally it is possible to extend the time out period in steps of 65,535 unit step intervals via the `TimeoutMultiple` value. A value of 2, for example, yields a timeout period of 2 × 65,535, or 131,070 unit step intervals. Assuming our PIC has a 20 MHz clock, each unit step is 1.4 μs, so the basic timeout interval is 65,535 × 1.4 μs or 91.75 ms. Upon passage of this time without an input pulse, execution will pass to `TimeoutLabel`. I found that it is not possible to set `TimeoutMultiple` without also supplying a `TimeoutLabel`, and vice versa and that any attempt to use only one of the two options produced a compiler error. We'll use these optional parameters in later programs, but not in Program 23.1.

Var—Holds the width of the measured pulse, in unit step intervals. `Var` is a 32-bit result, which calls for a long variable if you wish to measure the maximum interval. In our case, we know that the pulse length is under a few thousand microseconds, so we will use a word variable for `Var`.

```
;save in bins of 25.2 uSec
j=Width / 18
If j > 99 Then
          j = 99
EndIf
WidthArray(j) = WidthArray(j)+1
Next
```

Now that we have measured the positive pulse width, we wish to store its value. Our PIC doesn't have enough memory to store 1,000 word variables, and, in any event that isn't necessary, or even desirable. Instead, we categorize each measured width into one of 100 "width bins," each bin being about 25 μs wide. (With a 20 MHz clock, `Width` is measured in unit intervals of 1.4 μs duration, and 18 such units correspond to 25.2 μs.) We do this by dividing the measured width by 18 to yield the histogram "bin number" j and then incrementing `WidthArray(j)` by 1. (We trap for `j > 99`, as we only have allocated 100 bins for the data.) After we have taken 1,000 samples, `WidthArray(j)` holds the number of width measurements falling within each bin value. For example, `WidthArray(20)` holds the number of pulses with measured widths between 500 and 524 μs. In statistics, this is termed a "histogram." (By using a byte array to hold bin counts, we assume, implicitly, that not more than 255 width measurements will belong to any individual bin. That's a reasonably safe assumption for 1,000 total measurements with approximately half expected to be wide and half to be narrow and with an expected spread of pulse duration.)

Execution speed is a concern in this program, and even more so in later programs in this chapter. We wish our program to not miss any positive pulses, so we must be economical with the time available before we expect the next positive pulse to start. For the REC-80 code, we know the space between pulses (the IR-on time) runs between 600 and 800 μs. That's enough time, assuming the PIC is running with a 20 MHz clock, to perform the computations we require. But, we must be careful to not burden our program with lengthy computations during this inter-pulse interval, lest the next positive pulse be missed.

```
;dump histogram counters for later analysis
For j = 0 to 99
HSerOut ["Width units ",9,Dec j*18,9,"Width us ",9, |
                Dec (j*252)/10,9,Dec WidthArray(j),13]
Next
Pause 100
```

After we have taken 1,000 samples the histogram results are sent by serial interface where we capture them with a terminal program. To convert the bin number to microseconds, we

multiply by 25.2. To avoid floating point conversion, we instead multiply by 252 and divide by 10. But, for decoding it doesn't matter whether we use microseconds or unit intervals, as long as we are consistent in setting the boundary thresholds.

Here's a partial sample of the output from a Mitsubishi remote control:

```
Width units     576     Width µs    806     0
Width units     594     Width µs    831     10
Width units     612     Width µs    856     0
Width units     630     Width µs    882     70
Width units     648     Width µs    907     82
Width units     666     Width µs    932     104
Width units     684     Width µs    957     151
Width units     702     Width µs    982     5
Width units     720     Width µs    1008    5
Width units     738     Width µs    1033    0
*     *         *
Width units     1494    Width µs    2091    0
Width units     1512    Width µs    2116    8
Width units     1530    Width µs    2142    7
Width units     1548    Width µs    2167    80
Width units     1566    Width µs    2192    204
Width units     1584    Width µs    2217    0
Width units     1602    Width µs    2242    3
Width units     1620    Width µs    2268    4
Width units     1638    Width µs    2293    0
```

The displayed width value is the lower bound of the bin; *for example*, width 907 µs means between 907 and 932 µs. We can either analyze the data by inspection of the output, or we can plot it. Figure 23.10 shows the histograms for six REC80 remote control units. The data shows that we can safely set our narrow/wide bounds as:

Pulse Category	Minimum Length		Maximum Length	
	Units (1.4 µs)	Microseconds	Units (1.4 µs)	Microseconds
Narrow	250	350 µs	786	1100 µs
Wide	1004	1450 µs	1607	2250 µs
Start	2714	3800 µs	3429	4800 µs

One final point: oscilloscope measurements for each remote control showed a reasonably consistent start pulse of about 4500 µs.

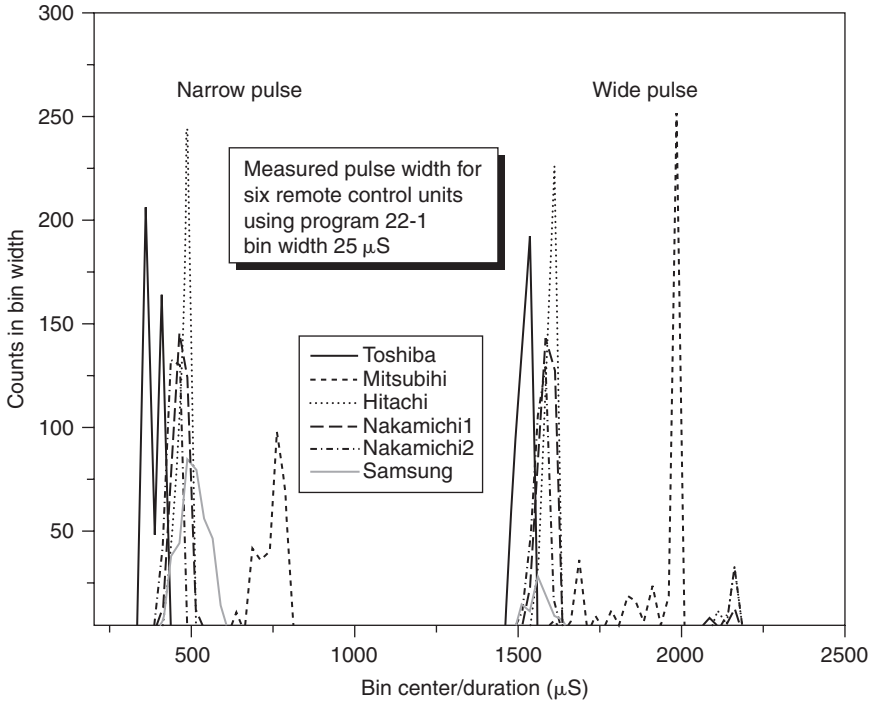


Figure 23.10: Wide and Narrow Pulse Distributions—6 Remote Control Units

Program 23.2

OK, enough preparatory work. Let's look at decoding a real remote control. First, though, let's work through our decoding algorithm in pseudo-code.

```
Searching:
Check input pin. If high (idle) continue checking
If input pin is low, it might be the lead-in to a start pulse
Using PulsIn check for a valid start pulse ~4.2ms
If no valid start pulse go back to Searching

Assumes we know the number of data pulses to receive = BitLength
For i = 0 to BitLength-1
        Read the pulse length
        Store the pulse length in array(bitlength) Warray

Next

Now Analyze Warray to categorize each measurement as
a Wide (W), a Narrow (N) or an Unknown (x) pulse length

Populate a string array TempArray() with W,N or x
based on the pulse width windows.
```

```
Use TempArray to determine validity of data; can't have any
X values if valid.

Substitute 1 for W, 0 for N in a bit-by-bit assembly of the
Decoded value.

Write output to serial port
```

To keep things simple, we'll start with an older Mitsubishi TV/VCR remote control with a 16-bit code. Mitsubishi uses a 32.75 KHz carrier frequency, so the particular IR receiver we use is one optimized for that carrier frequency. (In reading Program 23.2, remember the | character is a line continuation.)

```
;Program 23-2
;to Pin B0. Values for Mitsubishi remote control 16-bit width
;Mitsubishi is special case because it is 16-bits
;and uses 32.75 KHz carrier, not the more common 38KHz

;Constants
;-------------
;the values are in unit intervals of 1.4us which assumes
;a 20 MHz clock.
Wmin    Con             1036    ;1450us wide pulse min width
WMax    Con             1714    ;2400us wide pulse max width
NMin    Con             250     ;350us  narrow pulse min width
NMax    Con             786     ;1100us nNarrow Pulse max width

;Variables
;-----------
CodeValue       Var     Word            ;Holds the 16-bit decoded value
CodeValueR      Var     Word            ;Reversed bits within byte
TempArray       Var     Byte(16)        ;Holds W/N/x string array
i               Var     Byte            ;Counter
WArray          Var     Word(16)        ;Holds measured pulse widths
GoodFlag        Var     Byte            ;If 1, good data

;Initialization
;-------------
Clear
Input B0    ;Decoder to B0
EnableHSerial
SetHSerial H115200

HSerOut ["P 23.2",13,13]

Main
;-----
      ;Unlikw REC-80 spec, Mitsubishi uses no long start pulse
```

Program 23.2

```
       ;All we get is 16 data bits
       ;Wait for a start pulse i = 0
       ASM
       {
WaitForInput
                  banksel PortB
                  btfsc      PortB,0
                  GoTo       WaitForInput
                  NOP
      }

      ; pulse is followed by 16 data bits. Capture
      ;the widths to WArray
      Loop
                  PulsIn B0,1,Main,1,WArray(i)
                  ASM
                  {
                  banksel i
                  incf i&0x7F,f
                  banksel PortB
                  }
                  If i < 16 Then Loop

;GoTo Main
      GoSub CalculateCode
      GoSub GenerateValue
      GoSub DumpRawData

      If GoodFlag = 1 Then
            HSerOut ["Good: ",Str TempArray \16," MSB: ", |
                    iHex CodeValue]
            HSerOut [" LSB: ",iHex CodeValueR,13]
      ELSE
            ;Uncomment line below if want see bad data
            HSerOut ["Error: ",Str TempArray \16," LSB: ", |
                    iHex CodeValue,13]
      EndIf

GoTo Main

CalculateCode
;-----------
      ;Scan the measured with array and classify each as
      ;N (narrow), W (wide) or x (outside W and N windows)
      For i = 0 to 15
```

Program 23.2: Continued

```
                TempArray(i) = "x"
                If (WArray(i) > WMin) AND (WArray(i) < WMax) Then
                      TempArray(i) = "W"
                EndIf
                If (WArray(i) > NMin) AND (WArray(i) < NMax) Then
                      TempArray(i) = "N"
                EndIf
      Next
Return
GenerateValue
;-----------
      ;Now calculate the 16-bit Word equal to the received data
      CodeValue = 0
      ;Set to 0 if we have bad data
      GoodFlag = 1
      For i = 0 to 15
            CodeValue.Bit0 = %0 ;overwrite if W
            ;Now convert W/N/x to numbers
            If TempArray(i) = "W" Then
                  CodeValue.Bit0 = %1
            EndIf
            If TempArray(i) = "x" Then
                  GoodFlag = 0
            EndIf
            ;No need to check N or x, pre-populated with 0s
            ;read assuming sent MSB first. This may not
            ;be the case, but for our purpose order doesn't
            ;matter as we only look for unique values
            ;Data goes in at right, move to left every entry
            ;Don't shift on last bit as no more to be added
            If i <> 15 Then
                  CodeValue = CodeValue << 1
            EndIf
      Next
      CodeValueR.Byte0 = CodeValue.Byte0 REV 8
      CodeValueR.Byte1 = CodeValue.Byte1 REV 8
Return
DumpRawData
;---------
      For i = 0 to 15
            HSerOut [Dec i,9,Dec WArray(i),13]
      Next
Return
End
```

Program 23.2: Continued

There's something different here than in our pseudo-code; there is no code to check for a valid start pulse. That's because Mitsubishi uses a different approach, as seen in Fig. 23.11. The signal goes directly from idle to data, without the prolonged start pulse seen in Fig. 23.8. (It's entirely possible, of course, that the three wide pulses immediately after idle are a preamble and that the remaining 13 bits are data. That's one of the difficulties of reverse-engineering a protocol with only a limited sample of devices to test. Regardless, we will decode these as if they are valid data bits, and their status is immaterial to the technique we use in Program 23.2.)

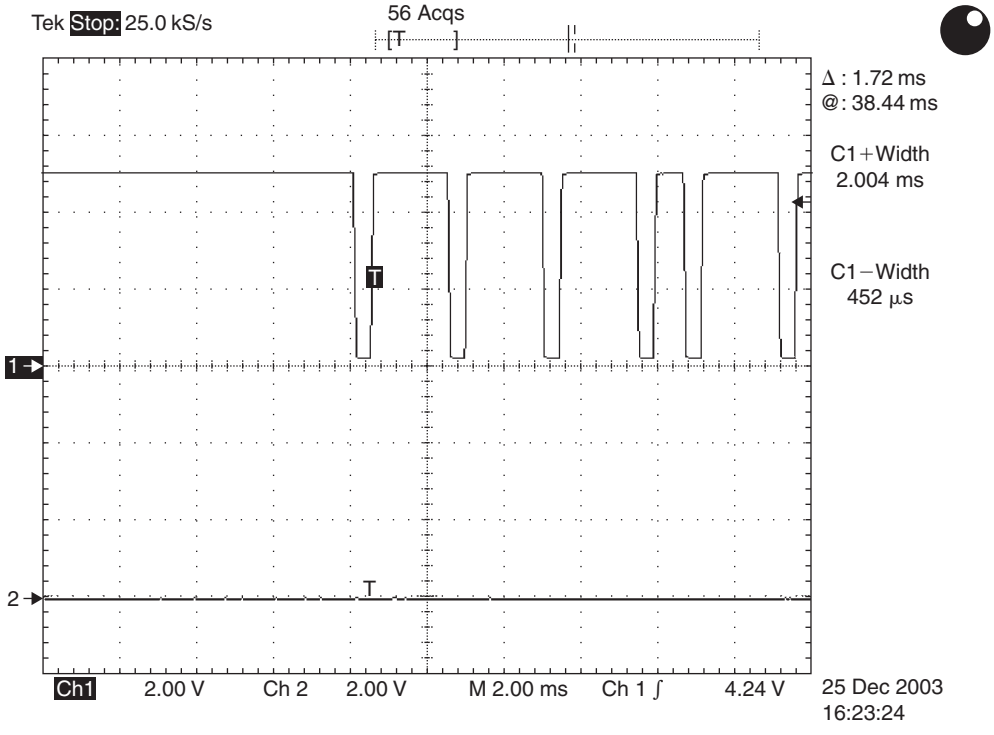


Figure 23.11: Lead-in to Mitsubishi 16-bit Data; Ch1: IR Receiver Output; Ch2: Not Used

```
Main
;------
WaitForInput
      banksel PortB
      btfsc             PortB,0
      GoTo              WaitForInput
      NOP
```

Since there is no start bit, we jump to capturing the next 16 positive pulse widths after the first time the input pin goes low. In MBasic, our wait for input routine could be written as:

```
WaitHere
      If PortB.Bit0 = 1 Then GoTo WaitHere
      EndIf
```

The problem with a pure MBasic solution is that its execution time is marginal, given that the low time is a few hundred microseconds. Accordingly, we use a simple assembler routine that reads PortB and checks the status of bit 0 (B0) through the btfsc operation. If B0 is set (B0 = 1) the next instruction, GoTo Wait-ForInput is executed in an endless loop. If B0 is clear (B0 = 0), the GoTo WaitForInput instruction is skipped, a null (NOP) operation is executed and code execution returns to MBasic. The assembler test and branch code executes in less than 1μs with a 20 MHz clock.

We then read the duration of the next 16 high pulses using PulsIn. The most direct way to code this is:

```
For i = 0 to 15
      PulsIn B0,1,WArray(i)
Next
```

This construction, however, has an execution speed problem. It turns out that the combination of the For … Next loop overhead (162 μs) and PulsIn's release time (166 μs) following the input pulse falling edge total approximately 328 μs, assuming a 20 MHz clock. Since the remote control inter-pulse dwell interval can be as short as 305 μs, we must save time someplace. We could write our own alternative PulsIn procedure, perhaps using timer 1, but before introducing that complexity, we first try an substitute for the For...Next loop. By mixing a bit of assembler with the MBasic code we shave the loop overhead from 162 μs to 73 μs, for a total execution time of 249 μs, well within our 305 μs dwell interval.

```
Loop
      PulsIn B0,1,Main,1,WArray(i)
      ASM
      {
      banksel i
      incf i&0x7F,f
      banksel PortB
      }
      If i < 16 Then Loop
```

Our assembler routine is the equivalent of i=i+1 where i is a byte variable, except it executes in under 1 μs. We then capture the next pulse width in the word-length array Warray(i). If we have read fewer than 16 pulses, the if...then test recycles to PulsIn to read the next input pulse. The if...then test in MBasic executes in about 70 μs, so the combined assembler increment and if...then test executes in less than half the time of the for... next code it replaces.

After a 250 ms pause to avoid reading duplicated pulse sequences, we call the subroutine CalculateCode to classify each of the 16 measured pulse widths as wide, narrow or unknown.

```
CalculateCode
;------------
      ;Scan the measured width array and classify each as
      ;N (narrow), W (wide) or x (outside W and N windows)
      For i = 0 to 15
            TempArray(i) = "x"
            If (WArray(i) > WMin) AND (WArray(i) < WMax) Then
                  TempArray(i) = "W"

            EndIf

            If (WArray(i) > NMin) AND (WArray(i) < NMax) Then
                        TempArray(i) = "N"
            EndIf
      Next
Return
```

Since we wish to display the output data for debugging, we'll use the string array TempArray for two purposes. First, we will use it to hold the status of each pulse, W, N or x, for wide, narrow or unknown, respectively. This array is later sent to the serial port for display. Second, we'll use it to help compute the numerical value of the received 16 bits.

Using a For i …Next loop, we first assign TempArray(i) the value "x", or unknown. Then, we test the pulse length stored in the corresponding Warray(i) to see if it within our narrow or wide windows and we assign an appropriate letter—"N" or "W"—to TempArray(i) if it a verifiable narrow or wide pulse. If it isn't within either the wide or narrow windows, the classification stays as "x".

After completing CalculateCode, a typical TempArray looks like the following: WWWNWNWNNNNWNWNN if it has no errors, or, perhaps, WWWxWNWNNxNWNxNN if it contains pulses outside the W or N width limits. (Of course, the Ws and Ns change, depending on the key sequence pressed.)

Subroutine GenerateValue then calculates the word value of the 16 received bits.

```
GenerateValue
;------------
      CodeValue = 0
      GoodFlag = 1
      For i = 0 to 15
              CodeValue.Bit0 = %0 ;overwrite if W
              If TempArray(i) = "W" Then
                  CodeValue.Bit0 = %1
              EndIf
              If TempArray(i) = "x" Then
                  GoodFlag = 0
```

```
            EndIf
            If i <> 15 Then
                  CodeValue = CodeValue << 1
            EndIf
      Next
      CodeValueR.Byte0 = CodeValue.Byte0 REV 8
      CodeValueR.Byte1 = CodeValue.Byte1 REV 8
Return
```

The returned word value, assuming the data is sent MSB-first, is in CodeValue. However, since we don't know if Mitsubishi sends its data in LSB or MSB order, we'll also calculate and display the LSB-first value, to be held in CodeValueR. We'll judge which is the correct order based on how these values change as we press different keys on the remote.

Our approach is to loop through the string array TempArray and, based upon the value of the character, add either a %0 or a %1 at the last bit in CodeValue. After each pass through the loop, we shift the bits in CodeValue one place to the left with the shift-left (<<) operator. Any "x" character in TempArray, sets the GoodFlag to 0, indicating the received data is corrupt. Note that since the shift-left gets us ready for the next bit, we won't execute it for the last bit, as there are no further bits to add. Hence we test for i = 15 and, when true, omit the shift-left.

Finally, we set the two bytes of CodeValueR equal to CodeValue, but with reversed bit order.

```
If GoodFlag = 1 Then
      HSerOut ["Good: ",Str TempArray \16," MSB: ", |
      iHex CodeValue]
      HSerOut [" LSB: ",iHex CodeValueR,13]
ELSE
      ;Uncomment line below if want see bad data
      ;HSerOut ["Error: ",Str TempArray \16," MSB: ", |
      iHex CodeValue,13]
EndIf
```

The remainder of the main program loop displays the received data. If you wish to see corrupt data as well as good data, uncomment the `HserOut` statement in the `ELSE` portion of the `If` conditional. Subroutine `DumpRawData` provides the individual measured pulse durations.

Here's the output for some sample key presses. (To save space, I've suppressed the raw pulse length output data.) This particular remote control has a slide switch to select between TV and VCR, so we'll exercise both options. I've added the Mode and Key comments to Program 23.2's output and I've deleted duplicates, where each of the repeated messages automatically sent by the remote controller are correctly decoded and separately reported.

	Comments	
	Mode	**Key**
Good: WWWNWNWNNWNWNNNN MSB: $EA50 LSB: $570A	VCR	Ch+
Good: WWWNWNWNNWNWNNNN MSB: $EA48 LSB: $5712	VCR	Ch-
Good: WWWNNNWNNWNNNWNN MSB: $E244 LSB: $4722	VCR	Vol+
Good: WWWNNNWNNWNWNWNN MSB: $E254 LSB: $472A	VCR	Vol-
Good: WWWNWNWNWNNWNNNN MSB: $EA90 LSB: $5709	VCR	0
Good: WWWNWNWNNNNNNNNN MSB: $EA00 LSB: $5700	VCR	1
Good: WWWNWNWNNNNWNNNN MSB: $EA10 LSB: $5708	VCR	2
Good: WWWNWNWNNNNNWNNN MSB: $EA08 LSB: $5710	VCR	3
Good: WWWNWNWNNNNWWNNN MSB: $EA18 LSB: $5718	VCR	4
Good: WWWNWNWNNNNNNWNN MSB: $EA04 LSB: $5720	VCR	5
Good: WWWNWNWNNNNWNWNN MSB: $EA14 LSB: $5728	VCR	6
Good: WWWNWNWNNNNNWWNN MSB: $EA0C LSB: $5730	VCR	7
Good: WWWNWNWNNNNWWWNN MSB: $EA1C LSB: $5738	VCR	8
Good: WWWNWNWNWNNNNNNN MSB: $EA80 LSB: $5701	VCR	9
Good: WWWNNNWNNWNWNNNN MSB: $E250 LSB: $470A	TV	Ch+
Good: WWWNNNWNNWNWNNNN MSB: $E248 LSB: $4712	TV	Ch-
Good: WWWNNNWNNWNNNWNN MSB: $E244 LSB: $4722	TV	Vol+
Good: WWWNNNWNNWNWNWNN MSB: $E254 LSB: $472A	TV	Vol-
Good: WWWNNNWNWNNWNNNN MSB: $E290 LSB: $4709	TV	0
Good: WWWNNNWNNNNNNNNN MSB: $E200 LSB: $4700	TV	1
Good: WWWNNNWNNNNWNNNN MSB: $E210 LSB: $4708	TV	2
Good: WWWNNNWNNNNNWNNN MSB: $E208 LSB: $4710	TV	3
Good: WWWNNNWNNNNWWNNN MSB: $E218 LSB: $4718	TV	4
Good: WWWNNNWNNNNNNWNN MSB: $E204 LSB: $4720	TV	5
Good: WWWNNNWNNNNWNWNN MSB: $E214 LSB: $4728	TV	6
Good: WWWNNNWNNNNNWWNN MSB: $E20C LSB: $4730	TV	7
Good: WWWNNNWNNNNWWWNN MSB: $E21C LSB: $4738	TV	8
Good: WWWNNNWNWNNNNNNN MSB: $E280 LSB: $4701	TV	9

Before deciding the most likely correct bit order, we should understand that it really doesn't matter in most cases whether we have it right or not. If we wish to use the decoded remote control to increase or decrease some value based on the TV/Channel+ or TV/Channel– key, it matters not whether we increase upon decoding `$E250` or `$470A`. All we are concerned with is matching a unique value, and that value can be the theoretically correct bit order, or its reverse; both values are unique and either serves our purpose. However, as an exercise, let's see if we can determine the correct order.

Before we decide the bit order, it appears that the high byte is a device code, where `$E2/$47` precedes control messages to the TV, and `$EA/$57` precedes messages to the VCR. If so, the low byte makes sense as a key code independent of the device address. For example, the low byte value `$90/$09` always represents the digit 0 key, regardless of whether the command is directed at the TV or the VCR. (The dual values separated by a / indicate both MSB-first and LSB-first values.)

To choose between LSB-first and MSB-first bit order, however, is more difficult. Examining the values sent when the 0…9 keys are pressed, we note that LSB-first order has fewer large value jumps than MSB-first order (the MSB-first order values associated with the 9 and 0 keys particularly stand out as unusual). On balance, it's likely the data is sent LSB first.

Finally, we also observe that, at least for the samples shown above, all key codes start with three consecutive Ws (1's) and all end with two consecutive N's (0's). It's a reasonable guess—logical, but still a guess—that these consecutive characters represent message start and message end sentinels. If so, the variable message content would be 11 bits and our view of LSB-first data order might change.

23.4 Decoding a REC-80 Controller

Let's now look at reading a true REC-80 compatible controller. Program 23.3 builds upon Program 23.2, so we'll concentrate on the differences.

```
;Program 23-3
;IR decoder output
;to Pin B0. Values for Generic
;remote control 32-bit width

;Constants
;-------------

                ;unit duration @ 20MHz clock
Smin    Con     2500        ;3500us start pulse minimum
Smax    Con     3572        ;5000us start pulse maximum
Wmin    Con     964         ;1350us Wide pulse min width
WMax    Con     1393        ;1950us Wide pulse max width
NMin    Con     214         ;300us Narrow pulse min width
NMax    Con     571         ;800us Narrow Pulse max width
MLen    Con     32          ;Number of data bits

;Variables
;-----------

CodeValue     Var    Long          ;holds decoded 32 bits LSB order
Temp          Var    Long          ;decoded bits MSB order
TempArray     Var    Byte(MLen)    ;Holds N/W/X strings
i             Var    Byte          ;Counter
WArray        Var    Word(MLen)    ;Holds pulse width measurements
GoodFlag      Var    Byte          ;If 1, good data
```

Program 23.3

```
;Initialization
;-------------
Clear               ;Zeroize
Input B0            ;Output of IR Receiver to B0
EnableHSerial
SetHSerial H115200
HSerOut ["P23.3",13,13]

Main
      i=0
      ;We first wait for start pulse
      PulsIn B0,1,Main,1,WArray(0)

      ;Is it a valid start pulse?
      If Warray(0) < Smin Then
            GoTo Main
      EndIf

      If Warray(0) > Smax Then
            GoTo Main
      EndIf

      ;Valid start pulse is followed by MLen data pulses
      Loop
            PulsIn B0,1,Main,1,WArray(i)
            ASM
            {
            banksel i
            incf i&0x7F,f
            banksel PortB
            }

      If i < MLen Then Loop
      GoSub CalculateCode
      GoSub GenerateValue

      If GoodFlag = 1 Then
            GoSub ErrorCheck
            If GoodFlag = 1 Then
                  HSerOut ["Good: ",Str TempArray \MLen, |
                  " MSB: ",iHex Temp]
                  HSerOut [" LSB: ",iHex CodeValue,13]
            EndIf
      ELSE
            ;Uncomment line below if want bad data output |
            ;HSerOut ["Error: ",Str TempArray \MLen,
            " MSB: ",iHex Temp,13]
      EndIf
```

Program 23.3: Continued

```
GoTo Main
CalculateCode
;-----------
      ;Scan the measured width array and classify each
      ;as N (narrow), W (wide) or X (outside N and W windows)
      For i = 0 to (MLen-1)
            TempArray(i) = "x" ;pre-load with bad flag
            If (WArray(i) > WMin) AND (WArray(i) < WMax) Then
                  TempArray(i) = "W"
            EndIf

            If (WArray(i) > NMin) AND (WArray(i) < NMax) Then
                  TempArray(i) = "N"
            EndIf
      Next
Return

GenerateValue
;-----------
      ;Now calculate the long=to the received 32-bits
      Temp = 0
      ;Set to 0 if get bad data
      GoodFlag = 1
      ;Now convert W/N/x to values
      For i = 0 to (MLen-1)
            Temp.Bit0 = %0
            If TempArray(i) = "W" Then
                  Temp.Bit0 = %1
            EndIf
            If TempArray(i) = "x" Then
                  GoodFlag = 0
            EndIf
            ;No need to check N or x, since pre-populated with 0s
            ;read assuming sent MSB first. This may not
            ;be the case, but for our purpose order doesn't
            ;matter as we only look for unique values
            ;Data in at right, move to left every entry w/ new
            ;data to follow. Skip at the end
            If i <> (MLen-1) Then
                  Temp = Temp << 1
            EndIf
      Next
      ;Some infor says data sent LSB first
      ;byte order is OK, but bits are reversed
      CodeValue.Byte3 = Temp.Byte3 REV 8
      CodeValue.Byte2 = Temp.Byte2 REV 8
```

Program 23.3: Continued

```
      CodeValue.Byte1 = Temp.Byte1 REV 8
      CodeValue.Byte0 = Temp.Byte0 REV 8
Return

ErrorCheck        ;check NOT bytes
;--------
      If (CodeValue.Byte3 + CodeValue.Byte2) <> $FF Then
            GoodFlag = 0
            HSerOut ["Bad Check Byte 3",13]
      EndIf
      If (CodeValue.Byte1 + CodeValue.Byte0) <> $FF Then
            GoodFlag = 0
            ;HSerOut ["Bad Check Byte 1",13]
      EndIf
Return
End
```

Program 23.3: Continued

First, we make the code more flexible and maintainable by defining the bit length with a constant:

```
MLen    Con         32
```

And, since the data length is 32 bits, not 16, we've changed the variables `CodeValue` and `Temp` to be type `long`, not `word`. Finally, in an attempt to cover a wide range of remote controllers, we expanded the wide and narrow window constants to match Fig. 23.10.

We noted that Mitsubishi's 16-bit protocol has no start pulse, but REC-80 compatible remote controls have a start pulse. We'll accordingly modify the start conditions for Program 23.3.

```
;We first wait for start pulse
PulsIn B0,1,Main,1,WArray(0)

;Is it a valid start pulse?
If Warray(0) < Smin Then
      GoTo Main
EndIf

If Warray(0) > Smax Then
      GoTo Main
EndIf
```

We start by measuring the positive pulse width using `PulsIn,` storing the value in `Warray(0)`. Unlike in Program 23.2, however, we now define the `TimeoutMultiple` and `TimeoutLabel`. We'll set the `TimeoutMultiple` as 1, and use `Main` as the `TimeoutLabel`. Thus, if a positive pulse isn't received within 1.4 μs × 65,535 (91.7 ms) program execution jumps back to `Main`, where `PulsIn` is again executed. Consequently, we are in a continual

loop, waiting for a positive pulse to arrive. When a positive pulse arrives, we save its value in `Warray(0)` and check it against the window we've defined for the start pulse, `Smin` and `Smax`. If we have received a valid start pulse, execution continues and we read the width of the next 32 data bits into `Warray`, starting with `Warray(0)`. If the possible start pulse is outside the `Smin...Smax` window, program execution returns to `Main` and we again wait for a possible start pulse.

I measured the REC-80 inter-pulse dwell time at 600μs so we have sufficient time to perform these comparisons in MBasic. In fact, we could use a for…next loop to read the pulse widths. However, since our mixed assembler/MBasic code works, we'll stick with it.

After the this start related code, the remaining portion of the main program segment has only a small change related to error checking, which we'll get to shortly. The two subroutines `CalculateCode` and `GenerateValue` are identical with their namesakes in Program 23.2, save for substituting `Mlen` for the hard coded bit length. Consequently, we'll limit our analysis to the error checking subroutine, `ErrorCheck`.

As we noted in our discussion of REC-80 code format, data is sent in the form

Typical REC-80 Data Organization Sending Order							
Byte 3		Byte 2		Byte 1		Byte 0	
Device Code		D.C. Check Byte		Function Code		F.C. Check Byte	
LSB	MSB	LSB	MSB	LSB	MSB	LSB	MSB

An REC-80 check byte is formed by taking the logical NOT of the associated data byte—that is, 1's are changed to 0's and vice versa. Consequently. if we look at the data byte and its associated check byte, for every corresponding bit position, one byte has a `%0` and the other byte has a `%1`. Thus, if we add the data byte and its check byte, we always get `%11111111` or `$FF`.

We'll accordingly implement a further error check by seeing if `CodeValue.Byte3 + CodeValue.Byte2 = $FF` and if `CodeValue.Byte1 + CodeValue.Byte0 = $FF`. The new subroutine `ErrorCheck` performs this action, and upon an error the `GoodFlag` is cleared, indicating corrupted data and a brief error message is emitted over the serial port. To avoid order of precedence problems with some MBasic versions, I've used parentheses to force the `CodeVal.Byte3 + CodeValue.Byte2` and `CodeVal.Byte1 + CodeVal.Byte0` expressions to be evaluated before the < > comparison.

```
ErrorCheck ;check NOT bytes
;--------
      If (CodeValue.Byte3 + CodeValue.Byte2) <> $FF Then
            GoodFlag = 0
            HSerOut ["Bad Check Byte 3",13]
      EndIf
```

```
      If (CodeValue.Byte1 + CodeValue.Byte0) <> $FF Then
            GoodFlag = 0
            ;HSerOut ["Bad Check Byte 1",13]
      EndIf
Return
```

Let's look at the output of Program 23.3. Bold face indicates annotations I've added to the output. I've also added leading zeros to the Hitachi MSB values.

```
Nakamichi RM-2CDP Remote Control                                    Key
P22-3
Good: WWNNNWWNNNNNWWNNWWNWNNNWWNWNWWWNN   MSB: $E619A35C LSB: $6798C53A Stop
Good: WWNNNWWNNNNNWWNNWWWNWNNNWWNNWWNNN   MSB: $E619D32C LSB: $6798CB34 Pause
Good: WWNNNWWNNNNNWWNNWNWWNNNWWWNNNWWNN   MSB: $E619738C LSB: $6798CE31 0
Good: WWNNNWWNNNNNWWNNWWWWNNNWWWWWWWNN   MSB: $E619F30C LSB: $6798CF30 1
Good: WWNNNWWNNNNNWWNNWNNNNWNWWWWWWWNN   MSB: $E6190BF4 LSB: $6798D02F 2
Good: WWNNNWWNNNNNWWNNWNNNWNWWNWWWWNN   MSB: $E6198B74 LSB: $6798D12E 3
Good: WWNNNWWNNNNNWWNNWNNNWNWWWNWWWNN   MSB: $E6194BB4 LSB: $6798D22D 4
Good: WWNNNWWNNNNNWWNNWWWNNNWWWNWWWNN   MSB: $E619CB34 LSB: $6798D32C 5
Good: WWNNNWWNNNNNWWNNWNNNWNWWWWWNWNN   MSB: $E6192BD4 LSB: $6798D42B 6
Good: WWNNNWWNNNNNWWNNWNWNWNWWNWNWNWNN   MSB: $E619AB54 LSB: $6798D52A 7
Good: WWNNNWWNNNNNWWNNWNWWNWWWWNNWNWN   MSB: $E6196B94 LSB: $6798D629 8
Good: WWNNNWWNNNNNWWNNWWWWNWNWWNNNWNWN   MSB: $E619EB14 LSB: $6798D728 9

Nakamichi RM-4TA Remote Control                                     Key
P22-3
Good: WWNNNWWNNNNNWWNNWWNWNNNWWNWNWWWNN   MSB: $E619A35C LSB: $6798C53A Stop
Good: WWNNNWWNNNNNWWNNWWWNWNNNWWNNWWNNN   MSB: $E619D32C LSB: $6798CB34 Pause
Good: WWNNNWWNNNNNWWNNWWWWNNNWWWNNNWWNN   MSB: $E619E31C LSB: $6798C738 >>
Good: WWNNNWWNNNNNWWNNWWNNNNNWWNWWWWNN   MSB: $E619837C LSB: $6798C13E <<
Good: WWNNNWWNNNNNWWNNWNNNNWNWWWWNWWNN   MSB: $E61913EC LSB: $6798C837 >|
Good: WWNNNWWNNNNNWWNNWNWNNNNWWWNWWWNN   MSB: $E61943BC LSB: $6798C23D |<

Toshiba SE-R0108 Remote Control VCR/DVD                             Key
P22-3
Good: WNWNNNWNNWNWWWNWNWNNWNNNNWNWWNWW   MSB: $A25D48B7 LSB: $45BA12ED Pwr
Good: WNWNNNWNNWNWWWNWNNNNNNWWWWWWWNNN   MSB: $A25D07F8 LSB: $45BAE01F VCR/DVD
Good: WNWNNNWNNWNWWWNWNWNWNNNNNWNWNWWW   MSB: $A25D50AF LSB: $45BA0AF5 0
Good: WNWNNNWNNWNWWWNWNWNNNNNNNNWWWWWW   MSB: $A25D807F LSB: $45BA01FE 1
Good: WNWNNNWNNWNWWWNWNWNNNNNNNWNWWWWW   MSB: $A25D40BF LSB: $45BA02FD 2
Good: WNWNNNWNNWNWWWNWNWWWNNNNNNNNWWWW   MSB: $A25DC03F LSB: $45BA03FC 3
Good: WNWNNNWNNWNWWWNWNNNWNNNNNNWNWWWW   MSB: $A25D20DF LSB: $45BA04FB 4
Good: WNWNNNWNNWNWWWNWNWNWNNNNNNWNWWWW   MSB: $A25DA05F LSB: $45BA05FA 5
Good: WNWNNNWNNWNWWWNWNWNWNNNNNNWNWWWW   MSB: $A25D609F LSB: $45BA06F9 6
```

```
Good: WNWNNNWNNWNWWWNWWWWNNNNNNNNNWWWWW MSB: $A25DE01F LSB: $45BA07F8 7
Good: WNWNNNWNNWNWWWNWNNNWNNNNNWWWNWWW MSB: $A25D10EF LSB: $45BA08F7 8
Good: WNWNNNWNNWNWWWNWWNNWNNNNNNWWNWWW MSB: $A25D906F LSB: $45BA09F6 9
Good: WNWNNNWNNWNWWWNWNNWNWNNNNNWWNWWW MSB: $A25D28D7 LSB: $45BA14EB Stop
Good: WNWNNNWNNWNWWWNWWWNNWNNNNNNWWNWW MSB: $A25DC837 LSB: $45BA13EC >>
Good: WNWNNNWNNWNWWWNWWNNWWNNNNNWWNWWW MSB: $A25D9867 LSB: $45BA19E6 <<
Good: WNWNNNWNNWNWWWNWNNNNNNNNNWWWWWW MSB: $A25D00FF LSB: $45BA00FF Pause
```

Hitachi "Remote Controller" **Key**

```
P22-3
Good: NNNNNWWNWWWWWNNWWWWNWNNNNNNWNWWW MSB: $06F9E817 LSB: $609F17E8 Pwr
Good: NNNNNWWNWWWWWNNWNWWNWNNNWNNWNWWW MSB: $06F96897 LSB: $609F16E9 VTR/TV
Good: NNNNNWWNWWWWWNNWWNNWWNNNNWWNNWWW MSB: $06F99867 LSB: $609F19E6 Ch+
Good: NNNNNWWNWWWWWNNWNNNWWNNNNWWWNWWW MSB: $06F918E7 LSB: $609F18E7 Ch-
Good: NNNNNWWNWWWWWNNWNNWNWNNNNWWNWWW MSB: $06F928D7 LSB: $609F14EB Play
Good: WNNNNWWNNWWWWNNWNNWNWNNNNWWNWWW MSB: $867928D7 LSB: $619E14EB Rec+Play
Good: NNNNNWWNWWWWWNNWWNNWNNNNNWWNWWW MSB: $06F9906F LSB: $609F09F6 >>
Good: NNNNNWWNWWWWWNNWNWNWNNNNNWNWWWW MSB: $06F950AF LSB: $609F0AF5 <<
Good: NNNNNWWNWWWWWNNWWWNWNNNNNNWNWWW MSB: $06F9D02F LSB: $609F0BF4 Stop
Good: NNNNNWWNWWWWWNNWNWNWNWNNNWNWNWW MSB: $06F958A7 LSB: $609F1AE5 Pause
```

None of these data reads caused a check byte error. And, that's not unexpected, as the fact that we've received 32 sequential correct width bits provides a high probability that the bits are correct. My limited test showed no 16-bit device and data codes, and no toggle bits.

If we check the MSB-first and LSB-first values against the key functions, the LSB-first values make more sense, so these manufactures have faithfully followed the REC-80 protocol. And, they seem to have made logical key assignments as well.

Numeric Key	Toshiba Key Code	Nakamichi Key Code
0	$0A	$CE
1	$01	$CF
2	$02	$D0
3	$03	$D1
4	$04	$D2
5	$05	$D3
6	$06	$D4
7	$07	$D5
8	$08	$D6
9	$09	$D7

Toshiba encodes the key value directly (key 0 is assigned value 10, or $0A), while Nakamichi's codes track the key value as well, but with an offset of $CE.

Let's use the decoded output to actually do something. We'll only turn LEDs on and off, but the techniques we've developed in other chapters permit you to extend the concepts in Program 23.4 to control other devices. For example, you might emit SPI control signals to an MCP42010 digital potentiometer and have the output of a stereo audio signal increase or decrease in 1 dB steps in response to the Vol+ and Vol– buttons.

I've written Program 23.4 to function with a Hitachi controller, chiefly because it belongs to a no-longerfunctional VCR and isn't needed elsewhere in my house. Of course, you may substitute any other codes. And, to show that bit order isn't important if—as is usually the case—we are only looking unique key codes, I've intentionally written Program 23.4 with MSB-first bit order.

Pressing the Play, Stop, Pause or Power buttons causes an LED connected to the PIC to latch on or off. Each key press reverses the status. Figure 23.12 shows the LED connection.

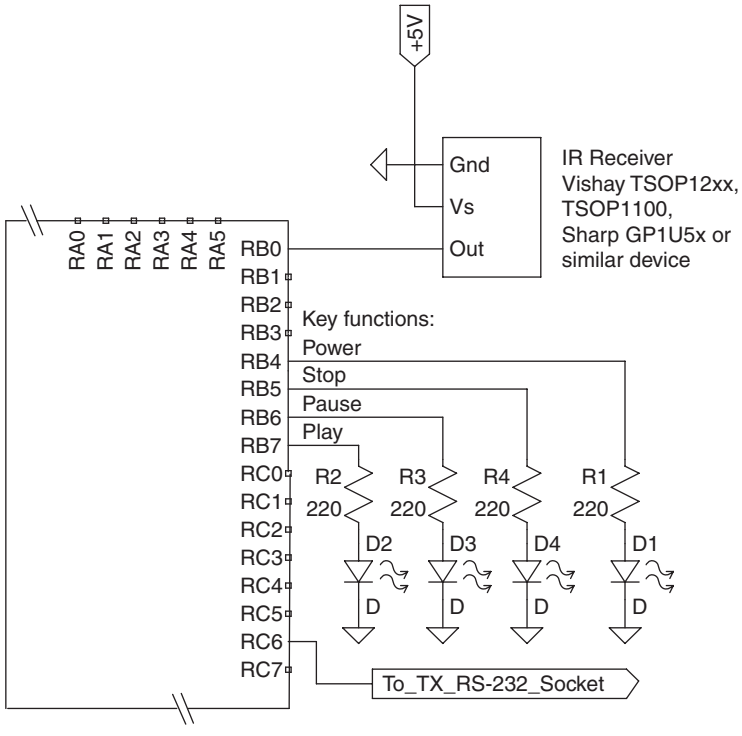


Figure 23.12: Controlling Four LEDs with an IR Remote Control

```
;Program 23-4
;IR decoder output
;to Pin B0. Values for Generic
;remote control 32-bit width

;Constants
;-------------

             ;unit duration @ 20MHz clock
Smin         Con       2500    ;3500us start pulse minimum
Smax         Con       3572    ;5000us start pulse maximum
Wmin         Con       964     ;1350us Wide pulse min width
WMax         Con       1393    ;1950us Wide pulse max width
NMin         Con       214     ;300us Narrow pulse min width
NMax         Con       571     ;800us Narrow Pulse max width
MLen         Con       32      ;Number of data bits
PowerPin     Con       B4      ;LED on PowerPin
StopPin      Con       B5      ;LED on StopPin
PausePin     Con       B6      ;LED on PausePin
PlayPin      Con       B7      ;LED on PlayPin

;Decode Hitachi in MSB order
Power        Con       $06F9E817       ;Power button
StopD        Con       $06F9D02F       ;Stop button
PauseP       Con       $06F958A7       ;Pause button
Play         Con       $06F928D7       ;Play button

;Variables
;-----------
CodeValue    Var       Long            ;holds decoded 32 bits
TempArray    Var       Byte(MLen)      ;holds measured widths
i            Var       Byte            ;counter
WArray       Var       Word(MLen)      ;for display
GoodFlag     Var       Byte            ;If 1, good data
FcnStr       Var       Byte(6)

;Initialization
;-------------
Clear
Input B0
;Have LEDs on B4...B7 pins
For i = B4 to B7
      Low i
Next
EnableHSerial
SetHSerial H115200
HSerOut ["P22-4",13,13]
```

Program 23.4

```
Main
;-----
      i=0
      ;We first wait for start pulse
      PulsIn B0,1,Main,1,WArray(0)

      ;Is it a valid start pulse?
      If Warray(0) < Smin Then
            GoTo Main
      EndIf
      If Warray(0) > Smax Then
            GoTo Main
      EndIf

      ;Valid start pulse followed by 32 data pulses
      Loop
            PulsIn B0,1,Main,1,WArray(i)
            ASM
            {
            banksel i
            incf  i&0x7F,f
            banksel PortB
            }
      If i < MLen Then Loop

      GoSub CalculateCode
      GoSub GenerateValue
      If GoodFlag = 1 Then
            GoSub ErrorCheck
      EndIf

      If GoodFlag = 1 Then
            Pause 100 ;got valid code this time
            FcnStr = "Unkwn "
            ;Now we toggle pins based on received codes
            If CodeValue = Power Then
                  Toggle PowerPin
                  FcnStr = "PWR "
            EndIf

            If CodeValue = StopD Then
                  Toggle StopPin
                  FcnStr = "Stop "
            EndIf
```

Program 23.4: Continued

```
            If CodeValue = PauseP Then
                  Toggle PausePin
                  FcnStr = "Pause "
            EndIf

            If CodeValue = Play Then
                  FcnStr = "Play "
                  Toggle PlayPin
            EndIf

            HSerOut [Str FcnStr\6, Str TempArray \MLen, |
            " MSB: ",iHex CodeValue,13]
      ELSE
            ;Remove comments if you want to see bad data
            ;HSerOut ["Error: ",Str TempArray \MLen, |
            " Value: ",iHex CodeValue,13]
      EndIf

GoTo Main

CalculateCode
;-----------
      ;Scan the measured width array and classify each as
      ;N (narrow), W (wide) or X (outside N or W window)
      For i = 0 to (MLen-1)
            TempArray(i) = "x"
            If (WArray(i) > WMin) AND (WArray(i) < WMax) Then
                  TempArray(i) = "W"
            EndIf
                  If (WArray(i) > NMin) AND (WArray(i) < NMax) Then
                        TempArray(i) = "N"
                  EndIf
      Next
Return

GenerateValue
;-----------
      ;Now calculate the Long=to the 32 bits
      CodeValue = 0
      ;set to 0 if we get bad data
      GoodFlag = 1
      ;Now convert W/N/X to values
      For i = 0 to (MLen-1)
            CodeValue.Bit0 = %0
            If TempArray(i) = "W" Then
                  CodeValue.Bit0 = %1
            EndIf
```

Program 23.4: Continued

```
            If TempArray(i) = "x" Then
                  GoodFlag = 0 ;if don't want bad data
            EndIf                        ;debug could bail out here
            ;No need to check N or x, pre-populated with 0s
            ;read assuming sent MSB first. This may not
            ;be the case, but for our purpose order doesn't
            ;matter as we only look for unique values
            ;Data goes in at right, move to left
            ;every entry except last one
            If i <> (MLen-1) Then
                  CodeValue = CodeValue << 1
            EndIf
      Next
Return
ErrorCheck        ;check NOT bytes
;--------
      If (CodeValue.Byte3 + CodeValue.Byte2) <> $FF Then
            GoodFlag = 0
            HSerOut ["Bad Check Byte 3",13]
      EndIf

      If (CodeValue.Byte1 + CodeValue.Byte0) <> $FF Then
            GoodFlag = 0
            HSerOut ["Bad Check Byte 1",13]
      EndIf
Return
End
```

Program 23.4: Continued

Since Program 23.4 duplicates much of Program 23.3, we'll concentrate on the differences.

```
PowerPin      Con      B4      ;LED on PowerPin
StopPin       Con      B5      ;LED on StopPin
PausePin      Con      B6      ;LED on PausePin
PlayPin       Con      B7      ;LED on PlayPin

;Decode Hitachi in MSB order
Power         Con      $06F9E817       ;Power button
StopD         Con      $06F9D02F       ;Stop button
PauseP        Con      $06F958A7       ;Pause button
Play          Con      $06F928D7       ;Play button
```

We've added several new constant definitions. We've aliased the pins `B4 . . . B7` to more understandable names, and defined the four MSB-order key values we discovered in the output of Program 23.3.

```
If GoodFlag = 1 Then
      GoSub ErrorCheck
EndIf

If GoodFlag = 1 Then
Pause 100   ;got valid code this time
      FcnStr = "Unkwn "
      ;Now we toggle pins based on received codes
      If CodeValue = Power Then
      Toggle PowerPin
      FcnStr = "PWR "
EndIf

If CodeValue = StopD Then
      Toggle StopPin
      FcnStr = "Stop "
EndIf
If CodeValue = PauseP Then
      Toggle PausePin
      FcnStr = "Pause "
EndIf

If CodeValue = Play Then
      FcnStr = "Play "
      Toggle PlayPin
EndIf

HSerOut [Str FcnStr\6, Str TempArray \MLen, |
" MSB: ",iHex CodeValue,13]
ELSE
;Remove comments if you want to see bad data
;HSerOut ["Error: ",Str TempArray \MLen,    |
" Value: ",iHex CodeValue,13]
EndIf
```

The individual pin-set/clear code is not complex. If we have received a valid 32-bit remote control signal, we test its value against the four "action" values. If an action value is received, we toggle the corresponding output pin, and define an output message string with the name of function. To avoid interpreting the built-in automatic repeat as repeated individual button presses, we've added a `Pause 100` delay statement after a successful decode.

We have slightly simplified the serial output routine to omit the LSB-first order value, but added the function identifying string. Here's a sample output from Program 23.4:

```
P22-4
PWR   NNNNNWWNWWWWWNNWWWWNWNNNNNNWNWWW  MSB:  $6F9E817
PWR   NNNNNWWNWWWWWNNWWWWNWNNNNNNWNWWW  MSB:  $6F9E817
Play  NNNNNWWNWWWWWNNWNNWNWNNNWWNWNWW  MSB:  $6F928D7
```

```
Play  NNNNNWWNWWWWWNNWNNWNWNNNWWNWNWWW  MSB:  $6F928D7
Stop  NNNNNWWNWWWWWNNWWWNWNNNNNNNWNWWW  MSB:  $6F9D02F
Stop  NNNNNWWNWWWWWNNWWWNWNNNNNNNWNWWW  MSB:  $6F9D02F
Pause NNNNNWWNWWWWWNNWNWNWWNNNWNWNNWW  MSB:  $6F958A7
Pause NNNNNWWNWWWWWNNWNWNWWNNNWNWNNWW  MSB:  $6F958A7
Unkwn NNNNNWWNWWWWWNNWWNNNWNNNNNWWNWW  MSB:  $6F9906F
Unkwn NNNNNWWNWWWWWNNWNWNWNNNNWNWNWW  MSB:  $6F950AF
Unkwn NNNNNWWNWWWWWNNWNWWNWNNNWNNWNW  MSB:  $6F96897
Unkwn NNNNNWWNWWWWWNNWWNNWWNNNNWWNNW  MSB:  $6F99867
Unkwn NNNNNWWNWWWWWNNWNNNNWWNNNWWNNW  MSB:  $6F918E7
Play  NNNNNWWNWWWWWNNWNNWNWNNNWWNWNW  MSB:  $6F928D7
Play  NNNNNWWNWWWWWNNWNNWNWNNNWWNWNW  MSB:  $6F928D7
```

The four function output pins are initialized low, so the first time a function key is decoded, the associated LED illuminates. The next time, it is extinguished. Toggling continues with additional key presses. Receipt of a code other than one associated with the four defined keys triggers an "unkwn" message and the status of the four LED pins is unchanged.

I tested a breadboard version of Fig. 23.11 with both a Sharp GP1U58x IR receiver module and a Vishay TSOP1238 integrated IR receiver module. Both modules are tuned to 38 KHz. I found reliable decoding at 30 feet with the GP1U58x receiver and at 36 feet with the TSOP1238 receiver. I tested the GP1U58x without an IR filter, but the TSOP1238 has an integrated IR filter.

Finally, of course the serial output data is intended to help developing, debugging and testing the program. It may be omitted if unnecessary.

23.5 Ideas for Modifications to Programs and Circuits

- Modify Program 23.4 to control an MCP42010 digital potentiometer using the circuitry and concepts developed in Chapter 21. With each press of the Vol+ button the MCP42010 should step up 1 dB; with each press of the Vol- button the MCP42010 should step down 1 dB. After each step change, store the current position in EEPROM. Upon program initiation, read the old position from EEPROM and set the MCP42010's value to the stored position.
- What else might you wish to control remotely? How would you go about it?
- Chapter 22 concentrates on receiving remote control signals. But, it's not difficult to transmit an REC80 signal with a PIC. As a starting point, develop a simple "repeater" or range extender. After a correct signal is received—one that passes the check byte test—replay the signal to an IR diode. Here's a pseudo-code outline of how this might be accomplished:

```
Send a start pulse:
      LED-on for 9.5ms,
      LED-off for 4.5ms
```

```
Step through the array TempArray(i)
      Send LED-on for 800uS
      If TempArray(i) = W, pause for 1600uS
      If TempArray(i) = N, pause for 600uS
```

- Going beyond repeating a received remote control, use the RS-232 port to generate selected transmitted control codes. For example, a received + might generate a Vol+ IR command; a – might generate a Vol-IR command.

References

[23.1] Microchip Technologies, Inc., *Decoding Infrared Remote Controls Using a PIC16C5X Microcontroller*, AN657, Document No. DS00657A (1997).

[23.2] Sonmez, Mehmet Z., *Infrared Learner (Remote Control)*, Cypress MicroSystems Application Note AN2092, Rev. A (2002).

[23.3] Archer (Radio Shack), Technical *Data for Catalog No. 276-137*, (undated). Brief data sheet on the Sharp GP1U52X IR decoder modules.

[23.4] Vishay Semiconductors, *TSOP1100 WideBand IR Receiver Module for Remote Control Systems*, Doc. No. 82262, Rev 1.3 (2003).

[23.5] Vishay Semiconductors, TSOP12.. *IR Receiver Modules for Remote Control Systems*, Doc. No. 82013, Rev 11 (2003).

[23.6] Vishay Semiconductors, *TSOP48.. IR Receiver Modules for Remote Control Systems*, Doc. No. 82013, Rev 11 (2003).

[23.7] Sharp Electronics Components Group, GP1UD26XK Series/ GP1UD27XK Series GP1UD28XK Series / GP1UD28YK Series Energy Saving Type Low Dissipation Current IR Detecting Unit for Remote Control, (undated).

[23.8] An excellent starting point for IR remote control Internet research is http://www.epanorama.net/links/irremote.html. In addition to a brief explanation of the underlying technology, this site has dozens of useful cross-links to other sites with useful IR information.

[23.9] Innotech Systems, Inc., A *Primer on Remote Control Technology*, (undated).

[23.10] Seerden, Paul, *Using the Philips 87LPC76x microcontroller as a remote control transmitter*, AN10210, Philips Semiconductors (2003).

SECTION V

Programming PIC Microcontrollers Using C

www.newnespress.com

This page intentionally left blank

CHAPTER 24

Getting Started

24.1 The Plan

This is going to be our first C language project with the PIC24 16-bit microcontroller and, for some of you, the first project with the MPLAB® IDE Integrated Development Environment and the MPLAB C30 language suite. Even if you have never heard of the C language before, you might have heard of the famous "Hello World!" programming example. If not, let me tell you about it.

Since the very first book on the C language, written by Kernighan and Ritchie several decades ago, every decent C-language book has featured an example program containing a single statement to display the words "Hello World" on the computer screen. Hundreds, if not thousands, of books have respected this tradition, and I don't want this book to be the exception. However, it will have to be just a little different. Let's be realistic—we are talking about programming microcontrollers because we want to design embedded-control applications. While the availability of a monitor screen is a perfectly safe assumption for any personal computer or workstation, this is definitely not the case in the embedded-control world. For our first embedded application, we better stick to a more basic type of output—a digital I/O pin. In a later and more advanced chapter, we will be able to interface to an LCD display and/or a terminal connected to a serial port. But by then we will have better things to do than writing "Hello World!"

24.2 Checklist

Let's verify we have all the necessary pieces of equipment ready and installed (from the attached CD-ROM and/or the latest version available for download from Microchip's web site at *http://www.microchip.com/mplab*):

- MPLAB IDE, free Integrated Development Environment
- MPLAB SIM, software simulator
- MPLAB C30, C compiler (free Student Version).

Then, let's follow the "New Project Set-up" checklist to create a new project with the MPLAB IDE:

1. Select "Project→Project Wizard" to activate the new project wizard, which will guide us automatically through the following steps…
2. Select the PIC24FJ128GA010 device, and click Next.
3. Select the MPLAB C30 Compiler Suite and click Next.
4. Create a new folder and name it "Hello"; name the project "Hello Embedded World" and click Next.
5. Simply click Next to the following dialog box—there is no need to copy any source files from any previous projects or directories.
6. Click on Finish to complete the Wizard set-up.

For this first time, let's continue with the following additional steps:

7. Open a new editor window.
8. Type the following three comment lines:

```
//
//    Hello  Embedded  World!
//
```

9. Select "File→Save As", to save the file as: `"Hello.c"`.
10. Select "Project→Save" to save the project.

24.3 Coding

It is time to start writing some code. I can see your trepidation, especially if you have never written any C code for an embedded-control application before. Our first line of code is going to be:

```
#include <p24fj128ga010.h>
```

This is not yet a proper C statement, but more of a pseudo-instruction for the preprocessor telling the compiler to read the content of a device-specific file before proceeding any further. The content of the device-specific `".h"` file chosen is nothing more than a long list of the names (and sizes) of all the internal special-function registers (SFRs) of the chosen PIC24 model. If the include file is accurate, those names reflect exactly those being used in the device datasheet. If any doubt, just open the file and take a look—it is a simple text file you can open

with the MPLAB editor. Here is a segment of the `p24fj128ga010.h` file where the program counter and a few other special-function registers (SFRs) are defined:

```
...
extern   volatile   unsigned   int    PCL __attribute__((__sfr__));
extern   volatile   unsigned   char   PCH __attribute__((__sfr__));
extern   volatile   unsigned   char   TBLPAG __attribute__((__sfr__));
extern   volatile   unsigned   char   PSVPAG __attribute__((__sfr__));
extern   volatile   unsigned   int    RCOUNT __attribute__((__sfr__));
extern   volatile   unsigned   int    SR __attribute__((__sfr__));
...
```

Going back to our `"Hello.c"` source file, let's add a couple more lines that will introduce you to the `main()` function:

```
main()
{
}
```

What we have now is already a complete, although still empty and pretty useless, C-language program. In between those two curly brackets is where we will soon put the first few instructions of our embedded-control application.

Independently of this function position in the file, whether in the first lines on top or the last few lines in a million-line file, the `main()` function is the place where the microcontroller (program counter) will go first at power-up or after each subsequent reset.

One caveat—before entering the `main()` function, the microcontroller will execute a short initialization code segment automatically inserted by the linker. This is known as the `c0` code. The `c0` code will perform basic housekeeping chores, including the initialization of the microcontroller stack, among other things.

We said our mission was to turn on one or more I/O pins: say PORTA, pins RA0–7. In assembly, we would have used a pair of `mov` instructions to transfer a literal value to the output port. In C it is much easier—we can write an "assignment statement" as in the following example:

```
#include <p24fj128ga010.h>

main()
{
    PORTA = 0xff;
}
```

First, notice how each individual statement in C is terminated with a semicolon. Then notice how it resembles a mathematical equation…it is not!

An assignment statement has a right side, which is computed first. A resulting value is obtained (in this case it was simply a literal constant) and it is then transferred to the left side, which acts as

a receiving container. In this case it was a special-function 16-bit register of the microcontroller (the name of which was predefined in the `.h` file).

> **Note:** In C language, by prefixing the literal value with `0x`, we indicate the use of the hexadecimal radix. Otherwise the compiler assumes the default decimal radix. Alternatively, the `0b` prefix can be used for binary literal values, while for historical reasons a single `0` (zero) prefix is used for the octal notation. (Does anybody use octal anymore?)

24.3.1 Compiling and Linking

Now that we have completed the `main()` and only function of our first C program, how do we transform the source into a binary executable?

Using the MPLAB Integrated Development Environment (IDE), it is very easy! It's a matter of a single click of your mouse. This operation is called a Project Build. The sequence of events is fairly long and complex, but it is composed mainly of two steps:

- Compiling: The C compiler is invoked and an object code file (`.o`) is generated. This file is not yet a complete executable. While most of the code generation is complete, all the addresses of functions and variables are still undefined. In fact, this is also called a relocatable code object. If there are multiple source files, this step is repeated for each one of them.
- Linking: The linker is invoked and a proper position in the memory space is found for each function and each variable. Also any number of precompiler object code files and standard library functions may be added at this time as required. Among the several output files produced by the linker is the actual binary executable file (`.hex`).

All this is performed in a very rapid sequence as soon as you select the option "Build All" from the Project menu.

Should you prefer a command-line interface, you will be pleased to learn that there are alternative methods to invoke the compiler and linker and achieve the same results without using the MPAB IDE, although you will have to refer to the MPLAB C compiler User Guide for instructions. In the remainder of this book, we will stick to the MPLAB IDE interface and we will make use of the appropriate checklists to make it even easier.

In order for MPLAB to know which file(s) need to be compiled, we will need to add their names (`Hello.c` in this case) to the project Source Files List.

In order for the linker to assign the correct addresses to each variable and function, we will need to provide MPLAB with the name of a device-specific "linker script" file (`.gld`). Just

like the include (.h) file tells the compiler about the names (and sizes) of device-specific, special-function registers (SFRs), the linker scripts (.gld) file informs the linker about their predefined positions in memory (according to the device datasheet) as well as provides essential memory space information such as: total amount of Flash memory available, total amount of RAM memory available, and their address ranges.

The linker script file is a simple text file and it can be opened and inspected using the MPLAB editor.

Here is a segment of the p24fj128ga010.gld file where the addresses of the program counter and a few other special-function registers are defined:

```
  ...
  PCL         = 0x2E;
 _PCL         = 0x2E;
  PCH         = 0x30;
 _PCH         = 0x30;
  TBLPAG      = 0x32;
 _TBLPAG      = 0x32;
  PSVPAG      = 0x34;
 _PSVPAG      = 0x34;
  RCOUNT      = 0x36;
 _RCOUNT      = 0x36;
  SR          = 0x42;
 _SR          = 0x42;
  ...
```

24.3.2 Building the First Project

Let's review the last few steps required to complete our first demo project:

1. Add the current source file to the "Project Source Files" list. There are three possible checklists to choose from, corresponding to three different methods to achieve the same result. This first time we will:
   a) Open the Project window, if not already open, selecting "View→Project".
   b) With the cursor on the editor window, right click to activate the editor pop-up menu.
   c) Select "Add to project".

2. Add the PIC24 "linker script" file to the Project.
   Following the appropriate checklist "Add linker script to project":
   a) Right click on the linker scripts list in the project window.
   b) Select "Add file," browse and select the "p24fj128ga010.gld" file found in the support/gld subdirectory of MPLAB.

Your Project window should now look similar to Fig. 24.1.

3. Select the "Project→Build" function and watch the C30 compiler, followed by the linker, work and generate the executable code as well as a few, hopefully reassuring, messages in the MPLAB IDE Build window.

Note: The "Project Build" checklist contains several additional steps that will help you in future and more complex examples. (See Fig. 24.2.)

4. Select "Debugger→Select Tool→MPLAB SIM" to select and activate the simulator as our main debugging tool for this lesson. Note: the "MPLAB SIM debugger set-up" checklist will help you properly configure the simulator.

If all is well, before trying to run the code, let's also open a Watch window and select and add the `PORTA` special-function register to it (type or select `PORTA` in the SFR combo box, and then click on the "Add SFR" button). (See Fig. 24.3.)

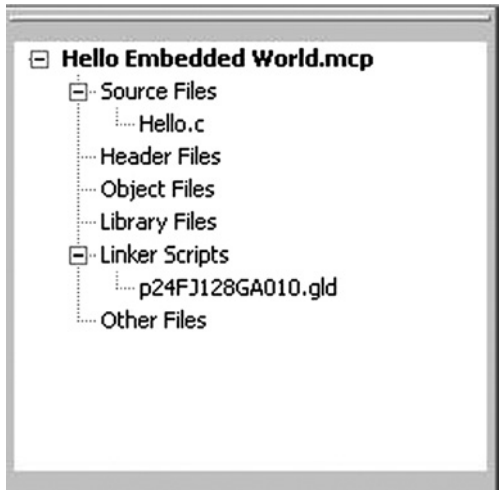


Figure 24.1: MPLAB IDE Project Window Set Up for the "Hello Embedded World" Project

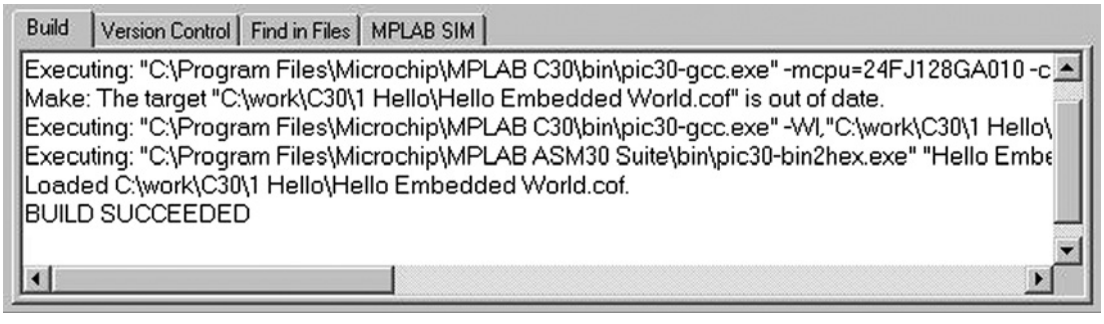


Figure 24.2: MPLAB IDE Output Window, Build Tab after Successfully Building a Project

5. Hit the simulator Reset button  (or select "Debugger→Reset") and observe the contents of PORTA. It should be cleared at reset. Then, place the cursor on the line containing the port assignment, inside the main function, and select the "Run to Cursor" option on the right-click menu (Fig. 24.4).

Figure 24.3: MPLAB IDE Watch Window

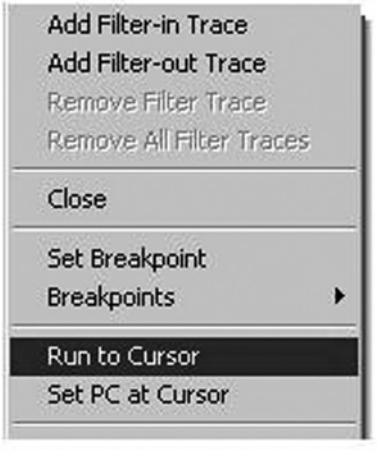


Figure 24.4: MPLAB IDE Editor Context Menu (Right Click)

This will let you skip all the C-compiler initialization code (`c0`) and get right to the beginning of our code.

6. Now single-step, (use the Step-Over  or Step-In  functions) to execute the one and only statement in our first program and observe how the content of `PORTA` changes in the Watch window. Or, notice how nothing happens: surprise!

24.3.3 PORT Initialization

It is time to hit the books, specifically the PIC24FJ128GA datasheet (Chapter 9, for the I/O ports detail). PORTA is a pretty busy, 16-pin wide, port (see Fig. 24.5 for a diagram of a typical PIC24 I/O port).

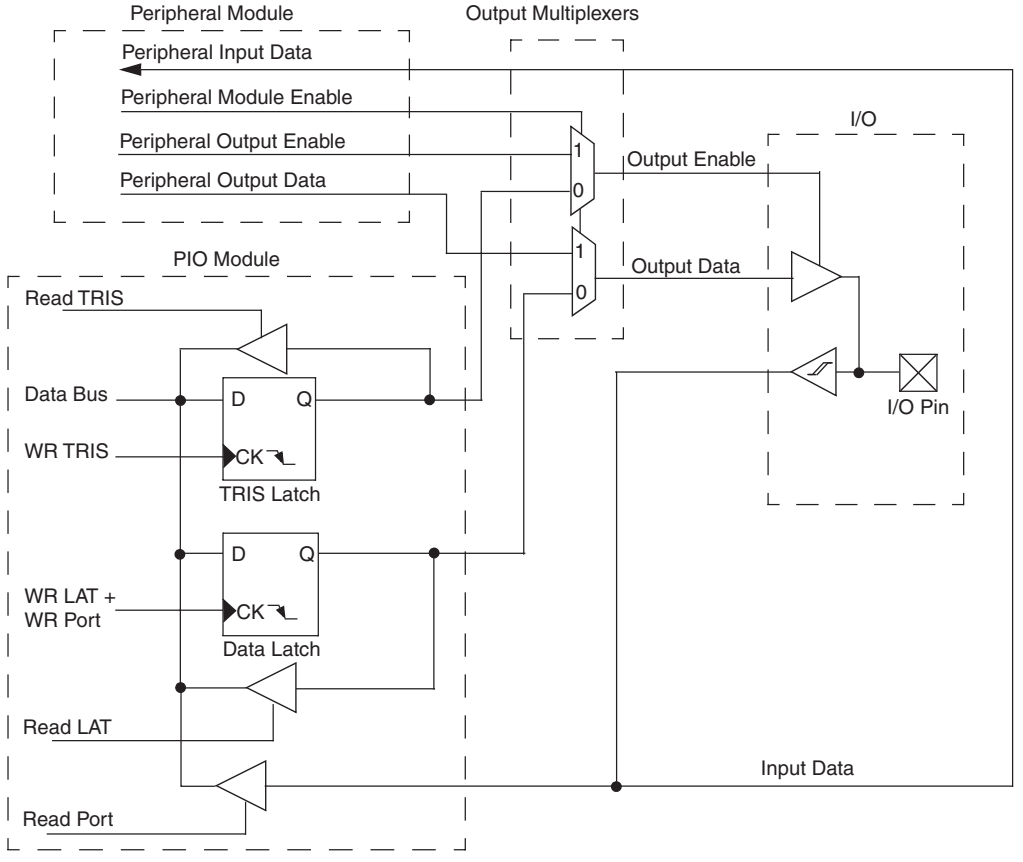


Figure 24.5: Diagram of a Typical PIC24 I/O Port

Looking at the pin-out diagrams on the datasheet, we can tell there are many peripheral modules being multiplexed on top of each pin. We can also determine what the default direction is for all I/O pins at reset: they are configured as inputs, which is a standard for all PIC® microcontrollers. The `TRISA` special-function register controls the direction of each pin on PORTA. Hence, we need to add one more assignment to our program, to change the direction of all the pins of PORTA to output, if we want to see their status change:

```
#include <p24fj128ga010.h>
main()
{   TRISA = 0;              // all PORTA pin output
    PORTA = 0xff;
}
```

24.3.4 Retesting PORTA

1. Rebuild the project now.
2. Set the cursor on the `TRISA` assignment.
3. Execute a "Run to Cursor" command to skip all the compiler initialization just as we did before.
4. Execute a couple of single steps and…we have it (Fig. 24.6)!

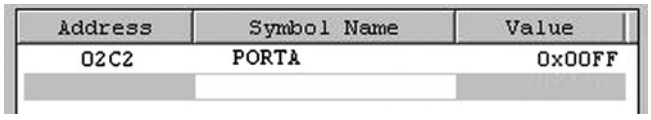


Figure 24.6: MPLAB IDE Watch Window Detail; PORTA Content has Changed

If all went well, you should see the content of `PORTA` change to 0x00FF, highlighted in the Watch window in red. Hello, World!

Our first choice of PORTA was dictated partially by the alphabetical order and partially by the fact that, on the popular Explorer16 demonstration boards, PORTA pins RA0 through RA7 are conveniently connected to 8 LEDs. So if you try to execute this example code on the actual demo board, you will have the satisfaction of seeing all the LEDs turn on, nice and bright.

24.3.5 Testing PORTB

To complete our lesson, we will now explore the use of one more I/O port, PORTB.

It is simple to edit the program and replace the two PORTA control register assignments with `TRISB` and `PORTB`. Rebuild the project and follow the same steps we did in the previous exercise and…you'll get a new surprise. The same code that worked for PORTA does not work for PORTB!

Don't panic! I did it on purpose. I wanted you to experience a little PIC24 migration pain. It will help you learn and grow stronger.

It is time to go back to the datasheet, and study in more detail the PIC24 pin-out diagrams. There are two fundamental differences between the 8-bit PIC microcontroller architecture and the new PIC24 architecture:

- Most of PORTB pins are multiplexed with the analog inputs of the analog-to-digital converter (ADC) peripheral. The 8-bit architecture reserved `PORTA` pins primarily for this purpose—the roles of the two ports have been swapped!

- With the PIC24, if a peripheral module input/output signal is multiplexed on an I/O pin, as soon as the module is enabled, it takes complete control of the I/O pin—independently of the direction (`TRISx`) control register content. In the 8-bit architectures, it was up to the user to assign the correct direction to each pin, even when a peripheral module required its use.

By default, pins multiplexed with "analog" inputs are disconnected from their "digital"input ports. This is exactly what was happening in the last example. All PORTB pins in the PIC24FJ128GA010 are, by default at power-up, assigned an analog input function; therefore, reading PORTB returns all 0s. Notice, though, that the output latch of PORTB has been correctly set although we cannot see it through the `PORTB` register. To verify it, check the contents of the `LATB` register instead.

To reconnect PORTB inputs to the digital inputs, we have to act on the analog-to-digital conversion (ADC) module inputs. From the datasheet, we learn that the special-function register `AD1PCFG` controls the analog/digital assignment of each pin (see Fig. 24.7).

Upper Byte:

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PCFG15	PCFG14	PCFG13	PCFG12	PCFG11	PCFG10	PCFG9	PCFG8
bit 15							bit 8

Lower Byte:

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PCFG7	PCFG6	PCFG5	PCFG4	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

bit 15-0 **PCFG15:PCFG0:** Analog Input Pin Configuration Control bits
1 = Pin for corresponding analog channel is configured in Digital mode; I/O port read enabled
0 = Pin configured in Analog mode; I/O port read disabled, A/D samples pin voltage

Figure 24.7: `AD1PCFG`: ADC Port Configuration Register

Assigning a 1 to each bit in the `AD1PCGF` special-function register will accomplish the task. Our new and complete program example is now:

```
#include <p24fj128ga010.h>

main()
{
     TRISB =       0;                // all PORTB pins output
     AD1PCFG =     0xffff;           // all PORTB pins digital
     PORTB =       0xff;
}
```

This time, compiling and single-stepping through it will give us the desired results (Fig. 24.8).

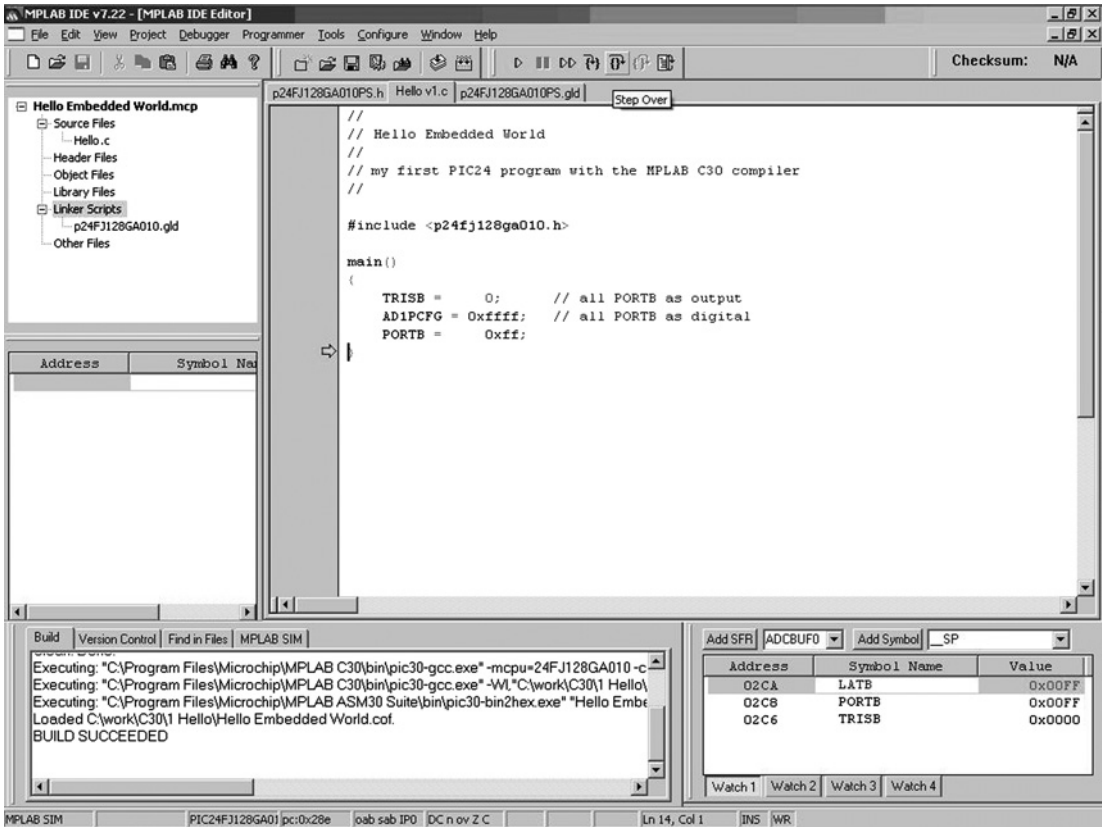


Figure 24.8: Hello Embedded World Project

24.4 Review

Writing a C program for a PIC24 microcontroller can be very simple, or at least no more complicated than the assembly-language equivalent. Two or three instructions, depending on which port we plan to use, can give us direct control over the most basic tool available to the microcontroller for communication with the rest of the world: the I/O pins.

Also, there is nothing the C30 compiler can do to read our mind. Just like in assembly, we are responsible for setting the correct direction of the I/O pins. And we are still required to study the datasheet and learn about the small differences between the 8-bit PIC microcontrollers we might be familiar with and the new 16-bit breed.

As high-level as the C programming language is touted to be, writing code for an embedded-control device still requires us to be intimately familiar with the finest details of the hardware we use.

24.4.1 Notes for Assembly Experts

If you have difficulties blindly accepting the validity of the code generated by the MPLAB C30 compiler, you might find comfort in knowing that, at any given point in time, you can decide to switch to the "Disassembly Listing" view (Fig. 24.9). You can quickly inspect the code generated by the compiler, as each C source line is shown in a comment that precedes the segment of code it generated.

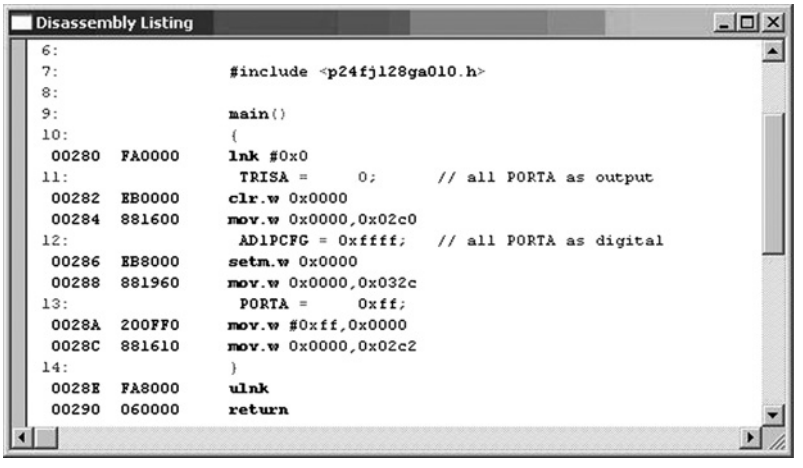


Figure 24.9: Disassembly Listing Window

You can even single-step through the code and do all the debugging from this view, although I strongly encourage you not to do so (or limit the exercise to a few exploratory sessions as we progress through the first chapters of this book). Satisfy your curiosity, but gradually learn to trust the compiler. Eventually, use of the C language will give a boost to your productivity and increase the readability and maintainability of your code.

As a final exercise, I encourage you to open the Memory Usage Gauge window—select "View→Memory Usage Gauge" (Fig. 24.10).

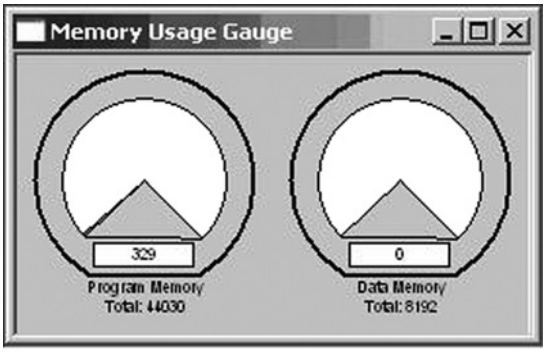


Figure 24.10: MPLAB IDE Memory Usage Gauge Window

Don't be alarmed! Even though we wrote only three lines of code in our first example and the amount of program memory used appears to already be up to 300+ bytes, this is not an indication of any inherent inefficiency of the C language. There is a minimum block of code that is always generated (for our convenience) by the C30 compiler. This is the initialization code `(c0)` that we mentioned briefly before. We will get to it, in more detail, in the following chapters as we discuss variables initialization, memory allocation and interrupts.

24.4.2 Notes for PIC MCU Experts

Those of you who are familiar with the PIC16 and PIC18 architecture will find it interesting that most PIC24 control registers, including the I/O ports, are now 16 bits wide. Looking at the PIC24 datasheet, note also how most peripherals have names that look very similar, if not identical, to the 8-bit peripherals you are already familiar with. You will feel at home in no time!

24.4.3 Notes for C Experts

Certainly we could have used the `printf` function from the standard C libraries. In fact the libraries are readily available with the MPLAB C30 compiler. But we are targeting embedded-control applications and we are not writing code for multigigabyte workstations. Get used to manipulating low-level hardware peripherals inside the PIC24 microcontrollers. A single call to a library function, like `printf`, could have added several kilobytes of code to your executable. Don't assume a serial port and a terminal or a text display will always be available to you. Instead, develop a sensibility for the "weight" of each function and library you use in light of the limited resources available in the embedded-design world.

24.4.4 Tips and Tricks

The PIC24FJ family of microcontrollers is based on a 3 V CMOS process with a 2.0 V to 3.6 V operating range. As a consequence, a 3 V power supply (Vdd) must be used and this limits the output voltage of each I/O pin when producing a logic high output. However, interfacing to 5 V legacy devices and applications is really simple:

- To drive a 5 V output, use the `ODCx` control registers (`ODCA` for PORTA, `ODCB` for PORTB and so on…) to set individual output pins in open-drain mode and connect external pull-up resistors to a 5 V power supply.
- Digital input pins instead are already capable of tolerating up to 5 V. They can be connected directly to 5 V input signals.

Be careful with I/O pins that are multiplexed with analog inputs though—they cannot tolerate voltages above Vdd.

Books

- Kernighan, B. and Ritchie, D., *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ.

  When you read or hear a programmer talk about the "K&R," they mean this book! Also known as "the white book," the C language has evolved since the first edition of this book was published in 1978! The second edition (1988) includes the more recent ANSI C standard definitions of the language, which is closer to the standard the MPLAB C30 compiler adheres to (ANSI90).

Links

- http://en.wikibooks.org/wiki/C_Programming

  This is a Wiki-book on C programming. It's convenient if you don't mind doing all your reading online. Hint: look for the chapter called "A taste of C" to find the omnipresent "Hello World!" exercise.

CHAPTER 25

Programming Loops

25.1 The Plan

Embedded-control programs need a framework, similar to a pilot's pattern, so that the flow of code can be managed. In this lesson, we will review the basics of the loops syntax in C and we'll also take the opportunity to introduce a new peripheral module: the 16-bit Timer1. Two new MPLAB® SIM features will be used for the first time: the "Animate" mode and the "Logic Analyzer."

25.2 Checklist

For this second lesson, we will need the same basic software components installed (from the attached CD-ROM and/or the latest versions, available for download from Microchip's website) and used before, including:

- MPLAB IDE, Integrated Development Environment
- MPLAB SIM, software simulator
- MPLAB C30 compiler (Student Version)

We will also reuse the "New Project Set-up" checklist to create a new project with the MPLAB IDE:

1. Select "Project→Project Wizard", to start creating a new project.
2. Select the PIC24FJ128GA010 device, and click Next.
3. Select the MPLAB C30 compiler suite and click Next.
4. Create a new folder and name it "Loop." name the project "A Loop in the Pattern," and click Next.

5. There is no need to copy any source files from the previous lessons; click Next once more.
6. Click Finish to complete the Wizard set-up.

This will be followed by the "Adding Linker Script file" checklist, to add the linker script "`p24fj128ga010.gld`" to the project. It can typically be found in the MPLAB IDE installation directory "`C:/Program Files/Microchip/`", within the subdirectory "`MPLAB C30/support/gld/`".

After completing the "Create New File and Add to Project" checklist:

7. Open a new editor window.
8. Type the main program header:

```
//
//  A loop in the pattern
//
```

9. Select "Project→AddNewFiletoProject", to save the file as: "`loop.c`" and have it automatically added to the project source files list.
10. Save the project.

25.3 Coding

One of the key questions that might have come to mind after working through the previous lesson is "What happens when all the code in the `main()` function has been executed?" Well, nothing really happens, or at least nothing that you would not expect. The device will reset, and the entire program will execute again…and again.

In fact, the compiler puts a special software reset instruction right after the end of the `main()` function code, just to make sure. In embedded control we want the application to run continuously, from the moment the power switch has been flipped on until the moment it is turned off. So, letting the program run through entirely, reset and execute again might seem like a convenient way to arrange the application so that it keeps repeating as long as there is "juice." This option might work in a few limited cases, but what you will soon discover is that, running in this "loop." you develop a "limp." Reaching the end of the program and executing a reset takes the microcontroller back to the very beginning to execute all the initialization code, including the `c0` code segment briefly mentioned in the previous lesson. So, as short as the initialization part might be, it will make the loop very unbalanced. Going through all the special-function register and global-variables initializations each time is probably not necessary and it will

certainly slow down the application. A better option is to design an application "main loop." Let's review the most basic loop-coding options in C first.

25.3.1 `While` Loops

In C there are at least three ways to code a loop; here is the first—the `while` loop:

```
while (x)
{
    // your code here...
}
```

Anything you put between those two curly brackets ({}) will be repeated for as long as the logic expression in parenthesis `(x)` returns a true value. But what is a logic expression in C?

First of all, in C there is no distinction between logic expressions and arithmetic expressions. In C, the Boolean logic TRUE and FALSE values are represented just as integer numbers with a simple rule:

- FALSE is represented by the integer 0.
- TRUE is represented by *any* integer except 0.

So `1` is true, but so are `13` and `-278`. In order to evaluate logic expressions, a number of logic operators are defined, such as:

`||` the logic OR operator,
`&&` the logic AND operator,
`!` the logic NOT operator.

These operators consider their operands as logical (Boolean) values using the rule mentioned above, and they return a logical value. Here are some trivial examples:

(when `a = 17` and `b = 1`, or in other words they are both true)

`(a || b)` is true,
`(a && b)` is true
`(!a)` is false

There are, then, a number of operators that compare numbers (integers of any kind and floating-point values, too) and return logic values. They are:

`==` the "equal-to" operator; notice it is composed of two equal signs to distinguish it from the "assignment" operator we used in the previous lesson,

`!=` the "NOT-equal to" operator,

`>` the "greater-than" operator,

`>=` the "greater-or-equal to" operator,

`<` the "less-than" operator,

`<=` the "less-than-or-equal to" operator.

Here are some examples:

assuming `a = 10`

`(a > 1)` is true

`(-a >= 0)` is false

`(a == 17)` is false

`(a != 3)` is true

Back to the `while` loop, we said that as long as the expression in parentheses produces a true logic value (that is any integer value but 0), the program execution will continue around the loop. When the expression produces a false logic value, the loop will terminate and the execution will continue from the first instruction after the closing curly bracket.

Note that the expression is evaluated first, before the curly bracket content is executed (if ever), and is then reevaluated each time.

Here are a few curious loop examples to consider:

```
While (0)
{
    // your code here...
}
```

A constant false condition means that the loop will never be executed. This is not very useful. In fact I believe we have a good candidate for the "world's most useless code" contest!

Here is another example:

```
while (1)
{
      // your code here...
}
```

A constant true condition means that the loop will execute forever. This is useful, and is in fact what we will use for our main program loops from now on. For the sake of readability, a few purists among you will consider using a more elegant approach, defining a couple of constants:

```
#define TRUE      1
#define FALSE     0
```

and using them consistently in their code, as in:

```
While (TRUE)
{
    // your code here...
}
```

It is time to add a few new lines of code to the "loop.c" source file now, and put the while loop to good use.

```
#include <p24fj128ga010.h>
main()
{
    // init the control registers
    TRISA =   0xff00;// PORTA pin 0..7 as output

    // application main loop
    while(1)
    {
       PORTA = 0xff; // turn pin 0-7 on
       PORTA = 0;          // turn all pin off
    }
}
```

The structure of this example program is essentially the structure of every embedded-control program written in C. There will always be two main parts:

- The initialization, which includes both the device peripherals initialization and variables initialization, executed only once at the beginning.
- The main loop, which contains all the control functions that define the application behavior, and is executed continuously.

25.3.2 An Animated Simulation

Use the Project Build checklist to compile and link the "loop.c" program. Also use the "MPLAB SIM simulator set-up" checklist to prepare the software simulator.

To test the code in this example with the simulator, I recommend you use the "Animate" mode (Debugger→Animate). In this mode, the simulator executes one C program line at a time, pausing for ½ second after each one to give us the time to observe the immediate results. If you add the PORTA special-function register to the Watch window, you should be able to see its value alternating rhythmically between 0xff and 0x00.

The speed of execution in Animate mode can be controlled with the "Debug→Settings" dialog box, selecting the "Animation/Real Time Updates" tab, and modifying the "Animation Step Time" parameter, which by default is set to 500 ms. As you can imagine, the "Animate" mode can be a valuable and entertaining debugging tool, but it gives you quite a distorted idea of

what the actual program execution timing will be. In practice, if our example code was to be executed on a real hardware target, say an Explorer16 demonstration board (where the PIC24 is running at 32 MHz), the LEDs connected to the PORTA output pins would blink too fast for our eyes to notice. In fact, each LED would be turned on and off several million times each second.

To slow things down to a point where the LEDs would blink nicely just a couple of times per second, I propose we use a timer, so that in the process we learn to use one of the key peripherals integrated in all PIC24 microcontrollers. For this example, we will choose the first timer, Timer1, of the five timers available inside the PIC24FJ128GA010. This is one of the most flexible and simple peripheral modules. All we need to do is take a quick look at the PIC24 datasheet, check the block diagram (Fig. 25.1) and the details of the Timer1 control registers, and find the ideal initialization values.

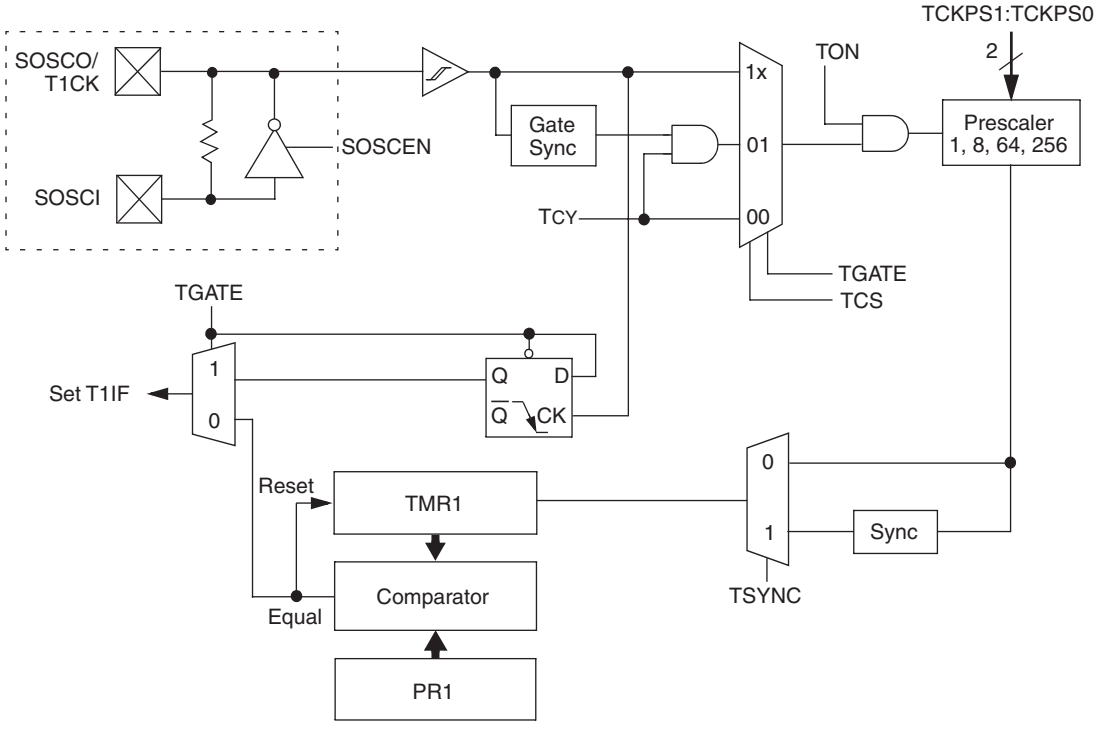


Figure 25.1: 16-bit Timer1 Module Block Diagram

We quickly learn that there are three special-function registers that control most of the Timer1 functions. They are:

- `TMR1`, which contains the 16-bit counter value.
- `T1CON`, which controls activation and the operating mode of the timer (Fig. 25.2).
- `PR1`, which can be used to produce a periodic reset of the timer (not required here).

We can clear the `TMR1` register to start counting from zero. `TMR1 = 0;` Then we can initialize `T1CON` so that the timer will operate in a simple configuration where:

- Timer1 is activated: `TON = 1`
- The main MCU clock serves as the source (Fosc/2): `TCS = 0`
- The prescaler is set to the maximum value (1:256): `TCKPS = 11`
- The input gating and synchronization functions are not required, since we use the MCU internal clock directly as the timer clock: `TGATE = 0, TSYNC = 0`
- We do not worry about the behavior in IDLE mode: `TSIDL = 0 (default)`

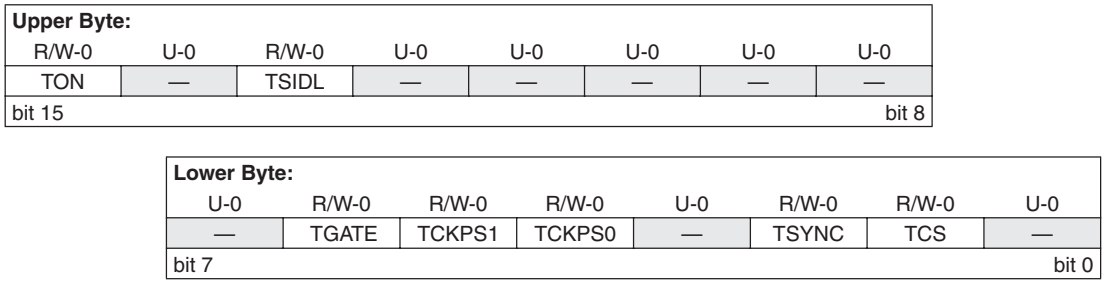


Figure 25.2: T1CON: Timer1 Control Register

Once we assemble all the bits in a single 16-bit value to assign to `T1CON`, we get:

```
T1CON = 0b1000000000110000;
```

or, in a more compact hexadecimal notation:

```
T1CON = 0x8030;
```

Once we are done initializing the timer, we enter a loop where we wait for TMR1 to reach the desired value set by the constant DELAY.

```
while(TMR1 < DELAY)
{
      // wait
}
```

Assuming a 32-MHz clock will be used, we need to assign quite a large value to DELAY so as to obtain a delay of about a quarter of a second. In fact the following formula dictates the total delay time produced by the `TMR1` loop:

```
Tdelay = (2/Fosc) * 256 * DELAY
```

With `Tdelay` = 256 ms and resolving for `DELAY`, we obtain the value 16,000:

```
#define DELAY 16000
```

By putting two such delay loops in front of each PORTA assignment inside the main loop, we get our latest and best code example:

```
#include <p24fj128ga010.h>

#define DELAY      16000

main()
{
    // init the control registers
    TRISA = 0xff00;              // PORTA pin 0..7 as output
    T1CON = 0x8030;      // TMR1 on, prescaler 1:256 Tclk/2

    // main application loop
    while(1)
    {
    // 1. turn pin 0-7 on and wait for ¼ second
    PORTA = 0xff;
    TMR1 = 0;     // restart the count
    while (TMR1 <DELAY)
    {
          // just wait
    }

    // 2. turn all pin off and wait for ¼ second
    PORTA = 0x00;
    TMR1 = 0;     // restart the count
    while (TMR1 < DELAY)
    {
          // just wait
    }

    } // main loop
} // main
```

Note: When programming in C, the number of opening and closing curly brackets tends to increase rapidly as your code grows. After a very short while, even if you stick religiously to the best indentation rules, it can become difficult to remember which closing curly brackets belong to which opening curly brackets. By putting little reminders (comments) on the closing brackets, I try to make it easier and more readable.

It is time now to build the complete project and verify that it is working. If you have an Explorer16 demonstration board available, you may try to run the code right away. The LEDs should flash at a comfortably slow pace, with a frequency of about two flashes per second.

If you try to run the same code with the MPLAB SIM simulator, though, you will discover that things are now way too slow. I don't know how fast your PC is, but on mine MPLAB-SIM cannot get anywhere close to the execution speed of a true 32-MHz PIC24 microcontroller.

If you use the Animate mode, things get even worse. As we saw before, the animation adds a further delay of about half a second between the execution of each individual line of code. So, for pure debugging purposes, on the simulator feel free to change the `DELAY` constant to a much smaller value (16, for example).

25.4 Using the Logic Analyzer

To complete this lesson and make things more entertaining, after building the project, I suggest we play with a new simulation tool: the MPLAB logic analyzer.

The logic analyzer gives you a graphical and extremely effective view of the recorded values for any number of the device output pins, but it requires a little care in the initial set-up.

Before anything else, you should make sure that the Tracing function of the simulator is turned on.

1. Select the "Debug→Settings" dialog box and then choose the Osc/Trace tab.
2. In the Tracing options section, check the Trace All box.
3. Now you can open the Analyzer window, from the "View→Simulator" Logic Analyzer menu (Fig. 25.3).

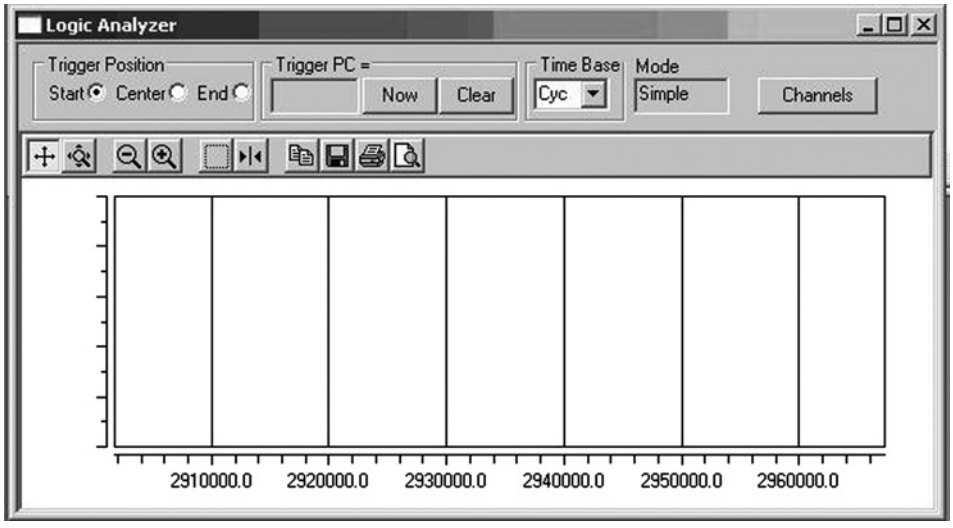


Figure 25.3: Logic Analyzer Window

4. Then click on the channel button, to bring up the channel-selection dialog box (Fig. 25.4).

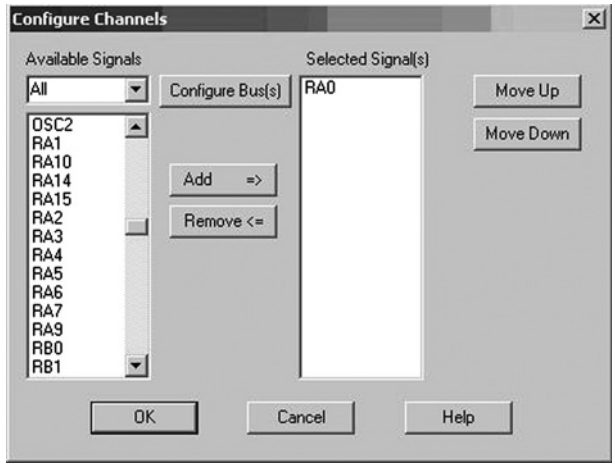


Figure 25.4: Channel Selection Dialog Box

5. From here, you can select the device output pins you would like to visualize. In our case, select RA0 and click "Add =>".
6. Click on OK to close the channel-selection dialog box.

> **Note:** For future reference, all the steps above are listed in the "Logic Analyzer Set-up" checklist.

Run the code  for a short while and then hit the Halt button  . The Logic Analyzer window should display a neat square-wave plot, as in Fig. 25.5.

25.5 Review

In this brief lesson, we learned about the way the MPLAB C30 compiler deals with program termination. For the first time, we gave our little project a bit of structure—separating the `main()` function in an initialization section and an infinite loop. To do so, we learned about the `while` loop statements and we took the opportunity to touch briefly on the subject of logical expressions evaluation. We closed the lesson with a final example, where we used a timer module for the first time and we played with the Logic Analyzer window to plot the RA0 pin output.

We will return to all these elements, so don't worry if you have more doubts now than when we started—this is all part of the learning experience.

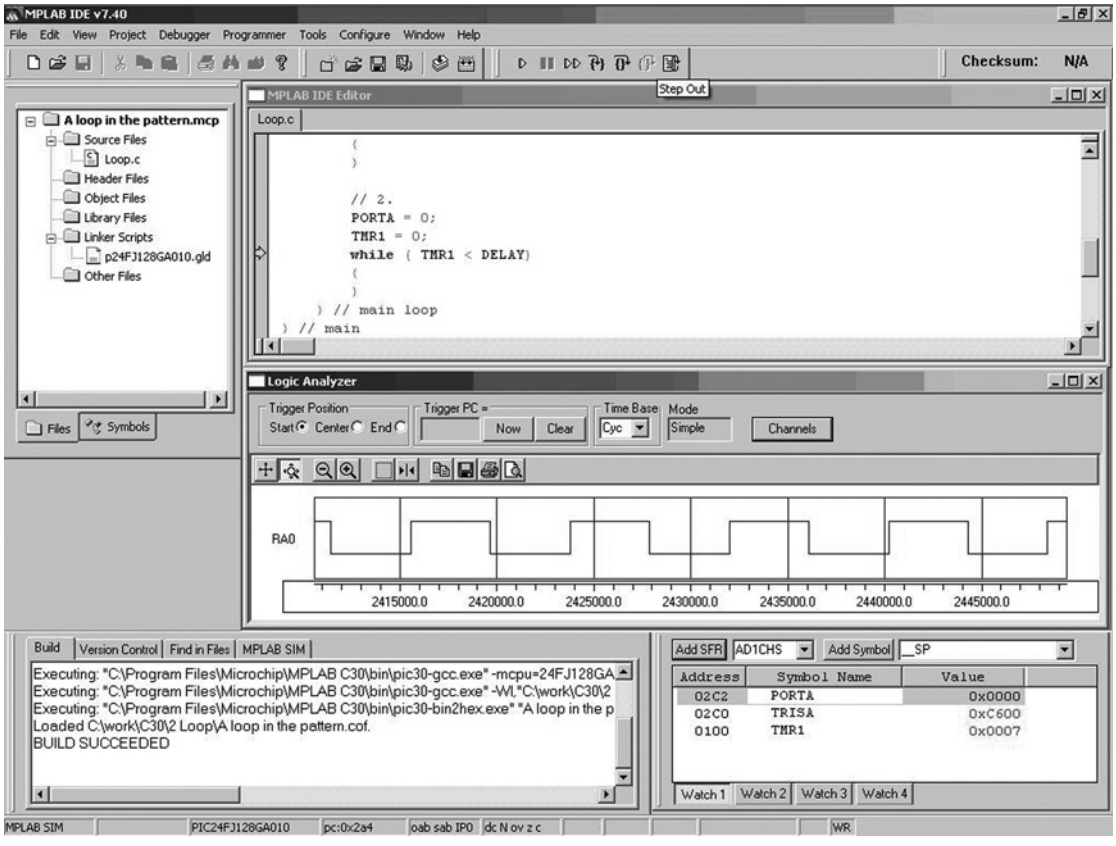


Figure 25.5: Logic Analyzer Showing Square-wave Plot

25.5.1 Notes for Assembly Experts

Logic expressions in C can be tricky for the assembly programmer who is used to dealing with *binary* operators of identical names (AND, OR, NOT…). C has a set of binary operators too, but I purposely avoided showing them in this lesson to avoid mixing things up. Binary logic operators take pairs of bits from each operand and compute the result according to the defined truth table. Logic operators, on the other hand, look at each operand (independently of the number of bits used) as a single Boolean value.

See the following examples of byte-sized operands:

```
            11110101                    11110101 (TRUE)
binary OR   00001000       logical OR   00001000 (TRUE)
            --------                    --------
gives       11111101       gives        00000001 (TRUE)
```

25.5.2 Notes for PIC Microcontroller Experts

I am sure you noticed: Timer0 has disappeared! The good news is: you are not going to miss it! In fact, the remaining five timers of a PIC24 are so loaded with features that there is no functionality of Timer0 that you are going to feel nostalgic about. All of the special-function registers that control the timers have names similar to the ones used on PIC16 and PIC18 microcontrollers, and are pretty much identical in structure. Still, keep an eye on the datasheet; the designers managed to cram in several new features, including:

- All timers are now 16 bits wide.
- Each timer has a 16-bit period register.
- A new 32-bit mode timer-pairing mechanism is available for Timer2/3 and Timer4/5.
- A new external clock gating feature has been added on Timer1.

25.5.3 Notes for C Experts

If you are used to programming in C on a personal computer or workstation, you expect that, upon termination of the `main()` function, control would be returned to the operating system. While several real-time operating systems (RTOSs) are available for the PIC24, a large number of applications don't need and won't use one. This is certainly true for all the simple examples in this book. By default, the C30 compiler assumes there is no operating system to return control to, and does the safest possible thing—it resets.

25.5.4 Tips and Tricks

Some embedded applications are designed to run their main loop for months or years in a row without ever being turned off or receiving a reset command. But the control registers of a microcontroller are simple RAM memory cells. The probability that a power-supply fluctuation (undetected by the brownout reset circuit), an electromagnetic pulse emitted by some noisy equipment in the proximity, or even a cosmic ray could alter their contents is a small but finite number. Given enough time, depending on the application, you will see it happen. When you design applications that have to operate reliably on such huge time scales, you should start seriously considering the need to provide a periodic "refresh" of the most important control registers of the essential peripherals used by the application.

Group the sequence of initialization instructions in one or more functions. Call the functions once at power-up, before entering the main loop, but also make sure that inside the main loop the initialization functions are called when no other critical task is pending and every control register is reinitialized periodically.

Books

- Ullman, L. and Liyanage, M. (2005)

  C Programming

  Peachpit Press, Berkeley, CA.

  This is a fast-reading and modern book, with a simple step-by-step introduction to the C programming language.

Links

- http://en.wikipedia.org/wiki/Control_flow#Loops

  A wide perspective on programming languages and the problems related to coding and taming loops.

- http://en.wikipedia.org/wiki/Spaghetti_code

  Your code gets out of control when you cannot fly the pattern.

This page intentionally left blank

CHAPTER 26

More Pattern Work, More Loops

26.1 The Plan

In the previous lesson, we learned there is a loop at the core of every embedded-control application. In this lesson, we will continue exploring a variety of other techniques available to the C programmer to perform loops. Along the way, we will take the opportunity to briefly review integer variables declarations, and increment and decrement operators, quickly touching the arrays declaration and use subject. As in any good flight lesson, the theory is immediately followed by the practice, and we will conclude the lesson with a, hopefully entertaining, exercise that will make use of all the concepts and tools acquired during the lesson.

26.2 Checklist

In this lesson we will continue using the MPLAB® SIM software simulator, but once more an Explorer16 demonstration board could add to the entertainment. In preparation for the new demonstration project, you can use the "New Project Set-up" checklist to create a new project called "More Loops" and create a new source file to be called "`More.c`".

26.3 Coding

In a `while` loop, a block of code enclosed by two curly brackets is executed if, and for as long as, a logic expression returns a Boolean true value (not zero). The logic expression is evaluated before the loop, which means that if the expression returns false right from the beginning, the code inside the loop might never be executed.

26.3.1 Do Loops

If you need a type of loop that gets executed at least once, but only subsequent repetitions are dependent on a logic expression, then you have to look at a different type of loop.

Let me introduce you to the `do` loop syntax:

```
do {
   // your code here...
} while ( x);
```

Don't be confused by the fact that the `do` loop syntax is using the `while` keyword again to close the loop—the behavior of the two loop types is very different.

In a `do` loop, the code (if any) found between the curly brackets is always executed first, and only then is the logic expression evaluated. Of course, if all we want is an infinite loop for our `main()` function, then it makes no difference if we choose the `do` or the `while`…

```
main()
{
     // initialization code
     ...

     // main application loop
     do {
            ...
     } while (1)
} // main
```

Looking for curious cases, we might analyze the behavior of the following loop:

```
do{
     // your code segment here...
} while (0);
```

You will realize that the code segment inside the loop is going to be executed once and, no matter what, only once. In other words, the loop syntax around the code is, in this case, a total waste of your typing efforts and another good candidate for the "most useless piece of code in the world" contest.

Let's now look at a more useful example, where we use a `while` loop to repeatedly execute a piece of code for a predefined and exact number of times. First of all, we need a variable to perform the count. In other words, we need to allocate one or more RAM memory locations to store a counter value.

> **Note:** In the previous two lessons we have been able to skip almost entirely the subject of variable declarations, as we relied exclusively on the use of what are in fact predefined variables: the special-function registers of the PIC24.

26.3.2 Variable Declarations

We can declare an integer variable with the following syntax:

```
int c;
```

Since we used the keyword `int` to declare `c` as a 16-bit (signed) integer, the MPLAB C30 compiler will make arrangements for two bytes of memory to be used. Later, the linker will

determine where those two bytes will be allocated in the physical RAM memory of the selected PIC24 model. As defined, the variable `c` will allow us to count from a negative minimum value −32,768 to a maximum positive value of +32,767. If we need a larger integer numerical range, we can use the `long` (signed) integer type as in:

```
long c;
```

The MPLAB C30 compiler will use 32 bits (four bytes) for the variable.

If we are looking for a smaller counter, and we can accept a range of values from −128 to +127, we can use the `char` integer type instead:

```
char c;
```

In this case the MPLAB C30 compiler will use 8 bits (a single byte). All three types can be further modified by the `unsigned` attribute as in:

```
unsigned char c;       // ranges from 0..255
unsigned int i;        // ranges from 0..65,535
unsigned long l;       // ranges from 0..4,294,967,295
```

There are then variable types defined for use in floating-point arithmetic:

```
float f;            // defines a 32 bit precision floating point
long double d;      // defines a 64 bit precision floating point
                    // variable
```

26.3.3 `for` Loops

We can now return to our counter example. All we need is a simple integer variable to be used as index/counter, capable of covering the range from 0 to 5; therefore a `char` integer type will do:

```
char i;                // declare i as an 8-bit integer with sign

i = 0;                 // init the index/counter

while ( i<5)
{
     // insert your code here...
     // it will be executed for i= 0, 1, 2, 3, 4

     i = i+1;             // increment
}
```

Whether counting up or down, this is something you are going to do a lot in your everyday programming life.

In C language, there is a third type of loop that has been designed specifically to make coding this common case easy. It is called the `for` loop, and this is how you would have used it in the previous example:

```
for ( i=0; i<5; i=i+1)
{
      // insert your code here...
      // it will be executed for i=0, 1, 2, 3, 4
}
```

You will agree that the `for` loop syntax is compact, and it is certainly easier to write. It is also easier to read and debug later. The three expressions separated by semicolons and enclosed in the brackets following the `for` keyword are exactly the same three expressions we used in the prior example:

- initialize the index.
- check for termination, using a logic expression.
- advance the index/counter…in this case incrementing it.

You can think of the `for` loop as an abbreviated syntax of the `while` loop. In fact, the logic expression is evaluated first and, if false from the beginning, the code inside the loop's curly brackets may never be executed.

Perhaps this is also a good time to review another convenient shortcut available in C. There is a special notation reserved for the increment and decrement operations that uses the operators:

```
++    to increment, as in    i++;    is equivalent to    i = i+1;
--    to decrement, as in    i--;    is equivalent to    i = i-1;
```

26.3.4 More Loop Examples

Let's see some more examples of the use of the `for` loop and the increment/decrement operators.

First, a count from 0 to 4:

```
for ( i=0; i<5; i++)
{
     // insert your code here...
     // it will be executed for i= 0, 1, 2, 3, 4
}
```

Then, a countdown from 4 to 0:

```
for ( i=4; I>=0; i--)
{
      // insert your code here...
      // it will be executed for i= 4, 3, 2, 1, 0
}
```

Can we use the `for` loop to code an (infinite) main program loop? Sure we can—here is an example:

```
main()
{
      // 0. initialization code
      ...

      // 1. the main application loop
      for ( ; 1; )
      {
            ...
      }
} // main
```

If you like it, feel free to use this form. As for me, from now on I will stick to the `while` syntax (it is just an old habit).

26.3.5 Arrays

Before starting to code our next project, we need to review one last C-language feature: array variable types. An array is just a contiguous block of memory containing a given number of identical elements of the same type. Once the array is defined, each element can be accessed via the array name and an index. Declaring an array is as simple as declaring a single variable—just add the desired number of elements in square brackets after the variable name:

```
char c[10];  // declares c as an array of 10 x 8-bit integers
int i[10];   // declares i as an array of 10 x 16-bit integers
long l[10];  // declares l as an array of 10 x 32-bit integers
```

The same squared-brackets notation is used to refer to the content or assign a value to each element of an array as in:

```
a = c[0];          // copy the value of the 1st element of c into a
c[1] = 123;        // assign the value 123 to the second element of c
i[2] = 12345;      // assign the value 12,345 to the third element of i
l[3] = 123* i[4]; // compute 123 x the value of the fifth element of i
```

> **Note:** In C language, the elements of an array of size N have indexes 0, 1, 2. . .(N – 1). It is when manipulating arrays that the `for` type of loop shows all its merits.

```
int a[10];    // declare array of 10 integers: a[0], a[1], a[2]...
              // a[9]
int i;        // the loop index
for ( i=0; I<10; i++)
{
   a[ i] = 1;
}
```

26.3.6 A New Demo

The best way to conclude this lesson would be to take all the elements of the C language we have reviewed so far and put them to use in our next project. This project will consist of making a row of LEDs, connected to PORTA (as they happen to be connected on the Explorer16 demo board), flash in a rapid sequence so that when moving the board left and right rhythmically they will display a short text message.

How about "Hello World! " or, perhaps more modestly, "HELLO"?

Here is the code:

```
#include <p24fj128ga010.h>

// 1. define timing constant
#define SHORT_DELAY 100
#define LONG_DELAY 800

// 2. declare and initialize an array with the message bitmap
char bitmap[30] = {
   0b11111111,    // H
   0b00001000,
   0b00001000,
   0b11111111,
   0b00000000,
   0b00000000,
   0b11111111,    // E
   0b10001001,
   0b10001001,
   0b10000001,
   0b00000000,
   0b00000000,
   0b11111111,    // L
   0b10000000,
   0b10000000,
   0b10000000,
   0b00000000,
   0b00000000,
   0b11111111,    // L
   0b10000000,
```

```
   0b10000000,
   0b10000000,
   0b00000000,
   0b00000000,
   0b01111110,    // O
   0b10000001,
   0b10000001,
   0b01111110,
   0b00000000,
   0b00000000
   };

// 3. the main program
main()
{
      // 3.1 variable declarations
      int i;               // i will serve as the index

      // 3.2 initialization
      TRISA = 0xff00;      // PORTA pins connected to LEDs are outputs
      T1CON = 0x8030;      // TMR1 on, prescale 1:256 Tclk/2

      // 3.3 the main loop
      while( 1)
      {

            // 3.3.1 display loop, hand moving to the right
            for( i=0; I<30; i++)
            {     // update the LEDs
                  PORTA = bitmap[i];
                  // short pause
                  TMR1 = 0;
                  while ( TMR1 < SHORT_DELAY)
                  {
                  }
            }  // for i

            // 3.3.2 long pause, hand moving back to the left
            PORTA = 0;              // turn LEDs off
            // long pause
            TMR1 = 0;
            while ( TMR1 < LONG_DELAY)
            {
            }
      } // main loop
} // main
```

In section 1, we define a couple of timing constants, so that we can control the flashing sequence speed for execution and debugging. In section 2, we declare and initialize an 8-bit integer array of 30 elements, each containing an LED configuration in the sequence.

Hint: using a highlighter you can mark the "1s" on the page to see the message emerge.

Section 3 contains the main program, with the variable declarations (3.1) at the top, followed by the microcontroller initialization (3.2) and eventually the main loop (3.3).

The main (`while`) loop, in turn, is divided in two sections:

1.1.1 Contains the actual LED flash sequence, all 30 steps, that is to be played when the board is swept from left to right. A `for` loop is used for accessing each element of the array, in order. A `while` loop is used to wait on Timer1 for the proper sequence timing.

1.1.2 Contains a pause for the sweep back, implemented using a `while` loop with a longer delay on Timer1.

26.4 Testing with the Logic Analyzer

To test the program, we will initially use the MPLAB SIM software simulator and the Logic Analyzer window.

1. Build the project (using the appropriate check list).
2. Open the Logic Analyzer window.
3. Click on the Channel button to add, in order, all the I/O pins from RA0 to RA7 connected to the row of LEDs.

The "MPLAB SIM Set-up" and "Logic Analyzer Set-up" checklists will help you make sure that you don't forget any detail.

Then, I suggest you go back to the editor window and set the cursor on the first instruction of the 3.3.2 section and select the "Run to Cursor" option from the right click (context) menu. This will let the program execute the entire portion containing the message output (3.3.1) and stop before the long delay. As soon as the simulation halts on the cursor line, you can switch to the Logic Analyzer window and verify the output waveforms. They should look like Fig. 26.1.

To help you visualize the output, I added a few dots to represent the LEDs being turned on during the first few steps of the sequence. If you train your eye to see an LED on wherever the corresponding pin is at the logic high level, you should be able to read the desired message.

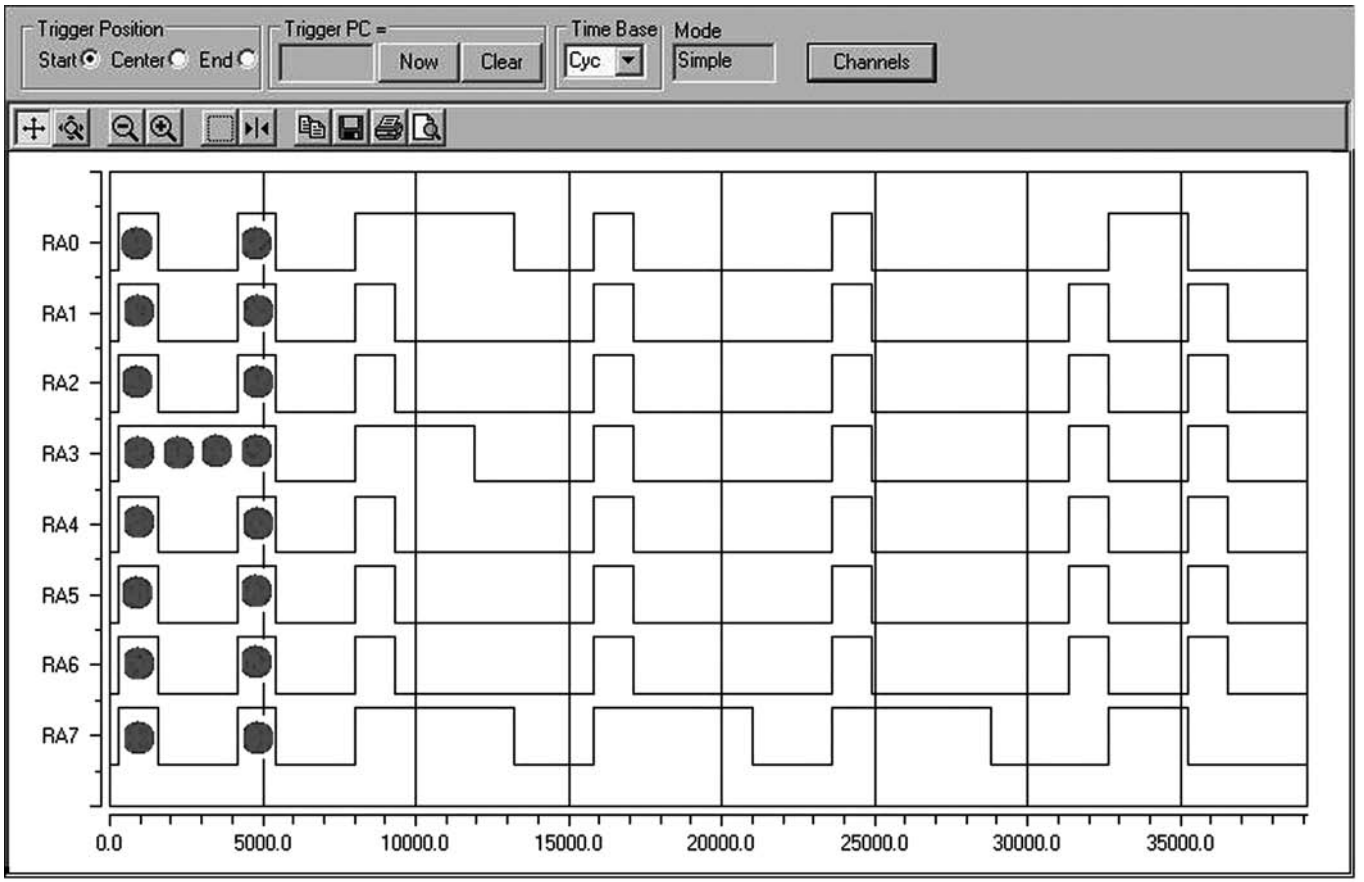


Figure 26.1: Snapshot of the Logic Analyzer Window After the First Sweep

26.5 Using the Explorer16 Demonstration Board

If you have an actual Explorer16 demonstration board available, the fun can be doubled.

1. Use the "MPLAB ICD2 Set-up" checklist to enable the in-circuit debugger.
2. Use the "MPLAB ICD2 Device Configuration" to verify the device configuration bits proper setting for use on the Explorer16 demonstration board.
3. Use the "MPLAB ICD2 Programming" checklist to program the PIC24 in circuit.

If successful, and if you dim the light a bit in the room, you should be able to see the message flashing as you "shake" the board. The experience is going to be far from perfect though. With the Simulator and the Logic Analyzer window, we can choose which part of the sequence we want to visualize with precision and "freeze" it on the screen. On the demonstration board, you might find it quite challenging to synchronize the board's movement with the LED sequence.

Consider adjusting the timing constants to your optimal speed. After some experimentation, I found the values 100 and 800 ideal, respectively, for the short and long delays, but your preferences might differ.

26.6 Review

In this lesson we reviewed the declaration of a few basic variable types, including integers and floating point of different sizes. Array declarations and their initialization were also used to create an LED display sequence and a `for` loop was used to play it back.

26.6.1 Notes for Assembly Experts

If you were wondering whether the increment and decrement operators were going to be translated by the C30 compiler with the `inc` and `dec` assembly instructions, you were mostly right. I am saying "mostly" and not "always" because the ++ and −− operators are actually much smarter than that. If the variable they are applied to is an integer, as in the trivial examples above, this is certainly the case. But if they are applied to a pointer (which is a variable type that contains a memory address) they actually increase the address by the exact number of bytes required to represent the quantity pointed to. For example, a pointer to 16-bit integers will increment its address by 2, a pointer to a 32-bit long integer will increment its address by 4, and so on. To satisfy your curiosity, switch to the disassembly view and see how the MPLAB C30 chooses the best assembly code, depending on the situation.

Loops in C can be confusing: should you test the condition at the beginning or the end? Should you use the `for` type or not? The fact is, in some situations the algorithm you are coding will dictate which one to use, but in many situations you will have a degree of freedom and more than one type might do. Choose the one that makes your code more readable, and if it really doesn't matter, as in the main loop, just choose the one you like and be consistent.

Depending on the target microcontroller architecture, and ultimately the arithmetic and logic unit (ALU), operating on bytes versus operating on word quantities can make a big difference in terms of code compactness and efficiency. While in the PIC16 and PIC18 8-bit architectures there is a strong incentive to use byte-sized integers wherever possible, in the PIC24, 16-bit architecture word-sized integers can be manipulated with the same efficiency. The only limiting factor preventing us from always using 16-bit integers with the MPLAB C30 compiler is the consideration of the relative preciousness of the internal resources of the microcontroller, and in this case the RAM memory.

26.6.2 Notes for C Experts

Even if PIC24 microcontrollers have a relatively large RAM memory array, embedded-control applications will always have to contend with the reality of cost and size limitations. If you learned to program in C on a PC or a workstation, you probably never considered using anything smaller than an `int` as a loop index. Well, this is the time to think again. Shaving one byte at a time off the requirements of your application might, in some cases, mean the ability to select a smaller model of PIC24 microcontroller, saving fractions of a dollar that, when multiplied by thousands or millions of units (depending on your production run rates), can mean real money added to the bottom line. In other words, if you learn to keep the size of your variables to the strict minimum necessary, you will become a better embedded-control designer and ultimately…this is what engineering is all about.

26.6.3 Tips and Tricks

This is the third lesson and, I am sure you will have noticed, for the third time I have been instructing you to start the simulation by setting a cursor on the first line of code and executing a Run To Cursor command (or setting a breakpoint) instead of more simply starting to single-step through the code. Why bother? Why can't we just start in Animation mode, for example, right after completing the project build?

As I briefly mentioned more than once, it is because of the `C0` initialization code. Let me add, it's also because of MPLAB's obsessive desire to shield you from the low-level details. In fact, MPLAB won't even show the cursor (the big green arrow) if you try to single-step through it—quite a disconcerting experience. It will not let you see any trace of the `C0` code even if you use the Disassembly window. But the `C0` code is starting to do interesting things for you, and you might be getting curious. For example, in this last exercise we declared an array called `bitmap[]` and we asked for it to be initialized with a specific series of values. The array, being a data structure, resides in RAM during execution, so the compiler has to instruct the `C0` initialization code to copy the contents of the array from a table in Flash memory immediately after the program start.

The only way to take a look at the `C0` inner workings is to open the Program Memory window ("View→Program Memory"), select the Symbolic mode (using the buttons at the bottom of

the window), and patiently inspect the assembly code. A few labels here and there will offer a little support. The first line of the program memory window will correspond to the reset vector of the PIC24 and will always contain a jump to the proper beginning of the program.

```
0000        goto _reset
```

You will have to scroll through several pages of what, you will learn shortly, is the interrupt vectors table. Eventually, you will find the `_reset` label. There, in a short sequence, you will recognize three essential pieces of code:

- the stack pointer (w15) initialization

  ```
  _reset mov.w #0x81e,w15
  ```

- a call to a subroutine for the variable (RAM) initialization

  ```
  rcall _data_init
  ```

- the call to the `main()` function

  ```
  call main
  ```

- a software reset instruction upon program termination

  ```
  reset
  ```

I hope this satisfies your curiosity for now. If during a future debugging session you are not able to find the cursor, chances are you will be able to find it in here. Something might have caused the processor to reset (a bug, an external event?) and you might be stepping through the very heart of the `C0` initialization code. Check out the many emergency checklists created to help you recover and find your way safely home.

Books

- Rony, P., Larsen D. and Titus J., 1976

  The 8080A Bugbook, Microcomputer Interfacing and Programming

  Howard W. Sams & Co., Inc., Indianapolis, IN

  This is the book that introduced me to the world of microprocessors and changed my life forever. No high-level language programming here, just the basics of assembly programming and hardware interfacing. (Too bad this book is already considered museum material; see link below.)

Links

- http://www.bugbookcomputermuseum.com/BugBook-Titles.html

  A link to the "Bugbooks museum"—30 years since the introduction of the Intel 8080 microprocessor and it is like centuries have already passed.

CHAPTER 27

NUMB3RS

27.1 The Plan

In this lesson we will review all the numerical data types offered by the MPLAB® C30 compiler. We will learn how much memory the compiler allocates for the numerical variables and we will investigate the relative efficiency of the routines used to perform arithmetic operations by using the MPLAB SIM Stopwatch as a basic profiling tool. This experience will help you choose the "right" numbers for your embedded-control application, understanding when and how to balance performance and memory resources, real-time constraints and complexity.

27.2 Checklist

This entire lesson will be performed exclusively with software tools including the MPLAB IDE, MPLAB C30 compiler and the MPLAB SIM simulator.

Use the "New Project Set-up" checklist to create a new project called "Numbers" and a new source file called `"numbers.c"`.

27.3 Coding

To review all the data types available, I recommend you take a look at the MPLAB C30 User Guide. You can start in Chapter 28, where you can find a first list of the supported integer types.

As you can see in Table 27.1, there are 10 different integer types as specified in the ANSI C standard including: `char, int, short, long, and long long,` both in the signed (default) and unsigned variant. The table shows the number of bits allocated specifically by the MPLAB C30 compiler for each type, and, for your convenience, spells out the minimum and maximum value that can be represented.

It is expected that, when the type is signed, one bit must be dedicated to the sign itself and the resulting numerical range is therefore halved. It is also interesting to note how the C30 compiler treats `int` and `short` as synonyms by allocating 16 bits for both of them. Both 8- and 16-bit quantities can be processed efficiently by the PIC24 arithmetic and logic unit

Table 27.1: Integer Data Types

Type	Bits	Min	Max
char, signed char	8	−128	127
unsigned char	8	0	255
short, signed short	16	−32768	32767
unsigned short	16	0	65535
int, signed int	16	−32768	32767
unsigned int	16	0	65535
long, signed long	32	-2^{31}	$2^{31}-1$
unsigned long	32	0	$2^{32}-1$
long long**, signed long long**	64	-2^{63}	$2^{63}-1$
unsigned long long**	64	0	$2^{64}-1$
** ANSI-89 extension			

(ALU), so that most of the arithmetic operations can be coded by the compiler using few and efficient instructions. The `long` integers are treated as 32-bit quantities, using four bytes, while the `long long` type (specified by the ANSI C extensions in 1989) requires eight bytes. Operations on long integers are performed by the compiler using short sequences of instructions inserted inline. So, there is a small performance penalty to pay for using `long` integers, and a proportionally larger penalty to pay for `long long` integers, that must be taken into account.

Let's see a first integer example; we'll start by typing the following code:

```
unsigned int i,j,k;

main ()

{
      i = 0x1234;    // assign an initial value to i
      j = 0x5678;    // assign an initial value to j
      k = i * j;     // perform the product and store the result in k
}
```

After building the project (Project→Build All or Ctrl + F10), we can open the Disassembly window ("View→Disassembly Listing") and take a look at the code generated by the compiler. Even without knowing the PIC24 instruction set in detail, we can recognize the two assignments. They are performed by loading the literal values to register w0 first and from there to the memory locations reserved for the variable `i`, and later for variable `j`.

```
          i = 1234;
204D20    mov.w #0x4d2,0x0000          // move literal value to W0
884290    mov.w 0x0000,0x0852          // move data from W0 to i
```

```
          j = 5678;
2162E0    mov.w #0x162e,0x0000          // move literal value to W0
8842A0    mov.w 0x0000,0x0854           // move data from W0 to j

          k = i * j;
804291    mov.w 0x0852,0x0002           // move data from i to W1
8042A0    mov.w 0x0854,0x0000           // move data from j to W0
B98800    mul.ss 0x0002,0x0000,0x0000
8842B0    mov.w 0x0000,0x0856           // move result to k
```

The multiplication is performed by transferring the values from the locations reserved for the two integer variables `i` and `j` back to registers `w0` and `w1`, and then performing a single `mul` instruction. The result, available in `w0`, is stored back into the locations reserved for `k`. Pretty straightforward.

27.3.1 On Optimization (or Lack Thereof)

You will notice how the overall program, as compiled, is somewhat redundant. The value of `j`, for example, is still available in register `w0` when it is reloaded again—just before the multiplication. Can't the compiler see that this operation is unnecessary?

In fact, the compiler does not see things this clearly—its role is to create "safe" code, avoiding (at least initially) any assumption and using standard sequences of instructions. Later on, if the proper optimization options are enabled, a second pass (or more) is performed to remove the redundant code. During the development and debugging phases of a project, though, it is always good practice to disable all optimizations as they might modify the structure of the code being analyzed and render single-stepping and breakpoint placement problematic. In the rest of this book, we will consistently avoid making use of any compiler optimization option; we will verify that the required levels of performance are obtained regardless. As a consequence, you will be able to execute all the examples presented in this and the following chapters using the C30 Compiler Student Edition, which is free and available on the companion CD-ROM.

27.3.2 Testing

To test the code, we can choose to work with the simulator from the Disassembly Listing window itself, single-stepping on each assembly instruction. Or we can choose to work from the C source in the editor window, single-stepping through each C language statement. In both cases, we can:

1. Set the cursor on the first line containing the initialization of the first variable, and perform a Run To Cursor command to let the program initialize and stop the execution just before the first instruction we want to observe.

2. Open the Watch window ("View→Watch") and select `WREG0` in the SFR selection box, then click on the "Add SFR" button.
3. Repeat the operation for `WREG1`.
4. Select `"i"` in the symbol selection box, and click on the "Add Symbol" button.
5. Repeat the operation for `j` and `k`.
6. Use the "Step Over" function to execute the next few program lines, observing the effects on the registers and variables in the Watch window. As we noted before, when the value of a variable in the Watch window changes, it is conveniently highlighted in red.

If you need to repeat the test, perform a Reset ("Debugger→Reset→Processor Reset") and again place the cursor on the first line of code to analyze, followed by a new Run To Cursor command.

27.3.3 Going Long

At this point, modifying only the first line of code, we can change the entire program to perform operations on `long` integer variables.

```
unsigned long i,j,k;

main ()
{
      i = 0x1234; // assign an initial value to i
      j = 0x5678; // assign an initial value to j
      k = i * j;  // perform the product and store the result in k
}
```

Rebuilding the project and switching again to the Disassembly Listing window (if you had the editor window maximized and you did not close the Disassembly Listing window, you could use the Ctrl + Tab command to quickly alternate between the editor and the Disassembly Listing), we can see how the newly generated code is considerably longer than the previous version. While the initializations are still straightforward, the multiplication is now performed using several more instructions.

```
         k = i * j;
8042C1   mov.w 0x0858,0x0002
8042E0   mov.w 0x085c,0x0000
B80A00   mul.uu 0x0002,0x0000,0x0008
8042C1   mov.w 0x0858,0x0002
8042F0   mov.w 0x085e,0x0000
B98800   mul.ss 0x0002,0x0000,0x0000
780105   mov.w 0x000a,0x0004
410100   add.w 0x0004,0x0000,0x0004
```

```
8042E1  mov.w 0x085c,0x0002
8042D0  mov.w 0x085a,0x0000
B98800  mul.ss 0x0002,0x0000,0x0000
410100  add.w 0x0004,0x0000,0x0004
780282  mov.w 0x0004,0x000a
884304  mov.w 0x0008,0x0860
884315  mov.w 0x000a,0x0862
```

The PIC24 arithmetic and logic unit can only process 16 bits at a time, so the 32-bit multiplication is actually performed as a sequence of 16-bit multiplications and additions. The sequence used by the compiler is generated with pretty much the same technique that we learned to use in elementary school, only performed on a word at a time rather than a digit at a time.

27.3.4 Note on the Multiplication of Long Integers

In practice, to perform a 32-bit multiplication using 16-bit instructions, there should be four multiplications and two additions, but you will note how the compiler has actually inserted only three multiplication instructions. What is going on here?

The fact is that multiplying two long integers (32 bits each) will produce a 64-bit wide result. But in the example above, we have specified that the result will be stored in yet another long variable, therefore limiting the result to a maximum of 32 bits. Doing so, we have clearly left the door open for the possibility (not so remote) of an overflow, but we have also given the compiler the permission to ignore the most significant bits of the result. Knowing those bits are not going to be missed, the compiler has eliminated completely the fourth multiplication step—in a way, already optimizing the code.

27.3.5 Long Long Multiplication

Changing the variables declarations to the `long long` integer type (64-bit) is just as simple:

```
unsigned long long i,j,k;

main ()
{
     i = 0x1234;  // assign an initial value to i
     j = 0x5678;  // assign an initial value to j
     k = i * j;   // perform the product and store the result in k
}
```

Recompiling and inspecting the new code in the Disassembly Listing window reveals that this time the compiler has chosen a different approach. Instead of a longer sequence inserted inline, there are now only a few instructions to transfer the data into predefined registers and

there is a call to a subroutine. The subroutine will appear in the disassembly listing, after all the main function code. This subroutine is clearly separated and identified by a comment line that indicates it is part of a library, a module called "`muldi3.c`". The source for this routine is actually available as part of the complete documentation of the C30 compiler and can be found in the subdirectory "`src/libm/src/`" under the same directory tree where the C30 compiler has been installed on your hard disk.

By selecting a subroutine in this case, the compiler has clearly made a compromise. Calling the subroutine means adding a few extra instructions and using extra space on the stack. On the other hand, fewer instructions will be added each time a new multiplication (among `long long` integers) is required in the program; therefore code space will be preserved.

27.3.6 Floating Point

Beyond integer data types, the C30 compiler offers support for a few more data types that can capture fractional values—the floating-point data types (Fig. 27.2). There are three types to choose from, corresponding to two levels of resolution: `float, double` and `long double`.

Notice how the MPLAB C30 compiler, by default, allocates the same number of bits for both the `float` and the `double` types, using the single precision floating-point format defined in the IEEE754 standard. Only the `long double` data type is treated as a true double-precision IEEE754 floating-point type.

Table 27.2: Floating Point Data Types

Type	Bits	E Min	E Max	N Min	N Max
`float`	32	−126	127	2^{-126}	2^{128}
`double*`	32	−126	127	2^{-126}	2^{128}
`long double`	64	−1022	1023	2^{-1022}	2^{1024}

E = Exponent
N = Normalized (approximate)
*double is equivalent long double if `-fno-short-double` is used.

27.4 Notes for C Experts

It is my belief that these floating-point settings were intentionally used by the MPLAB C30 designers to simplify and make more efficient the porting of complex math algorithms to embedded-control target applications. Most of the algorithms and libraries available in literature are designed for the performance and resources of personal computers and workstations, and make use of double-precision floating-point arithmetic whenever possible to maximize accuracy. Most often in embedded control, we are willing to compromise some of that accuracy for the level of performance necessary to achieve real-time response.

If needed, this behavior can be changed either locally, by turning doubles into long doubles in selected cases, or globally, by using special compiler options (open the "Project→Build Options→Project" dialog box, check the Use alternate Setting check box and add "`-fno-short-double`" to the edit box underneath).

Since the PIC24 doesn't have a hardware floating point unit (FPU), all operations on floating-point types must be coded by the compiler using floating-point arithmetic libraries whose size and complexity is considerably larger/higher than any of the integer libraries. You should expect a major performance penalty if you choose to use these data types, but, again, if the problem calls for fractional quantities to be taken into account, the C30 compiler certainly makes dealing with them easy.

Let's modify our previous example to use floating-point variables:

```
float i,j,k;

main ()
{
      i = 12.34;  // assign an initial value to i
      j = 56.78;  // assign an initial value to j
      k = i * j;  // perform the product and store the result in k
}
```

After recompiling and inspecting the Disassembly Listing window, you will notice that the compiler has immediately chosen to use a subroutine instead of inline code.

Changing the program again to use double-precision floating-point type, `long double`, produces very similar results. Only the initial assignments seem to be affected, and all we can see is a subroutine call.

The C compiler makes using any data type so easy that we might be tempted to always use the largest integer or floating-point type available, just to stay on the safe side and avoid the risk of overflows and underflows. On the contrary, choosing the right data type for each application can be critical in embedded control to balance performance and optimize the use of resources. In order to make an informed decision, we need to know more about the level of performance we can expect when choosing the various precision data types.

27.5 Measuring Performance

Let's use what we have learned so far about simulation tools to measure the actual relative performance of the arithmetic libraries (integer and floating-point) used by the C30 compiler. We can start using the software simulator's (MPLAB SIM) built-in Stopwatch tool, with the following code:

```
//
// Numbers
//
```

```
int i1, i2, i3;
long l1, l2, l3;
long long ll1, ll2, ll3;
float f1,f2, f3;
long double d1, d2, d3;

main ()
{

      i1 = 1234;    // testing integers (16-bit)
      i2 = 5678;
      i3= i1 * i2;  // 1. int multiplication

      l1 = 1234;    // testing long integers (32-bit)
      l2 = 5678;
      l3= l1 * l2;  // 2. long multiplication

      ll1 = 1234;   // testing long long integers (64-bit)
      ll2 = 5678;
      ll3= ll1 * ll2; // 3.

      f1 = 12.34;   // testing single precision (32-bit) floating point
      f2 = 56.78;
      f3= f1 * f2;  // 4. single precision multiplication

      d1 = 12.34;   // testing double precision (64-bit) floating point
      d2 = 56.78;
      d3= d1 * d2;  // 5. double precision multiplication
}
```

After compiling and linking the project, we can set the cursor on the line containing the first integer multiplication (`// 1.`) in the editor window and perform a Run To Cursor, to position the program counter for our test. Open the Stopwatch window ("Debugger→Stopwatch") and position the window according to your preferences (personally I like it docked to the bottom of the screen so that it does not overlap with the editor window and it is always visible and accessible).

Zero the Stopwatch timer and execute a Step-Over command ("Debug→StepOver", or press F8). As the Simulator completes updating the Stopwatch window, you can manually record the execution time required to perform the integer operation. The time is provided by the simulator in the form of a cycle count along with an indication in milliseconds derived by the cycle count multiplied by the simulated clock frequency, a parameter specified in the Debugger Settings ("Debugger→Settings→Osc/Trace" tab).

Proceed by setting the cursor over the next multiplication (`// 2.`), and execute a new Run To Cursor command or simply continue Stepping until you reach it. Again zero the Stopwatch, execute a Step-Over and record the second time. Continue until all five types have been tested.

In Table 27.3, I have recorded the results (cycle counts) in the first column and then added more columns to show the relative performance ratios, obtained by dividing the cycle count of each row by the cycle count recorded for the reference type. Don't be alarmed if you happen to record different values; several factors can affect the measure. Future versions of the compiler could possibly use more efficient libraries, and/or optimization features could be introduced or enabled at the time of testing.

Table 27.3: Relative Performance Test Results Using MPLAB C30 rev. 1.30 (All Optimizations Disabled)

Multiplication Test	Cycle Count	Performance Relative to		
		`int`	`long`	`Float`
Integer	4	1	–	–
Long Integer	15	3.75	1	–
Long-Long Integer	99	24.75	6.6	–
Single Precision f.p.	121	30	8	1
Double Precision f.p.	317	79	21	2.6

Keep in mind that this type of test lacks any of the rigorousness required by a true performance benchmark. What we are looking for here is just a basic understanding of the impact on the performance we can expect from choosing to perform our calculations with one data type versus another. We are looking for the big picture—relative orders of magnitude. For that purpose, the table we just obtained can already give us some interesting indications.

As expected, 16-bit operations appear to be the fastest. Long-integer (32-bit) multiplications are about four times slower, while long-long-integer (64-bit) multiplications are one order of magnitude slower. Again, it was expected that single precision floating-point operations would require more effort than integer operations. Multiplying a 32-bit integer is only about four times slower than multiplying a 16-bit integer. However, multiplying 32-bit floating-point numbers is more than 30 times slower than multiplying 16-bit integers. That means it is eight times slower than the corresponding 32-bit integer multiplication, or about an order of magnitude. Going to double-precision floating-point (64-bit) though, only doubles the number of cycles. This tells us that, apparently, the double-precision floating-point libraries used by the compiler are more efficient than the corresponding 64-bit integer libraries.

So, when should we use floating point and when should we use integer arithmetic?

Beyond the obvious, from the little we have learned so far we can perhaps extract the following rules:

1. Use integers every time you can (i.e., when fractions are not required, or the algorithm can be rewritten for integer arithmetic).

2. Use the smallest integer type that will not produce an overflow or underflow.
3. If you have to use a floating-point type (fractions are required), expect an order-of-magnitude reduction in the performance of the compiled program.
4. Double-precision floating-point (`long double`) seems to only reduce the performance further by a factor of two.

Keep in mind also that floating-point types offer the largest value ranges, but also are always introducing approximations. As a consequence, floating-point types are not recommended for financial calculations. Use `long` or `long long` integers instead, and perform all operations in cents (instead of dollars and fractions).

27.6 Review

In this lesson, we have learned not only what data types are available and how much memory is allocated to them, but also how they affect the resulting compiled program—code size and the execution speed. We used the MPLAB SIM simulator Stopwatch function to measure the number of instruction cycles (and therefore time) required for the execution of a series of code segments. Some of the information gathered will, hopefully, be useful to guide our actions in the future when balancing our needs for precision and performance in embedded-control applications.

27.6.1 Notes for Assembly Experts

The brave few assembly experts that have attempted to deal with floating-point numbers in their applications tend to be extremely pleased and forever thankful for the great simplification achieved by the use of the C compiler. Single- or double-precision arithmetic becomes just as easy to code as integer arithmetic has always been.

When using integer numbers, though, there is sometimes a sense of loss of control, as the compiler hides the details of the implementation and some operations might become obscure or much less intuitive/readable. Here are some examples of conversion and byte-manipulation operations that can induce some anxiety:

1. Converting an integer type into a smaller/larger one.
2. Extracting or setting the most or least significant byte of a 16-bit data type.
3. Extracting or setting one bit out of an integer variable.

The C language offers convenient mechanisms for covering all such cases via implicit type conversions as in:

```
int   i;    // 16-bit
long  l;    // 32-bit
```

```
l = i;      // the value of i is transferred into the two LSB of l
            // the two MSB of l are cleared
```

Explicit conversions might be required (called "type casting") in some cases where the compiler would otherwise assume an error, as in:

```
int     i;   // 16-bit
long    l;   // 32-bit
i=(int) l;   // (int) is a type cast that results in the two
             // MSB of l
             // to be discarded as l is treated as a 16-bit value
```

Bit fields are used to cover the conversion to and from integer types that are smaller than one byte. Bit fields are treated by the MPLAB C30 compiler with great efficiency and will result in the use of bit-manipulation instructions whenever possible. The PIC24 library files contain numerous examples of definitions of bit fields for the manipulation of all the control bits in the peripheral and the core special-function registers.

Here is an example extracted from the include file used in our project, where the Timer1 control register `T1CON` is defined and each individual control bit is exposed in a structure defined as `T1CONbits`:

```
extern unsigned int T1CON;
extern union {
    struct {
      unsigned :1;
      unsigned TCS:1;
      unsigned TSYNC:1;
      unsigned :1;
      unsigned TCKPS0:1;
      unsigned TCKPS1:1;
      unsigned TGATE:1;
      unsigned :6;
      unsigned TSIDL:1;
      unsigned :1;
      unsigned TON:1;
    };
    struct {
    unsigned :4;
    unsigned TCKPS:2;
  };
} T1CONbits;
```

27.6.2 Notes for PIC Microcontroller Experts

The PIC microcontroller user, familiar with the 8-bit PIC microcontrollers and their respective compilers, will notice a considerable improvement in the performance, both with integer arithmetic and floating-point arithmetic. The 16-bit ALU available in the PIC24 architecture is clearly providing a great advantage by manipulating twice the number of bits per cycle, but the performance improvement is further accentuated by the availability of up to eight working registers, which make the coding of critical arithmetic routines and numerical algorithms more efficient.

27.6.3 Tips and Tricks

27.6.3.1 Math Libraries

The MPLAB C30 compiler supports several standard ANSI C libraries including:

- "`limits.h`", which contains many useful macros defining implementation-dependent limits, such as, for example, the number of bits composing a char type (`CHAR_BIT`) or the largest integer value (`INT_MAX`).
- "`float.h`" , which contains similar implementation-dependent limits for floating-point data types, such as, for example, the largest exponent for a single-precision floating-point variable (`FLT_MAX_EXP`).
- "`math.h`", which contains trigonometric functions, rounding functions, logarithms and exponentials.

27.6.3.2 Complex Data Types

The MPLAB C30 compiler supports `complex` data types, as an extension of both integer and floating-point types. Here is an example declaration for a single-precision floating-point type:

```
__complex__ float z;
```

Notice the use of a double underscore before and after the keyword `complex`. The variable `z` so defined has now a real and an imaginary part that can be individually addressed using the syntax: `__real__ z` and `__imag__ z`, respectively.

Similarly, the next declaration produces a complex variable of 16-bit integer type:

```
__complex__ int x;
```

Complex constants are easily created adding the suffix "i" or "j" as in the following examples:

```
x = 2 + 3j;
z = 2.0f + 3.0fj;
```

All standard arithmetic operations (+, −, *, /) are performed correctly on complex data types; additionally, the "~" operator produces the complex conjugate.

Complex types could be pretty handy in some types of applications, making the code more readable and helping avoid trivial errors. Unfortunately, as of this writing, the MPLAB IDE support of complex variables during debugging is only partial, giving access only to the "real" part through the Watch window and the mouse-over function.

Links

- http://en.wikipedia.org/wiki/Taylor_series

  If you are curious how the C compiler can approximate some of the functions in the math library.

This page intentionally left blank

CHAPTER 28

Interrupts

28.1 The Plan

In this lesson we will see how the MPLAB® C30 compiler allows us to easily manage the interrupt mechanisms offered by the PIC24 microcontroller architecture. After a brief review of some of the C language extensions and some practical considerations, we will present a short example of how to use the secondary (low-frequency) oscillator to maintain a real-time clock.

28.2 Checklist

This entire lesson will be performed exclusively with software tools, including the MPLAB IDE, MPLAB C30 compiler and the MPLAB SIM simulator.

Use the "New Project Set-up" checklist to create a new project called "Interrupts" and a new source file similarly called "`interrupts.c`".

28.3 Coding

An interrupt is an internal or external event that requires quick attention from the CPU. The PIC24 architecture provides a rich interrupt system that can manage as many as 118 distinct sources of interrupts. Each interrupt source can have a unique piece of code, called the Interrupt Service Routine (ISR) directly associated via a pointer, also called a "vector," to provide the required response action. Interrupts can be completely asynchronous with the execution flow of the main program. They can be triggered at any point in time and in an unpredictable order. Responding quickly to interrupts is essential to allow prompt reaction to the trigger event and a fast return to the main program execution flow. Therefore, the goal is to minimize the interrupt latency, defined as the time between the triggering event and the execution of the first instruction of the Interrupt Service Routine (ISR). In the PIC24 architecture, the latency is not only very short but it is also fixed for each given interrupt source—only three instruction cycles for internal events and four instruction cycles for external events. This is a highly desirable quality that makes the PIC24 interrupt management superior to most other architectures.

The MPLAB C30 compiler helps manage the complexity of the interrupt system by providing a few language extensions. The PIC24 keeps all interrupt vectors in one large Interrupt Vector Table (IVT) and the MPLAB C30 compiler can automatically associate interrupt vectors with "special" user-defined C functions as long as a few limitations are kept in consideration, such as:

- They are not supposed to return any value (use type `void`).
- No parameter can be passed to the function (use parameter `void`).
- They cannot be called directly by other functions.
- Ideally, they should not call any other function.

The first three limitations should be obvious given the nature of the interrupt mechanism—since it is triggered by an external event, there cannot be parameters or a return value because there is no proper function call in the first place. The last limitation is more of a recommendation to keep in mind for efficiency considerations.

The following example illustrates the syntax that could be used to associate a function to the Timer1 interrupt vector:

```
void __attribute__ (( interrupt)) _T1Interrupt ( void)
{
  // interrupt service routine code here...
} // _InterruptVector
```

The function name `_T1Interrupt` was not an arbitrary choice, but is actually the predefined identifier for the Timer 1 interrupt as found in the Interrupt Vectors Table of the PIC24, (defined in the datasheet) and as coded in the linker script, the "`.gld`" file loaded for the current project.

The `__attribute__(( ))` mechanism is used by the C30 compiler in this and many other circumstances as a way to specify special features such as a C language extension. Personally, I find this syntax too lengthy and hard to read. I recommend the use of a couple of macros that can be found in each PIC24 include ("`.h`") files and that greatly improve the code readability. In the following example, the `_ISR` macro is used to the same effect as the previous code snippet:

```
void _ISR _T1Interrupt (void)
{
  // interrupt service routine code here...
} // _InterruptVector
```

From Tables 28.1a and 28.1b, taken from the PIC24FJ128GA010 family datasheet, you can see which events can be used to trigger an interrupt. Among the external sources available for the PIC24FJ128GA010, there are:

- 5 × External pins with level trigger detection
- 22 × External pins connected to the Change Notification module

- 5 × Input Capture modules
- 5 × Output Compare modules
- 2 × Serial port interfaces (UARTs)
- 4 × Synchronous serial interfaces (SPI and I^2C™)
- Parallel Master Port

Among the internal sources we count:

- 5 × 16-bit Timers
- 1 × Analog-to-Digital Converter
- 1 × Analog Comparators module
- 1 × Real-time Clock and Calendar
- 1 × CRC generator

Many of these sources in their turn can generate several different interrupts. For example, a serial-port interface peripheral (UART) can generate three type of interrupts:

- When new data has been received and is available in the receive buffer for processing.
- When data in the transmit buffer has been sent and the buffer is empty, ready and available, to transmit more.
- When an error condition has been generated and action might be required to re-establish communication.

Each interrupt source also has five associated control bits, allocated in various special-function registers (see Table 28.1):

- The Interrupt Enable bit (typically represented with a suffix `-IE`):
  – When cleared, the specific trigger event is prevented from generating interrupts.
  – When set, it allows the interrupt to be processed.

At power on, all interrupt sources are disabled by default.

- The Interrupt Flag (typically represented with a suffix `-IF`). This single bit of data is set each time the specific trigger event is activated, independently by the status of the enable bit. Notice how, once set, it must be cleared (manually) by the user. In other words, it must be cleared before exiting the interrupt service routine, or the same interrupt service routine will be immediately called again.

Table 28.1: Interrupt Vectors as Implemented in the PIC24FJ128GA010 Family

Interrupt Source	Vector Number	IVT Address	AIVT Address	Interrupt Bit Locations		
				Flag	Enable	Priority
ADC1 Conversion Done	13	00002Eh	00012Eh	IFS0<13>	IEC0<13>	IPC3<6:4>
Comparator Event	18	000038h	000138h	IFS1<2>	IEC1<2>	IPC4<10:8>
CRC Generator	67	00009Ah	00019Ah	IFS4<3>	IEC4<3>	IPC16<14:12>
External Interrupt 0	0	000014h	000114h	IFS0<0>	IEC0<0>	IPC0<2:0>
External Interrupt 1	20	00003Ch	00013Ch	IFS1<4>	IEC1<4>	IPC5<2:0>
External Interrupt 2	29	00004Eh	00014Eh	IFS1<13>	IEC1<13>	IPC7<6:4>
External Interrupt 3	53	00007Eh	00017Eh	IFS3<5>	IEC3<5>	IPC13<6:4>
External Interrupt 4	54	000080h	000180h	IFS3<6>	IEC3<6>	IPC13<10:8>
I2C1 Master Event	17	000036h	000136h	IFS1<1>	IEC1<1>	IPC4<6:4>
I2C1 Slave Event	16	000034h	000034h	IFS1<0>	IEC1<0>	IPC4<2:0>
I2C2 Master Event	50	000078h	000178h	IFS3<2>	IEC3<2>	IPC12<10:8>
I2C2 Slave Event	49	000076h	000176h	IFS3<1>	IEC3<1>	IPC12<6:4>
Input Capture 1	1	000016h	000116h	IFS0<1>	IEC0<1>	IPC0<6:4>
Input Capture 2	5	00001Eh	00011Eh	IFS0<5>	IEC0<5>	IPC1<6:4>
Input Capture 3	37	00005Eh	00015Eh	IFS2<5>	IEC2<5>	IPC9<6:4>
Input Capture 4	38	000060h	000160h	IFS2<6>	IEC2<6>	IPC9<10:8>
Input Capture 5	39	000062h	000162h	IFS2<7>	IEC2<7>	IPC9<14:12>
Input Change Notification	19	00003Ah	00013Ah	IFS1<3>	IEC1<3>	IPC4<14:12>
Output Compare 1	2	000018h	000118h	IFS0<2>	IEC0<2>	IPC0<10:8>
Output Compare 2	6	000020h	000120h	IFS0<6>	IEC0<6>	IPC1<10:8>
Output Compare 3	25	000046h	000146h	IFS1<9>	IEC1<9>	IPC6<6:4>
Output Compare 4	26	000048h	000148h	IFS1<10>	IEC1<10>	IPC6<10:8>
Output Compare 5	41	000066h	000166h	IFS2<9>	IEC2<9>	IPC10<6:4>
Parallel Master Port	45	00006Eh	00016Eh	IFS2<13>	IEC2<13>	IPC11<6:4>
Real-Time Clock/ Calendar	62	000090h	000190h	IFS3<14>	IEC3<13>	IPC15<10:8>
SPI1 Error	9	000026h	000126h	IFS0<9>	IEC0<9>	IPC2<6:4>
SPI1 Event	10	000028h	000128h	IFS0<10>	IEC0<10>	IPC2<10:8>
SPI2 Error	32	000054h	000154h	IFS2<0>	IEC0<0>	IPC8<2:0>
SPI2 Event	33	000056h	000156h	IFS2<1>	IEC2<1>	IPC8<6:4>
Timer1	3	00001Ah	00011Ah	IFS0<3>	IEC0<3>	IPC0<14:12>
Timer2	7	000022h	000122h	IFS0<7>	IEC0<7>	IPC1<14:12>
Timer3	8	000024h	000124h	IFS0<8>	IEC0<8>	IPC2<2:0>

Table 28.1: Continued

Interrupt Source	Vector Number	IVT Address	AIVT Address	Interrupt Bit Locations		
				Flag	Enable	Priority
Timer4	27	00004Ah	00014Ah	IFS1<11>	IEC1<11>	IPC6<14:12>
Timer5	28	00004Ch	00014Ch	IFS1<12>	IEC1<12>	IPC7<2:0>
UART1 Error	65	000096h	000196h	IFS4<1>	IEC4<1>	IPC16<6:4>
UART1 Receiver	11	00002Ah	00012Ah	IFS0<11>	IEC0<11>	IPC2<14:12>
UART1 Transmitter	12	00002Ch	00012Ch	IFS0<12>	IEC0<12>	IPC3<2:0>
UART2 Error	66	000098h	000198h	IFS4<2>	IEC4<2>	IPC16<10:8>
UART2 Receiver	30	000050h	000150h	IFS1<14>	IEC1<14>	IPC7<10:8>
UART2 Transmitter	31	000052h	000152h	IFS1<15>	IEC1<15>	IPC7<14:12>

- The priority level (typically represented with a suffix `-IP`). Interrupts can have up to 7 levels of priority. Should two interrupt events occur at the same time, the highest priority event will be served first. Three bits encode the priority level of each interrupt source. At any given point in time, the PIC24 execution priority level value is kept in the `SR` register in three bits referred to as `IPL0..IPL2`. Interrupts with a priority level lower than the current value of `IPL` will be ignored. At power on, all interrupt sources are assigned a default level of four and the processor priority is initially set at level zero.

Within an assigned priority level there is also a relative (default) priority among the various sources in the fixed order of appearance in the IVT table.

28.3.1 Nesting of Interrupts

Interrupts can be nested, so that a lower-priority interrupt service routine can be interrupted by a higher-priority routine. This behavior can be controlled by the `NSTDIS` bit in the `INTCON1` register of the PIC24.

When the `NSTDIS` bit is set, as soon as an interrupt is received the priority level of the processor (`IPL`) is set to the highest level (7) independently of the specific interrupt level assigned to the event. This prevents new interrupts from being serviced until the present one is completed. In other words, when the `NSTDIS` bit is set, the priority level of each interrupt is used only to resolve conflicts, should multiple interrupts occur simultaneously, and all interrupts are serviced sequentially.

28.3.2 Traps

Eight additional vectors occupy the first locations on top of the IVT table (Table 28.2). They are used to capture special error conditions such as a failure of the selected CPU oscillator, an incorrect address (word access to odd address), stack underflow, or a divide by zero (math error).

Table 28.2: TRAP Vector Details

Vector Number	IVT Address	Trap Source
0	000004h	Reserved
1	000006h	Oscillator Failure
2	000008h	Address Error
3	00000Ah	Stack Error
4	00000Ch	Math Error
5	00000Eh	Reserved
6	000010h	Reserved
7	000012h	Reserved

Since these types of errors have generally fatal consequences for a running application, they have been assigned fixed priority levels above the seven basic levels available to all other interrupts. This also means that they cannot be inadvertently masked (or delayed by the `NSTDIS` mechanism) and it provides an extra level of security for the application. The MPLAB C30 compiler associates all trap vectors with a single default routine that will produce a processor reset. You can change such behavior using the same technique illustrated for all generic interrupt service routines.

28.3.3 A Template and an Example for Timer1 Interrupt

This all might seem extremely complicated, but we will quickly see that, by following some simple guidelines, we can put it to use in no time. Let's create a template, which we will reuse in future practical examples that demonstrate the use of the Timer1 peripheral module as the interrupt source. We will start by writing the interrupt service routine function:

```
// 1. Timer1 interrupt service routine
void _ISR _T1Interrupt( void)
{

     // insert your code here
     // ...

     // remember to clear the interrupt flag before exit
     _T1IF = 0;

} //T1Interrupt
```

We used the `_ISR` macro just like before and made sure to declare the function type and parameters as `void`. Remembering to clear the interrupt flag (`_T1IF`) before exiting the function is extremely important, as we have seen. In general, the application code should be very concise.

The goal of any interrupt service routine is to perform a simple task quickly and efficiently in rapid response to an event. As a general rule, I would say that if you should find yourself writing more than a page of code (or contemplating calling other functions) you should most probably stop and reconsider the goals and the structure of your application. Lengthy calculations have a place in the main function and specifically in the main loop, not inside an interrupt service routine where time is at premium.

Let's complete the template with a few lines of code that we will add to the main function:

```
main()
{
  // 2. initializations
  _T1IP = 4;       // set Timer1 priority, (4 is the default value)
  TMR1 = 0;        // clear the timer
  PR1 = period-1; // set the period register

  // 2.1 configure Timer1 module clock source and sync setting
  T1CON = 0x8000; // check T1CON register options

  // 2.2 init the Timer1 Interrupt control bits
  _T1IF = 0;       // clear the interrupt flag, before
  _T1IE = 1;       // enable the T1 interrupt source

  // 2.3 init the processor priority level
  _IP = 0;         // 0 is the default value

  // 3. the main loop
  while( 1)
  {

                   // your main code here...
  }                // main loop

} // main
```

In 2, we assign a priority level to the Timer1 interrupt source, although this might not be strictly necessary, as we know that all interrupt sources are assigned a default level-four priority at power on. We also clear the timer and assign a value to its period register.

In 2.1, we complete the configuration of the timer module, by turning the timer on with the chosen settings.

In 2.2, we clear the interrupt flag just before enabling the interrupt source.

The interrupt-trigger event for the timer module is defined as the instant the timer value reaches the value assigned to the period register. In that instant, the interrupt flag is set and the timer

is reset to begin a new cycle. If the interrupt-enable bit is set as well, and the priority level is higher than the processor current priority (`_IP`), the interrupt service function is immediately called.

In 2.3, we initialize the processor priority level although, once more, this is not strictly necessary as the processor priority is initialized to zero by default at power on.

In 3.0, we will insert the main loop code. If everything goes as planned, the main loop will execute continuously, interrupted periodically by a brief call to the interrupt service routine.

28.3.4 *A Real Example with Timer1*

By adding only a couple of lines of code, we can turn this template into a more practical example where Timer1 is used to maintain a real-time clock, with tenths of a second, seconds and minutes. As a simple visual feedback we can use the lower 8 bits of PORTA as a binary display showing the seconds running. Here is what we need to add:

- Before 1., add the declaration of a few new integer variables that will act as the seconds and minutes counters:

```
int dSec = 0;
int Sec = 0;
int Min = 0;
```

- In 1.2, have the interrupt service routine increment the counter:

```
dSec++;
```

  A few additional lines of code will be added to take care of the carry-over into seconds and minutes.

- In 2, set the period register for Timer1 to a value that (assuming a 32-MHz clock) will give us a tenth of a second period between interrupts.

```
PR1 = 25000-1; // 25,000 * 64 * 1 cycle (62.5ns) = 0.1 s
```

- Set PORTA lsb as output:

```
TRISA = 0xff00;
```

- In 2.1, set the Timer1 prescaler to 1:64 to help achieve the desired period.

```
T1CON = 0x8020;
```

- In 3., add code inside the main loop to continuously refresh the content of `PORTA` (lsb) with the current value of the milliseconds counter.

```
PORTA = Sec;
```

The new project is ready to build:

```
#include <p24fj128ga010.h>

int dSec = 0;
int Sec = 0;
int Min = 0;

// 1. Timer1 interrupt service routine
void _ISR _T1Interrupt( void)
{
   // 1.1 your code here
   dSec++;                // increment the tens of a second counter
   if ( dSec > 9)                // 10 tens in a second
   {
          dSec = 0;
          Sec++;                 // increment the minute counter

          if ( Sec > 59)  // 60 seconds make a minute
          {
                 Sec = 0;
   // 1.2 increment the minute counter
                 Min++;

                 if ( Min > 59)// 59 minutes in an hour
                        Min = 0;
          } // minutes
   } // seconds

   // 1.3 clear the interrupt flag
   _T1IF = 0;

} //T1Interrupt

main()
{
   // 2. init Timer 1, T1ON, 1:1 prescaler, internal clock source
   _T1IP = 4;        // this is the default value anyway
   TMR1 = 0;                 // clear the timer
   PR1 = 25000-1;    // set the period register
   TRISA = 0xff00;           // set PORTA lsb as output

   // 2.1 configure Timer1 module
   T1CON = 0x8020;   // enabled, prescaler 1:64, internal clock

   // 2.2 init the Timer 1 Interrupt, clear the flag, enable the source
   _T1IF = 0;
   _T1IE = 1;

   // 2.3 init the processor priority level
   _IP = 0;          // this is the default value anyway
```

```
        // 3. main loop
        while( 1)
        {

                // your main code here
                PORTA = Sec;

         } // main loop

} // main
```

28.3.5 Testing the Timer1 Interrupt

1. Open the Watch window (dock it to your favorite spot).
2. Add the following variables:
   - `dSec`, select from the Symbol pulldown box, then click on Add
   - `TMR1`, select from the SFR pulldown box, then click on Add
   - `SR`, select from the SFR pulldown box, then click on Add
3. Open the Simulator Stopwatch window ("Debugger→StopWatch").
4. Set a breakpoint on the first instruction of the interrupt response routine after 1.1.
5. Set the cursor on the line and from the right click menu select: Set Breakpoint, or simply double click. By setting the breakpoint here, we will be able to observe whether the interrupt is actually being triggered.
6. Execute a Run ("Debugger→Run" or F9). The simulation should stop quickly, with the program counter cursor (the green arrow) pointing right at the breakpoint inside the interrupt service routine.

So we did stop inside the interrupt service routine! This means that the trigger event was activated; that is, the Timer1 reached a count of 24,999 (remember though that the Timer1 count starts with 0, therefore 25,000 counts have been performed) which, multiplied by the prescaler value, means that 25,000 × 64 or exactly 1.6 million cycles, have elapsed.

The Stopwatch window will confirm that the total number of cycles executed so far is, in fact, slightly higher than 1.6 million. The Stopwatch count includes the time required by the initialization part of our program too. At the PIC24's execution rate (16 million instructions per second or 62.5 ns per cycle) this all happened in a tenth of a second!

From the Watch window, we can now observe the current value of processor priority level (`IP`). Since we are inside an interrupt service routine that was configured to operate at level

four, we should be able to verify that bits 3, 4 and 5 of the status register (`SR`) contain exactly this value. For convenience, the MPLAB IDE shows the completely decoded contents of the status register in a small box, as part of the status bar located at the bottom of the main window.

In Figure 28.1, I have circled the IP indication in the status bar (showing IP4 to indicate interrupt priority level four) as well as the SR register content and the Stopwatch actual value (in milliseconds). Single stepping from the current position (using either the StepOver or the StepIn commands), we can monitor the execution of the next few instructions inside the interrupt service routine. Upon its completion, we can observe how the priority level returns back to the initial value—look for the IP0 indication in the status bar and the SR register bits 5, 6 and 7 to be cleared.

7. After executing another Run command, we should find ourselves again with the program counter (represented graphically by the green arrow) pointing inside the interrupt service routine. This time, you will notice how exactly 1.6 million cycles have been added to the previous count.
8. Add the Sec and Min variables to the Watch window.
9. Execute the Run command a few more times to verify that, after 10 iterations, the seconds counter is incremented.

To test the minutes increment, you might want to remove the current breakpoint and place a new one a few lines below—otherwise you will have to execute the Run command exactly 600 times!

10. Place the new breakpoint on the `Min++` statement in 1.2.
11. Execute Run once and observe that the seconds counter has already been cleared.
12. Execute the StepOver command once and the minute counter will be incremented.

The interrupt routine has been executed 600 times, in total, at precise intervals of one tenth of a second. Meanwhile, the code present in the main loop has been executed continuously to use the vast majority of the grand total of 960 million cycles. In all honesty, our demo program did not make much use of all those cycles—wasting them all in a continuous update of the PORTA content. In a real application, we could have performed a lot of work, all the while maintaining a precise real-time clock count.

28.3.6 The Secondary Oscillator

There is another feature of the PIC24 Timer1 module (common to all previous generations of 8-bit PIC® microcontrollers) that we could have used to obtain a real-time clock. In fact, there is a low-frequency oscillator (known as the secondary oscillator) that can be used to feed just

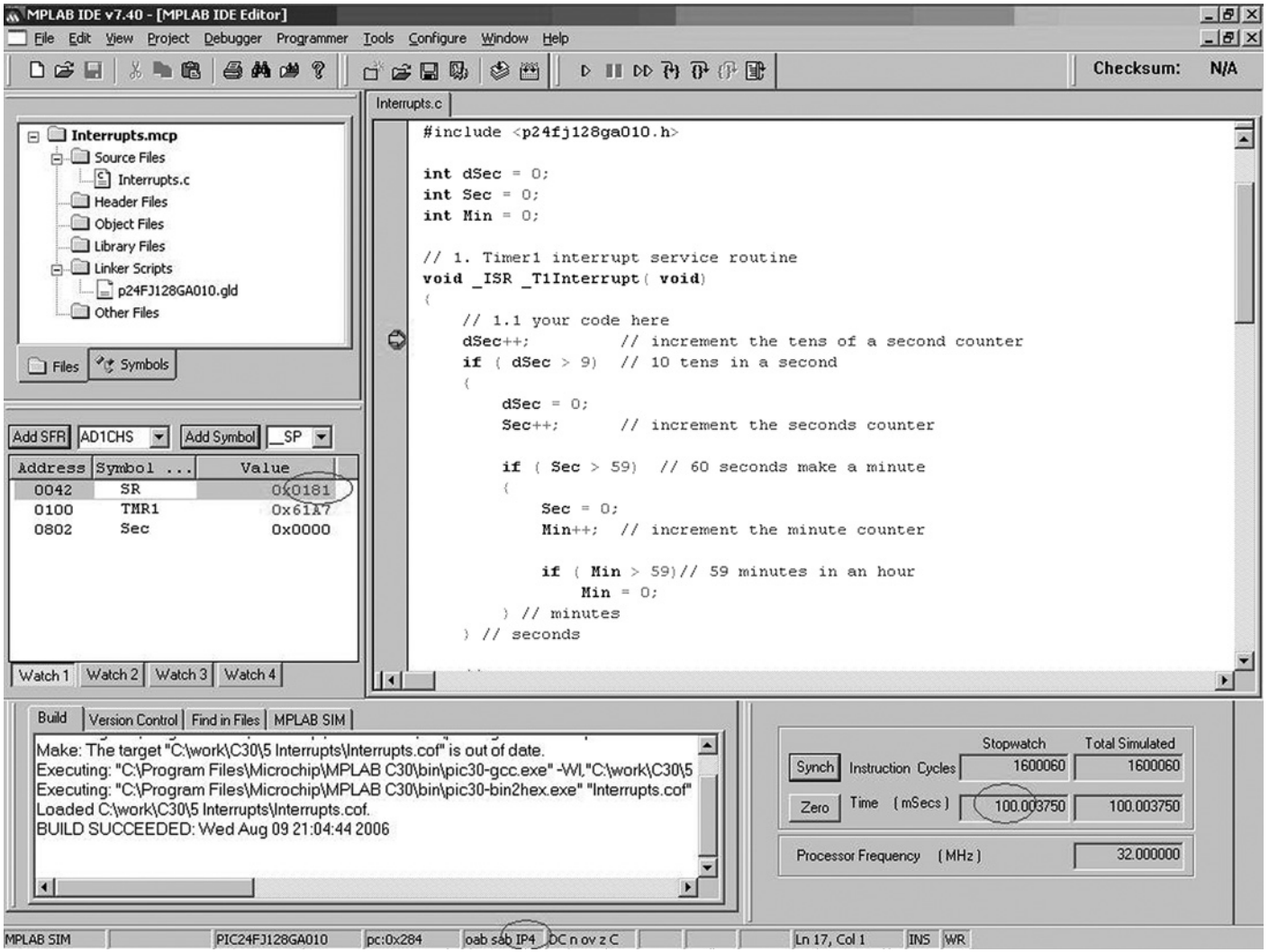


Figure 28.1: Screenshot Showing the Processor Status after Timer1 Interrupt

the Timer1 module instead of the high-frequency main clock. Since it is designed for low-frequency operation (typically it is used in conjunction with an inexpensive 32,768-Hz crystal), it requires very little power to operate. And since it is independent from the main clock circuit, it can be maintained in operation when the main clock is disabled and the processor enters one of the many possible low-power modes. In fact, the secondary oscillator is an essential part for many of those low-power modes. In some cases it is used to replace the main clock, while in others it remains active only to feed the Timer1 or a selected group of peripherals.

To convert our previous example for use with the secondary oscillator, we will need to perform only a few minor modifications, such as:

- change the interrupt routine to count only seconds and minutes (the much slower clock rate does not require the extra step for the tenth of a second).

```
// 1. Timer1 interrupt service routine
void _ISR _T1Interrupt( void)
{
    // 1.1 clear the interrupt flag
    _T1IF = 0;

    // 1.2 your code here
    Sec++;                      // increment the seconds counter

    if ( Sec > 59)              // 60 seconds make a minute
    {
          Sec = 0;
          Min++;                // increment the minute counter

          if ( Min > 59)        // 59 minutes in an hour
               Min = 0;
    } // minutes
} //T1Interrupt
```

- in 2, change the period register to generate one interrupt every 32,768 cycles.

```
PR1 = 32768-1;// set the period register
```

- in 2.1, change the Timer1 configuration word (the prescaler is not required anymore).

```
T1CON = 0x8002; // enabled, prescaler 1:1, use secondary oscillator
```

Unfortunately, you will not be able to immediately test this new configuration with the simulator, since the secondary oscillator input is not automatically simulated.

In a later lesson, we will learn how a new set of tools will help us to generate a stimulus file that could be used to provide a convenient emulation of a 32-kHz crystal connected to the T1CK and SOSCI pins of the PIC24.

28.3.7 The Real-time Clock Calendar (RTCC)

Building on the previous two examples, we could evolve the real-time clock implementations to include the complete functionality of a calendar, adding the count of days, day of the week, months and years. These few new lines of code would be executed only once a day, once a month or once a year, and therefore would produce no decrease in the performance of the overall application whatsoever. Although it would be somewhat entertaining to develop such code, considering lapsed years and working out all the details, the PIC24FJ128GA010 already has a complete Real-time Clock and Calendar (RTCC) module built in and ready for use. How convenient! Not only does it feed from the same low-power secondary oscillator, but it comes with all the bells and whistles, including a built-in Alarm function that can generate interrupts. In other words, once the module is initialized, it is possible to activate the RTCC alarm and wait for an interrupt to be generated, for example, on the exact month, day, hour, minute and second desired once a year (or if set on February 29th, even once every four years!).

This is what the interrupt service routine would look like:

```
// 1. RTCC interrupt service routine
void _ISR _RTCCInterrupt( void)

{
      // 1.1 clear the interrupt flag
      _RTCIF = 0;

      // 1.2 your code here, will be executed only once a year
      // that is once every 365 x 24 x 60 x 60 x 16,000,000 MCU
      // cycles
// that is once every 504,576,000,000,000 MCU cycles

} // RTCCInterrupt
```

28.4 Managing Multiple Interrupts

It is typical of an embedded-control application to require several interrupt sources to be serviced. For example, a serial communication port might require periodic attention at the same time that a PWM module is active and requires periodic updates to control an analog output. Multiple timer modules might be used simultaneously to produce pulsed outputs, while multiple inputs could be sampled by the analog-to-digital converter and their values would need to be buffered. There is almost no limit to the number of things that can be done with 118 interrupt sources available. At the same time, there is no limit to the complexity of the bugs that can be generated, thanks to the same sophisticated mechanisms, if a little discipline and some common sense are not applied.

Here are some of the rules to keep in mind:

1. Keep it short and simple. Make sure the interrupt routines are the shortest/fastest possible, and under no circumstances should you attempt to perform any processing of the incoming data. Limit the activity to buffering, transferring and flagging.
2. Use the priority levels to determine which event deserves to be serviced first, in case two events are triggered simultaneously.
3. But consider very carefully whether you want to face the additional complexity and headaches that result from enabling the use of nested interrupt calls. After all, if the interrupt service routines are short and efficient, the extra latency introduced by waiting for the current interrupt to be completed before a new one is serviced is going to be extremely small. If you determine that you don't need it that bad, make sure the `NSTDIS` control bit is set to prevent nesting:

```
_NSTDIS = 1; // disable interrupt nesting (default)
```

28.5 Review

In this lesson, we have seen how an interrupt service routine can be simple to code, thanks to the language extensions built into the C30 compiler and the powerful interrupt-control mechanisms offered by the PIC24 architecture. Interrupts can be an extremely efficient tool in the hands of the embedded-control programmer, to manage multiple tasks while maintaining precious timing and resources constraints. At the same time, they can be an extremely powerful source of trouble. In the PIC24 reference manual and the MPLAB C30 User Guide, you will find more useful information than we could possibly cram into one single lesson. Finally, in this lesson we took the opportunity to learn more about the uses of Timer1 and the secondary oscillator, and we got a glimpse of the features of the new Real-Time Clock and Calendar (RTCC) module.

28.5.1 Notes for C Experts

The interrupt vector table (IVT) is an essential part of the `C0` code segment for the PIC24. Actually two copies of it are required to be present in the first 256 locations of the program memory—one is used during normal program execution and the second (or Alternate IVT) during debugging. These tables account for most of the size of the `C0` code in all the examples we have been developing in these first five lessons. Subtract 256 words (or 768 bytes) from the file size of each example to obtain the "net" code size.

28.5.2 Notes for Assembly Experts

The `_ISRFAST` macro can be used to declare a function as an interrupt service routine, and to further specify that it will use an additional and convenient feature of the PIC24

architecture: a set of four shadow registers. By allowing the processor to automatically save the content of the first four working registers (W0-W3—i.e., the most frequently used ones) and most of the content of the SR register in special reserved locations, without requiring the use of the stack, the shadow registers provide the fastest possible interrupt response time. Naturally, since there is only one set of such registers, their use is limited to applications where only one interrupt will be served at any given time. This does not limit us to use only one interrupt in the entire application, but rather to use _ISRFAST only in applications that have all interrupts with the same priority level or, if multiple levels are in use, reserve the _ISRFAST options only for the interrupt service routines with the highest level of priority.

28.5.3 Notes for PIC Microcontroller Experts

Notice that on the PIC24 architecture there is no single control bit that disables all interrupts, but there is an instruction (DISI) that can disable interrupts for a limited number of cycles. If there are portions of code that require all interrupts to be temporarily disabled, you can use the following inline assembly command:

```
__asm__ volatile("disi #0x3FFF"); // disable temporarily all interrupts

// your code here
// ...

DISICNT = 0; // re-enable all interrupts
```

28.5.4 Tips and Tricks

According to the PIC24 datasheet, to activate the secondary low-power oscillator you need to set the SOSCEN bit in the OSCCON register. But before you rush to type the code in the last example and try to execute it on a real target board, notice that the OSCCON register, containing vital controls for the MCU affecting the choice of the main active oscillator and its speed, is protected by a locking mechanism. As a safety measure, you will have to perform a special unlock sequence first or your command will be ignored. Here is an example, using inline assembly:

```
// OSCCON unlock sequence, setting SOSCEN
    asm volatile ("mov #OSCCON,W1");
    asm volatile ("mov.b #0x46, W2");
    asm volatile ("mov.b #0x57, W3");
    asm volatile ("mov.b #0x02, W0");    // SOSCEN =1
    asm volatile ("mov.b W2, [W1]");
    asm volatile ("mov.b W3, [W1]");
    asm volatile ("mov.b W0, [W1]");
```

A similar combination lock mechanism has been put in place to protect the key RTCC register RCFGCAL. A special bit must be set (RTCWREN) to allow writing to the register, but this bit requires its own special unlock sequence to be executed first. Here is an example using, once more, inline assembly code:

```
// RCFGCAL unlock sequence, setting RTCWREN
asm volatile("disi #5");
asm volatile("mov #0x55, w7");
asm volatile("mov w7,_NVMKEY");
asm volatile("mov #0xAA, w8");
asm volatile("mov w8,_NVMKEY");
asm volatile("bset _RCFGCAL, #13"); // RTCWREN =1;
asm volatile("nop");
asm volatile("nop");
```

After these two steps, which initialize the RTCC, setting the date and time is trivial:

```
_RTCEN = 0; // disable the module

// example set 12/01/2006 WED 12:01:30
_RTCPTR = 3;        // start the loading sequence
RTCVAL = 0x2006;    // YEAR
RTCVAL = 0x1100;    // MONTH-1/DAY-1
RTCVAL = 0x0312;    // WEEKDAY/HOURS
RTCVAL = 0x0130;    // MINUTES/SECONDS

// optional calibration
//_CAL = 0x00;

// enable and lock
_RTCEN = 1;         // enable the module
_RTCWREN = 0;       // lock settings
```

Setting the alarm does not require any special unlock combination. Here is an example that will help you remember my birthday:

```
// disable alarm
_ALRMEN = 0;

// set the ALARM for a specific day of the year (my birthday)
_ALRMPTR = 2;       // start the sequence
ALRMVAL = 0x1124;   // MONTH-1/DAY-1
ALRMVAL = 0x0006;   // WEEKDAY/HOUR
ALRMVAL = 0x0000;   // MINUTES/SECONDS

// set the repeat counter
_ARPT = 0;          // once
_CHIME = 1;         // indefinitely
```

```
// set the alarm mask
_AMASK = 0b1001;    // once a year

_ALRMEN = 1;        // enable alarm
_RTCIF = 0;         // clear interrupt flag
_RTCIE = 1;         // enable interrupt
```

Books

- Curtis, K. E. (2006)

  Embedded Multitasking
  Newnes, Burlington, MA
  Keith knows multitasking and what it takes to create small and efficient embedded-control applications.

Links

- http://www.aopa.org

  This is the web site of the Aircraft Owners and Pilots Association. Feel free to browse through the web site and access the many magazines and free services offered by the association. You will find a lot of useful and interesting information in here.

CHAPTER 29

Taking a Look Under the Hood

29.1 The Plan

In this lesson we will review the basics of string declaration as an excuse to introduce the memory-allocation techniques used by the MPLAB C30 compiler. The RISC architecture of the PIC24 poses some interesting challenges and offers innovative solutions. We will use several tools, including the Disassembly Listing window, the Program Memory window and the Map file to investigate how the MPLAB C30 compiler and linker operate in combination to generate the most compact and efficient code.

29.2 Checklist

This lesson will be performed exclusively with software tools including the MPLAB IDE, MPLAB C30 compiler and the MPLAB SIM simulator.

Use the "New Project Set-up" checklist to create a new project called "Strings" and a new source file similarly called "`strings.c`".

29.3 Coding

Strings are treated in C language as simple ASCII character arrays. Every character composing a string is assumed to be stored sequentially in memory in consecutive 8-bit elements of the array. After the last character of the string an additional byte containing a value of zero (represented in a character notation with '\0') is added as a termination flag.

> Notice though, that this is just a convention that applies to the standard C string manipulation library "`string.h`". It would be entirely possible, for example, to define a new library and store strings in arrays where the first element is used to record the length of the string—in fact, Pascal programmers would be very familiar with this method. Also, if you are developing "international" applications—i.e., applications that communicate using languages that require large character sets (like Chinese, Japanese, Korean)—you might want to consider using Unicode, a technology that allocates multiple bytes per character, in place of plain ASCII. The MPLAB C30 library "`stdlib.h`" provides basic support for the translation from/to multibyte strings according to the ANSI90 standard.

Let's get started by reviewing the declaration of a variable containing a single character:

```
char c;
```

As we have seen from the previous lessons, this is how we declare an 8-bit integer (character), that is treated as a signed value (−128 . . . +127) by default.

We can declare and initialize it with a numerical value:

```
char c = 0x41;
```

Or, we can declare and initialize it with an ASCII value:

```
char c = 'a';
```

Note the use of the single quotes for ASCII character constants. The result is the same, and to the C compiler there is absolutely no distinction between the two declarations—characters ARE numbers.

We can now declare and initialize a string as an array of 8-bit integers (characters):

```
char s[5] = { 'H', 'E', 'L', 'L', 'O'};
```

In this example, we initialized the array using the standard notation for numerical arrays. But, we could have also used a far more convenient notation (a shortcut) specifically created for string initializations:

```
char s[5] = "HELLO";
```

To further simplify things, and save you from having to count the number of characters composing the string (thus preventing human errors), you can use the following notation:

```
char s[] = "HELLO";
```

The MPLAB C30 compiler will automatically determine the number of characters required to store the string, while automatically adding a termination character (zero) that will be useful to the string manipulation routines later to correctly identify the string length. So, the example above is, in truth, equivalent to the following declaration:

```
char s[6] = { 'H', 'E', 'L', 'L', 'O', '\0' };
```

Assigning a value to a char (8-bit integer) variable and performing arithmetic on it is no different than performing the same operation on any integer type:

```
char c;    // declare c as an 8-bit signed integer

c = 'a';   // assign to it the value corresponding to 'a' in the
           // ASCII table

c ++;      // increment it... it will represent the ASCII character
           // 'b' now
```

The same operations can be performed on any element of an array of characters (string), but there is no simple shortcut, similar to the one used above for the initialization that can assign a new value to an entire string:

```
char s[15];      // declare s as a string of 15 characters
s = "Hello!";    // Error! This does not work!
```

Including the "`string.h`" file at the top of your source file, you'll gain access to numerous useful functions that will allow you to:

- copy the content of a string onto another:

  ```
  strcpy( s, "HELLO");          // s : "HELLO"
  ```

- append (or concatenate) two strings:

  ```
  strcat( s, " WORLD");// s : "HELLO WORLD"
  ```

- determine the length of a string:

  ```
  i = strlen( s);               // i : 11
  ```

- and many more.

29.3.1 Memory Space Allocation

Just as with numerical initializations, every time a string variable is declared and initialized as in:

```
char s[] = "Flying with the PIC24";
```

three things happen:

1. the MPLAB C30 linker reserves a contiguous set of memory locations in RAM (data space) to contain the variable: 22 bytes in the example above. This space is part of the `ndata` (near) data section.
2. the MPLAB C30 linker stores the initialization value in a 22-byte long table (in program memory). This space is part of the `init` code section.
3. the MPLAB C30 compiler creates a small routine that will be called before the `main` program (part of the `C0` code we mentioned in previous chapters) to copy the values from code to data memory, therefore initializing the variable.

In other words, the string "Flying with the PIC24" ends up using twice the space you would expect, as a copy of it is stored in Flash program memory and space is reserved for it in RAM memory, too. Additionally, you must consider the initialization code and the time spent in the actual copying process. If the string is not supposed to be manipulated during the program,

but is only used "as is," transmitted to a serial port or sent to a display, then there is no need to waste precious resources. Declaring the string as a "constant" will save RAM space and initialization code/time:

```
const char s[] = "Flying with the PIC24";
```

Now, the MPLAB C30 linker will only allocate space in program memory, in the `const` code section, where the string will be accessible via the Program Space Visibility window—an advanced feature of the PIC24 architecture that we will review shortly.

The string will be treated by the compiler as a direct pointer into program memory and, as a consequence, there will be no need to waste RAM space.

In the previous examples of this lesson, we saw other strings implicitly defined as constants:

```
strcpy( s, "HELLO");
```

The string "HELLO" was implicitly defined as of `const char` type, and similarly assigned to the `const` section in program memory to be accessible via the Program Space Visibility window.

Note that, if the same constant string is used multiple times throughout the program, the MPLAB C30 compiler will automatically store only one copy in the `const` section to optimize memory use, even if all optimization features of the compiler have been turned off.

29.3.2 Program Space Visibility

The PIC24 architecture is somewhat different from most other 16-bit microcontroller architectures you might be familiar with. It was designed for maximum efficiency according to the Harvard model, as opposed to the more common Von Neumann model. The big difference between the two is that there are two completely separated and independent buses available, one for access to the Program Memory (Flash) and one for access to the Data Memory (RAM). The net result is a doubled bandwidth; while the data bus is in use during the execution of one instruction, the program memory bus is available to fetch the next instruction code and initiate the decoding. In traditional Von Neumann architectures, the two activities must be interleaved instead, with a consequent performance penalty. The drawback of this architectural choice is that access to constants and data stored in program memory requires special considerations.

The PIC24 architecture offers two methods to read data from program memory: using special table access instructions (`tblrd`) and through a second mechanism, called the Program Space Visibility or PSV. This is a window of up to 32K bytes of program memory accessible from the data memory bus (Fig. 29.1). In other words the PSV is a bridge between the program memory bus and the data memory bus.

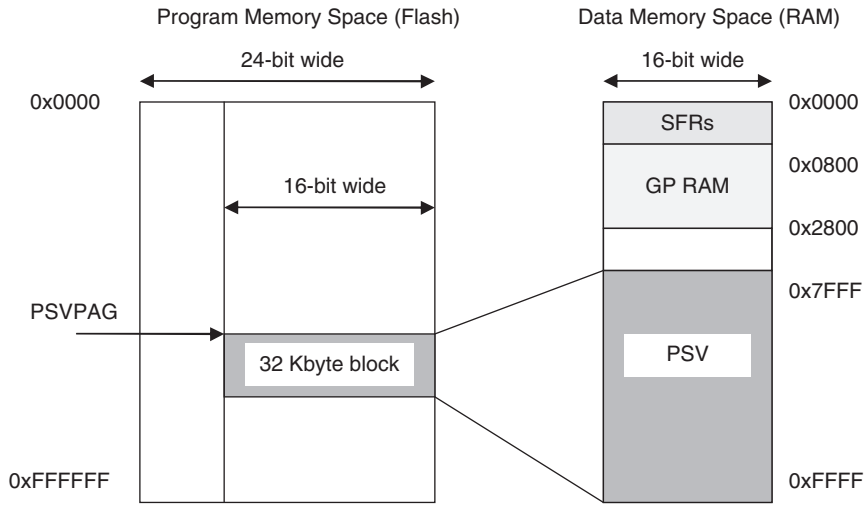


Figure 29.1: PIC24FJ128GA010 Program Space Visibility Window

Notice that the PIC24 uses a 24-bit wide program memory bus but operates only on a 16-bit wide data bus. The mismatch between the two buses makes the PSV "bridge" a little more interesting. In practice the PSV connects only the lower 16 bits of the program memory bus to the data memory bus. The upper portion (8 bits) of each program memory word is not accessible with the PSV window. On the contrary, when using the table-access instructions, all parts of the program memory word become accessible, but at the cost of having to differentiate the manipulation of data in RAM (using direct addressing) from the manipulation of data in program memory (using the special table-access instructions).

The PIC24 programmer can therefore choose between a more convenient but relatively memory-inefficient method for transferring data between the two buses such as the PSV, or a more memory-efficient but less-transparent solution offered by the table-access instructions.

The designers of the MPLAB C30 compiler considered the trade-offs and chose to use both mechanisms, although to solve different problems at different times:

- the PSV is used to manage constant arrays (numeric and strings) so that a single type of pointer (to the data memory bus) can be used uniformly for constants and variables.

- the table-access mechanism is used to perform the variable initializations (limited to the `C0` segment) for maximum compactness and efficiency.

29.3.3 Investigating Memory Allocation

We will start investigating these issues with the MPLAB SIM simulator and the following short snippet of code:

```
/*
** Strings
*/

#include <p24fj128ga010.h>
#include <string.h>

// 1. variable declarations
const char a[] = "Learn to fly with the PIC24";
char b[100] = "";

// 2. main program
main()
{
      strcpy( b, "MPLAB C30");        // assign new content to b
} //main
```

Now, follow these steps:

1. Build the project using the Project Build checklist.
2. Add the Watch window (and dock it to the preferred position).
3. Select the two variables "a" and "b" from the symbol selection box and click "Add Symbol" to add them to the Watch window (Fig. 29.2).

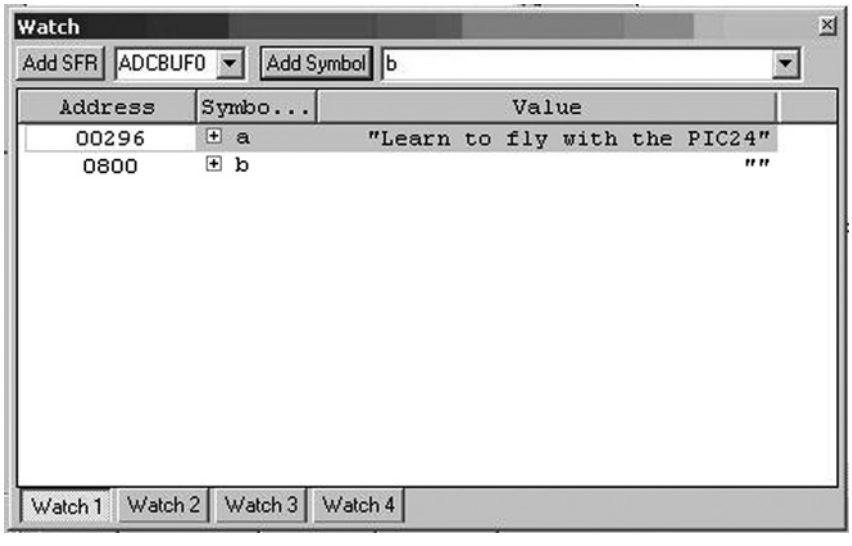


Figure 29.2: Adding Arrays to the Watch Window

A little "+" symbol enclosed in a box will identify these variables as arrays and will allow you to expand the view to identify each individual element (Fig. 29.3).

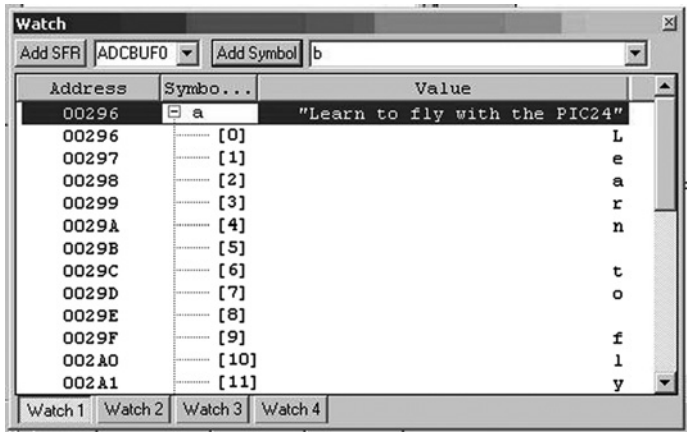


Figure 29.3: Expanding an Array in the Watch Window

By default MPLAB shows each element of the array as an ASCII character, but you can change the display to reflect your personal preferences:

4. Select one element of the array with the left button of your mouse.
5. Right click to show the Watch window menu.
6. Select "Properties" (the last item in the menu).

You will be presented with the Watch window Properties dialog box (Fig. 29.4).

From this dialog box you can change the format used to display the content of the selected array element, but you can also observe the "Memory" field (grayed) that tells you where the selected variable is allocated: data or code space.

If you select the Properties dialog box for the constant string "`a`", you will notice that the memory space is indicated as "Program", confirming that the constant string is using only the minimum amount of space required in the Flash program memory of the PIC24 and will be accessed through the PSV so that no RAM needs to be assigned to it.

On the contrary, the Properties dialog box will reveal how the string "`b`" is allocated in a File register, or in other words RAM memory.

Continuing our investigation, notice how the string "`a`" appears to be already initialized, as the Watch window shows it's ready to use, right after the project build.

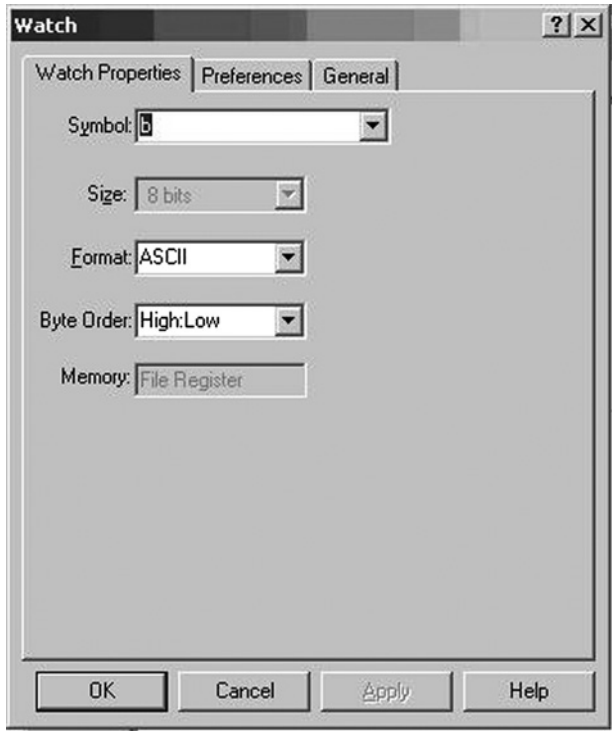


Figure 29.4: Watch Window Properties Dialog Box

The string "b", on the contrary, appears to be still empty, and uninitialized. Only when we set the cursor on the first line of code inside the `main` routine and we execute a Run To Cursor command, the string "b" is initialized with the proper value (Fig. 29.5).

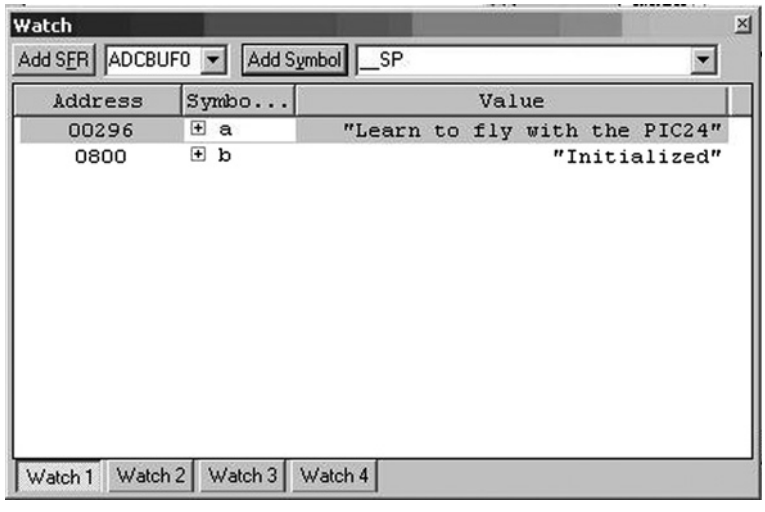


Figure 29.5: Array "b" Initialized

As we have seen, "b" is allocated in RAM space, and the `C0` segment of code must be executed first for the variable to be initialized and "ready for use."

> Just a warning—the Watch window aligns all strings to the right, so if there is a long string (like "a" in our example) and the window is too narrow, you might not be able to see the content of other variables containing shorter strings. Undock the Watch window if necessary, and resize it to be able to see the entire Value column.

Once more we can make use of the Disassembly Listing window to observe the code produced by the compiler:

```
---C:\work\C30\j\6 Strings\Strings.c --------------------------------
1:                /*
2:                ** Strings
3:                */
4:
5:                #include <p24fj128ga010.h>
6:                #include <string.h>
7:
8:                // 1. variable declarations
9:
10:               const char a[] = "Learn to fly with the PIC24";
11:               char b[100] = "Initialized";
12:
13:               // 2. main program
14:               main()
15:               {
  0028A   FA0000  lnk #0x0
16:                 strcpy( b, "MPLAB C30"); // assign new content to b
  0028C   282B21    mov.w #0x82b2,0x0002
  0028E   208000    mov.w #0x800,0x0000

00290     07FFF7  rcall 0x000280
17:
18:               } // main
 00292    FA8000  ulnk
 00294    060000  return

---c:\pic30-build\build_20060131\src\standardc\sxl\strcpy.c -------

 00280    780100  mov.w 0x0000,0x0004
 00282    784931  mov.b [0x0002++],[0x0004]
 00284    E00432  cp0.b [0x0004++]
 00286    3AFFFD  bra nz, 0x000282
 00288    060000  return
```

We can see the `main()` function and the `strcpy()` library function full disassembly appended at the bottom of the listing.

Notice how compact the code is produced for the `strcpy()` routine, barely five instructions. You will also appreciate how this is the only routine attached. Although the "`string.h`" library contains dozens of functions, and the include file "`string.h`" contains the declarations for all of them, the linker is wisely appending only the functions that are actually being used.

What the Disassembly Listing window does not show, though, is the initialization code `C0`. As mentioned in a previous chapter, in order to see it, you will have to rely on the Program Memory window (I recommend you select the Symbolic view tab at the bottom). There the most curious and patient readers will discover how the initialization of the string "`b`" is performed using the Table Read (`tblrd`) instructions to extract the data from the program memory (Flash) and to store the values in the allocated space in data memory (RAM).

29.3.4 Looking at the MAP

Another tool we have at our disposal to help us understand how strings (and in general any array variable) are initialized and allocated in memory is the "map file". This text file, produced by the MPLAB C30 linker, can be easily inspected with the MPLAB editor and is designed specifically to help you understand and resolve memory allocation issues.

To find this file, look for it in the main project directory where all the project source files are (Fig. 29.6). Select "File→Open" and then browse until you reach the project directory. By default the MPLAB editor will list all the "`.c`" files, but you can change the File Type field to "`.map`".

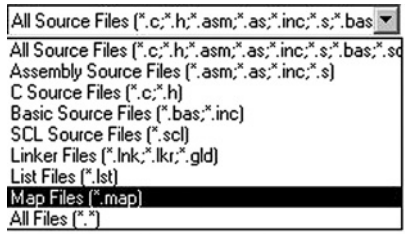


Figure 29.6: Selecting the ".map" File Type

Map files tend to be pretty long and verbose but, by learning to inspect only a few critical sections, you will be able to find a lot of useful data. The Program Memory Usage summary, for example, is found among the very first few lines:

```
Program Memory Usage

section    address    length (PC units)    length (bytes)    (dec)
-------    -------    -----------------    ----------------------

.reset           0                  0x4               0x6    (6)

.ivt           0x4                 0xfc             0x17a    (378)

.aivt        0x104                 0xfc             0x17a    (378)

.text        0x200                 0x96              0xe1    (225)

.const       0x296                 0x26              0x39    (57)

.dinit       0x2bc                 0x4c              0x72    (114)

.isr         0x308                  0x2               0x3    (3)

          Total program memory used (bytes): 0x489 (1161) <1%
```

This is a list of small sections of code assembled by the MPLAB C30 linker in a specific order (dictated by the `.gld` linker script file) and position.

Most section names are pretty intuitive, other are. . .historical:

- the `.reset` section is where the reset vector is placed.
- the `.ivt` is the Interrupt Vector Table, discussed in the previous chapter.
- the `.aivt` is the Alternate Interrupt Vector Table.
- the `.text` section is where all the code generated by the MPLAB C30 compiler from your source files will be placed (the name of this section has been used since the original implementation of the very first C compiler).
- the `.const` section is where the constants (integers and strings) will be placed for access via the PSV.
- the `.dinit` section is where the variables initialization data (used by the `C0` code) will be found.
- the `.isr` is where the Interrupt Service Routine (in this case a default one) will be found.

It's in the `.const` section that the "`a`" constant string, as well as the "MPLAB C30" (implicit) constant string, are stored for access via the PSV window. You can confirm this by inspecting the Program Memory window at the address 0x296. Note the two-by-two character grouping; remember how the PSV allows us to use only 16 bits of each 24-bit program memory word.

```
00290    ----   07FFF7   FA8000   060000   00654C   ........   ....Le..
00298    ----   007261   00206E   006F74   006620   ar..n ..   to.. f..
002A0    ----   00796C   007720   007469   002068   ly.. w..   it..h ..
002A8    ----   006874   002065   004950   003243   th..e ..   PI..C2..
002B0    ----   000034   00504D   00414C   002042   4...MP..   LA..B ..
002B8    ----   003343   000030   000800   000064   C3..0...   ....d...
```

In .dinit is where the "b" variable initialization string, will be found. It is prepared for access via the table instructions, so it uses each and every one of the 24 bits available in each program memory word. Note the character grouping in three by three:

```
002C0    ----   000002   696E49   616974   7A696C   ....Ini.   tia.liz.
002C8    ----   006465   000000   000000   000000   ed......   ........
002D0    ----   000000   000000   000000   000000   ........   ........
```

The next part of the map file we might want to inspect is the Data Memory Usage (RAM) summary:

```
Data Memory Usage

section    address    alignment gaps    total length (dec)
-------    -------    --------------    ------------------
.ndata     0x800                  0            0x64 (100)

        Total data memory used (bytes):          0x64 (100) 1%
```

In our simple example, it contains only one section: .ndata, and in it, only one variable "b" for which 100 bytes are reserved starting at the address 0x800, the first location available in the PIC24 RAM.

29.3.5 Pointers

Pointers are variables used to refer indirectly (point to) other variables or part of their contents. Pointers and strings go hand in hand in C programming, as they are in general a powerful mechanism to work on any array data type. So powerful, in fact, that they are also one of the most dangerous tools in the programmer's hands and a source of a disproportionately large share of programming bugs. Some programming languages, like Java, have gone to the extreme of banning completely the use of pointers in an effort to make the language more robust and verifiable.

The MPLAB C30 compiler takes advantage of the PIC24 16-bit architecture to manage with ease large amounts of data memory (up to 32 kbytes of RAM available in current models). In particular, thanks to the PSV window, the MPLAB C30 compiler doesn't make any distinction between pointers to data memory objects and const objects allocated in program memory

space. This allows a single set of standard functions to manipulate variables and/or generic memory blocks as needed from both spaces.

The following classic program example will compare the use of pointers versus indexing to perform sequential access to an array of integers:

```
int *pi;            // define a pointer to an integer
int i;              // index/counter
int a[10];          // the array of integers

// 1. sequential access using array indexing
for( i=0; i<10; i++)
   a[ i] = i;

// 2. sequential access using a pointer
pi = a;
for( i=0; i<10; i++)
{
      *pi = i;
      pi++;
}
```

In 1. we performed a simple `for` loop and at each round in the loop we used "`i`" as an index in the array. To perform the assignment, the compiler will have to take the value of "`i`", multiply it by the size of the array element in bytes (2), and add the resulting offset to the initial address of the array "`a`".

In 2. we initialized a pointer to point to the initial address of the array "`a`". At each round in the loop we simply used the pointer (`*`) to perform the assignment, and then we just incremented the pointer.

Comparing the two cases, we see how, by using the pointer, we can save at least one multiplication step for each round in the loop. If inside the loop the array element is used more times, the performance improvement is going to be proportionally greater.

Pointers syntax can become very "concise" in C, allowing for some effective code to be written, but also opening the door to more bugs.

At a minimum, you should become familiar with the most common contractions. The previous snippet of code is more often reduced to the following:

```
// 2. sequential access to array using pointers
for( i=0, p=a; i<10; i++)
   *pi++ = i;
```

Also note that an empty pointer—that is, a pointer without a target—is assigned a special value `NULL`, which is implementation specific and defined in "`stddef.h`".

29.3.6 The Heap

One of the advantages offered by the use of pointers is the ability to manipulate objects that are defined dynamically (at run time) in memory. The "heap" is the area of data memory reserved for such use, and a set of functions, part of the standard C library "`stdlib.h`", provides the tools to allocate and free the memory blocks. They include at a minimum the fundamental functions:

```
void *malloc(size_t size);
```

takes a block of memory of requested size from the heap, and returns a pointer to it.

```
void free(void *ptr);
```

returns the block of memory pointed to by `ptr` to the heap.

The MPLAB C30 linker places the heap in the RAM memory space left unused above all project global variables and the reserved stack space. Although the amount of memory left unused is known to the linker and listed in the map file of each project, you will have to explicitly instruct the linker to reserve an exact amount for use by the heap.

Use the "Project→BuildOptions→Project" menu command to open the Build Options dialog box, select the MPLAB Link30 tab, and define the heap size in bytes.

As a general rule, allocate the largest amount of memory possible, as this will allow the `malloc()` function to make the most efficient use of the memory available. After all, if it is not assigned to the heap it will remain unused.

29.3.7 MPLAB C30 Memory Models

The PIC24 architecture allows for a very efficient (compact) instruction encoding for all operations performed on data memory within the first 8 kbytes of addressing space. This is referred to as the "near" memory area and in the case of the PIC24FJ128gA010 it corresponds to the group of SFRs (first 2 kbytes) and the following 6 kbytes of general-purpose RAM. Only the top 2 kbytes of RAM are actually outside the near space.

Access to memory beyond the 8-kbyte limit requires the use of indirect addressing methods (pointers) and could be less efficient if not properly planned. The stack (and with it all the local variables used by C functions) and the heap (used for dynamic memory allocation) are naturally accessed via pointers and are correspondingly ideal candidates to be placed in the upper RAM space. This is exactly what the linker will attempt to do by default. It will also try to place all the global variables defined in a project in the near memory space for maximum efficiency. If a variable cannot be placed within the near memory space, it has to be "manually"

declared with a "`far`" attribute, so that the compiler will generate the appropriate access code. This behavior is referred to as the Small Data Memory Model as opposed to the Large Memory model, where each variable is assumed to be far unless the "`near`" attribute is specified.

In practice, while using the PIC24FJ128gA010, you will use almost uniquely the default Small Memory model and in rare occasions you will find it necessary to identify a variable with the "far" attribute. In lesson number 12, we will observe one such case, where a very large array that would otherwise not fit in the near memory space will have to be declared as "`far`". As a consequence, not only will the compiler generate the correct addressing instructions, but the linker will also push it to an upper area of RAM, giving priority to the other global variables and allowing them to be accessed in the near space.

Since access to elements of an array (explicitly via pointers or by indexing) is performed via indirect addressing anyway, there will be no performance or code size penalty.

A similar concept applies to the program memory space. In fact, within each compiled module, functions are called by making use of a more compact addressing scheme that relies on a maximum range of 32 kbytes. Program memory models (small and large) define the default behavior of the compiler/ linker with regards to the addressing of functions within or outside such 32-kbyte range.

29.4 Review

In the C language, strings are defined as simple arrays of characters, but the C language standard had no concept of different memory regions (RAM vs. Flash) nor of the particular mechanisms required to cross the bridge between different buses in a Harvard architecture. The programmer using the MPLAB C30 compiler needs a basic understanding of the trade-offs of the various mechanisms available and the allocation strategies adopted to make the most of the precious resources (RAM especially) available to embedded-control applications.

29.4.1 Notes for C Experts

The `const` attribute is normally used in the C language, together with most other variable types, only to assist the compiler in catching common parameter usage errors. When a parameter is passed to a function as a `const` or a variable is declared as a `const`, the compiler can in fact help flag any following attempt to modify it. The MPLAB C30 use of the PSV extends this semantic in a very natural way, allowing for a more efficient implementation, as we have seen.

29.4.2 Notes for Assembly Experts

The "`string.h`" library contains many useful block manipulation functions that can be useful, via the use of pointers, to perform operations on any type of data array, not just strings, like `memcpy()`, `memcmp()`, `memset()` and `memmove()`.

The "ctype.h" library contains instead functions that help discriminate individual characters according to their position in the ASCII table, to discriminate lower case from upper case, and/or convert between the two.

29.4.3 Notes for PIC Microcontroller Experts

Since the PIC24 program memory is implemented using Flash technology, programmable with a single supply voltage even at run time, during code execution, it is possible to design boot-loaders—that is, applications that automatically update part or all of their own code. It is also possible to utilize a section of the Flash program memory as a nonvolatile memory storage area, within some pretty basic limitations. To write to the Flash program memory, though, you will need to utilize the table-access methods and exercise extreme caution. The PSV window is a read-only device and, as we have seen before, it gives access only to 16 of the 24 bits of each program memory location.

Also, notice that the memory can only be written in complete rows of 64 words each and must be first erased in blocks of 8 rows (512 words) each. This can make frequent updates impractical if single words or small data structures in general are being managed.

29.4.4 Tips and Tricks

String manipulation can be fun in C once you realize how to make the zero termination character work for you efficiently. Take, for example, the mycpy() function below:

```
void mycpy( char *dest, char * src)
{
      while( *dest++ = *src++);
}
```

This is quite a dangerous piece of code, as there is no limit to how many characters could be copied, there is no check whether the dest pointer is pointing to a buffer that is large enough, and you can imagine what would happen should the src string be missing the termination character. It would be very easy for this code to continue beyond the allocated variable spaces and to corrupt the entire contents of the data RAM, including the all precious SFRs.

At a minimum, you should try to verify that pointers passed to your functions have been initialized before use. Compare them with the NULL value (declared in "stdlib.h" and/or "stddef.h") to catch the error.

Add a limit to the number of bytes to be copied; it is reasonable to assume that you will know the size of the strings/arrays used by your program, and if you don't, use the sizeof() operator. A better implementation of mycpy() could be the following:

```
void mycpy( char *dest, char *src, int max)
{
```

```
      if ((dest != NULL) && ( src != NULL))
            while (( max-- > 0) && ( *src))
                  *dest++ = *src++;
}
```

Books

- Wirth, N. (1976)

  Algorithms + Data Structures = Programs

  Prentice-Hall, Englewood Cliffs, NJ

  With unparalleled simplicity, Wirth (the father of the Pascal programming language) takes you from the basics of programming all the way up to writing your own compiler.

Links

- http://en.wikipedia.org/wiki/Pointers#Support_in_various_programming_languages

  Learn more about pointers and see how they are managed in various programming languages.

This page intentionally left blank

SECTION VI

Appendices

This page intentionally left blank

APPENDIX A

The PIC® 16 Series Instruction Set

Table A1.1: PIC 16 Series Instruction Set Summary

Mnemonic, Operands		Description	Cycles	14-Bit Opcode				Status Affected	Notes
				MSb			LSb		
BYTE-ORIENTED FILE REGISTER OPERATIONS									
ADDWF	f, d	Add W and f	1	00	0111	dfff	ffff	C,DC,Z	1,2
ANDWF	f, d	AND W with f	1	00	0101	dfff	ffff	Z	1,2
CLRF	f	Clear f	1	00	0001	lfff	ffff	Z	2
CLRW	-	Clear W	1	00	0001	0xxx	xxxx	Z	
COMF	f, d	Complement f	1	00	1001	dfff	ffff	Z	1,2
DECF	f, d	Decrement f	1	00	0011	dfff	ffff	Z	1,2
DECFSZ	f, d	Decrement f, Skip if 0	1 (2)	00	1011	dfff	ffff		1,2,3
INCF	f, d	Increment f	1	00	1010	dfff	ffff	Z	1,2
INCFSZ	f, d	Increment f, Skip if 0	1 (2)	00	1111	dfff	ffff		1,2,3
IORWF	f, d	Inclusive OR W with f	1	00	0100	dfff	ffff	Z	1,2
MOVF	f, d	Move f	1	00	1000	dfff	ffff	Z	1,2
MOVWF	f	Move W to f	1	00	0000	lfff	ffff		
NOP	-	No Operation	1	00	0000	0xx0	0000		
RLF	f, d	Rotate Left f through Carry	1	00	1101	dfff	ffff	C	1,2
RRF	f, d	Rotate Right f through Carry	1	00	1100	dfff	ffff	C	1,2
SUBWF	f, d	Subtract W from f	1	00	0010	dfff	ffff	C,DC,Z	1,2
SWAPF	f, d	Swap nibbles in f	1	00	1110	dfff	ffff		1,2
XORWF	f, d	Exclusive OR W with f	1	00	0110	dfff	ffff	Z	1,2
BIT-ORIENTED FILE REGISTER OPERATIONS									
BCF	f, b	Bit Clear f	1	01	00bb	bfff	ffff		1,2
BSF	f, b	Bit Set f	1	01	01bb	bfff	ffff		1,2
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)	01	10bb	bfff	ffff		3
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)	01	11bb	bfff	ffff		3
LITERAL AND CONTROL OPERATIONS									
ADDLW	k	Add literal and W	1	11	111x	kkkk	kkkk	C,DC,Z	
ANDLW	k	AND literal with W	1	11	1001	kkkk	kkkk	Z	
CALL	k	Call subroutine	2	10	0kkk	kkkk	kkkk		
CLRWDT	-	Clear Watchdog Timer	1	00	0000	0110	0100	$\overline{TO}$,$\overline{PD}$	
GOTO	k	Go to address	2	10	1kkk	kkkk	kkkk		
IORLW	k	Inclusive OR literal with W	1	11	1000	kkkk	kkkk	Z	
MOVLW	k	Move literal to W	1	11	00xx	kkkk	kkkk		
RETFIE	-	Return from interrupt	2	00	0000	0000	1001		
RETLW	k	Return with literal in W	2	11	01xx	kkkk	kkkk		
RETURN	-	Return from Subroutine	2	00	0000	0000	1000		
SLEEP	-	Go into standby mode	1	00	0000	0110	0011	$\overline{TO}$,$\overline{PD}$	
SUBLW	k	Subtract W from literal	1	11	110x	kkkk	kkkk	C,DC,Z	
XORLW	k	Exclusive OR literal with W	1	11	1010	kkkk	kkkk	Z	

Note 1: When an I/O register is modified as a function of itself (e.g., `MOVF PORTB, 1`), the value used will be that value present on the pins themselves. For example, if the data latch is '1' for a pin configured as input and is driven low by an external device, the data will be written back with a '0'.

2: If this instruction is executed on the TMR0 register (and, where applicable, d = 1), the prescaler will be cleared if assigned to the Timer0 Module.

3: If Program Counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a `NOP`.

This page intentionally left blank

APPENDIX B

The Electronic Ping-Pong

This small embedded system is a game for two players, each of whom has a push-button "paddle. " Either player can start the game by pressing his/her paddle. The ball, represented by the row of eight LEDs, then flies through the air to the opposing player, who must press his/her paddle only when the ball is at the end LED and at none other. The ball continues in play until either player violates this rule of play. Once this happens, the nonviolating player scores and the associated LED is briefly lit up. When the ball is out of play, an "out-of-play" LED is lit.

A circuit diagram appears in Fig. B.1, with its program listing forming the remainder of this appendix.

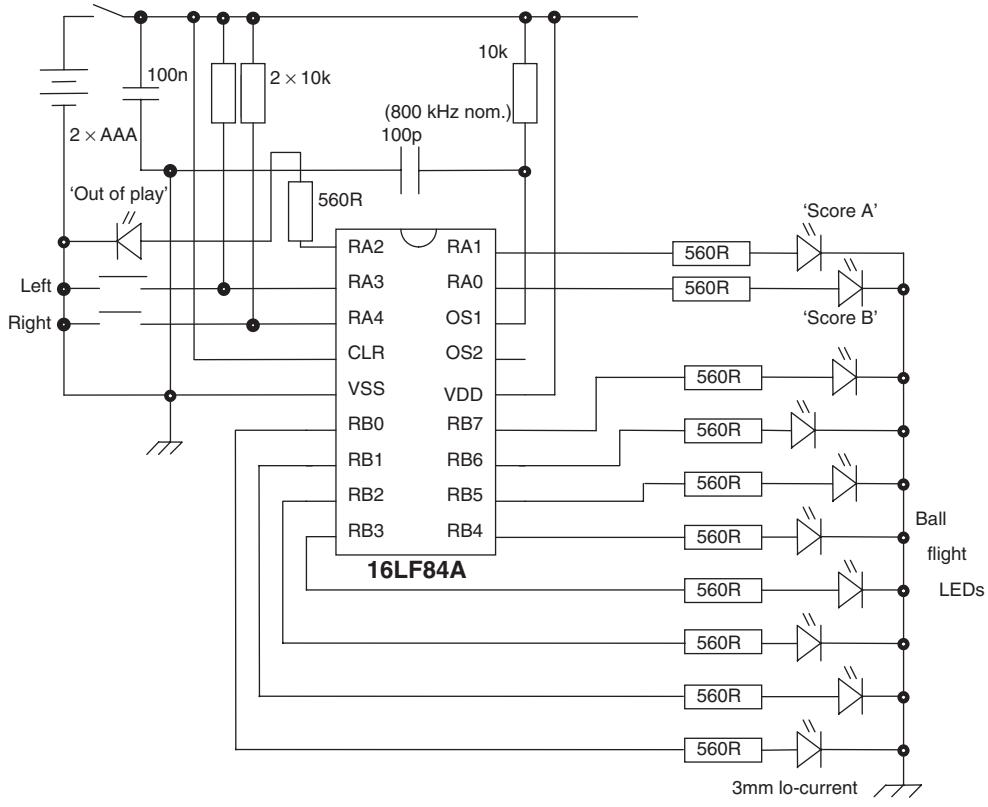


Figure B.1: The Electronic Ping-Pong Circuit Diagram

```
;*************************************************************
;ELECTRONIC PING-PONG!
;This program drives the electronic ping-pong game,
;fixed speed, single mode of play.
;TJW 21.6.01
;*************************************************************
;
;Clock freq 800kHz approx (RC osc.)
;Port A 4       right paddle (ip)
;          3      left paddle (ip)
;          2      "out of play" led (op)
;          1      "Score Left" led (op)
;          0      "Score Right" led (op)
;Port B 7-0    "play" leds (all op)
;
;Config Word:       RC oscillator, WDT off,
;          power-up timer on, code protect off
;No Interrupts used
;
        list p=16F84A
;specify SFRs
status equ        03
porta  equ        05
trisa  equ        05
portb  equ        06
trisb  equ        06
;
;specify a constant
led_durn equ      20       ;no. of time inner loop is iterated, hence
                              ;time duration each led is lit.
;
;specify RAM locations
delcntr1 equ  10   ;used in 5ms delay SR
delcntr2 equ  11   ;used in 500ms delay SR
led_posn equ  12   ;holds current ball led posn.
loop_cntr equ 13   ;preloaded with value led_durn for every
                        ;led illumination, and counts down to 0 before
                        ;ball moves on
;
       org 00
       goto start
;
; **"Initialise" State**
; Initialise
       org 0010
```

Program Example B.1: The Electronic Ping-Pong Program

```
start   bsf     status,5    ;select memory bank 1
        movlw   B'00011000'
        movwf   trisa       ;port A according to above pattern
        movlw   00
        movwf   trisb       ;all port B bits op
        bcf     status,5    ;select bank 0
;
;**"Wait"** State
;set up initial led patterns
wait    movlw   04
        movwf   porta       ;switch on "out of play" led
        movlw   00
        movwf   portb       ;all play leds off
;
;check that both paddles are clear before allowing play to commence
        btfss   porta,4     ;right paddle pressed?
        goto    wait        ;yes, so wait
        btfss   porta,3     ;left paddle pressed?
        goto    wait        ;yes, so wait
;
;now ready for action, now wait until paddle pressed
wait1   btfss   porta,4     ;right paddle pressed?
        goto    r_to_l      ;yes, so start play
        btfss   porta,3     ;left paddle pressed?
        goto    l_to_r      ;yes, so start play
        goto    wait1
;
;**"Right-to-Left" State**
;play has started
r_to_l  movlw   00          ;switch off "out of play"
        movwf   porta
        movlw   80          ;define ball start posn.
        movwf   led_posn
;loop to here every time led is to change
rtl_0   movlw   led_durn
        movwf   loop_cntr   ;preset length of led illumination
        movf    led_posn,w  ;output new ball posn
        movwf   portb
;loop to here n times for every led, where n = led_durn.
;Check for rule violations. Special conditions apply if
;ball is at start or end.
rtl_1 btfss led_posn,7 ;is ball at start (ie posn 7)?
        goto    rtl_2  ;no, so move on
        ;yes, it's OK if right paddle still pressed, so don't test
        btfss   porta,3     ;left paddle pressed?
```

Program Example B.1: Continued

```
        goto    rt_myscore  ;yes, so score
        goto    rtlend
rtl_2   btfss   led_posn,0  ;is ball at end (ie posn 0)?
        goto    rtl_3       ;no, so move on
;here if ball at end, left paddle can force direction change
        btfss   porta,3     ;left paddle pressed?
        goto    l_to_r      ;yes, so change direction - **state exit**
        btfss   porta,4     ;right paddle pressed?
        goto    rt_yrscore  ;yes, so left scores
        goto    rtlend
;here if neither start nor end posn.
rtl_3   btfss   porta,4     ;right paddle pressed?
        goto    score_left  ; yes, so score
        btfss   porta,3     ;left paddle pressed?
        goto    rt_myscore  ;yes, so score
;at then end of each loop call a delay
rtlend  call    delay5
        decfsz loop_cntr    ;decrement loop counter, check if led is
                            to move
        goto    rtl1
;here if ball moving on
        bcf     status,0
        rrf     led_posn,1
        btfsc   status,0    ;ball off end?
        goto    rt_myscore  ;yes, right's point
        goto    rtl0
;**state exit**
rt_myscore goto score_right
rt_yrscore goto score_left
;
;**"Left-to-Right" State**
l_to_r  movlw   00          ;switch off "out of play"
        movwf   porta
        movlw   01          ;define ball start posn.
        movwf   led_posn
ltr_0   movlw   led_durn
        movwf   loop_cntr ;determine length of led illumination
;go round this loop 'duration' times, for every ball position
ltr_1   movf    led_posn,w ;output new ball posn
        movwf   portb
        btfss   led_posn,0 ;is ball at start (ie posn 0)?
        goto    ltr_2       ;no, so move on
        ;yes,   OK if left paddle still pressed (so only test rt
                paddle)
```

Program Example B.1: Continued

```
        btfss   porta,4     ;right paddle pressed?
        goto    lft_myscore ;yes, so score
        goto    ltrend
ltr_2   btfss   led_posn,7  ;is ball at end (ie posn 7)?
        goto    ltr_3       ;no, so move on
;here if ball at end, right paddle will change dirn, score right if
;left paddle
        btfss   porta,4     ;right paddle pressed?
        goto    r_to_l      ;yes, so change direction
        btfss   porta,3     ;left paddle pressed?
        goto    lft_yrscore ;yes, so right score
        goto    ltrend
;here if neither start nor end posn.
ltr_3   btfss   porta,4     ;right paddle pressed?
        goto    lft_myscore  ;yes, so score
        btfss   porta,3     ;left paddle pressed?
        goto    lft_yrscore ;yes, so score
ltrend  call    delay5
        decfsz  loop_cntr   ;decrement loop counter, check if led is
                            ;to move
        goto    ltr_1
;here if ball moving on
        bcf     status,0    ;Clear Carry, as rlf rotates through it
        rlf     led_posn,1
        btfsc   status,0    ;ball off end?
        goto    lft_myscore ;yes, left's point
        goto    ltr_0
;**state exit**
lft_myscore goto score_left
lft_yrscore goto score_right
;
;**"Score" State**
;here if Left has scored
score_left
        movlw   00
        movwf   portb ;all play leds off
        bsf     porta,1
        call    delay500
        bcf     porta,1
        goto    wait
;here if Right has scored
score_right
        movlw   00
        movwf   portb ;all play leds off
        bsf     porta,0
```

Program Example B.1: Continued

```
        call    delay500
        bcf     porta,0
        goto    wait
;
;*****************************************
;SUBROUTINES
;*****************************************
;Delay of 5ms approx. Instruction cycle time is 5us.
delay5  movlw D'200' ;200 cycles called,
                                ;each taking 5x5=25us
        movwf   delcntr1
del1    nop                ;5 inst cycles in this loop
        nop
        decfsz delcntr1,1
        goto del1
        return
;
; Delay of 500ms (approx) - 100 calls to delay5
delay500 movlw D'100'
        movwf   delcntr2
del2    call    delay5
        decfsz  delcntr2,1
        goto    del2
        return
;
        end
```

Program Example B.1: Continued

APPENDIX C

DIZI-2 Board and Lock Application

DIZI-2 Demonstration Board

A circuit was required to demonstrate a range of basic microcontroller programming techniques via a set of simple applications for the PIC 16F84. The DIZI (DIsplay, buZzer and Interrupt) board was designed to allow the special hardware features of the PIC chip to be exercised, including interrupts, timer and EEPROM memory. In-circuit programming was not incorporated, in order to emphasize the stand-alone operation of the microcontroller. The chip would be programmed separately and then physically transferred to the target system. The enhanced DIZI-2 board described in this appendix incorporates an on-board battery supply, a finger pot to provide an analog input and hardware switch debouncing to improve the reliability of the push button operation.

The circuit is built on a 100 × 100 mm piece of stripboard which has copper tracks for making the component connections on a standard 0.1″ grid. The design includes a 2 × 1.5 V battery pack on the board. The power is switched on via a nonlatching push button so that it cannot be left on accidentally, and thereby exhaust the batteries; it must be held on manually while the circuit is in operation.

The basic demonstration programs in this book can be run on the DIZI-2 board, while the motor programs must be run on the MOTA demo board. The simple introductory circuits and the motor board can be constructed using the same techniques as will be described for the DIZI board. The reader who is inexperienced in prototype construction is encouraged to attempt these tasks. The binary output counts from the BINx programs will be seen on the corresponding LED segments of the DIZI display, although the binary number is not displayed so clearly as it would be on a set of eight discrete LEDs, or an LED bar graph module.

DIZI-2 Board Design

A seven-segment display allows decimal and hexadecimal digits to be shown. A range of applications with a numerical output can thus be demonstrated, for example, the LOCK application detailed below. Port B has eight I/O bits; seven are used for the LED display, leaving RB0 free for use as both an audio output and a push button (interrupt) input. A small audio

transducer, a piezo electric buzzer, provides a simple and effective way of monitoring audio output frequencies or generating status signals.

Port A has five pins. RA4 can be used as an input to the TMR0 counter, so this was allocated as another push button input. A 4-bit switch bank is useful for setting coded inputs (e.g., BCD inputs for the LOCK application), so RA0–RA3 were allocated for this purpose. The switch and push button inputs have 100k pull-up resistors, and the push buttons have 22 F debouncing capacitors. These are fitted to such inputs because, when a switch closes, the metal contacts can bounce open again several times before finally closing. The CR network prevents this from causing multiple transitions on the logic signal input to which the switch is connected, because after the capacitor has quickly discharged on the first contact, it must recharge via the 100k. This takes a relatively long time, preventing the voltage from jumping back to a high level when the contacts re-open. By the same process, the CR network also ensures a smooth transition from low to high logic levels when the push button is released.

In addition, the CR network on RA4 was modified with a potentiometer (pot) connected as a variable resistance in series with the 100k, so that it could also be used to demonstrate the analogue input process in the LOCK program.

Circuit Diagram

Developing circuits for microcontrollers is not always too difficult, because the same circuit elements can often be reused in different designs. Thus, in the DIZI circuit (Fig. C.1), the

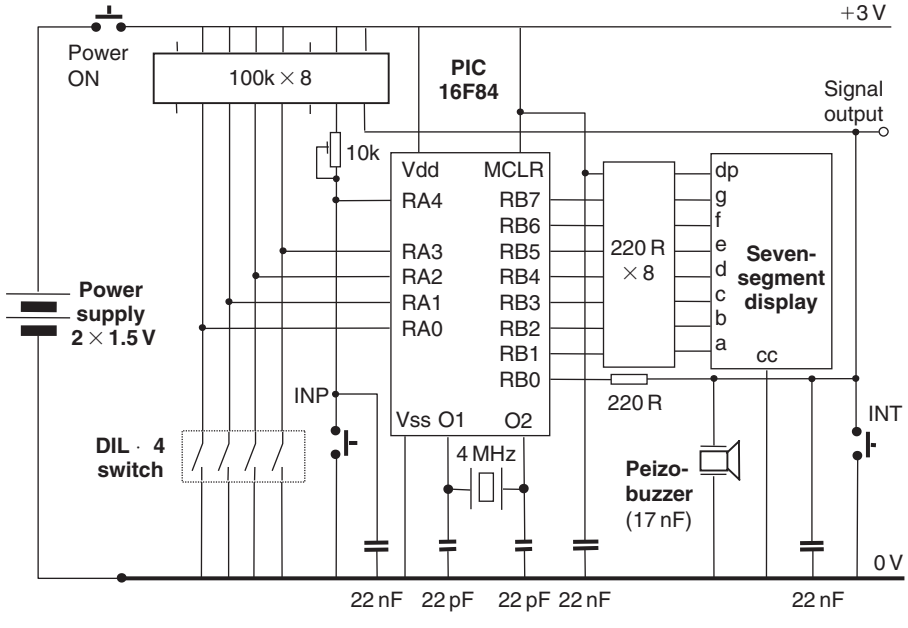


Figure C.1: DIZI-2 Board Circuit Diagram

switch inputs, display outputs and the clock circuit are standard arrangements of components. Remember, however, that, unlike the PIC, some microcontrollers cannot drive LEDs directly, but need a current driver stage inserted between each output and the LED.

DIZI-2 Board Layout

The layout of a PCB or prototype circuit is derived from the circuit diagram. The pins on DIL (Dual In-Line) chips are spaced 0.1″ apart, so the circuits must be laid out on a 0.1″ grid. When the pin out of each component has been established by reference to the data sheet or catalogue information, the connections can be mapped out on a square grid on paper. Alternatively, it is not too difficult to use the basic drawing tools in a word processor to do the same job. Examples of such connection diagrams are given in the text. A more refined layout drawing for the DIZI-2 board is shown in Fig. C.2.

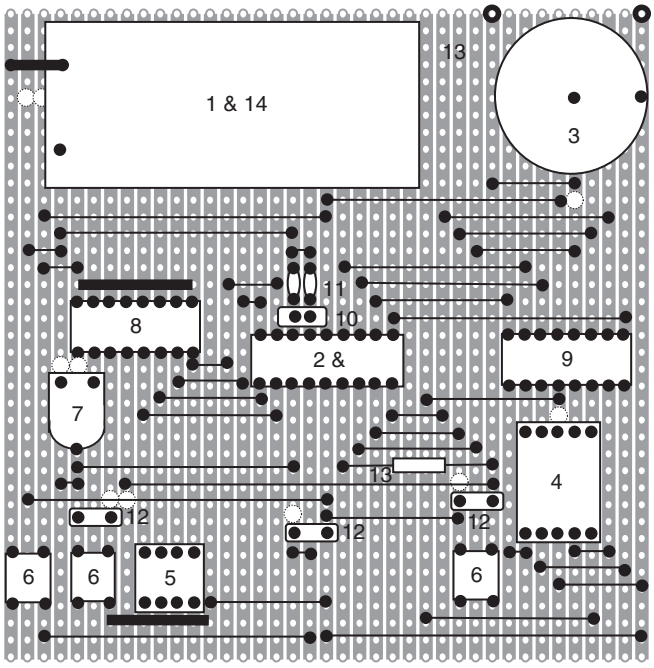


Figure B.2: DIZI Stripboard Layout

The board is viewed from the front (component) side, with the tracks on the back shown vertically. The components are numbered for reference to the parts listed below. The chips are all placed in the same orientation, so that pin 1 is bottom left. The seven-segment display has the decimal point bottom right. The ICs must obviously be fitted across the tracks, so that their pin connections are separated. The PIC chip should be fitted in a socket so that it can be removed for programming.

Horizontal links of tinned copper wire (TCW) complete the connections required. A solder joint is shown as a solid black dot. The broader solid lines indicate a continuous link across the tracks on the back of the board, where a set of adjacent tracks must be connected. Where required, the tracks are cut with a hand drill; these positions are shown as small white circles. The tracks must also be cut between the opposite pairs of pins on each DIL component, and, in this case, under the clock circuit capacitors (11).

A computer-drawing method allows component positioning to be easily adjusted so that the minimum area of stripboard is used. However, with experience, the circuit may be built directly onto the board without necessarily drawing the layout, perhaps with some wastage of board area.

Parts List

A parts list is required to specify the exact component when ordering from a suitable supplier. The availability of components varies over time, so updating is sometimes necessary. For example, the finger preset pot originally used in the design is no longer listed by the UK supplier, and had to be replaced with an equivalent part. The layout then had to be amended because the pin arrangement of the new component was different.

Layout	Description
1	Battery box, 2 × AA cells, PCB mounting
2	Microcontroller, PIC 16LF84-04
3	Piezo electric sounder, PCB mounting
4	Seven-segment LED display, 0.5″, Common Cathode
5	Piano DIL switch, 4-way
6	Tactile switch, PCB mounting (3 off) Caps for above: Red Blue Yellow
7	Preset potentiometer, 10 k, H-mount
8	DIL isolated resistor network, 100 k × 8
9	DIL isolated resistor network, 220 R × 8
10	Quartz crystal, general purpose, 4 MHz
11	Capacitor, 22 pF, ceramic (2 off)
12	Capacitor, 22 nF, polyester (3 off)
13	Stripboard, SRBP 3939 100 × 100 mm
14	Batteries, 1.5 V, size AA, Duracell (2 off)
15	18-pin DIL IC socket

Construction

When the layout has been checked against the circuit diagram, the main components can be inserted in the board and retained by, if necessary, slightly bending the corner pins outwards. All the pins should then be soldered to the tracks using the minimum amount of solder necessary, whilst ensuring that the joint is covered evenly with no cavities. At the same time, the soldering iron should be in contact with the joint for the shortest possible time, to avoid component overheating. The TCW links can also be retained before soldering by bending the ends towards each other. If a very neat job is required, one end can be soldered and the link stretched slightly before fixing the other end, to ensure that the link has no kinks in it, and that adjacent links do not touch; insulated TCW may be used on longer links if necessary. The tracks should then be cut where necessary, and the track side brushed with a small stiff brush to clear any debris. Rake between the tracks with a small screwdriver or knife to ensure that there are no short circuits left between adjacent tracks and solder joints.

Static Testing

Thoroughly re-inspect the board for correct connections, and that there are no debris, solder splashes or whiskers or dry joints. With the batteries not yet fitted, check with a multimeter that there is no short circuit between the power supplies. Fit the batteries, but not the PIC chip, and hold down the power button. The display decimal point should light. Check the supply voltages on the supply tracks and PIC socket pins: Pin 5 = 0 V and Pin 14 = +3 V. Check that the voltages at the PIC inputs change correctly as the switches are toggled. A DMM (Digital Multimeter) or oscilloscope is required for this test, because of the high impedance of the pull-up resistors. Connect a temporary link between Pin 14 (+3 V) on the PIC IC socket and each PIC output in turn, RB0–RB7. The peizo-buzzer should produce an audible "tick" and the LED segments should light.

Test Program

To complete testing of the DIZI-2 board, a program should be blown into the PIC which exercises all the hardware, while remaining as simple as possible so that there is no question of the software being faulty. A suitable program is listed in Program C.1; it does not test the analogue input operation, which will be covered later. When this program has been loaded, the following test procedure will confirm correct hardware operation.

```
; diz1.asm
; Test DIZI hardware ......................
          GOTO  inter  ; jump over delay
; Delay Subroutine ........................
```

Program C.1: DIZI Board Test Program

```
delay     MOVLW     0FF     ; Load FF
          MOVWF     0C      ; into counter
down      DECFSZ    0C      ; and decrement
          GOTO      down    ; until zero
          RETURN

; Check Interrupt Button ..................

inter     BTFSC     06,0    ; Test Button RB0
          GOTO      inter   ; until pressed

; Check   Display ............................

          MOVLW     00      ; Set PortB bits
          TRIS      06      ; as outputs
          MOVLW     0FF     ; Switch on all
          MOVWF     06      ; display segments

; Check Input Button ......................

input     BTFSC     05,4    ; Test Button RA4
          GOTO      input   ; until pressed

; Check DIP Switches and Buzzer ...........

again     MOVF      05,W    ; get DIL input &
          MOVWF     06      ; send to display
          RLF       06      ; rotate bits left
          BSF       06,0    ; set buzzer high
          CALL      delay   ; delay about 1ms
          BCF       06,0    ; reset buzzer low
          CALL      delay   ; delay about 1ms

          GOTO      again   ; and keep going..
          END               ; End of code
```

Program C.1: Continued

Step	Test	Result
1	Power Button On	Decimal point ON
2	Button B pressed and released	All display segments ON
3	Button A pressed and released	Buzzer sounds
4	Operate DIL switches	Segments a, b, c, d change

If faults are found, it is quite possible that there are still short circuits on the board. Check also that all the tracks have been cut as required, and that all connections are correct.

Analog to Digital Conversion

One feature of the DIZI board not described in the main text is the analogue input. A similar input is available on the MOTA board, so the method for using it will now be explained. Some

PIC chips, and other microcontrollers, have built-in ADCs, which allow analogue voltages to be converted to binary form for input to the processor. An ADC would be needed, for example, if a temperature is to be measured by the controller in a process system. The general block diagram for a counting ADC is shown in Fig. C.3.

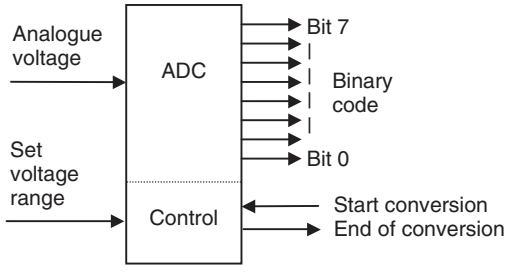


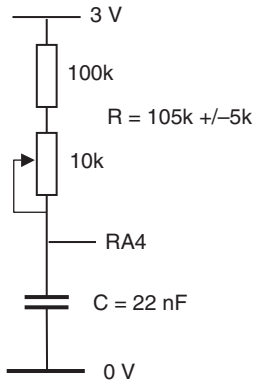
Figure C.3: General Analog to Digital Converter

The 8-bit ADC shown in Fig. C.3 converts the analog voltage present at its input to an 8-bit binary number, which means it can detect 256 different voltage levels. If the input range is set to, say, 0–2.55 V, then 255 steps of 10 mV can be detected. When the "Start Conversion" (GO bit) signal goes active, the DAC converts the binary number into a corresponding voltage. The "End of Conversion" signal to the processor is to indicate completion of the conversion process.

CR-ADC

The hardware for the standard ADC above is fairly complex, while the control process is relatively simple. An alternative is to use simple external hardware with a software conversion procedure, if there is no hardware ADC available. The CR-ADC is based on the measurement, using a counter register, of the rise time in a CR network connected to the processor system input. The CR converter will generally be slower and less accurate than a hardware-based ADC, but may be quite adequate in simple applications.

The components connected to RA4 in the DIZI circuit are shown in Fig. C.4. The capacitor-charging curve in Fig. C.5 shows the time constant of the circuit as 2.3 ms, assuming that the pot is set midway. This is the time taken to reach 63% of the final value (3 V) as the C charges via R. The PIC chip is a CMOS device, so the voltage level at which an input changes from logic 0 to 1, the threshold voltage, is around half the supply voltage, 1.5 V. Therefore the time taken to reach this level, here called the charging time, is estimated at 1.5 ms. This could be calculated more accurately from the formula for the charging of a capacitor, but as long as the circuit operation is consistent, it is not necessary for this application.



Estimation of charging time

Capacitor value
$= 22\ nF = 22 \times 10^{-9}\ F$
Resistor value with 10 k pot
set midway = 100k + 5k
$= 105k = 105 \times 10^{3}\ \Omega$
∴ Network time constant = C.R
$= 22 \times 10^{-9} \times 105 \times 10^{6} = 2.31$ ms
Estimated charging time ≈1.5 ms

Figure C.4: Components Connected to RA4

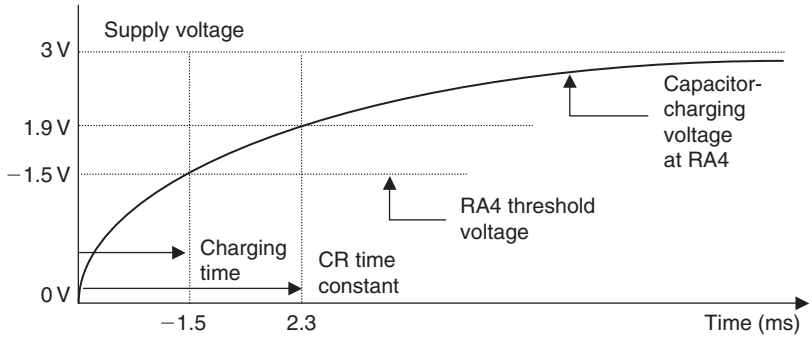


Figure C.5: CR Network Characteristic

The resistance, R, varies between 100 k and 110 k, depending on the position of the pot. The variation in the pot value will produce a corresponding variation in the rise time of the circuit. The rise time can be measured by discharging the capacitor and then counting while the voltage rises back towards the threshold. The capacitor is discharged by setting RA4 as an output and then setting the port data bit to zero. RA4 is then reconfigured as an input and checked at fixed time intervals while a register is incremented. The count is stopped when RA4 goes high.

The waveform which will be seen at RA4 is illustrated in Fig. C.6 (not to scale). A register labelled PotVal is incremented, and RA4 checked, within a loop taking, in the LOCK program, 20 μs to execute (see Fig. C.7). An adjustable delay routine allows the timing to be modified to suit the application and CR component values.

The result of the process is that a count is obtained which represents the setting of the pot. This could be converted to a resistance value if required, but in the LOCK program all we need is a variation in the displayed digit between 0 and 9, to allow the user to input a decimal

combination. Therefore, the delay associated with the count was simply adjusted to give one decade on the display with one turn of the pot. Only the low four bits of the count were required, so any decade of values could be used. The upper end of the 4-bit range, hex numbers A–F, are displayed as '-'. These can be used as "hidden" digits for extra security, if required.

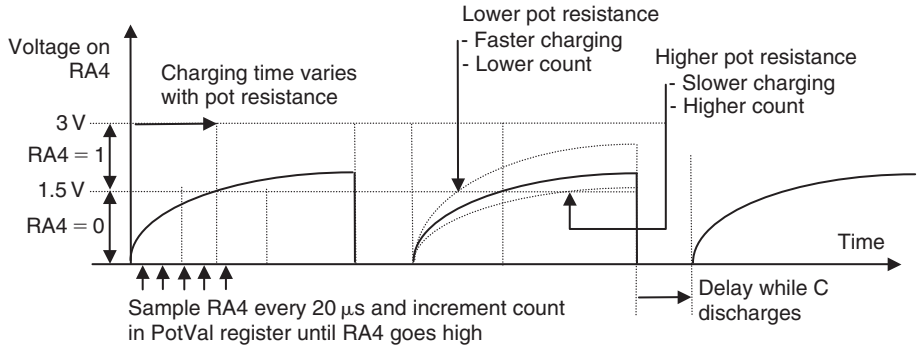


Figure C.6: Conversion Waveform at RA4

```
Set RA4 as Output
Clear RA4 to 0V to discharge C
Clear Counter Register
Set RA4 as Input
Test RA4 while C charges through R:
      Increment Counter Register
      Delay 20us
      Until RA4 = 1
Convert Count to Resistance or Pot Position
```

Figure C.7: CR-ADC Algorithm

A similar method can be used to operate the analog input on the MOTA board. The motor speed could then be controlled by a voltage source or sensor, or analog feedback employed to implement closed loop control.

EEPROM Memory

Nonvolatile read and write memory is very useful because data input by the user or acquired by the processor during its operation can be retained while the power is off. One important application area is data security and encryption. PIC devices are used, for example, in smart cards for controlling access to satellite television broadcast channels. The LOCK application illustrates this feature of the PIC 16F84 by using the EEPROM memory to store a 4-digit security code.

The PIC 16F84 has 64 bytes of EEPROM, with addresses 00–3F. The memory is accessed via EEDATA and EEADR in the SFR set. The EEPROM address is loaded into EEADR, and the data byte to be stored in EEDATA. An artificially complex write initialization sequence is then executed to actually write to the EEPROM memory, using EECON1 and EECON2 page 1 SFRs. The sequence is designed to reduce the possibility of an accidental write to the EEPROM, because a high level of reliability is required for security applications. This code sequence is given in the data sheet and LOCK program listing.

The read sequence, for retrieving the data, is more straightforward. Using EECON1, the data in the address pointed to by EEADR is returned in EEDATA. For accessing sequences of locations, EEADR can be incremented directly.

LOCK Application

In this demonstration application, a sequence of four decimal digits is stored in the PIC EEPROM memory from the DIL switch inputs. This sets the combination for the lock. To "open" the lock, the pot is rotated, and the input decimal digits are displayed and entered. This simulates the rotary action of mechanical combination locks. If the sequence of four input digits matches those previously stored in EEPROM, a siren sound is made to indicate the opening of the lock.

In the actual application, a solenoid-operated lock mechanism would be activated from this output, by replacing the siren sequence with an instruction to set an output bit. A suitable current driver interface for the solenoid would be required. Only the Power button, Enter button, Digit Select pot and display would be accessible to the user in the final design. The hardware would need to be reconfigured so that the unit would appear as shown in Fig. C.8 to the user. The DIL switch bank and its button for setting the entry code would be concealed.

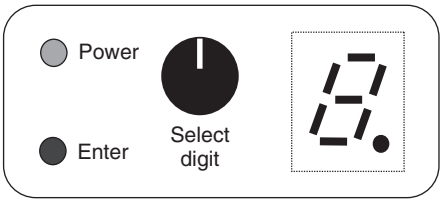


Figure C.8: Lock User Interface

Program Structure

The program contains the following blocks:

1. declaration of register and bit Labels
2. initialization of registers

3. sequence 1—Store combination
4. sequence 2—Check combination
5. end 1—Continuous siren output
6. end 2—Sleep
7. Subroutine 1—Display code table
8. Subroutine 2—Variable delay
9. Subroutine 3—Output one tone cycle
10. Subroutine 4—Get digit from pot

The program blocks should be ordered in such a way that labels referenced have already been defined when the program is assembled. Thus, the subroutines should be placed before the main sequences in the source code. However, when actually developing the code, if you are working "top down", the subroutines may actually be written after the main sequences. To place them correctly in the source code, they can then be inserted when editing, or cut and pasted later.

The program has two main sequences, for inputting and checking a combination, and two alternate endings. The processor goes to sleep after completion of the input sequence, or an incorrect digit match. The DIZI board must be re-powered to try again, as there are no other interrupts enabled to restart it. If the combination checks out correctly, the siren ending is used, which continues until the power goes off.

Pseudocode

The program is outlined below using "pseudocode", which is a text method of designing the program, which may be used instead of a flowchart. The pseudocode is developed in a word processor or the program source code editor until the statements are detailed enough to be converted into assembly code statements. In this case, it must be written in a form which allows it to be easily converted to PIC assembly language. The program structure and logic can thus be worked out before attempting to write the source code itself. To use pseudocode effectively, the programmer must be reasonably expert in using the language syntax.

The conventions used in the pseudocode are as follows:

- block structure applied
- target hardware specified

- Register and bit labels defined
- user inputs included in the sequence
- GOTO [deslab]
  – Jump to destination address label
- CALL [subnam]
  – Call subroutine at address label
  – Values passed to and received from subroutine defined
- GOTO [addlab] UNTIL (condition)
  – Implemented using Bit Test, Skip & GoTo operation
- (regname) = Contents of register labelled "regname"
- program block type defined:
  INIT = Initialize
  MAIN = Main program
  EQn = Sequence ending with GOTO
  ENDn = End operation
  SUBn = Subroutine, optionally receiving and/or returning values

LOCK List File

The list file for the LOCK program contains the source code and machine code. If the reader wishes to test the program, the machine code (column 2 of the list file) can be entered directly into the program memory buffer in the programming software. This avoids the need to type in the source code, if the hex file itself is not available.

The source code file uses the following conventions:

- full details of hardware and operation of application in source code;
- SFR, user and bit labels defined in separate blocks for clarity;
- block and line comments in source code;
- lower case for address labels;
- upper case for instruction mnemonics and SFR labels;
- capitalization of user register labels;
- identification and separation of block types.

The pseudocode and list files are reproduced in Programs C.2 and C.3.

```
LOCK PROGRAM PSEUDOCODE          MPB           29/8/99
************************************************
Hardware: DIZI PIC 16F84 Demo Board
************************************************
General Purpose Register Labels:
   0C = Period = Delay Period Preload Value
   0D = Count  = Delay Counter
   0E = PotVal = Count from ADC conversion
   0F = DigVal = Low 4 bits of PotVal
User Bit Labels:
   butA (RA4 input) - Normally 1
   butB (RB0 input) - Normally 1
   buzO (RB0 output)
See Data Sheet for SFR Labels and addresses
{Power Button On}
INIT: Initialise Port B ************************

   Port A defaults to inputs
   RA0 - RA3 = DIL Switches = 4-bit input
   RA4 = Input = butA = INP Button
   RB0 = Input = butB = INT Button
   RB1 - RB7 = Output = 7 Seg Display

MAIN: Select Set or Check Combination ***********

select        {Press Button A or B}
              If (butA)=0, GOTO [stocom]
              If (butB)=0, GOTO [checom]
              GOTO [select]

SEQ1: Store 4 digits in EEPROM, beep after each *

stocom        {Release Button A}
              CALL [delay] with (W)=FF
              GOTO [stocom] UNTIL (butA)=1

              Clear (EEADR)
getdil        {Set DIL Switches or Press A}
              Read (PORTA) into (W)
              Calc (W) AND 0F
              Store (W) in (EEDATA)
              CALL [codtab] with (W)=00-0F
              {Returns with '7SegCode' in (W)}
              Output (W) to (PORTB)
              GOTO [getdil] UNTIL (butA)=0
```

Program C.2: Lock Program Pseudocode

```
waita       {Release Button A}
            GOTO [waita] UNTIL (butA)=1
            Store (EEDATA) in (EEADR)
            CALL [beep]
            Increment (EEADR) from 00 to 04
            GOTO [getdil] UNTIL (EEADR)=4
            CALL [beep]
            CALL [beep]
            GOTO [done]

SEQ2: Check 4 digits from pot for match *************

checom      {Release Button B}
            CALL [delay] with (W)=FF
            GOTO [checom] UNTIL (butB)=1

            Clear (EEADR)
potin       {Adjust Pot or Press Button B}

            CALL [getpot] for (DigVal)
            {Returns with (DigVal)=00-0F}
            GOTO [potin] UNTIL (butB)=0
            Read (EEDATA) at (EEADR)
            Compare (EEDATA) with (DigVal)
            If (Z)=0 GOTO [done]

waitb       {Release Button B}
            GOTO [waitb] UNTIL (butB)=1
            CALL [beep]
            Increment (EEADR)
            GOTO [potin] UNTIL EEADR=4
            GOTO [siren]

END1: Sequences matches, sound siren ******************
siren       CALL [beep]
            GOTO [siren]

END2: Digit compare failed, finish ********************
done        Clear (PORTB)
            Sleep

SUBROUTINES ****************************************

SUB1: Get Display Code
      Receives: Table Offset in W
      Returns: 7-Segment Display Code in W
```

Program C.2: Continued

```
codtab          Add (W) to (PCL)
                RETURN with '7SegCode' in (W)

SUB2: Variable Delay
      Receives: (Count) in W

delay           Load (Count) from (W)
                Decrement (Count) UNTIL (Count)=0
                RETURN

SUB3: Outputs one cycle of sound output
      Receives: (Period)

beep            Load (Period) with FF
                Set RB0 as 0utput

cycle           Set (BuzO)=1
                CALL [delay] with (Period) in W
                Set BuzO=0
                CALL [delay] with (Period) in W
                Decrement (Period) from FF to 00
                GOTO [cycle] UNTIL (Period)=0
                Reset RB0 as Input
                RETURN

SUB4: Get Pot Value using CR ADC method

      Returns: (DigVal)=00-0F

getpot          Set RA4 as Output
                Clear (RA4)
                CALL [delay] with (W)=FF
                Reset RA4 as Input

                Clear (PotVal)
check           Increment (PotVal) from 00 to XX
                CALL [delay] with (W)=3
                GOTO [check] UNTIL (RA4)=1

                (DigVal) = (PotVal) AND 0F
                CALL [codtab] with (DigVal)=00-0F
                RETURN

END OF LOCK PROGRAM ********************************
```

Program C.2: Continued

```
00001 ;*************************************************************
00002 ; LOCK.ASM                                          MPB 17/8/99
00003 ;*************************************************************
00004 ;
00005 ; Four digit combination lock simulation demonstrates the
      ; hardware
00006 ; features of the DIZI demo board and the PIC 16F84.
00007 ;
00008 ; Hardware: DIZI Demo Board with PIC 16F84 (4MHz)
00009 ; Setup:     RA0-RA3      DIL Switch Inputs
00010 ;            RA4          Push Button Input / Analogue Input
00011 ;            RB0          Push Button Input / Audio Output
00012 ;            RB1-RB7      7-Segment Display Output
00013 ; Fuses:     WDT off, PuT on, CP off
00014 ;
00015 ; Operation ----------------------------------------
00016 ;
00017 ; To set the combination, a sequence of 4 digits is input on
      ; the DIL
00018 ; piano switches; this is retained in the EEPROM when power
      ; is off.
00019 ; To 'open' the lock, a sequence of 4 digits is input via
00020 ; the potentiometer. These are compared with the stored data,
00021 ; and an audio output generated to indicate the correct
      ; sequence.
00022 ; The processor halts if any digit fails to match, and the
00023 ; program must be restarted.
00024 ;
00025 ;        To set a combination:
00026 ;        1. Hold Power On Button
00027 ;        2. Press Button A
00028 ;        3. Set a digit on DIL switches and Press A -beeps
00029 ;        4. Repeat step 3 for 3 more digits
00030 ;        5. Release Power Button
00031 ;
00032 ;        To check a combination:
00033 ;        1. Hold Power On Button
00034 ;        2. Press Button B
00035 ;        3. Set a digit on pot and Press B -beeps if matched
00036 ;        4. Repeat step 3 for 3 more digits
00037 ;           - if digits all match, siren is sounded
00038 ;           - if any digit fails to match, the processor
;                  halts
00039 ;        5. Release Power Button
00040 ;
```

Program C.3: Lock Program List File

```
      00041 ; *******************************************************
      00042              PROCESSOR 16F84    ; Processor Type Directive
      00043 ; *******************************************************
      00044
      00045 ; EQU: Special Function Register Equates.................
      00046
0002  00047 PCL     EQU   02  ; Program Counter Low
0005  00048 PORTA   EQU   05  ; Port A Data
0006  00049 PORTB   EQU   06  ; Port B Data
0003  00050 STATUS  EQU   03  ; Flags
0008  00051 EEDATA  EQU   08  ; EEPROM Memory Data
0009  00052 EEADR   EQU   09  ; EEPROM Memory Address
0008  00053 EECON1  EQU   08  ; EEPROM Control Register 1
0009  00054 EECON2  EQU   09  ; EEPROM Control Register 2
      00055
      00056 ; EQU: User Register Equates.............................
      00057
000C  00058 Period  EQU   0C  ; Period of Output Sound
000D  00059 Count   EQU   0D  ; Delay Down Counter
000E  00060 PotVal  EQU   0E  ; Analogue Input Value
000F  00061 DigVal  EQU   0F  ; Current Digit Value 00 to 09
      00062
      00063 ; EQU: SFR Bit Equates..................................
      00064
0005  00065 RP0     EQU   5   ; STATUS - Register Page Select
0000  00066 RD      EQU   0   ; EECON1 - EEPROM Memory Read Byte Initiate
0001  00067 WR      EQU   1   ; EECON1 -EEPROM Memory Write Byte
                              ; Initiate
0002  00068 WREN    EQU   2   ; EECON1 -EEPROM Memory Write Enable
0002  00069 Z       EQU   2   ; STATUS - Zero Flag
      00070
      00071 ;       EQU:  User Bit Equates.........................
      00072
0004  00073 butA    EQU   4   ; PORTA - RA4 Input Button
0000  00074 butB    EQU   0   ; PORTB - RB0 Input Button
0000  00075 buzO    EQU   0   ; PORTB - RB0 Output Buzzer
      00076
      00077 ; *******************************************************
      00078
      00079 ; INIT: Initialise Port B (Port A defaults to inputs)
      00080
```

Program C.3: Continued

```
0000  3001 00081   start    MOVLW  001    ; RB0 = Input, RB1-RB7 = Outputs
0001  0066 00082            TRIS   PORTB  ; Set Data Direction
0002  0086 00083            MOVWF  PORTB  ; Clear Data
0003  286D 00084            GOTO   select ; Select Combination Read or Write
           00085
           00086 ; SUBROUTINES ***************************************
           00087
           00088 ; SUB1: 7-Segment Code Table using PCL + offset in W
                         ; Returns
           00089 ;       digit display codes, with '-' for numbers
                         A to F
           00090
0004  0782 00091  codtab ADDWF  PCL          ; Add offset to Program Counter
0005  347E 00092         RETLW  B'01111110'  ; Return with display code for '0'
0006  340C 00093         RETLW  B'00001100' ; Return with display code for '1'
0007  34B6 00094         RETLW  B'10110110' ; Return with display code for '2'
0008  349E 00095         RETLW  B'10011110' ; Return with display code for '3'
0009  34CC 00096         RETLW  B'11001100' ; Return with display code for '4'
000A  34DA 00097         RETLW  B'11011010' ; Return with display code for '5'
000B  34FA 00098         RETLW  B'11111010' ; Return with display code for '6'
000C  340E 00099         RETLW  B'00001110' ; Return with display code for '7'
000D  34FE 00100         RETLW  B'11111110' ; Return with display code for '8'
000E  34DE 00101         RETLW  B'11011110' ; Return with display code for '9'
000F  3480 00102         RETLW  B'10000000' ; Return with display code for '-'
0010  3480 00103         RETLW  B'10000000' ; Return with display code for '-'
0011  3480 00104         RETLW  B'10000000' ; Return with display code for '-'
0012  3480 00105         RETLW  B'10000000' ; Return with display code for '-'
0013  3480 00106         RETLW  B'10000000' ; Return with display code for '-'
0014  3480 00107         RETLW  B'10000000' ; Return with display code for '-'
           00108
           00109 ; -------------------------------------------------
           00110 ; SUB2:   Delay routine
           00111 ;         Receives delay count in W
           00112
0015  008D 00113  delay    MOVWF    Count       ; Load counter from W
0016  0B8D 00114  loop     DECFSZ   Count       ; and decrement
0017  2816 00115          GOTO     loop        ; until zero
0018  0008 00116          RETURN               ; and return
           00117
           00118 ; -------------------------------------------------------
           00119 ; SUB3: Output One Beep Cycle to BuzO
           00120
0019  30FF 00121  beep     MOVLW    0FF         ; Load FF into
001A  008C 00122  MOVWF    Period               ; Period counter
           00123
```

Program C.3: Continued

```
001B 3000 00124         MOVLW    B'00000000'  ; Set RB0
001C 0066 00125         TRIS     PORTB        ; as output
          00126
          00127 ; Do one cycle of rising tone....
          00128
001D 1406 00129 cycle   BSF      PORTB,buzO   ; Output High
001E 080C 00130         MOVF     Period,W     ; Load W with Period
                                              ; value
001F 2015 00131         CALL     delay        ; and delay for Period
          00132
0020 1006 00133         BCF      PORTB,buzO   ; Output Low
0021 2015 00134         CALL     delay        ; and delay for same
                                              ; Period
0022 0B8C 00135         DECFSZ   Period       ; Decrement Period
0023 281D 00136         GOTO     cycle        ; and do next cycle
                                              ; until 0
          00137
          00138 ; Set RB0 to input again.............................
          00139
0024 3001 00140         MOVLW    B'00000001'  ; Reset RB0
0025 0066 00141         TRIS     PORTB        ; as input
0026 0008 00142         RETURN                ; from tone cycle
          00143
          00144 ; ------------------------------------------------
          00145 ; SUB4: Get pot value (Rv) using rise time due to C
                        ; and R on RA4
          00146 ;       Returns with digit value (0-F) in DigVal
          00147
          00148 ; Discharge external capacitor on RA4
          00149
0027 300F 00150 getpot MOVLW   B'00001111' ; Set RA4
0028 0065 00151        TRIS    PORTA       ; as output
0029 1205 00152        BCF     PORTA,4     ; and discharge C setting
                                           ; output low
002A 30FF 00153        MOVLW   0FF         ; Delay for about 1ms
002B 2015 00154        CALL    delay       ; to ensure C is discharged
002C 301F 00155        MOVLW   B'00011111' ; Reset RA4
002D 0065 00156        TRIS    PORTA       ; as input
          00157
          00158 ; Increment a counter until RA4 goes high due to
                ; charging of C
          00159
002E 018E 00160        CLRF    PotVal      ; Clear input value counter
002F 0A8E 00161 check  INCF    PotVal      ; increment counter
0030 3003 00162        MOVLW   03          ; Set delay count to 3
```

Program C.3: Continued

```
0031 2015 00163         CALL    delay       ; and delay between
                                            ; input checks
0032 1E05 00164         BTFSS   PORTA,4     ; Check input bit RA4
0033 282F 00165         GOTO    check       ; and repeat if not yet high
          00166
          00167 ; Mask out high bits of count value, and store &
                  display
          00168 ; 4-bit digit value, 0-F
          00169
0034 080E 00170         MOVF    PotVal,W    ; Put count value in W
0035 390F 00171         ANDLW   00F         ; and set high 4 bits to 0
0036 008F 00172         MOVWF   DigVal      ; Store 4-bit value
0037 2004 00173         CALL    codtab      ; Get 7-segment code, 0-9
0038 0086 00174         MOVWF   PORTB       ; and display
          00175
0039 0008 00176         RETURN              ; with DigVal from setting
                                            ; of pot
          00177
          00178 ; MAIN SEQUENCES **********************************
          00179
          00180 ; SEQ1: Store 4 Digits in non volatile EEPROM
          00181 ; Beep after each digit, and twice when 4 done
          00182
          00183 ; Complete Button A input operation
          00184
003A 30FF 00185 stocom MOVLW   0FF         ; Delay for about 1ms
003B 2015 00186         CALL    delay       ; to avoid Button A switch
                                            ; bounce
003C 1E05 00187         BTFSS   PORTA,butA  ; Wait for Button A
003D 283A 00188         GOTO    stocom      ; to be released
          00189
          00190 ; Read 4-bit binary number from DIL switches into
                ; EEDATA and display
          00191
003E 0189 00192         CLRF    EEADR       ; Zero EEPROM address
                                            ; register
003F 0805 00193 getdil MOVF    PORTA,W     ; Read DIL switches
0040 390F 00194         ANDLW   0F          ; and set high 4 bits to 0
0041 0088 00195         MOVWF   EEDATA      ; Put DIL value in EEPROM
                                            ; data
          00196
0042 2004 00197         CALL    codtab      ; Display DIL input as
                                            ; decimal
0043 0086 00198         MOVWF   PORTB       ;
          00199
```

Program C.3: Continued

```
0044 1A05 00200        BTFSC   PORTA,butA  ; Check if Button A pressed
0045 283F 00201        GOTO    getdil      ; If not, keep reading DIL
                                           ; input
          00202
          00203 ; Store the current DIL input in EEPROM at current
                  address
          00204
0046 1683 00205 store  BSF     STATUS,RP0  ; Select Register Bank 1
0047 1508 00206        BSF     EECON1,WREN ; Enable EEPROM write
0048 3055 00207        MOVLW   055         ; Write initialisation
                                           ; sequence
0049 0089 00208        MOVWF   EECON2      ;
004A 30AA 00209        MOVLW   0AA         ;
004B 0089 00210        MOVWF   EECON2      ;
004C 1488 00211        BSF     EECON1,WR   ; Write data into current
                                           ; address
004D 1283 00212        BCF     STATUS,RP0  ; Re-select Register Bank 0
          00213
004E 1E05 00214 waita  BTFSS   PORTA,butA  ; Wait for Button A to be
                                           ; released
004F 284E 00215        GOTO    waita       ;
0050 2019 00216        CALL    beep        ; Beep to indicate digit
                                           ; write done
          00217
          00218 ; Checkif 4 digits have been stored yet, if not, get
                  next
0051 0A89 00220        INCF    EEADR       ; Select next EEPROM
                                           ; address
0052 1D09 00221        BTFSS   EEADR,2     ; Is the address now = 4?
0053 283F 00222        GOTO    getdil      ; If not, get next digit
0054 2019 00224        CALL    beep        ; Beep twice when 4 digits
                                           ; stored
0055 2019 00225        CALL    beep        ;
0056 2874 00226        GOTO    done        ; Go to sleep when done
          00228 ; -----------------------------------------------
          00230 ; SEQ2:Check PotVal v EEPROM
          00231
0057 30FF 00232 checom MOVLW   0FF         ; Delay for about 1ms
0058 2015 00233        CALL    delay       ; to avoid Button B switch
                                           ; bounce
0059 1C06 00234        BTFSS   PORTB,butB  ; Wait for Button B to be
                                           ; released
005A 2857 00235        GOTO    checom      ;
          00236
          00237 ; Read the value set on the input pot
```

Program C.3: Continued

```
          00238
005B 0189 00239        CLRF    EEADR       ; Zero EEPROM address
005C 2027 00240 potin  CALL    getpot      ; Get a digit value set
                                           ; onpot (Rv)
005D 1806 00241        BTFSC   PORTB,butB  ; Check in Button pressed
                                           ; again
005E 285C 00242        GOTO    potin       ; If not, keep reading the
                                           ; pot
          00243
          00244 ; Get a digit value from EEPROM and compare with the
                ; pot input
          00245
005F 1683 00246        BSF     STATUS,RP0  ; Select Register Bank 1
0060 1408 00247        BSF     EECON1,RD   ; Read selected EEPROM
                                           ; location
0061 1283 00248        BCF     STATUS,RP0  ; Re-select Register Bank 0
0062 0808 00249        MOVF    EEDATA,W    ; Copy EEPROM data to W
          00250
0063 068F 00251        XORWF   DigVal      ; Compare the input with
                                           ; EEPROM data
0064 1D03 00252        BTFSS STATUS,Z      ; If it does not match, go
                                           ; to sleep
0065 2874 00253        GOTO    done        ;
          00254
          00255 ; If digit match obtained, check if 4 done and do
                ; next if not
          00256
0066 1C06 00257 waitb BTFSS PORTB,butB     ; Wait for Button B to be
                                           ; released
0067 2866 00258        GOTO    waitb       ;
0068 2019 00259        CALL    beep        ; Beep to confirm successful
                                           ; match
          00260
0069 0A89 00261        INCF    EEADR       ; Select next EEPROM
                                           ; location
006A 1D09 00262        BTFSS   EEADR,2     ; 4 digits checked yet?
006B 285C 00263        GOTO    potin       ; If not, do the next
006C 2872 00264        GOTO    siren       ; When 4 digits done, run
                                           ; siren
          00265
          00266 ; ************************************************
          00267
          00268 ; MAIN: Select Set or Check Combination
          00269
006D 1E05 00270 select BTFSS   PORTA,butA  ; Button A pressed?
```

Program C.3: Continued

```
006E 283A 00271        GOTO    stocom      ; If so, store a
                                           ; combination
006F 1C06 00272        BTFSS   PORTB,butB  ; Button B pressed?
0070 2857 00273        GOTO    checom      ; If so, check a
                                           ; combination
0071 286D 00274        GOTO    select      ; repeat endlessly
          00275
          00276 ; ************************************************
          00277
          00278 ; END1: When combination successfully matched, make
                ; siren sound
          00279
0072 2019 00280 siren  CALL    beep        ; Do a tone cycle
0073 2872 00281        GOTO    siren       ; and repeat endlessly
          00282
          00283 ; ------------------------------------------
          00284
          00285 ; END2: When a digit check fails, go to sleep, and
                ; try again
          00286
0074 0186 00287 done   CLRF    PORTB       ; Switch off display
0075 0063 00288        SLEEP               ; Processor halts
          00289
          00290 ; ************************************************
          00291        END                 ; of program source code

SYMBOL TABLE
LABEL                  VALUE

Count                  0000000D
DigVal                 0000000F
EEADR                  00000009
EECON1                 00000008
EECON2                 00000009
EEDATA                 00000008
PCL                    00000002
PORTA                  00000005
PORTB                  00000006
Period                 0000000C
PotVal                 0000000E
RD                     00000000
RP0                    00000005
STATUS                 00000003
WR                     00000001
WREN                   00000002
Z                      00000002
```

Program C.3: Continued

```
__16C84                 00000001
beep                    00000019
butA                    00000004
butB                    00000000
buzO                    00000000
check                   0000002F
checom                  00000057
codtab                  00000004
cycle                   0000001D
delay                   00000015
done                    00000074
getdil                  0000003F
getpot                  00000027
loop                    00000016
potin                   0000005C
select                  0000006D
siren                   00000072
start                   00000000
stocom                  0000003A
store                   00000046
waita                   0000004E
waitb                   00000066

MEMORY USAGE MAP ('X' = Used, '-' = Unused)
0000 : XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXX
0040 : XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX XXXXXX -----

All other memory blocks unused
```

Program C.3: Continued

APPENDIX D

Program M

```
; ******************************
; written by: John Morton      *
; date: 10/01/05               *
; version: 1.0                 *
; file saved as: dice.asm      *
; for P12F508                  *
; clock frequency: Int. 4 MHz  *
; ******************************

; PROGRAM FUNCTION: A pair of dice.

          list      P=12F508
          include   "c:\pic\p12f508.inc"

          __config  _MCLRE_OFF & _CP_OFF & _WDT_OFF & _IntRC_OSC

;==============
; Declarations:

Die1num   equ       10h
Die2num   equ       11h
Mark60    equ       12h
PostX     equ       13h
PostVal   equ       14h
Ran1      equ       15h
Ran2      equ       16h
General   equ       17h
Random    equ       18h

#define   slow      General, 0

          org       0           ; first instruction to be executed
          movwf     OSCCAL      ; calibrates internal oscillator
          goto      Start       ;

;=======================================
; Subroutines:

Init      movlw     b'100000'   ; turns off all LEDs
          movwf     GPIO        ;
```

```
            movlw    b'001000'    ; sets up which pins are inputs
            tris     GPIO         ;   and which are outputs

            movlw    b'01000111'  ; enable wake-on-change, disable weak
            option                ;   pull-ups, TMR0 prescaled by 256

            movlw    d'4'         ; sets up postscalers
            movwf    PostX        ;
            movwf    PostVal      ;

            clrf     Die1num      ; clears display registers
            clrf     Die2num      ;
            clrf     Ran1         ; clears random number registers
            clrf     Ran2         ;
            bcf      slow         ; clears 'slow-down' flag
            retlw    0            ;

;==================================
Display     btfss    TMR0, 4      ; uses bit 4 of TMR0 to choose die
            goto     Die2         ;

            movfw    Die1num      ; gets number to display
            call     Code1        ; converts to code
            movwf    GPIO         ; outputs
            retlw    0            ;

Die2        movfw    Die2num      ; gets number to display
            call     Code2        ; converts to code
            movwf    GPIO         ; outputs
            retlw    0            ;

; arrangement for dice 1 is : CTLR, D, -, A, C, B

Code1       addwf    PCL, f       ;
            retlw    b'100000'    ; all off
            retlw    b'100100'    ; 1
            retlw    b'100001'    ; 2
            retlw    b'100101'    ; 3
            retlw    b'100011'    ; 4
            retlw    b'100111'    ; 5
            retlw    b'110011'    ; 6
            retlw    b'110111'    ; all on

; arrangement for dice 2 is : CTLR, C, -, B, A, D

Code2       addwf    PCL, f       ;
            retlw    b'010111'    ; all off
            retlw    b'010101'    ; 1
            retlw    b'010011'    ; 2
```

```
          retlw    b'010001'    ; 3
          retlw    b'000011'    ; 4
          retlw    b'000001'    ; 5
          retlw    b'000010'    ; 6
          retlw    b'000000'    ; all on

;==================================
Timing    movfw    Mark60       ; 1/40th second delay
          subwf    TMR0, w      ;
          btfss    STATUS, Z    ;
          retlw    0            ;

          movlw    d'60'        ; resets marker
          addwf    Mark60, f    ;

          decfsz   PostX, f     ; variable further delay
          retlw    0            ;

          call     RandomGen    ; generate new pseudo-random
          swapf    Random, w    ;  number
          andlw    b'00000111'  ; converts to 0-7 and moves
          movwf    Die1num      ;  into Die1num

          call     RandomGen    ; generate new pseudo-random
          swapf    Random, w    ;  number
          andlw    b'00000111'  ; converts to 0-7 and moves
          movwf    Die2num      ;  into Die2num

          btfsc    slow         ; should this slow down?
          call     Slowdown     ; yes

          movfw    PostVal      ; updates variable delay length
          movwf    PostX        ;
          retlw    0

Slowdown  incf     PostVal, f   ; increases delay length
          btfsc    PostVal, 5   ; has PostVal reached 32?
          clrf     PostVal      ; resets, telling 'Released' section
          retlw    0            ;  that the dice have stopped rolling

RandomGen
          movlw    d'63'        ; newRandom =
          addwf    Random, w    ;  63 + oldRandom x 3
          addwf    Random, w    ;
          addwf    Random, f    ;
          retlw    0            ;
```

```
;==================================
RandomScroll
          incf      Ran1, f         ; v. quickly scrolls through
          movlw     d'6'            ;   has Ran1 reached 6?
          subwf     Ran1, w         ;
          btfss     STATUS, Z       ;
          retlw     0               ; no, so returns

          clrf      Ran1            ;
          incf      Ran2, f         ;
          movlw     d'6'            ; has Ran1 reached 6?
          subwf     Ran2, w         ;
          btfss     STATUS, Z       ;
          retlw     0               ; no, so returns
          clrf      Ran2            ;
          retlw     0               ;

;==================================
; PROGRAM START

Start     call      Init            ; initial settings

Pressed   btfsc     GPIO, 3         ; tests button
          goto      Released        ; branches when released
          call      RandomScroll    ; quickly scrolls through no.s
          call      Timing          ; keeps flashing going
          call      Display         ; keeps displays changing
          goto      Pressed         ;

Released  bsf       slow            ; tells Timing to slow down
          call      Timing          ; keeps flashing going
          call      Display         ; keeps displays going
          movf      PostVal, f      ; have dice stopped rolling?
          btfss     STATUS, Z       ;
          goto      Released+1      ; no, so keeps looping

          incf      Ran1, w         ; moves 1+ the random number
          movwf     Die1num         ;   into the display regs.
          incf      Ran2, w         ;
          movwf     Die2num         ;

          movlw     d'240'          ; 240 x 1/40th second = 6 second
          movwf     PostX           ;   delay

EndLoop   call      Display         ; 6 second delay, after which all
          movfw     Mark60          ; LEDs are turned off
          subwf     TMR0, w         ;
```

```
btfss     STATUS, Z      ;
goto      EndLoop        ;

movlw     d'60'          ;
addwf     Mark60, f      ;

decfsz    PostX, f       ;
goto      EndLoop        ;

movlw     b'100000'      ; turns off all LEDs
movwf     GPIO           ;
sleep                    ; goes to low power mode

END
```

This page intentionally left blank

APPENDIX E

Program N

```
;**************************************
; written by: John Morton              *
; date: 14/03/05                       *
; version: 1.0                         *
; file saved as: quiz.asm              *
; for PIC12F675                        *
; clock frequency: Int. 4 MHz          *
;**************************************

; Program Description: Quiz controller for 3 players, including reset
;  button for the quiz master.

          list      P=12F675
          include   "c:\pic\p12f675.inc"

;==============
; Declarations:

temp      equ       20h
Post16    equ       21h

          org       0               ; first instruction to be executed
          goto      Start           ;

          org       4               ; interrupt service routine
          goto      isr             ;

;==============
; Subroutines:

Init      bsf       STATUS, RP0     ; goes to Bank 1
          call      3FFh            ; calls calibration address
          movwf     OSCCAL          ; moves w. reg into OSCCAL

          movlw     b'011110'       ; GP5: Buzzer, GP3: Reset button
          movwf     TRISIO          ; GP1,2,4: LEDs/Buttons (inputs
                                    ;   to start with), GP0: LED enable
```

```
        movlw   b'010110'       ; GP1,2,4 have weak pull-ups
        movwf   WPU             ;   enabled

        movlw   b'00000111'     ; pull-ups enabled, TMR0 presc.
        movwf   OPTION_REG      ;   by maximum amount (256)
        clrf    PIE1            ; turns off peripheral interrupts
        movlw   b'010110'       ; enables GPIO change interrupt
        movwf   IOC             ;   on GP1, GP2 and GP4 only
        clrf    VRCON           ; turns off comparator V. ref.
        clrf    ANSEL           ; makes GP0:3 digital I/O pins
        bcf     STATU S, RP0    ; back to Bank 0
        clrf    GPIO            ; resets input/output port
        movlw   b'00001000'     ; enables GPIO change interrupt
        movwf   INTCON          ;   only
        movlw   b'00000111'     ; turns off comparator
        movwf   CMCON           ;
        clrf    T1CON           ; turns off TMR1
        clrf    ADCON0          ; turns off A to D converter

        movlw   d'16'           ; sets up postscaler
        movwf   Post16          ;
        retfie                  ; returns, enabling interrupts

;====================
; Interrupt Service Routine
isr     btfss   INTCON, 0       ; checks GPIO change int. flag
        goto    Timer           ; TMR0 interrupt occurred . . .
                                ; GPIO interrupt occurred . . .
        bcf     INTCON, 0       ; resets interrupt flag

        comf    GPIO, w         ; stores state of GPIO
        andlw   b'010110'       ; masks all except buttons
        movwf   temp            ;
        btfsc   STATUS, Z       ; are any buttons actually pressed?
        retfie                  ; false alarm

        bsf     STATUS, RP0     ; moves to Bank 1
        movlw   b'001000'       ; makes GP1,2,4 outputs
        movwf   TRISIO          ;
        bcf     STATUS, RP0     ; moves to Bank 0
        movfw   temp            ; moves temp back into GPIO,
        addlw   b'100001'       ;   sets GP5 and GP0 (turns on
        movwf   GPIO            ;   buzzer and enables LEDs)

        movlw   b'00100000'     ; enables TMR0 interrupt, disables
        movwf   INTCON          ;   the GPIO change interrupt
        retfie                  ; returns, enabling GIE
```

```
Timer     bcf       INTCON, 2       ; resets TMR0 interrupt flag
          decfsz    Post16, f       ; is this the 16th TMR0 interrupt
          retfie                    ;

          bcf       GPIO, 5         ; turn off buzzer
          clrf      INTCON          ; turns off all interrupts
          sleep                     ; goes into low power mode

;============
; Program Start

Start     call      Init            ; initialization routine

Main      goto      Main            ; keeps looping

          END
```

This page intentionally left blank

APPENDIX F

Program O

```
;****************************************
; written by: John Morton                *
; date: 10/01/05                         *
; version 1.0                            *
; file saved as phonecard.asm            *
; for P12F675                            *
; clock frequency: internal 4 MHz        *
; ****************************************

; Program Description: A smart card for a phone box.

                list          P=12F675
                include       "c:\pic\p12f675.inc"

;======================
; Declarations:

W_temp          equ     20h
STATUS_temp     equ     21h
Temp            equ     22h
Mark125         equ     23h
Post125         equ     24h
Post15          equ     25h

                org     0               ; first instruction to be executed
                goto    Start           ;
                org     4               ; interrupt service routine
                goto    isr             ;

;===========
; Subroutines:

Init            bsf     STATUS, RP0     ; goes to Bank 1
                call    3FFh            ; calls calibration address
                movwf   OSCCAL          ; moves w. reg into OSCCAL
                movlw   b'111110'       ; all inputs except GP0
                movwf   TRISIO          ;
                clrf    WPU             ; weak pull-ups disabled
```

```
                movlw   b'11000111'     ; sets up timer and some pin
                movwf   OPTION_REG      ;   settings
                clrf    PIE1            ; turns off peripheral ints.
                clrf    IOC             ; disables GPIO change int.
                clrf    VRCON           ; turns off comparator V. ref.
                clrf    ANSEL           ; makes GP0:3 digital I/O pins

                bcf     STATUS, RP0     ; back to Bank 0
                clrf    GPIO            ; resets input/output port
                movlw   b'00010000'     ; sets up interrupts
                movwf   INTCON          ;
                movlw   b'00000111'     ; turns off comparator
                movwf   CMCON           ;
                clrf    T1CON           ; turns off TMR1
                clrf    ADCON0          ; turns off A to D conv.

                movlw   d'125'          ; sets up postscalers
                movwf   Post125         ;
                movlw   d'15'           ;
                movwf   Post15          ;
                retfie                  ; returns from Init

isr             movwf   W_temp          ; stores w. reg in temp register
                movfw   STATUS          ; stores STATUS in temp
                movwf   STATUS_temp     ;   register

                bcf     INTCON, 1       ; resets INT interrupt flag
                bcf     STATUS, RP0     ; makes sure we're in Bank 0
                movfw   GPIO            ; reads value of GPIO
                movwf   temp            ;
                rrf     temp, f         ; rotates right three times...
                rrf     temp, f         ;
                rrf     temp, w         ; ...leaving result in w. reg
                andlw   b'000111'       ; masks bits 3-5
                call    CardValue       ; converts code into minutes

                bsf     STATUS, RP0     ; goes to Bank 1
                movwf   EEDATA          ; stores minutes in EEDATA
                clrf    EEADR           ; selects EEPROM address 00h
                bsf     EECON1, 2       ; enables a write operation
                movlw   0x55            ; now follows the 'safe
                movwf   EECON2          ;   combination'
                movlw   0xAA            ;
                movwf   EECON2          ;
                bsf     EECON1, 1       ; starts the write operation
EELoop          btfsc   EECON1, 1       ; has write operation finished?
                goto    EELoop          ; no, still high, so keeps looping
```

```
                movfw   STATUS_temp     ; restores STATUS register to
                movwf   STATUS          ;  original value
                swapf   W_temp, f       ; restores working register to
                swapf   W_temp, w       ;  original value
                retfie                  ; returns, enabling GIE

CardValue       addwf   PCL, f          ; returns with new number of
                retlw   d'2'            ;  minutes for the card
                retlw   d'5'            ;
                retlw   d'10'           ;
                retlw   d'20'           ;
                retlw   d'40'           ;
                retlw   d'60'           ; one hour
                retlw   d'120'          ; two hours
                retlw   0               ; (erases card)

;===========
; Program Start

Start           call    Init            ; initialisation routine

Main            bsf     STATUS, RP0     ; selects Bank 1
                clrf    EEADR           ; selects EEPROM address 00h
                bsf     EECON1, 0       ; initiates an EEPROM read
                movfw   EEDATA          ; reads EEDATA
                bcf     STATUS, RP0     ; selects Bank 0
                btfss   STATUS, Z       ; is it 0?
                goto    Active          ; no, so goes to Active
                bcf     GPIO, 0         ; turns off GP0
                sleep                   ; goes to sleep
                nop                     ;
                goto    Main            ; loops back to Main

Active          bsf     GPIO, 0         ; turns on GP0
                btfss   GPIO, 1         ; is a call in progress?
                goto    Active          ; no, so keeps waiting
                movfw   Mark125         ; has one minute passed?
                subwf   TMR0, w         ;
                btfss   STATUS, Z       ;
                goto    Active          ; no, so keeps looping

                movlw   d'125'          ;
                addwf   Mark125         ;
                decfsz  Post125         ;
                goto    Active          ;

                movlw   d'125'          ;
                movwf   Post125         ;
```

```
                decfsz  Post15          ;
                goto    Active          ;

                movlw   d'15'           ; one minute has passed, so
                movwf   Post15          ;   resets final postscaler
                bsf     STATUS, RP0     ; goes to Bank 1
                clrf    EEADR           ; selects EEPROM address 00h
                bsf     EECON1, 0       ; reads EEPROM address 00h
                decf    EEDATA          ; subtracts 1 minute from card
                bsf     EECON1, 2       ; enables a write operation
                bcf     INTCON, 7       ; disables global interrupts
                movlw   0x55            ; now follows the 'safe
                movwf   EECON2          ;   combination'
                movlw   0xAA            ;
                movwf   EECON2          ;
                bsf     EECON1, 1       ; starts the write operation
EELoop          btfsc   EECON1, 1       ; has write operation finished?
                goto    EELoop          ; no, still high, so keeps looping

                bcf     STATUS, RP0     ; back to Bank 0
                bsf     INTCON, 7       ; enables global interrupts
                goto    Main            ; loops back to start

                END
```

APPENDIX G

Program P

```
;**************************************
; written by: John Morton              *
; date: 14/03/05                       *
; version: 1.0                         *
; file saved as: tempsense.asm         *
; for PIC12F675                        *
; clock frequency: Int. 4 MHz          *
; **************************************

; Program Description: Bath temperature measuring device.

       list      P=12F675
       include   "c:\pic\p12f675.inc"

;============
; Declarations:

W_temp  equ      20h
STATUS_temp
       equ       21h

       org       0                 ; first instruction to be executed
       goto      Start             ;

       org       4                 ; interrupt service routine
       goto      isr               ;

;============
; Subroutines:

Init   bsf       STATUS, RP0     ; goes to Bank 1
       call      3FFh            ; calls calibration address
       movwf     OSCCAL          ; moves w. reg into OSCCAL
       movlw     b'010000'       ; GP0-2 are LEDs, GP4 analogue
       movwf     TRISIO          ;   input
       clrf      WPU             ; weak pull-ups disabled

       movlw     b'10000000'     ; weak pull-ups disabled, no timer
       movwf     OPTION_REG      ;   used
```

```
        movlw   b'01000000'     ; enables A/D interrupt
        movwf   PIE1            ;
        clrf    IOC             ; disables GPIO change int.
        clrf    VRCON           ; turns off comparator V. ref.
        movlw   b'00011000'     ; A/D clock: Fosc/8 = 2 µs;
                                ;   AN3/GP4
        movwf   ANSEL           ; is anal. input, others are digital
        bcf     STATUS, RP0     ; back to Bank 0
        clrf    GPIO            ; resets input/output port
        movlw   b'01000000'     ; enables peripheral interrupts
        movwf   INTCON          ;
        movlw   b'00000111'     ; turns off comparator
        movwf   CMCON           ;
        clrf    T1CON           ; turns off TMR1
        movlw   b'00001101'     ; turns on ADC, selects AN3,
        movwf   ADCON0          ;   relative to VDD, left-justified

        retfie                  ;

isr     movwf   W_temp          ; stores w. reg in temp register
        movfw   STATUS          ; stores STATUS in temporary
        movwf   STATUS_temp     ;   register

        bcf     STATUS, RP0     ; goes to Bank 0
        bcf     PIR1, 6         ; clears A/D interrupt flag

        bsf     STATUS, RP0     ; goes to Bank 1
        movlw   0x80            ; subtracts lower byte
        subwf   ADRESL, w       ;
        comf    STATUS, w       ; inverts carry flag (bit 0 of STATUS)
        andlw   b'00000001'     ; masks all other bits
        bcf     STATUS, RP0     ; goes to Bank 0
        addlw   0x12            ; add this to the number we are
        subwf   ADRESH, w       ;   subtracting from the higher byte
        btfss   STATUS, C       ;
        goto    Cold            ; ADRESH:L < 0x1280, so "cold!"

        bsf     STATUS, RP0     ; goes to Bank 1
        movlw   0x80            ; subtracts lower byte
        subwf   ADRESL, w       ;
        comf    STATUS, w       ; inverts carry flag (bi 0 of STATUS)
        andlw   b'00000001'     ; masks all other bits
        bcf     STATUS, RP0     ; goes to Bank 0
        addlw   0x15            ; add this to the number we are
        subwf   ADRESH, w       ;   subtracting from the higher byte
        btfss   STATUS, C       ;
        goto    OK              ; ADRESH:L < 0x1580, so OK
        goto    Hot             ; ADRESH:L = 0x1580,
                                ; so Hot!
```

```
Cold        movlw       b'000001'       ; turns on 'cold' LED
            Movwf       GPIO            ;
            goto        prereturn       ;

OK          movlw       b'000010'       ; turns on 'OK' LED
            movwf       GPIO            ;
            goto        prereturn       ;

Hot         movlw       b'000100'       ; turns on 'Hot' LED
            movwf       GPIO            ;
            goto        prereturn       ;

prereturn   movfw       STATUS_temp     ; restores STATUS register to
            movwf       STATUS          ;  original value
            swapf       W_temp, f       ; restores working register to
            swapf       W_temp, w       ;   original value
            retfie                      ; returns, enabling GIE

;============
; Program Start

Start       call        Init            ; sets everything up
Main        bsf         ADCON0, 1       ; start A/D conversion
            Goto        Main            ;

            END
```

This page intentionally left blank

APPENDIX H

Program Q

```
;*************************************
; written by: John Morton             *
; date: 14/03/05                      *
; version: 1.0                        *
; file saved as: gardenlights.asm     *
; for PIC12F675                       *
; clock frequency: Int. 4 MHz         *
; *************************************

; Program Description: Intelligent garden lights controller.

                list      P=12F675
                include   "c:\pic\P12F675.inc"

                __config  _INTRC_OSC_NOCLKOUT & _WDT_OFF &
                          _PWRTE_ON & _MCLRE_ON & _BODEN_ON &
                          _CP_OFF & _CPD_OFF

Midnight        equ       20
Threshold       equ       21
Mark125         equ       22
Post125         equ       23
Post75          equ       24
FiveMins        equ       25
W_temp          equ       26
STATUS_temp     equ       27

                #define   summer      GPIO, 1

;===============
; Declarations:

                org       0           ; first instruction to be
                                      ;   executed
                goto      Start       ;

                org       4           ; interrupt service routine
                goto      isr         ;
```

```
;==============
; Subroutines:

Init            bsf     STATUS, RP0  ; goes to Bank 1
                movlw   b'001111'    ; GP5: lights, GP4: day/night
                                     ;  LED
                movwf   TRISIO       ; GP3: button, GP2: summer
                                     ;  switch
                                     ; GP0: analogue input
                clrf    WPU          ; no weak pull-ups
                movlw   b'10000111'  ; pull-ups disabled, TMR0
                                     ;  prescaled
                movwf   OPTION_REG   ; by maximum amount (256)

                clrf    PIE1         ; turns off peripheral
                                     ;  interrupts
                clrf    IOC          ; turns off interrupt on
                                     ;  change int.
                clrf    VRCON        ; turns off comparator V. ref.
                movlw   b'00110001'  ; AN0 is only analogue input
                movwf   ANSEL        ;  and analogue clock = RC
                call    3FFh         ; calls calibration address
                movwf   OSCCAL       ; moves w. reg into OSCCAL

                bcf     STATUS, RP0  ; back to Bank 0
                movlw   b'00000111'  ; turns off comparator
                movwf   CMCON        ;
                clrf    T1CON        ; turns off TMR1
                movlw   b'00000001'  ; turns on ADC, input: AN0
                movwf   ADCON0       ;  left justified
                clrf    GPIO         ; lights off, 'day' LED on
                movlw   b'00010000'  ; enables INT interrupt only
                movwf   INTCON       ;

                retfie               ; returns, enabling interrupts

;=================================================================
isr             movwf   W_temp       ; stores w. reg in temp register
                movfw   STATUS       ; stores STATUS in temporary
                movwf   STATUS_temp  ;  register

                bcf     INTCON, 1    ; resets interrupt flag
                movfw   GPIO         ;
                xorlw   b'100000'    ; toggles state of lights
                movwf   GPIO         ;

                movfw   STATUS_temp  ; restores STATUS register to
                movwf   STATUS       ;  original value
```

```
                swapf     W_temp, f   ; restores working register to
                swapf     W_temp, w   ;  original value

                retfie                ;
;=========================================
ADconv          bsf       ADCON0, 1   ; starts AD conversion
                btfsc     ADCON0, 1   ; has it finished?
                goto      ADconv+1    ; no
                return                ;

;=========================================
Delay5min       movfw     TMR0        ; resets timing registers
                addlw     d'125'      ;
                movwf     Mark125     ;
                movlw     d'125'      ; sets up timing registers
                movwf     Post125     ;
                movlw     d'75'       ;
                movwf     Post75      ;

TimeLoop        movfw     Mark125     ; creates a five minute delay
                subwf     TMR0, w     ;
                btfss     STATUS, Z   ;
                goto      TimeLoop    ;
                movlw     d'125'      ;
                addwf     Mark125, f  ;
                decfsz    Post125, f  ;
                goto      TimeLoop    ;
                movlw     d'125'      ;
                ovwf      Post125     ;
                decfsz    Post75, f   ;
                goto      TimeLoop    ;

                return                ; 5 minutes have passed

;=========================================
; Program Start

Start           call      Init        ; initialisation routine

                bsf       STATUS, RP0 ; Bank 1
                btfsc     PCON, 1     ; Power-Up or MCLR reset?
                goto      SetThreshold ; MCLR reset
                bsf       PCON, 1     ; Power-up; resets POR bit

                clrf      EEADR       ;
                bsf       EECON1, 0   ; reads EEPROM address 0
                movfw     EEDATA      ; moves read data into w. reg
                movwf     Midnight    ;
                incf      EEADR       ;
                bsf       EECON1, 0   ; reads EEPROM address 1
```

```
                movfw     EEDATA        ;
                bcf       STATUS, RP0   ; Bank 0
                movwf     Threshold     ;
                goto      Main          ;

SetThreshold
                bcf       STATUS, RP0   ; Bank 0
                call      ADconv        ; perform A/D conversion
                movfw     ADRESH        ; takes 8 most significant bits
                movwf     Threshold     ;

                bsf       STATUS, RP0   ; Bank 1
                movwf     EEDATA        ; stores Threshold in EEPROM
                movlw     1             ; selects EEPROM address 1
                movwf     EEADR         ;
                bsf       EECON1, 2     ; enables a write operation
                bcf       INTCON, 7     ; disables global interrupts

                movlw     0x55          ; now follows the 'safe
                movwf     EECON2        ;  combination'
                movlw     0xAA          ;
                movwf     EECON2        ;
                bsf       EECON1, 1     ; starts the write operation
EELoop          btfsc     EECON1, 1     ; has write operation finished?
                goto      EELoop        ; no, so keeps looping
                bcf       STATUS, RP0   ; Bank 0
                bsf       INTCON, 7     ; re-enables global interrupts

                clrf      Midnight      ; resets Midnight register
                goto      Dusk          ;

;*****************************************************************
Main            call      ADconv        ; this is the standard loop
                movfw     Threshold     ; is it Dusk?
                subwf     ADRESH, w     ;
                btfsc     STATUS, C     ;
                goto      Main          ; no

;====
Dusk            clrf      FiveMins      ; resets timing register
                movlw     b'110000'     ; turns on garden lights and
                movwf     GPIO          ;  'night' LED

Night           call      Delay5min     ; inserts 5 minute delay
                incf      FiveMins      ; counts up no. of 5 minutes

                movlw     d'12'         ; has 1 hour passed?
                subwf     FiveMins, w   ;
                btfss     STATUS, C     ;
                goto      Night         ; no
```

```
            movfw   Midnight      ; is it past midnight?
            subwf   FiveMins, w   ;
            btfss   STATUS, C     ;
            goto    Night         ; no

LightsOff   movlw   b'010000'     ; turns off garden lights and
            movwf   GPIO          ;  keeps 'night' LED on
            call    ADconv        ; performs A/D conversion
            movfw   Threshold     ; is it Dawn?
            subwf   ADRESH, w     ;
            btfss   STATUS, C     ;
            goto    Night         ; no

;====
Dawn        bsf     day           ; turns on 'day' LED
                                  ; determines new midnight
            bcf     STATUS, C     ;
            rrf     FiveMins, w   ; divides time by 2
            btfss   summer        ; are we in summer time?
            sublw   d'24'         ; yes, subtracts 2 hours
            movwf   Midnight      ;
            bsf     STATUS, RP0   ; Bank 1
            movwf   EEDATA        ; stores Midnight in EEDATA
            clrf    EEADR         ; selects EEPROM address 00
            bsf     EECON1, 2     ; enables a write operation
            bcf     INTCON, 7     ; disables global interrupts

            movlw   0x55          ; now follows the 'safe
            movwf   EECON2        ;  combination'
            movlw   0xAA          ;
            movwf   EECON2        ;
            bsf     EECON1, 1     ; starts the write operation
EELoop2     btfsc   EECON1, 1     ; has write operation finished?
            goto    EELoop2       ; no, so keeps looping
            bcf     STATUS, RP0   ; Bank 0
            bsf     INTCON, 7     ; re-enables global interrupts
            clrf    FiveMins      ;

DawnLoop    call    Delay5min     ; one hour delay before looping back
            incf    FiveMins      ;
            movlw   d'12'         ; has one hour passed?
            subwf   FiveMins, w   ;
            btfss   STATUS, C     ;
            goto    DawnLoop      ; no
            goto    Main          ;

            END
```

This page intentionally left blank

APPENDIX I

Useful PIC Data

Specifications of Some Flash PIC Microcontrollers

Device	Pins	I/O	Program Memory	RAM	EEPROM	ADC	Other Features
PIC10F200	6/8*	4	256	16	No	No	Internal 4 MHz oscillator, weak pull-ups, wake-up on change, 2-level stack
PIC10F202	6	4	512	24	No	No	As PIC10F202
PIC10F206	6	4	512	24	No	No	As PIC10F202, with comparator
PIC10F222	6	4	512	24	No	Yes	As PIC10F202
PIC12F508	8	6	512	25	No	No	As PIC10F202
PIC12F509	8	6	1024	41	No	No	As PIC12F508
PIC12F510	8	6	1024	38	No	Yes	As PIC12F508
PIC16F54	18	12	512	25	No	No	2-level stack
PIC16F57	28	20	2048	72	No	No	As PIC16F54
PIC16F59	40	32	2024	134	No	No	As PIC16F54
PIC16F84A	18	13	1024	64	64 bytes	No	8-level stack, interrupts
PIC12F675	8	6	1024	64	128 bytes	Yes	As PIC12F508, 16-bit TMR1, Comparator, 8-level stack, Interrupts
PIC16F676	14	12	1024	64	128 bytes	Yes	As PIC12F675

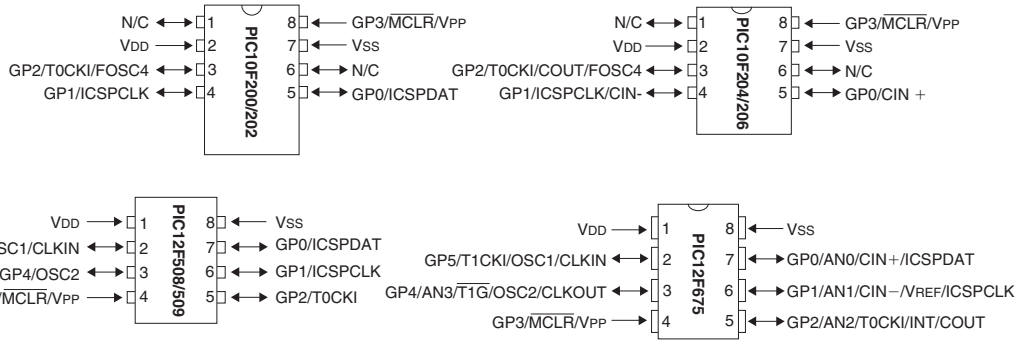
(*Continued*)

Device	Pins	I/O	Program memory	RAM	EEPROM	ADC	Other features
PIC16F627	8	16	1024	224	128 bytes	No	TMR1 (16-bit), TMR2 (8-bit), Comparator, 8-level stack, Interrupts, Capture/ Compare/PWM

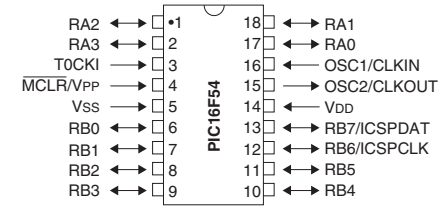
Common features: All the PIC microcontrollers listed here have an 8-bit TMR0, a WDT (Watchdog timer), a DRT (device reset timer), POR (power-on reset), a lower power sleep mode, and support ICSP (In-circuit serial programming).

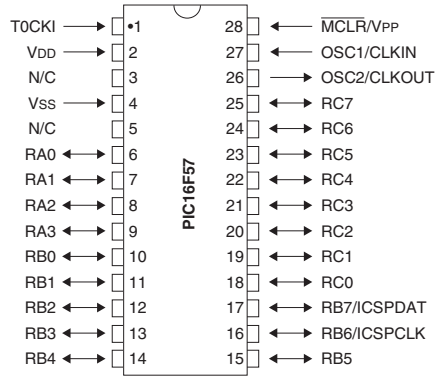
* The P10F2xx series have 6 pins in the surface mount package, but 8 pins in the larger packages (the two extra pins are N/C).

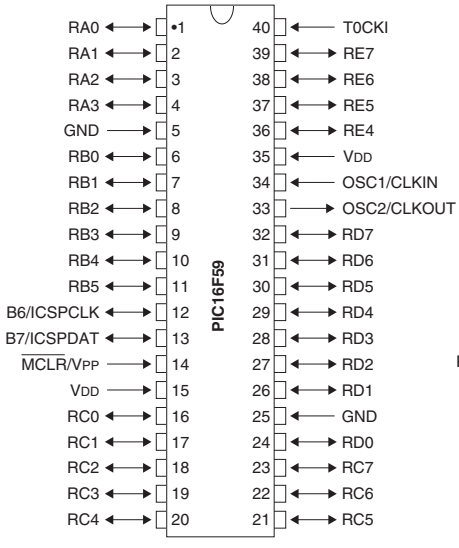
Pin Layouts of Some Flash PIC Microcontrollers

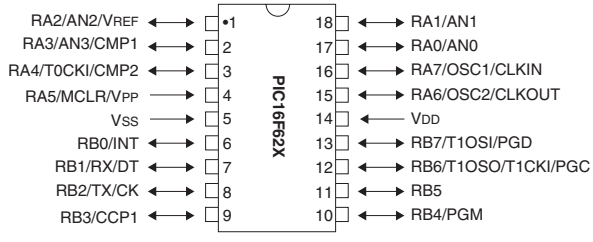


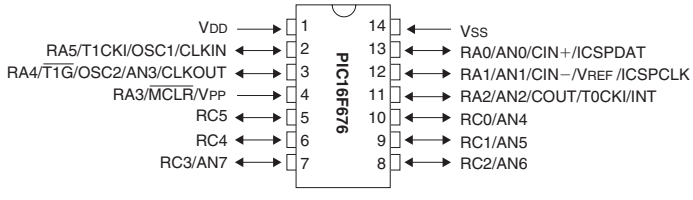
(*Continued*)











Instructions Glossary

addlw number

– (Not for PIC5x series) – **add**s a **number** with the number in the **w**orking register.

addwf FileReg, f

– **add**s the number in the **w**orking register to the number in a **f**ile register and puts the result in the **f**ile register.

addwf FileReg, w

– **add**s the number in the **w**orking register to the number in a **f**ile register and puts the result back into the **w**orking register, leaving the file register unchanged.

andlw number
– **AND**s a **number** with the number in the **w**orking register, leaving the result in the working register.

andwf FileReg, f
– **AND**s the number in the **w**orking register with the number in a **f**ile register and puts the result in the **f**ile register.

bcf FileReg, bit
– **c**lears a **b**it in a **f**ile register (i.e., makes the bit 0).

bsf FileReg, bit
– **s**ets a **b**it in a **f**ile register (i.e., makes the bit 1).

btfsc FileReg, bit
– **t**ests a **b**it in a **f**ile register and **s**kips the next instruction if the result is **c**lear (i.e. if that bit is 0).

btfss FileReg, bit
– **t**ests a **b**it in a **f**ile register and **s**kips the next instruction if the result is **s**et (i.e., if that bit is 1).

call AnySub
– makes the chip **call** a subroutine, after which it will return to where it left off.

clrf FileReg
– **cl**ea**r**s (makes 0) the number in a **f**ile register.

clrw
– **cl**ea**r**s the number in the **w**orking register.

clrwdt
– **cl**ea**r**s the number in the **w**atch**d**og **t**imer.

comf FileReg, f
– **com**plements (inverts, ones become zeroes, zeroes become ones) the number in a **f**ile register, leaving the result in the **f**ile register.

decf FileReg, f
– **dec**rements (subtracts one from) a **f**ile register and puts the result in the **f**ile register.

decfsz FileReg, f
– **dec**rements a **f**ile register and if the result is **z**ero it **s**kips the next instruction. The result is put in the **f**ile register.

goto Anywhere
– makes the chip **go to** somewhere in the program which YOU have labeled “Anywhere”.

incf **FileReg, f**
– **inc**rements (adds one to) a **f**ile register and puts the result in the **f**ile register.

incfsz **FileReg, f**
– **inc**rements a **f**ile register and if the result is **z**ero it **s**kips the next instruction. The result is put in the **f**ile register.

iorlw **number**
– **i**nclusive **OR**s a **number** with the number in the **w**orking register.

iorwf **FileReg, f**
– **i**nclusive **OR**s the number in the **w**orking register with the number in a **f**ile register and puts the result in the **f**ile register.

movfw **FileReg**
or **movf** **FileReg, w**
– **mov**es (copies) the number in a **f**ile register in to the **w**orking register.

movlw **number**
– **mov**es (copies) a **number** into the **w**orking register.

movwf **FileReg**
– **mov**es (copies) the number in the **w**orking register into a **f**ile register.

nop
– this stands for: **n**o **op**eration, in other words – do nothing (it seems useless, but it's actually quite useful!).

option
– (Not to be used except in PIC5x series) – takes the number in the working register and moves it into the **option** register.

retfie
– (Not for PIC5x series) – **ret**urns **f**rom a subroutine and enables the Global **I**nterrupt **E**nable bit.

retlw **number**
– **ret**urns from a subroutine with a particular **number** (**l**iteral) in the **w**orking register.

return
– (Not for PIC5x series) – **return**s from a subroutine.

rlf **FileReg, f**
– **r**otates the bits in a **f**ile register to the **l**eft, putting the result in the **f**ile register.

rrf **FileReg, f**
– **r**otates the bits in a **f**ile register to the **r**ight, putting the result in the **f**ile register.

sleep

– sends the PIC to **sleep**, a lower power consumption mode.

sublw **number**

– (Not for PIC5x series) – **sub**tracts the number in the **w**orking register from a **number**.

subwf **FileReg, f**

– **sub**tracts the number in the **w**orking register from the number in a **f**ile register and puts the result in the **f**ile register.

swapf **FileReg, f**

– **swap**s the two halves of the 8 bit binary number in a **f**ile register, leaving the result in the **f**ile register.

tris **PORTX**

– (Not to be used except in PIC16C5x series) – uses the number in the working register to specify which bits of a port are inputs (correspond to a binary 1) and which are outputs (correspond to 0).

xorlw **number**

– e**x**clusive **OR**s a **number** with the number in the **w**orking register.

xorwf **FileReg, f**

– e**x**clusive **OR**s the number in the **w**orking register with the number in a **f**ile register and puts the result in the **f**ile register.

Number System Conversion

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Bit Assignments of Various File Registers

OPTION_REG

Bit no.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Bit name	**…**	**…**	**T0CS**	**T0SE**	**PSA**	**PS2**	**PS1**	**PS0**

Prescaler value…			**TMR0**	**WDT**
0	0	0	1:2	1:1
0	0	1	1:4	1:2
0	1	0	1:8	1:4
0	1	1	1:16	1:8
1	0	0	1:32	1:16
1	0	1	1:64	1:32
1	1	0	1:128	1:64
1	1	1	1:256	1:128

Prescaler Assignment
0: Prescaler assigned to TMR0
1: Prescaler assigned to WDT

TMR0 Source Edge Select
0: TMR0 counts on falling edge
1: TMR0 counts on rising edge

TMR0 Clock Source Select
0: TMR0 counts signal from oscillator
1: TMR0 counts signals on T0CKI pin

(PIC16F5X) **Unassigned**

(PIC12F5xx) **Weak Pull-ups Enable**
0: Enabled
1: Disabled

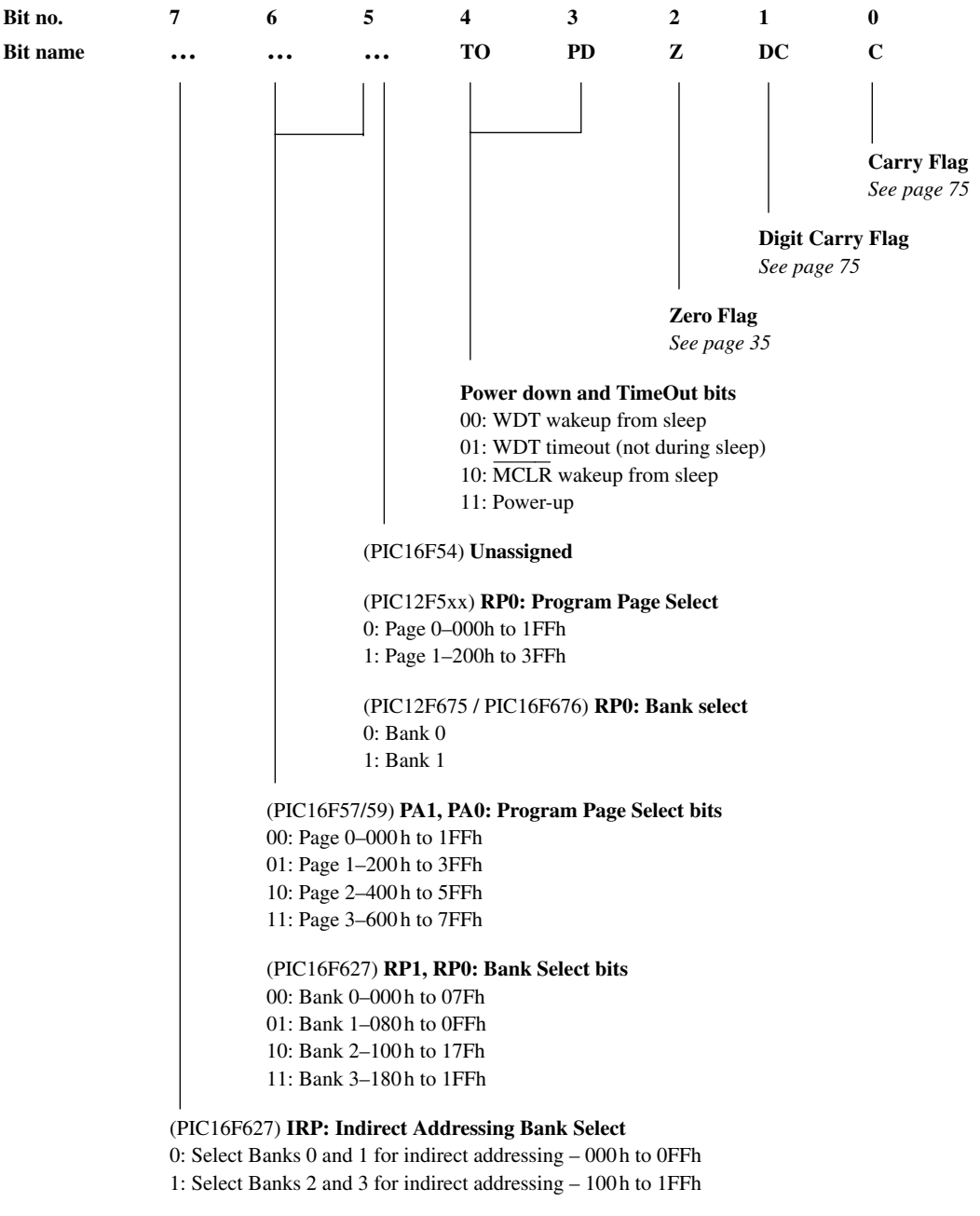
(PIC12F675 / PIC16F676 / F627) **Ext. Interrupt Edge Select**
0: Interrupt on falling edge of INT pin
1: Interrupt on rising edge of INT pin

(PIC16F5X) **Unassigned**

(PIC12F5xx) **Wake-up on Change Enable**
0: Enabled
1: Disabled

(PIC12F675 / PIC16F676 / PIC16F627) **Port B Pull-up Enable**
0: Weak pull-ups enabled on GPIO/Port B, if selected in WPU
1: Weak pull-ups disabled

STATUS


Bit no. 7 6 5 4 3 2 1 0
Bit name ... ... TO PD Z DC C
Carry Flag
See page 75
Digit Carry Flag
See page 75
Zero Flag
See page 35
Power down and TimeOut bits
00: WDT wakeup from sleep
01: WDT timeout (not during sleep)
10: MCLR wakeup from sleep
11: Power-up
(PIC16F54) Unassigned
(PIC12F5xx) RP0: Program Page Select
0: Page 0–000h to 1FFh
1: Page 1–200h to 3FFh
(PIC12F675 / PIC16F676) RP0: Bank select
0: Bank 0
1: Bank 1
(PIC16F57/59) PA1, PA0: Program Page Select bits
00: Page 0–000h to 1FFh
01: Page 1–200h to 3FFh
10: Page 2–400h to 5FFh
11: Page 3–600h to 7FFh
(PIC16F627) RP1, RP0: Bank Select bits
00: Bank 0–000h to 07Fh
01: Bank 1–080h to 0FFh
10: Bank 2–100h to 17Fh
11: Bank 3–180h to 1FFh
(PIC16F627) IRP: Indirect Addressing Bank Select
0: Select Banks 0 and 1 for indirect addressing – 000h to 0FFh
1: Select Banks 2 and 3 for indirect addressing – 100h to 1FFh

INTCON

Bit no.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Bit name	**GIE**	**PEIE**	**T0IE**	**INTE**	**GPIE**	**T0IF**	**INTF**	**GPIF**

Port Change flag
0: It hasn't [**Note:** Must be cleared by you]
1: GPIO/Port B change int. occurred

External INT flag
0: It hasn't
[**Note:** Must be cleared by you]
1: An interrupt has occurred on the INT pin

TMR0 Overflow Interrupt flag
0: TMR0 has not overflowed
[**Note:** Must be cleared by you]
1: TMR0 has overflowed

Port Change Interrupt Enable
0: Disables GPIO/Port B change interrupt
1: Enables it

External INT Interrupt Enable
0: Disables the INT pin interrupt
1: Enables it

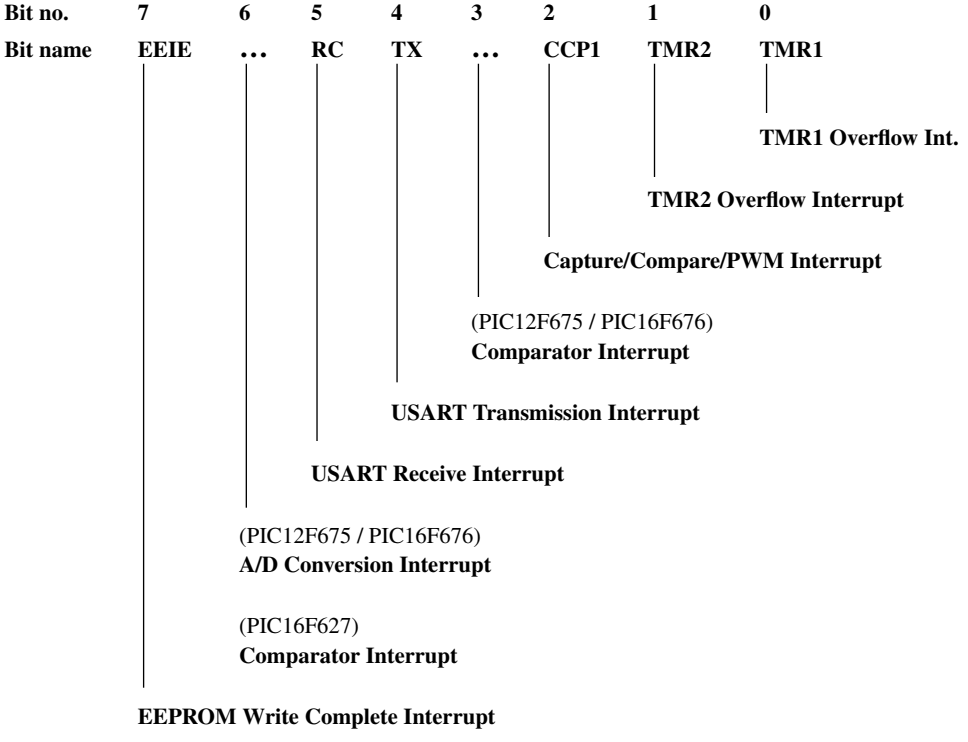
TMR0 Overflow Interrupt Enable
0: Disables TMR0 overflow interrupt
1: Enables it

Peripheral Interrupt Enable
0: Disables any enabled 'peripheral interrupts'
1: Enables all 'peripheral interrupts'

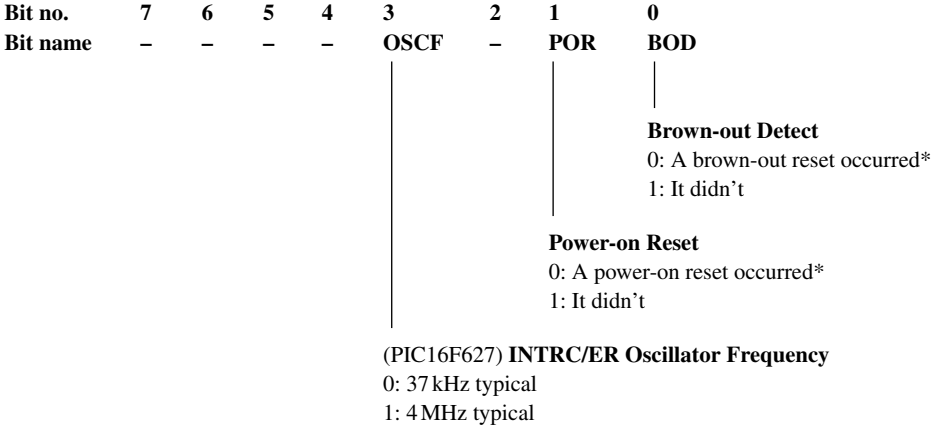
Global Interrupt Enable
0: Disables ALL interrupts
1: Enables any enabled interrupts

PIE1/PIR1

The bit assignments in the peripheral interrupt registers PIE1 and PIR1 are identical. In PIE1 they refer to interrupt enable bits, and in PIR1 they refer to interrupt flags. The interrupt flags must be cleared by you in the interrupt service routine.



PCON



* Both these bits must be set in software, when cleared by the relevant reset.

EECON1

Bit no.	**7, 6, 5, 4**	**3**	**2**	**1**	**0**
Bit name	*unused*	**WRERR**	**WREN**	**WR**	**RD**

Read Control Bit
1: Starts an EEPROM read (gets cleared when read finishes)

Write Control Bit
1: Starts an EEPROM write operation (gets cleared when write finishes)

EEPROM Write Enable Bit
0: Forbids writing to the EEPROM
1: Permits writing to the EEPROM

EEPROM Write Error Flag
0: The write operation completed without error
1: An EEPROM write has prematurely terminated

VRCON

Bit 7: Comparator Voltage Reference Enable bit
0: Voltage reference module off (consuming no current)
1: Voltage reference module on

VRCON, 5 = 1 (Low Range)		VRCON, 5 = 0 (High Range)	
VRCON 3:0	**VRef (V_{DD} = 5 V)**	**VRCON 3:0**	**VRef (V_{DD} = 5 V)**
0000	0.00	0000	1.25
0001	0.21	0001	1.41
0010	0.42	0010	1.56
0011	0.63	0011	1.72
0100	0.83	0100	1.88
0101	1.04	0101	2.03
0110	1.25	0110	2.19
0111	1.46	0111	2.34
1000	1.67	1000	2.50
1001	1.88	1001	2.66
1010	2.08	1010	2.81
1011	2.29	1011	2.97
1100	2.50	1100	3.13
1101	2.71	1101	3.28
1110	2.92	1110	3.44
1111	3.13	1111	3.59

CMCON

Bit no.	7	6	5	4	3	2:0
Bit name	–	COUT	–	CINV	CIS	CM

Comparator Model bits
See table below

Comparator Input Select
(In type '**C**' mode)
0: V_{IN-} connects to GP1/C_{IN-}
1: V_{IN-} connects to GP0/C_{IN+}

Comparator Output Inversion bit
0: Doesn't invert comparator output
1: Inverts comparator output

Comparator Output
If CINV = 0…
0: $V_{IN+} > V_{IN-}$
1: $V_{IN+} < V_{IN-}$
(inverted if CINV = 1)

CMCON 2:0		V_{IN+}	V_{IN-}	V_{OUT}
000		GP0/C_{IN+}	GP1/C_{IN-}	*Disabled: CMCON, 6* = 0
001	**A**	GP0/C_{IN+}	GP1/C_{IN-}	GP2/C_{OUT} *and* CMCON, 6
010		GP0/C_{IN+}	GP1/C_{IN-}	CMCON, 6
011	**B**	Internal ref.	GP1/C_{IN-}	GP2/C_{OUT} *and* CMCON, 6
100		Internal ref.	GP1/C_{IN-}	CMCON, 6
101	**C**	Internal ref.	GP0 *or* GP1	GP2/C_{OUT} *and* CMCON, 6
110		Internal ref.	GP0 *or* GP1	CMCON, 6
111		*Comparator off and consumes no current (CMCON, 6* = 0)		

ADCON0

Bit no.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Bit name	**ADFM**	**VCFG**	**–**	**–**	**CHS1**	**CHS0**	**GO/DONE**	**ADON**

A/D on bit
1: ADC is on
0: ADC is off

GO/DONE
1: Starts A/D conversion. Stays high until finished
0: A/D conversion finished

Channel select bits
00: Channel 00 (AN0)
01: Channel 01 (AN1)
10: Channel 02 (AN2)
11: Channel 03 (AN3)

Voltage reference bit
1: Measures relative to V_{REF} pin
0: Measures relative to V_{DD} (supply voltage)

A/D result formed select
1: Right justified – result stored in ADRESL and ADRESH (bits 0:2)
0: Left justified – result stored in ADRESL (bits 6:7) and ADRESH

ANSEL

ANSEL bits 6:4	A/D Conversion Clock	Device Frequency			
		1.25 MHz	2.46 MHz	4 MHz	20 MHz
000	Fosc/2	1.6 μs	800 ns	500 ns	100 ns
001	Fosc/8	6.4 μs	3.2 μs	2 μs	400 ns
010	Fosc/32	25.6 μs	12.8 μs	8 μs	1.6 μs
011	Frc: Internal oscillator	~4 μs	~4 μs	~4 μs	~4 μs
100	Fosc/4	3.2 μs	1.6 μs	1 μs	200 ns
101	Fosc/16	12.8 μs	6.4 μs	4 μs	800 ns
110	Fosc/64	51.2 μs	25.6 μs	16 μs	3.2 μs
111	Frc: Internal oscillator	~4 μs	~4 μs	~4 μs	~4 μs

ANSEL Bits 3:0 correspond to the four A/D input channels AN3:0.
0: Makes the pin a digital I/O pin
1: Makes the pin an analog input, disabling weak pull-ups, interrupt-on-change, etc.

If all Else Fails, Read This

You should find that there are certain mistakes which you make time and time again. Here is a list of the popular ones:

1. Look for: **subwf FileReg** … are you sure you don't mean …
   subwf FileReg, w
2. Have you remembered the correct addresses for general purpose file registers for your particular PIC model? (e.g. on the PIC12F675 they don't start until address **20h**).
3. *You are using a PIC microcontroller which has weak pull-ups*—have you remembered to set up bit 7 of the OPTION register correctly?
4. Are your subroutines in the correct page or half of page?
5. Are you adding something to the program counter in the wrong place on a page or on the wrong page?
6. Are you remembering to reset a file register you are using to keep track of how many times something has happened (e.g. a postscaler)?
7. *You think you are doing something to a file register but it isn't happening* … are you in the correct bank?
8. *If you are having a total nightmare and NOTHING is working* … have you specified the correct PIC microcontroller at the top of the program?
9. Have you set the configuration bits correctly when programming/simulating?

APPENDIX J

PIC 16F84A Data Sheet

PIC16F84A
Data Sheet

18-pin Enhanced FLASH/EEPROM
8-bit Microcontroller

Note the following details of the code protection feature on PICmicro® MCUs.

- The PICmicro family meets the specifications contained in the Microchip Data Sheet.
- Microchip believes that its family of PICmicro microcontrollers is one of the most secure products of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the PICmicro microcontroller in a manner outside the operating specifications contained in the data sheet. The person doing so may be engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable".
- Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our product.

If you have any further questions about this matter, please contact the local sales office nearest to you.

DNV Certification, Inc. USA — ANSI-RAB QMS ACCREDITED — DNV MSC The Netherlands Accredited by the RvA — DNV — ISO 9001 / QS-9000 REGISTERED FIRM

Microchip received QS-9000 quality system certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona in July 1999. The Company's quality system processes and procedures are QS-9000 compliant for its PICmicro® 8-bit MCUs, KEELOQ® code hopping devices, Serial EEPROMs and microperipheral products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001 certified.

MICROCHIP

PIC16F84A

18-pin *Enhanced* FLASH/EEPROM 8-Bit Microcontroller

High Performance RISC CPU Features:

- Only 35 single word instructions to learn
- All instructions single-cycle except for program branches which are two-cycle
- Operating speed: DC - 20 MHz clock input
  DC - 200 ns instruction cycle
- 1024 words of program memory
- 68 bytes of Data RAM
- 64 bytes of Data EEPROM
- 14-bit wide instruction words
- 8-bit wide data bytes
- 15 Special Function Hardware registers
- Eight-level deep hardware stack
- Direct, indirect and relative addressing modes
- Four interrupt sources:
  - External RB0/INT pin
  - TMR0 timer overflow
  - PORTB<7:4> interrupt-on-change
  - Data EEPROM write complete

Peripheral Features:

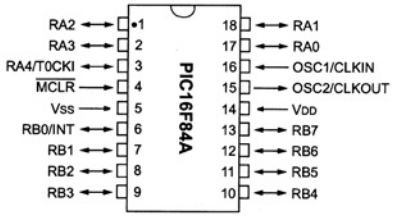
- 13 I/O pins with individual direction control
- High current sink/source for direct LED drive
  - 25 mA sink max. per pin
  - 25 mA source max. per pin
- TMR0: 8-bit timer/counter with 8-bit programmable prescaler

Special Microcontroller Features:

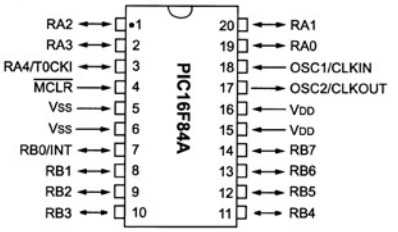
- 10,000 erase/write cycles *Enhanced* FLASH Program memory typical
- 10,000,000 typical erase/write cycles EEPROM Data memory typical
- EEPROM Data Retention > 40 years
- In-Circuit Serial Programming™ (ICSP™) - via two pins
- Power-on Reset (POR), Power-up Timer (PWRT), Oscillator Start-up Timer (OST)
- Watchdog Timer (WDT) with its own On-Chip RC Oscillator for reliable operation
- Code protection
- Power saving SLEEP mode
- Selectable oscillator options

Pin Diagrams

PDIP, SOIC



SSOP



CMOS Enhanced FLASH/EEPROM Technology:

- Low power, high speed technology
- Fully static design
- Wide operating voltage range:
  - Commercial: 2.0V to 5.5V
  - Industrial: 2.0V to 5.5V
- Low power consumption:
  - < 2 mA typical @ 5V, 4 MHz
  - 15 µA typical @ 2V, 32 kHz
  - < 0.5 µA typical standby current @ 2V

PIC16F84A

Table of Contents

TO OUR VALUED CUSTOMERS

It is our intention to provide our valued customers with the best documentation possible to ensure successful use of your Microchip products. To this end, we will continue to improve our publications to better suit your needs. Our publications will be refined and enhanced as new volumes and updates are introduced.

If you have any questions or comments regarding this publication, please contact the Marketing Communications Department via E-mail at **docerrors@mail.microchip.com** or fax the **Reader Response Form** in the back of this data sheet to (480) 792-4150. We welcome your feedback.

Most Current Data Sheet

To obtain the most up-to-date version of this data sheet, please register at our Worldwide Web site at:

http://www.microchip.com

You can determine the version of a data sheet by examining its literature number found on the bottom outside corner of any page. The last character of the literature number is the version number, (e.g., DS30000A is version A of document DS30000).

Errata

An errata sheet, describing minor operational differences from the data sheet and recommended workarounds, may exist for current devices. As device/documentation issues become known to us, we will publish an errata sheet. The errata will specify the revision of silicon and revision of document to which it applies.

To determine if an errata sheet exists for a particular device, please check with one of the following:

- Microchip's Worldwide Web site; http://www.microchip.com
- Your local Microchip sales office (see last page)
- The Microchip Corporate Literature Center; U.S. FAX: (480) 792-7277

When contacting a sales office or the literature center, please specify which device, revision of silicon and data sheet (include literature number) you are using.

Customer Notification System

Register on our web site at **www.microchip.com/cn** to receive the most current information on all of our products.

PIC16F84A

1.0 DEVICE OVERVIEW

This document contains device specific information for the operation of the PIC16F84A device. Additional information may be found in the PICmicro™ Mid-Range Reference Manual, (DS33023), which may be downloaded from the Microchip website. The Reference Manual should be considered a complementary document to this data sheet, and is highly recommended reading for a better understanding of the device architecture and operation of the peripheral modules.

The PIC16F84A belongs to the mid-range family of the PICmicro® microcontroller devices. A block diagram of the device is shown in Figure 1-1.

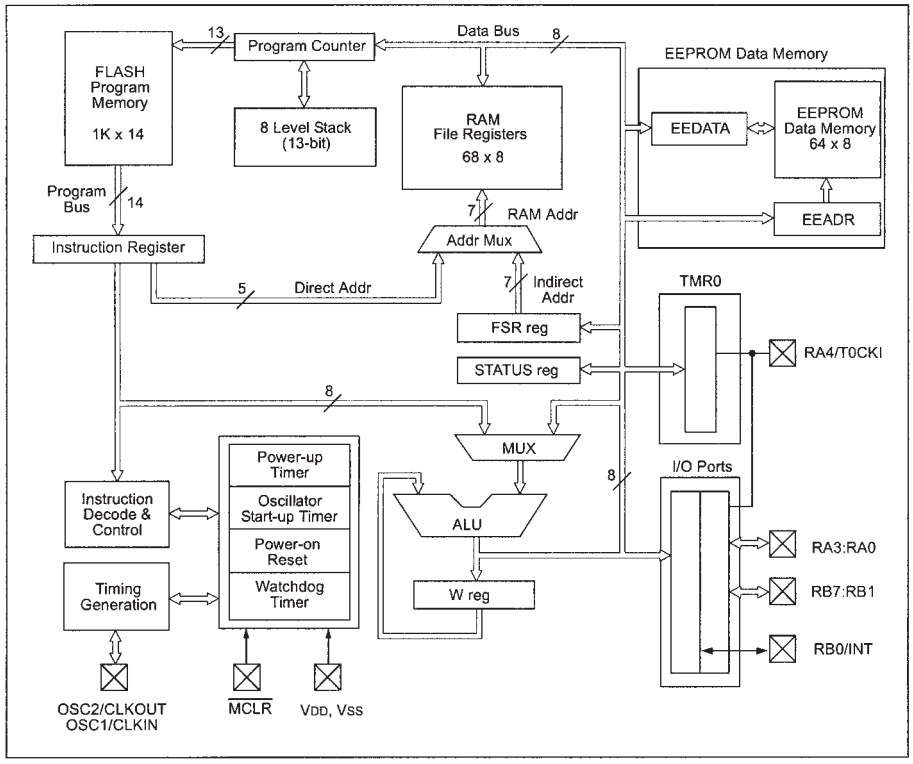
The program memory contains 1K words, which translates to 1024 instructions, since each 14-bit program memory word is the same width as each device instruction. The data memory (RAM) contains 68 bytes. Data EEPROM is 64 bytes.

There are also 13 I/O pins that are user-configured on a pin-to-pin basis. Some pins are multiplexed with other device functions. These functions include:

- External interrupt
- Change on PORTB interrupt
- Timer0 clock input

Table 1-1 details the pinout of the device with descriptions and details for each pin.

FIGURE 1-1: PIC16F84A BLOCK DIAGRAM



PIC16F84A

TABLE 1-1: PIC16F84A PINOUT DESCRIPTION

Pin Name	PDIP No.	SOIC No.	SSOP No.	I/O/P Type	Buffer Type	Description
OSC1/CLKIN	16	16	18	I	ST/CMOS(3)	Oscillator crystal input/external clock source input.
OSC2/CLKOUT	15	15	19	O	—	Oscillator crystal output. Connects to crystal or resonator in Crystal Oscillator mode. In RC mode, OSC2 pin outputs CLKOUT, which has 1/4 the frequency of OSC1 and denotes the instruction cycle rate.
$\overline{MCLR}$	4	4	4	I/P	ST	Master Clear (Reset) input/programming voltage input. This pin is an active low RESET to the device.
						PORTA is a bi-directional I/O port.
RA0	17	17	19	I/O	TTL	
RA1	18	18	20	I/O	TTL	
RA2	1	1	1	I/O	TTL	
RA3	2	2	2	I/O	TTL	
RA4/T0CKI	3	3	3	I/O	ST	Can also be selected to be the clock input to the TMR0 timer/counter. Output is open drain type.
						PORTB is a bi-directional I/O port. PORTB can be software programmed for internal weak pull-up on all inputs.
RB0/INT	6	6	7	I/O	TTL/ST(1)	RB0/INT can also be selected as an external interrupt pin.
RB1	7	7	8	I/O	TTL	
RB2	8	8	9	I/O	TTL	
RB3	9	9	10	I/O	TTL	
RB4	10	10	11	I/O	TTL	Interrupt-on-change pin.
RB5	11	11	12	I/O	TTL	Interrupt-on-change pin.
RB6	12	12	13	I/O	TTL/ST(2)	Interrupt-on-change pin. Serial programming clock.
RB7	13	13	14	I/O	TTL/ST(2)	Interrupt-on-change pin. Serial programming data.
VSS	5	5	5,6	P	—	Ground reference for logic and I/O pins.
VDD	14	14	15,16	P	—	Positive supply for logic and I/O pins.

Legend: I= input O = Output I/O = Input/Output P = Power
— = Not used TTL = TTL input ST = Schmitt Trigger input

Note 1: This buffer is a Schmitt Trigger input when configured as the external interrupt.
2: This buffer is a Schmitt Trigger input when used in Serial Programming mode.
3: This buffer is a Schmitt Trigger input when configured in RC oscillator mode and a CMOS input otherwise.

PIC16F84A

2.0 MEMORY ORGANIZATION

There are two memory blocks in the PIC16F84A. These are the program memory and the data memory. Each block has its own bus, so that access to each block can occur during the same oscillator cycle.

The data memory can further be broken down into the general purpose RAM and the Special Function Registers (SFRs). The operation of the SFRs that control the "core" are described here. The SFRs used to control the peripheral modules are described in the section discussing each individual peripheral module.

The data memory area also contains the data EEPROM memory. This memory is not directly mapped into the data memory, but is indirectly mapped. That is, an indirect address pointer specifies the address of the data EEPROM memory to read/write. The 64 bytes of data EEPROM memory have the address range 0h-3Fh. More details on the EEPROM memory can be found in Section 3.0.

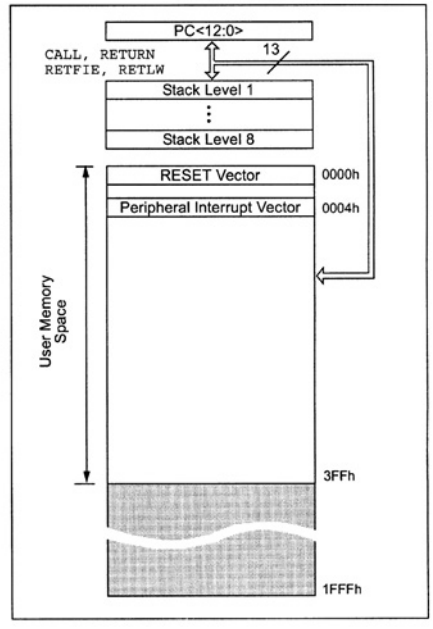
Additional information on device memory may be found in the PICmicro™ Mid-Range Reference Manual, (DS33023).

2.1 Program Memory Organization

The PIC16FXX has a 13-bit program counter capable of addressing an 8K x 14 program memory space. For the PIC16F84A, the first 1K x 14 (0000h-03FFh) are physically implemented (Figure 2-1). Accessing a location above the physically implemented address will cause a wraparound. For example, for locations 20h, 420h, 820h, C20h, 1020h, 1420h, 1820h, and 1C20h, the instruction will be the same.

The RESET vector is at 0000h and the interrupt vector is at 0004h.

FIGURE 2-1: PROGRAM MEMORY MAP AND STACK - PIC16F84A



PIC16F84A

2.2 Data Memory Organization

The data memory is partitioned into two areas. The first is the Special Function Registers (SFR) area, while the second is the General Purpose Registers (GPR) area. The SFRs control the operation of the device.

Portions of data memory are banked. This is for both the SFR area and the GPR area. The GPR area is banked to allow greater than 116 bytes of general purpose RAM. The banked areas of the SFR are for the registers that control the peripheral functions. Banking requires the use of control bits for bank selection. These control bits are located in the STATUS Register. Figure 2-2 shows the data memory map organization.

Instructions `MOVWF` and `MOVF` can move values from the W register to any location in the register file ("F"), and vice-versa.

The entire data memory can be accessed either directly using the absolute address of each register file or indirectly through the File Select Register (FSR) (Section 2.5). Indirect addressing uses the present value of the RP0 bit for access into the banked areas of data memory.

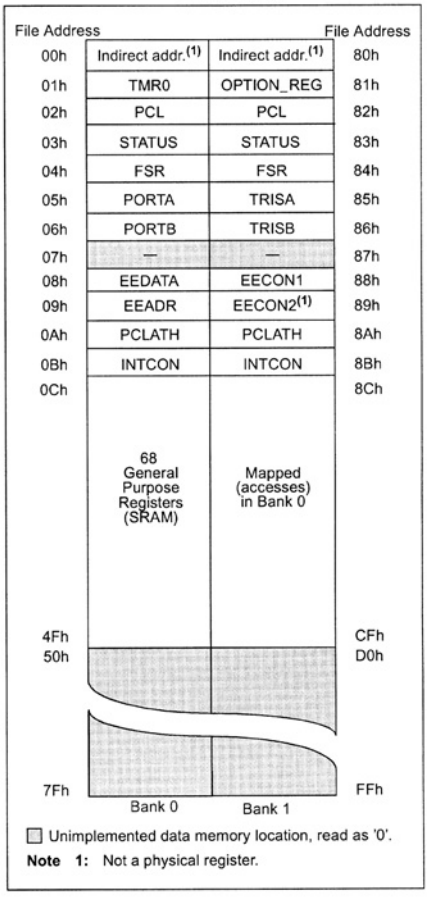
Data memory is partitioned into two banks which contain the general purpose registers and the special function registers. Bank 0 is selected by clearing the RP0 bit (STATUS<5>). Setting the RP0 bit selects Bank 1. Each Bank extends up to 7Fh (128 bytes). The first twelve locations of each Bank are reserved for the Special Function Registers. The remainder are General Purpose Registers, implemented as static RAM.

2.2.1 GENERAL PURPOSE REGISTER FILE

Each General Purpose Register (GPR) is 8-bits wide and is accessed either directly or indirectly through the FSR (Section 2.5).

The GPR addresses in Bank 1 are mapped to addresses in Bank 0. As an example, addressing location 0Ch or 8Ch will access the same GPR.

FIGURE 2-2: REGISTER FILE MAP - PIC16F84A



2.3 Special Function Registers

The Special Function Registers (Figure 2-2 and Table 2-1) are used by the CPU and Peripheral functions to control the device operation. These registers are static RAM.

The special function registers can be classified into two sets, core and peripheral. Those associated with the core functions are described in this section. Those related to the operation of the peripheral features are described in the section for that specific feature.

TABLE 2-1: SPECIAL FUNCTION REGISTER FILE SUMMARY

Addr	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on Power-on RESET	Details on page
Bank 0											
00h	INDF	Uses contents of FSR to address Data Memory (not a physical register)								---- ----	11
01h	TMR0	8-bit Real-Time Clock/Counter								xxxx xxxx	20
02h	PCL	Low Order 8 bits of the Program Counter (PC)								0000 0000	11
03h	STATUS(2)	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	8
04h	FSR	Indirect Data Memory Address Pointer 0								xxxx xxxx	11
05h	PORTA(4)	—	—	—	RA4/T0CKI	RA3	RA2	RA1	RA0	---x xxxx	16
06h	PORTB(5)	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0/INT	xxxx xxxx	18
07h	—	Unimplemented location, read as '0'								—	—
08h	EEDATA	EEPROM Data Register								xxxx xxxx	13,14
09h	EEADR	EEPROM Address Register								xxxx xxxx	13,14
0Ah	PCLATH	—	—	—	Write Buffer for upper 5 bits of the PC(1)					---0 0000	11
0Bh	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	10
Bank 1											
80h	INDF	Uses Contents of FSR to address Data Memory (not a physical register)								---- ----	11
81h	OPTION_REG	RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	9
82h	PCL	Low order 8 bits of Program Counter (PC)								0000 0000	11
83h	STATUS (2)	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	8
84h	FSR	Indirect data memory address pointer 0								xxxx xxxx	11
85h	TRISA	—	—	—	PORTA Data Direction Register					---1 1111	16
86h	TRISB	PORTB Data Direction Register								1111 1111	18
87h	—	Unimplemented location, read as '0'								—	—
88h	EECON1	—	—	—	EEIF	WRERR	WREN	WR	RD	---0 x000	13
89h	EECON2	EEPROM Control Register 2 (not a physical register)								---- ----	14
0Ah	PCLATH	—	—	—	Write buffer for upper 5 bits of the PC(1)					---0 0000	11
0Bh	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	10

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0', q = value depends on condition

Note 1: The upper byte of the program counter is not directly accessible. PCLATH is a slave register for PC<12:8>. The contents of PCLATH can be transferred to the upper byte of the program counter, but the contents of PC<12:8> are never transferred to PCLATH.

2: The $\overline{TO}$ and $\overline{PD}$ status bits in the STATUS register are not affected by a $\overline{MCLR}$ Reset.

3: Other (non power-up) RESETS include: external RESET through $\overline{MCLR}$ and the Watchdog Timer Reset.

4: On any device RESET, these pins are configured as inputs.

5: This is the value that will be in the port output latch.

PIC16F84A

2.3.1 STATUS REGISTER

The STATUS register contains the arithmetic status of the ALU, the RESET status and the bank select bit for data memory.

As with any register, the STATUS register can be the destination for any instruction. If the STATUS register is the destination for an instruction that affects the Z, DC or C bits, then the write to these three bits is disabled. These bits are set or cleared according to device logic. Furthermore, the $\overline{TO}$ and $\overline{PD}$ bits are not writable. Therefore, the result of an instruction with the STATUS register as destination may be different than intended.

For example, `CLRF STATUS` will clear the upper three bits and set the Z bit. This leaves the STATUS register as `000u uluu` (where `u` = unchanged).

Only the `BCF`, `BSF`, `SWAPF` and `MOVWF` instructions should be used to alter the STATUS register (Table 7-2), because these instructions do not affect any status bit.

Note 1: The IRP and RP1 bits (STATUS<7:6>) are not used by the PIC16F84A and should be programmed as cleared. Use of these bits as general purpose R/W bits is NOT recommended, since this may affect upward compatibility with future products.

2: The C and DC bits operate as a $\overline{borrow}$ and digit $\overline{borrow}$ out bit, respectively, in subtraction. See the `SUBLW` and `SUBWF` instructions for examples.

3: When the STATUS register is the destination for an instruction that affects the Z, DC or C bits, then the write to these three bits is disabled. The specified bit(s) will be updated according to device logic

REGISTER 2-1: STATUS REGISTER (ADDRESS 03h, 83h)

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C
bit 7							bit 0

bit 7-6 **Unimplemented:** Maintain as '0'

bit 5 **RP0**: Register Bank Select bits (used for direct addressing)
01 = Bank 1 (80h - FFh)
00 = Bank 0 (00h - 7Fh)

bit 4 $\overline{TO}$: Time-out bit
1 = After power-up, `CLRWDT` instruction, or `SLEEP` instruction
0 = A WDT time-out occurred

bit 3 $\overline{PD}$: Power-down bit
1 = After power-up or by the `CLRWDT` instruction
0 = By execution of the `SLEEP` instruction

bit 2 **Z**: Zero bit
1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero

bit 1 **DC**: Digit carry/$\overline{borrow}$ bit (`ADDWF`, `ADDLW`, `SUBLW`, `SUBWF` instructions) (for $\overline{borrow}$, the polarity is reversed)
1 = A carry-out from the 4th low order bit of the result occurred
0 = No carry-out from the 4th low order bit of the result

bit 0 **C**: Carry/$\overline{borrow}$ bit (`ADDWF`, `ADDLW`, `SUBLW`, `SUBWF` instructions) (for $\overline{borrow}$, the polarity is reversed)
1 = A carry-out from the Most Significant bit of the result occurred
0 = No carry-out from the Most Significant bit of the result occurred

Note: A subtraction is executed by adding the two's complement of the second operand. For rotate (`RRF`, `RLF`) instructions, this bit is loaded with either the high or low order bit of the source register.

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
- n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

2.3.2 OPTION REGISTER

The OPTION register is a readable and writable register which contains various control bits to configure the TMR0/WDT prescaler, the external INT interrupt, TMR0, and the weak pull-ups on PORTB.

Note: When the prescaler is assigned to the WDT (PSA = '1'), TMR0 has a 1:1 prescaler assignment.

REGISTER 2-2: OPTION REGISTER (ADDRESS 81h)

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

bit 7 **$\overline{\text{RBPU}}$:** PORTB Pull-up Enable bit
1 = PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled by individual port latch values

bit 6 **INTEDG**: Interrupt Edge Select bit
1 = Interrupt on rising edge of RB0/INT pin
0 = Interrupt on falling edge of RB0/INT pin

bit 5 **T0CS**: TMR0 Clock Source Select bit
1 = Transition on RA4/T0CKI pin
0 = Internal instruction cycle clock (CLKOUT)

bit 4 **T0SE**: TMR0 Source Edge Select bit
1 = Increment on high-to-low transition on RA4/T0CKI pin
0 = Increment on low-to-high transition on RA4/T0CKI pin

bit 3 **PSA**: Prescaler Assignment bit
1 = Prescaler is assigned to the WDT
0 = Prescaler is assigned to the Timer0 module

bit 2-0 **PS2:PS0**: Prescaler Rate Select bits

Bit Value	TMR0 Rate	WDT Rate
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
- n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

PIC16F84A

2.3.3 INTCON REGISTER

The INTCON register is a readable and writable register that contains the various enable bits for all interrupt sources.

Note: Interrupt flag bits are set when an interrupt condition occurs, regardless of the state of its corresponding enable bit or the global enable bit, GIE (INTCON<7>).

REGISTER 2-3: INTCON REGISTER (ADDRESS 0Bh, 8Bh)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF
bit 7							bit 0

bit 7 **GIE:** Global Interrupt Enable bit
1 = Enables all unmasked interrupts
0 = Disables all interrupts

bit 6 **EEIE**: EE Write Complete Interrupt Enable bit
1 = Enables the EE Write Complete interrupts
0 = Disables the EE Write Complete interrupt

bit 5 **T0IE**: TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 interrupt
0 = Disables the TMR0 interrupt

bit 4 **INTE**: RB0/INT External Interrupt Enable bit
1 = Enables the RB0/INT external interrupt
0 = Disables the RB0/INT external interrupt

bit 3 **RBIE**: RB Port Change Interrupt Enable bit
1 = Enables the RB port change interrupt
0 = Disables the RB port change interrupt

bit 2 **T0IF**: TMR0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed (must be cleared in software)
0 = TMR0 register did not overflow

bit 1 **INTF**: RB0/INT External Interrupt Flag bit
1 = The RB0/INT external interrupt occurred (must be cleared in software)
0 = The RB0/INT external interrupt did not occur

bit 0 **RBIF**: RB Port Change Interrupt Flag bit
1 = At least one of the RB7:RB4 pins changed state (must be cleared in software)
0 = None of the RB7:RB4 pins have changed state

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
- n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

2.4 PCL and PCLATH

The program counter (PC) specifies the address of the instruction to fetch for execution. The PC is 13 bits wide. The low byte is called the PCL register. This register is readable and writable. The high byte is called the PCH register. This register contains the PC<12:8> bits and is not directly readable or writable. If the program counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a NOP. All updates to the PCH register go through the PCLATH register.

2.4.1 STACK

The stack allows a combination of up to 8 program calls and interrupts to occur. The stack contains the return address from this branch in program execution.

Mid-range devices have an 8 level deep x 13-bit wide hardware stack. The stack space is not part of either program or data space and the stack pointer is not readable or writable. The PC is PUSHed onto the stack when a CALL instruction is executed or an interrupt causes a branch. The stack is POPed in the event of a RETURN, RETLW or a RETFIE instruction execution. PCLATH is not modified when the stack is PUSHed or POPed.

After the stack has been PUSHed eight times, the ninth push overwrites the value that was stored from the first push. The tenth push overwrites the second push (and so on).

2.5 Indirect Addressing; INDF and FSR Registers

The INDF register is not a physical register. Addressing INDF actually addresses the register whose address is contained in the FSR register (FSR is a *pointer*). This is indirect addressing.

EXAMPLE 2-1: INDIRECT ADDRESSING

- Register file 05 contains the value 10h
- Register file 06 contains the value 0Ah
- Load the value 05 into the FSR register
- A read of the INDF register will return the value of 10h
- Increment the value of the FSR register by one (FSR = 06)
- A read of the INDF register now will return the value of 0Ah.

Reading INDF itself indirectly (FSR = 0) will produce 00h. Writing to the INDF register indirectly results in a no-operation (although STATUS bits may be affected).

A simple program to clear RAM locations 20h-2Fh using indirect addressing is shown in Example 2-2.

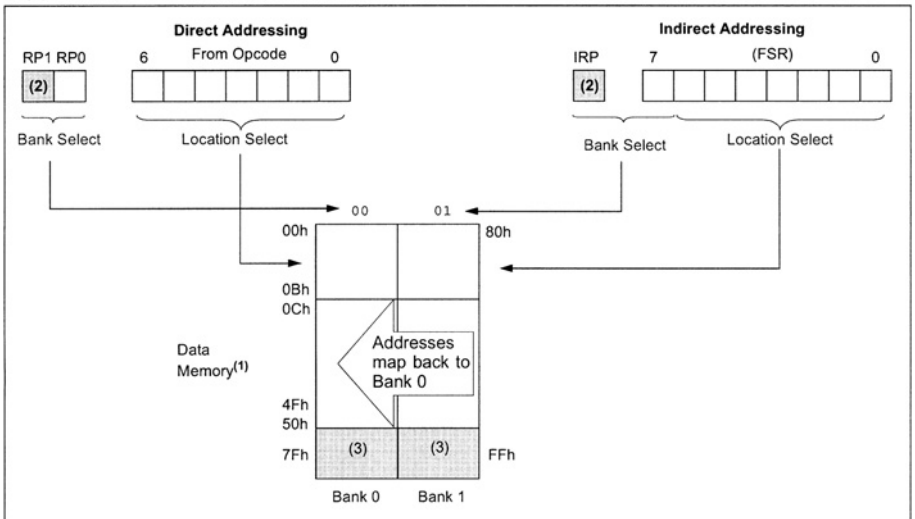
EXAMPLE 2-2: HOW TO CLEAR RAM USING INDIRECT ADDRESSING

```
        movlw   0x20    ;initialize pointer
        movwf   FSR     ;to RAM
NEXT    clrf    INDF    ;clear INDF register
        incf    FSR     ;inc pointer
        btfss   FSR,4   ;all done?
        goto    NEXT    ;NO, clear next
CONTINUE
        :               ;YES, continue
```

An effective 9-bit address is obtained by concatenating the 8-bit FSR register and the IRP bit (STATUS<7>), as shown in Figure 2-3. However, IRP is not used in the PIC16F84A.

PIC16F84A

FIGURE 2-3: DIRECT/INDIRECT ADDRESSING



Note 1: For memory map detail, see Figure 2-2.
2: Maintain as clear for upward compatibility with future products.
3: Not implemented.

PIC16F84A

3.0 DATA EEPROM MEMORY

The EEPROM data memory is readable and writable during normal operation (full VDD range). This memory is not directly mapped in the register file space. Instead it is indirectly addressed through the Special Function Registers. There are four SFRs used to read and write this memory. These registers are:

- EECON1
- EECON2 (not a physically implemented register)
- EEDATA
- EEADR

EEDATA holds the 8-bit data for read/write, and EEADR holds the address of the EEPROM location being accessed. PIC16F84A devices have 64 bytes of data EEPROM with an address range from 0h to 3Fh.

The EEPROM data memory allows byte read and write. A byte write automatically erases the location and writes the new data (erase before write). The EEPROM data memory is rated for high erase/write cycles. The write time is controlled by an on-chip timer. The write-time will vary with voltage and temperature as well as from chip to chip. Please refer to AC specifications for exact limits.

When the device is code protected, the CPU may continue to read and write the data EEPROM memory. The device programmer can no longer access this memory.

Additional information on the Data EEPROM is available in the PICmicro™ Mid-Range Reference Manual (DS33023).

REGISTER 3-1: EECON1 REGISTER (ADDRESS 88h)

U-0	U-0	U-0	R/W-0	R/W-x	R/W-0	R/S-0	R/S-0
—	—	—	EEIF	WRERR	WREN	WR	RD
bit 7							bit 0

bit 7-5 **Unimplemented:** Read as '0'

bit 4 **EEIF:** EEPROM Write Operation Interrupt Flag bit
1 = The write operation completed (must be cleared in software)
0 = The write operation is not complete or has not been started

bit 3 **WRERR:** EEPROM Error Flag bit
1 = A write operation is prematurely terminated (any $\overline{MCLR}$ Reset or any WDT Reset during normal operation)
0 = The write operation completed

bit 2 **WREN:** EEPROM Write Enable bit
1 = Allows write cycles
0 = Inhibits write to the EEPROM

bit 1 **WR:** Write Control bit
1 = Initiates a write cycle. The bit is cleared by hardware once write is complete. The WR bit can only be set (not cleared) in software.
0 = Write cycle to the EEPROM is complete

bit 0 **RD:** Read Control bit
1 = Initiates an EEPROM read RD is cleared in hardware. The RD bit can only be set (not cleared) in software.
0 = Does not initiate an EEPROM read

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
- n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

PIC16F84A

3.1 Reading the EEPROM Data Memory

To read a data memory location, the user must write the address to the EEADR register and then set control bit RD (EECON1<0>). The data is available, in the very next cycle, in the EEDATA register; therefore, it can be read in the next instruction. EEDATA will hold this value until another read or until it is written to by the user (during a write operation).

EXAMPLE 3-1: DATA EEPROM READ

```
BCF     STATUS, RP0   ; Bank 0
MOVLW   CONFIG_ADDR   ;
MOVWF   EEADR         ; Address to read
BSF     STATUS, RP0   ; Bank 1
BSF     EECON1, RD    ; EE Read
BCF     STATUS, RP0   ; Bank 0
MOVF    EEDATA, W     ; W = EEDATA
```

3.2 Writing to the EEPROM Data Memory

To write an EEPROM data location, the user must first write the address to the EEADR register and the data to the EEDATA register. Then the user must follow a specific sequence to initiate the write for each byte.

EXAMPLE 3-2: DATA EEPROM WRITE

```
        BSF     STATUS, RP0  ; Bank 1
        BCF     INTCON, GIE  ; Disable INTs.
        BSF     EECON1, WREN ; Enable Write
        MOVLW   55h          ;
Required
Sequence
        MOVWF   EECON2       ; Write 55h
        MOVLW   AAh          ;
        MOVWF   EECON2       ; Write AAh
        BSF     EECON1,WR    ; Set WR bit
                             ;   begin write
        BSF     INTCON, GIE  ; Enable INTs.
```

The write will not initiate if the above sequence is not exactly followed (write 55h to EECON2, write AAh to EECON2, then set WR bit) for each byte. We strongly recommend that interrupts be disabled during this code segment.

Additionally, the WREN bit in EECON1 must be set to enable write. This mechanism prevents accidental writes to data EEPROM due to errant (unexpected) code execution (i.e., lost programs). The user should keep the WREN bit clear at all times, except when updating EEPROM. The WREN bit is not cleared by hardware.

After a write sequence has been initiated, clearing the WREN bit will not affect this write cycle. The WR bit will be inhibited from being set unless the WREN bit is set.

At the completion of the write cycle, the WR bit is cleared in hardware and the EE Write Complete Interrupt Flag bit (EEIF) is set. The user can either enable this interrupt or poll this bit. EEIF must be cleared by software.

3.3 Write Verify

Depending on the application, good programming practice may dictate that the value written to the Data EEPROM should be verified (Example 3-3) to the desired value to be written. This should be used in applications where an EEPROM bit will be stressed near the specification limit.

Generally, the EEPROM write failure will be a bit which was written as a '0', but reads back as a '1' (due to leakage off the bit).

EXAMPLE 3-3: WRITE VERIFY

```
        BCF   STATUS,RP0 ; Bank 0
        :                ; Any code
        :                ; can go here
        MOVF  EEDATA,W   ; Must be in Bank 0
        BSF   STATUS,RP0 ; Bank 1
READ
        BSF   EECON1, RD ; YES, Read the
                         ; value written
        BCF   STATUS, RP0 ; Bank 0
                         ;
                         ; Is the value written
                         ; (in W reg) and
                         ; read (in EEDATA)
                         ; the same?
                         ;
        SUBWF EEDATA, W  ;
        BTFSS STATUS, Z  ; Is difference 0?
        GOTO  WRITE_ERR  ; NO, Write error
```

TABLE 3-1: REGISTERS/BITS ASSOCIATED WITH DATA EEPROM

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on Power-on Reset	Value on all other RESETS
08h	EEDATA	EEPROM Data Register								xxxx xxxx	uuuu uuuu
09h	EEADR	EEPROM Address Register								xxxx xxxx	uuuu uuuu
88h	EECON1	—	—	—	EEIF	WRERR	WREN	WR	RD	---0 x000	---0 q000
89h	EECON2	EEPROM Control Register 2								---- ----	---- ----

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0', q = value depends upon condition.
Shaded cells are not used by data EEPROM.

PIC16F84A

4.0 I/O PORTS

Some pins for these I/O ports are multiplexed with an alternate function for the peripheral features on the device. In general, when a peripheral is enabled, that pin may not be used as a general purpose I/O pin.

Additional information on I/O ports may be found in the PICmicro™ Mid-Range Reference Manual (DS33023).

4.1 PORTA and TRISA Registers

PORTA is a 5-bit wide, bi-directional port. The corresponding data direction register is TRISA. Setting a TRISA bit (= 1) will make the corresponding PORTA pin an input (i.e., put the corresponding output driver in a Hi-Impedance mode). Clearing a TRISA bit (= 0) will make the corresponding PORTA pin an output (i.e., put the contents of the output latch on the selected pin).

Note: On a Power-on Reset, these pins are configured as inputs and read as '0'.

Reading the PORTA register reads the status of the pins, whereas writing to it will write to the port latch. All write operations are read-modify-write operations. Therefore, a write to a port implies that the port pins are read. This value is modified and then written to the port data latch.

Pin RA4 is multiplexed with the Timer0 module clock input to become the RA4/T0CKI pin. The RA4/T0CKI pin is a Schmitt Trigger input and an open drain output. All other RA port pins have TTL input levels and full CMOS output drivers.

EXAMPLE 4-1: INITIALIZING PORTA

```
BCF     STATUS, RP0 ;
CLRF    PORTA       ; Initialize PORTA by
                    ; clearing output
                    ; data latches
BSF     STATUS, RP0 ; Select Bank 1
MOVLW   0x0F        ; Value used to
                    ; initialize data
                    ; direction
MOVWF   TRISA       ; Set RA<3:0> as inputs
                    ; RA4 as output
                    ; TRISA<7:5> are always
                    ; read as '0'.
```

FIGURE 4-1: BLOCK DIAGRAM OF PINS RA3:RA0

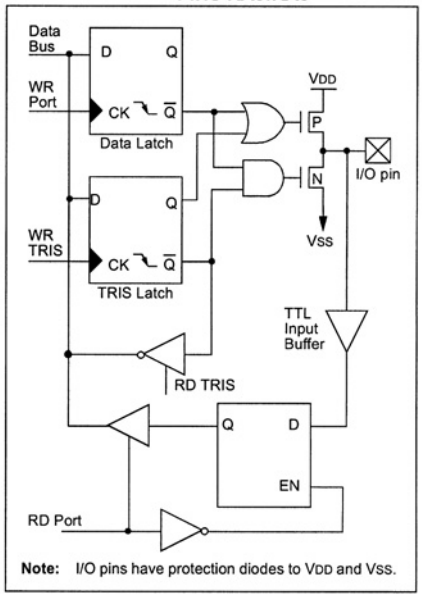
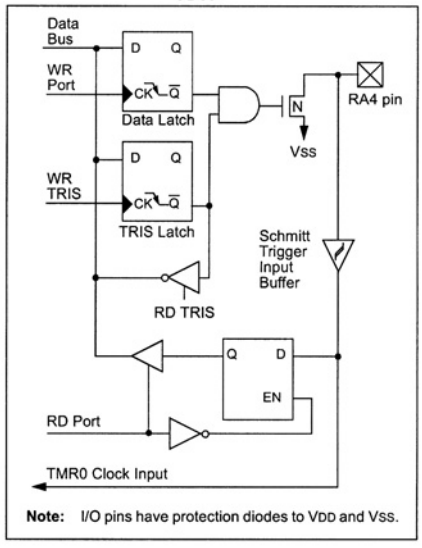


FIGURE 4-2: BLOCK DIAGRAM OF PIN RA4



PIC16F84A

TABLE 4-1: PORTA FUNCTIONS

Name	Bit0	Buffer Type	Function
RA0	bit0	TTL	Input/output
RA1	bit1	TTL	Input/output
RA2	bit2	TTL	Input/output
RA3	bit3	TTL	Input/output
RA4/T0CKI	bit4	ST	Input/output or external clock input for TMR0. Output is open drain type.

Legend: TTL = TTL input, ST = Schmitt Trigger input

TABLE 4-2: SUMMARY OF REGISTERS ASSOCIATED WITH PORTA

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on Power-on Reset	Value on all other RESETS
05h	PORTA	—	—	—	RA4/T0CKI	RA3	RA2	RA1	RA0	---x xxxx	---u uuuu
85h	TRISA	—	—	—	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0	---1 1111	---1 1111

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0'. Shaded cells are unimplemented, read as '0'.

4.2 PORTB and TRISB Registers

PORTB is an 8-bit wide, bi-directional port. The corresponding data direction register is TRISB. Setting a TRISB bit (= 1) will make the corresponding PORTB pin an input (i.e., put the corresponding output driver in a Hi-Impedance mode). Clearing a TRISB bit (= 0) will make the corresponding PORTB pin an output (i.e., put the contents of the output latch on the selected pin).

EXAMPLE 4-2: INITIALIZING PORTB

```
BCF     STATUS, RP0 ;
CLRF    PORTB       ; Initialize PORTB by
                    ; clearing output
                    ; data latches
BSF     STATUS, RP0 ; Select Bank 1
MOVLW   0xCF        ; Value used to
                    ; initialize data
                    ; direction
MOVWF   TRISB       ; Set RB<3:0> as inputs
                    ; RB<5:4> as outputs
                    ; RB<7:6> as inputs
```

Each of the PORTB pins has a weak internal pull-up. A single control bit can turn on all the pull-ups. This is performed by clearing bit $\overline{\text{RBPU}}$ (OPTION<7>). The weak pull-up is automatically turned off when the port pin is configured as an output. The pull-ups are disabled on a Power-on Reset.

Four of PORTB's pins, RB7:RB4, have an interrupt-on-change feature. Only pins configured as inputs can cause this interrupt to occur (i.e., any RB7:RB4 pin configured as an output is excluded from the interrupt-on-change comparison). The input pins (of RB7:RB4) are compared with the old value latched on the last read of PORTB. The "mismatch" outputs of RB7:RB4 are OR'ed together to generate the RB Port Change Interrupt with flag bit RBIF (INTCON<0>).

This interrupt can wake the device from SLEEP. The user, in the Interrupt Service Routine, can clear the interrupt in the following manner:

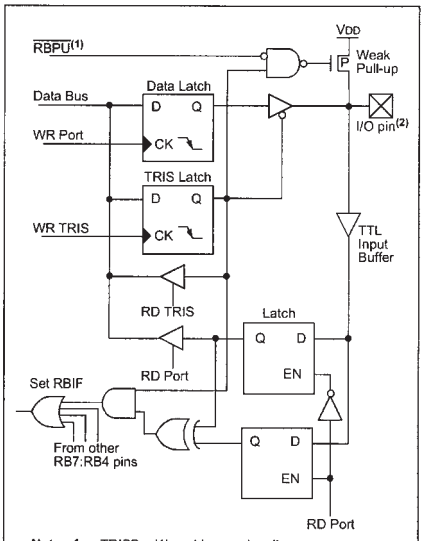
a) Any read or write of PORTB. This will end the mismatch condition.

b) Clear flag bit RBIF.

A mismatch condition will continue to set flag bit RBIF. Reading PORTB will end the mismatch condition and allow flag bit RBIF to be cleared.

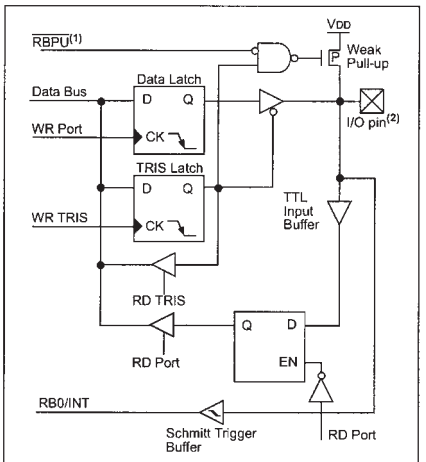
The interrupt-on-change feature is recommended for wake-up on key depression operation and operations where PORTB is only used for the interrupt-on-change feature. Polling of PORTB is not recommended while using the interrupt-on-change feature.

FIGURE 4-3: BLOCK DIAGRAM OF PINS RB7:RB4



Note 1: TRISB = '1' enables weak pull-up (if $\overline{\text{RBPU}}$ = '0' in the OPTION_REG register).
2: I/O pins have diode protection to VDD and VSS.

FIGURE 4-4: BLOCK DIAGRAM OF PINS RB3:RB0



Note 1: TRISB = '1' enables weak pull-up (if $\overline{\text{RBPU}}$ = '0' in the OPTION_REG register).
2: I/O pins have diode protection to VDD and VSS.

PIC16F84A

TABLE 4-3: PORTB FUNCTIONS

Name	Bit	Buffer Type	I/O Consistency Function
RB0/INT	bit0	TTL/ST(1)	Input/output pin or external interrupt input. Internal software programmable weak pull-up.
RB1	bit1	TTL	Input/output pin. Internal software programmable weak pull-up.
RB2	bit2	TTL	Input/output pin. Internal software programmable weak pull-up.
RB3	bit3	TTL	Input/output pin. Internal software programmable weak pull-up.
RB4	bit4	TTL	Input/output pin (with interrupt-on-change). Internal software programmable weak pull-up.
RB5	bit5	TTL	Input/output pin (with interrupt-on-change). Internal software programmable weak pull-up.
RB6	bit6	TTL/ST(2)	Input/output pin (with interrupt-on-change). Internal software programmable weak pull-up. Serial programming clock.
RB7	bit7	TTL/ST(2)	Input/output pin (with interrupt-on-change). Internal software programmable weak pull-up. Serial programming data.

Legend: TTL = TTL input, ST = Schmitt Trigger.

Note 1: This buffer is a Schmitt Trigger input when configured as the external interrupt.

2: This buffer is a Schmitt Trigger input when used in Serial Programming mode.

TABLE 4-4: SUMMARY OF REGISTERS ASSOCIATED WITH PORTB

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on Power-on Reset	Value on all other RESETS
06h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0/INT	xxxx xxxx	uuuu uuuu
86h	TRISB	TRISB7	TRISB6	TRISB5	TRISB4	TRISB3	TRISB2	TRISB1	TRISB0	1111 1111	1111 1111
81h	OPTION_REG	RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	1111 1111
0Bh,8Bh	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000u

Legend: x = unknown, u = unchanged. Shaded cells are not used by PORTB.

5.0 TIMER0 MODULE

The Timer0 module timer/counter has the following features:

- 8-bit timer/counter
- Readable and writable
- Internal or external clock select
- Edge select for external clock
- 8-bit software programmable prescaler
- Interrupt-on-overflow from FFh to 00h

Figure 5-1 is a simplified block diagram of the Timer0 module.

Additional information on timer modules is available in the PICmicro™ Mid-Range Reference Manual (DS33023).

5.1 Timer0 Operation

Timer0 can operate as a timer or as a counter.

Timer mode is selected by clearing bit T0CS (OPTION_REG<5>). In Timer mode, the Timer0 module will increment every instruction cycle (without prescaler). If the TMR0 register is written, the increment is inhibited for the following two instruction cycles. The user can work around this by writing an adjusted value to the TMR0 register.

Counter mode is selected by setting bit T0CS (OPTION_REG<5>). In Counter mode, Timer0 will increment, either on every rising or falling edge of pin RA4/T0CKI. The incrementing edge is determined by the Timer0 Source Edge Select bit, T0SE (OPTION_REG<4>). Clearing bit T0SE selects the rising edge. Restrictions on the external clock input are discussed below.

When an external clock input is used for Timer0, it must meet certain requirements. The requirements ensure the external clock can be synchronized with the internal phase clock (TOSC). Also, there is a delay in the actual incrementing of Timer0 after synchronization.

Additional information on external clock requirements is available in the PICmicro™ Mid-Range Reference Manual, (DS33023).

5.2 Prescaler

An 8-bit counter is available as a prescaler for the Timer0 module, or as a postscaler for the Watchdog Timer, respectively (Figure 5-2). For simplicity, this counter is being referred to as "prescaler" throughout this data sheet. Note that there is only one prescaler available which is mutually exclusively shared between the Timer0 module and the Watchdog Timer. Thus, a prescaler assignment for the Timer0 module means that there is no prescaler for the Watchdog Timer, and vice-versa.

The prescaler is not readable or writable.

The PSA and PS2:PS0 bits (OPTION_REG<3:0>) determine the prescaler assignment and prescale ratio.

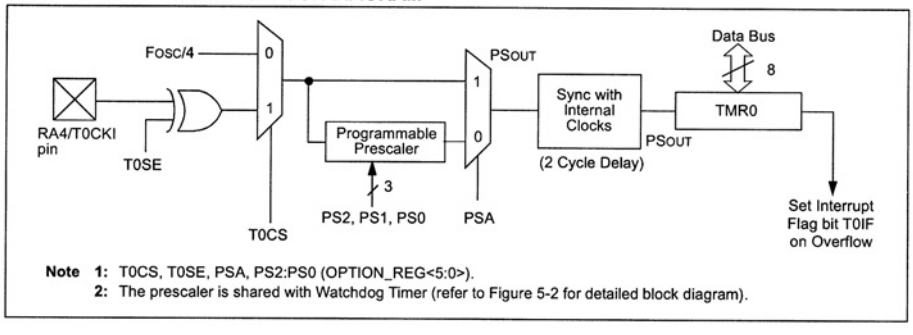
Clearing bit PSA will assign the prescaler to the Timer0 module. When the prescaler is assigned to the Timer0 module, prescale values of 1:2, 1:4, ..., 1:256 are selectable.

Setting bit PSA will assign the prescaler to the Watchdog Timer (WDT). When the prescaler is assigned to the WDT, prescale values of 1:1, 1:2, ..., 1:128 are selectable.

When assigned to the Timer0 module, all instructions writing to the TMR0 register (e.g., `CLRF 1`, `MOVWF 1`, `BSF 1`, etc.) will clear the prescaler. When assigned to WDT, a `CLRWDT` instruction will clear the prescaler along with the WDT.

Note: Writing to TMR0 when the prescaler is assigned to Timer0 will clear the prescaler count, but will not change the prescaler assignment.

FIGURE 5-1: TIMER0 BLOCK DIAGRAM



Note 1: T0CS, T0SE, PSA, PS2:PS0 (OPTION_REG<5:0>).
2: The prescaler is shared with Watchdog Timer (refer to Figure 5-2 for detailed block diagram).

5.2.1 SWITCHING PRESCALER ASSIGNMENT

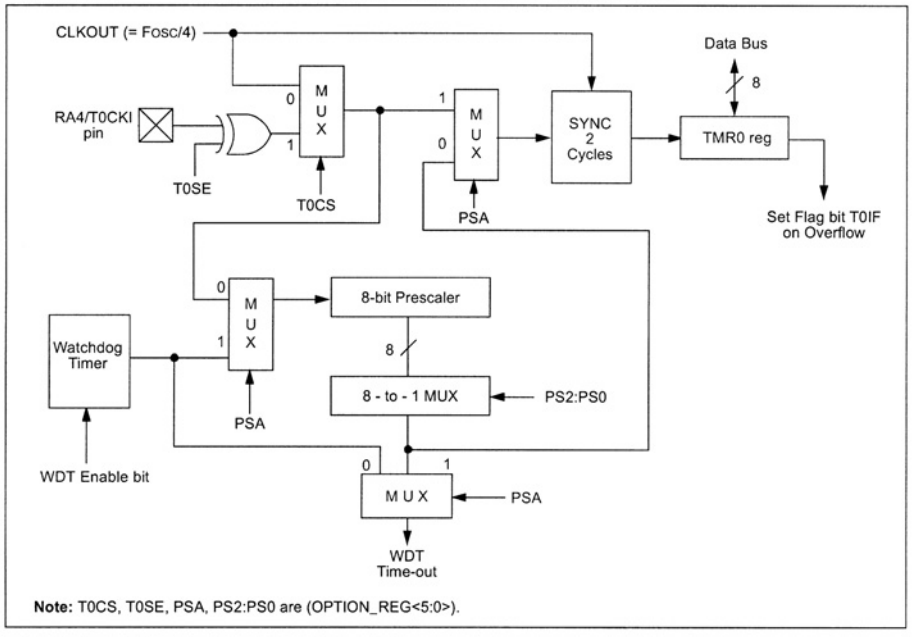
The prescaler assignment is fully under software control (i.e., it can be changed "on the fly" during program execution).

Note: To avoid an unintended device RESET, a specific instruction sequence (shown in the PICmicro™ Mid-Range Reference Manual, DS33023) must be executed when changing the prescaler assignment from Timer0 to the WDT. This sequence must be followed even if the WDT is disabled.

5.3 Timer0 Interrupt

The TMR0 interrupt is generated when the TMR0 register overflows from FFh to 00h. This overflow sets bit T0IF (INTCON<2>). The interrupt can be masked by clearing bit T0IE (INTCON<5>). Bit T0IF must be cleared in software by the Timer0 module Interrupt Service Routine before re-enabling this interrupt. The TMR0 interrupt cannot awaken the processor from SLEEP since the timer is shut-off during SLEEP.

FIGURE 5-2: BLOCK DIAGRAM OF THE TIMER0/WDT PRESCALER



Note: T0CS, T0SE, PSA, PS2:PS0 are (OPTION_REG<5:0>).

TABLE 5-1: REGISTERS ASSOCIATED WITH TIMER0

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR	Value on all other RESETS
01h	TMR0	Timer0 Module Register								xxxx xxxx	uuuu uuuu
0Bh,8Bh	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000u
81h	OPTION_REG	RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	1111 1111
85h	TRISA	—	—	—	PORTA Data Direction Register					---1 1111	---1 1111

Legend: x = unknown, u = unchanged, - = unimplemented locations read as '0'. Shaded cells are not used by Timer0.

PIC16F84A

6.0 SPECIAL FEATURES OF THE CPU

What sets a microcontroller apart from other processors are special circuits to deal with the needs of real time applications. The PIC16F84A has a host of such features intended to maximize system reliability, minimize cost through elimination of external components, provide power saving operating modes and offer code protection. These features are:

- OSC Selection
- RESET
  - Power-on Reset (POR)
  - Power-up Timer (PWRT)
  - Oscillator Start-up Timer (OST)
- Interrupts
- Watchdog Timer (WDT)
- SLEEP
- Code Protection
- ID Locations
- In-Circuit Serial Programming™ (ICSP™)

The PIC16F84A has a Watchdog Timer which can be shut-off only through configuration bits. It runs off its own RC oscillator for added reliability. There are two timers that offer necessary delays on power-up. One is the Oscillator Start-up Timer (OST), intended to keep the chip in RESET until the crystal oscillator is stable. The other is the Power-up Timer (PWRT), which provides a fixed delay of 72 ms (nominal) on power-up only. This design keeps the device in RESET while the power supply stabilizes. With these two timers on-chip, most applications need no external RESET circuitry.

SLEEP mode offers a very low current power-down mode. The user can wake-up from SLEEP through external RESET, Watchdog Timer Time-out or through an interrupt. Several oscillator options are provided to allow the part to fit the application. The RC oscillator option saves system cost while the LP crystal option saves power. A set of configuration bits are used to select the various options.

Additional information on special features is available in the PICmicro™ Mid-Range Reference Manual (DS33023).

6.1 Configuration Bits

The configuration bits can be programmed (read as '0'), or left unprogrammed (read as '1'), to select various device configurations. These bits are mapped in program memory location 2007h.

Address 2007h is beyond the user program memory space and it belongs to the special test/configuration memory space (2000h - 3FFFh). This space can only be accessed during programming.

REGISTER 6-1: PIC16F84A CONFIGURATION WORD

R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u
CP	CP	CP	CP	CP	CP	CP	CP	CP	CP	$\overline{\text{PWRTE}}$	WDTE	F0SC1	F0SC0
bit13													bit0

bit 13-4 **CP:** Code Protection bit
1 = Code protection disabled
0 = All program memory is code protected

bit 3 **$\overline{\text{PWRTE}}$**: Power-up Timer Enable bit
1 = Power-up Timer is disabled
0 = Power-up Timer is enabled

bit 2 **WDTE**: Watchdog Timer Enable bit
1 = WDT enabled
0 = WDT disabled

bit 1-0 **FOSC1:FOSC0**: Oscillator Selection bits
11 = RC oscillator
10 = HS oscillator
01 = XT oscillator
00 = LP oscillator

PIC16F84A

6.2 Oscillator Configurations

6.2.1 OSCILLATOR TYPES

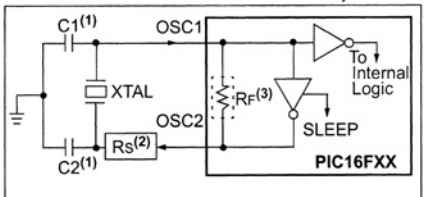
The PIC16F84A can be operated in four different oscillator modes. The user can program two configuration bits (FOSC1 and FOSC0) to select one of these four modes:

- LP Low Power Crystal
- XT Crystal/Resonator
- HS High Speed Crystal/Resonator
- RC Resistor/Capacitor

6.2.2 CRYSTAL OSCILLATOR/CERAMIC RESONATORS

In XT, LP, or HS modes, a crystal or ceramic resonator is connected to the OSC1/CLKIN and OSC2/CLKOUT pins to establish oscillation (Figure 6-1).

FIGURE 6-1: CRYSTAL/CERAMIC RESONATOR OPERATION (HS, XT OR LP OSC CONFIGURATION)



Note 1: See Table 6-1 for recommended values of C1 and C2.

2: A series resistor (Rs) may be required for AT strip cut crystals.

The PIC16F84A oscillator design requires the use of a parallel cut crystal. Use of a series cut crystal may give a frequency out of the crystal manufacturers specifications. When in XT, LP, or HS modes, the device can have an external clock source to drive the OSC1/CLKIN pin (Figure 6-2).

FIGURE 6-2: EXTERNAL CLOCK INPUT OPERATION (HS, XT OR LP OSC CONFIGURATION)

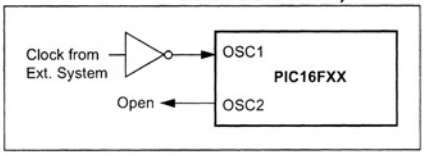


TABLE 6-1: CAPACITOR SELECTION FOR CERAMIC RESONATORS

Ranges Tested:			
Mode	**Freq**	**OSC1/C1**	**OSC2/C2**
XT	455 kHz	47 - 100 pF	47 - 100 pF
	2.0 MHz	15 - 33 pF	15 - 33 pF
	4.0 MHz	15 - 33 pF	15 - 33 pF
HS	8.0 MHz	15 - 33 pF	15 - 33 pF
	10.0 MHz	15 - 33 pF	15 - 33 pF

Note: Recommended values of C1 and C2 are identical to the ranges tested in this table. Higher capacitance increases the stability of the oscillator, but also increases the start-up time. These values are for design guidance only. Since each resonator has its own characteristics, the user should consult the resonator manufacturer for the appropriate values of external components.

Note: When using resonators with frequencies above 3.5 MHz, the use of HS mode rather than XT mode, is recommended. HS mode may be used at any VDD for which the controller is rated.

TABLE 6-2: CAPACITOR SELECTION FOR CRYSTAL OSCILLATOR

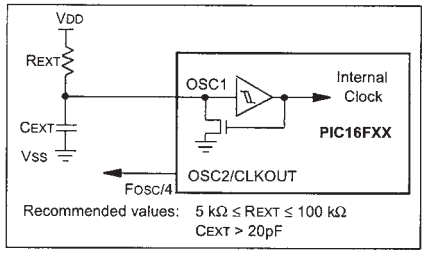
Mode	Freq	OSC1/C1	OSC2/C2
LP	32 kHz	68 - 100 pF	68 - 100 pF
	200 kHz	15 - 33 pF	15 - 33 pF
XT	100 kHz	100 - 150 pF	100 - 150 pF
	2 MHz	15 - 33 pF	15 - 33 pF
	4 MHz	15 - 33 pF	15 - 33 pF
HS	4 MHz	15 - 33 pF	15 - 33 pF
	20 MHz	15 - 33 pF	15 - 33 pF

Note: Higher capacitance increases the stability of the oscillator, but also increases the start-up time. These values are for design guidance only. Rs may be required in HS mode, as well as XT mode, to avoid overdriving crystals with low drive level specification. Since each crystal has its own characteristics, the user should consult the crystal manufacturer for appropriate values of external components.
For V_{DD} > 4.5V, C1 = C2 ≈ 30 pF is recommended.

6.2.3 RC OSCILLATOR

For timing insensitive applications, the RC device option offers additional cost savings. The RC oscillator frequency is a function of the supply voltage, the resistor (R_{EXT}) values, capacitor (C_{EXT}) values, and the operating temperature. In addition to this, the oscillator frequency will vary from unit to unit due to normal process parameter variation. Furthermore, the difference in lead frame capacitance between package types also affects the oscillation frequency, especially for low C_{EXT} values. The user needs to take into account variation, due to tolerance of the external R and C components. Figure 6-3 shows how an R/C combination is connected to the PIC16F84A.

FIGURE 6-3: RC OSCILLATOR MODE



PIC16F84A

6.3 RESET

The PIC16F84A differentiates between various kinds of RESET:

- Power-on Reset (POR)
- $\overline{MCLR}$ during normal operation
- $\overline{MCLR}$ during SLEEP
- WDT Reset (during normal operation)
- WDT Wake-up (during SLEEP)

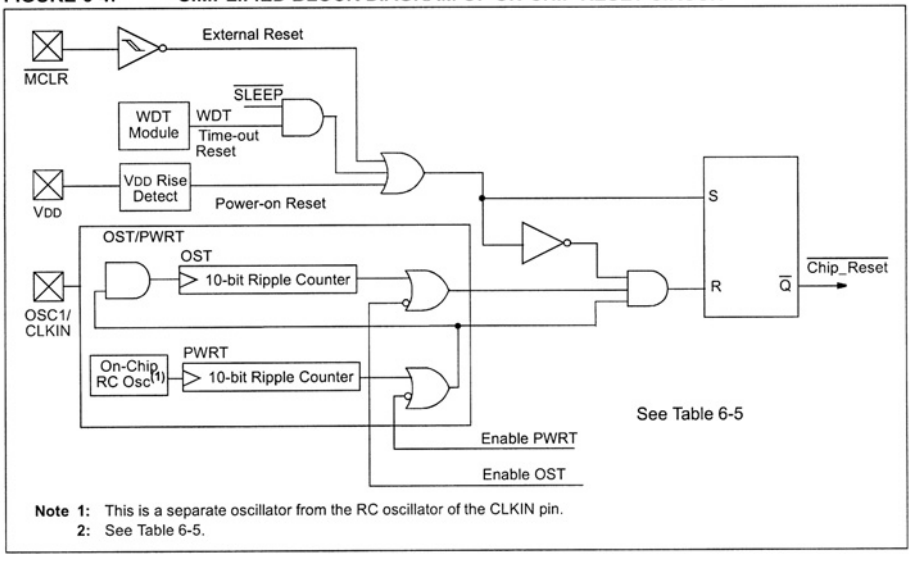
Figure 6-4 shows a simplified block diagram of the On-Chip RESET Circuit. The $\overline{MCLR}$ Reset path has a noise filter to ignore small pulses. The electrical specifications state the pulse width requirements for the $\overline{MCLR}$ pin.

Some registers are not affected in any RESET condition; their status is unknown on a POR and unchanged in any other RESET. Most other registers are reset to a "RESET state" on POR, $\overline{MCLR}$ or WDT Reset during normal operation and on $\overline{MCLR}$ during SLEEP. They are not affected by a WDT Reset during SLEEP, since this RESET is viewed as the resumption of normal operation.

Table 6-3 gives a description of RESET conditions for the program counter (PC) and the STATUS register. Table 6-4 gives a full description of RESET states for all registers.

The $\overline{TO}$ and $\overline{PD}$ bits are set or cleared differently in different RESET situations (Section 6.7). These bits are used in software to determine the nature of the RESET.

FIGURE 6-4: SIMPLIFIED BLOCK DIAGRAM OF ON-CHIP RESET CIRCUIT



Note 1: This is a separate oscillator from the RC oscillator of the CLKIN pin.
2: See Table 6-5.

TABLE 6-3: RESET CONDITION FOR PROGRAM COUNTER AND THE STATUS REGISTER

Condition	Program Counter	STATUS Register
Power-on Reset	000h	0001 1xxx
$\overline{MCLR}$ during normal operation	000h	000u uuuu
$\overline{MCLR}$ during SLEEP	000h	0001 0uuu
WDT Reset (during normal operation)	000h	0000 1uuu
WDT Wake-up	PC + 1	uuu0 0uuu
Interrupt wake-up from SLEEP	PC + 1(1)	uuu1 0uuu

Legend: u = unchanged, x = unknown
Note 1: When the wake-up is due to an interrupt and the GIE bit is set, the PC is loaded with the interrupt vector (0004h).

PIC16F84A

TABLE 6-4: RESET CONDITIONS FOR ALL REGISTERS

Register	Address	Power-on Reset	$\overline{MCLR}$ during: – normal operation – SLEEP WDT Reset during normal operation	Wake-up from SLEEP: – through interrupt – through WDT Time-out
W	—	xxxx xxxx	uuuu uuuu	uuuu uuuu
INDF	00h	---- ----	---- ----	---- ----
TMR0	01h	xxxx xxxx	uuuu uuuu	uuuu uuuu
PCL	02h	0000 0000	0000 0000	PC + 1(2)
STATUS	03h	0001 1xxx	000q quuu(3)	uuuq quuu(3)
FSR	04h	xxxx xxxx	uuuu uuuu	uuuu uuuu
PORTA(4)	05h	---x xxxx	---u uuuu	---u uuuu
PORTB(5)	06h	xxxx xxxx	uuuu uuuu	uuuu uuuu
EEDATA	08h	xxxx xxxx	uuuu uuuu	uuuu uuuu
EEADR	09h	xxxx xxxx	uuuu uuuu	uuuu uuuu
PCLATH	0Ah	---0 0000	---0 0000	---u uuuu
INTCON	0Bh	0000 000x	0000 000u	uuuu uuuu(1)
INDF	80h	---- ----	---- ----	---- ----
OPTION_REG	81h	1111 1111	1111 1111	uuuu uuuu
PCL	82h	0000 0000	0000 0000	PC + 1(2)
STATUS	83h	0001 1xxx	000q quuu(3)	uuuq quuu(3)
FSR	84h	xxxx xxxx	uuuu uuuu	uuuu uuuu
TRISA	85h	---1 1111	---1 1111	---u uuuu
TRISB	86h	1111 1111	1111 1111	uuuu uuuu
EECON1	88h	---0 x000	---0 q000	---0 uuuu
EECON2	89h	---- ----	---- ----	---- ----
PCLATH	8Ah	---0 0000	---0 0000	---u uuuu
INTCON	8Bh	0000 000x	0000 000u	uuuu uuuu(1)

Legend: u = unchanged, x = unknown, - = unimplemented bit, read as '0', q = value depends on condition

Note 1: One or more bits in INTCON will be affected (to cause wake-up).

2: When the wake-up is due to an interrupt and the GIE bit is set, the PC is loaded with the interrupt vector (0004h).

3: Table 6-3 lists the RESET value for each specific condition.

4: On any device RESET, these pins are configured as inputs.

5: This is the value that will be in the port output latch.

PIC16F84A

6.4 Power-on Reset (POR)

A Power-on Reset pulse is generated on-chip when VDD rise is detected (in the range of 1.2V - 1.7V). To take advantage of the POR, just tie the MCLR pin directly (or through a resistor) to VDD. This will eliminate external RC components usually needed to create Power-on Reset. A minimum rise time for VDD must be met for this to operate properly. See Electrical Specifications for details.

When the device starts normal operation (exits the RESET condition), device operating parameters (voltage, frequency, temperature, etc.) must be met to ensure operation. If these conditions are not met, the device must be held in RESET until the operating conditions are met.

For additional information, refer to Application Note AN607, "*Power-up Trouble Shooting*."

The POR circuit does not produce an internal RESET when VDD declines.

6.5 Power-up Timer (PWRT)

The Power-up Timer (PWRT) provides a fixed 72 ms nominal time-out (TPWRT) from POR (Figures 6-6 through 6-9). The Power-up Timer operates on an internal RC oscillator. The chip is kept in RESET as long as the PWRT is active. The PWRT delay allows the VDD to rise to an acceptable level (possible exception shown in Figure 6-9).

A configuration bit, PWRTE, can enable/disable the PWRT. See Register 6-1 for the operation of the PWRTE bit for a particular device.

The power-up time delay TPWRT will vary from chip to chip due to VDD, temperature, and process variation. See DC parameters for details.

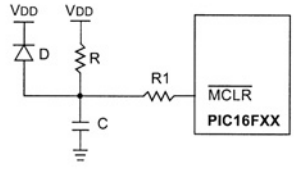
6.6 Oscillator Start-up Timer (OST)

The Oscillator Start-up Timer (OST) provides a 1024 oscillator cycle delay (from OSC1 input) after the PWRT delay ends (Figure 6-6, Figure 6-7, Figure 6-8 and Figure 6-9). This ensures the crystal oscillator or resonator has started and stabilized.

The OST time-out (TOST) is invoked only for XT, LP and HS modes and only on Power-on Reset or wake-up from SLEEP.

When VDD rises very slowly, it is possible that the TPWRT time-out and TOST time-out will expire before VDD has reached its final value. In this case (Figure 6-9), an external Power-on Reset circuit may be necessary (Figure 6-5).

FIGURE 6-5: EXTERNAL POWER-ON RESET CIRCUIT (FOR SLOW VDD POWER-UP)



Note 1: External Power-on Reset circuit is required only if VDD power-up rate is too slow. The diode D helps discharge the capacitor quickly when VDD powers down.

2: R < 40 kΩ is recommended to make sure that voltage drop across R does not exceed 0.2V (max leakage current spec on MCLR pin is 5 µA). A larger voltage drop will degrade VIH level on the MCLR pin.

3: R1 = 100Ω to 1 kΩ will limit any current flowing into MCLR from external capacitor C, in the event of a MCLR pin breakdown due to ESD or EOS.

FIGURE 6-6: TIME-OUT SEQUENCE ON POWER-UP (MCLR NOT TIED TO VDD): CASE 1

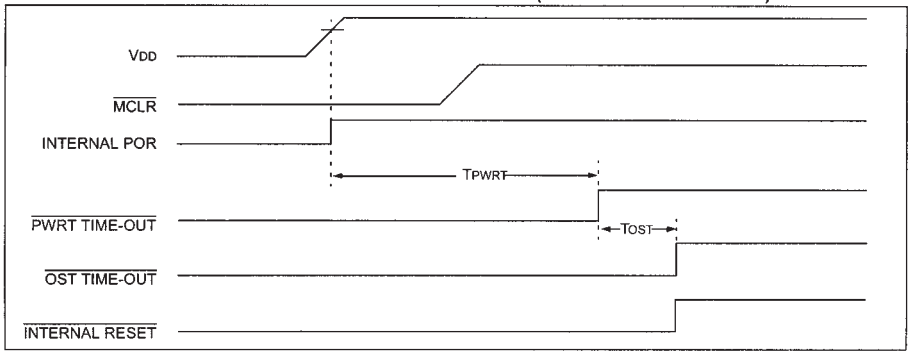


FIGURE 6-7: TIME-OUT SEQUENCE ON POWER-UP (MCLR NOT TIED TO VDD): CASE 2

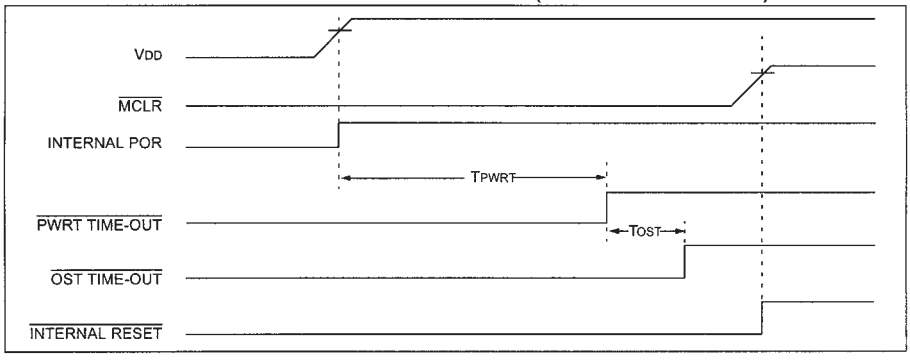
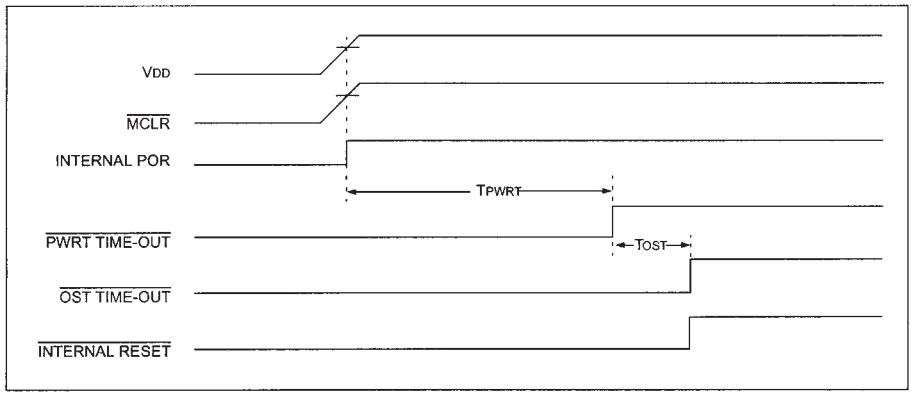
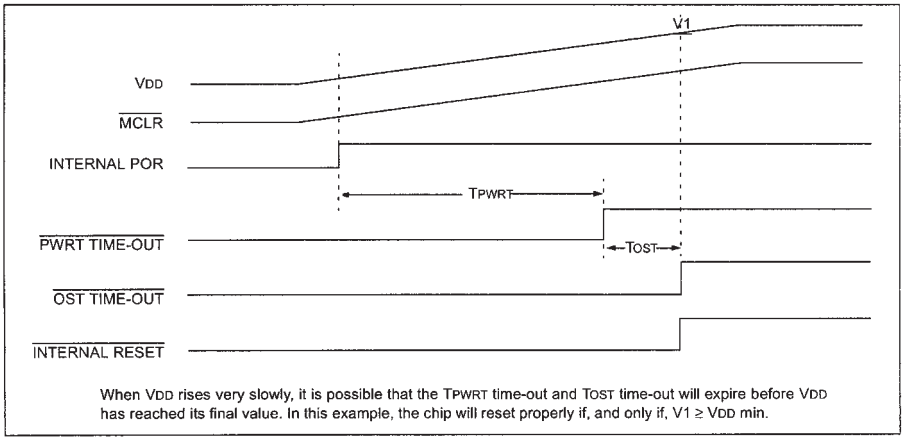


FIGURE 6-8: TIME-OUT SEQUENCE ON POWER-UP (MCLR TIED TO VDD): FAST VDD RISE TIME



PIC16F84A

FIGURE 6-9: TIME-OUT SEQUENCE ON POWER-UP (MCLR TIED TO VDD): SLOW VDD RISE TIME



When VDD rises very slowly, it is possible that the TPWRT time-out and TOST time-out will expire before VDD has reached its final value. In this example, the chip will reset properly if, and only if, V1 ≥ VDD min.

6.7 Time-out Sequence and Power-down Status Bits (TO/PD)

On power-up (Figures 6-6 through 6-9), the time-out sequence is as follows:

1. PWRT time-out is invoked after a POR has expired.
2. Then, the OST is activated.

The total time-out will vary based on oscillator configuration and PWRTE configuration bit status. For example, in RC mode with the PWRT disabled, there will be no time-out at all.

TABLE 6-5: TIME-OUT IN VARIOUS SITUATIONS

Oscillator Configuration	Power-up		Wake-up from SLEEP
	PWRT Enabled	PWRT Disabled	
XT, HS, LP	72 ms + 1024Tosc	1024Tosc	1024Tosc
RC	72 ms	—	—

Since the time-outs occur from the POR pulse, if MCLR is kept low long enough, the time-outs will expire. Then bringing MCLR high, execution will begin immediately (Figure 6-6). This is useful for testing purposes or to synchronize more than one PIC16F84A device when operating in parallel.

Table 6-6 shows the significance of the TO and PD bits. Table 6-3 lists the RESET conditions for some special registers, while Table 6-4 lists the RESET conditions for all the registers.

TABLE 6-6: STATUS BITS AND THEIR SIGNIFICANCE

TO	PD	Condition
1	1	Power-on Reset
0	x	Illegal, TO is set on POR
x	0	Illegal, PD is set on POR
0	1	WDT Reset (during normal operation)
0	0	WDT Wake-up
1	1	MCLR during normal operation
1	0	MCLR during SLEEP or interrupt wake-up from SLEEP

6.8 Interrupts

The PIC16F84A has 4 sources of interrupt:

- External interrupt RB0/INT pin
- TMR0 overflow interrupt
- PORTB change interrupts (pins RB7:RB4)
- Data EEPROM write complete interrupt

The interrupt control register (INTCON) records individual interrupt requests in flag bits. It also contains the individual and global interrupt enable bits.

The global interrupt enable bit, GIE (INTCON<7>), enables (if set) all unmasked interrupts or disables (if cleared) all interrupts. Individual interrupts can be disabled through their corresponding enable bits in INTCON register. Bit GIE is cleared on RESET.

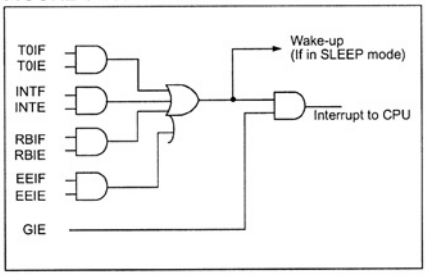
The "return from interrupt" instruction, RETFIE, exits interrupt routine as well as sets the GIE bit, which re-enables interrupts.

The RB0/INT pin interrupt, the RB port change interrupt and the TMR0 overflow interrupt flags are contained in the INTCON register.

When an interrupt is responded to, the GIE bit is cleared to disable any further interrupt, the return address is pushed onto the stack and the PC is loaded with 0004h. For external interrupt events, such as the RB0/INT pin or PORTB change interrupt, the interrupt latency will be three to four instruction cycles. The exact latency depends when the interrupt event occurs. The latency is the same for both one and two cycle instructions. Once in the Interrupt Service Routine, the source(s) of the interrupt can be determined by polling the interrupt flag bits. The interrupt flag bit(s) must be cleared in software before re-enabling interrupts to avoid infinite interrupt requests.

Note:	Individual interrupt flag bits are set regardless of the status of their corresponding mask bit or the GIE bit.

FIGURE 6-10: INTERRUPT LOGIC



6.8.1 INT INTERRUPT

External interrupt on RB0/INT pin is edge triggered: either rising if INTEDG bit (OPTION_REG<6>) is set, or falling if INTEDG bit is clear. When a valid edge appears on the RB0/INT pin, the INTF bit (INTCON<1>) is set. This interrupt can be disabled by clearing control bit INTE (INTCON<4>). Flag bit INTF must be cleared in software via the Interrupt Service Routine before re-enabling this interrupt. The INT interrupt can wake the processor from SLEEP (Section 6.11) only if the INTE bit was set prior to going into SLEEP. The status of the GIE bit decides whether the processor branches to the interrupt vector following wake-up.

6.8.2 TMR0 INTERRUPT

An overflow (FFh → 00h) in TMR0 will set flag bit T0IF (INTCON<2>). The interrupt can be enabled/disabled by setting/clearing enable bit T0IE (INTCON<5>) (Section 5.0).

6.8.3 PORTB INTERRUPT

An input change on PORTB<7:4> sets flag bit RBIF (INTCON<0>). The interrupt can be enabled/disabled by setting/clearing enable bit RBIE (INTCON<3>) (Section 4.2).

Note:	For a change on the I/O pin to be recognized, the pulse width must be at least TCY wide.

6.8.4 DATA EEPROM INTERRUPT

At the completion of a data EEPROM write cycle, flag bit EEIF (EECON1<4>) will be set. The interrupt can be enabled/disabled by setting/clearing enable bit EEIE (INTCON<6>) (Section 3.0).

PIC16F84A

6.9 Context Saving During Interrupts

During an interrupt, only the return PC value is saved on the stack. Typically, users wish to save key register values during an interrupt (e.g., W register and STATUS register). This is implemented in software.

The code in Example 6-1 stores and restores the STATUS and W register's values. The user defined registers, W_TEMP and STATUS_TEMP are the temporary storage locations for the W and STATUS registers values.

Example 6-1 does the following:

a) Stores the W register.
b) Stores the STATUS register in STATUS_TEMP.
c) Executes the Interrupt Service Routine code.
d) Restores the STATUS (and bank select bit) register.
e) Restores the W register.

EXAMPLE 6-1: SAVING STATUS AND W REGISTERS IN RAM

```
PUSH    MOVWF   W_TEMP              ; Copy W to TEMP register,
        SWAPF   STATUS,     W       ; Swap status to be saved into W
        MOVWF   STATUS_TEMP         ; Save status to STATUS_TEMP register
ISR     :                           :
        :                           ; Interrupt Service Routine
        :                           ; should configure Bank as required
        :                           ;
POP     SWAPF   STATUS_TEMP,W       ; Swap nibbles in STATUS_TEMP register
                                    ; and place result into W
        MOVWF   STATUS              ; Move W into STATUS register
                                    ; (sets bank to original state)
        SWAPF   W_TEMP,     F       ; Swap nibbles in W_TEMP and place result in W_TEMP
        SWAPF   W_TEMP,     W       ; Swap nibbles in W_TEMP and place result into W
```

6.10 Watchdog Timer (WDT)

The Watchdog Timer is a free running On-Chip RC Oscillator which does not require any external components. This RC oscillator is separate from the RC oscillator of the OSC1/CLKIN pin. That means that the WDT will run even if the clock on the OSC1/CLKIN and OSC2/CLKOUT pins of the device has been stopped, for example, by execution of a SLEEP instruction. During normal operation, a WDT time-out generates a device RESET. If the device is in SLEEP mode, a WDT wake-up causes the device to wake-up and continue with normal operation. The WDT can be permanently disabled by programming configuration bit WDTE as a '0' (Section 6.1).

6.10.1 WDT PERIOD

The WDT has a nominal time-out period of 18 ms, (with no prescaler). The time-out periods vary with temperature, VDD and process variations from part to part (see DC specs). If longer time-out periods are desired, a prescaler with a division ratio of up to 1:128 can be assigned to the WDT under software control by writing to the OPTION_REG register. Thus, time-out periods up to 2.3 seconds can be realized.

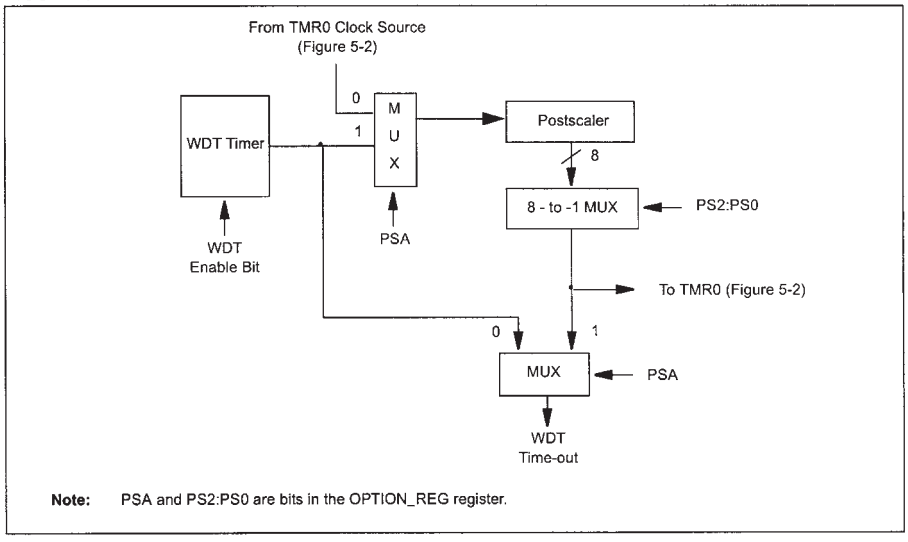
The CLRWDT and SLEEP instructions clear the WDT and the postscaler (if assigned to the WDT) and prevent it from timing out and generating a device RESET condition.

The $\overline{TO}$ bit in the STATUS register will be cleared upon a WDT time-out.

6.10.2 WDT PROGRAMMING CONSIDERATIONS

It should also be taken into account that under worst case conditions (V_{DD} = Min., Temperature = Max., Max. WDT Prescaler), it may take several seconds before a WDT time-out occurs.

FIGURE 6-11: WATCHDOG TIMER BLOCK DIAGRAM



Note: PSA and PS2:PS0 are bits in the OPTION_REG register.

TABLE 6-7: SUMMARY OF REGISTERS ASSOCIATED WITH THE WATCHDOG TIMER

Addr	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on Power-on Reset	Value on all other RESETS
2007h	Config. bits	(2)	(2)	(2)	(2)	PWRTE(1)	WDTE	FOSC1	FOSC0	(2)	
81h	OPTION_REG	RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	1111 1111

Legend: x = unknown. Shaded cells are not used by the WDT.

Note 1: See Register 6-1 for operation of the $\overline{PWRTE}$ bit.

2: See Register 6-1 and Section 6.12 for operation of the code and data protection bits.

PIC16F84A

6.11 Power-down Mode (SLEEP)

A device may be powered down (SLEEP) and later powered up (wake-up from SLEEP).

6.11.1 SLEEP

The Power-down mode is entered by executing the SLEEP instruction.

If enabled, the Watchdog Timer is cleared (but keeps running), the $\overline{PD}$ bit (STATUS<3>) is cleared, the $\overline{TO}$ bit (STATUS<4>) is set, and the oscillator driver is turned off. The I/O ports maintain the status they had before the SLEEP instruction was executed (driving high, low, or hi-impedance).

For the lowest current consumption in SLEEP mode, place all I/O pins at either VDD or VSS, with no external circuitry drawing current from the I/O pins, and disable external clocks. I/O pins that are hi-impedance inputs should be pulled high or low externally to avoid switching currents caused by floating inputs. The T0CKI input should also be at VDD or VSS. The contribution from on-chip pull-ups on PORTB should be considered.

The $\overline{MCLR}$ pin must be at a logic high level (VIHMC).

It should be noted that a RESET generated by a WDT time-out does not drive the $\overline{MCLR}$ pin low.

6.11.2 WAKE-UP FROM SLEEP

The device can wake-up from SLEEP through one of the following events:

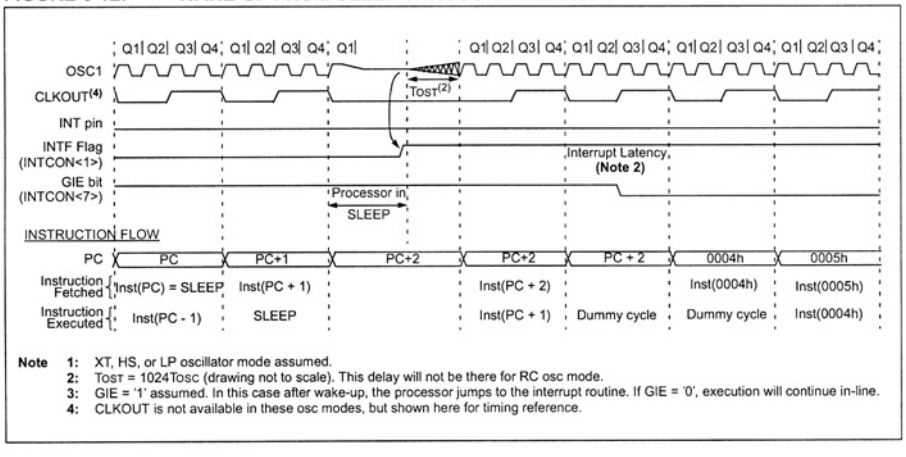
1. External RESET input on $\overline{MCLR}$ pin.
2. WDT wake-up (if WDT was enabled).
3. Interrupt from RB0/INT pin, RB port change, or data EEPROM write complete.

Peripherals cannot generate interrupts during SLEEP, since no on-chip Q clocks are present.

The first event ($\overline{MCLR}$ Reset) will cause a device RESET. The two latter events are considered a continuation of program execution. The $\overline{TO}$ and $\overline{PD}$ bits can be used to determine the cause of a device RESET. The $\overline{PD}$ bit, which is set on power-up, is cleared when SLEEP is invoked. The $\overline{TO}$ bit is cleared if a WDT time-out occurred (and caused wake-up).

While the SLEEP instruction is being executed, the next instruction (PC + 1) is pre-fetched. For the device to wake-up through an interrupt event, the corresponding interrupt enable bit must be set (enabled). Wake-up occurs regardless of the state of the GIE bit. If the GIE bit is clear (disabled), the device continues execution at the instruction after the SLEEP instruction. If the GIE bit is set (enabled), the device executes the instruction after the SLEEP instruction and then branches to the interrupt address (0004h). In cases where the execution of the instruction following SLEEP is not desirable, the user should have a NOP after the SLEEP instruction.

FIGURE 6-12: WAKE-UP FROM SLEEP THROUGH INTERRUPT



Note 1: XT, HS, or LP oscillator mode assumed.
2: TOST = 1024TOSC (drawing not to scale). This delay will not be there for RC osc mode.
3: GIE = '1' assumed. In this case after wake-up, the processor jumps to the interrupt routine. If GIE = '0', execution will continue in-line.
4: CLKOUT is not available in these osc modes, but shown here for timing reference.

6.11.3 WAKE-UP USING INTERRUPTS

When global interrupts are disabled (GIE cleared) and any interrupt source has both its interrupt enable bit and interrupt flag bit set, one of the following will occur:

- If the interrupt occurs **before** the execution of a SLEEP instruction, the SLEEP instruction will complete as a NOP. Therefore, the WDT and WDT postscaler will not be cleared, the $\overline{TO}$ bit will not be set and $\overline{PD}$ bits will not be cleared.
- If the interrupt occurs **during or after** the execution of a SLEEP instruction, the device will immediately wake-up from SLEEP. The SLEEP instruction will be completely executed before the wake-up. Therefore, the WDT and WDT postscaler will be cleared, the $\overline{TO}$ bit will be set and the $\overline{PD}$ bit will be cleared.

Even if the flag bits were checked before executing a SLEEP instruction, it may be possible for flag bits to become set before the SLEEP instruction completes. To determine whether a SLEEP instruction executed, test the $\overline{PD}$ bit. If the $\overline{PD}$ bit is set, the SLEEP instruction was executed as a NOP.

To ensure that the WDT is cleared, a CLRWDT instruction should be executed before a SLEEP instruction.

6.12 Program Verification/Code Protection

If the code protection bit(s) have not been programmed, the on-chip program memory can be read out for verification purposes.

6.13 ID Locations

Four memory locations (2000h - 2004h) are designated as ID locations to store checksum or other code identification numbers. These locations are not accessible during normal execution but are readable and writable only during program/verify. Only the four Least Significant bits of ID location are usable.

6.14 In-Circuit Serial Programming

PIC16F84A microcontrollers can be serially programmed while in the end application circuit. This is simply done with two lines for clock and data, and three other lines for power, ground, and the programming voltage. Customers can manufacture boards with unprogrammed devices, and then program the microcontroller just before shipping the product, allowing the most recent firmware or custom firmware to be programmed.

For complete details of Serial Programming, please refer to the In-Circuit Serial Programming™ (ICSP™) Guide, (DS30277).

PIC16F84A

NOTES:

PIC16F84A

7.0 INSTRUCTION SET SUMMARY

Each PIC16CXX instruction is a 14-bit word, divided into an OPCODE which specifies the instruction type and one or more operands which further specify the operation of the instruction. The PIC16CXX instruction set summary in Table 7-2 lists **byte-oriented**, **bit-oriented**, and **literal and control** operations. Table 7-1 shows the opcode field descriptions.

For **byte-oriented** instructions, 'f' represents a file register designator and 'd' represents a destination designator. The file register designator specifies which file register is to be used by the instruction.

The destination designator specifies where the result of the operation is to be placed. If 'd' is zero, the result is placed in the W register. If 'd' is one, the result is placed in the file register specified in the instruction.

For **bit-oriented** instructions, 'b' represents a bit field designator which selects the number of the bit affected by the operation, while 'f' represents the address of the file in which the bit is located.

For **literal and control** operations, 'k' represents an eight or eleven bit constant or literal value.

TABLE 7-1: OPCODE FIELD DESCRIPTIONS

Field	Description
f	Register file address (0x00 to 0x7F)
W	Working register (accumulator)
b	Bit address within an 8-bit file register
k	Literal field, constant data or label
x	Don't care location (= 0 or 1) The assembler will generate code with x = 0. It is the recommended form of use for compatibility with all Microchip software tools.
d	Destination select; d = 0: store result in W, d = 1: store result in file register f. Default is d = 1
PC	Program Counter
TO	Time-out bit
PD	Power-down bit

The instruction set is highly orthogonal and is grouped into three basic categories:

- **Byte-oriented** operations
- **Bit-oriented** operations
- **Literal and control** operations

All instructions are executed within one single instruction cycle, unless a conditional test is true or the program counter is changed as a result of an instruction. In this case, the execution takes two instruction cycles with the second cycle executed as a NOP. One instruction cycle consists of four oscillator periods. Thus, for an oscillator frequency of 4 MHz, the normal instruction execution time is 1 µs. If a conditional test is true or the program counter is changed as a result of an instruction, the instruction execution time is 2 µs.

Table 7-2 lists the instructions recognized by the MPASM™ Assembler.

Figure 7-1 shows the general formats that the instructions can have.

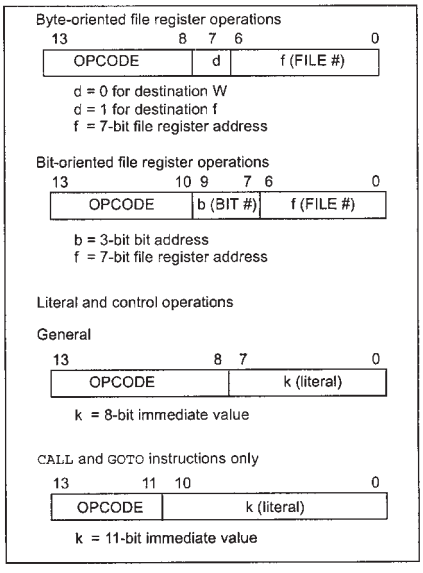
Note: To maintain upward compatibility with future PIC16CXX products, do not use the OPTION and TRIS instructions.

All examples use the following format to represent a hexadecimal number:

0xhh

where h signifies a hexadecimal digit.

FIGURE 7-1: GENERAL FORMAT FOR INSTRUCTIONS



A description of each instruction is available in the PICmicro™ Mid-Range Reference Manual (DS33023).

PIC16F84A

TABLE 7-2: PIC16CXXX INSTRUCTION SET

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb	LSb	Status Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS							
ADDWF	f, d	Add W and f	1	00 0111 dfff ffff		C,DC,Z	1,2
ANDWF	f, d	AND W with f	1	00 0101 dfff ffff		Z	1,2
CLRF	f	Clear f	1	00 0001 lfff ffff		Z	2
CLRW	-	Clear W	1	00 0001 0xxx xxxx		Z	
COMF	f, d	Complement f	1	00 1001 dfff ffff		Z	1,2
DECF	f, d	Decrement f	1	00 0011 dfff ffff		Z	1,2
DECFSZ	f, d	Decrement f, Skip if 0	1 (2)	00 1011 dfff ffff			1,2,3
INCF	f, d	Increment f	1	00 1010 dfff ffff		Z	1,2
INCFSZ	f, d	Increment f, Skip if 0	1 (2)	00 1111 dfff ffff			1,2,3
IORWF	f, d	Inclusive OR W with f	1	00 0100 dfff ffff		Z	1,2
MOVF	f, d	Move f	1	00 1000 dfff ffff		Z	1,2
MOVWF	f	Move W to f	1	00 0000 lfff ffff			
NOP	-	No Operation	1	00 0000 0xx0 0000			
RLF	f, d	Rotate Left f through Carry	1	00 1101 dfff ffff		C	1,2
RRF	f, d	Rotate Right f through Carry	1	00 1100 dfff ffff		C	1,2
SUBWF	f, d	Subtract W from f	1	00 0010 dfff ffff		C,DC,Z	1,2
SWAPF	f, d	Swap nibbles in f	1	00 1110 dfff ffff			1,2
XORWF	f, d	Exclusive OR W with f	1	00 0110 dfff ffff		Z	1,2
BIT-ORIENTED FILE REGISTER OPERATIONS							
BCF	f, b	Bit Clear f	1	01 00bb bfff ffff			1,2
BSF	f, b	Bit Set f	1	01 01bb bfff ffff			1,2
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)	01 10bb bfff ffff			3
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)	01 11bb bfff ffff			3
LITERAL AND CONTROL OPERATIONS							
ADDLW	k	Add literal and W	1	11 111x kkkk kkkk		C,DC,Z	
ANDLW	k	AND literal with W	1	11 1001 kkkk kkkk		Z	
CALL	k	Call subroutine	2	10 0kkk kkkk kkkk			
CLRWDT	-	Clear Watchdog Timer	1	00 0000 0110 0100		$\overline{TO}$,$\overline{PD}$	
GOTO	k	Go to address	2	10 1kkk kkkk kkkk			
IORLW	k	Inclusive OR literal with W	1	11 1000 kkkk kkkk		Z	
MOVLW	k	Move literal to W	1	11 00xx kkkk kkkk			
RETFIE	-	Return from interrupt	2	00 0000 0000 1001			
RETLW	k	Return with literal in W	2	11 01xx kkkk kkkk			
RETURN	-	Return from Subroutine	2	00 0000 0000 1000			
SLEEP	-	Go into standby mode	1	00 0000 0110 0011		$\overline{TO}$,$\overline{PD}$	
SUBLW	k	Subtract W from literal	1	11 110x kkkk kkkk		C,DC,Z	
XORLW	k	Exclusive OR literal with W	1	11 1010 kkkk kkkk		Z	

Note 1: When an I/O register is modified as a function of itself (e.g., `MOVF PORTB, 1`), the value used will be that value present on the pins themselves. For example, if the data latch is '1' for a pin configured as input and is driven low by an external device, the data will be written back with a '0'.

2: If this instruction is executed on the TMR0 register (and, where applicable, d = 1), the prescaler will be cleared if assigned to the Timer0 Module.

3: If Program Counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a `NOP`.

Note: Additional information on the mid-range instruction set is available in the PICmicro™ Mid-Range MCU Family Reference Manual (DS33023).

7.1 Instruction Descriptions

ADDLW	Add Literal and W
Syntax:	[*label*] ADDLW k
Operands:	0 ≤ k ≤ 255
Operation:	(W) + k → (W)
Status Affected:	C, DC, Z
Description:	The contents of the W register are added to the eight-bit literal 'k' and the result is placed in the W register.

ADDWF	Add W and f
Syntax:	[*label*] ADDWF f,d
Operands:	0 ≤ f ≤ 127 d ∈ [0,1]
Operation:	(W) + (f) → (destination)
Status Affected:	C, DC, Z
Description:	Add the contents of the W register with register 'f'. If 'd' is 0, the result is stored in the W register. If 'd' is 1, the result is stored back in register 'f'.

ANDLW	AND Literal with W
Syntax:	[*label*] ANDLW k
Operands:	0 ≤ k ≤ 255
Operation:	(W) .AND. (k) → (W)
Status Affected:	Z
Description:	The contents of W register are AND'ed with the eight-bit literal 'k'. The result is placed in the W register.

ANDWF	AND W with f
Syntax:	[*label*] ANDWF f,d
Operands:	0 ≤ f ≤ 127 d ∈ [0,1]
Operation:	(W) .AND. (f) → (destination)
Status Affected:	Z
Description:	AND the W register with register 'f'. If 'd' is 0, the result is stored in the W register. If 'd' is 1, the result is stored back in register 'f'.

BCF	Bit Clear f
Syntax:	[*label*] BCF f,b
Operands:	0 ≤ f ≤ 127 0 ≤ b ≤ 7
Operation:	0 → (f<b>)
Status Affected:	None
Description:	Bit 'b' in register 'f' is cleared.

BSF	Bit Set f
Syntax:	[*label*] BSF f,b
Operands:	0 ≤ f ≤ 127 0 ≤ b ≤ 7
Operation:	1 → (f<b>)
Status Affected:	None
Description:	Bit 'b' in register 'f' is set.

BTFSS	Bit Test f, Skip if Set
Syntax:	[*label*] BTFSS f,b
Operands:	0 ≤ f ≤ 127 0 ≤ b < 7
Operation:	skip if (f<b>) = 1
Status Affected:	None
Description:	If bit 'b' in register 'f' is '0', the next instruction is executed. If bit 'b' is '1', then the next instruction is discarded and a NOP is executed instead, making this a 2TCY instruction.

PIC16F84A

BTFSC	Bit Test, Skip if Clear
Syntax:	[*label*] BTFSC f,b
Operands:	0 ≤ f ≤ 127 0 ≤ b ≤ 7
Operation:	skip if (f<b>) = 0
Status Affected:	None
Description:	If bit 'b' in register 'f' is '1', the next instruction is executed. If bit 'b' in register 'f' is '0', the next instruction is discarded, and a NOP is executed instead, making this a 2TCY instruction.

CALL	Call Subroutine
Syntax:	[*label*] CALL k
Operands:	0 ≤ k ≤ 2047
Operation:	(PC)+ 1→ TOS, k → PC<10:0>, (PCLATH<4:3>) → PC<12:11>
Status Affected:	None
Description:	Call Subroutine. First, return address (PC+1) is pushed onto the stack. The eleven-bit immediate address is loaded into PC bits <10:0>. The upper bits of the PC are loaded from PCLATH. CALL is a two-cycle instruction.

CLRF	Clear f
Syntax:	[*label*] CLRF f
Operands:	0 ≤ f ≤ 127
Operation:	00h → (f) 1 → Z
Status Affected:	Z
Description:	The contents of register 'f' are cleared and the Z bit is set.

CLRW	Clear W
Syntax:	[*label*] CLRW
Operands:	None
Operation:	00h → (W) 1 → Z
Status Affected:	Z
Description:	W register is cleared. Zero bit (Z) is set.

CLRWDT	Clear Watchdog Timer
Syntax:	[*label*] CLRWDT
Operands:	None
Operation:	00h → WDT 0 → WDT prescaler, 1 → $\overline{TO}$ 1 → $\overline{PD}$
Status Affected:	$\overline{TO}$, $\overline{PD}$
Description:	CLRWDT instruction resets the Watchdog Timer. It also resets the prescaler of the WDT. Status bits $\overline{TO}$ and $\overline{PD}$ are set.

COMF	Complement f
Syntax:	[*label*] COMF f,d
Operands:	0 ≤ f ≤ 127 d ∈ [0,1]
Operation:	($\bar{f}$) → (destination)
Status Affected:	Z
Description:	The contents of register 'f' are complemented. If 'd' is 0, the result is stored in W. If 'd' is 1, the result is stored back in register 'f'.

DECF	Decrement f
Syntax:	[*label*] DECF f,d
Operands:	0 ≤ f ≤ 127 d ∈ [0,1]
Operation:	(f) - 1 → (destination)
Status Affected:	Z
Description:	Decrement register 'f'. If 'd' is 0, the result is stored in the W register. If 'd' is 1, the result is stored back in register 'f'.

DECFSZ	Decrement f, Skip if 0
Syntax:	[*label*] DECFSZ f,d
Operands:	0 ≤ f ≤ 127 d ∈ [0,1]
Operation:	(f) - 1 → (destination); skip if result = 0
Status Affected:	None
Description:	The contents of register 'f' are decremented. If 'd' is 0, the result is placed in the W register. If 'd' is 1, the result is placed back in register 'f'. If the result is 1, the next instruction is executed. If the result is 0, then a NOP is executed instead, making it a 2TCY instruction.

GOTO	Unconditional Branch
Syntax:	[*label*] GOTO k
Operands:	0 ≤ k ≤ 2047
Operation:	k → PC<10:0> PCLATH<4:3> → PC<12:11>
Status Affected:	None
Description:	GOTO is an unconditional branch. The eleven-bit immediate value is loaded into PC bits <10:0>. The upper bits of PC are loaded from PCLATH<4:3>. GOTO is a two-cycle instruction.

INCF	Increment f
Syntax:	[*label*] INCF f,d
Operands:	0 ≤ f ≤ 127 d ∈ [0,1]
Operation:	(f) + 1 → (destination)
Status Affected:	Z
Description:	The contents of register 'f' are incremented. If 'd' is 0, the result is placed in the W register. If 'd' is 1, the result is placed back in register 'f'.

INCFSZ	Increment f, Skip if 0
Syntax:	[*label*] INCFSZ f,d
Operands:	0 ≤ f ≤ 127 d ∈ [0,1]
Operation:	(f) + 1 → (destination), skip if result = 0
Status Affected:	None
Description:	The contents of register 'f' are incremented. If 'd' is 0, the result is placed in the W register. If 'd' is 1, the result is placed back in register 'f'. If the result is 1, the next instruction is executed. If the result is 0, a NOP is executed instead, making it a 2TCY instruction.

IORLW	Inclusive OR Literal with W
Syntax:	[*label*] IORLW k
Operands:	0 ≤ k ≤ 255
Operation:	(W) .OR. k → (W)
Status Affected:	Z
Description:	The contents of the W register are OR'ed with the eight-bit literal 'k'. The result is placed in the W register.

IORWF	Inclusive OR W with f
Syntax:	[*label*] IORWF f,d
Operands:	0 ≤ f ≤ 127 d ∈ [0,1]
Operation:	(W) .OR. (f) → (destination)
Status Affected:	Z
Description:	Inclusive OR the W register with register 'f'. If 'd' is 0, the result is placed in the W register. If 'd' is 1, the result is placed back in register 'f'.

PIC16F84A

MOVF	Move f
Syntax:	[*label*] MOVF f,d
Operands:	0 ≤ f ≤ 127 d ∈ [0,1]
Operation:	(f) → (destination)
Status Affected:	Z
Description:	The contents of register f are moved to a destination dependant upon the status of d. If d = 0, destination is W register. If d = 1, the destination is file register f itself. d = 1 is useful to test a file register, since status flag Z is affected.

MOVLW	Move Literal to W
Syntax:	[*label*] MOVLW k
Operands:	0 ≤ k ≤ 255
Operation:	k → (W)
Status Affected:	None
Description:	The eight-bit literal 'k' is loaded into W register. The don't cares will assemble as 0's.

MOVWF	Move W to f
Syntax:	[*label*] MOVWF f
Operands:	0 ≤ f ≤ 127
Operation:	(W) → (f)
Status Affected:	None
Description:	Move data from W register to register 'f'.

NOP	No Operation
Syntax:	[*label*] NOP
Operands:	None
Operation:	No operation
Status Affected:	None
Description:	No operation.

RETFIE	Return from Interrupt
Syntax:	[*label*] RETFIE
Operands:	None
Operation:	TOS → PC, 1 → GIE
Status Affected:	None

RETLW	Return with Literal in W
Syntax:	[*label*] RETLW k
Operands:	0 ≤ k ≤ 255
Operation:	k → (W); TOS → PC
Status Affected:	None
Description:	The W register is loaded with the eight-bit literal 'k'. The program counter is loaded from the top of the stack (the return address). This is a two-cycle instruction.

RETURN	Return from Subroutine
Syntax:	[*label*] RETURN
Operands:	None
Operation:	TOS → PC
Status Affected:	None
Description:	Return from subroutine. The stack is POPed and the top of the stack (TOS) is loaded into the program counter. This is a two-cycle instruction.

RLF **Rotate Left f through Carry**

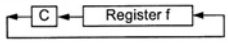
Syntax: [*label*] RLF f,d

Operands: 0 ≤ f ≤ 127
d ∈ [0,1]

Operation: See description below

Status Affected: C

Description: The contents of register 'f' are rotated one bit to the left through the Carry Flag. If 'd' is 0, the result is placed in the W register. If 'd' is 1, the result is stored back in register 'f'.



RRF **Rotate Right f through Carry**

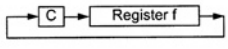
Syntax: [*label*] RRF f,d

Operands: 0 ≤ f ≤ 127
d ∈ [0,1]

Operation: See description below

Status Affected: C

Description: The contents of register 'f' are rotated one bit to the right through the Carry Flag. If 'd' is 0, the result is placed in the W register. If 'd' is 1, the result is placed back in register 'f'.



SLEEP

Syntax: [*label*] SLEEP

Operands: None

Operation: 00h → WDT,
0 → WDT prescaler,
1 → $\overline{TO}$,
0 → $\overline{PD}$

Status Affected: $\overline{TO}$, $\overline{PD}$

Description: The power-down status bit, $\overline{PD}$ is cleared. Time-out status bit, $\overline{TO}$ is set. Watchdog Timer and its prescaler are cleared.
The processor is put into SLEEP mode with the oscillator stopped.

SUBLW **Subtract W from Literal**

Syntax: [*label*] SUBLW k

Operands: 0 ≤ k ≤ 255

Operation: k - (W) → (W)

Status Affected: C, DC, Z

Description: The W register is subtracted (2's complement method) from the eight-bit literal 'k'. The result is placed in the W register.

SUBWF **Subtract W from f**

Syntax: [*label*] SUBWF f,d

Operands: 0 ≤ f ≤ 127
d ∈ [0,1]

Operation: (f) - (W) → (destination)

Status Affected: C, DC, Z

Description: Subtract (2's complement method) W register from register 'f'. If 'd' is 0, the result is stored in the W register. If 'd' is 1, the result is stored back in register 'f'.

SWAPF **Swap Nibbles in f**

Syntax: [*label*] SWAPF f,d

Operands: 0 ≤ f ≤ 127
d ∈ [0,1]

Operation: (f<3:0>) → (destination<7:4>),
(f<7:4>) → (destination<3:0>)

Status Affected: None

Description: The upper and lower nibbles of register 'f' are exchanged. If 'd' is 0, the result is placed in W register. If 'd' is 1, the result is placed in register 'f'.

PIC16F84A

XORLW	Exclusive OR Literal with W
Syntax:	[*label*] XORLW k
Operands:	$0 \le k \le 255$
Operation:	(W) .XOR. k → (W)
Status Affected:	Z
Description:	The contents of the W register are XOR'ed with the eight-bit literal 'k'. The result is placed in the W register.

XORWF	Exclusive OR W with f
Syntax:	[*label*] XORWF f,d
Operands:	$0 \le f \le 127$ $d \in [0,1]$
Operation:	(W) .XOR. (f) → (destination)
Status Affected:	Z
Description:	Exclusive OR the contents of the W register with register 'f'. If 'd' is 0, the result is stored in the W register. If 'd' is 1, the result is stored back in register 'f'.

Index

Q

R

S