PIC Assembly Language for the Complete Beginner

Michael A. Covington

Artificial Intelligence Center

The University of Georgia

Athens, Georgia 30602-7415

http://www.ai.uga.edu/mc

This article appeared in *Electronics Now* Magazine in 1999 and is reprinted here by permission. Some web addresses have been updated but the content has not; you will find that MPLAB, for instance, now looks somewhat different.

You may print out this article for personal use but not for further publication.

Copyright © 1999 Gernsback Publications, Inc.
Copyright © 1999, 2004 Michael A. Covington.

These days, the field of electronics is divided into "haves" and "have-nots" – people who can program microcontrollers and people who can't. If you're one of the "have-nots," this article is for you.

Microcontrollers are one-chip computers designed to control other equipment, and almost all electronic equipment now uses them. The average American home now contains about 100 computers, almost all of which are microcontrollers hidden within appliances, clocks, thermostats, and even automobile engines.

Although some microcontrollers can be programmed in C or BASIC, you need assembly language to get the best results with the least expensive micros. The reason is that assembly language lets you specify the exact instructions that the CPU will follow; you can control exactly how much time and memory each step of the program will take. On a tiny computer, this can be important. What's more, if you're not already an experienced programmer, you may well find that assembly language is *simpler* than BASIC or C. In many ways it's more like designing a circuit than writing software.

The trouble with assembly language is that it's different for each kind of CPU. There's one assembly language for Pentiums, another for PIC microcontrollers, still another for Motorola 68000s, and so forth. There are even slight differences from one model of PIC to another. And that leads to a serious problem – each assembly-language manual seems to assume that you already know the assembly language for some other processor! So as you look from one manual to another in puzzlement, *there's no way to get started.*

That's the problem this article will address. I won't teach you all of PIC assembly language, just enough to get you started. For concreteness, I'll use just one processor, the PIC16F84. To be very precise, I'll use the

PIC16F84-04P, which operates up to 4 MHz and is housed in a plastic DIP package.[1] This is a product of Microchip, Inc. (Chandler, Arizona), and it's closely related to the rest of the PIC family – which, however, I'll ignore to prevent confusion.

To do the experiments described in this article, you'll need one or more PIC16F84-04P chips; we strongly recommend having more than one so you can rule out a damaged PIC if your circuit doesn't work. You'll also need the other parts for the circuits you want to build (see the schematics). And you'll need a PC-compatible personal computer, the MPASM assembler software (which you can download from http://www.microchip.com), and a PIC programmer such as Ramsey Electronics' "PICPRO-1" or the NOPPP programmer published in this magazine, September 1998, and described at http://www.covingtoninnovations.com/noppp. The PIC16F8X data sheet, actually a 122-page manual, will also come in handy; it's called PIC16F8X because it covers both PIC16F84 and PIC14F83, and you can download it or request a printed copy from Microchip.

1 PART 1 - MEET THE PIC

1.1 What's inside a PIC?

Figure 1 shows the pinout of the PIC16F84, and Figure 2 shows the most important parts inside. The PIC is a tiny but complete computer. It has a CPU (central processing unit), program memory (PROM), working mem-

[1] Note added 2004: The 10-MHz version is now more common and will work in all the same circuits.

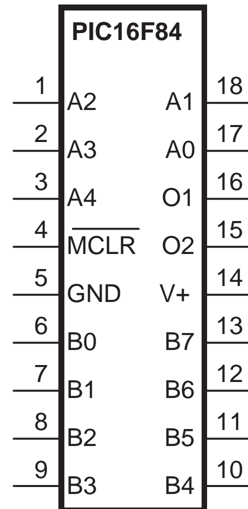

Figure 1: Pinout of PIC16F84.

ory (RAM), and two input-output ports.

The CPU is, of course, the "brain" of the computer. It reads and executes instructions from the program memory. As it does so, it can store and retrieve data in working memory (RAM). Some CPUs make a distinction between "registers" located within the CPU and "RAM" located outside it; the PIC doesn't, and its general-purpose working RAM is also known as "file registers." On the 'F84, there are 68 bytes of general-purpose RAM, located at addresses hex 0C to hex 4F.

Besides the general-purpose memory, there is a special "working register" or "W register" where the CPU holds the data it's working on. There are also several special-function registers each of which controls the oper-

ation of the PIC in some way.

The program memory of the ’F84 consists of flash EPROM; it can be recorded and erased electrically, and it retains its contents when powered off. Many other PICs require ultraviolet light for erasure and are not erasable if you buy the cheaper version without the quartz window. The ’F84, however, is always erasable and reprogrammable.

There are two input-output ports, port A and port B, and each pin of each port can be set individually as an input or an output. The bits of each port are numbered, starting at 0. In output mode, bit 4 of port A has an open collector (or rather open drain); the rest of the outputs are regular CMOS. (Working with microcontrollers, you have to remember details like this; there’s no programming language or operating system to hide the details of the hardware from you.) The CPU treats each port as one 8-bit byte of data even though only five bits of port A are actually brought out as pins of the IC.

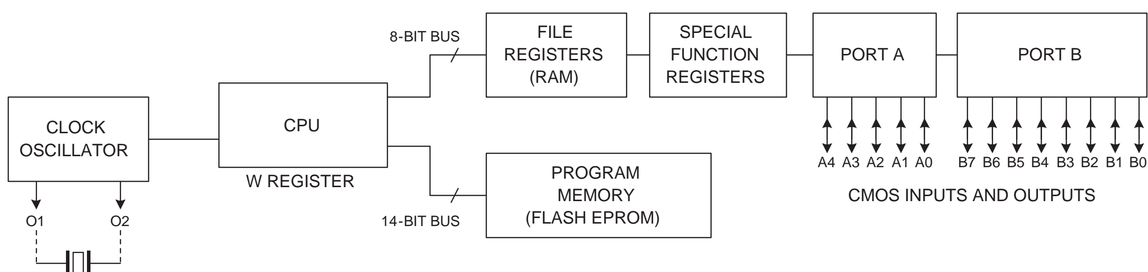

Figure 2: Main components of the PIC16F84.

PIC inputs are CMOS-compatible; PIC outputs can drive TTL or CMOS logic chips. Each output pin can source or sink 20 mA as long as only one pin is doing so at a time. Further information about electrical limits is given in the PIC16F84 data sheet.

The 'F84 also has some features we won't be using, including an EEPROM for long-term storage of data, an onboard timer-counter module, and optional pull-up resistors on port B.

1.2 Power and clock requirements

The PIC16F84 requires a 5-volt supply; actually, any voltage from 4.0 to 6.0 volts will do fine, so you can run it from three 1.5-volt cells. Figure 3 shows several power-supply options. The PIC consumes only 1 mA – even less, at low clock speeds – but the power supply must also provide the current flowing through LEDs or other high-current devices that the PIC may be driving. Thus, the last circuit, with the Zener diode, is only for PICs that aren't driving LEDs.

All four power supply circuits rely on a 0.1-μF capacitor from pin 14 (V+) to ground, mounted close to the PIC, to protect the PIC and adjacent components from electrical noise. This capacitor should be present no matter how clean you think your DC supply is.

The MCLR pin is normally connected to V+ through a 10k resistor. Grounding it momentarily will clear RAM and reset the PIC. If your power supply voltage comes up slowly, the PIC may start up in a confused state; in that case you should add a normally-open reset button from MCLR to ground.

Corrected 2021

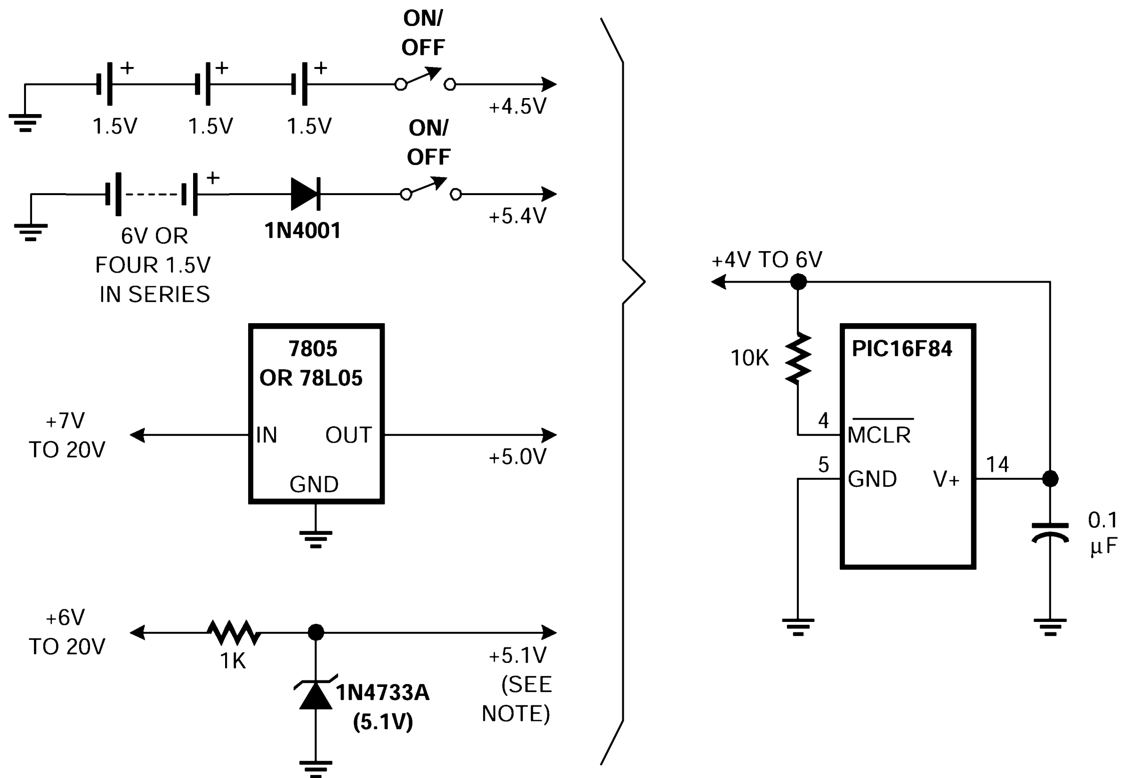

Figure 3: Some ways to power a PIC. The last one is only for a PIC that is not powering an LED or other high-current load.

Like any CPU, the PIC needs a clock – an oscillator to control the speed of the CPU and step it through its operations. The maximum clock frequency of the PIC16F84-04P is, as already noted, 4 MHz. There is no lower limit. Low clock frequencies save power and reduce the amount of counting the PIC has to do when timing a slow operation. At 30 kHz, a PIC can run on 0.1 mA.

Figure 4 shows the most popular clock circuits. The clock signal can be fed in from an external source, or you can use the PIC's on-board oscillator with either a crystal or a resistor and capacitor. Crystals are preferred for high accuracy; 3.58-MHz crystals, mass-produced for color TV circuits, work well and are very cheap. The resistor-capacitor oscillator is cheaper yet, but the frequency is somewhat unpredictable; don't use it if your circuit needs to keep time accurately.

2 PART 2 - YOUR FIRST PROGRAM

2.1 Assembly language

A PIC spends its time reading instructions from the program memory, one after another, and doing whatever these instructions say. Each instruction consists of 14 bits. If you could see the bits as binary ones and zeroes, the program in Figure 5 would look like this:

```
11000000000000
00000001100110
11000000000001
```

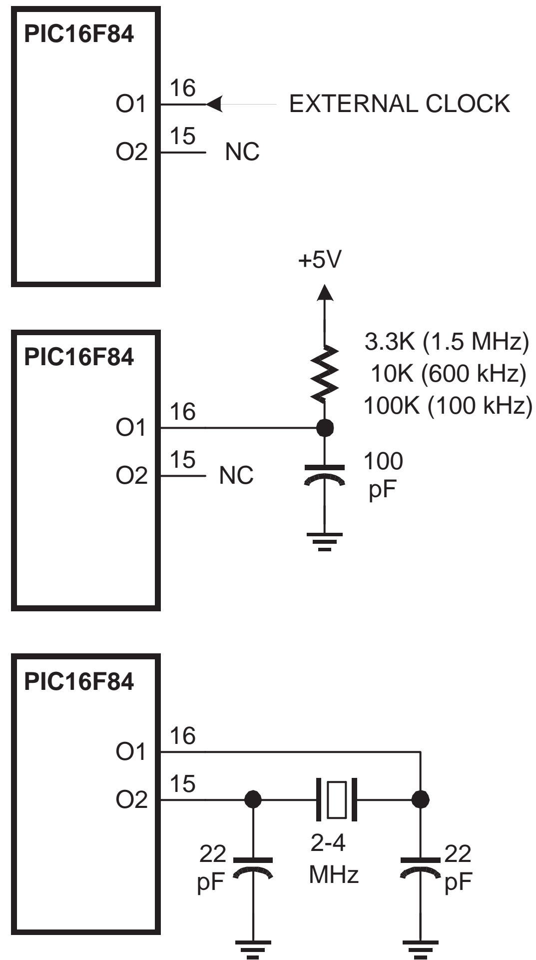

Figure 4: Three ways to provide the clock signal to a PIC.

```
; File TURNON.ASM
; Assembly code for PIC16F84 microcontroller

; Turns on an LED connected to B0.
; Uses RC oscillator, about 100 kHz.

; CPU configuration
;       (It's a 16F84, RC oscillator,
;       watchdog timer off, power-up timer on.)

        processor 16f84
        include   <p16f84.inc>
        __config  _RC_OSC & _WDT_OFF & _PWRTE_ON

; Program

        org     0       ; start at address 0

        ; At startup, all ports are inputs.
        ; Set Port B to all outputs.

        movlw   B'00000000'     ; w := binary 00000000
        tris    PORTB           ; copy w to port B control reg

        ; Put a 1 in the lowest bit of port B.

        movlw   B'00000001'     ; w := binary 00000001
        movwf   PORTB           ; copy w to port B itself

        ; Stop by going into an endless loop

fin:    goto    fin

        end                     ; program ends here
```

Figure 5: A complete PIC assembly-language program.

```
00000010000110
10100000000100
```

The earliest computers were programmed by technicians writing binary codes just like this. As you can see, though, binary codes are very hard for human beings to read or write because they're completely arbitrary; they look like gibberish.

Another reason binary codes are hard to write is that many of them refer to locations in memory. For instance, a "go to" instruction will have to say what memory address to jump to. Programming would be much easier if you could label a location in the program and have the computer figure out its address.

For both of these reasons, *assembly language* was invented over forty years ago. Or, to be more precise, many assembly languages have been invented, one for each type of CPU. What assembly languages have in common is that the instructions are abbreviated by readable codes (*mnemonics*) such as `GOTO` and locations can be represented by programmer-assigned labels. For example, in assembly language, the binary instructions just mentioned would be:

```
      movlw  B'00000000'
      tris   PORTB
      movlw  B'00000001'
      movwf  PORTB
fin:  goto   fin
```

In English: Put the bit pattern 00000000 into the W register and copy it to the tri-state control register for port B, thereby setting up port B for output;

then put 00000001 into W and copy it to port B itself; and finally stop the program by going into an endless loop.

2.2 Program layout

Figure 5 shows a complete, ready-to-assemble program. Look closely at its layout. The semicolon (`;`) is the comment marker; the computer ignores everything after the semicolon on each line. Much of the program consists of comments; that's as it should be, because although it's not as bad as binary code, assembly language is still relatively hard to read.

Each instruction is divided into three parts, the *label*, the *opcode* (operation code or instruction code), and the *operand* (also called *argument*). For example, in the line

```
fin:  goto   fin
```

the label is `fin:` (with a colon), the opcode is `goto`, and the operand is `fin`.

The label, opcode, and operand are separated by spaces. The assembler doesn't care how many spaces you use; one is enough, but most programmers use additional spaces to make their instructions line up into neat columns.

If there's no label, there must be at least one blank before the opcode, or the assembler will think the opcode is a label. Although current PIC assemblers can often recover from this kind of error, it is an error, and other assemblers aren't so tolerant.

2.3 Assembling a program

A computer "assembles" the assembly-language program into the binary instructions, which, for brevity, are actually written in hexadecimal (more about this shortly) and stored on what is called a .HEX file. Some computers run their own assemblers, but the PIC is far too small for that; instead, you'll type and assemble your PIC programs on a DOS or Windows PC. Then you'll download the .HEX file into a PIC using a PIC programmer and its associated software.

The program in Figure 5 does one very simple thing – it turns on an LED connected to pin B0. Figure 6 shows the circuit needed to try this program out. You can also use the circuit in Figure 10 or the demonstration board included with the Ramsey Electronics PICPRO-1. Admittedly, turning on one LED is not a great feat of computation, but it's enough to show that the PIC works.

To assemble this program, you'll need MPASM, the free PIC assembler downloadable from http://www.microchip.com. You also need the file P16F84.INC, which comes with MPASM and tells the assembler the particulars of the 'F84 as opposed to the numerous other varieties of PIC. You won't need the other .INC files also included with the assembler.

What you do is type your program onto a file with a name ending in .ASM, using Windows Notepad, DOS EDIT, or any other text editor. Don't use a word processor unless you are sure you can save your file as plain ASCII.

Then run MPASM from a DOS prompt (a DOS box under Windows is OK). If your program file is named `turnon.asm`, type the command

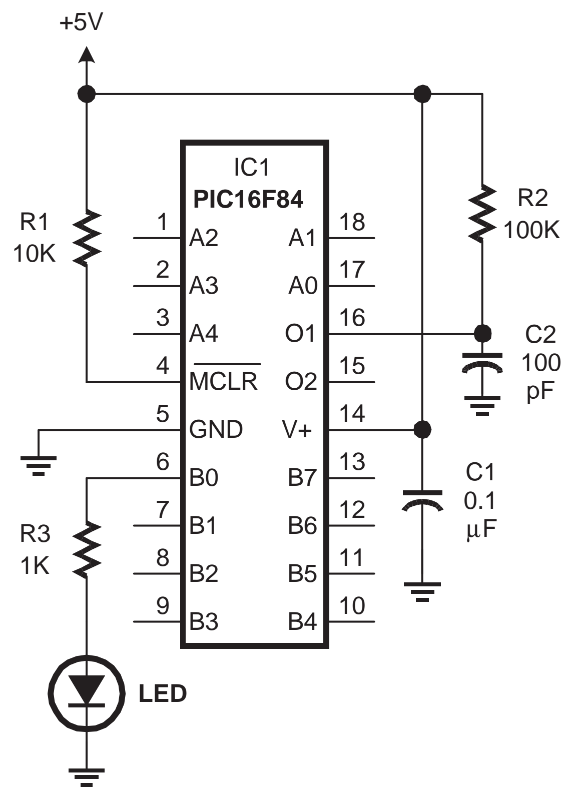

Figure 6: Circuit for simple program that turns on an LED (Fig. 5).

```
MS-DOS Prompt - MPASM
8 x 12
MPASM 02.15.06 Intermediate (c)1993-98 Microchip Technology Inc./Byte Craft Limi
Checking E:\PIC\TURNON.ASM for symbols...
Assembling...
TURNON.ASM    50
Building files...

Errors          :     0
Warnings        :     1 reported,     0 suppressed
Messages        :     0 reported,     0 suppressed
Lines Assembled :   186

Press any key to continue.
```

Figure 7: Running MPASM, the PIC assembler.

```
mpasm turnon.asm
```

and Figure 7 shows what you'll see on the screen.

What MPASM is telling you is that it assembled your .ASM file, generating one warning message (which is unimportant – more about this later) and no error messages. The results consists of a .HEX file containing the assembled instructions and a .LST file containing a detailed program listing with error messages. If the program contained serious errors, no .HEX file would be generated and you should study the .LST file to see what went wrong.

MPASM is the simple way to go. Microchip also gives away a development environment called MPLAB (Figure 8) that contains an assembler plus a simulator so you can make your PC pretend to be a PIC and actually see your program run. MPLAB is very useful but its operation is beyond the scope of this article. See http://www.covingtoninnovations.com/noppp for some tips.

Now that you have a .HEX file, you have to get it into the PIC. This is done with a programmer such as Microchip's "Picstart Plus" or the NOPPP programmer featured in *Electronics Now,* September 1998, and now marketed by Ramsey Electronics as PICPRO-1. On your PC, you run whatever software your programmer requires and follow the instructions.

Finally, put the programmed PIC into the circuit (handling it carefully to prevent static damage) and apply 5 volts. The LED should turn on. There – you've made a PIC do something.

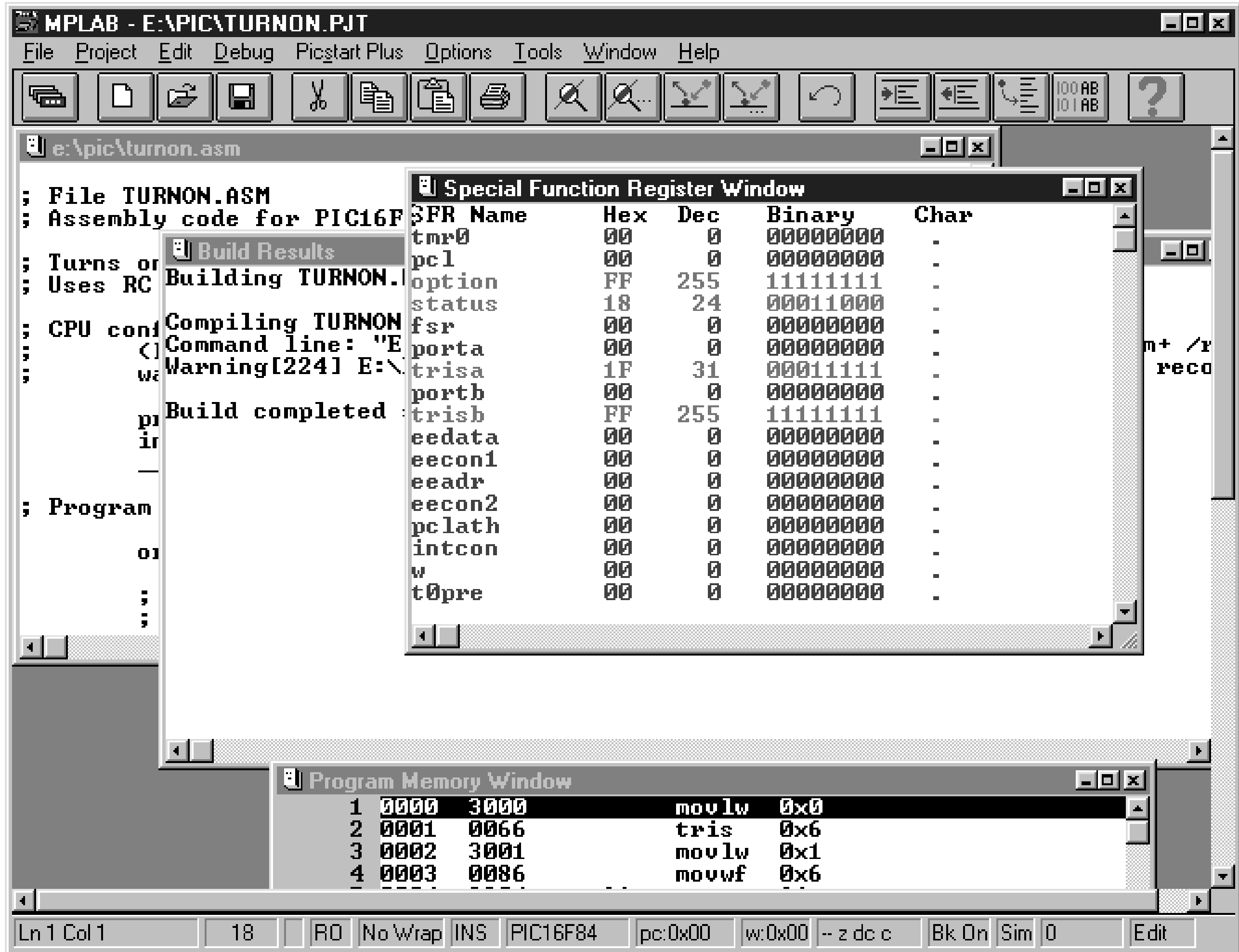

Figure 8: MPLAB, the PIC development environment.

2.4 How the program works

Now look back at Figure 5 and consider the program in detail. More than half of what you see there consists of comments; everything after the first semicolon (`;`) in each line is a comment ignored by the computer.

The program starts with a number of *pseudo instructions,* commands that give information to the assembler but are not translated into machine instructions. The first pseudo instruction,

```
processor PIC16F84
```

tells the assembler what kind of CPU you're using. It's immediately followed by another instruction,

```
include <p16f84.inc>
```

telling the assembler to read the file P16F84.INC, which contains a lot more pseudo instructions giving the memory addresses of the ports and other particulars of this CPU.

Next comes the `__config` macro instruction:

```
__config _RC_OSC & _WDT_OFF & _PWRTE_ON
```

Notice that there are two underscore marks at the beginning of `__config`. This instruction specifies some configuration settings to be programmed into the PIC. It says you're using an RC oscillator (resistor and capacitor, not crystal); the watchdog timer is off; and the power-up timer is on. The power-up timer imposes a slight delay at startup to give the 5-volt supply time to stabilize. The watchdog timer is a built-in device for rebooting the PIC every 18 milliseconds; some programs use this to protect themselves

from endless loops, but it's very important to turn it off if you're not using it, or your program will keep restarting itself at inopportune moments.

The last pseudo instruction is

```
org 0
```

which means, "The next instruction should go at address 0 in program memory." That is, you're telling the assembler where to start. Then come the program instructions, and finally the pseudo instruction

```
end
```

which tells the assembler that the program is over. Unlike END in BASIC, `end` in assembly language does *not* tell the program to stop running.

2.5 What the instructions do

Now look at the actual CPU instructions in the program, namely:

```
      movlw  B'00000000'
      tris   PORTB
      movlw  B'00000001'
      movwf  PORTB
fin:  goto   fin
```

What the program needs to do is set up port B for output, place a 1 into the lowest bit of port B (causing pin B0 to go high), and stop. Consider the last of these first. How do you stop a program? Not by making the processor go dead, because then the output at B0 would disappear and

the LED would turn off. Nor by exiting to the operating system, because there isn't an operating system. Your program has the PIC all to itself.

The way you stop this program is by putting it in an endless loop. That's accomplished by the instruction

```
fin:  goto   fin
```

which simply jumps back to itself, over and over.

Now look at the previous step. How do you place a 1 in the lowest bit of port B? From the CPU's viewpoint, port B is an 8-bit register and we want to place binary 00000001 in it. But there is no CPU instruction to place a specified value (a "literal") directly into a port. Instead, the program places 00000001 into the W register using a `movlw` instruction ("move literal to W") and then copies W to port B using a `movwf` instruction ("move W to file register").

Note that in assembly language, "move" always means "copy." That is, every "move" instruction actually *copies* data from one place to another, leaving the original unchanged. This is true of all the assembly languages I've seen, regardless of the kind of CPU.

Note also that in PIC assembly language, the name of port B is `PORTB` (all capitals), not `portb` (lower case). This name was defined in P16F84.INC. MPLAB will recognize both names but the assembler won't – a possible source of confusion.

In order for port B to work as intended, it has to be set up as an output port. Actually, each of its eight bits can be set as input or output, independently of the others, but in this program, all eight bits are set to out-

put. This is done by zeroing the corresponding bits in the `TRISB` special-function register.

2.6 Deprecated instructions – Error messages to ignore

And here we run into a "deprecated instruction," a instruction that works perfectly well but which the assembler tells us not to use. That is `tris`, the instruction that copies W into that special function register. To set up port B for output, we use the instructions

```
movlw  B'00000000'
tris   PORTB
```

and the second of these always generates a warning message.

The reason for the warning message is that some other PICs lack the `tris` instruction, and in the interest of program portability, Microchip, Inc., would like us not to use it. Instead, they want us to do something a good bit more complicated, involving switching over to a different bank of registers, then addressing the `TRISB` register by its address.

That's their preference, but it need not be yours or mine. The `tris` instruction is fully supported on the 'F84; there's nothing wrong with using it; it is not unreliable or risky in any way. It just generates an error message you have to ignore. The same is true of the `option` instruction, which lets you set some CPU configuration options without switching register banks.

3 PART 3 - MAKING THE PIC COMPUTE

3.1 A more elaborate program

Figure 9 shows a more elaborate PIC program that makes eight LEDs blink on, one at a time, in a "chaser" sequence. The sequence reverses direction when you change the logic level on pin A0.

Figure 10 is the circuit for running it. You can make this circuit out of the demo board that comes with the Ramsey PICPRO-1 if you add a switch or jumper so that you can connect pin 17 (A0) to either V+ or ground. With this program, do not leave pin 17 floating; it's a CMOS input and will pick up noise. Also, do not modify the program to light more than one LED at a time; the PIC can't output that much power.

3.2 Describing bit patterns

Before going any further, we need to lay some groundwork. Each PIC file register, `i/o` port, and memory address consists of one 8-bit byte. Thus, any PIC assembly-language program is going to contain a lot of groups of eight bits. There are three main ways to describe a bit pattern: in binary, in hexadecimal, or in decimal.

Obviously, you can describe a bit pattern by writing out the bits themselves, such as `B'00011101'`. In PIC assembly language, the `B` and quotes are required. The lowest-numbered bit comes last; if you output `B'00000001'` on port B, pin B0 will go high, but `B'10000000'` will make pin B7 go high.

You may have been told that computers treat everything as numbers.

```
; File CHASER.ASM
; Blinks LEDs on outputs (Port B) in a rotating pattern.
; Reverses direction if port A, Bit 0, is high.
        processor 16f84
        include   <p16f84.inc>
        __config  _RC_OSC & _WDT_OFF & _PWRTE_ON
; Give names to 2 memory locations (registers)
J       equ     H'1F'   ; J = address hex 1F
K       equ     H'1E'   ; K = address hex 1E
; Program
        org     0       ; start at address 0
        ; Set Port B to output and initialize it.
        movlw   B'00000000'     ; w := binary 00000000
        tris    PORTB           ; copy w to port B control reg
        movlw   B'00000001'     ; w := binary 00000001
        movwf   PORTB           ; copy w to port B itself
        bcf     STATUS,C        ; clear the carry bit
        ; Main loop. Check Port A, Bit 0, and rotate either left
        ; or right through the carry register.
mloop:
        btfss   PORTA,0         ; skip next instruction if bit=1
        goto    m1
        rlf     PORTB,f         ; rotate port B bits to left
        goto    m2
m1:
        rrf     PORTB,f         ; rotate port B bits to right
m2:
        ; Waste some time by executing nested loops
        movlw   D'50'           ; w := 50 decimal
        movwf   J               ; J := w
jloop:  movwf   K               ; K := w
kloop:  decfsz  K,f             ; K := K-1, skip next if zero
        goto    kloop
        decfsz  J,f             ; J := J-1, skip next if zero
        goto    jloop
        goto    mloop       ; do it all again
        end                     ; program ends here
```

Figure 9: A more elaborate PIC program.

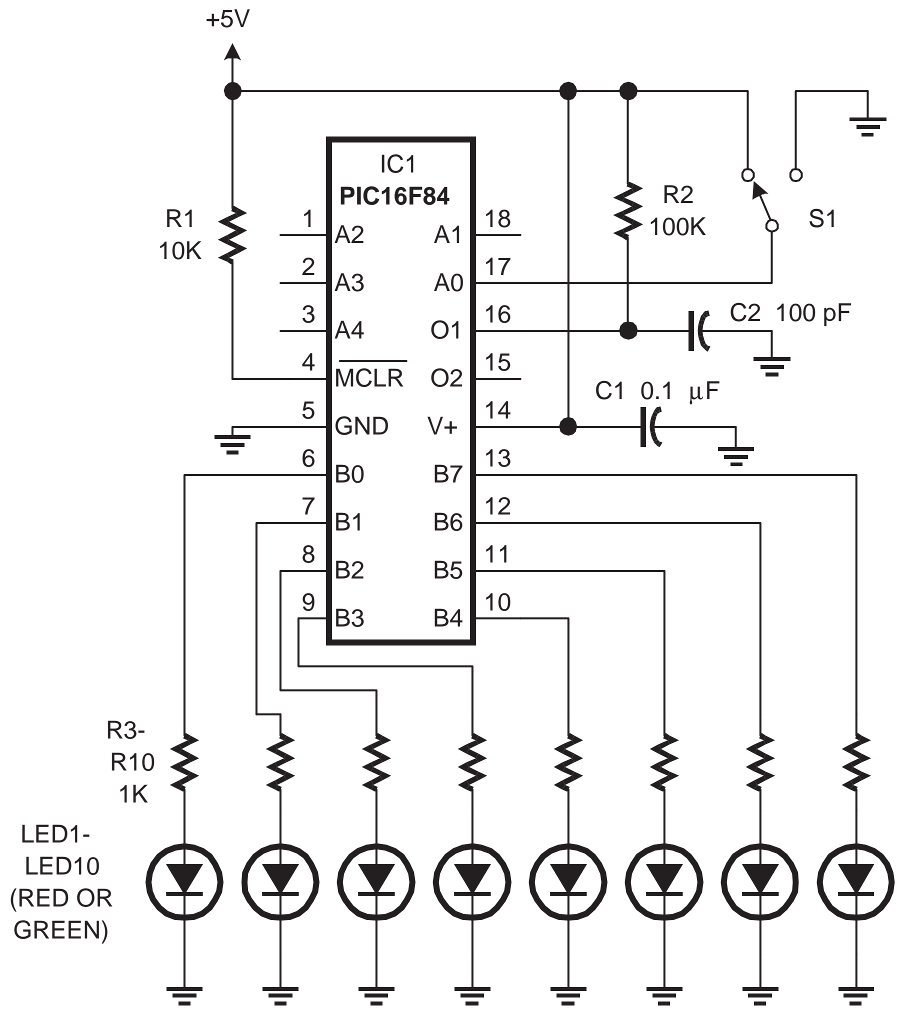

Figure 10: With program in Fig. 9, LEDs flash in chaser sequence; switch S1 reverses direction.

That's not true; computers treat everything as bit patterns. Nonetheless, one common way of interpreting bit patterns is to take them as binary numbers. When this is done, you can write the same number more concisely in another base (radix). For example, binary 00000110 is $1 \times 4 + 1 \times 2$, or 6 in ordinary (decimal) notation. (In this context "decimal" does not imply that there are digits to the right of the point.) Converted to decimal, the 256 possible 8-bit bytes, `B'00000000'` to `B'11111111'`, range from 0 to 255.

It turns out to be even more convenient to convert binary numbers to base 16 (hexadecimal, or hex for short). In hex, there are sixteen digits, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. As you might guess, A means 10, B means 11, and so forth up to F = 15. Thus FF means fifteen sixteens plus fifteen ones, or 255 (binary 11111111).

The reason hex is used is that each hex digit corresponds to four binary digits. If binary 0110 equals hex 6, then binary 01100110 equals hex 66. With some practice, you can learn to do hex-binary conversions in your head. Many scientific calculators can interconvert hex, binary, and decimal. Figure 11 is a complete conversion chart for all possible 8-bit patterns.

3.3 A word of warning

In PIC assembly language you should normally specify the base of every number you write, such as `B'11111111'`,`D'255'`, or `H'FF'`. If you write numbers without the quotes and base indication, they will normally be taken as hex, but it's best to be safe. If you write a character between

	Hexadecimal	Binary		Hexadecimal	Binary		Hexadecimal	Binary		Hexadecimal	Binary
0	00	00000000	64	40	01000000	128	80	10000000	192	C0	11000000
1	01	00000001	65	41	01000001	129	81	10000001	193	C1	11000001
2	02	00000010	66	42	01000010	130	82	10000010	194	C2	11000010
3	03	00000011	67	43	01000011	131	83	10000011	195	C3	11000011
4	04	00000100	68	44	01000100	132	84	10000100	196	C4	11000100
5	05	00000101	69	45	01000101	133	85	10000101	197	C5	11000101
6	06	00000110	70	46	01000110	134	86	10000110	198	C6	11000110
7	07	00000111	71	47	01000111	135	87	10000111	199	C7	11000111
8	08	00001000	72	48	01001000	136	88	10001000	200	C8	11001000
9	09	00001001	73	49	01001001	137	89	10001001	201	C9	11001001
10	0A	00001010	74	4A	01001010	138	8A	10001010	202	CA	11001010
11	0B	00001011	75	4B	01001011	139	8B	10001011	203	CB	11001011
12	0C	00001100	76	4C	01001100	140	8C	10001100	204	CC	11001100
13	0D	00001101	77	4D	01001101	141	8D	10001101	205	CD	11001101
14	0E	00001110	78	4E	01001110	142	8E	10001110	206	CE	11001110
15	0F	00001111	79	4F	01001111	143	8F	10001111	207	CF	11001111
16	10	00010000	80	50	01010000	144	90	10010000	208	D0	11010000
17	11	00010001	81	51	01010001	145	91	10010001	209	D1	11010001
18	12	00010010	82	52	01010010	146	92	10010010	210	D2	11010010
19	13	00010011	83	53	01010011	147	93	10010011	211	D3	11010011
20	14	00010100	84	54	01010100	148	94	10010100	212	D4	11010100
21	15	00010101	85	55	01010101	149	95	10010101	213	D5	11010101
22	16	00010110	86	56	01010110	150	96	10010110	214	D6	11010110
23	17	00010111	87	57	01010111	151	97	10010111	215	D7	11010111
24	18	00011000	88	58	01011000	152	98	10011000	216	D8	11011000
25	19	00011001	89	59	01011001	153	99	10011001	217	D9	11011001
26	1A	00011010	90	5A	01011010	154	9A	10011010	218	DA	11011010
27	1B	00011011	91	5B	01011011	155	9B	10011011	219	DB	11011011
28	1C	00011100	92	5C	01011100	156	9C	10011100	220	DC	11011100
29	1D	00011101	93	5D	01011101	157	9D	10011101	221	DD	11011101
30	1E	00011110	94	5E	01011110	158	9E	10011110	222	DE	11011110
31	1F	00011111	95	5F	01011111	159	9F	10011111	223	DF	11011111
32	20	00100000	96	60	01100000	160	A0	10100000	224	E0	11100000
33	21	00100001	97	61	01100001	161	A1	10100001	225	E1	11100001
34	22	00100010	98	62	01100010	162	A2	10100010	226	E2	11100010
35	23	00100011	99	63	01100011	163	A3	10100011	227	E3	11100011
36	24	00100100	100	64	01100100	164	A4	10100100	228	E4	11100100
37	25	00100101	101	65	01100101	165	A5	10100101	229	E5	11100101
38	26	00100110	102	66	01100110	166	A6	10100110	230	E6	11100110
39	27	00100111	103	67	01100111	167	A7	10100111	231	E7	11100111
40	28	00101000	104	68	01101000	168	A8	10101000	232	E8	11101000
41	29	00101001	105	69	01101001	169	A9	10101001	233	E9	11101001
42	2A	00101010	106	6A	01101010	170	AA	10101010	234	EA	11101010
43	2B	00101011	107	6B	01101011	171	AB	10101011	235	EB	11101011
44	2C	00101100	108	6C	01101100	172	AC	10101100	236	EC	11101100
45	2D	00101101	109	6D	01101101	173	AD	10101101	237	ED	11101101
46	2E	00101110	110	6E	01101110	174	AE	10101110	238	EE	11101110
47	2F	00101111	111	6F	01101111	175	AF	10101111	239	EF	11101111
48	30	00110000	112	70	01110000	176	B0	10110000	240	F0	11110000
49	31	00110001	113	71	01110001	177	B1	10110001	241	F1	11110001
50	32	00110010	114	72	01110010	178	B2	10110010	242	F2	11110010
51	33	00110011	115	73	01110011	179	B3	10110011	243	F3	11110011
52	34	00110100	116	74	01110100	180	B4	10110100	244	F4	11110100
53	35	00110101	117	75	01110101	181	B5	10110101	245	F5	11110101
54	36	00110110	118	76	01110110	182	B6	10110110	246	F6	11110110
55	37	00110111	119	77	01110111	183	B7	10110111	247	F7	11110111
56	38	00111000	120	78	01111000	184	B8	10111000	248	F8	11111000
57	39	00111001	121	79	01111001	185	B9	10111001	249	F9	11111001
58	3A	00111010	122	7A	01111010	186	BA	10111010	250	FA	11111010
59	3B	00111011	123	7B	01111011	187	BB	10111011	251	FB	11111011
60	3C	00111100	124	7C	01111100	188	BC	10111100	252	FC	11111100
61	3D	00111101	125	7D	01111101	189	BD	10111101	253	FD	11111101
62	3E	00111110	126	7E	01111110	190	BE	10111110	254	FE	11111110
63	3F	00111111	127	7F	01111111	191	BF	10111111	255	FF	11111111

Figure 11: Decimal–hex–binary conversion chart.

quotes without a base specifier, such as `'3'`, it will be interpreted as the bit pattern of that character's ASCII code.

3.4 How the program works

We're finally ready to look at the program. It uses rotate-right and rotate-left instructions, `rrf` and `rlf`, to shift bits around in a byte; thus 00000001, rotated left, becomes 00000010, then 00000100 and so forth. Rotating isn't just for LED chasers; it's a handy way to pick off the bits in a byte, one by one, for serial data transmission.

The bits being rotated pass through a ninth bit called the *carry flag;* as you might imagine, the carry flag is also used to keep track of the ninth digit when adding two eight-digit binary numbers. Accordingly, the first thing the program does, other than set up its ports, is clear the carry bit (set it to zero):

```
bcf     STATUS,C
```

Here `STATUS` and `C` are names given by P16F84.INC to the status register and the carry bit within it. They must be written in all capitals. You could refer to the carry bit by its address if you wanted to.

The actual rotating is done by one of the instructions

```
rlf     PORTB,f
```

```
rrf     PORTB,f
```

but the program also has to decide which one to use, depending on the signal at pin A0, and then introduce a time delay after each rotation.

3.5 Making decisions

At this point it's a good idea to look at the full set of PIC instructions, shown in Figures 12 and 13. A selling point of the PIC is that there are so few instructions (only 35 if you count them the way the brochure-writers do); what's more, every instruction that doesn't involve a jump executes in just one clock cycle. Thus, the PIC qualifies as a form of RISC (reduced instruction set computer). By contrast, the Pentium has hundreds of instructions, each of which takes several clock cycles to execute.

There is no "if" statement in PIC assembly language. Instead, the PIC has several instructions that test for a condition and skip the next instruction if it is true. For example, in this program, the instructions

```
btfss PORTA,0
goto  m1
```

mean "Go to `m1` *unless* bit 0 of port A is set (=1)." The first instruction, `btfss`, stands for "bit test file-register, skip next instruction if set." So if the appropriate bit is set, the `goto` is skipped. Even though port A is not a file register, this instruction treats it as if it were.

By interleaving skips and `goto`s in this way, the series of instructions

```
        btfss PORTA,0
        goto  m1
        rlf   PORTB,f
        goto  m2
m1:
        rrf   PORTB,f
```

```
ADDLW   value           Add W to value, place result in W
ADDWF   address,F       Add W to contents of address, store result at address
ADDWF   address,W       Add W to contents of address, place result in W

ANDLW   value           Logical-AND W with value, place result in W
ANDWF   address,F       Logical-AND W with contents of address, store result at address
ANDWF   address,W       Logical-AND W with contents of address, place result in W

BCF     address,bitnumber       Set specified bit to 0
BSF     address,bitnumber       Set specified bit to 1

BTFSC   address,bitnumber       Test bit, skip next instruction if bit is 0
BTFSS   address,bitnumber       Test bit, skip next instruction if bit is 1

CALL    label           Call subroutine (will return with RETURN or RETLW)

CLRF    address         Set contents of address to binary 00000000
CLRW                    Set W to binary 00000000

CLRWDT                  Reset (clear) the watchdog timer

COMF    address,W       Reverse all the bits of contents of address, place result in W
COMF    address,F       Reverse all the bits of contents of address, store result at address

DECF    address,W       Subtract 1 from contents of address, place result in W
DECF    address,F       Subtract 1 from contents of address, store result at address

DECFSZ  address,W       Like DECF address,W and skip next instruction if result is 0
DECFSZ  address,F       Like DECF address,F and skip next instruction if result is 0

GOTO    label           Jump to another location in the program

INCF    address,W       Add 1 to contents of address, place result in W
INCF    address,F       Add 1 to contents of address, store result at address

INCFSZ  address,W       Like INCF address,W and skip next instruction if result is 0
INCFSZ  address,F       Like INCF address,F and skip next instruction if result is 0

IORLW   value           Logical-OR W with value, place result in W
IORWF   address,F       Logical-OR W with contents of address, store result at address
IORWF   address,W       Logical-OR W with contents of address, place result  in W
```

Figure 12: Complete instruction set of PIC16F84 (part 1 of 2).

```
MOVLW   value           Place value in W
MOVF    address,W       Copy contents of address to W
MOVF    address,F       Copy contents of address to itself (not useless; sets Z flag if zero)
MOVWF   address         Copy contents of W to address

NOP                     Do nothing

OPTION                  Copy W to option register (deprecated instruction)

RETFIE                  Return from interrupt

RETLW   value           Return from subroutine, placing value into W

RETURN                  Return from subroutine

RLF     address,F       Rotate bits left through carry flag, store result at address
RLF     address,W       Rotate bits left through carry flag, place result in W

RRF     address,F       Rotate bits right through carry flag, store result at address
RRF     address,W       Rotate bits right through carry flag, place result in W

SLEEP                   Go into standby mode

SUBLW   value           Subtract W from value, place result in W
SUBWF   address,F       Subtract W from contents of address, store result at address
SUBWF   address,W       Subtract W from contents of address, place result in W

SWAPF   address,W       Swap half-bytes at address, place result in W
SWAPF   address,F       Swap half-bytes at address, store result at address

TRIS    PORTA           Copy W into i/o control register for Port A (deprecated)
TRIS    PORTB           Copy W into i/o control register for Port B (deprecated)

XORLW   value           Logical-XOR W with value, place result in W
XORWF   address,F       Logical-XOR W with contents of address, store result at address
XORWF   address,W       Logical-XOR W with contents of address, place result  in W
```

Figure 13: Complete instruction set of PIC16F84 (part 2 of 2).

```
m2:
```

rotates the bits of port A to the left if A0=1 and to the right if A0=0.

If you know another assembly language, you may be wondering how the PIC gets away without having a byte compare (CMP) instruction. The answer is that bytes are compared by subtracting them and then checking whether the result is zero.

3.6 Minding your F's and W's

Many PIC instructions, including `rlf` and `rrf`, come in two varieties, one ending in `f` and the other ending in `w`, which are actually abbreviations for 0 and 1 respectively. These are destination codes: `f` means the result should to in the file register or other memory address that you have specified; `w` means the result should go in the W register.

3.7 Looping

Obviously, the main program will be an endless loop – shift the bits of Port A, delay a few milliseconds, and then go back and do the whole thing again. That's taken care of by the `mloop` label at the beginning of the main loop and the `goto mloop` instruction at the end.

The time delay loop is more complicated because it requires counting. In BASIC, it would look roughly like this:

```
FOR J=50 TO 1 STEP -1
  FOR K=50 TO 1 STEP -1
    REM do nothing
```

```
  NEXT K
NEXT J
```

That is, waste time by counting $50 \times 50 = 2500$ steps.

But we're not programming in BASIC. In PIC assembly language, the time delay loop looks like this:

```
        movlw   D'50'
        movwf   J
jloop:  movwf   K
kloop:  decfsz  K,f
        goto    kloop
        decfsz  J,f
        goto    jloop
```

Here's how it works. First, note that counting down is easier than counting up because it's easier to test whether a byte equals 0 than whether it equals some other number. So we stuff decimal 50 into variables J and K at the beginning of the loop. The value 50 is stored in W, which doesn't change during the whole process; thus it can be copied into K fifty times.

The crucial instruction here is

```
decfsz  K,f
```

which means, "Decrement (that is, subtract 1) and skip the next instruction if the result of decrementing was zero." Thus, a `decfsz` followed by a goto means "Subtract 1 and go back *unless* the result of the subtraction was 0." Two `decfsz` loops, nested, produce the time delay we need. On

the last pass through the loop, the value of J or K respectively is 1; then, when it reaches 0, the program exits the loop.

3.8 Defining variables

But wait a minute. Where did we get the variables J and K? Those aren't the names of file registers, are they?

Actually, they are names *now,* because we defined them. At the beginning of the program, the pseudo instructions

```
J     equ H'1F'
K     equ H'1E'
```

define J and K to be abbreviations for two hex numbers, 1F and 1E, which are the addresses of two suitable file registers. These registers were chosen by looking at the memory map in the PIC16F84 manual; it's entirely up to the programmer to choose a suitable address for each variable.

3.9 Debugging your programs

Finally, a few words about debugging your own programs. Any program should be designed so that you can build and test it in stages, as we did with the two programs in this article: first figure out how to turn on one LED, then get all eight blinking in the proper sequence. In real life there would probably be several stages in between, in which you turn on various LEDs and test the time delay loop. Because the PIC16F84 can be reprogrammed instantly, there's no need to implement the whole program before you test any of it.

Unused output lines can be put to good use for debugging. If your program is complicated, make it send some signals out one or more spare port pins so that you can use a voltmeter or oscilloscope to tell what part of the program is running. If nothing else, output a “heartbeat” bit that toggles every time the program goes through its main loop.

3.10 Where to go from here

Now all that’s left is to learn the rest of the assembly language and the art of microcontroller software development – not something you’ll do in one weekend, but at least your career has been launched. A good book to start with is *Easy PIC’n,* by David Benson, published by Square One Press (http://www.sq-1.com; you can special-order it through any bookstore). Together with its sequels, *PIC’n Up the Pace* and *PIC’n Techniques,* it takes you through PIC programming from the very beginning. These books use the PIC16F84 or its software-compatible twin, the PIC16C84, for most of the projects.

More advanced books about PICs include *Programming and Customizing the PIC Microcontroller,* by Myke Predko (McGraw-Hill, 1998), and *Design with PIC Microcontrollers,* by John B. Peatman (Prentice Hall, 1998). The last of these is a very professional guide based on the PIC16C74A.