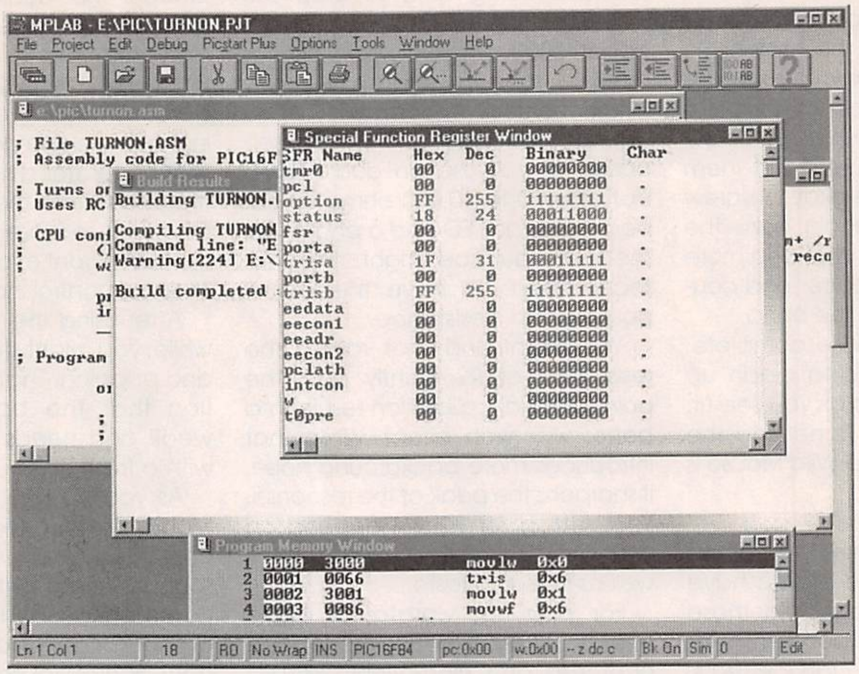
PIC Assembly Language


MPLAB - E:\PIC\TURNON.PJT
File Project Edit Debug Picstart Plus Options Tools Window Help
e:\pic\turnon.asm
Build Results
Special Function Register Window
Program Memory Window

```
; File TURNON.ASM
; Assembly code for PIC16F
; Turns or
; Uses RC
; CPU con
; Program
```

```
Building TURNON.

Compiling TURNON
Command line: "E
Warning[224] E:\

Build completed
```

```
SFR Name     Hex   Dec    Binary     Char
tmr0         00      0    00000000    .
pcl          00      0    00000000    .
option       FF    255    11111111    .
status       18     24    00011000    .
fsr          00      0    00000000    .
porta        00      0    00000000    .
trisa        1F     31    00011111    .
portb        00      0    00000000    .
trisb        FF    255    11111111    .
eedata       00      0    00000000    .
eecon1       00      0    00000000    .
eeadr        00      0    00000000    .
eecon2       00      0    00000000    .
pclath       00      0    00000000    .
intcon       00      0    00000000    .
w            00      0    00000000    .
t0pre        00      0    00000000    .
```

```
1 0000   3000          movlw   0x0
2 0001   0066          tris    0x6
3 0002   3001          movlw   0x1
4 0003   0086          movwf   0x6
```

```
Ln 1 Col 1   18   RO  No Wrap  INS  PIC16F84  pc:0x00  w:0x00  --z dc c  Bk On  Sim 0   Edit
```

for the Complete Beginner

Microcontrollers have revolutionized the world of electronics, but they are useless to you if you don't know how to program them. This month, we show you how easy that is to do.

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. We won't teach you all of PIC assembly language; just enough to get you started. For simplicity, deal with just one processor, the PIC16F84. To be *very* precise, it will be the PIC16F84-04P, which operates up to 4 MHz and is housed in a plastic DIP package. This is a product of Microchip, Inc. (Chandler, Arizona, Web: www.microchip.com), and it's closely related to the rest of the PIC family.

What You'll Need. 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

Michael Covington does research on advanced microcontroller applications at the University of Georgia's Artificial Intelligence Center. He also conducts the monthly "Q&A" section in **Electronics Now** Magazine.

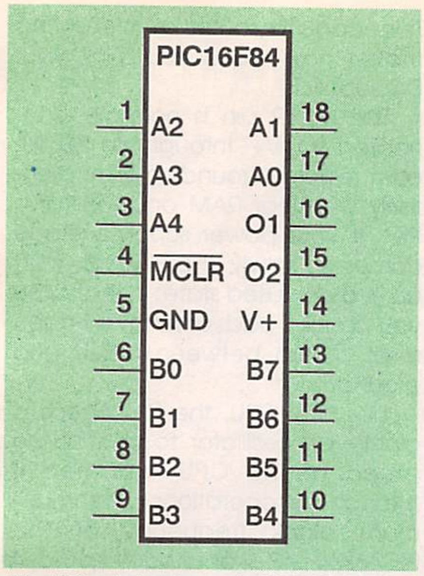


Fig. 1. Here's the pinout of a 16F84 PIC microprocessor that's being used as our example device.

LISTING 1

```
; 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
```

the other parts for the circuits you want to build (refer to the schematics as we go along). And you'll need a PC-compatible personal computer, the MPASM assembler software (which you can download from www.microchip.com), and a PIC programmer such as Ramsey Electronics' PICPRO (available for $59.95 plus $6.95 postage and handling in the U.S. from Ramsey Electronics, 793 Canning Parkway, Victor, NY 14564, Tel: 716-924-4560, Fax: 716-924-4886, Web: www.ramseyelectronics.com), which is based on this author's NOPPP programmer published in the September 1998 issue of this magazine and described at www.mindspring.com/~covington/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 PIC16F83, and you can download it or request a printed copy from Microchip.

What's Inside a PIC? The pinout of the PIC16F84 is shown in Fig. 1, and Fig. 2 shows the most important parts inside of the device. The PIC is a tiny but complete computer. It has a CPU (central processing unit), program memory (PROM), working memory (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

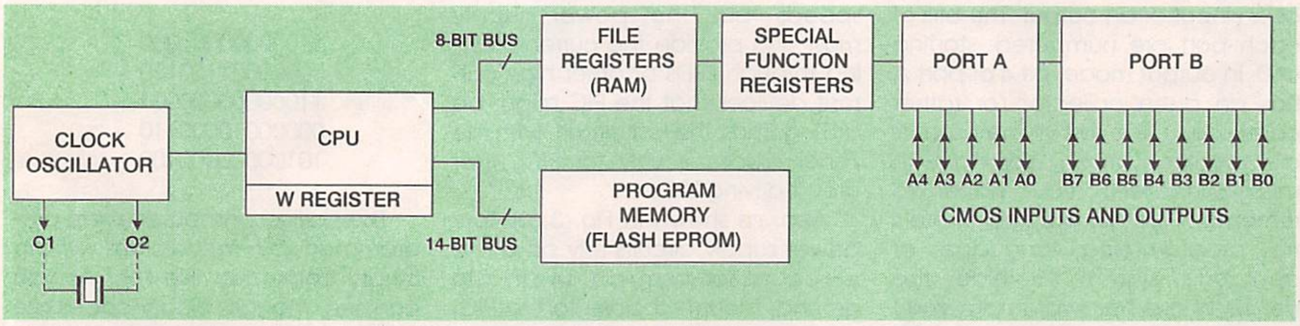


Fig. 2. As you can see from this simplified block diagram of the 16F84, the device is essentially a one-chip computer.

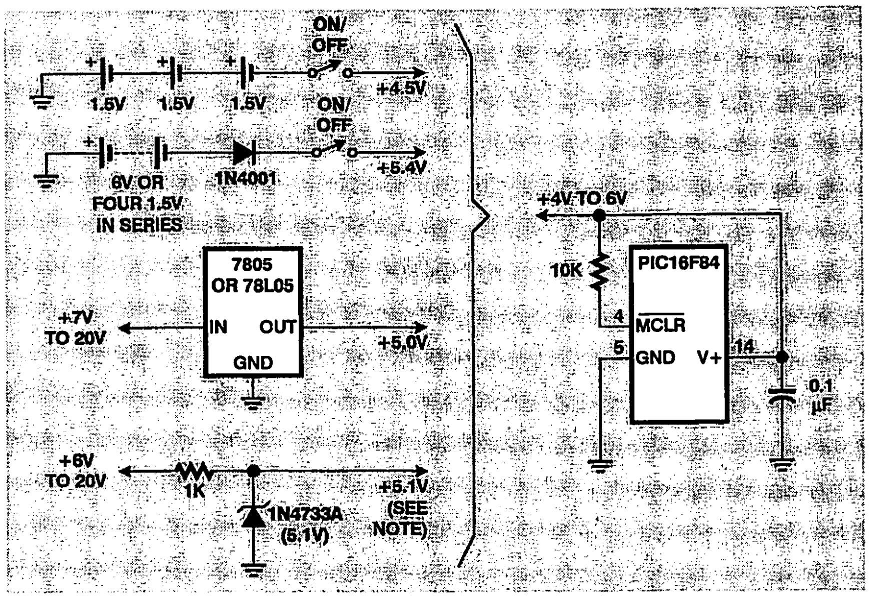


Fig. 3. Any of the four schemes on the left can be used to power a PIC, though the last one should only be used where the device is not driving an LED or a high-current load. Regardless of which power scheme you use, it is important to connect a 0.1-µF capacitor to pin 14 as shown on the left.

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 that it's working on. There are also several special-function registers each of which controls the operation 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.

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.

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. Several power-supply options are shown in Fig. 3. 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 might be driving. Thus, the last circuit, with the Zener diode, is only for PICs that aren't driving LEDs.

Also, as shown in Fig. 3, 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 10,000-ohm 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 between MCLR and ground.

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.

A selection of the most popular clock circuits is shown in Fig. 4. 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.

Assembly Language. A PIC spends its time reading instructions from the program memory, one after another, and doing whatever those instructions say. Each instruction consists of 14 bits. If you could see the bits as binary ones and zeroes, a program like the one in Listing 1 would look like this:

```
11000000000000
00000001100110
11000000000001
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 complete-

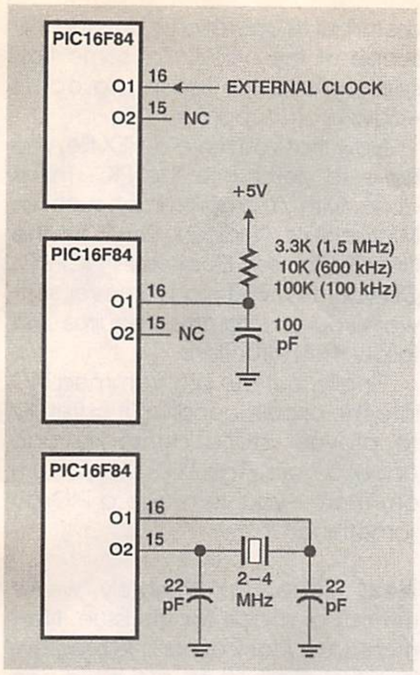


Fig. 4. Three ways are shown to generate the clock signal that is required by the PIC.

ly 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 those 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. The result from the outside world's point of view is that pin 6 of the 'F84 goes high, while pins 7 through 13 remain low.

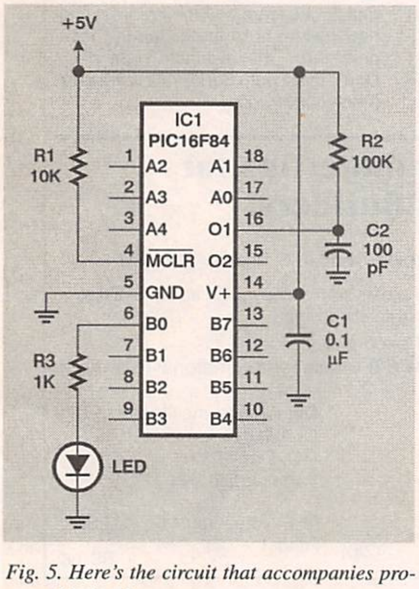
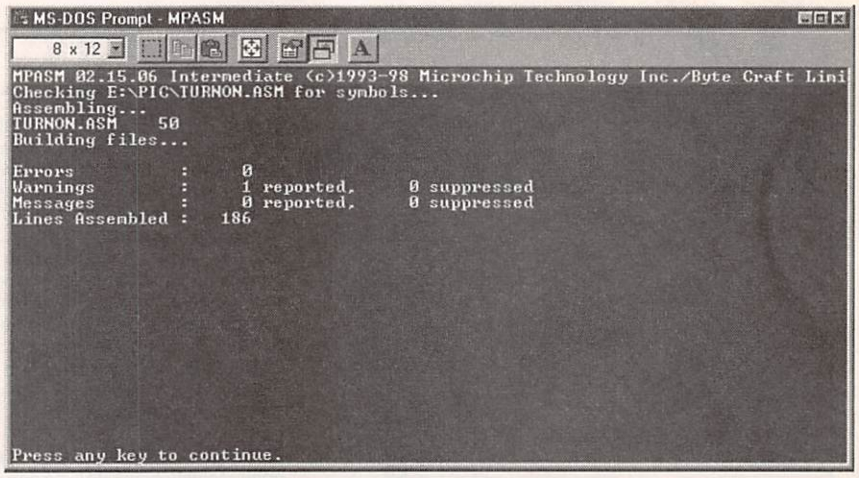


Fig. 5. Here's the circuit that accompanies program Listing 1.



```
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.
```

Fig. 6. To assemble the program in Listing 1, you'll need MPASM, a free program downloadable from www.microchip.com, or a similar PIC assembler.

Program Layout. Listing 1 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 as tolerant.

Assembling a Program. A computer "assembles" the assembly-language program into the binary instructions, which, for brevity, are actually written in hexadecimal (more about that shortly) and stored in 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 Listing 1 does one very simple thing—it turns on an LED connected to pin B0. The circuit needed to try this program out is shown in Fig. 5. 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 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:

```
mpasm turnon.asm
```

and Fig. 6 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 next month) 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 (shown at the beginning of this article) 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. For some tips, see www. mindspring.com/~covington/noppp.

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/Ramsey Electronics PICPRO. 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.

Next Time. Unfortunately, we've run out of space for this issue. Next month, we'll look at our little program in more depth, then see if we can tackle something that's a little more ambitious. We'll also look at some resources you can use to extend your new-found ability to program microprocessors even further. Ω