This article describes how to write PIC 16C84 assembler code programs using a compiler written for the Windows 95/98 platform.
The PIC compiler allows programs to be written in a high-level language and it generates the necessary assembler code. The code can be modified, assembled or simulated using the (free) Microchip MPLAB software.

By Roger Thomas

compiler for PIC16C84

with code optimisation

Compiler main features for PIC16C84

- *Extensive manual (56 pages) on disk*
- *Three worked out examples on disk*
- *Syntax similar to Visual BASIC and Pascal*
- *Windows 95/98 compatible, no DLLs required*
- *Generates assembler code for Microchip MPLAB (freeware)*
- *Variables:* *Boolean, byte, word*
- *Arithmetic Operations:* *+, –, /, *, mod*
- *Numeric Formats:* *decimal, hexadecimal, binary, char*
- *Boolean Functions:* *=, >, <, >=, <=, <>*
- *Boolean Operators:* *AND, OR, XOR*
- *Compiler Commands:* *if...then…else, select/case, while…loop, table, read, write/read, EEPROM, procedure, directive, ASM directive, input, output, alias, pin-name, RTTC, prescaler, wait, picfuse,*
- *Equation Handler*
- *Code Optimiser*
- *Error messages*
- *Interrupt handling*

There are considerable advantages in using a compiler. The ability to write a PIC program using English like commands is easier than programming directly in assembler language. Time taken to write and test software is usually much less with a compiled language, and to prove the point the Windows PIC compiler itself was written using a compiler.
The PIC compiler is not based on any one high-level language but has elements of Pascal and Visual BASIC. Furthermore the compiler is flexible and allows for different program syntax. The compiler is written for the Windows environment, which should make the software easy to use. As all the necessary PIC codes are defined within the compiler program, no external setup or header files are needed.

The compiler produces assembler code directly from the high-level source program, so that the programmer need not worry about the intricacies of assembler code programming. It can be educational for those learning to program in assembler to see how easily understood high-level commands are translated into the equivalent PIC assembler code.

Assembler code output files produced by the compiler contain both the original high-level program (commented out) and the PIC assembler code ready to be assembled (or simulated) by the Microchip MPLAB software version 4.12 or later. Having both the source and assembler code helps in debugging the program. Writing in compiled language does not preclude modifying or adding assembler code to the program when using the MPLAB software. MPLAB software is freely available from Microchip's web site at *http://www.microchip.com*.

The cost (in programming terms) of using compiled code can be reduced speed of execution as the machine code program may not be as efficient as a program written directly in assembler.

With this PIC compiler this is not the case, in most circumstances the assembler code produced is the fastest code possible. There is very little compiler overhead on the assembler code in terms of needing extra variables or increased number of assembler instructions. The only additional program code required is support for the Boolean and arithmetic commands. Arithmetic is either 8 bit (unsigned) or 16 bit (unsigned). The compiler requires several bytes for storing arithmetic

results, these are labelled _STACKxx in the output assembler file.
The compiler makes two passes of the source program. The first pass creates a list of procedure declarations as the compiler may come across a call to a procedure before finding the procedure declaration. On the second pass the procedure calls are reconciled with the procedure declarations.
Compiler syntax is not case sensitive but the MPLAB software can be, this option is selected in the project hex file. For this reason all the procedure names and variables appear in uppercase in the assembler output file.

To allow the compiler to be used for any similar PIC microcontroller the compiler does not impose any code restrictions. This is left to the MPLAB assembler which will check program size and can more easily produce memory usage maps and cross-reference files.

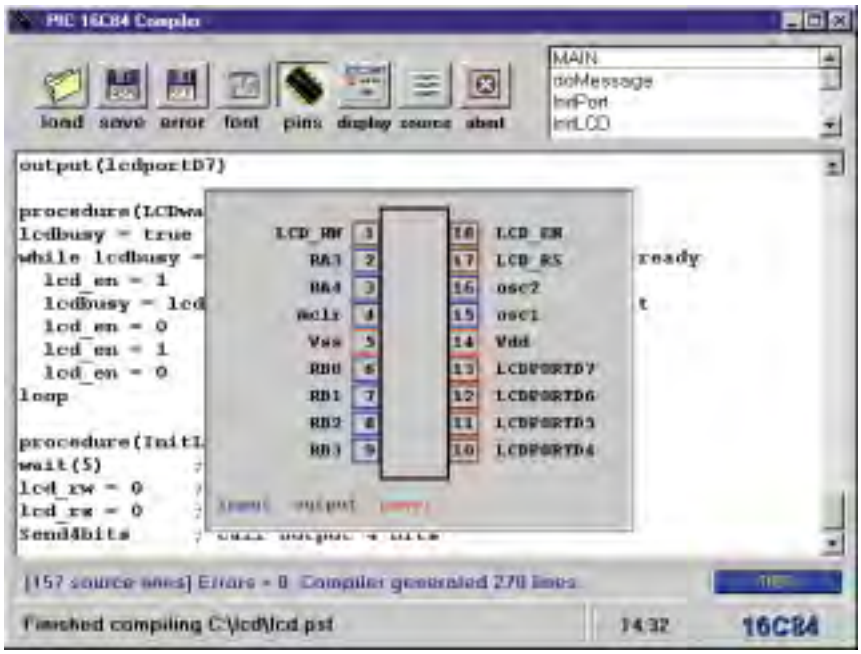


Figure 1. The Compiler window. Note that PIC pin functions can be seen at a glance.

Using the compiler

Use a text editor (such as Notepad, WordPad or the MPLAB editor) to create the high-level source program and save the text file with a .psf file extension (PIC Source File). Ensure that the saved file is text only and does not contain any embedded text formatting information.
Unlike an assembler program that requires a strict code column order (labels, mnemonics, operands, comments), a high-level program freely uses spaces to indent the program. These spaces have no relevance to the program execution and are ignored by the compiler. Using spaces should make the reading and de-bugging the program easier.

User interface

The Windows PIC compiler is very easy to use, apart from using load and save file the compiler does everything else!

As shown in **Figure 1**, the taskbar has a number of icons.

load - press the load button and a directory dialog box will appear listing all the source (filenames.psf) files in the directory, select and load the relevant source file. The compiler will default to the directory that was last used. When the compiler is run for the first time the directory will be where the compiler program is located.

save - after a successful compilation save the assembler source file (same file name but with filename.asm file extension) by pressing this button. This file will be saved in the same directory as the source file. This assembler output file will contain all the additional PIC code required as the compiler will automatically add any support routines.

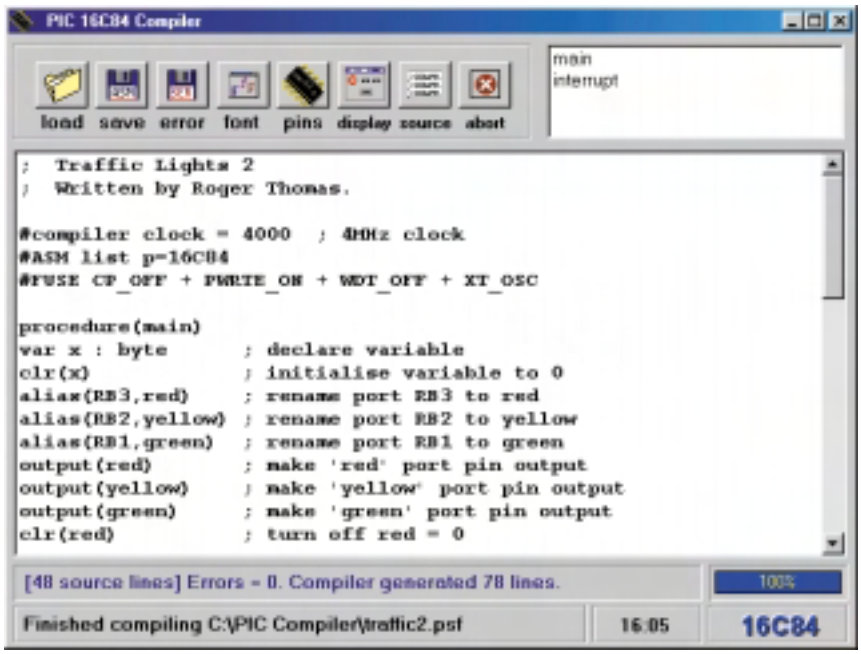
error - saves the error file as a text file to the same directory as the source file. This file has the same file name but with an '.ser' (source error) file extension. It contains all the error messages (which includes the source line number) but not any of the source or assembler code.

font - to change the font or font size of the text displayed on the screen press the font button. A font dialogue box will appear, from which you choose the required font and font size.

pins - will display the PIC pin names and colour coded input or output port pins.

display - if there are any compiler error messages these are inserted into the assembler output file, optionally the compiler will stop and display a mes-

Figure 2. Another example of the PIC Compiler in action. Here, a traffic lights program is being written. Note the procedure names in the top right-hand window.



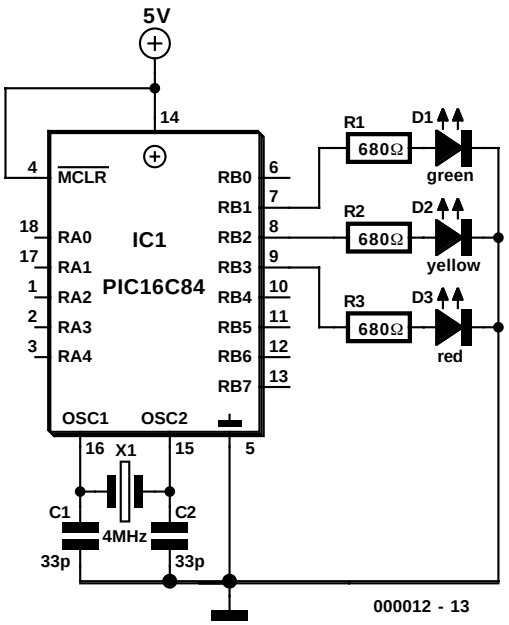


Figure 3. Traffic Lights demo hardware.

sage dialog box (default is to display error message). On the first pass any error messages will be displayed irrespective of this setting.

source - selects whether the source lines should be included in the output assembler file as comments (default is to include source code). Source lines that only contain a comment will always be included.

abort - stops the compilation process.

Next to the eight buttons is a list of all the procedure names used in the program. To find and display a particular procedure, select it from the list (the name will be briefly highlighted) by clicking on the left mouse button. The screen text should scroll and display the relevant procedure (**Figure 2**).

The main procedure of the PIC program is called 'main'. Program execution will start at this procedure, it is called whenever the PIC is reset. The PIC application program is generally held in a continuous loop after any initialisation is done, waiting for events to happen. It is extremely rare for a PIC program to be required to run only once.

Compiler code optimisation

After the compiler has successfully compiled a segment of source program, the code optimiser checks the assembler program for redundant code. Redundant code usually takes the form of unnecessary setting or reading the various PIC status flags.

For example, part of a traffic lights output procedure is shown, the compiler will generate the following code:

(Original source code fragment)

```
green = 0  ; turn green off
yellow = 0 ; turn yellow off
red = 1    ; turn on red

; code not optimised
; green = 0   ; turn green off
   MOVLWH'00'
   BTFSS_STATUS,_Z
   BSF   PORTB,GREEN
   BTFSC_STATUS,_Z
   BCF   PORTB,GREEN
; yellow = 0 ; turn yellow off
   MOVLWH'00'
   BTFSS_STATUS,_Z
   BSF   PORTB,YELLOW
   BTFSC         _STATUS,_Z
   BCF   PORTB,YELLOW
; red = 1       ; turn on red
   MOVLWH'01'
   BTFSS_STATUS,_Z
   BSF           PORTB,RED
   BTFSC_STATUS,_Z
   BCF           PORTB,RED
```

With the code not optimised the compiler has calculated the equation after the equals sign and sets the bit according to the equation result: zero or one. The optimiser looks at the code and finds it has a constant value as it always has the same result and deletes the intermediate calculation.

```
; code optimised
; green = 0   ; turn green off
   BCF   PORTB,GREEN
; yellow = 0 ; turn yellow off
   BCF   PORTB,YELLOW
; red = 1    ; turn on red
   BSF   PORTB,RED
```

PIC interrupts

When an interrupt occurs the program counter is loaded with address location 4, this contains code to save the program status and calls the interrupt handler procedure. After completion of the interrupt handler code within procedure(interrupt), the PIC executes a retfie instruction (return from interrupt). It is not necessary for the source program to re-enable global interrupts as the retfie instruction will do it automatically. The routine to handle interrupts must be called interrupt.
The use of interrupts makes a PIC program more efficient as the alternative is having to continually poll flags to see if a particular event has taken place.
There are four sources of interrupts that the PIC 16C84 interrupt handler software has to deal with:

1. external interrupt on pin RB0;
2. interrupt on change to pins RB4-RB7;
3. RTCC timer overflow;
4. assigned to EEPROM write complete.

The EEPROM write interrupt is taken care of by the writeEEPROM function.

To enable the interrupts use the *irq_enable = true* command. The individual interrupt source must be selected before this command is invoked.
To disable all interrupt use the *irq_enable = false* command.
This command acts globally using the Global Interrupt Enable (GIE) flag irrespective of the individual interrupt being used.

The interrupt handler procedure needs a few bytes for intermediate storage, the interrupt routine has to have its own variable storage as it cannot share storage with the rest of the program.
As an interrupt can occur at any time it is possible that with 16-bit arithmetic this could happen half way through an arithmetic procedure. Assigning a 16-bit variable to another 16-bit variable requires multiple instructions to move the value of the lower and then the higher byte. If an interrupt occurs half way through the process then the variable may end up with the value of the old value (lower byte) and the new value of the higher byte. For this reason the compiler will not allow 16-bit arithmetic in the interrupt handler procedure.

If only one source of interrupts has been enabled then it is not necessary to look at the individual interrupt enable flags. In general it is best to make the interrupt handler procedure as small and execute as fast as possible using simple equations. Note that other procedures cannot be called from within the interrupt handler procedure.

It is better to make a copy of any variable that the interrupt handler may use and use the copy. Referencing a variable that the interrupt handler directly uses can have unforeseen results. For example, if x is changed by the interrupt handler then the following program might not function as intended. The value of x may have been altered after the first but before the second comparison command, so no statements are executed.

```
var x : byte
procedure(main)
if x >=6  and x <= 10 then
      ;  x = 4
begin
    statement
end   ;  interrupt occurs here
if x >= 0 and x <=5 then
      ;  x changes to 6
begin
    statement
end
```

If a byte variable needs to be incremented or decremented or set to zero within the interrupt handler then use inc(x) or dec(x) or clr(x) statements as these compile to a single assembler instruction.

Program examples

To help demonstrate the advantages of using the compiler and clarify the language syntax, the documentation file contains some example programs to help illustrate the various compiler commands. As these programs are for didactic purposes, they do not necessarily represent the best software solution. Note that some of the comment lines have been deleted and the assembler file tidied up for publication.

All variable labels that the compiler generates are preceded by an underscore to differentiate them from variables used in the source program.

The easiest method of implementing a **traffic lights sequence** would be to use the wait command after setting the appropriate LED on or off.

```
green = 0  ; turn green off
yellow = 0 ; turn yellow off
red = 1    ; turn on red
wait(3000) ; wait for 3 seconds
```

If you intend to build the circuit shown in **Figure 3** please observe the PIC current limits. The maximum total current output on Port B is 100 mA, any pin has

Listing 1. Traffic Lights (1) Source program

```
;  Traffic Lights 1
;  Written by Roger Thomas.

#compiler clock = 4000   ; 4MHz clock
#ASM list p=16C84
#FUSE CP_OFF + PWRTE_ON + WDT_OFF + XT_OSC

var x : word   ; create 16 bit variable
var y : byte   ; create 8 bit variable

procedure(main)
alias(RB3,red)     ; rename port RB3 to red
alias(RB2,yellow)  ; rename port RB2 to yellow
alias(RB1,green)   ; rename port RB1 to green
output(red)        ; make 'red' port pin output
output(yellow)     ; make 'yellow' port pin output
output(green)      ; make 'green' port pin output
clr(red)           ; turn off red = 0
clr(yellow)        ; turn off yellow = 0
clr(green)         ; turn off green = 0

clr(x)             ; initialise = 0
clr(y)             ; initialise = 0

while true
  inc(x)           ; x = x + 1
  if x = 1500 then
  begin
    inc(y)         ; y = y + 1
    clr(x)         ; x = 0
  end
  else
  begin
    if (y >= 0) AND (y<= 49) then
    begin
      red = 1      ; turn on red
      yellow = 0   ; turn off yellow
      green = 0    ; turn off green
    end

    if (y >= 50) AND (y<= 75) then
    begin
      red = 1      ; turn on red
      yellow = 1   ; turn on yellow
      green = 0    ; turn off green
    end

    if (y >= 76) AND (y<= 110) then
    begin
      red = 0      ; turn off red
      yellow = 0   ; turn off yellow
      green = 1    ; turn on green
    end

    if (y >= 111) AND (y<= 130) then
    begin
      red = 0      ; turn off red
      yellow = 1   ; turn on yellow
      green = 0    ; turn off green
    end

    if y = 131 then
    begin
      clr(x)
      clr(y)
    end
  end
loop
```

```
;                       16C84
;            RA2   1  | i    i | 18   RA1
;            RA3   2  | i    i | 17   RA0
;            RA4   3  | i    i | 16   osc2
;           mclr   4  | i    i | 15   osc1
;            Vss   5  | p    p | 14   Vdd
;            RB0   6  | i    i | 13   RB7
;          GREEN   7  | o    i | 12   RB6
;         YELLOW   8  | o    i | 11   RB5
;            RED   9  | o    i | 10   RB4


_PCL    EQU H'02'
_STATUS EQU H'03'
_C EQU H'00'
_Z EQU H'02'
_RP0    EQU H'05'
PORTB   EQU H'06'
_PCLATH EQU H'0A'
_INTCON EQU H'0B'
IRQ_ENABLE     EQU H'07'
_STACK0 EQU H'0C'
_STACK1 EQU H'0D'
_STACK2 EQU H'0E'
_STACK3 EQU H'0F'
_STACK4 EQU H'10'
_STACK5 EQU H'11'
_STACK6 EQU H'12'
_STACK7 EQU H'13'
_STACK8 EQU H'14'
_STACK9 EQU H'15'
X  EQU H'16'
XH EQU H'17'
Y  EQU H'18'
RED     EQU H'03'
YELLOW  EQU H'02'
GREEN   EQU H'01'

   ORG 0

   goto MAIN

;  Traffic Lights 1
;  Written by Roger Thomas.

   list p=16C84
   __config H'3FF9'

; var x : word    ; create 16 bit variable
; var y : byte    ; create 8 bit variable

MAIN
; alias(RB3,red)     ; rename port RB3 to red
; alias(RB2,yellow) ; rename port RB2 to yellow
; alias(RB1,green)  ; rename port RB1 to green
; output(red)        ; make 'red' port pin output
   BSF  _STATUS,_RP0
   BCF  PORTB,RED
; output(yellow)     ; make 'yellow' port pin out-
put
   BCF  PORTB,YELLOW
; output(green)      ; make 'green' port pin output
   BCF  PORTB,GREEN
; clr(red)           ; turn off red = 0
   BCF  _STATUS,_RP0
   BCF  PORTB,RED
; clr(yellow)        ; turn off yellow = 0
   BCF  PORTB,YELLOW
; clr(green)         ; turn off green = 0
   BCF  PORTB,GREEN

; clr(x)             ; initialise = 0
   CLRF X
   CLRF XH
; clr(y)             ; initialise = 0
   CLRF Y

; while true
_WHILE0
;    inc(x)           ; x = x + 1
   INCF X,F
   BTFSC        _STATUS,_Z
   INCF XH,F
;    if x = 1500 then
_IF1
   MOVF X,W
   MOVWF        _STACK0
   MOVF XH,W
   MOVWF        _STACK1
   MOVLW        H'FF'
   MOVWF        _STACK2
   MOVLW        H'DC'
   SUBWF        _STACK0,F
   BTFSS        _STATUS,_Z
   CLRF _STACK2
   MOVLW        H'05'
   SUBWF        _STACK1,F
   BTFSS        _STATUS,_Z
   CLRF _STACK2
   MOVF _STACK2,W
   MOVWF        _STACK0
   MOVWF        _STACK1
   BTFSC        _STATUS,_Z
   GOTO _ELSE1
;    begin
;      inc(y)         ; y = y + 1
   INCF Y,F
;      clr(x)         ; x = 0
   CLRF X
   CLRF XH
;    end
;    else
   GOTO _END1
_ELSE1
;    begin
;      if (y >= 0) AND (y<= 49) then
_IF2
   MOVF Y,W
   MOVWF        _STACK0
   MOVLW        H'00'
   SUBWF        _STACK0,W
   CLRW
   BTFSC        _STATUS,_C
   ADDLW        H'FF'
   MOVWF        _STACK4
   MOVF Y,W
   SUBLW        H'31'
   CLRW
   BTFSC        _STATUS,_C
   ADDLW        H'FF'
   ANDWF        _STACK4,W
   BTFSC        _STATUS,_Z
   GOTO _ELSE2
;      begin
;        red = 1      ; turn on red
   BSF  PORTB,RED
;        yellow = 0   ; turn off yellow
   BCF  PORTB,YELLOW
;        green = 0    ; turn off green
   BCF  PORTB,GREEN
;      end

;      if (y >= 50) AND (y<= 75) then
_ELSE2
_IF3
   MOVF Y,W
   MOVWF        _STACK0
   MOVLW        H'32'
   SUBWF        _STACK0,W
   CLRW
   BTFSC        _STATUS,_C
   ADDLW        H'FF'
```

```
   MOVWF       _STACK4
   MOVF Y,W
   SUBLW       H'4B'
   CLRW
   BTFSC       _STATUS,_C
   ADDLW       H'FF'
   ANDWF       _STACK4,W
   BTFSC       _STATUS,_Z
   GOTO _ELSE3
;      begin
;        red = 1      ; turn on red
   BSF  PORTB,RED
;        yellow = 1   ; turn on yellow
   BSF  PORTB,YELLOW
;        green = 0    ; turn off green
   BCF  PORTB,GREEN
;      end

;      if (y >= 76) AND (y<= 110) then
_ELSE3
_IF4
   MOVF Y,W
   MOVWF       _STACK0
   MOVLW       H'4C'
   SUBWF       _STACK0,W
   CLRW
   BTFSC       _STATUS,_C
   ADDLW       H'FF'
   MOVWF       _STACK4
   MOVF Y,W
   SUBLW       H'6E'
   CLRW
   BTFSC       _STATUS,_C
   ADDLW       H'FF'
   ANDWF       _STACK4,W
   BTFSC       _STATUS,_Z
   GOTO _ELSE4
;      begin
;        red = 0      ; turn off red
   BCF  PORTB,RED
;        yellow = 0   ; turn off yellow
   BCF  PORTB,YELLOW
;        green = 1    ; turn on green
   BSF  PORTB,GREEN
;      end

;      if (y >= 111) AND (y<= 130) then
_ELSE4
_IF5
   MOVF Y,W
   MOVWF       _STACK0
   MOVLW       H'6F'
   SUBWF       _STACK0,W
   CLRW
   BTFSC       _STATUS,_C
   ADDLW       H'FF'
   MOVWF       _STACK4
   MOVF Y,W
   SUBLW       H'82'
   CLRW
   BTFSC       _STATUS,_C
   ADDLW       H'FF'
   ANDWF       _STACK4,W
   BTFSC       _STATUS,_Z
   GOTO _ELSE5
;      begin
;        red = 0      ; turn off red
   BCF  PORTB,RED
;        yellow = 1   ; turn on yellow
   BSF  PORTB,YELLOW
;        green = 0    ; turn off green
   BCF  PORTB,GREEN
;      end

;      if y = 131 then
_ELSE5
_IF6
   MOVF Y,W
   SUBLW       H'83'
   MOVLW       H'00'
   BTFSC       _STATUS,_Z
   ADDLW       H'FF'
   ANDLW       H'FF'
   BTFSC       _STATUS,_Z
   GOTO _ELSE6
;      begin
;        clr(x)
   CLRF X
   CLRF XH
;        clr(y)
   CLRF Y
;      end
;   end
_ELSE6
_END1
; loop
   GOTO _WHILE0

   END
```

Listing 2. Traffic Lights (1) Assembler program

a absolute maximum current output of 20 mA. Incorporate an appropriate current limiting resistor (R) in series with the LED (in the range 470 Ω to 1 kΩ depending on the LED). Here, 680 Ω is suggested.

The example program will continue to execute until the supply voltage is removed from the PIC.

The source code of the program is shown in **Listing 1**. The 'x' variable is incremented on each loop of the program. After reaching a certain number it then increments the 'y' variable. This is needed to slow the program down — if the 'x' variable was used directly the lights would switch too fast. The brackets separating the 'y' conditions are not required by the compiler but help document the program. The resulting assembly-code file is shown in **Listing 2**. Other programming examples found in the documentation file are *Traffic Lights (2)* and *LCD Display Driver*. The source code and assembly-code listings of these programs may be found in the documentation file.

Syntax and command descriptions

A full description of all available commands and the syntax the Compiler wants to see may be found in the 56-page project documentation file. This file, an MS Word document, may be found on diskette no. **996033-1** which may be ordered through our Readers Services. The disk also contains the example source code files (.psf) and, of course, the Compiler itself (Compiler84.EXE). The readme.txt file explains the extremely simple installation.

(000012-1)

Article editing: Jan Buiting