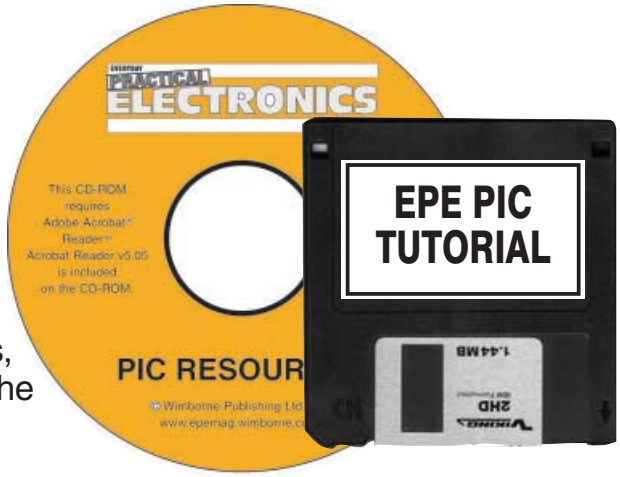
FREE SUPPLEMENT

EPE PIC TUTORIAL V2

JOHN BECKER PART THREE

Quite simply the easiest low-cost way to learn about using PIC Microcontrollers!

In this final part we look at some of the more sophisticated aspects of using PICs, and highlight some differences between the '84 and the '87x and '62x families.



REFERRING back to Listing 30 and Program TK3TUT30 of Part 2 last month, we continue examining the program for a 24-hour clock displayed on an alphanumeric l.c.d.

TIME OUT TO L.C.D.

As with 7-segment l.e.d. clock counting routines, with the l.c.d. program the numerical values are held as BCD counts and each digit is, of course, between 0 and 9 decimal. To the l.c.d., though, values 0 to 9 represent the characters which it holds at its character register addresses 0 to 9, which is not the same thing. The l.c.d.'s characters which "look like" our 0 to 9, are held at its addresses 48 to 57, in other words, they are ASCII characters.

With the 7-segment display, we had to use a table to convert from decimal to a code that it would show meaningfully. With the l.c.d., the conversion is much easier, we simply add the difference between the decimal value and its ASCII value, i.e. we increase the value by 48.

Conveniently, 48 decimal has a binary value of 00110000. The BCD values for decimal 0 to 9 lie between binary 00000000 and 00001001. All we need to do, therefore, is to set bits 4 and 5 of the time digit value in order to increase it by 48, i.e. decimal 9 becomes binary 00111001, which equals 57, the ASCII code for numeral 9.

The easiest way to set bits 4 and 5 is to either add 48 to the digit's value, or to OR 48 with it. In other words, to use either ADDLW 48 or IORLW 48 as the command. In this situation they both have the same effect. To use BSF would require two commands instead of just one. In the following conversion example, the additive technique is used (IOR is used in the program), as shown in Listing 30A.

The SWAPF STORE2,W swaps the nibbles of the value held in STORE2 and holds the result in W, putting the tens into the LSN position. Command ANDLW 15 isolates that nibble, zeroing the MSN. Now ADDLW 48 converts the value to the ASCII character, and LCDOUT is called, which sends the data to the l.c.d. (Have you noticed the similarity to the nibble extraction used for 7-segment displays?)

Next, MOVF STORE2,W brings the entire byte into W, ANDLW 15 isolates the nibble which is in the correct LSN position. Again ADDLW 48 and CALL LCDOUT are performed. Following that, the ASCII value for a colon (58) is sent to the l.c.d., using the single quotes method previously seen in tables.

LISTING 30A

```
SWAPF STORE2,W    ; get tens
ANDLW 15
ADDLW 48
CALL LCDOUT
MOVF STORE2,W     ; get units
ANDLW 15
ADDLW 48
CALL LCDOUT
MOVLW ':'         ; insert colon
CALL LCDOUT
```

CLOCKING ON

Hours, minutes and seconds values are dealt with similarly, although minutes are followed by the decimal point (ASCII 46). Seconds are not followed by any character, although they could have a space character (ASCII 32) sent after the units. The sequence of events, from individually incrementing time to outputting the data to the l.c.d. is shown in Listing 30 (Part Two).

Now compare this listing with that for outputting the time data to the 7-segment displays (TK3TUT28). Look especially at the clock count section (from CLKADD to end of ADDCL2). The second version, of which the main part is shown in Listing 30, is considerably more compact.

After initialisation and general set-up, the program enters the MAIN routine. At each 1/25th second time-out, CLKADD is called and the CLKCNT counter decremented, as we saw earlier. Only if the value of CLKCNT is zero is the next routine entered. After resetting CLKCNT, the address of CLKSEC is set in the indirect address register FSR, a loop (LOOP) is set for three operations and STORE1 is cleared for use as an up-counter. In the three steps round the loop, CLKSEC is dealt with first, then CLKMIN and then CLKHRS.

First time round the loop, at ADDCLK the first byte to be incremented is, of course, CLKSEC. This is then checked for a units value greater than nine and action taken accordingly. Next, the value within STORE1 is copied into W and a table (CHKVAL – see full listing) is called, returning with the maximum permitted value for the byte being processed, storing it in STORE2.

The value of the time byte (CLKSEC at this moment) is then copied into W, which is then subtracted from STORE2. If the Carry flag is set, then STORE2 is greater than CLKSEC (there is no borrow) and an exit is made from the loop, no further action being needed, and a jump is made to the display routine (CLKSHW).

If CLKSEC is greater than STORE2 a borrow occurs, thus CLKSEC is cleared, counter STORE1 is incremented for the sake of the table jump address, and the FSR address is incremented (to point now to CLKMIN). The loop counter (LOOP) is decremented and, if it is not zero, the loop is repeated, this time incrementing and checking CLKMIN in the same way as CLKSEC was dealt with. If CLKMIN is reset, the loop is repeated for the third occasion, this time for CLKHRS.

Two sub-routines are used with CLKSHW, to save repetition of too many commands. The routines are LCDLIN and LCDFRM. The former is responsible for setting the starting display cell position on the l.c.d. Since in a larger program this position could change frequently, it is worthwhile having a generalised routine for this purpose. In this case, we want the time to be shown at the start of the second l.c.d. line, so the value B'11000000' (the address of line 2 cell 0) is moved into W and LCDLIN called. All LCDLIN does is set the RSLINE flag for command mode (BCF RSLINE,4), call LCDOUT, and reset the RSLINE flag to character mode (BSF RSLINE,4).

Next, the value of CLKHRS is moved into W and LCDFRM called. This routine does the swapping, ANDing and ORing necessary for numerical conversion to the ASCII value. After this, the colon is sent directly to LCDOUT. Similar commands are then given with regard to CLKMIN and CLKSEC. The program then returns to the MAIN routine to begin again.

It is worth commenting at this point that the starting l.c.d. cell position can be set to any value via the LCDLIN routine. Line 1 needs bits 7 and 6 set to 1 and 0 respectively. Line 2 needs bits 7 and 6 set to 1 and 1 respectively. The cell position on the lines is controlled by the final four binary digits, bit 3, 2, 1, 0. For example, to set for line 1 cell 8 (regarding the first cell as zero) the value of B'10001000' should be sent to LCDLIN, for line 2 cell 15 the value is B'11001111'.

EXERCISE 23

23.1. Extend the program so that the clock also keeps track of months and years.

TUTORIAL 24

CONCEPTS EXAMINED

Adding time-setting switches

CONNECTIONS NEEDED

L.C.D. as in Fig.7 (Part Two)
CP20 to +5V OUT
CP21 to 0V OUT
Crystal oscillator

The clock program of TK3TUT30 that is now being run is perfectly usable as a real-time clock, as is the 7-segment version (but see later). They both have a major problem though, the programs have to be started (reset) at exactly midnight for the time shown to be accurate. What we need is the ability to set the current time via switches, as with most other time-keepers. Here we show how switched time-setting can be programmed into the l.c.d. version.

We have already looked quite heavily at the use of switches in earlier sections. It is not hard to implement switched time-setting routines, but it takes quite a few commands (as Listing 31 shows), especially as we are allowing you a luxury: the ability to count upwards or downwards on both minutes and hours. Many clocks do not allow this, and it can be a right pain if you overshoot the time you want! We also allow you the option of a fifth switch to reset the seconds, although you will need to add that switch externally to *TK3*'s p.c.b.

First, though, attention must be paid to the rate at which the digits are changed by the switches. We could easily insert a switch checking routine either on each 1/25th count, or on each second. However, the first is too fast for convenience, and the other too slow. A better rate is on every half-second. This can be arranged by halving the prescaler rate, setting it for a ratio of 1:64 instead of 1:128. Thus, in the initialisation, instead of commands MOVLW B'10000110' and MOVWF OPTION_REG, we use:

```
MOVLW B'10000101'
MOVWF OPTION_REG
```

LISTING 31 – PROGRAM TK3TUT31

```
MAIN    BTFSS INTCON,2
        GOTO MAIN
        BCF INTCON,2
        CALL CLKADD
        GOTO MAIN
CLKADD  DECFSZ CLKCNT,F
        RETURN
        MOVLW 25
        MOVWF CLKCNT
        CALL GETKEY
        INCF HLFSEC,F
        BTFSC HLFSEC,0
        CALL CLKIT
        RETURN

(Section from CLKIT to end of LCDLIN omitted)

GETKEY  BTFSS PORTA,3
        GOTO CHKSW2
        BSF EVENT,0
        MOVLW CLKHRS
        GOTO TIMSET
CHKSW2  BTFSS PORTA,2
        RETURN
        CLRF EVENT
        MOVLW CLKMIN
TIMSET  MOVWF FSR
        BTFSC PORTA,0
        GOTO SUBTIM
        BTFSS PORTA,1
        RETURN
ADDTIM  INCF INDF,F
        MOVLW 6
        ADDWF INDF,W
        BTFSC STATUS,DC
        MOVWF INDF
        INCF EVENT,W
        CALL CHKVAL
        MOVWF STORE2
        MOVF INDF,W
        SUBWF STORE2,F
        BTFSS STATUS,C
        CLRF INDF
        GOTO CLKSHW
SUBTIM  MOVLW 1
        SUBWF INDF,F
        BTFSS STATUS,C
        GOTO SUBSET
        BTFSC STATUS,DC
        GOTO ENDSUB
        MOVF INDF,W
        ANDLW B'11110000'
        IORLW 9
        MOVWF INDF
        GOTO ENDSUB
SUBSET  INCF EVENT,W
        CALL CHKVAL
        MOVWF INDF
ENDSUB  GOTO CLKSHW
```

Counter CLKCNT is still set for 25 but we use an additional counter HLFSEC for half seconds, so that although the switches are sampled every half second, the seconds themselves are still incremented correctly.

Referring to Listing 31, you will see the command CALL GETKEY, which is then followed by INCF HLFSEC,F. Only if bit 0 of HLFSEC is 1 will the CLKADD routine be entered. Imagine now that the switches on PORTA are designated as follows:

SW4 = seconds reset
SW3 = hours
SW2 = minutes
SW1 = plus (+)
SW0 = minus (–)
(SW4 is the optional switch referred to a moment ago)

At GETKEY, if switch SW3 is pressed (hours), EVENT bit 0 is set to 1. This file value will be used when accessing the CHKVAL table for the maximum roll-over value for hours or minutes. Now the address of CLKHRS is moved into W and a jump to TIMSET is made.

At this routine, the plus (+) and minus (–) keys are read for their status, and the addition (ADDTIM) or subtraction (SUBTIM) routine is jumped to and processed. In these routines, not only have the units to be checked for values greater than nine, but the overall BCD value has to be checked for greater than 23 (hours) and greater than 59 (minutes).

In the addition routine, the excess value is checked for, and the value is reset to zero if it is exceeded. In the subtraction routine, zero is checked for, in which case the maximum allowed value is moved into the byte as the reset value. In both instances, the value within EVENT is moved into W and table CHKVAL is called for the maximum value.

All the commands involved in these routines should by now be familiar to you without further explanation. Load TK3TUT31.HEX and experiment with setting the time.

TIMING ACCURACY

It is important to be aware that the accuracy of a crystal controlled clock using coding such as this is not perfect. Crystals are subject to manufacturing tolerance in respect of the exact frequency at which they oscillate. Unless the crystal on the p.c.b. is oscillating at exactly 3276800·00Hz, the timing will drift over extended periods.

In other hardware designs it is possible to include a trimmer capacitor in the oscillating circuit to adjust for timing drift. It is also possible to include sophisticated adjustment routines in the program to compensate. Such an example is included with the *PICronos L.E.D. Wall Clock* published in *EPE* June '03. Such techniques, though, are beyond the scope of these Tutorials.

EXERCISE 24

24.1. Change the role of switch SW4 and create a routine that will also show how many hours, minutes and seconds there are until midnight (00:00.00 hours or 24:00.00 if you find it easier) when you press a switch, clearing the answer when the switch is released.

LISTING 32A – PROGRAM TK3TUT32.ASM

```
SETPRM   MOVWF EEADR      ; Copy W into EEADR to set EEPROM address
         BANK1
         BSF EECON1,WREN  ; enable write flag
         BANK0
         MOVF STORE1,W    ; get data value from STORE1 and hold in W
         MOVWF EEDATA     ; copy W into EEPROM data byte register
MANUAL   BANK1            ; these next 12 lines are according to
         MOVLW H'55'      ; Microchip manual dictated factors
         MOVWF EECON2     ; they cause the action required by
         MOVLW H'AA'      ; by the EEPROM to store the data in EEDATA
         MOVWF EECON2     ; at the address held by EEADR.
         BSF EECON1,WR    ; set the "perform write" flag
CHKWRT   BTFSS EECON1,4   ; wait until bit 4 of EECON1 is set
         GOTO CHKWRT
         BCF EECON1,WREN  ; disable write
         BCF EECON1,4     ; clear bit 4 of EECON1
         BANK0
         BCF INTCON,6     ; clear bit 6 of INTCON
         RETURN           ; and RETURN
```

LISTING 32B

```
GETPRM   MOVWF EEADR      ; copy W into EEADR to set EEPROM address
         BANK1
         BSF EECON1,RD    ; enable read flag
         BANK0
         MOVF EEDATA,W    ; read EEPROM data now in EEDATA into W
         RETURN           ; and RETURN
```

24.2. This one is more complicated! By doing exercise 24.1 you have lost the ability to reset the seconds count when you want to – unless you amend the program, seconds will be reset whenever SW4 is pressed. It is possible to amend the program so that switch SW1 and SW0 still serve as plus and minus controls, but SW2 could be used to select whether it is the minutes or the hours that are amended, with a suitable symbol indicating which value is under control. SW3 could then be used to reset the seconds, with SW4 simply controlling the midnight countdown display. Have a go at this challenge!

TUTORIAL 25

CONCEPTS EXAMINED

Writing and reading EEPROM file data
Register EECON1
Register EECON2
Register EEDATA
Register EEADR

We have already shown how convenient it is to be able to repeatedly change the program data within a PIC. The demos and your experiments would simply not have been practical had we been using a microcontroller which required erasing by ultraviolet light each time a new program had to be loaded into it.

Now we come to another great advantage of most PICs, including the PIC16F84, the presence of an EEPROM data memory which can be written to and read from whenever we want, and which will not lose the data when the power is switched off.

We shall now show the commands needed for EEPROM data memory read/write operation and then in Tutorial 26 demonstrate a simple program that makes use of the facility.

The full program for this initial discussion is on your disk as TK3TUT32, its main contents are shown in Listings 32A and 32B. Note that this program cannot be run as it stands and is for use as a subroutine within a main program. Also note that the program is specific to the PIC16F84 and that other PIC families may require slightly different coding. Examples for the PIC16F87x and PIC16F62x families are discussed later.

In some respects, use of the EEPROM read/write facility is similar to that used in indirect addressing, a special register (EEADR) is loaded with the address within the EEPROM at which the data is to be stored or retrieved. This register can be likened to FSR.

The data which has to be written to the EEADR register is loaded into register EEDATA (equivalent to INDF).

On retrieving data from the EEPROM, register EEADR is loaded with the address from which the data is to come, and then the PIC copies the data from that position into EEDATA.

Prior to writing data to the EEPROM, a write-enable flag has to be set in register EECON1. Another flag is set in EECON1 when data is to be read from the EEPROM.

To transfer data from EEDATA to the EEPROM file pointed to by EEADR, an obligatory routine as specified in the PIC's datasheet has to be performed. This routine initialises operations built into the PIC and which last for a predetermined time.

A flag (EECON1,4) is set by the PIC when these operations have occurred and its setting has to be waited for before further program commands can be performed. Failure to wait for the flag setting can disrupt the correct storage of the data.

An example of how the writing routine is used is shown in Listing 32A. Prior to entry into the routine at SETPRM, the data to be written is temporarily placed in file STORE1 (or any name you like). Then the EEPROM address at which the data is to be stored is moved into W and the call to SETPRM is issued.

On entry to SETPRM, the contents of W are copied into EEADR, and then, via BANK1, the command BSF EECON1,WREN is given, setting the EEPROM into write-enable mode, after which follows a reset to BANK0. Data is then copied from STORE1 into W and then into EEDATA.

Now the routine specified in the PIC's datasheet is started at label MANUAL. The 12 lines of this routine, from BANK1 down to BCF INTCON,6 should be followed parrot-fashion in any other EEPROM-writing program. The final return command could be replaced by a GOTO, or by the program immediately following on into another routine.

Reading data from the EEPROM is very simple, as Listing 32B shows. The routine is entered at GETPRM with the EEPROM file address held in W. This is copied into EEADR then, via BANK1, the enable read flag is set (BSF EECON1,RD) and BANK0 reset. The data required is immediately available to be copied into W by the command MOVF EEDATA,W.

EXERCISE 25

There is no exercise for this Tutorial.

TUTORIAL 26

CONCEPTS EXAMINED

Illustrating use of EEPROM data read/write
Converting binary value to hexadecimal

CONNECTIONS NEEDED

L.C.D. as in Fig.7
CP20 to +5V OUT
CP21 to 0V OUT
Crystal oscillator

Program TK3TUT33.ASM illustrates an example of writing to and reading from the PIC's Data EEPROM. It uses the l.c.d. to display three values, and switches SW0 to SW2 to increment them. Switch SW3 causes the new values to be stored into the Data EEPROM at consecutive addresses from 0 to 2.

After the program's initialisation sequence, data currently stored in the EEPROM is recalled as shown in routine GETVALUES in Listing 33. To retrieve a value the address at which it is stored in the EEPROM is first loaded into W (0 in the first instance). The routine at GETPRM (discussed in Tutorial 25) is then called, and a return is made to the calling routine with the retrieved EEPROM value held in W. This is then stored into the register required (in the first instance VALUE0).

The procedure is repeated three times and then a call is made to the display routine SHOWVALS in which the values are shown in hexadecimal.

The MAIN routine is then entered in which the four switches are read at 1/5th of a second intervals (set by routine PAUSIT). If any of switches SW0 to SW2 are found to be pressed, the associated VALUE is incremented in binary and the display routine called again, followed by a return to label MAIN.

If switch SW3 is found to be pressed, the data storage routine is called, at STOREIT. The value to be stored (VALUE0 in the first instance) is called into W and moved into a temporary store, STORE1. The EEPROM address at which the data is to be stored (0 in the first instance) is then called into W and the SETPRM routine called,

LISTING 33 – PROGRAM TK3TUT33.ASM

```
GETVALUES   MOVLW 0          ; get values from EEPROM address 0 to 2
            CALL GETPRM      ; store into VALUE
            MOVWF VALUE0
            MOVLW 1
            CALL GETPRM
            MOVWF VALUE1
            MOVLW 2
            CALL GETPRM
            MOVWF VALUE2
            CALL SHOWVALS    ; show values
MAIN        CALL PAUSIT      ; 1/5th sec pause
            BTFSC PORTA,0    ; is SW0 pressed?
            GOTO INCVAL0     ; yes
            BTFSC PORTA,1    ; no, is SW1 pressed?
            GOTO INCVAL1     ; yes
            BTFSC PORTA,2    ; no, is SW2 pressed?
            GOTO INCVAL2     ; yes
            BTFSC PORTA,3    ; no, is SW3 pressed?
            GOTO STOREIT     ; yes
            GOTO MAIN        ; no
INCVAL0     INCF VALUE0,F    ; inc VALUEs as called and then show
            CALL SHOWVALS
            GOTO MAIN
INCVAL1     INCF VALUE1,F
            CALL SHOWVALS
            GOTO MAIN
INCVAL2     INCF VALUE2,F
            CALL SHOWVALS
            GOTO MAIN
STOREIT     MOVF VALUE0,W    ; store all VALUEs into EEPROM
            MOVWF STORE1
            MOVLW 0
            CALL SETPRM
            MOVF VALUE1,W
            MOVWF STORE1
            MOVLW 1
            CALL SETPRM
            MOVF VALUE2,W
            MOVWF STORE1
            MOVLW 2
            CALL SETPRM
(routine to display STORED)
WAITSW      BTFSC PORTA,3    ; wait until switch SW3 released
            GOTO WAITSW
(routine to clear STORED)
            GOTO MAIN
```

where the data is stored at the required address (as discussed in Tutorial 25).

Following storage of all values, the word STORED is displayed on screen for as long as switch SW3 is held pressed. When the switch is released, the word is cleared and a jump back to label MAIN is made, to await the next switch press.

For convenience in this Tutorial, in the data display routine the binary values are converted to hexadecimal and then displayed. In Tutorial 33 later, conversion of binary numbers to decimal values suitable for l.c.d. display use is illustrated.

In the binary to hex routine, an extract of which is shown in Listing 33A, the byte value is first swapped into W to put the MSN (most significant nibble) into the righthand position, and a call made to a conversion table, HEXTABLE (see full listing). Here the value in W is ANDed with B'00001111' to extract just the lower nibble, and ADDed to PCL. The table jump then returns with the hex value for that nibble, which is then sent to the l.c.d. The procedure is repeated for the LSN of the value byte.

To prove that the data has been stored, note the three values, switch off the power for a few seconds and then switch back on. The data displayed on screen when the program restarts will be the same as that noted. You can also examine all the Data EEPROM values via *TK3*'s EEPROM Message Read facility.

LISTING 33A

```
SWAPF VALUE2,W
CALL HEXTABLE
CALL LCDOUT
MOVF VALUE2,W
CALL HEXTABLE
CALL LCDOUT
```

EEPROM data storage and retrieval has many applications in practical situations. For example, the option is valuable when setting up a program during the testing or tuning stages, allowing the values to be recalled next time the program is run.

The values to be stored need not have originated from switches, they could be provided by other functions within a program. Storage of the values can be in response to a switch press, as illustrated, or could again be triggered by some aspect within the running program, such as in response to clocked timing values.

It is vital to appreciate that a PIC's Data EEPROM has a finite number of times that it can be written to – around one million times according to the PIC16F84's datasheet. This may seem a large number, but it can soon be consumed by incautious programming causing the EEPROM to be repeatedly written to.

During program development when automatic EEPROM writing is included, it is worthwhile putting in a temporary intercept counter and l.e.d. or l.c.d. display routine to monitor the number of times that calls are made to the EEPROM write routine.

EXERCISE 26

The following two exercises are complicated and should only be attempted by those who have successfully followed the Tutorials so far!

33.1 To illustrate EEPROM writing and reading, the author considered modifying the 4-note playing program of TK3TUT19 so that it became eight notes, which were played in the order specified by data in the EEPROM. The data would have been entered via the switches, using the l.c.d. to display which note numbers were being stored at which EEPROM addresses.

A separate switch would have been used to start and stop the note sequence being played. Each note would have had a duration of one second, although it would be possible to set their durations by switches. It would seem preferable to have separate up/down switches, a MODE switch to select the functions of the other switches on a cyclic basis, and a Start/Stop switch. It should be possible to do it with the four switches on *TK3*'s p.c.b. Can you do it?

33.2. You will have discovered that precise musical tuning of the four notes in program TK3TUT19 is not possible using the length of a loop to determine the frequency. It *is* possible though, if within the loop you use a 24-bit counter (three bytes, MSB, NSB, LSB) and add a 16-bit number (two bytes) to it. The toggling of PORTA RA4 would then depend on one of the bit values of the counter's MSB. The additive value depends on the note to be generated and so eight notes need one each.

It is suggested that you try this technique with the program modified in 33.1. The principle was illustrated in the author's *StyloPIC* of July '02 (on the PIC Resources CD-ROM)..

(A similar additive technique can also be used to adjust the precise timing of a crystal controlled clock, as used in the *PICronos* clock referred to earlier).

TUTORIAL 27

CONCEPT EXAMINED

Interrupts
Command RETFIE

CONNECTIONS NEEDED

SW0 to RB0
LD0 to RA0
LD1-LD7 to RB1-RB7
CP20 to +5V OUT
CP21 to 0V OUT
Crystal oscillator

LISTING 34 – PROGRAM TK3TUT34

```
            ORG 0
            GOTO 5
            ORG 4
            GOTO INTRPT
            ORG 5

            CLRF PORTA
            CLRF PORTB
            BANK1
            CLRF TRISA
            CLRF TRISB
            MOVLW B'10000111'
            MOVWF OPTION_REG
            BANK0

            MOVLW B'10100000'
            MOVWF INTCON

START       NOP
            GOTO START

TEST        BSF PORTA,0
INTRPT      MOVLW 2
            ADDWF PORTB,F
            BCF INTCON,2
```

From here on, none of the commands examined directly relate to extending or modifying any of the foregoing programs. Connect l.e.d. LD7 to PORTA pin RA0 and switch SW0 to PORTB RB0.

INTERRUPTS

Early on in this series, mention was made of Interrupts, saying they would be examined later. That "later" has arrived!

An Interrupt, as the term implies, literally is an "interrupt" to the program, causing it to stop what it is currently doing, and perform another action or set of actions, returning to where it left off when the interrupt occurred.

Interrupts can be set to occur from several sources, of which two seem the most likely ones to be required: externally from another piece of equipment, such as a switch or from a trigger pulse generated by another electronic circuit; internally, at the end of a time-out period generated by the PIC's own timer.

There are other interrupt possibilities, but which are probably of more benefit to experienced programmers and which will not be detailed here. Readers wanting more information on interrupts are referred to Malcolm Wiles' *Programming PIC Interrupts* of *EPE* Mar/Apr '02 (on the PIC Resources CD-ROM).

There are countless situations where interrupts can be put to good use. Let's examine two of them.

First, the address to which the program must jump when interrupted has to be specified. This is where the ORG 4 statement now comes into its own. Following that statement, and prior to the ORG 5 statement, the jump address is inserted. Let's call the jump address INTRPT. So, at the beginning of the program listing we make the following statements:

```
ORG 0          ; Reset Vector address
GOTO 5         ; go to PIC address
                 location 5
ORG 4          ; Interrupt Vector
                 address
GOTO INTRPT; go to interrupt routine
ORG 5          ; Start of Program
                 Memory
```

Since the program, once triggered by an interrupt, automatically jumps to the program address stated, we can simply set up a holding routine which waits until the interrupt occurs, and then the routine specified at the interrupt address is performed.

We could actually allow the entire program to be performed without using a holding routine, jumping to the specified routine when the interrupt does occur. This is tricky, though, and can be dangerous to the correct operation of the main program. Allowance has to made for a particular operation to be completed before the interrupt routine is performed. It is this type of information and how to handle it that Malcolm discusses in his article.

For our purposes now the use of a holding routine illustrates the essential point about an interrupt action. It can be as simple as:

```
START: NOP
       GOTO START
```

The program would normally be constantly looping through the two commands NOP and GOTO START, waiting for an interrupt to occur. On its occurrence, the loop would be exited, and a jump made to the routine at INTRPT. Obviously, at the end of the routine caused by the interrupt, a return to the program point from where the interrupt jump was made must be specified. There is a command which is used for this purpose, RETFIE.

TIMER INTERRUPT

A simple program which makes use of an internally timer-generated interrupt to increment a count on PORTB is shown in Listing 34. Here, the timer is set in the same way as we have been doing previously. Then the INTCON register is told that an interrupt is to be generated when the timer rolls over to zero:

LISTING 35 – PROGRAM TK3TUT35

```
            ORG 0
            GOTO 5
            ORG 4
            GOTO INTRPT
            ORG 5

            CLRF PORTA
            CLRF PORTB
            BANK1
            CLRF TRISA
            MOVLW B'00000001'
            MOVWF TRISB
            MOVLW B'11000111'
            MOVWF OPTION_REG
            BANK0

            MOVLW B'10010000'
            MOVWF INTCON

START       NOP
            GOTO START

INTRPT      MOVLW 2
            ADDWF PORTB,F
            BCF INTCON,1
            RETFIE
```

```
MOVLW B'10100000'
MOVWF INTCON
```

Setting bit 7 of INTCON enables the program to respond to any interrupts generated. Setting INTCON bit 5 enables the timer as the source of the interrupt. The stage is now set and the START loop entered. Each time a timer interrupt occurs, a jump is made to INTRPT, where PORTB has a value of 2 added to it (to bypass LD0) and a return made to START by the command RETFIE, to await another interrupt.

To prove that the program is not just "dropping out" of the START loop, a command to set PORTA RA0 high has been included immediately following the GOTO START command. As you will see via l.e.d. LD7, this action is never performed.

Load and run TK3TUT34.HEX which illustrates this interrupt.

EXTERNAL INTERRUPT

If, instead of using the timer to generate interrupts, we want an external source to generate them, one pin that can be used for this purpose is PORTB RB0, designated in the pinout diagram as RB0/INT. (Logic level changes on PORTB RB4 to RB7 are other possible interrupt sources.) To use RB0 as the interrupt source, INTCON bit 4 must be set, as follows:

```
MOVLW B'10010000'
MOVWF INTCON
```

INTCON bit 7 must, as shown, also be set to enable the interrupt.

(Note that if the l.c.d. is connected, the PIC's Light Pull-ups option, discussed later, must be off by setting OPTION_REG bit 7 high, as shown in Listing 35, otherwise the influence of the l.c.d. may prevent the interrupt from being generated.)

Suppose now that we want an external interrupt on RB0 to cause the rest of PORTB to be incremented. Each time this interrupt occurs, the jump from the holding loop is performed as before. However, it is now INTCON bit 1 which is set on the interrupt and has to be cleared before returning to the holding loop, i.e. BCF INTCON,1.

Load TK3TUT35 which illustrates this external interrupt. The interrupt is generated using switch SW0.

Since the switches used on the p.c.b. are probably only low-cost types, it is possible that switch-bounce will cause slightly erratic behaviour of the l.e.d.s. It should become clear, however, that the count is basically incremented when the switch is pressed, not when it is released.

If a signal generator that outputs a square wave is connected to RB0 and monitored on a scope, the triggering edge should be obvious when the generator's rate is set very slow. The signal generator must produce clean 0V to +5V pulses.

INTERRUPT EDGE

It is possible to change the interrupt response to occur on either edge of the external pulse. As illustrated in TK3TUT35, it is in response to the rising edge. To use the falling edge, OPTION_REG bit 6 must be cleared during the BANK1 setup routine, e.g.:

TABLE 6: INTCON REGISTER (Courtesy MICROCHIP)

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
bit7							bit0

R = Readable bit
W = Writable bit
U = Unimplemented bit, read as '0'
- n = Value at POR reset

bit 7: **GIE:** Global Interrupt Enable bit
1 = Enables all un-masked interrupts
0 = Disables all interrupts

Note: For the operation of the interrupt structure, please refer to Section 8.5.

bit 6: **EEIE**: EE Write Complete Interrupt Enable bit
1 = Enables the EE write complete interrupt
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 Interrupt Enable bit
1 = Enables the RB0/INT interrupt
0 = Disables the RB0/INT 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 has overflowed (must be cleared in software)
0 = TMR0 did not overflow

bit 1: **INTF**: RB0/INT Interrupt Flag bit
1 = The RB0/INT interrupt occurred
0 = The RB0/INT interrupt did not occur

bit 0: **RBIF**: RB Port Change Interrupt Flag bit
1 = When 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

```
MOVLW B'10000111'
MOVWF OPTION_REG
```

Change TK3TUT35 to respond to the falling edge and observe the result when you now press switch SW0. Note that the settings of OPTION_REG bits 0, 1 and 2 are irrelevant in this interrupt mode.

As you have seen, INTCON bit 7 is used for enabling (1) and disabling (0) the interrupts, in addition to any other bits required for an interrupt to be enabled. It is possible that at the moment of wishing to disable the interrupts, however, that an interrupt could be in the process of occurring. This would result in the disabling command not taking effect. To ensure that all interrupts are fully disabled (except WDT – see later), the following routine can be used:

```
DISABL  BCF INTCON,GIE
        BTFSC INTCON,GIE
        GOTO DISABL
```

The term GIE is that equated for use as INTCON bit 7. It should be equated as such in the initialising commands. Its use is in keeping with the PIC datasheet, which calls this bit by that name, standing for Global Interrupt Enable.

Malcolm discusses disabling GIE in his *Interrupts* article.

EXERCISE 27

27.1. Modify one of the early counting programs so that it is automatically triggered by an interrupt from line RB0 without the need to read the INTCON register flag.

27.2. You know that INTCON bit 1 and INTCON bit 2 are both flags for interrupts. Modify your program from 27.1 so it automatically responds to interrupts from RB0 and from the TMR0 timer.

Hint: once an interrupt has occurred, the INTCON flags can be read to see which source has caused the interrupt. You can also inhibit one interrupt from occurring while you process the first by using other INTCON bits. Use the l.e.d.s to show respective counts from each source.

TUTORIAL 28

CONCEPT EXAMINED

Command SLEEP

CONNECTIONS NEEDED

SW0 to RB0
LD0 to RA0
LD1-LD7 to RB1-RB7
CP20 to +5V OUT
CP21 to 0V OUT
Crystal oscillator

SLEEP is a command that is rarely likely to find use by most readers. The function sets the PIC into a very low current power-down mode. This can be useful if the PIC is monitoring or controlling something at a very slow rate. In this situation, there are power saving advantages if the PIC can be put to sleep during periods when it is not required to perform. The PIC can be awoken from SLEEP by a WDT time-out or through an external interrupt.

The program which illustrates the latter is TK3TUT36. First connect l.e.d. LD0 to RA0, and switch SW0 to RB0, then load the program.

The program adds two to the count on PORTB from zero up to the roll-over at 256, at which point PORTA RA0 is set to turn on l.e.d. LD0. At this point, the program is told to SLEEP.

It can only be awoken by pressing switch SW0. Whereupon, RA0 is cleared to turn off LD0, and the PORTB count resumes, until again it rolls over to zero and setting RA0, then falling asleep once more. (This might remind you of your occasional behaviour on a Monday morning after "the weekend before"!) Note the use of two delays (DELAY1 and DELAY2) slowing the program down by 256 × 256 looped actions for the sake of the demo.

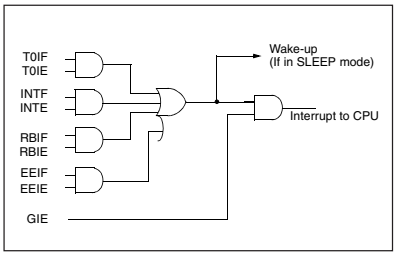


Fig.9. PIC16F84 interrupt logic.

LISTING 36 – PROGRAM TK3TUT36

```
        CLRF PORTA
        CLRF PORTB
        BANK1
        CLRF TRISA
        MOVLW B'00000001'
        MOVWF TRISB
        MOVLW B'11000111'
        MOVWF OPTION_REG
        BANK0
        CLRF DELAY1
        CLRF DELAY2
        MOVLW B'10010000'
        MOVWF INTCON

MAIN    DECFSZ DELAY1,F
        GOTO MAIN
        DECFSZ DELAY2,F
        GOTO MAIN
        MOVLW 2
        ADDWF PORTB,F
        BTFSS STATUS,C
        GOTO BYPASS
        BSF PORTA,0
        SLEEP

BYPASS  BCF INTCON,1
        BCF PORTA,0
        GOTO MAIN
```

EXERCISE 28

28.1. Put the PIC to sleep between each detection of a TMR0 interrupt occurring every 1/25th of a second while allowing it to appropriately increment a seconds counter and show its value on any of the display types covered.

TUTORIAL 29

CONCEPTS EXAMINED

Watchdog timer (WDT)
Command CLRWDT

CONNECTIONS NEEDED

SW0 to RB0
LD0 to RA0
LD1-LD7 to RB1-RB7
CP20 to +5V OUT
CP21 to 0V OUT
Crystal oscillator

The Watchdog Timer (WDT) facility is also probably one for which most readers are unlikely to find much use. The purpose of a WDT is to give the PIC a type of protection against becoming stuck in a perpetual loop. This can happen in several ways, but particularly in the event of unforeseen program errors, or waiting for an external event to happen but which does not (for many and varied reasons, including equipment malfunction).

It is also possible for electrical spikes on power lines to cause the malfunction, although it can be argued that the use of a good power supply should be mandatory in situations where this could be an unacceptable problem.

In effect, the WDT provides a "last-ditch" time-out timer which, if it is allowed to time-out, causes a complete system reset. The idea is that the WDT is set with a timing value, and then at regular intervals in the main loop of the program, this value is repeatedly reloaded into it, i.e. it is reset, using the command CLRWDT. Should a problem occur which prevents the WDT value from being reloaded, the WDT will timeout and cause a full program reset.

The difficulty of using a WDT in many programs is that when the full reset occurs, any variables which are specifically set to known values at the start of the program will once more be reset to them. This means, for example, that event counters within the program will also be reset.

When the existing count value is of importance, rather than use the WDT, the program should be written so that an interrupt (from a switch, for instance) can cause the program to resume running without being reset. However, if it does not matter that the program restarts from the beginning, as in some burglar alarm systems perhaps, then the WDT can be beneficially used.

To use the WDT, the PIC has to be set for this function using the PIC Configuration program. You will recall that when configuring the PIC for RC and crystal modes, WDT was not selected. Now, though, reconfigure the PIC and set the Watchdog on.

Connect RB0 to SW0 instead of LD0, then load TK3TUT37.

WATCH IT!

Observing the l.e.d.s on PORTB, press and release switch SW0, setting on l.e.d.s LD7 to LD1. After a brief pause following the switch being released, the l.e.d.s will all go out as the WDT times out, causing a program reset. Repeatedly pressing SW0 overrides the WDT, resetting its count-down value.

The WDT timing period can be changed in the same way that we set the timing prescaler for the real-time clock, i.e. using bits 0 to 2 of OPTION_REG. Bit 3 of OPTION_REG must always be set so that the prescaler is allocated to the WDT. Try changing the values of OPTION_REG bits 0 to 2; also see what happens when STATUS bit 3 is set to zero. The prescaler is set for its slowest rate in TK3TUT37.

The WDT cannot be disabled from within an operational program. It can only be turned off by reconfiguring the PIC. Consequently, when you have finished experimenting with the WDT, once again reconfigure the PIC with WDT disabled. Unless you do this, none of the other demonstration programs will run correctly.

An independent RC oscillator is used by the WDT and its timing is unaffected by the frequency of the external oscillator that controls the rest of the PIC.

Be aware that during development of the Tutorials, it was found necessary to occasionally run the WDT configuration twice before the PIC would accept this mode. The reason is unknown. If TK3TUT37 does not behave as expected, re-run the configuration for WDT.

TABLE 7: OPTION REGISTER (Courtesy MICROCHIP)

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
$\overline{\text{RBPU}}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit7							bit0

R = Readable bit
W = Writable bit
U = Unimplemented bit, read as '0'
- n = Value at POR reset

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 assigned to the WDT
0 = Prescaler assigned to TMR0

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

EXERCISE 29

29.1. Experiment with different timeout periods for WDT, using the OPTION_REG register settings.

TUTORIAL 30

CONCEPT EXAMINED

Misc Special Register bits

We have examined the use of quite a few bits in the Special Functions Registers, but not all of them (see Tables 4, 6 and 7).

OPTION BIT 7

The first bit that we have not demonstrated, although earlier reference has been made to it, is the "light pull-ups" bit of the OPTION register, bit 7. The pull-ups facility allows switches to be used without biassing resistors as it introduces its own pull-up biassing on each of PORTB's pins that are used as inputs (assume as a rule-of-thumb that biassing has an equivalent resistance of about 100kΩ).

Throughout the Tutorials so far, OPTION_REG bit 7 has been held high, initially because the default setting for the register is 11111111, and secondly because when we have been using the register we have been setting bit 7 high in the initialisation block.

We have not needed the pull-ups to be on because the switches have usually been connected to PORTA (which does not have the facility) and they have their own external biassing resistors connected (R17 to R20). In other applications, though, switches can be used on PORTB and external biassing resistors omitted, using the command BCF OPTION_REG,7 to activate the internal pull-ups.

Making use of these pull-ups, however, means that PORTB's input pins are active low (rather than active high as in most of the switch monitoring examples in the Tutorials). This means that the PORTB pins to which the switches are connected are normally held high, a switch press then taking them low. Consequently, it is the low condition which needs to be looked for when a switch is pressed.

A convenient way of detecting if any switch has been pressed is to read PORTB as an inverted value, using the command COMF PORTB,W and read the Z flag of STATUS. If a switch has not been pressed a zero condition will exist following COMF. However, if one or more switches have been pressed, a zero condition will not exist and appropriate action can then be taken, as in the following example:

```
TESTSW  COMF PORTB,W    ; invert
                          PORTB
                          into W
        BTFSS STATUS,Z  ; is result =
                          0?
        GOTO ACTION     ; no, a
                          switch has
                          been
                          pressed, so
                          process it
```

LISTING 37 – PROGRAM TK3TUT37

```
        CLRF PORTA
        CLRF PORTB
        BANK1
        CLRF TRISA
        MOVLW B'00000001'
        MOVWF TRISB
        MOVLW B'10001111'
        MOVWF OPTION_REG
        BANK0

TESTON  BTFSS PORTB,0
        GOTO TESTON
        CLRWDT
        MOVLW 255
        MOVWF PORTB
        GOTO TESTON
```

```
        RETURN          ; yes, a
                          switch
                          has not
                          been
                          pressed
ACTION
; (routine that results if a switch has been
pressed goes here)
        RETURN
```

Following COMF an AND statement can be made to eliminate those pins which are not associated with the switches.

You could try the light pull-ups option by connecting "flying leads" to the PORTB pins and touch their stripped ends to the +5V and 0V lines. It is suggested that you modify some of the earlier routines that use switches to prove how light pull-ups can be used.

It is worth appreciating that if PIC pins set as inputs are not connected to anything when the light pull-ups are off, erratic behaviour can result as the pins are not sure which condition they should be in, high or low (see the section in Tutorial 4, Part 1, that discussed port pin safety).

OTHER BITS NOT DISCUSSED

There are two bits in the STATUS register (see Table 4, Part 1) whose purpose and use seem obscure:

STATUS bit 3 (PD): POWER-DOWN bit. Set to 1 during power up or by a CLRWDT command. Cleared to 0 by a SLEEP command.

STATUS bit 4 (TO): TIME-OUT bit. Set to 1 during power up and by the CLRWDT and SLEEP commands. Cleared to 0 by a WDT time-out.

There are also other bits which are similar in their setting to those that we have already discussed, and so their examination is not justified here. The bits are principally in the INTCON and OPTION_REG registers (Tables 6 and 7). A summary is as follows:

INTCON bit 0 (RBIF): RB port change interrupt flag. Set when any of RB4 to RB7 inputs change logic state. Has to be reset in software.

INTCON bit 3 (RBIE): RBIF interrupt enable bit; 0 = disable, 1 = enable.

(Malcolm discusses INTCON RBIF and RBIE is his *Interrupts* article.)

OPTION_REG bit 5 (RTS): TMR0 signal edge response to signal on RA4/TOCKI pin;

0 = increment on low-to-high transition
1 = increment on high-to-low transition.

Note that the default value for each bit in OPTION_REG at power-up and reset is 1 (i.e. 11111111).

It is suggested that you write simple routines, along the lines of those that have been used in other demonstrations, to establish for yourself what can be achieved using these bits. It is also well worthwhile reading through the PIC's datasheet in its entirety. There are minor aspects relating to some of the commands that we have discussed that deserve recognition if you wish to delve more deeply into programming these devices.

TUTORIAL 31

CONCEPT EXAMINED

INCLUDE files command
Embedded Configuration data
Embedded Data EEPROM values
Embedded PIC type data
Embedded Radix

CONNECTIONS NEEDED

L.C.D. as in Fig.7
CP20 to +5V OUT
CP21 to 0V OUT
Crystal oscillator

INCLUDE FILES COMMAND

It is possible to input "library" files into the main body of your program. This can be done in one of two ways: either by copying the section from a previous program (as preferred by the author), or to call in a particular file to be "included" at an appropriate point within the program.

The latter files can have any extension but it is customary to give them an INC extension, standing for "include" and can be called in by name as in the two examples given in Listing 38, and any number of INC files can be called in.

In this example, the first included file is called by the command:

INCLUDE TK3PIC16F84.INC

This file is a variant of one of Microchip's Include files and contains the full range of EQUated register and bit values as listed in the datasheet for the PIC16F84.

Microchip have INClude files available for other PIC types (see later). It is **essential** that the EQUates INClude file should be for the PIC family for which the main code is written.

The second INClude file is at the program's end. The named file simply contains the program code for the LCDFRM and LCDOUT routines that are used in TK3TUT30. The full listing of the example program in TK3TUT38 shows that the main body of the code is the same as that used in TK3TUT30, but without the EQUates held in TK3PIC16F84.INC, and without the LCDFRM and LCDOUT routines.

There are a number of restrictions that apply when using INClude files, some of which may depend on the assembly program you use. *TK3* operates on the following principle:

- The INCluded filename must not contain directory (folder) or drive path information
- The file must be in the same directory as the main calling code. Thus if the directory address of the main code is C:\PIC\TestFile.ASM then the INClude file must also be within directory C:\PIC\ e.g.:

C:\PIC\TK3PIC16F84.INC

There are two main regions in PIC source codes from which INClude files may be called when using *TK3*. Those containing EQUates and Defines must be placed near the top of the main code in the region where the main code EQUates and Defines are placed. They must occur before the first ORG statement is made in the main code. INClude files called in this region must *not* contain true program commands.

The second region is at any suitable point within the body of the main code, and may be the first command of that code (i.e. at address location 5 – ORG 5), at the end as shown here, or anywhere else that you prefer.

Include files in the main body of the code may contain their own Equates and Defines, but not Includes. Beware of using ORG statements within Include files since they may disrupt the correct assembly of the rest of the code.

In this example the Defines are stated first, then the basic EQUates data is called in, then the EQUates specifically required as register names of your choice are then stated. These must include the register names needed by any INCluded program code routines, in this case those needed by

LISTING 38 – PROGRAM TK3TUT38

```
#DEFINE BANK0 BCF STATUS,5
#DEFINE BANK1 BSF STATUS,5

        INCLUDE TK3PIC16F84.INC ; call in EQUates data held in INC file

LOOP    EQU H'20'               ; loop counter 1 – general
CLKCNT  EQU H'21'               ; pre-counter for CLOCK
CLKSEC  EQU H'22'               ; CLOCK main counter – secs
CLKMIN  EQU H'23'               ; CLOCK – mins
CLKHRS  EQU H'24'               ; CLOCK – hours
STORE1  EQU H'25'               ; general store 1
STORE2  EQU H'26'               ; general store 2
RSLINE  EQU H'27'               ; RS line flag for LCD
LOOPA   EQU H'28'               ; loop counter for LCD
        ORG 0                   ; Reset Vector address
        GOTO 5                  ; go to PIC address location 5
        ORG 4                   ; Interrupt Vector address
        GOTO 5                  ; go to PIC address location 5
        ORG 5                   ; Start of Program Memory at location 5
; (main body of program)

        INCLUDE TK3TUT38.INC    ; call in program code held in INC file

        END                     ; final line
```

TK3TUT38.INC (but also see CBLOCK in Tutorial 48).

Whilst some assemblers (including *TK3*) permit INClude files to contain EQUates and program code, users need to be aware of the danger of the same EQUates names being used for different purposes in different INC files when more than one is called.

When using *TK3* as the assembler, INClude files must *not* contain Macros, which *TK3* does not recognise.

(A Macro is a set of programmer-defined instructions and directives which are given a name. When the name is encountered by the assembler, it is automatically expanded into the defined set of instructions. Thus a Macro is a useful shorthand notation for sequences of code that recur frequently within the program. The subject of Macros is covered in Malcolm Wiles' *PIC Macros and Computed GOTOs, EPE* Jan '03, on the PIC Resources CD-ROM.)

EMBEDDED CONFIGURATION DATA

At various points in these Tutorials you have had to set the PIC's configuration data via the PIC Configuration facility. It is possible to embed the data into the program itself rather than set it separately. This ensures that when the program is loaded the PIC is automatically configured correctly for the purpose required.

If you look to the right of the Config screen you will see a hexadecimal value that changes when the various Config option buttons are used. It is this hex value which is used at the head of a program in the form:

```
__CONFIG H'3FF1'
```

The value is preceded by __CONFIG (two underscore characters plus the word CONFIG). When the program is assembled the CONFIG value is noted and then automatically programmed into the designated PIC location (at H'2007'). The statement is placed in the second column of the assembly code, as shown in the full program of TK3TUT38.ASM.

To establish what value to use for a particular program, simply set the Config option buttons to the functions required and use the resulting hex value shown.

The value in this example is for Code Protection (CP) off, Power on Reset (POR) off, Watchdog Timer (WDT) off, crystal XT (crystal frequencies up to 4MHz).

Near the head of TK3TUT38 you will also see a statement commencing LIST P, which will be discussed shortly.

EMBEDDED DATA EEPROM VALUES

In program TK3TUT33 the values for the Data EEPROM are entered via switches on the p.c.b. It is possible to set such values into the program itself and which are then automatically placed into the Data EEPROM at the correct locations when the program is loaded.

The PIC's Data EEPROM region commences at H'2100' and to program it the ASM file requires an ORG statement at the end pointing to this location, e.g.

```
ORG H'2100'
```

The values to be sent are prefixed "DE" and then entered in any of the numerical forms, such as follow:

```
DE 10, 9, 8 ,7 ,6 ,5, 4, 3, 2, 1, 0
DE 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
DE H'A', H'B', H'C', H'FF'
DE 'A', 'B', 'c'
DE B'00000001', B'00000011'
```

When the data is sent it is placed at consecutive addresses commencing from the ORG value stated.

It is permissible to start the data at later locations, for example H'2109', which would place the first value at EEPROM location 9, rather than 0 as stated by H'2100 '.

EMBEDDED PIC TYPE DATA

TK3 and Microchip's MPASM assembler allows the PIC type to be embedded into the program. Whilst *TK3* does not need this to be embedded, the statement is useful if the program is to be assembled by MPASM (as might be the case with programs distributed to *EPE* readers who use MPASM but not *TK3*).

TK3 only makes use of it to set the basic PIC type, as can also be done through the Select PIC Type screen option.

The PIC type statement is made in the form:

```
LIST P = PIC16F84
```

and placed at the head of the program, as shown in TK3TUT38.ASM.

EMBEDDED RADIX

As stated several times in these Tutorials, numerical values can be expressed in various ways, depending on the prefix used (e.g. H' or B'). *TK3* automatically assumes that values without a prefix are in decimal.

However, *TK3* also follows MPASM's option and allows users to express hex or binary values without the use of a prefix and enclosing quotes, defining this requirement in an initialisation line which is prefixed with the statement "LIST". The term *radix* is used in this context, and the radix is then equated to be in decimal, hex or binary, according to the user's preference. For example, the following sets the radix to be in hexadecimal:

```
LIST R=HEX
```

Having specified the radix, any unprefixed value encountered during assembly will be taken to be in that notation. Thus if the radix is decimal (R=DEC), 10 will be taken as decimal ten. If the chosen radix is hex (R=HEX), then 10 will be taken as H'10' (decimal 16). For a binary radix (R=BIN), 10 will be interpreted as B'00000010' (decimal 2).

Obviously, when using a radix of hexadecimal or binary, any decimal values must use the D' prefix and final quote, e.g. D'10'.

A single LIST statement can be used to apply to the PIC type and the radix, separating the two definitions by a comma, i.e.

```
LIST P = PIC16F84, R=DEC
```

It is worth noting perhaps that the author has never wished to use a radix other than decimal.

EXERCISE 31

31.1 Modify TK3TUT38 so that it calls in the PAUSIT routine as a separate INC file, named as you wish, and used in place of that routine in the main code.

31.2 Examine printouts of the LST files created for TK3TUT30, TK3TUT38 and your own program as modified in 31.1. Confirm for yourself that the three listings contain the same equates and program code. (Additional information about the include files will be seen in the last two listings.)

31.3 Set the CONFIG value of TK3TUT38 so that program runs in RC mode (set the RC/XTAL switch appropriately otherwise the program will not run).

31.4 Modify program TK3TUT33 in conjunction with TK3TUT19 so that a tune is held as Data EEPROM statements which are played when a switch is pressed. There are 64 Data EEPROM locations available (from H'2100' to H'213F').

Note that *TK3* also allows Data EEPROM values to be sent via a MSG (message) file but ignore the facility for this exercise (the facility is not available with all assembly programs).

TUTORIAL 32

CONCEPT EXAMINED

PIC16F8x, PIC16F87x, PIC16F6x family coding differences
PIC16F87x PORTA
PIC16F87x Data EEPROM use
PIC16F62x PORTA
PIC16F62x Data EEPROM use

We said earlier that most aspects of using a PIC16F84 are common to other PIC families, but that there are some differences. Whilst these tutorials are not intended to cover all aspects of PIC programming, it is worth highlighting some routines that have been discussed in relation to the PIC16F84 and for which slightly different techniques are required for the PIC16F87x and PIC16F62x families. This section looks at those differences.

EXTRA BANKS

The first thing to note is that the PIC16F87x and PIC16F62x devices both have four Banks, whereas the PIC16F84 only has two. The extra two Banks are numbered 2 and 3. STATUS bits RP0 and RP1 (bits 5 and 6) control Bank selection. It is preferable to ensure that only BANK0 is active when the program starts by issuing the follow commands immediately prior to the initialisation block:

```
BCF STATUS,RP0
BCF STATUS,RP1
```

RP0 and RP1 should be equated in the general EQUates section as:

```
RP0     EQU 5   ; STATUS reg
RP1     EQU 6   ; STATUS reg
```

Unlike the PIC16F8x family, PIC16F87x and PIC16F62x devices

LISTING 39A

```
EEDATA  EQU H'0C'  ; Bank 2
EECON1  EQU H'0C'  ; Bank 3
PIR2    EQU H'0D'  ; Bank 0
EEADR   EQU H'0D'  ; Bank 2
EECON2  EQU H'0D'  ; Bank 3
STATUS  EQU 3      ; STATUS register
W       EQU 0      ; Working register
                     flag
RP0     EQU 5      ; STATUS bank
                     reg
RP1     EQU 6      ; STATUS bank
                     reg
RD      EQU 0      ; EECON1 reg
EEPGD   EQU 7      ; EECON1 reg
EEIF    EQU 4      ; PIR2 reg
WR      EQU 1      ; EECON1 reg
WREN    EQU 2      ; EECON1 reg
STORE1  EQU H'20'  ; or any conve-
                     nient address
```

have additional memory that can be accessed in their Banks 1, 2 and 3. Accessing this memory is accomplished via PCLATH. See the author's *PIC16F87x Extended Memory,* June '01, and John Waller's *PCLATH* text referred to earlier. Both texts are on the PIC Resources CD-ROM.

PIC16F87x PORTA

PIC16F87x devices allow their PORTA pins to be variously used for analogue input purposes as well as digital. The default condition prior to a program being run is for the analogue aspect to be active. This means that the PIC has to be told when PORTA is to be used for normal digital input/output, normally making this statement as part of the initialisation process.

To set PORTA for digital use, the ADCON1 Special Register value has to be set via BANK1 to 0000011x (where x means that the value of that bit does not matter), e.g.:

```
BANK1
MOVLW B'00000111
MOVWF ADCON1
BANK0
```

These MOVLW and MOVWF commands would normally be included in the BANK0 to BANK1 section of the program's initialisation routine where TRISA etc. are being set.

ADCON1 should be equated in the general EQUates section as:

```
ADCON1  EQU H'1F'        ; (Bank 1)
```

PIC16F87x DATA EEPROM USE

The PIC16F87x family need the EQUates shown in Listing 39A when the Data EEPROM read/write routines are used. The full Write and Read listings are held in TK3TUT39.ASM.

PIC16F87x WRITING TO DATA EEPROM

Writing to the PIC16F87x family is carried out by the routine in Listing 39B, which is entered with W holding the EEPROM byte address at which data is to be stored. The data to be stored is held in STORE1.

LISTING 39B

```
SETPRM  BSF STATUS,RP1      ; set for Bank 2
        BCF STATUS,RP0
        MOVWF EEADR         ; copy W into EEADR to set EEPROM address
        BCF STATUS,RP1      ; set for Bank 0
        MOVF STORE1,W       ; get data value from STORE1 and hold in W
        BSF STATUS,RP1      ; set for Bank 2
        MOVWF EEDATA        ; copy W into EEPROM data byte register
        BSF STATUS,RP0      ; set for Bank 3
        BCF EECON1,EEPGD    ; point to Data memory
        BSF EECON1,WREN     ; enable write flag
MANUAL  MOVLW H'55'         ; these lines cause the action required
        MOVWF EECON2        ; by the EEPROM to store the data in EEDATA
        MOVLW H'AA'         ; at the address held by EEADR.
        MOVWF EECON2
        BSF EECON1,WR       ; set the "perform write" flag
        BCF STATUS,RP1      ; set for Bank 0
        BCF STATUS,RP0
CHKWRT  BTFSS PIR2,EEIF     ; wait until bit 4 of PIR2 is set
        GOTO CHKWRT
        BCF PIR2,EEIF       ; clear bit 4 of PIR2
        RETURN
```

LISTING 39C

```
PRMGET  BSF STATUS,RP1      ; set for Bank 2
        BCF STATUS,RP0
        MOVWF EEADR         ; copy W into EEADR to set EEPROM address
        BSF STATUS,RP0      ; set for Bank 3
        BCF EECON1,EEPGD    ; point to data memory
        BSF EECON1,RD       ; enable read flag
        BCF STATUS,RP0      ; set for Bank 2
        MOVF EEDATA,W       ; read EEPROM data now in EEDATA into W
        BCF STATUS,RP1      ; set for Bank 0
        RETURN
```

LISTING 40B

```
SETPRM  BANK1
        MOVWF EEADR         ; copy W into EEADR to set EEPROM address
        MOVF PROMVAL,W      ; get data value from PROMVAL and hold in W
        MOVWF EEDATA        ; copy W into EEPROM data byte register
        BSF EECON1,WREN     ; enable write flag
MANUAL  MOVLW H'55'         ; these lines cause the action required by
        MOVWF EECON2        ; by the EEPROM to store the data in EEDATA
        MOVLW 'AA'          ; at the address held by EEADR.
        MOVWF EECON2
        BSF EECON1,WR       ; set the "perform write" flag
        BANK0
CHKWRT  BTFSS PIR1,EEIF     ; wait until bit 4 of PIR2 is set
        GOTO CHKWRT
        BCF PIR1,EEIF       ; clear bit 4 of PIR2
        RETURN
```

PIC16F87x READING FROM DATA EEPROM

Reading from the PIC16F87x family is carried out by the routine in Listing 39C, which is entered with W holding the EEPROM byte address to be read. The readback value is then held in W on exit.

PIC16F62x PORTA

PIC16F62x devices allow their PORTA pins to be variously used for analogue input purposes as well as digital. The default condition prior to a program being run is for the analogue aspect to be active. This means that the PIC has to be told when PORTA is to be used for normal digital input/output, normally making this statement as part of the initialisation process. To do so register CMCON has to be set with the value of 7, e.g.:

```
MOVLW 7
MOVWF CMCON
```

LISTING 40A

```
PIR1    EQU H'0C'  ; Bank 0
EEDATA  EQU H'1A'  ; Bank 1
EEADR   EQU H'1B'  ; Bank 1
EECON1  EQU H'1C'  ; Bank 1
EECON2  EQU H'1D'  ; Bank 1
WREN    EQU 2      ; EECON1 reg
                     EEPROM
                     write enable
                     flag
EEIF    EQU 7      ; PIR2 reg
RD      EQU 0      ; EECON1 reg
                     EEPROM
                     read enable
                     flag
WR      EQU 1      ; EECON1 reg
                     EEPROM
                     write initiate
                     flag
PROMVAL EQU H'70'  ; accessed via
                     Bank 0 and
                     Bank 1
```

This action takes place in BANK0 and can be done immediately prior to the BANK1 initialisation block.

CMCON should be equated in the general EQUates section as:

```
CMCON    EQU H'1F'    ; Bank 0
```

PIC16F62x DATA EEPROM USE

The PIC16F62x family need the EQUates shown in Listing 40A when the Data EEPROM read/write routines are used:

Note that one value, PROMVAL, is required to be accessed via both BANK0 and BANK1. It needs to be EQUated to occur at a location between H'70' and H'7F'. These locations are automatically accessible via any of the four Banks, unlike the "normal" locations between H'20' and H'6F' (see Figure 4.3 in the PIC16F62x datasheet).

The full Write and Read listings are held in TK3TUT40.ASM.

PIC16F62x WRITING TO DATA EEPROM

Writing to the PIC16F62x family is carried out by the routine shown in Listing 40B, which is entered with W holding the EEPROM byte address at which data is to be stored. The data to be stored is held in PROMVAL, which is located in both Banks at or above H'70'

PIC16F62x READING FROM DATA EEPROM

Reading EEPROM data from the PIC16F62x family is carried out by the routine in Listing 40C, which is entered with W holding the EEPROM byte address to be read. The read-back value is then held in W on exit.

LISTING 40C

```
GETPRM BANK1
       MOVWF EEADR    ; copy W into EEADR to set EEPROM address
       BSF EECON1,RD  ; enable read flag
       MOVF EEDATA,W  ; read EEPROM data now in EEDATA into W
       BANK0
       RETURN
```

TUTORIAL 33

CONCEPT EXAMINED

Converting binary values to decimal

CONNECTIONS NEEDED

L.C.D. as in Fig.7
CP20 to +5V OUT
CP21 to 0V OUT
Crystal oscillator

You will recall that the clock counting examples were carried out using binary coded decimal (BCD) routines, which then made outputting the data to the 7-segment and l.c.d. displays comparatively easy. There are many occasions when it is considerably more convenient to process data in binary than in BCD and then to convert it to BCD prior to display.

An excellent routine for doing this conversion was provided to *EPE* by reader Peter Hemsley. It allows a 3-byte value to be converted to eight separate BCD digits. An example of using his code in a practical situation is held in TK3TUT41.ASM – run it and then examine its code.

Peter's code can be copied in to your program and used as a "library" routine, accessed via preparatory commands such as shown in Listing 41.

The values BIN0, BIN1 and BIN2 in the listing hold the processed values (in order of LSB, NSB, MSB), which are then copied into COUNT0, COUNT1 and COUNT2 prior to conversion.

A call to Peter's BIN2DEC routine is then made, after which the values are ORed with 48 to suit them to the l.c.d. and leading zeros are blanked. The decimal results are then output to the l.c.d. on line 1 starting at cell 0. If BIN1 and/or BIN2 are not used, they should be cleared prior to calling the BIN2DEC routine (e.g. CLRF COUNT2 if BIN2 is not used).

Note that command SKPNC in the BIN2DEC routine is an MPASM shortform for "Skip if no Carry". A list of such commands recognised by *TK3* and Microchip's MPASM assembly programs is shown later in Table 8.

LISTING 41 – PROGRAM TK3TUT41

```
; Binary to BCD conversion example
            MOVF BIN0,W          ; copy binary into COUNT
            MOVWF COUNT0
            MOVF BIN1,W
            MOVWF COUNT1
            MOVF BIN2,W
            MOVWF COUNT2
            CALL BIN2DEC         ; call conversion
            MOVLW B'10000000'    ; set LCD line
            CALL LCDLIN
            MOVF DIGIT8,W        ; output to LCD
            CALL LCDOUT
(repeat for DIGIT7 to DIGIT1)
            RETURN
```

EXERCISE 33

33.1 Experiment with using different values held in BIN.

33.2 Write a clock program in which the hours, minutes and seconds are each incremented as binary values and output to the l.c.d. in decimal.

TUTORIAL 34

CONCEPT EXAMINED

Multiplication routine

CONNECTIONS NEEDED

L.C.D. as in Fig.7
CP20 to +5V OUT
CP21 to 0V OUT
Crystal oscillator

PIC16F84, PIC16F62x and PIC16F87x families do not have multiplication commands. *EPE* reader Peter Hemsley's code for multiplying a 2-byte value by another 2-byte value is shown TK3TUT42.ASM. Run it and then examine its code.

An example of its use is shown in Listing 42. The binary value to be multiplied is held in BIN0 (LSB) and BIN1 (MSB) and copied into MULCLSB and MULCMSB. The value by which BIN is to be multiplied is placed into MULPLSB and MULPMSB and may be a numeral as shown (H'0575) or from another pair of bytes whose value has been set elsewhere in the program. The answer is held, in descending order of bytes, in PRODMSB, PRODLSB, MULPMSB and MULPLSB,

LISTING 42 – PROGRAM TK3TUT42

```
            MOVF BIN0,W          ; place value to be multiplied into MULC
            MOVWF MULCLSB
            MOVF BIN1,W
            MOVWF MULCMSB
            MOVLW H'75'          ; place value of multiplier into MULP
            MOVWF MULPLSB
            MOVLW H'05'          ; (multiply by H'0575')
            MOVWF MULPMSB
            CALL MULTIPLY        ; call multiply routine
            MOVF MULPLSB,W       ; copy answer into ANSWER
            MOVWF ANSWER0
            MOVF MULPMSB,W
            MOVWF ANSWER1
            MOVF PRODLSB,W
            MOVWF ANSWER2
            MOVF PRODMSB,W
            MOVWF ANSWER3
            RETURN
```

which are then copied into the four bytes of ANSWER for use elsewhere as required.

If the MSBs of MULC or MULP are not used, they should be cleared before calling MULTIPLY.

Peter's code can be copied into your program and used as a "library" routine, accessed via preparatory commands such as shown in Listing 42.

EXERCISE 34

34.1 Experiment with different BIN and multiplying values.

34.2 Using the Multiply routine, extend the clock program in TK3TUT30.ASM so that a total seconds count since midnight is shown in addition to the basic hours, minutes and seconds. Hint, first convert the BCD values to binary. Several more EQUated registers will be needed to hold the interim calculations.

TUTORIAL 35

CONCEPT EXAMINED

Division routine

CONNECTIONS NEEDED

L.C.D. as in Fig.7
CP20 to +5V OUT
CP21 to 0V OUT
Crystal oscillator

PIC16F8x, PIC16F62x and PIC16F87x families do not have division commands. Peter Hemsley's code for dividing a 2-byte value by another 2-byte value is shown TK3TUT43.ASM.

An example of its use is shown in Listing 43. The binary value to be divided is held in BIN0 (LSB) and BIN1 (MSB) and copied into DIVIDLSB and DIVIDMSB. The value by which BIN is to be divided is placed into DIVISLSB and DIVISMSB and may be a numeral as shown (H'0103') or from another pair of bytes whose value has been set elsewhere in the program. The answer is held, in descending order of bytes, in DIVIDMSB, DIVIDLSB, REMDRMSB, REMDRLSB (REMDR being the undivided remainder). DIVIDMSB and DIVIDLSB are then copied into ANSWER1 and ANSWER0 for use elsewhere as required.

If either DIVIDMSB or DIVISMSB are not used, the respective one should be cleared before calling DIVISION.

Peter's code can be copied into your program and used as a "library" routine, accessed via preparatory commands such as shown in Listing 43.

LISTING 43 – PROGRAM TK3TUT43

```
        MOVF BIN0,W
        MOVWF DIVIDLSB
        MOVF BIN1,W
        MOVWF DIVIDMSB
        MOVLW 3
        MOVWF DIVISLSB
        MOVLW 1
        MOVWF DIVISMSB
        CALL DIVISION
        ; divide by H'0103'
        MOVF DIVIDLSB,W
        MOVWF ANSWER0
        MOVF DIVIDMSB,W
        MOVWF ANSWER1
        RETURN
```

LISTING 44 – PROGRAM TK3TUT44

```
MAIN    BTFSS INTCON,2
        GOTO MAIN
        BCF INTCON,2
        MOVLW B'10000001'
        IORWF CHAN0,W
        MOVWF ADCON0
        CALL DELAYB
        BSF ADCON0,GO
        CALL GETADC
        MOVLW B'10000000'
        CALL LCDLIN
        CALL SHOWVAL
        MOVLW B'10000001'
        IORWF CHAN1,W
        MOVWF ADCON0
        CALL DELAYB
        BSF ADCON0,GO
        CALL GETADC
        MOVLW B'11000000'
        CALL LCDLIN
        CALL SHOWVAL
        GOTO MAIN

GETADC  BTFSC ADCON0,GO
        GOTO GETADC
        MOVF ADRESH,W
        MOVWF ADCMSB
        BANK1
        MOVF ADRESL,W
        BANK0
        MOVWF ADCLSB
        RETURN
```

EXERCISE 35

35.1 Write a clock program in which only seconds are counted. Using the Division facility, convert the seconds to hours, minutes and seconds and output the values to the l.c.d. in decimal.

TUTORIAL 36

CONCEPT EXAMINED

ADC conversion routine for PIC16F87x family

This example is not capable of being run using the PIC16F84 which does not have analogue-to-digital (ADC) capabilities.

Analogue to digital conversion (ADC) is a facility provided with the PIC16F87x family. A full discussion of ADC conversion is too lengthy for this tutorial and readers are referred to Microchip's data sheets for this family.

However, in a nutshell, devices in the PIC16F87x family each have several pins which can be set for use as ADC inputs, the allocation depending on the PIC type. The PIC16F877 has eight pins, RA0 to RA5, plus RE0 to RE2.

There is a choice of which pins are used in ADC mode and the selection is made via register ADCON1 (data sheet section 11-2). ADC channels are accessed via register ADCON0, which also allows the conversion clock rate to be adjusted.

PIC16F87x devices have a 10-bit ADC, held in two bytes, ADRESH (MSB) and ADRESL (LSB). Bit 7 (ADFM) of ADCON1 controls whether the value is justified to the left or right of those registers.

An example program is given in TK3TUT44.ASM, part of which is shown in Listing 44. In the full listing, and from within BANK1, ADCON1 is set for RA0, RA1 and RA3 as ADC inputs, with ADFM set for righthand justification (ADC bits 0 to 7 in LSB, and bits 8 and 9 as bits 0 and 1 of the MSB). It will be seen that TRISA is then set for RA0, RA1 and RA3 as inputs, for ADC use, but also RA4 as an input for normal digital use, and RA2, RA5 as digital output.

The channel selection codes have been set into named registers, CHAN0 to CHAN7, and during the program the desired ADC channel is set via ADCON0 using these register values.

In Listing 44, sampling is done each time the timer rolls over, first for CHAN0 (RA0). Command MOVLW B'10000001' configures bits 7 and 6 for a sampling oscillator rate of Fosc/32, and sets bit 0 to turn on the ADC facility. The value is then ORed with the channel code (IORWF CHAN0,W), and the combined value MOVed into ADCON0.

Following a brief delay, conversion is initiated (BSF ADCON0,GO) and the ADC retrieval routine called (GETADC). Once ADCON0,GO bit is found to be low, the converted value is retrieved and output in decimal to the l.c.d. (via SHOWVAL, see full listing).

Channel 1 is then activated and processed in a similar way, after which the process is repeated from label MAIN.

EXERCISE 36

36.1 Experiment with the ADC routine, accessing other available PIC16F877 pins. A variable voltage can be applied to any ADC pin via TK3's preset VR3. Only apply a voltage within the range 0V to 5V d.c. if you are using an external source.

36.2 Study Microchip's data sheet (DS30292C) for more detail about using the PIC16F877's ADC facility.

TUTORIAL 37

CONCEPTS EXAMINED

CBLOCK equates defining function

I^{2}C interfacing to serial EEPROM chips (e.g. 24LC256) from PIC16F87x family

CBLOCK COMMAND STRUCTURE

Up to now you have been equating values and addresses using the FILE1 EQU H'20' type of command. This has allowed you to give names to Special File Register address values, and to the normal data registers. There is another method through which data register names can be given addresses, and that is through Microchip's CBLOCK option, as can be used when writing ASM files in their MPASM dialect.

TK3 also recognises the CBLOCK function (since version V1.4) and it represents a very easy way to name addresses which are not critical in terms of their actual locations. This function applies to any of the data registers that lie typically between H'20' and the maximum address available for a specific PIC type. It is not suitable for use with Special Function Registers, whose locations are fixed, or bit values, whose values are also fixed.

CBLOCK is relevant to the discussion now as programs TK3TUT45 and TK3TUT46 will be combined in program

TK3TUT47, along with some other files to create a larger overall function, that of a data logger with facilities for outputting to a PC. Such combining of several program sources is much aided by the CBLOCK function.

Listing 45 shows an example of using CBLOCK to allocate some of the registers used in TK3TUT45.

LISTING 45 – PROGRAM TK3TUT45

```
        CBLOCK
; automatically allocates equated addresses to the following registers:

MEMHI
MEMLO
WADDRL
WADDRH
RADDRH
RADDRL
        ENDC    ; end of allocation block
```

When the assembler comes across the CBLOCK statement it allocates ascending address numbers to each of the names in the list that follows. The program initially sets the starting value in relation to any equated data registers already declared.

For example, the program might have declared the starting value to be H'20'. In Listing 45, MEMHI would then be equated by the assembler to be register H'20', and MEMLO then equated as H'21', and so on. The function is terminated when the assembler finds the ENDC command at the end of the list. The CBLOCK list in TK3TUT45 is much longer than shown in Listing 45.

It is also possible to state the CBLOCK starting address through the CBLOCK command itself. This can take the form of:

CBLOCK H'35'

in which case H'35' will become the starting address.

Or the form could be:

STARTVAL EQU H'22'
CBLOCK STARTVAL

This would allocate H'22' as the start address holding it in STARTVAL. Any name can be used instead of STARTVAL, and any value given to it, in any of the previous discussed forms.

The great advantage of CBLOCK is that Include files can have a series of named registers included within their own CBLOCK function, and for those to be automatically equated appropriately within the main program. Examples of this will be seen in the full listing of TK3TUT47 and the programs it calls as Include files.

In TK3TUT45, because CBLOCK has not been given a starting value, the value is automatically allocated as *TK3*'s default of H'20', and all of its named registers are numbered accordingly. The last value used is then stored by *TK3*, for further use if another file having a CBLOCK list is Included.

Each time a called Include file is pulled in by *TK3*'s assembler, any CBLOCK names within that file are numbered consecutively from the last value. Care has to be taken, of course, that multiple CBLOCK names do not cause the maximum register limit of the PIC to be exceeded, and that identical names are not found in two or more Included files.

The latter situation will be reported as an error condition by *TK3*. The former, though, can only be assessed by inspection of the .LST file generated during assembly.

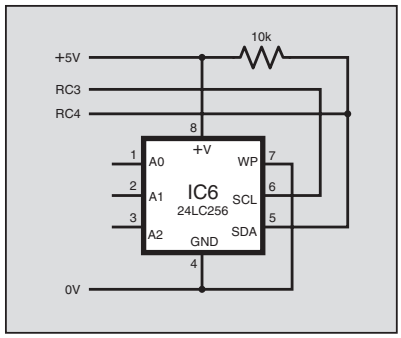


Fig. 10 Basic connections between a 24LC256 and a PIC16F877x.

WRITING TO SERIAL EEPROM

Microchip manufacture a variety of non-volatile serial EEPROM devices which can be readily interfaced to the PIC16F87x family, and possibly other PIC families too, although the author has not done so. Serial EEPROM devices allow large blocks of data to be stored indefinitely and retrieved at a later date, even after power has been switched off and back on again. The facility is of enormous value in applications such as data logging, for example.

The serial devices used by the author in a number of designs are from the 24LCxxx family, in particular the 24LC256 which can store 256K bits of data, i.e. 32768 (32K) bytes of data. The "xxx" in the code number indicates the thousands of bits that the device can store. Microsoft datasheet DS21203F covers the 24LC256.

The basic interfacing to a PIC16F87x is shown in Fig.10.

The interface protocol used by these chips is known as I^2C, pronounced I-squared-C (although you may also hear I-TWO-C used as well). The meaning is Inter-Integrated Circuit, i.e. one i.c. talking to another. This protocol requires just two signal wires to be connected between the PIC and the EEPROM chip, one for data, transmitted in serial format, and the other for a clock signal.

The devices are manufactured so that up to eight can be used in a block without additional circuitry. The choice of which device is accessed is determined by a 3-bit binary address code that is transmitted to the device as part of the data transfer process.

Allocating a device to a particular address is done by connections to the three address pins, A0 to A2. The address is set by the binary code on those pins (from decimal 0 to 7, binary 000 to 111). Internally, the pins are biased to 0V and so to set a pin for 0V you simply leave it unconnected. To set it high the pin is wired to the +5V power line.

The program used by the author since his *8-Channel Data Logger* of Aug/Sept '99 is a slightly modified variant of that supplied by Microchip on their Technical Library CD-ROM disk 2, **download\appnote\category\eeproms\00567.ZIP**. The routine for writing to the serial EEPROM chip is in file **2WDPOLL.ASM**, and reading back from it is in file **2WSEQR.ASM**. The author has slightly modified the programs, particularly to allow the Write facility to be used with a variety of PIC clock rates.

The code shown in Listing 45A is that which calls the Write to EEPROM routine in Microchip's program, allowing the MSB and LSB of a 16-bit data value (two bytes) to be stored at consecutive addresses in the selected EEPROM chip. The full listing is in program TK3TUT45.ASM.

Before calling the entry to the routine at SAVESAMPLE your own program must:

- place the MSB of the data to be stored into MEMHI
- place the LSB of the data to be stored into MEMLO
- set WADDRH with the MSB of the EEPROM chip address required
- set WADDRL with the LSB of the EEPROM chip address required
- load W with the value corresponding to the EEPROM chip to be accessed (between 0 and 7)
- then use the command CALL SAVESAMPLE

On entry to SAVESAMPLE, the value in W (the required EEPROM chip number) is placed into ECHAN. Then at label WRMSB the value in MEMHI is pulled into W, and Microchip's WRBYTE routine is called. The EEPROM address is then incremented and the value in MEMLO is pulled into W and again WRBYTE is called. The EEPROM address is then decremented back to the previous value and an exit made from the routine.

It is important that your program should avoid entry address values which would cause LSB rollover when command INCF WADDRL,F is made.

There is an additional requirement to be met before this routine can be used and it is set at the head of your program. The requirement is to set a delay value for use by Microchip's program to suit the PIC's clock rate.

LISTING 45A

```
SAVESAMPLE
; Entry point for storing double byte data to serial EEPROM
                MOVWF ECHAN
        ; set EEPROM chip number held in W into ECHAN
WRMSB   MOVF MEMHI,W        ; get data MSB
        CALL WRBYTE         ; store it to EEPROM
        INCF WADDRL,F       ; inc EEPROM address
WRLSB   MOVF MEMLO,W        ; get data LSB
        call WRBYTE         ; store it to EEPROM
        DECF WADDRL,F       ; set EEPROM address
                              back to entry value
        RETURN
```

Microchip's original program used several series of NOP commands to provide several delays in the Write routine, varying between three and five NOPs, depending on the sub-section of the routine. Five NOPs is taken as the delay unit for Microchip's program, which was written for use at 4MHz, and each NOP provides a delay of 1μs at 4MHz.

To simplify the program's use with different clock rates, the author has replaced the several series of NOPs with commands that call a given number of NOPs through a definition at the head of the program:

#DEFINE SERIALDELAY CALL CYCLESx

where "x" in CYCLESx is replaced by a number between 4 and 25. A value of 4 gives the minimum delay possible. A value of 5 is taken as suitable to clock rates of 4MHz and below. A value of 25 would suit the program to a clock rate of 20MHz.

A series of consecutive labels is included in the author's program that accesses Microchip's routines, e.g.:

```
CYCLES25    NOP
CYCLES24    NOP
CYCLES23    NOP
etc., through to:
CYCLES5     NOP
CYCLES4     RETURN
```

Thus if the definition is

#DEFINE SERIALDELAY CALL CYCLES25

the program is assembled such that command SERIALDELAY is replaced by CALL CYCLES25. When run, the program thus calls label CYCLES25 from the various points at which the command SERIALDELAY is given. At the called label, each NOP in the list down to CYCLES5 is then actioned, followed by an exit at CYCLES4.

The number of NOPs introduced in the sequence is related to the rate at which the PIC performs each command with respect to the clock rate. Recall that PICs effectively divide the clock rate by four. Thus a clock rate of 4MHz results in an effective cycle rate of 1MHz.

PIC commands take either one or two cycles to complete (as shown in Table 1, part 1). A CALL takes two cycles to perform, NOP takes one, and RETURN takes two.

If CALL CYCLES4 is performed, the delay is four cycles, CALL = 2, RETURN = 2. CALL CYCLES5 takes five cycles, four for CALL/RETURN plus one for the NOP, and so on.

The object is to introduce a delay of a minimum of 1μs between various actions within Microchip's program, although it does not matter if the delay is longer. To calculate the CALL CYCLESx value use the following method:

- Required clock rate = 20MHz
- Assume Microchip's 4MHz rate requires 5 cycles
- Your rate therefore = 5 × (20 / 4) = 5 × 5 = 25
- Thus you would need to use CALL CYCLES25

READING FROM SERIAL EEPROM

The technique for reading double-byte data back from a serial EEPROM chip is very straightforward. It just involves the following:

- set RADDRH with the MSB of the EEPROM chip address required
- set RADDRL with the LSB of the EEPROM chip address required
- then use the command CALL READ

Microchip's read data routine is then entered and the data recalled from the given EEPROM address is put as the MSB into MEMHI, and as LSB into MEMLO. You can then use these values as you wish.

TUTORIAL 38

CONCEPTS EXAMINED

Outputting serial data from PIC16F87x to PC at selected Baud rate

The PIC16F87x family allow data to be output serially from PORTC RC5 and RC6 as a serial stream conforming to the RS-232 protocol and at a chosen Baud rate. The Baud rate required is normally set via a routine called during a program's initialisation procedure. That Baud rate then remains set for the rest of the program.

The Baud rate can be set for one of many options offered through Microchip's datasheet DS30292A for the PIC16F87x family. The selection can be made as discussed in the datasheet's section 10, with particular reference to Tables 10.3, 10.4 and 10.5.

The datasheet also discusses many aspects of serial output and input and you are referred to it for detailed information. In this Tutorial we simply show a practical example of outputting data, in relation to the mode the author has frequently used.

SETTING BAUD RATE

The routine for setting the Baud rate is shown in Listing 46. On entry at label SETBAUD, a value (20 in this case) is loaded into W (within BANK1) and placed into the PIC's SPBRG register. The value is in respect of a 9600 Baud rate when using a clock rate of 3·2768MHz.

Regrettably, Microchip do not give the SPBRG value for a 3·2768MHz crystal, but they do give two formulae for calculating the value in relation to PIC clock and asynchronous Baud rates. The first formula is used when register TXSTA bit BRGH = 1 (high speed). The second is when BRGH = 0 (slow speed).

Baud = Fosc / (16 × (X + 1) for BRGH = 1
Baud = Fosc / (64 × (X + 1) for BRGH = 0

where *Fosc* is the PIC's clock rate and *X* is the SPBRG value.

Microchip state that it may be advantageous to use BRGH = 1 even for slower Baud clocks, because its equation can reduce Baud rate error in some cases.

Transposing the formula when BRGH = 1, we get:

X = (Fosc / (Baud × 16)) – 1

Thus for Fosc = 3.2768MHz and Baud = 9600 we get:

X = (3276800 / (9600 × 64)) – 1
= (3276800 / 153600) – 1
= 20.333

Only integer (whole number) values can be used and so the calculated SPBRG value is 20. Putting the value of 20 into *X* of the formula, we get an actual Baud rate of:

3276800 / (16 × (20 + 1))
= 3276800 / 336 = 9752·38

This represents an error of (9752.38 – 9600) / 9600 = 0.015873%

An SPBRG value of 21 produces an error of 0·030303%, and 19 would give 1·06667%. An SPBRG value of 20 is thus a reasonable one to use (a certain amount of latitude is normal and the PC's serial input routines should tolerate minor slippage).

Asynchronous mode is needed for the example in Listing 46. The datasheet shows in its Fig.10-1 that the TXSTA register needs bit 4 (SYNC) cleared for this mode. Bit 2 is BRGH and, as indicated above, is set to 1.

Bit 7 (CSRC) is "Don't Care" for asynchronous mode.

Bit 6 (TX9) selects between 9-bit and

LISTING 46 – PROGRAM TK3TUT46

```
; set serial output Baud rate
; as shown is set for 9600 Baud with a 3.2768MHz clock rate

SETBAUD
        BANK1
        MOVLW 20              ; BRG for 9600 Baud from
                                3·2768MHz, brgh=1
        MOVWF SPBRG
        MOVLW B'00000100'     ; set sync=0, brgh=1 + ninth bit
                                not set
        MOVWF TXSTA
        BCF PIE1,TXIE         ; clear interrupt bit
        BANK0
        MOVLW B'10000000'     ; set SPEN bit (7) of RCSTA reg
        MOVWF RCSTA
        BANK1
        BSF TXSTA,TXEN        ; enable transmission (bit TXEN)
        BANK0
        RETURN
```

LISTING 46A

```
; routine that causes the actual output of a serial data byte
SERIALSEND
        BTFSS PIR1,TXIF       ; wait for TXIF bit 4 to go high
        GOTO SERIALSEND       ; (showing TXREG empty)
        MOVWF TXREG           ; put val (held in W) in TXREG
                                ready for sending
        RETURN
```

8-bit transmission (1 and 0 respectively). 8-bit is required in the example.

Bit 5 (TXEN) enables/disables transmission (1 and 0 respectively).

Bit 3 is unimplemented in the PIC16F87x and so ignored.

Bit 1 (TRMT) indicates whether the transmit shift register is empty (1) or full (0).

Bit 0 is the 9th bit of transmit data (and can also be used as a parity bit). It is not needed in the example, and so is set to 0.

Consequently, following the setting of SPBRG, TXSTA is loaded with B'00000100'. Next the Transmit Interrupt Enable bit (TXIE) of register PIE1 is cleared to disable the interrupt, and BANK0 is then selected again.

Next the SPEN bit (7) of register RCSTA is set to enable pins RC6 and RC7 for serial port mode. After which register TXSTA bit TXEN is set within BANK1 to enable transmission. Setting back to BANK0 follows, and the Baud rate setting routine is exited.

In other applications, the only values that concern you are those for Baud rate (SPBRG) and the initial setting of TXSTA. The other statements can be repeated parrot-fashion.

The Microchip INClude file PIC16F877.INC holds the values for the named registers and bits, and allocates them when imported to the program.

SERIAL OUTPUT

Listing 46A shows the routine that causes the actual output of a serial data byte. The SERIALSEND routine is entered with W holding the value to be transmitted. A wait is then made until bit TXIF of the PIR1 register goes high, indicating that register TXREG is ready to be loaded with the byte to be sent.

As soon as TXREG is loaded with the value in W, the transmission is then automatic, and out of your hands! Consequently, the routine is then exited and your program can get on with what else it wants to do, typically at this time to get the next value to be transmitted until all values have been sent.

PIR1 TXIF automatically goes low when TXREG is loaded, staying low until transmission of that byte has been completed.

So that's all there is to transmitting data from a PIC as a stream of serial data at a known Baud rate. The destination is likely to be a PC, but it could be sent to any device capable of reading serial data.

Describing how a PC (or other device) receives the data is beyond the scope this series, but *EPE* has published several examples of PC programs that do so. The *BioPIC Heart Monitor* of June '02, shows an example written in QBasic. The *Earth Resistivity Logger* of Apr/May '03 shows an example written in Visual Basic (VB6) using Robert Penfold's INPOUT32.DLL as the active software interface.

We shall also be publishing shortly Joe Farr's serial input/output OCX facility for use with Visual Basic. Joe has written it specially for *EPE* and it will be a boon to users of VB. Details will be announced in due course.

It should be noted that whilst the author has found QBasic can input serial data directly from a PIC, VB6 behaves more reliably when a dedicated RS-232 chip is used between the PIC and the PC. An example of using a MAX232 device in this role is shown in the *Earth Resistivity Logger* Part 1, April '03.

TUTORIAL 39

CONCEPT EXAMINED

Practical example of recording analogue data to serial EEPROM and subsequent outputting as RS-232 serial data.

This Supplement does not have enough space to include Tutorial 39 and it has been placed on the PIC Resources CD-ROM as EPE PIC Tutorial V2 Extra.

Nor has there been space to include the table of Reset Conditions for the PIC16F84. This may be found on Microchip's datasheet 30430C. Datasheets 30292C and 40300 are for the PIC16F87x and PIC16F62x families respectively. Browse **www.microchip.com**.

TUTORIAL 40

CONCEPTS EXAMINED

Programming
PICs vs. Hardware
Summing-up

PROGRAMMING

To the uninitiated, it may seem that a software programmer simply sits down and writes all the commands in a single operation. If only it were that simple! Before a single line of code is written, there is a great deal of thought involved about the overall objective and how each step on the way to achieving it might be performed. Part of this consideration relates not only to the logic of the software routines, but also to the control requirements of external interfaces.

There are two schools of thought about the planning. The first considers that the use of flow charts is an essential requirement. The other doesn't! The advantage of using a flow chart is that it shows the questions and answers of each stage of the program in a diagrammatic form. Theory says that this chart then enables the code to be written to meet each of the requirements illustrated.

The use of a flow chart certainly helps in concentrating immediate thought processes, and in recapturing concepts in the future, but it cannot display the command by command reasoning of each line of code. Only the code itself shows that, unless you translate each line of code into lengthy textual comments, in which case there is the danger of getting bogged down with words.

Additionally, there is always the possibility that some logical consideration has been omitted from the flow chart and which only comes to light once you try to run the program, requiring the chart to be redrawn as well as the software having to be rewritten. The author finds that the detailed thinking about the program structure already builds up as a mental flow chart which does not require to be set down on paper.

It is acknowledged that in a commercial situation it would be mandatory for the program structure to be well documented with flow charts – the program might eventually need to be changed by someone other than the original program writer. In that case, the flow chart would give a more immediate insight into the original programmer's thought processes.

Although the author does not use flow charts when programming, let us not deter you from drawing them if you prefer to do so. You may well find that they help you to grasp what you are doing more readily than just relying on your mental "visualisation" processes.

To discuss flow charts more fully is beyond the scope of these Tutorials, but you will find examples of them on Microchip's Technical Library CD-ROM. It has to be said, though, that even in that publication, which is full of PIC datasheets and program listings, flow charts are not widely used.

STAGE-BY-STAGE

Whether or not you use flow charts, you should never attempt to write the entire program from beginning to end in one operation. That way can lead to extensive problems when you try to debug the program having found that it does not do what you expected. Take each routine stage-by-stage. Get one small section of code working before you move onto the next. Then get that next small section working before you try to join it to the previous part. "Be methodical" is the key command when programming.

In many ways, the manner in which these Tutorials have been presented has been along those lines. We have tried to show you individual structures which first stand on their own, and then are extended or joined to others to achieve a larger operational system. Taking TK3TUT2 as the effective starting point, that program used only 16 command lines. Gradually we developed other programs as stand-alone routines. Then things began to take a broader structure as concepts were integrated into a more sophisticated whole. With TK3TUT33, 232 commands were sent to the PIC.

As you get further into PIC programming, you may decide that you would like to write code in conjunction with a simulation program. These help you to debug code on your PC before downloading it to the PIC. They will not replace the thought processes needed when writing code, but they will let you find many (but not all) of the errors more quickly.

However, the author finds it very easy to check program operation when the code is in the PIC and the PIC is connected to its various interfaces. Had the PIC16F84 not been an EEPROM device, then this would not be an acceptable technique, but it *is* rapidly reprogrammable and so is usable as a live test-bed.

One further point, when writing a program the author finds it useful to supplement its software file name with a suffix number, increasing the number at each save of a major addition or change to the previous code written. This allows an earlier version to be recalled should the need arise. Thus you would number as, for example, PICIT01.ASM, PICIT02.ASM, PICIT03.ASM, etc.

PICS vs. HARDWARE

As enormously beneficial as the use of a microcontroller can be, there is the likelihood that it may be regarded by the inexperienced as the ultimate answer to all electronic circuit design. This is most definitely not the case. All that a microcontroller will do is assist in using software commands to replace a fair number of operations for which many electronic components would otherwise be needed. It cannot substitute for all electronic requirements.

There are also situations in which a microcontroller *can* be used but it is not necessarily desirable that it should. What you will discover as you get further into programming, is that the act of programming a PIC to replace a given number of logic chips takes far longer than if you were to design a circuit that performed the same function but only used such chips.

Unless you actually *want* to get a PIC to do something because it *can* and you see it as a challenge, always ask yourself if the additional development time is worth it in order to save a chip or two.

SUMMING-UP

When writing software, you will find much frustration through the inability to immediately see the bug in a program routine. Eventually, though, you will spot it and the relief and exhilaration of at last getting that part to work is enormous. In that frame of mind, you will move onto writing the next sub-routine with the utmost confidence and anticipation of not making a mistake on *this* one.

Would that it were so! You can, and you will, make mistakes. But the ultimate satisfaction of a complete working design makes it all worthwhile.

If you can't take occasional bouts of desperation, isolation from friends and family, followed by periods of ecstasy and feelings of well-being towards all humanity, leave programming alone. The author, though, has become a "programming-addict" and thrives on the challenges, come what may!

Finally, remember that Murphy's Law has its most powerful influence when programming is involved. If the microcontroller or other computer *can* misunderstand what you *mean* by your commands, it *will*. It is up to you to see the way in which each and every one of your commands will *actually* be interpreted. You are the intelligent one, the computer simply obeys your commands!

APPENDIX A: BUGGED TEASER

Now to give your understanding of PIC programming (and your logical thinking) a bit of a test!

Load TK3TUT48.ASM into your text editor. Add an appropriate initialisation routine at the beginning, and add a suitable set of l.c.d. operating routines as illustrated earlier. Save the code as two slightly different file names, then work on the second file.

TK3TUT48 has a number of bugs deliberately included – your task is to debug the program and get it working as a frequency counter! Some of the deliberate errors will be reported by the Assembler following assembly. These are "literal" errors which anyone might make while creating a PIC program – simple slips in thinking. The others, though, are "logical" errors – much more significant errors in a programmer's analysis of a situation and its interpretation, but still errors anyone could make.

First of all, get the l.c.d. to display the opening message correctly (and without the program "crashing"). Then, with the aid of a signal generator (0V/5V output logic level) set to about 10kHz (you must decide which PIC input pin to use), solve the remaining logical problems. You'll probably curse the author a few times before you solve it all, but keep at it!

Having solved it (and felt the satisfaction of success!), think about how switches and other routines could extend the counter's range.

TABLE 8 MPASM SHORTHAND COMMANDS The following MPASM shorthand commands are recognised by *TK3*.

Command	*Equiv. coding*	*Meaning*
ADDCF f,d	BTFSC STATUS,C INCF f,d	Add Carry to File
ADDDCF f,d	BTFSC STATUS,DC INCF f,d	Add Digit Carry to File
B k	GOTO k	Branch to
BC k	BTFSC STATUS,C GOTO k	Branch on Carry to k
BDC k	BTFSC STATUS,DC GOTO k	Branch on Digit Carry
BNC k	BTFSS STATUS,C GOTO k	Branch on No Carry
BNZ k	BTFSS STATUS,Z GOTO k	Branch on Not Zero
BNDC k	BTFSS STATUS,DC GOTO k	Branch on No Digit Carry
BZ k	BTFSC STATUS,Z GOTO k	Branch on Zero
CLRC	BCF STATUS,C	Clear Carry
CLRDC	BCF STATUS,DC	Clear Digit Carry
CLRZ	BCF STATUS,Z	Clear Zero
MOVFW,f	MOVF f,W	Move File to W
NEGF f,d	COMF f,F INCF f,d	Negate file
SETC	BSF STATUS,C	Set Carry
SETDC	BSF STATUS,DC	Set Digit Carry
SET Z	BSF STATUS,Z	Set Zero
SKPC	BTFSS STATUS,C	Skip on Carry
SKPDC	BTFSS STATUS,DC	Skip on Digit Carry
SKPZ	BTFSS STATUS,Z	Skip on Zero
SKPNC	BTFSC STATUS,C	Skip on No Carry
SKPNDC	BTFSC STATUS,DC	Skip on No Digit Carry
SKPNZ	BTFSC STATUS,Z	Skip on Not Zero
SUBCF f,d	BTFSC STATUS,C DECF f,d	Subtract Carry from File
SUBDCF f,d	BTFSC STATUS,DC DECF f,d	Subtract Digit Carry from File
TSTF f	MOVF f,F	Test File

Where: d = destination (0 or 1 – W or F) f = file k = literal value

APPENDIX B: USEFUL PIC INFORMATION

Microchip web site:
http://www.microchip.com

EPE web site:
http://www.epemag.wimborne.co.uk

EPE PIC-project source code files:
ftp://ftp.epemag.wimborne.co.uk/pub/PICS

FURTHER READING

The following texts are on the PIC Resources CD-ROM:

Asynchronous Serial Communications (RS-232), John Waller, unpublished

EPE StyloPIC (precision tuning of musical notes), John Becker, July '02

How to Use Graphics L.C.D.s with PICs (detailed control information for PIC16F877), John Becker, Feb '01 (Supplement)

How to Use Intelligent L.C.D.s, Julyan Ilett, Feb/Mar '97

PIC Macros and Computed GOTOs, Malcolm Wiles, Jan '03.

PIC Magick Musick (illustrates use of 40kHz ultrasonic transducers), John Becker, Jan '02

PIC to Printer Interfacing (Epson dot matrix printers), John Becker July '01

PIC Toolkit Mk3, John Becker (PIC programmer p.c.b./circuit for TK3), Oct '01

PIC Toolkit TK3 for Windows (software details), John Becker, Nov '01

PIC16F87x Additional Memory (how to use it), John Becker, June '01

PIC16F87x Microcontrollers (review), John Becker, April '99

PIC16F87x Mini Tutorial, John Becker, Oct '99

Programming PIC Interrupts, Malcolm Wiles, Mar/Apr '02

Using I^2C Facilities in the PIC16F877, John Waller, unpublished

Using PICs with Keypads (16-key "data" keypads), John Becker, Jan '01

Using Square Roots with PICs, Peter Hemsley, Aug '02

Using TK3 with Windows XP and 2000, Mark Jones, Oct '02

Using the PIC's PCLATH Command, John Waller, July '02

□