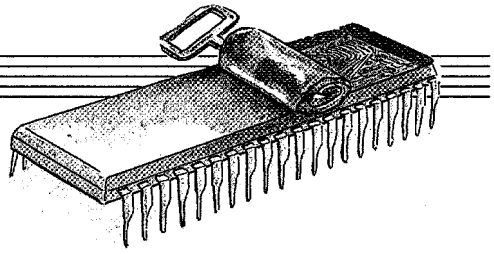
Special Feature



PIC16F87x MINI TUTORIAL

JOHN BECKER

Taking a programmer's look at some of the PIC16F87x family's facilities.

AT LONG last Microchip's new PIC16F87x family of microcontrollers is actually available through component distributors and retailers. Forecast in late 1998, production release commenced in July 1999.

The author has had engineering samples for some months and has examined them with considerable interest. A general look at the family was taken in the *PIC16F87x Review* (Apr '99). Now we can encourage you to explore the possibilities of these very welcome PICs, knowing that you can obtain the production devices.

The aim of this Mini Tutorial is to share with you some of the findings made by the author when designing the *8-Channel Analogue Data Logger*, published in Aug/Sept '99. The findings relate to investigations using the PIC16F877 but apply in principle to the PIC16F873, '874, and '876 devices as well.

Readers should also study the 200-page data sheet that covers the devices, Microchip code DS30292A (see later).

The following *EPE* subject material is also referred to in this Mini Tutorial:

- *PIC Tutorial* (Mar-May '98)
- *PICtutor* (CD-ROM version of the *PIC Tutorial*)
- *PIC Toolkit Mk1* (Jul '98)
- *PIC Toolkit Mk2* (May-Jun '99)
- *Virtual Scope* (Jan-Feb '98)

PROTOCOL

In the course of this article, reference is made to figures from Microchip data sheet DS30292A and the reference style used is, for example, DS-FIG.11-2 – meaning data sheet Figure 11-2.

Note also the use of the $ (dollar) sign when referring to hexadecimal numbers, and the % (per cent) sign for binary numbers, e.g. $9F and %11001111.

Where PIC source code examples are given, the dialect used is TASM (the differences between TASM and MPASM were discussed in *Toolkit Mk1*, *PIC Tutorial* and *PICtutor*).

All registers and bits referred to by name should have those names and their values "equated" at the head of any program that uses them.

PAGES 0 TO 3

The subject of Pages in relation to PIC16x84 devices was well covered in the *PIC Tutorial* and *PICtutor* texts, and a lot of readers expressed gratitude for the explanation.

Whereas the 'x84 had only PAGE0 and PAGE1 to be set in relation to the use of some Special Registers, the 'F87x devices have four Pages (Microchip actually calls them Banks – but we'll stick with Pages).

In the STATUS register, two bits (instead of one) control the Page selection, bits 6 and 5 respectively. Consequently, the shorthand technique used with the 'x84 of defining the instruction PAGE0 or PAGE1 to mean clearing or setting of STATUS bit 5 accordingly, is less easy to use when two STATUS bits are affected. Therefore, in common with the Microchip data sheet, the STATUS bits will be referred to by their allocated names, of RP1 (bit 6) and RP0 (bit 5).

Their settings are as follows:

PAGE	RP1	RP0
PAGE0	0	0
PAGE1	0	1
PAGE2	1	0
PAGE3	1	1

You will make life easier for yourself if you "equate" these two bits at the head of your programs:

```
RP0:  .EQU 5
RP1:  .EQU 6
```

The Register File Maps in DS-FIG.2-3 and DS-FIG.2-4 show which registers are in which Pages (Banks).

PCFG3: PCFG0	AN7[1] RE2	AN6[1] RE1	AN5[1] RE0	AN4 RA5	AN3 RA3	AN2 RA2	AN1 RA1	AN0 RA0	VREF+	VREF-	CHAN / REFS
0000	A	A	A	A	A	A	A	A	VDD	VSS	8/0
0001	A	A	A	A	VREF+	A	A	A	RA3	VSS	7/1
0010	D	D	D	A	A	A	A	A	VDD	VSS	5/0
0011	D	D	D	A	VREF+	A	A	A	RA3	VSS	4/1
0100	D	D	D	D	A	D	A	A	VDD	VSS	3/0
0101	D	D	D	D	VREF+	D	A	A	RA3	VSS	2/1
011x	D	D	D	D	D	D	D	D	VDD	VSS	0/0
1000	A	A	A	A	VREF+	VREF-	A	A	RA3	RA2	6/2
1001	D	D	A	A	A	A	A	A	VDD	VSS	6/0
1010	D	D	A	A	VREF+	A	A	A	RA3	VSS	5/1
1011	D	D	A	A	VREF+	VREF-	A	A	RA3	RA2	4/2
1100	D	D	D	A	VREF+	VREF-	A	A	RA3	RA2	3/2
1101	D	D	D	D	VREF+	VREF-	A	A	RA3	RA2	2/2
1110	D	D	D	D	D	D	D	A	VDD	VSS	1/0
1111	D	D	D	D	VREF+	VREF-	D	A	RA3	RA2	1/2

U-0	U-0	R/W-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM				PCFG3	PCFG2	PCFG1	PCFG0
bit7							bit0

Note 1: These channels are not available on the 28-pin devices.

Fig.1. Selectable functions of the ADCON1 register.

PIC PORTA AND PORTE

PORTA is a 6-bit bi-directional I/O (input/output) port whose bits RA0 to RA3 and RA5 can alternatively be used for analogue input.

PORTE is a 3-bit bi-directional I/O port whose bits RE0 to RE2 can alternatively be used for analogue input.

The first matter of interest that came to light when experimenting with the PIC16F877 was that its PORTA did not behave as expected when used as a normal I/O data port.

The PIC16F877 was originally loaded with the TUTTEST program that allows users of the *PIC Tutorial* and *PICtutor* development boards (which are for use with PIC16x84 devices) to initially test their system. PORTB behaved as expected, but not so PORTA.

Examination of the data sheet revealed that PORTA's default mode is not for digital data I/O, but for analogue data input.

The register which controls PORTA's digital or analogue use is ADCON1, whose settings options are shown in Fig.1 (taken from data sheet DS-FIG.11-2). This same register controls PORTE as well.

Register ADCON1 is at address $9F, which is in PAGE1. Fig.1 shows that PORTA and PORTE are jointly set for analogue use when ADCON1's bits 0 to 3 are set to 0000 (the default condition on power-up). To jointly use PORTA and PORTE for digital purposes, the bits are set to 0111 (or 0110 since bit 0 can be 0 or 1 in this instance), i.e. ADCON1 has to be loaded with %xxxx111x – where x can be 0 or 1.

You will already be familiar with having to set the data direction (TRISx) registers depending on whether a port's bits are to be used for input or output. So, for both PORTA and PORTE to be fully set for digital output, TRISA and TRISE have to be cleared, which also has to be done in PAGE1.

Thus, an example of initialising your code to use PORTA and PORTE for digital output is:

```
BCF STATUS,RP1    ; clear
                  ; PAGE2/3 bit
BSF STATUS,RP0    ; set PAGE1
                  ; bit
MOVLW %00000111 ; all-digital
                  ; I/O code
MOVWF ADCON1      ; load it
                  ; into
                  ; ADCON1
CLRF TRISA        ; all PORTA
                  ; for output
CLRF TRISE        ; all PORTE
                  ; for output
BCF STATUS,RP0    ; reset to
                  ; PAGE0
```

Conversely, an example of initialising your code to use PORTA and PORTE for digital input is:

```
BCF STATUS,RP1    ; clear
                  ; PAGE2/3 bit
BSF STATUS,RP0    ; set PAGE1
                  ; bit
MOVLW %00000111 ; all-digital
                  ; I/O code
MOVWF ADCON1      ; load it into
                  ; ADCON1
MOVLW %00111111 ;
MOVWF TRISA       ; all PORTA
                  ; for input
MOVWF TRISE       ; all PORTE
                  ; for input
BCF STATUS,RP0    ; reset to
                  ; PAGE0
```

Note that TRISE ignores bits 3 to 7.

A-TO-D PREPARATION

From Fig.1, you will recognise that ADCON1 is set to xxxx0000 when PORTA and PORTE are to be used for analogue input. You also need to set the two associated TRIS registers for input (in the same way as setting them for digital input).

There is another option that can be chosen for ADCON1 as well – its bit 7 (ADFM) controls how the 10-bit digital value of the converted analogue signal is stored in the PIC's registers dedicated to this purpose, ADRESH ($1E – PAGE0) and ADRESL ($9E – PAGE1).

There is the choice of whether the result is justified to the left or right in these bytes. Take the example of a conversion result of 1023 (the maximum for a 10-bit conversion). The binary value of 1023 is, as two bytes, 00000011 11111111, stored as 00000011 in ADRESH and 11111111 stored in ADRESL.

This is the case when the ADFM bit (bit 7) is set to 1 (DS-FIG.11-8). However, if the ADFM bit is set to 0, the result is formatted as 11111111 11000000. This choice can be useful in some post-processing operations where it can save the use of multiplying (rotation commands).

As used in the *Data Logger*, a right-justified result was required, and so ADFM was set to 1. Since this design uses all of PORTA and PORTE for analogue input, the ADCON1 register was set with the commands:

```
BCF STATUS,RP1    ; clear
                  ; PAGE2/3 bit
BSF STATUS,RP0    ; set PAGE1
                  ; bit
MOVLW %10000000 ; all-analogue
                  ; input code
MOVWF ADCON1      ; load it into
                  ; ADCON1
MOVLW %00111111 ;
MOVWF TRISA       ; all PORTA
                  ; for input
MOVWF TRISE       ; all PORTE
                  ; for input
BCF STATUS,RP0    ; reset to
                  ; PAGE0
```

ADFM ANOMALY

When the first tests were made using the PIC16F877 samples provided by Microchip, the above commands were duly given in the *Data Logger* software that was being developed. To the author's consternation, the ADFM bit had no affect on the justification, which remained doggedly to the left when the result was displayed on an alphanumeric l.c.d.

It was eventually found that bit 5, not bit 7, was being used as the ADFM bit, and using that bit produced the required justification. What led the author to the conclusion that it should be bit 5 rather than bit 7 is that DS-Table 2-1 showed the power-on-reset condition for ADCON1 as being --0-0000, even though ADFM was shown as bit 7 on the left of the table.

Taking it up with Microchip, it transpired that the samples provided were Microchip's *early* engineering samples (version A) in which bit 5 was indeed the ADFM. They went on to say that production devices (version B) do have bit 7 as ADFM. This was proved when they sent version B samples to the author.

However, Microchip have asked us to advise anyone who has early engineering samples that this situation exists. Readers buying the chips from distributors or retailers should automatically be supplied with the production devices where bit 7 is the ADFM.

The fact remains, though, that the ADCON1 reset values given in DS-Table 2-1 of the 1998 DS30292A data sheet are incorrect and should read as 0---0000. Other tables in the same data sheet are similarly incorrect (3-1, 3-9, 3-12, 11-2).

A-TO-D CONVERSION

We have seen that ADCON1 is the register that sets PORTA and PORTE bits for analogue or digital use, and that ADRESH and ADRESL hold the 10-bit conversion value. A fourth register, ADCON0 ($1F – PAGE0) is responsible for the A-to-D control modes, as itemised in DS-FIG.11-1, but summarised as:

Bits 7-6	Conversion clock rate
Bits 5-3	Channel selection
Bit 2	Conversion status
Bit 1	Not used
Bit 0	A/D facility on/off

Prior to an A-to-D conversion being made, all bits have to be set appropriately. An example of the conversion sequence is shown in Listing 1, with entry point at ADCGET.

LISTING 1

```
ADCGET:  BCF STATUS,C     ; clear carry flag
         MOVF CHAN,W      ; get channel number
         MOVWF STORE      ; temporarily store it
         RLF STORE,F      ; multiply it by 8
         RLF STORE,F      ; to set it into correct bits
         RLF STORE,F      ; suited to ADCON0
         MOVLW %10000001  ; set clock Fosc/32 with A/D on
         IORWF STORE      ; OR this with stored value
         MOVWF ADCON0     ; move new value into ADCON0
         CLRF DELAY       ; set delay counter to 0
WAIT1:   DECFSZ DELAY,F   ; dec delay 256 times
         GOTO WAIT1       ;
SAMPLE:  BCF PIR1,ADIF    ; clear A/D interrupt flag bit
         BSF ADCON0,GO    ; start A-D conversion
WAIT2:   DECFSZ DELAY,F   ; again delay for 256 cycles
         GOTO WAIT2       ;
WAIT3:   BTFSC ADCON0,GO  ; is conversion complete?
         GOTO WAIT3       ; no
GETVAL:  MOVF ADRESH,W    ; get conversion high byte
         MOVWF MEMHI      ; store it in MEMHI
         BCF STATUS,RP1   ; clear Page 2/3
         BSF STATUS,RP0   ; set Page 1
         MOVF ADRESL,W    ; get conversion low byte
         BCF STATUS,RP0   ; reset Page 0
         MOVWF MEMLO      ; store low byte in MEMLO
         RETURN           ;
```

The channel number is held in CHAN and could be any value between 0 and 7. It has to be set into ADCON0 bits 5-3, consequently the first six commands are concerned with taking a copy of the CHAN value and rotating it to fill bits 5-3 of the temporary STORE. If the Channel number is 7, for example, CHAN holds %00000111, and the rotation into STORE produces %00111000.

The conversion clock rate is determined from DS-Table 11-1. The *Data Logger's* crystal clock runs at 3.2768MHz and the nearest value to this, as shown in DS-Table 11-1, is 5MHz and that a conversion factor of 32Tosc is required, which is set into ADCON0 with the bit 7-6 code of binary 10 (DS-FIG.11-1).

The converter is obviously required to be On, so bit 0 is set to 1.

The values for bits 7, 6 and 0 are thus set into W as %10000001, which is then ORed with the channel bits 5-3 held in the STORE (%00111000 in this example).

This composite value (%10111001) is then MOVed into ADCON0. Notice that bit 2, the Start bit (quaintly called the GO/DONE bit by Microchip!), is still at zero. The conversion does not start until this bit is set to 1. Also note that ADCON0 bit 1 has no function.

SOPHISTICATION

At this point it is possible to get rather sophisticated – but we're not going to!

The sophisticated way would be to use interrupt routines and precise timings, and a lengthy section in the data sheet discusses the options. They include calculation of the time taken to acquire the sample (which is relative to temperature, line impedance and capacitance values), and then to wait for an interrupt to occur on completion of the conversion.

Since the *Data Logger* is only required to take samples at the maximum batch rate of eight (all eight channels) every 0.5 seconds, time is not at a premium and so two simple delay loops are used, shown in Listing 1 as WAIT1 and WAIT2.

When WAIT1 has ended, the conversion is started by first clearing the A/D interrupt flag bit (bit 6 of register PIR1 – $8C, PAGE0), and then setting ADCON0 bit 2 (the GO bit).

The delay in WAIT2 then occurs, after which the GO bit is polled in WAIT3 until it is read as 0, signifying that the conversion is complete.

The values of the 2-byte conversion can now be read. The high byte (MSB) is held in ADRESH, which is in PAGE0 and so can be immediately read and stored in the user's own nominated location, in this case MEMHI. The low byte (LSB), however, is held in ADRESL, which is in PAGE1, which has to be set before the byte can be read. After which a reset to PAGE0 occurs and the byte is stored in MEMLO, so ending the conversion sequence.

As used in the *Data Logger*, the sequence is somewhat more padded than shown here since it is written to read a conversion for each of eight channels in turn, and then to store the 2-byte result in a chain of eight serial EEPROM memories, and to decimalise the result for showing on an alphanumeric l.c.d. screen.

Apart from the ADFM problem mentioned earlier, no surprises occurred when using the A/D facilities on all eight PIC16F877 channels, and the program worked first time. Which is more than can be said for storing the result in the serial EEPROM memories, as we shall reveal in a moment, plus the solution, of course!

ADC REFERENCE VOLTAGES

Referring again to Fig.1, you will see that pins RA2 and RA3 can be used as the pins on which external ADC reference voltages can be set, depending on the code set into ADCON1 bits 0 to 3. The columns 10 and 11 show how the reference voltage selection takes effect.

Although not discussed in the *Data Logger* text, the p.c.b. for that unit has been designed so that preset potentiometers can be inserted on it to set desired reference voltages on pins RA2 and RA3 for other applications. (The *Data Logger* itself does not offer this option through its software – readers must write their own software to suit their personal needs in this respect.)

The circuit diagram and p.c.b. assembly details for the insertions are shown in Fig.2 and Fig.3. Note the additional two link wires that are needed in order to complete the circuit between the wipers of the presets and their respective RA2/RA3 pins.

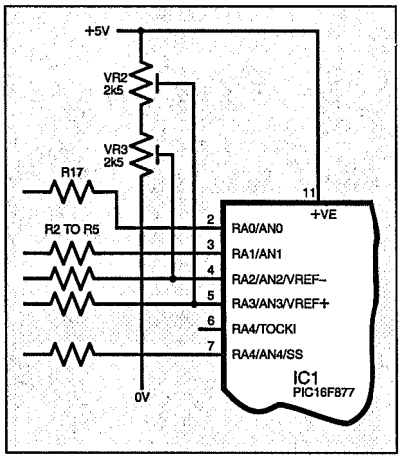
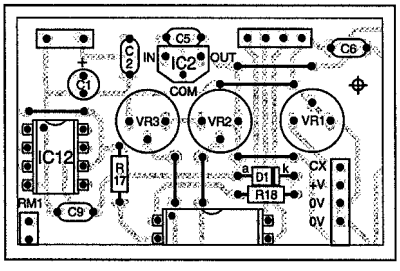


Fig.2. Reference voltage setting presets added to Data Logger circuit diagram Fig.1, and (below) Fig.3, their positioning on the Data Logger p.c.b.



Preset VR2 allows a reference voltage variation between approximately 2·5V and 5V on RA2, whilst VR3 allows variation between 2·5V and 0V on RA3. The use of a narrower range of reference voltage (which is normally 0V to 5V i.e. V_{ss} to V_{dd}) has the effect of providing amplification to the analogue input signal being converted.

By variously selecting different values for ADCON1 bits 0 to 3, different reference voltage combinations could be chosen for individual input channels.

It should be noted that when either RA2 or RA3 is selected as a reference voltage input, the selected pin cannot be used as an A-to-D input channel.

SERIAL MEMORY ACCESS

Once you get to know them, Microchip's 24LC256 serial EEPROM memories are really rather super little chips! It was their data sheet (DS21203D, 1998) that the author had difficulty with.

Attempts were made to write the routine which would store data at consecutive addresses in a 24LC256. Somehow, the logic of the data sheet's description and illustration eluded the author (it's not often he fails in such situations, but he failed this time!)

Running out of patience, he resorted to seeing if programs were available amongst Microchip's Applications Notes (on their CD-ROM and web-site).

There were several options available, of which the programs 2WDPOLL.ASM and 2WSEQR.ASM were selected as appearing to have the best options available (even though they were not written for PIC16F87x devices) and disk copies were made.

Being Microchip's own programs, they were naturally written in MPASM, whereas the author has a great preference for working in the TASM dialect. Consequently, the MPASM source codes were processed by the author's *PIC Toolkit Mk2* and converted to a TASM format.

Various modifications were then made to the programs to suit them to the *Data Logger's* needs, whereupon success was achieved! Data could now be written to the serial EEPROMs and, equally importantly, could be read back as well.

Those of you studying the *Data Logger* source code will find entry into the serial memory Write routine at label WRBYTE, and entry to the Read routine at READ. The routines are far too lengthy to list or describe here.

As a further plug for Microchip's Application Notes, they are well worth examining for all sorts of information and ideas, plus an awful lot of source code listings as well. Do have a good browse through them.

SERIAL OUTPUT

With the ability to write/read serial data assured, the next stage to be solved was that of instructing the PIC16F877 to output the data as a serial stream at a known baud rate. This turned out to be very straightforward.

The PIC16F87x family have a built-in structure which allows serial output through a dedicated pin, RC6, and for the rate of output to be selected according to the needs of the destination for that data (e.g. a PC) and in relation to the PIC's crystal controlled clock rate.

Additionally, these PICs can be instructed on such matters as synchronous or asynchronous transmission, parity, stop bits and byte size.

The 'F87x data sheet, once you have studied it for details of serial interfacing, is actually quite helpful.

It was decided to use asynchronous serial transmission from the PIC to the PC. This requires only two lines to be connected between the two systems, one for data and one for the ground (0V).

It was further decided that the rate of transmission should be at the maximum likely to be found on the majority of readers' PCs, 9600 baud, with no parity, one stop bit and with 8-bit transmission.

The registers associated with serial transmission are:

SPBRG ($99 – PAGE1) Baud rate generator
TXSTA ($98 – PAGE1) Transmit status and control
RCXTA ($18 – PAGE0) Receive status and control
PIE1 ($8C – PAGE0) Peripheral interrupt control

Data sheet tables are provided for establishing the value to be set into register SPBRG in relation to several examples of clock rate: DS-Table 10-4 and DS-Table 10-5. Since these do not quote a value for a 3·2748MHz clock, as used in the *Data Logger*, the formula quoted in the data sheet's Example 10-1 was more useful on this occasion.

TIMING FORMULAE

In fact, two formulae can be derived from Example 10-1, depending on whether the PIC is to divide the clock rate by 64 or 16. The idea is to select a division rate to produce as large an SPBRG value as possible, up to a maximum of 255 (it's an 8-bit register).

In Listing 2 is shown a Basic program derived from the formula in Example 10-1. It calculates the SPBRG value in relation to the baud rate required, the clock rate available, and the two division factors of 64 and 16. Bit BRGH (bit 2) of the TXSTA register has to be set low if 64 is the divider, and high for a divider of 16.

Running the program in Listing 2 produces SPBRG answers of 4·333333 (BRGH = 0) and 20.3333 (BRGH = 1). Since the latter is the higher, the BRGH bit has to be set to 1.

The bit which tells the PIC whether it is required to transmit synchronously or asynchronously is TXSTA register bit 4 (SYNC). DS-Table 10-1 shows that for asynchronous transmission the SYNC bit is set to 0.

Prior to commencing transmission, the basic transmission parameters are set as in Listing 3, with entry at label SETBAUD.

Since some of the affected registers are in PAGE1, this is set first, as in command lines 1 and 2. Then the integer of the chosen SPBRG value (20) is stored into the SPBRG register, register TXSTA is conditioned for SYNC = 0 and BRGH = 1, TRISC bit 6 is cleared for pin RC6 to be used as an output, and the transmission interrupt bit (PIE1,4) is cleared (interrupt not required). A reset to PAGE0 is made and the SPEN bit (bit 7) of register RCSTA is set.

This sequence is in accordance with the first three steps listed in the data sheet at Section 10.2.1. Step 4 (9-bit transmission) is not required. Step 5 (enable transmission) requires a reset to PAGE1, and the transmission bit (bit 5) of TXSTA is set, followed by a reset to PAGE0. The scene is now ready for data to be transmitted from PIC pin RC6, and the SETBAUD routine is exited.

LISTING 2

```
'calculate SPBRG for required baud rate

BAUD = 9600     ; replace this value with your own
FOSC = 3276800 ; replace this value with your own
X = (FOSC / (64 * BAUD)) - 1
Y = (FOSC / (16 * BAUD)) - 1
PRINT "BAUD =";  BAUD, "CLOCK =";  FOSC
PRINT "SPBRG at div 64 = ";  X;  "bit BRGH = 0"
PRINT "SPBRG at div 16 = ";  Y;  "bit BRGH = 1"
```

SERIAL FORMAT

In the *Data Logger* the SETBAUD sequence is performed when power is first switched on and it remains in a state of readiness to send data to the PC until power is switched off again.

Consequently, having read the 2-byte sample data stored in the selected serial memory, it can be sent to the computer through the routine commencing at label SENDPC, shown in Listing 4.

However, before it can be sent, a slight readjustment of the data format is needed. The PC register which receives the serial data shifts it left by one place (multiplying it by two). This means that the PIC cannot send a data byte whose value is greater than 127. If it were to, bit 7 would be "lost" at the PC end.

To over come this, the least significant byte of recalled data (held in MEMLO, as discussed earlier) has to be limited to a value of less than 128 and its eighth bit (bit 7) combined with the most significant byte (MEMHI), whose recalled value is never greater than three.

On entry into routine SENDPC in Listing 4, this rearrangement takes place in the first four lines. An example of what happens is as follows:

Suppose MEMHI holds a value of %00000011 and MEMLO holds %11111111. MEMLO is first rotated left into the W register (leaving MEMLO itself untouched) so that its bit 7 "drops" into the Carry register. MEMHI is now rotated left and in doing so the Carry bit is rotated into it from the right. MEMLO's bit 7 can now be cleared, limiting MEMLO's value to less than 128.

The result is that MEMHI now holds %00000111 and MEMLO holds %01111111 and it is these values that are

LISTING 3

```
SETBAUD: BCF RP1          ; clear PAGE2/3
         BSF RP0          ; set PAGE1
         MOVLW 20         ; BRG val for 9600 baud
                          ; from 3.2768MHz, brgh=1
         MOVWF SPBRG      ; put into SPBRG reg
         MOVLW %00000100  ; sync=0 (bit 4), brgh=1
                          ; (bit 2), clear other bits
         MOVWF TXSTA      ; put into TXSTA
         BCF TRISC,6      ; set RC6 as output
         BCF PIE1,4       ; clear interrupt bit (TXIE)
         BCF RP0          ; set back to PAGE0
         MOVLW %10000000  ; set SPEN bit of RCSTA reg
         MOVWF RCSTA      ;
         BSF RP0          ; set for PAGE1
         BSF TXSTA,5      ; enable transmission (TXEN)
         BCF RP0          ; set for PAGE0
         RETURN           ;
```

LISTING 4

```
SENDPC:  BCF STATUS,C     ; clear Carry flag
         RLF MEMLO,W      ; rotate MEMLO left into W,
                          ; bit 7 enters Carry
         RLF MEMHI,F      ; rotate MEMHI left, Carry
                          ; enters bit 0
         BCF MEMLO,7      ; clear MEMLO bit 7
         MOVF MEMLO,W     ; get MEMLO
         CALL SERIAL1     ; send it to PC
         MOVF MEMHI,W     ; get MEMHI
         CALL SERIAL1     ; send it to PC
         RETURN           ; end of routine
```

LISTING 5

```
SERIAL1: BTFSS PIR1,4     ; wait for bit 4 = 1
                          ; (showing TXREG empty)
         GOTO SERIAL1     ;
         NOP              ;
         MOVWF TXREG      ; put val in TXREG ready
                          ; for auto-transmission
         RETURN           ; end of routine
```

transmitted to the PC via the called routine whose label is SERIAL1 (see Listing 5). How they are restored to their "true" values will be discussed shortly. Once that byte pair has been transmitted, the next pair can be read from one of the serial memories, and again transmitted via the SENDPC routine.

REGISTERS TXREG AND TSR

The routine that allows each byte to be transmitted, SERIAL1, has only three active commands. As soon as the PIC's serial transmission register (TXREG) is loaded with a byte of data, transmission is started by the PIC's own internal facilities.

To summarise the PIC data sheet, the heart of the transmitter is the Transmit Serial Register (TSR), which obtains its data from the read/write transmit buffer register TXREG. The user's software loads TXREG with the data byte to be transmitted, where it stays until the Stop bit from the previously loaded data has been sent.

As soon as the Stop bit has been transmitted, the TSR is automatically loaded with the new data from TXREG. Once this has occurred, TXREG is now empty and a flag bit is set in register PIR1 – its bit 4, named TXIF. When this flag is set, the software can load the next byte of data into TXREG, an action which clears the TXIF bit.

The routine entered at SERIAL1 first checks the status of PIR1 bit 4. If the bit is low, a previous transmission is still taking place. The routine loops continuously checking bit 4 until it is set. Prior to entry to SERIAL1, the W register was loaded with the data byte (as in Listing 4). When PIR1 bit 4 is found to be set, the W data is loaded into TXREG, to be automatically transmitted out by the PIC.

As soon as TXREG has been loaded, the SERIAL1 routine ends, and software can get the next byte, or do whatever it is told to do, such as end the full transmission sequence because all the bytes have been sent.

IRREGULAR TRANSMISSION

The *Data Logger* sends its serial data to the PC in consecutive blocks of data. However, in a another design on which the author is working, the need is for serial data to be output to the PC at irregular intervals, two bytes a time.

Whereas for consecutive data blocks, setting the Baud Rate factors at the head of the program proved satisfactory, in the random transmission design, the author found it necessary to send a byte of zero prior to each double-byte being sent.

What subtlety of difference between the two programs makes this action necessary has not been established, although it is believed that it might be to do with PORTC being read between data words in the second design. PORTC is that which has to be used for serial input/output, and it seems possible that reading it (for switch status) between data words might affect the serial registers. It has yet to be more fully investigated.

PC RECEPTION

A program for use on a PC to receive serial data from a PIC is not included in Microchip's Applications Notes software listings library. A browse of the Internet for suitable software did not reveal anything that the author felt was suitable either. Consequently, he wrote his own routines specifically to import double-byte serial data from the *Data Logger*.

The program is written in a mixture of Basic (suited to running from QBasic or QuickBASIC) and machine code. The Basic program loads and calls the machine code, which does the actual serial data importing, and then formats the data for output to disk, in several different file styles, as discussed in the *Data Logger* text.

There are, in fact, several ways in which machine code can be accessed from Basic. The example shown in Listing 6 is the one on which the author has standardised for several years.

LISTING 6

```
LOADDATACODE:                          ' load machine code subroutine
OPEN FILE$ FOR BINARY AS #1            ' open file named in FILE$
B = LOF(1)                             ' get length of file
SERIAL$ = INPUT$(B, #1)                ' load file into SERIAL$
CLOSE 1                                ' close file
A = 256                                ' define A as value 256
MA = VARSEG(MA%(0))                    ' get segment address of MA%(0)
IF MA < 0 THEN MA = MA + 65536!        ' correct for negative value
MB = VARPTR(SERIAL$)                   ' get pointer to SERIAL$
MC = PEEK(MB + 3) * A + PEEK(MB + 2)   ' get address of SERIAL$
MD = INT(MA / 256)                     ' get MSB
MB = MA - (MD * 256)                   ' get LSB
POKE MC + 5, MB                        ' poke LSB into machine code
POKE MC + 6, MD                        ' poke MSB into machine code
RETURN                                 ' end of subroutine
```

The majority of the Basic routines are self-explanatory to anyone who knows QBasic or QuickBASIC and will not be discussed here. However, the routines which access the machine code, and the machine code itself, deserve a bit of explanation.

On running the Basic program, integer variable array MA%(x) is first DIMmed for the maximum number of separate values (32766) as are required for access by the machine code. All the values are in consecutive order within the PC's memory and their exact locations are accurately predictable.

Referring to Listing 6, the machine code whose file name is held in FILE$ is then loaded as binary data into string variable SERIAL$, at label LOADDATACODE. The address at which MA%(0) resides is then obtained, and POKED into the machine code at two predetermined consecutive addresses.

Note that in line 10 (MC =) attempting to multiply by 256 as a "live" value would result in an "overflow" error and so variable A is used, having been allocated that value in line 6. Also, because integer variables whose true values are greater than 32767 are returned as negative numbers, line 8 intercepts them if they occur, restoring them to their correct positive value. Additionally note that variable MB is used for two different purposes.

Once the routine in Listing 6 has been run, the machine code is accessed through the command:

CALL ABSOLUTE(SADD(SERIAL$))

When the machine code routine has ended, the program reverts to Basic.

It is important to note that there is a slight difference between using QBasic and QuickBASIC in that QuickBASIC has to be loaded with the command QB/L, which automatically loads an additional library program (part of the QuickBASIC suite) which allows machine code to be run. QBasic does not require the additional library program and is simply loaded in the usual way with the command QB.

The machine code routine is shown in Listing 7. It is based on two PC interrupt calls to INT 14H, whose functions are documented in the *PC Sourcebook* – a publication which itemises the principal registers and interrupt calls for base-standard PCs (seemingly compatible with processors from the 8086 upwards, including Pentiums).

On entry to the machine code, the address of MA%(0) is acquired from the value held at label SETSEGMENT, as previously POKED there from Basic.

Transmission format data passed from Basic (held in MA%(0)) is then read and loaded into the AX register. The data details the baud rate, parity, stop bit, and bit count configuration (assembled from

the details in Table 1). The Basic software also passes details on which COM port (COM1 or COM2) is to be read (held in MA%(1)). This is loaded into the BX register to be then loaded into register DX, whereupon interrupt INT 14H is called, which passes the data to the PC's serial operating system.

Routine WAITDATASET is now called, in which INT 14H is polled until register AX returns with a value less than 256, signifying that a byte of serial data has been received and that it is now available in the low byte (AL) of register AX.

This data byte is an inversion of the byte presented to the PIC for transmission and is shifted left by one place. The inversion is corrected by subtracting the byte from 255.

Returning to the calling point, the byte is stored in register CL as the least significant byte of the double-byte required. WAITDATASET is again called, where the second of the two bytes needed is similarly acquired, and then stored in register CH. Note that CH and CL are the high and low bytes, respectively, of register CX.

Now a mixture of rotation and ANDing reconstitutes the double-byte data to its original value as stored in the *Data Logger's* serial memory. The value is then stored in the pre-determined location in the PC's memory relative to the integer array position back in Basic (from MA%(0) onwards).

The value is also checked to see if it is the end marker value transmitted by the PIC when a serial memory has been fully read. If it is not that value (two bytes each of value 127), the next double byte of transmitted data is acquired, and stored at the next consecutive PC memory location.

Back in Basic, the locations in which the machine code was stored (MA%(0) onwards) is then saved as a block whose length is the count value reached at the end of the machine code sequence.

KEYBOARD INTERRUPT

A second serial input machine code routine is included with the *Data Logger* software. This includes the ability to press "Q" to quit from the machine code without waiting for the data transfer to be complete. This routine makes use of the INT 16H keyboard access interrupt to read which keys are pressed and to return a value accordingly.

LISTING 7

```
JMP STARTALL          ; entry point from Basic

SETSEGMENT:
MOV DS,01111          ; value changed from Basic
RET                   : for VARSEG(MA%(0))

STARTALL:
PUSH DS               ; store current data segment
CALL SETSEGMENT       ; get MA%(0) segment address
MOV SI,0              ; set source address count to zero
MOV AX,[SI]           ; get baud rate etc from MA%(0)
MOV BX,[SI+2]         ; get COM port number from MA%(1)
MOV DX,BX             ; load config data via
INT 14H               ; interrupt call INT 14H

GETDATA:
CALL WAITDATASET      ; get first data byte
MOV [SI],CL           ; temporarily store it
CALL WAITDATASET      ; get second data byte
ROR CL,1              ; rearrange bytes to orig format
MOV [SI+1],CL         ; and store them
RCR [SI+1][B],1       ; rotate byte right
RCR [SI][B],1         ; rotate byte right
AND [SI+1][B],127     ; clear bit 7
INC SI                ; double-increment store address
INC SI                ;
CMP [SI-2][B],255     ; is this byte = 127?
JNE GETDATA           ; no, so get next sample
CMP [SI-1][B],63      ; yes, is this byte now = 63?
JNE GETDATA           ; no, so get next sample
MOV [SI][W],0         ; yes, clear last stored word
MOV CX,SI             ; get count value
ROR CX,1              ; multiply by 2
MOV SI,0              ; and store it
MOV [SI],CX           ; in MA%(0)
POP DS                ; recall orig data segment
RETF                  ; exit back to Basic

WAITDATASET:          ; get data byte
MOV AH,2              ; set interrupt factors
MOV AL,0              ;
MOV DX,BX             ;
INT 14H               ; call interrupt INT 14H
CMP AX,255            ; is returned value =< 255?
JA WAITDATASET        ; no, repeat INT call
MOV CL,255            ; yes, invert received value
SUB CL,AL             ; (subtract from 255)
RET                   ; return to calling routine
```

A point of interest, however, is that although three of the author's computers would respond to the INT 16H call, a fourth (a "custom-built" machine used at *EPE* HQ) would not. This is puzzling since it was believed that the basic interrupt calls on PCs are upwards compatible – seemingly not in some instances. Can anyone throw light on this?

TABLE 1: INT 14H Com port parameter byte

7	6	5	4	3	2	1	0	Description	Allowable Values
✔	✔	✔						Baud rate	000=110 baud 001=150 010=30 011=600 100=1200 (default) 101=2400 110=4800 111=9600
			✔	✔				Parity	00=No parity 01=Odd parity 10=No parity 11=Even parity
					✔			Stop bits	0=1 stop bit, 1=2 stop bits
						✔	✔	Word length	10=7 bits 11=8 bits

PIC EEPROM DATA MEMORY

The routines for writing to and reading from the PIC16F87x family's internal EEPROM data memory are worth highlighting. You will no doubt be familiar with the same routines as used with the PIC16x84 devices, and the 'F87x routines are similar, but not exactly the same since the 'F87x devices use different register locations to those used by the 'x84, as shown in Table 2.

The EEPROM data memory Write/Read routines are shown in full in Listing 7 and Listing 8. More information on them is on Microchip data sheet DS30292A page 43.

As with the 'x84 programming examples given in the *PIC Tutorial* and *PICtutor* texts (to which readers are also referred – either text source will do), the EEPROM Write routine at label SETPRM 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. Whereas the 'x84 routine was actioned in PAGE0 and PAGE1, the 'F87x routine is actioned in PAGE0, PAGE2 and PAGE3, hence the various RP0 and RP1 paging instructions.

It is worth noting that the 'F87x data sheet gives a programming example (page 43) in which a SLEEP command and interrupt are used to determine when the EEPROM Write function has been completed, rather than polling EECON1 bit 4. However, when the author tried the SLEEP method it failed to work. It is probable that another undocumented action has to be taken in addition to those shown, but this has not been investigated.

The EEPROM Read routine in Listing 8 is entered at label PRMGET with W holding the EEPROM byte address to be read. It is exited with W holding the data read from the EEPROM.

ASSEMBLER SOFTWARE

Readers who are interested in learning how to program in machine code suited to the 8086 and above processors are recommended to obtain the *excellent* shareware *A86/D86 Assembler/Disassembler* from the Public Domain Shareware Library (PDSL) whose details are given later.

The author has been using it for many years and for many PC-controlled *EPE* projects, including the *Virtual Scope* and *PIC Toolkit Mk1*. It is nearly as easy to learn as PIC programming, but has far more commands available. A useful associated book is Intel's *8086/8088 User's Manual*.

OBTAINING BASIC

Until fairly recently, by far the vast majority of PCs will have been supplied with either QBasic or QuickBASIC installed. However, *EPE* does sometimes get questions from readers who do not have either and ask where they can obtain one or the other.

So far as is known, neither of the programs is actually supplied with PCs any longer – Microsoft wishing, perhaps, that users should acquire the more advanced VisualBASIC. The once-popular GW-Basic has long since been outdated and is not compatible with regard to using it with the QB machine code routines illustrated here.

LISTING 7

```
SETPRM: BSF STATUS,RP1    ; set for PAGE2
        BCF STATUS,RP0
        MOVWF EEADR       ; copy W into EEADR to set
                          ; EEPROM address
        BCF STATUS,RP1    ; set for PAGE0
        MOVF STORE1,W     ; get data value from STORE1
                          ; and hold in W
        BSF STATUS,RP1    ; set for PAGE2
        MOVWF EEDATA      ; copy W into EEPROM data
                          ; byte register
        BSF STATUS,RP0    ; set for PAGE3
        BCF EECON1,EEPGD  ; point to Data memory
        BSF EECON1,WREN   ; enable write flag

MANUAL: MOVLW $55         ; these lines cause the action
        MOVWF EECON2      ; required by the EEPROM to
        MOVLW $AA         ; store the data in EEDATA
        MOVWF EECON2      ; at the address held by EEADR.
        BSF EECON1,WR     ; Set `perform write' flag
        BCF STATUS,RP1    ; set for PAGE0
        BCF STATUS,RP0    ;

CHKWRT: BTFSS PIR2,EEIF   ; wait till bit 4 of PIR2 set
        GOTO CHKWRT       ;
        BCF PIR2,EEIF     ; clear bit 4 of PIR2
        RETURN            ;
```

LISTING 8

```
PRMGET: BSF STATUS,RP1    ; set for PAGE2
        BCF STATUS,RP0    ;
        MOVWF EEADR       ; copy W into EEADR to set
                          ; EEPROM address
        BSF STATUS,RP0    ; set for PAGE3
        BCF EECON1,EEPGD  ; point to data memory
        BSF EECON1,RD     ; enable read flag
        BCF STATUS,RP0    ; set for PAGE2
        MOVF EEDATA,W     ; read EEPROM data now in
                          ; EEDATA into W
        BCF STATUS,RP1    ; set for PAGE0
        RETURN            ;
```

Readers who would like to get one or other of the QB versions are recommended to obtain it through the Internet. There are quite a lot of sites once you start looking. A general "search" call with the keyword QBASIC should begin to unravel the web of leads.

TABLE 2: Comparison of EEPROM data memory registers

PIC16F87x			PIC16x84 Equivalent		
Register	**Address**	**Page**	**Register**	**Address**	**Page**
PIR2	$0D	0	None	–	–
EEDATA	$10C	2	EEDATA	$08	0
EEADR	$10D	2	EEADR	$09	0
EECON1	$18C	3	EECON1	$88	1
EECON2	$18D	3	EECON2	$89	1
Register	**Bit**	**Name/No**	**Register**	**Bit**	**Name/No**
PIR2	EEIF	4	EECON1	EEIF	4
EECON1	RD	0	EECON1	RD	0
EECON1	WR	1	EECON1	WR	1
EECON1	WREN	2	EECON1	WREN	2
EECON1	EEPGD	7	None	–	–

SOURCES

The full data sheets for the Microchip devices used in the Data Logger are available from Microchip: PIC16F87x family, code DS30292A, serial EEPROM memories: DS21203D (24AA256), DS21191C (24AA128), DS21189B (24AA64), DS21162C (24AA32). There are three ways to obtain them from Microchip: as downloads from their web site, from their fully inclusive CD-ROM (all products data and applications info), or as individual booklets.

Microchip Technology Ltd., Microchip House, 505 Eskdale Road, Winnersh Triangle, Woking, Berks RG41 5TU. Tel: 0118 921 5800. Fax: 0118 921 5835.

E-mail: techdesk@arizona.co.uk. Web: http://www.microchip.com.

Public Domain Shareware Library: PDSL, Dept EPE, Winscombe House, Beacon Road, Crowborough, East Sussex TN16 1UL. Tel: 01892 663298. Fax: 01892 667473.

Intel data books are available from Electromail, Tel: 01536 204555. □

The Programmer's PC Sourcebook is a Microsoft Press publication, ISBN 1-55615-321-X.