EPE PIC TUTORIAL V2

JOHN BECKER PART TWO

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

In this part we play with switches, make noises, count times, and generally have fun with some more PIC16F84 commands!

TUTORIAL 7
CONCEPTS EXAMINED

Switch monitoring
Command ANDLW
Command ANDWF
Command ADDWF
Command ADDLW
Nibbles
STATUS bit 1
Digit Carry flag
Bit code DC

CONNECTIONS NEEDED

All Port B to all l.e.d.s.
Port A RA0-RA3 to switches SW0-SW3 (via CP19-CP16)
CP21 to +5V OUT
CP20 to 0V OUT
Capacitor C7 as 1μF
Preset VR1 set to minimum resistance (fully clockwise)

From hereon we shall usually omit the program initialisation commands that have up to now been shown at the top of each listing. Some will be included where they help to clarify the program. Otherwise, assume that any name used in the listing extracts shown will have been defined or equated in the headings. The commands are included in full on the disk file program listings (source code).

We now turn to looking at how data is input via switches and shall continue to show the results on individual l.e.d.s. In Tutorials 21 and 22 we shall look at 7-segment l.e.d.s and alphanumeric l.c.d.s as the output displays.

First connect TK3's pushbutton switches SW0, SW1, SW2, SW3 (via CP19, CP18, CP17, CP16) to PORTA pins RA0, RA1, RA2, RA3 respectively. Connect the switch power pin CP21 to the +5V OUT pin, and switch power pin CP20 to the 0V OUT pin. Port pins RA0 to RA3 are now connected so that they are normally biased low (to 0V) but will go high (+5V) when their respective switches are pressed.

LISTING 8 – PROGRAM TK3TUT8

```
BEGIN   CLRF COUNT
LOOP    MOVF PORTA,W
        ANDLW B'00000001'
        ADDWF COUNT,F
        MOVF COUNT,W
        MOVWF PORTB
        GOTO LOOP
```

Run TK3TUT8.HEX. Pushing switch SW0 on and off, PORTB's l.e.d.s will be seen to go on and off in a binary sequence when the switch is on (pressed), but will remain in the last condition when the switch is off (released). In this example, the program tests whether the status of switch SW0, which is connected to PORTA RA0 (bit 0), is on or off. If the switch is on then the counter variable, COUNT, is repeatedly added to (by 1 in this example). A value of zero is added to the count if the switch is off. The count value is output to PORTB.

First let's look at two of the commands introduced here, ANDLW and ADDWF. Their counterparts ANDWF and ADDLW will also be examined.

COMMANDS ANDLW AND ANDWF

As no doubt most of you are aware, if one binary number is ANDed with another, then only if the same bits of both numbers are set (1) will the answer also have a 1 in that position. Any zeros on either or both sides for any bit will automatically produce a result of 0, e.g.:

First number:	01110010
Second number:	01011001
ANDed answer:	01010000

This technique is widely used in electronics and computing, the final answer determining the subsequent action to be taken by a circuit or software routine.

There are two ANDing commands available with PICs, ANDLW (AND Literal to W), and ANDWF (AND W with File value). Suppose that the first number in the foregoing examples (01110010) is already contained within W, we then wish to AND it with a fixed number as stated in a program command. Assuming that the fixed number is the second number quoted, the command is:

ANDLW B'01011001'

The PIC ANDs the second (literal) number with that already held in W. The answer (01010000) is retained by W and is available to be further manipulated or copied into any file as specified by the command which follows ANDLW. You could, for example, use the command MOVWF PORTB which will turn on l.e.d.s LD6 and LD4 (01010000).

Any of the three numerical formats may be used with ANDLW, e.g. B'00011111' (binary), H'1F' (hexadecimal), 31 (decimal), are all legitimate and equal. It is also legitimate to use a name that has been equated with a value, e.g. ANDLW PORTB (which would AND 6 with W since we have previously specified that the name PORTB represents the value 6).

The command ANDWF is used to AND an existing value within W to a value within a named file, either retaining the answer in W (ANDWF FILENAME,W) or putting back in the named file (ANDWF FILENAME,F).

It is not possible to directly AND the contents of two files together, the value of one or other file must have already been moved into W before the ANDing can take place. With both commands ANDLW and ANDWF, if the answer is zero, the Zero

flag of STATUS is set. If the answer is greater than zero, the Zero flag is cleared. Zero is the only flag affected by an AND command.

COMMANDS ADDLW AND ADDWF

There are two ADDing commands available with PICs, ADDLW (ADD Literal to W), and ADDWF (ADD W to a File value). Command ADDLW is used where a fixed number (literal) within a program is to be added to an existing value within W and which has been obtained by a previous operation. Suppose that W holds the answer produced in the previous ANDing example, 01010000 (decimal 80), and you wish to add a fixed value to it, 53 decimal (00110101), for instance. The command would be:

ADDLW 53 (or ADDLW H'35' hexadecimal, or ADDLW B'00110101' binary).

The answer in this instance is 10000101 (decimal 133) and is retained in W for further use or copying into a file, e.g. MOVWF PORTB.

Command ADDWF adds the contents of W to the value within a stated file. The answer can be held in W (ADDWF PORTB,W) or put back into the named file (ADDWF PORTB,F).

Three flags within STATUS are affected by any ADD command, Carry, Zero and Digit Carry. If the answer to an addition is greater than 255, the Carry flag is set, otherwise it is cleared. If the answer equals zero, the Zero flag is set, otherwise it is cleared. The third flag, Digit Carry, you have not encountered yet. Although the concept is not illustrated until later (Tutorial 19), it is appropriate to describe it now.

If you imagine that an 8-bit binary number (e.g. 10110110) is split into two halves (known as "nibbles"), 1011 and 0110, the righthand nibble is monitored by the PIC as a separate digit and it is served by its own flag, the Digit Carry flag. If an addition takes place which produces a result greater than 15 (binary 1111) for that nibble, the Digit Carry flag is set, otherwise it is cleared.

LISTING 8 FLOW

Having described the new terms, we shall now detail what happens in Listing 8. As said at the start of Tutorial 7, switches SW0 to SW3 are biassed so that their respective PORTA pins are normally at 0V (low) but go high when pressed. In this example program, at the label LOOP the contents of PORTA are copied into W (MOVF PORTA,W), which then holds the status of all five usable bits of that port. We are only interested, though, in the status of the switch on PORTA bit 0, switch SW0. Therefore, in the next command (ANDLW B'00000001') bit 0 is ANDed with 1 to isolate its value, the other seven bits in W being cleared by the respective zeros of the ANDed value.

The answer in W is then added to the contents of the counter (ADDWF COUNT,F). Next, the contents of the counter are brought back into W (MOVF COUNT,W) and then copied into PORTB (MOVWF PORTB), whose l.e.d.s are turned on or off depending on the binary count value. With the command GOTO LOOP, the sequence is repeated.

It will be seen that there is only an increase in the count value if PORTA bit 0 holds a 1, therefore the count will only change if the switch is on (pressed). Pressing any other switch connected to PORTA has no effect. When the counter passes 255, its value rolls over to zero and starts counting upwards again.

EXERCISE 7

7.1. Can you see another way of writing the first two lines using MOVLW and ANDWF?

7.2. Can you see how the BTFSS or BTFSC commands might be used to achieve the same output result; the use of MOVLW 1 or ADDLW 1 could be useful here.

7.3. There is also the opportunity to use INCF in this type of situation. Try rewriting to include this command.

TUTORIAL 8

CONCEPTS EXAMINED

Increasing speed of TK3TUT8
Bit testing for switch status

CONNECTIONS NEEDED

All Port B to all l.e.d.s.
Port A RA0-RA3 to switches SW0-SW3 (via CP19-CP16)
CP21 to +5V OUT
CP20 to 0V OUT
Capacitor C7 as 1μF
Preset VR1 set to minimum resistance (fully clockwise)

LISTING 9 – PROGRAM TK3TUT9

```
LOOP    BTFSS PORTA,0
        GOTO LOOP
        INCF COUNT,F
        MOVF COUNT,W
        MOVWF PORTB
        GOTO LOOP
```

In TK3TUT8 we saw that the count adding commands etc. were performed even if the count value was zero. This is a waste of processing speed, why bother to add zero to a count? The program in Listing 9 shows a faster alternative. Run TK3TUT9.HEX.

By using the command BTFSS to check the status of a switch (in this case still SW0 on PORTA bit 0), if the switch is not pressed we can avoid the count incrementing procedure, jumping immediately to a further switch status test. Alternatively, in another program, by substituting another destination instead of LOOP, we could jump to a totally different routine and perform some other procedure.

Another choice is to use the command RETURN instead of GOTO LOOP to return to another routine which had called this one. Commands CALL and RETURN will be covered in Tutorial 13.

It is expected that you will recognise from Listing 9 what the program does and how it does it. If you don't, re-read Tutorial 4 and the section on BTFSS.

EXERCISE 8

8.1. What happens if you use BTFSC instead of BTFSS?

8.2. Could one of the Zero flag testing commands be used instead of BTFSS? If so, how, and would an AND command be useful? (Remember that PORTA has more bits than just bit 0).

TUTORIAL 9

CONCEPT EXAMINED

Responding to a switch press only at the moment of pressing

CONNECTIONS NEEDED

All Port B to all l.e.d.s.
Port A RA0-RA3 to switches SW0-SW3 (via CP19-CP16)
CP21 to +5V OUT
CP20 to 0V OUT
Capacitor C7 as 1μF
Preset VR1 set to minimum resistance (fully clockwise)

LISTING 10 – PROGRAM TK3TUT10

```
BEGIN   CLRF COUNT
        CLRF SWITCH
TESTIT  BTFSC PORTA,0
        GOTO TSTPRV
        BCF SWITCH,0
        GOTO TESTIT
TSTPRV  BTFSC SWITCH,0
        GOTO TESTIT
        INCF COUNT,F
        MOVF COUNT,W
        MOVWF PORTB
        BSF SWITCH,0
        GOTO TESTIT
```

In the switch press examples of Listings 8 and 9, we saw that the counter was incremented for the entire duration of the switch being on. Often, only a single response to a change of switch status might be required. This entails testing the switch status and comparing it with a previous test. Only if the switch is on and if that on condition has not yet been responded to will the next action be performed.

Load TK3TUT10.HEX. You are still monitoring PORTA bit 0 for the switch press (SW0), responding to it via the l.e.d.s on PORTB. Observe the l.e.d.s while pressing SW0 on and off. For each pressing, only one change of the l.e.d. count will occur (but note that low-cost switches may cause switch-bounce, resulting in the count increasing for each bounce – a matter covered later).

Study Listing 10: the entry to the routine is at BEGIN where two variables, COUNT and SWITCH are cleared. At the label TESTIT, the command is BTFSC PORTA,0, testing the status of PORTA bit 0 (is it clear?). Remember that we are only interested in the bit being set. If it is false that bit 0 is clear (i.e. that it is set – the switch is pressed) the command GOTO TSTPRV is performed and then the status of SWITCH bit 0 is tested, BTFSC SWITCH,0. This bit serves as the flag to keep track of the previous status of the switch. At this moment, the bit will be clear because the whole byte was cleared on entry to the routine. Consequently, the GOTO TESTIT command is skipped, the count is incremented and its value output to PORTB.

Now SWITCH bit 0 is set (BSF SWITCH,0) to indicate that the count has been incremented for this switch press (i.e. the flag is set), and the program jumps back to TESTIT. If the switch is still pressed, then at TSTPRV the BTFSC SWITCH,0 command will produce a false answer and the command GOTO TESTIT will be performed, thus preventing the counter from being further incremented at this time.

What is now needed is for the switch to be released so that the two commands BCF SWITCH,0 (clear the flag) and GOTO TESTIT can occur. The stage is then once again set for the next switch press to be responded to by the counter.

EXERCISE 9

9.1. In Listing 10, AND and MOV commands could have been used instead of BTFSC and BCF. How, and with what other command?

9.2. Would using BTFSS instead of BTFSC involve more commands and labels having to be used as well?

9.3. Because low cost switches have probably been used, there is the danger that mechanical switch bounce might occur, causing the count to be incremented undesirably. Another counter could be used to cause a delay in the rate of switch testing to eliminate the effects of switch bounce. How would you implement the delay, and where would you put the commands required. Hint, another label will be needed as well.

TUTORIAL 10

CONCEPTS EXAMINED

Performing different functions depending upon which of two switches is pressed

The use of a common sub-routine serving two other routines

CONNECTIONS NEEDED

All Port B to all l.e.d.s.

Port A RA0-RA3 to switches SW0-SW3 (via CP19-CP16)

CP21 to +5V OUT

CP20 to 0V OUT

Capacitor C7 as 1μF

Preset VR1 set to minimum resistance (fully clockwise)

Run TK3TUT11.HEX and experiment with the switches on PORTA bits 0 and 2 (SW0 and SW2). You will discover that switch SW0 causes the count displayed on the l.e.d.s to be increased, and that switch SW2 decreases the count. The basic logic flow is the same as that in Listing 10, except that two switches are used and each switch is responsible for a different routine.

Note that whilst each switch could have had its own routine to output to PORTB, the two routines would be the same. Consequently, each switch routine is routed into a common output sub-routine (OUTPUT). At the end of SW0's routine, the command GOTO OUTPUT needs to be given, but at the end of SW2's routine, no GOTO OUTPUT command is needed because OUTPUT follows immediately after it. It is said to reach OUTPUT by *default* because it does not need to be told to go there.

LISTING 11 – PROGRAM TK3TUT11

```
BEGIN   CLRF COUNT
        CLRF SWITCH
TEST1   BTFSC PORTA,0
        GOTO TSTPR1
        BCF SWITCH,0
        GOTO TEST2
TSTPR1  BTFSC SWITCH,0
        GOTO TEST2
        BSF SWITCH,0
        INCF COUNT,F
        GOTO OUTPUT
TEST2   BTFSC PORTA,2
        GOTO TSTPR2
        BCF SWITCH,2
        GOTO TEST1
TSTPR2  BTFSC SWITCH,2
        GOTO TEST1
        BSF SWITCH,2
        DECF COUNT,F
OUTPUT  MOVF COUNT,W
        MOVWF PORTB
        GOTO TEST1
```

EXERCISE 10

10.1. How do you think a single test for *neither* of the switches being pressed could be introduced, shortening the testing time? Could an AND be used with a STATUS check, or can a STATUS check be used on its own without an AND? (Think carefully about the latter.)

10.2. How would you increase the count by more than one, say two, at each press of switch SW0? With the knowledge you've gained so far, three ways should come to mind, one of them including the use of a new named variable.

10.3. If you want to add 255 each time a switch SW0 press occurs, do you need an ADD command, or is there another command which will do the same job? (Think *rollover*.)

TUTORIAL 11

CONCEPTS EXAMINED

The ease of reflecting PORTA's switches on PORTB's l.e.d.s!

Command COMF

Command SWAPF

Inverting a byte's bit logic

Swapping a byte's nibbles

CONNECTIONS NEEDED

All Port B to all l.e.d.s.

Port A RA0-RA3 to switches SW0-SW3 (via CP19-CP16)

CP21 to +5V OUT

CP20 to 0V OUT

Capacitor C7 as 1μF

Preset VR1 set to minimum resistance (fully clockwise)

Load TUT12.HEX. Experiment with pressing any combination of the four switches on PORTA (SW0 to SW3) while observing the l.e.d.s on PORTB. This routine should need no further comment. Another way of expressing the first two commands is:

LISTING 12 – PROGRAM TK3TUT12

```
LOOP    MOVF PORTA,W
        ANDLW B'00001111'
        MOVWF PORTB
        GOTO LOOP
```

```
LOOP    MOVLW B'00001111'
        ANDWF PORTA,W
```

Now load TUT13.HEX and run it, again experimenting with pressing any combination of the switches on PORTA (SW0 to SW3) and observing the l.e.d.s on PORTB.

LISTING 13 – PROGRAM TK3TUT13

```
LOOP    SWAPF PORTA,W
        ANDLW B'11110000'
        MOVWF PORTB
```

You will see while you press PORTA's four switches, that they are having their status displayed on PORTB's four lefthand l.e.d.s (LD7 to LD4), even though you have not changed the wiring to PORTB and the l.e.d.s. Had there been a fifth switch, on PORTA RA4, it would be affecting the first l.e.d. on the right (LD0) – if a different AND value were used (what value?).

What is happening is that the software has been told to swap and move into W (SWAPF PORTA,W) the left and righthand four bits of PORTA (its nibbles, as introduced in Tutorial 7). The answer is then ANDed with bits that reflect the swapped status in order to remove any possibility of influence by the unused bits of PORTA's register.

The SWAPF command is especially useful if the values of the two nibbles are required separately as values of up to 15 (00001111). A good example of its use will be seen in Tutorial 21. It is illustrated now because of its programming similarity to TK3TUT12 and TK3TUT14.

The F suffix can be used with SWAPF instead of W, as with other files discussed. There is no command which allows nibbles to be swapped once the byte is in W. If a byte within W needs swapping, it must be put out to a file, and then the SWAPF (FILENAME),W command given to bring it back into W.

Let's look now at another command which uses a similar demonstration routine to TK3TUT12 and TK3TUT13. Run TK3TUT14.HEX. Once more, experiment with pressing any combination of switches SW0 to SW3 while watching PORTB's l.e.d.s.

LISTING 14 – PROGRAM TK3TUT14

```
LOOP    COMF PORTA,W
        ANDLW B'00001111'
        MOVWF PORTB
        GOTO LOOP
```

You will now discover that instead of l.e.d.s being turned on when a switch is pressed, they are turned off, and vice versa. This is due to the command COMF, which automatically inverts each bit of a byte, 1s becoming 0s, 0s becoming 1s, i.e. it performs a task known as *complementing*, hence COMF, which means COMplement File.

There are several uses for this command, one of which is the situation when all the switches are biased to the +5V line instead of 0V. In that instance, and using the switch testing techniques shown earlier, pressing the switches would produce the wrong bit levels for the commands shown: switches would need to be held pressed for off, releasing them for on. Not an easy thing to do with push-switches!

Swap over the 0V and +5V connections to pins CP21 and CP20 so that PORTA pins RA0 to RA3 are biassed to +VE, going low when switches SW0 to SW3 are pressed. Run the program again. You will find that the l.e.d.s respond as they did for TK3TUT12. Now run TK3TUT12 again and confirm that the l.e.d. results are the inverse of that previously seen with it.

Another use for COMF is in subtraction. This is a concept for experienced programmers and will not be demonstrated here. In a nutshell, the use of COMF allows addition to be used instead of subtraction while still achieving the desired objective. This technique can be easier in some instances than using the available subtraction commands.

The F suffix can be used with COMF instead of W, as with other files discussed. There is no command which allows the inversion of a byte once it is in W. If a byte within W needs inversion, it must be put out to a file, and then the COMF (FILENAME),W command given to bring it back into W.

EXERCISE 11

With these exercises, reconnect the +5V connection to CP21 and 0V to CP20.

11.1 If SWAPF was not available as a command, how would you write a routine which produced the same result (would RLF or RRF be suitable commands)?

11.2 Rewrite TK3TUT13 and TK3TUT14, putting the contents of W out to a file of any name (which you must equate at the beginning of the program), performing another COMF or SWAPF action, and then bringing it back into W for output to PORTB. Can PORTB be used as the temporary file store in these rewrites?

11.3. Write a routine that allows the nibbles of a byte to be put into separate files and each having a value no greater than H'0F' (decimal 15); there are several ways of doing it.

TUTORIAL 12

CONCEPTS EXAMINED

Generating an output frequency in response to a switch press

The use of two port bits set to different input/output modes

Command NOP

CONNECTIONS NEEDED

All Port B to all l.e.d.s.

Port A RA0-RA3 to switches SW0-SW3 (via CP19-CP16)

Port A RA4 connected as in Fig.3 (audio connection)

CP21 to +5V OUT

CP20 to 0V OUT

1μF capacitor C7 omitted (from hereon)

Preset VR1 set to minimum resistance (fully clockwise)

LISTING 15 – PROGRAM TK3TUT15

```
SOUND   MOVLW 80
        MOVWF NOTE
        MOVWF FREQ
GETKEY  BTFSS PORTA,0
        GOTO GETKEY
        DECFSZ NOTE,F
        GOTO GETKEY
        MOVF FREQ,W
        MOVWF NOTE
        MOVLW B'00010000'
        ADDWF PORTA,F
        GOTO GETKEY
```

So far we have been outputting data to l.e.d.s, and at a comparatively slow rate. We have also been using one port as a switch input and the other port as the output. Here we examine how the same port can be used simultaneously for input and output via different bits. In doing so, we use sound as the medium by which we indicate the status of a switch, generating an audible frequency when it is pressed.

The 1μF capacitor used up till now for C7 should be omitted from hereon.

Connect a 330Ω resistor between RA4 and the +5V connection at CP21. Pin RA4 is an open-collector pin and this resistor biasses it so that an output can be generated on it. Connect a 1μF capacitor with its positive lead on the junction of RA4 and the resistor. Connect the negative lead of the capacitor to the signal terminal of a jack socket that suits your personal (high-impedance) headphones (see Fig.3).

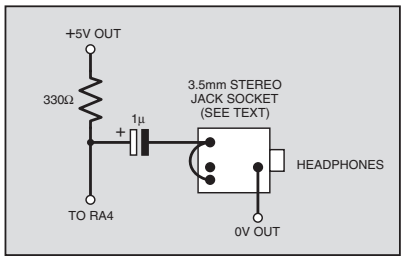

Fig.3. Audio output connections.

Do not connect a loudspeaker directly to this circuit as there is insufficient power to drive it.

Load TK3TUT15.HEX and press switch SW0 on and off. A frequency tone will be heard when the switch is pressed.

In the initialising statements at the head of the full TK3TUT15.ASM program, PORTA has been set with bits 0 to 3 as inputs and bit 4 as an output (MOVLW B'00001111', MOVWF TRISA). You should now recognise all the commands given in the heart of the program shown. Only a general commentary on what happens is now given.

On entry into the routine headed SOUND, a value of 80 is loaded into the files named NOTE and FREQ. The value is arbitrary as far as this demonstration is concerned. You may choose any other from 1 up to 255; the lower the value, the higher the frequency generated.

PORTA's status is monitored at GETKEY and the setting (logic 1) of PORTA bit 0 by switch SW0 is being looked for. Switch testing is repeated until SW0 is pressed, setting RA0 high. When that occurs, file NOTE is decremented and its zero status tested. If it is not yet zero, the routine jumps back to the switch test.

When the switch has been pressed for long enough (mere thousandths of a second), NOTE will eventually reach zero, at which point the command MOVF FREQ,W is reached, followed by the fixed value of FREQ being reset into NOTE. Next, the value in PORTA has 16 (binary 00010000) added to it to increment the count at bit 4 (so alternating the bit between 0 and 1), and then there is a jump back to further switch testing.

For as long as switch SW0 is pressed, PORTA bit 4 will be periodically incremented. The speed at which the routine runs causes this bit to change at the audio frequency rate to which you are listening. If you adjust the rate setting preset, VR1, you will hear the change in the resulting frequency.

In a real-life situation, of course, the operating frequency of the system would normally be fixed. One frequency correction choice then is to change the value of FREQ.

There is, though, another factor that will affect the resulting audio frequency: the number of commands within the controlling loops. To illustrate the point, let's change the number of commands involved.

You may think that to add more commands would be difficult, what would they do which would not interfere in the completion of the loop? Well, there are several options, such as repeating some of the existing commands, MOVF FREQ,W for example, or MOVWF NOTE. Neither of these commands would actually change anything, except for the rate of operation. However, a tailor-made command is already available in the PIC's command codes which is intended for use where minor delay tactics are needed, command NOP.

COMMAND NOP

Command NOP simply stands for No OPeration. Responding to this command takes the PIC just as long as responding to any other single-cycle command but its response is to just do nothing!

This command, then, can be used here to slow down the resulting note frequency. Insert it immediately before DECFSZ NOTE,F. When running the amended program you will notice that a change in the output frequency has occurred.

EXERCISE 12

12.1. Experiment with different values for FREQ. What happens if you set FREQ to zero – does it stop a note being generated? Explain the result.

12.2. Experiment with more than one NOP command in the loop.

12.3. At which other places can you alternatively insert NOP, and is the frequency change still noticeable?

12.4. Are there any places where you cannot use NOP?

12.5. When the audio frequency is not being generated there is the likelihood that RA4 will be set low, so sinking current through the 330Ω resistor. Can the program be modified so that this cannot occur.

TUTORIAL 13

CONCEPTS EXAMINED

Command CALL
Command RETURN
Command RETLW

CONNECTIONS NEEDED

All Port B to all l.e.d.s.
Port A RA0-RA3 to switches SW0-SW3 (via CP19-CP16)
CP21 to +5V OUT
CP20 to 0V OUT
Preset VR1 set to minimum resistance (fully clockwise)

LISTING 16 – PROGRAM TK3TUT16

```
LOOP    CALL PROG1
        MOVWF PORTB
        GOTO LOOP
PROG1   MOVF PORTA,W
        RETURN
```

Before looking further into sound generation, there are several commands that we should examine. Three of those are associated with calling sub-routines: CALL, RETURN and RETLW.

Load TK3TUT16.HEX and experiment with pressing different combinations of switches SW0 to SW3 while observing PORTB's l.e.d.s.

COMMANDS CALL, RETURN AND RETLW

Programs can be written as a series of sub-routines which can be reached in one of three ways, directly by default (without being told to go there), via a GOTO command, or by a CALL command. (Routing following automatic detection of an interrupt event is another matter and is discussed later.)

We have shown several examples of the first two. Program TK3TUT4 (Tutorial 4) uses them both: the sub-routine LOOP1 is entered directly following the initialisation routine. LOOP2 is also entered directly from the end of LOOP1. Both LOOP1 and LOOP2 are then further accessed by GOTO commands.

However, a CALL command can be used if one routine needs to make use of another and then once that has been completed, for the program to jump back to continue from the command that follows the call. The use of sub-routines allows the same routine to be accessed from many other areas within the overall program, so saving on program space.

A second command always has to be used before the program returns to the calling origin. That command takes one of two forms, RETURN (which is an obvious command – return to where you came from) or RETLW (RETurn to where you came from with a Literal value held in W). There is a third return command, RETFIE, which we shall meet later in connection with interrupts.

A GOTO command can never be used to end a sub-routine call – the PIC will continue to expect a return command and, if repeated calls to a sub-routine are made without a RETURN or RETLW command, it will become confused and unpredictable results could occur. For example, the following is "illegal":

```
PROG1  CALL PROG2
       GOTO PROG1
PROG2  GOTO PROG1
```

This is "legal", though:

```
PROG1  CALL PROG2
       GOTO PROG1
PROG2  RETURN
```

When the program returns from a CALL following a RETURN command, the contents of W are those which were put there by the last command which used W. Consequently, you can perform a complex sub-routine, end up with an answer in W and, using the RETURN command, return to the main program with that result still retained in W.

Command RETLW, though, returns to the main program with W holding the value which RETLW has acquired as part of that command. A literal value is always specified as part of the RETLW command, e.g. RETLW 127 or RETLW 0. That value replaces any other value within W and is the one which is held in W on the return to the calling point. The value may be expressed in decimal, hexadecimal, binary or as a "named" value equated during program initialisation.

To explain Listing 16 then, at LOOP the sub-routine at PROG1 is CALLed from where the value held in PORTA is moved into W. A return is made to the loop where the next command to be performed is MOVWF PORTB, after which the GOTO LOOP command again takes us back to CALL PROG1 again.

It is important to be aware that PICs have a limit to the number of calls that can be nested (calls being made from within calls). This is due to the PIC's Stack (the area that monitors the return addresses when calls are completed) being limited to only eight address values. If the Stack receives more than eight addresses it will over-write the earlier ones, causing a program crash.

There is no way to read or write to the Stack or to determine how full it is. It is therefore imperative that if you are using nested calls then you must keep very careful track of how many you are using. In such cases consider whether you could achieve the same result by using GOTO commands for some of the calls, or by returning to the previous calling routine before making the next call.

Run program TK3TUT16 and confirm that l.e.d.s LD0 to LD3 respond as expected to the pressing of the four switches.

EXERCISE 13

13.1. Rearrange TK3TUT16 so that reading PORTA is in the main loop and outputting data to PORTB is in the called routine.

13.2. Try adding other commands in the subroutine, such as AND or ADD.

13.3. Use RETLW as the final statement in the subroutine, using any literal value of your choice, verifying its operation!

TUTORIAL 14

CONCEPTS EXAMINED

Tables
Register PCL (again)
Register PCLATH

CONNECTIONS NEEDED

All Port B to all l.e.d.s.
Port A RA0-RA3 to switches SW0-SW3 (via CP19-CP16)
CP21 to +5V OUT
CP20 to 0V OUT
Preset VR1 set to minimum resistance (fully clockwise)

LISTING 17 – PROGRAM TK3TUT17

```
        BANK0
        GOTO LOOP
TABLE   ANDLW B'00001111'     ; AND W with 15
        ADDWF PCL,F           ; ADD to PCL
        RETLW 255             ; 0  11111111
        RETLW 1               ; 1  00000001
        RETLW '5'             ; 2  00110101
        RETLW 0               ; 3  00000000
        RETLW 31              ; 4  00011111
        RETLW 193             ; 5  11000001
        GOTO OTHER            ; 6  00100000
        RETURN                ; 7  00000111
        RETLW B'10101010'     ; 8  10101010
        RETLW H'C7'           ; 9  11000111
        RETLW 'A'             ; 10 01000001
        RETLW 65              ; 11 01000001
        RETLW 'B'             ; 12 01000010
        RETLW 'x'             ; 13 01111000
        GOTO OTHER1           ; 14 10001110
                              ; or 10011110
        MOVF STORE,W          ; 15 00000000
        RETURN
LOOP    MOVF PORTA,W
        CALL TABLE
        MOVWF PORTB
        GOTO LOOP
OTHER   RETLW STORE
OTHER1  MOVLW 128
        ADDWF PORTA,W
        RETURN
```

The use of look-up tables, whose tabulated commands or values are determined by a value set elsewhere in a program, is of enormous benefit. Tables depend on the use of the Program Counter (PCL – discussed in Tutorial 4) and the commands CALL, RETLW, RETURN and GOTO. They can be used with other calls within them, but this usually requires making additional commands prior to accessing the table. When a table is accessed, the value already held in W is added to PCL and causes the program to jump forward by the same number of program commands as are in W. The command at the jump address is then performed.

Load TK3TUT17.HEX, run it and experiment by pushing switches SW0 to SW3 in any combination while observing the l.e.d.s. on PORTB. The l.e.d.s should come on according to the binary value shown in the comments column of Listing 17, i.e. all l.e.d.s will be on if no switch is pressed.

In TK3TUT17, the instruction BANK0, although individually stated in the extract shown here, follows the initialisation in the normal way. After initialisation, and before any tables are encountered, the command GOTO LOOP bypasses the table commands. Failure to bypass them would cause confusion to the PIC.

At the first command of LOOP, switch data from PORTA is brought into W. The CALL TABLE command then routes the program to the first command within the table, ANDLW B'00001111' (decimal 15).

The AND command is essential here to limit the possible value which can be added to the Program Counter (PCL). Although only the four switches SW0 to SW3 are in use, in another situation another switch might be connected to pin RA4, and so the binary value at PORTA could be greater than 15 (all five switches on = 11111 binary = 31 decimal) and we also know that the number of "routing" commands within the table is 16 (0 to 15). If the table were to be given a value greater than 15, the additive PCL address jump would cause the program to jump beyond the boundary permitted, with unpredictable results. The ANDing could, alternatively, have been done immediately prior to CALL TABLE.

OMITTING THE AND COMMAND

There are circumstances when the AND statement is not needed. For example, if it is known that the value present in W on the call can never be greater than five, AND would not be needed and the table could be limited to six jump options only (remember that 0 counts as a jump value). However, if in doubt about the maximum value that could be in W, always use a value limiter of some sort (techniques other than AND can be used).

This limiting is especially necessary when a program is being developed since errors in other regions of the program could result in an excessive W value, resulting in a system "crash". When consequential crashes of this type occur, it can be difficult sometimes to establish the primary cause of the problem which is elsewhere.

At the command ADDWF PCL,F the ANDed value remaining in W is added to the Program Counter and the command within the table which corresponds to the new address is performed. For clarity, W's entry value is shown alongside each of the 16 table jumps.

If the W value is 0, then the command performed within the table is the first one (0), RETLW 255. As instructed, the program now returns to the calling point with 255 in W. If the value added to PCL is 5, the command performed is RETLW 193. In all instances of the RETLW command within the table, the stated literal value is copied into W and the return is made. You will see that, as with other xxxLW commands, the literal value can be expressed in decimal, hexadecimal, binary or equated name values.

What you have not encountered yet is the use of characters in single quotes. Any standard ASCII character from the full 0 to 255 set can be entered in this way, numbers, upper or lower case letters, symbols, etc.

During assembly, any character within the quotes is translated into its ASCII value and it is that value which is returned in W. (In reality, a lot of the ASCII codes will not be available on your keyboard.) Note that only the "apostrophe" type of quote is permitted ('), that normally residing on your keyboard between the semicolon (;) and the hash symbol (#). The double-quote symbol (") is not permitted, nor is the "left-hand" single quote (`) found on many keyboards (to the left of numeral 1 and the exclamation mark).

Four examples of "quoted" characters are shown in the table. Quoted '5' will be translated as ASCII 53 (not as the value 5); 'A' and 'B' will become ASCII 65 and 66 respectively; lower case 'x' will be returned as ASCII 120. You will find this conversion technique invaluable when compiling tables of messages for output to an alphanumeric l.c.d. (Tutorial 22).

The simple command RETURN at jump 7 will cause the current value already within W to be returned; i.e. the value on the switches after it has been ANDed with 15. It may not be immediately clear what this action would achieve, but an example is given in Tutorial 15.

TABLED GOTO

There are two examples of a tabled GOTO command in Listing 17, at jumps 6 and 14. These cause the program to jump to the sub-routines named, OTHER and OTHER1. At OTHER, the command MOVLW STORE is executed, after which the program returns to the calling program (not back into the table) with the equated address value of STORE (see full ASM listing).

The routine at OTHER1 shows how a table jump can go to a routine in which more than one action can be performed, in this case adding 128 to the value at PORTA, then returning as usual. Any action can be performed here, on any file, for any purpose, and there is no limit to the number of commands performed before the final RETURN (within the program space available, of course).

The command at table jump 15 is interesting. It looks as though a command other than GOTO, RETURN or RETLW is being performed. However, this jump is the last in the table and so it is perfectly legitimate to perform any other action(s) here since the program will automatically follow them through without interfering with the normal table action.

Here the simple action of getting the value held in STORE is performed, immediately followed by a RETURN. Note that the value returned from jump 15 may not necessarily be zero as shown, since STORE has not been given any value when the program is initialised and so could take any random value between 0 and 255.

What would cause table difficulties is if the command at a mid-table jump did not allow an immediate exit from the table. For example, consider the following mid-table jump commands:

```
RETLW 0          ; 3
MOVF STORE,W     ; 4
RETLW 193        ; 5
```

Jump 3 would be OK, so would jump 5. Jump 4, though, would perform MOVF STORE,W (bringing the value within STORE into W), but the exit route for that command is via the address of jump 5, which is RETLW 193, immediately replacing the value acquired in jump 4 with the value 193. Not very helpful!

Mind you, the commands GOTO or RETURN could be at jump 5, which would be fine for jump 4, but what of the result of actually jumping to jump 5, would you necessarily want to just RETURN or GOTO?

One could, perhaps, envisage a table consisting only of INCF STORE,F commands, for example, in which the number of increments generated would be the equivalent of the entry point value of W. But the use of a loop or an addition would, though, probably be more appropriate to that requirement.

It is legitimate to GOTO a table, or arrive at it from the end of another routine, but in this case it may be necessary to only exit the table by GOTO commands. Unless you are already in the middle of a call, "return" commands will cause a program crash.

Advanced use of Tabled GOTOs is discussed by Malcolm Wiles in his feature *PIC Macros and Computed GOTOs* of *EPE* Jan '03 – this is on the PIC Resources CD-ROM.

TABLE SPAN

There is a significant restriction on tables which must not be overlooked. Because of the way in which the Program Counter handles the calls to and from tables, all of the tabulated data must be contained within the first 256 addresses of the program (0 to 255). Not a single jump address must fall outside this block (except as discussed in a moment).

When writing software, it can sometimes be difficult, depending on program structure, to ascertain from the code editing program (word-processing software) whether or not the tables overlap beyond the block. If this is the case, come out of the WP package and assemble the code. Don't send it to the PIC, but come back into the WP and examine the .LST file that has been generated for the program as it now stands. Look at the address numbers (in the third column as you saw earlier in Listing 3A) and see if any part of the table(s) occurs beyond the H'00FF' hex address (decimal 255). Any overlap beyond (even H'0100' – 256 decimal) is unacceptable.

PCLATH

Advanced programmers do have a way round the table block limit should they need to find one. It is through the use of the PCLATH register which allows additional 256 byte blocks to be used elsewhere in the program. This command will, of course, be useful if the total number of tabulated items is greater than 256. Being an advanced programmer's command, we shall not illustrate PCLATH here. Interested readers are referred to John Waller's *Using the PIC's PCLATH Command* in *EPE* July '02 – again it is on the PIC Resources CD-ROM.

With both the "normal" and PCLATH modified table areas there is no limit to the number of tables within them, and the calling routines can be anywhere within the program, start, middle or end. It is perfectly legitimate to have sub-routines placed between different tables, but remember that their length also consumes part of the 256 byte block.

EXERCISE 14

14.1. Write a routine that calls a table which multiplies a binary number by seven. Use the switches as the source of that number (pressing more than one switch as necessary) and restrict it to between 0 and 7, showing the results on PORTB's l.e.d.s.

14.2. Create a table to convert the binary numbers generated by the switches (multiple pressing again) to a BCD (binary coded decimal) format; tens of units in the left four l.e.d.s, units in right four l.e.d.s. (If you are not familiar with BCD, think about what it might mean and how it might be shown on l.e.d.s. The use of BCD formats is discussed in Tutorial 19.)

TUTORIAL 15

CONCEPTS EXAMINED

Using four switches to create four different notes

Use of a table to selectively route program flow

CONNECTIONS NEEDED

All Port B to all l.e.d.s.

Port A RA0-RA3 to switches SW0-SW3 (via CP19-CP16)

Port A RA4 connected as in Fig.3 (audio connection)

CP21 to +5V OUT

CP20 to 0V OUT

Preset VR1 set to minimum resistance (fully clockwise)

The program in Listing 18 allows any one of four notes to played by the switches on PORTA RA0 to RA3 (SW0 to SW3). As with Tutorial 12, the audio output is on RA4. Reconnect your audio monitor, load TK3TUT18.HEX and press some switches. You will immediately notice that the "note" frequencies belong to no musical scale known to man. There is nothing we propose to do about that, we are interested in more mundane matters!

The object of this program is to show the use of a table and several sub-routines which allow four notes to be played (singly) depending on the switch presses. Multiple pressing of switches is ignored.

To conserve page space only one note routine is shown. The others are identical except that they process different notes and PLAY4 omits the GOTO OUTPUT command since OUTPUT immediately follows its final command. You will see the now-familiar commands in the GETKEY and PLAY1 routines. The table should seem recognisable as well.

As in Listing 17, when the program first starts, the table is bypassed and the first main command is at PRESET. Here the frequency values for the four notes are set up as NOTE and FREQ variables.

Switches are monitored as before and calls made to the table. There, routing to different notes occurs only if individual switches are pressed (jumps 1, 2, 4, 8). Any other switch setting, including none, results in a return to the calling point.

When the selected note routine has been processed, a jump to OUTPUT occurs from where the output pin RA4 is toggled, causing a note to be heard. A RETURN command follows, returning the program to the calling point.

Even from this cut-down version of the program, it is apparent that a lot of commands are involved and that many of them are similar (PRESET) or even identical (PLAY by four).

You will also see that only five calls to the table achieve useful results. The others are wasted but have to be included because four switches can generate 16 permutations of settings. You can't just say to the musician "never press more than one key at once", you have to allow for human fallibility. If an error can be made by the program user, it will at some time *be* made – Murphy's Law. Programmers must always think about what *might* happen and write the software accordingly (making it "user-friendly" is another way of putting it!).

The programmer must usually also think about program speed and program compactness. Sometimes they can both achieve the same result, but not always. However, for the sake of discussing program options available, in a moment we'll look at how TK3TUT18.ASM could be written in another way. First an exercise for you:

EXERCISE 15

15.1. Try to change the frequency values in TK3TUT18.ASM to produce notes that are somewhat more harmonically related! What problems do you come up against?

TUTORIAL 16

CONCEPTS EXAMINED

Indirect addressing

Using unnamed file locations

Register FSR

Register INDF

LISTING 18 – PROGRAM TK3TUT18

```
TABLE   ANDLW B'00001111'
        ADDWF PCL,F
        RETURN          ; 0
        GOTO PLAY1      ; 1
        GOTO PLAY2      ; 2
        RETURN          ; 3
        GOTO PLAY3      ; 4
        RETURN          ; 5
        RETURN          ; 6
        RETURN          ; 7
        GOTO PLAY4      ; 8
        RETURN          ; 9
        RETURN          ; 10
        RETURN          ; 11
        RETURN          ; 12
        RETURN          ; 13
        RETURN          ; 14
        RETURN          ; 15

PRESET  MOVLW 80
        MOVWF NOTE1
        MOVWF FREQ1
        MOVLW 110
        MOVWF NOTE2
        MOVWF FREQ2
        MOVLW 140
        MOVWF NOTE3
        MOVWF FREQ3
        MOVLW 160
        MOVWF NOTE4
        MOVWF FREQ4
GETKEY  MOVF PORTA,W
        CALL TABLE
        GOTO GETKEY
PLAY1   DECFSZ NOTE1,F
        RETURN
        MOVF FREQ1,W
        MOVWF NOTE1
        GOTO OUTPUT
  (PLAY2 to PLAY4 are similar to PLAY1)
OUTPUT  MOVLW B'00010000'
        ADDWF PORTA,F
        RETURN
```

LISTING 19 – PROGRAM TK3TUT19

```
TABLE   ANDLW B'00000011'
        ADDWF PCL,F
        RETLW 10
        RETLW 20
        RETLW 40
        RETLW 80

SETUP   MOVLW 4
        MOVWF LOOPA
        CLRF COUNT
        MOVLW NOTE1
        MOVWF FSR
SETUP1  MOVF COUNT,W
        CALL TABLE
        MOVWF INDF
        INCF FSR,F
        INCF COUNT,F
        DECFSZ LOOPA,F
        GOTO SETUP1
GETKEY  MOVF PORTA,W
        ANDLW B'00001111'
        MOVWF STORE
        MOVLW 4
        MOVWF LOOPA
ROTATE  BTFSC STORE,3
        GOTO PLAY
        BCF STATUS,C
        RLF STORE,F
        DECFSZ LOOPA,F
        GOTO ROTATE
        GOTO GETKEY
PLAY    DECF LOOPA,W
        ADDLW NOTE1
        MOVWF FSR
        DECFSZ INDF,F
        GOTO GETKEY
        DECF LOOPA,W
        CALL TABLE
        MOVWF INDF
OUTPUT  MOVLW B'00010000'
        ADDWF PORTA,F
        GOTO GETKEY
```

CONNECTIONS NEEDED

All Port B to all l.e.d.s.

Port A RA0-RA3 to switches SW0-SW3 (via CP19-CP16)

Port A RA4 connected as in Fig.3 (audio connection)

CP21 to +5V OUT

CP20 to 0V OUT

Preset VR1 set to minimum resistance (fully clockwise)

Time now to examine a concept that allows us to access generalised routines which can manipulate file values without actually specifying the file names within them. This concept is called "Indirect Addressing". It also has profound implications for the ability to minimise the number of sub-routines required by a program. Program TK3TUT19, which uses the technique, will then be discussed and demonstrated.

Indirect Addressing allows the use of generalised routines which do not apply to any specific files. The file which the routine accesses is specified prior to entry into the routine and can be changed at will to suit different aspects of the program.

COMMANDS FSR AND INDF

The two key commands (or, rather, "file registers") in Indirect Addressing are FSR (File Special Register) and INDF (INDirect File). The idea of Indirect Addressing is that you place the address of the file that you wish to access in file FSR. Commands to access the specified file address are then made via file INDF.

Not only does this facility allow the same routine to be applied to different calling routines, it also allows a loop to access a sequence of files without having to specify their individual addresses other than that for one of them in the sequence.

In the following example, assume that we have a sequence of files between addresses H'20' and H'2F' (16 files). Let's call the first file FILE0. Its address will have been equated at the head of the program in the usual way. However, provided we assume the next three addresses to be reserved for 15 files which are consecutive to FILE0, we do not have to give them names unless we actually need to use the names in the body of the program. Even then the names could be anything we like; they do not have to called FILE1, FILE2 etc., unless we wish to.

Suppose, for example, we wished to clear all 16 of these files prior to another routine and that we shall do it in ascending order using a loop. Prior to entering the loop we get the address of the first file, in this case FILE0, copy it into FSR and reset the loop counter, let's call it LOOPA:

```
        MOVLW FILE0
        MOVWF FSR
        CLRF LOOPA
```

Now all we need to do is use the following simple routine:

```
RESET   CLRF INDF,F
        INCF FSR,F
        INCF LOOPA,F
        BTFSS LOOPA,4
        GOTO RESET
```

Command CLRF INDF,F clears the file whose address is held in FSR. Next, INCF FSR,F increments the value held by FSR, in other words FSR is incremented to point to the next file we wish to clear (FILE0 in the first instance of the loop, FILE1 in the next). Next, we increment the loop counter, INCF LOOPA,F, and test its bit 4 (BTFSS LOOPA,4) to see if a count value of 16 (00010000) has been reached (remember we started at 0). If the count is not yet 16, the loop is repeated, GOTO RESET. If the count equals 16, the next command after GOTO RESET is performed, whatever that might be in a full program. Another way of doing it (and there are several ways) is:

```
        MOVLW FILE0
        MOVWF FSR
        MOVLW 16
        MOVWF LOOPA
RESET   CLRF INDF,F
        INCF FSR,F
        DECFSZ LOOPA,F
        GOTO RESET
```

You can also use similar constructions to access a sequence of table values (from anywhere within that table) and add them to the values within a sequence of indirectly addressed files, keeping the maximum resulting addition to less than the maximum number of temporary registers that the PIC provides.

In the following example (nothing directly to do with TK3TUT19), the first address required in the table is at jump 3. This value is first placed into COUNT (MOVLW 3, MOVWF COUNT). We want to start adding the acquired table value to the file starting six bytes beyond FILE0 so the value of 6 is then added to the address of FILE0 and the result placed into FSR (MOVLW 6, ADDLW FILE0, MOVWF FSR). We also want to perform the action five times, once for each note, so a loop (LOOPA) is set up with the initial value of 5 (MOVLW 5, MOVWF LOOPA).

The real action then starts at label GETVAL. The current value held in COUNT is copied into W (MOVF COUNT,W). The table is called (CALL TABLE) and value held in the table at the location indicated by the value in W is retrieved from the table, being automatically placed into W. The value from the table now in W is then added to the value in the file held via INDF and pointed to by FSR, and the result is stored back into the same file (ADDWF INDF,F). File FSR is now incremented (INCF FSR,W), so incrementing the address of the file held via INDF. Count is incremented (INCF COUNT,F), and LOOPA is decremented. If LOOPA is not yet zero the process repeats.

```
        MOVLW 3
        MOVWF COUNT
        MOVLW 6
        ADDLW FILE0
        MOVWF FSR
        MOVLW 5
        MOVWF LOOPA
GETVAL  MOVF COUNT,W
        CALL TABLE
        ADDWF INDF,F
        INCF FSR,F
        INCF COUNT,F
        DECFSZ LOOPA,F
        GOTO GETVAL
```

INDIRECT ADDRESSING DEMONSTRATED

In the following worked example, part of whose program is shown in Listing 19, we demonstrate how Indirect Addressing allows generalised file accessing routines to be used, how a table can help in that process, and how it helps code to be compacted to achieve more actions within the space available. Because only four switches are available, the program is limited to four notes, but if more switches were to be added somehow, the process could readily be extended to suit.

With your audio monitor still connected, load TK3TUT19.HEX and play with the four push-switches on PORTA (SW0 to SW3). You will find that all four switches produce "notes", but not musically tuned, though! The technique used is, in effect, the same as that demonstrated in TK3TUT18. There are, though, some notable (no pun!) differences:

First, if you look at the full listing on your disk, you will see that in the initialisation, we have only equated NOTE1 and there is no mention of FREQ1 etc. Yet, we are actually using four files to behave as NOTE1 to NOTE4 and we use a table instead of FREQ1 to FREQ4.

What we have done (as discussed a moment ago) is to consider a block of consecutive file addresses to be allocated to NOTE1/NOTE4, starting at H'20'. To remind us at some future time, there is a comment alongside NOTE1 to this effect in the full disk listing. The next address which we specify cannot, therefore, occur until the fifth byte later, at H'24', where LOOPA is equated. Any consecutive block of four bytes could have been used.

As seen in the full program and the extract in Listing 19, a table has four values in it and an AND command limits the jump span from zero to three. The values shown are the tuning values which will be accessed periodically throughout the program while it is running.

INDIRECTION

Routines SETUP and SETUP1 make use of the indirect addressing facility to set the initial (FREQ) values into the four notes. Next comes routine GETKEY in which the status of the four switches is obtained in the usual way. There are 16 possible combinations of the switches and we only want four of them, those for any single switches being pressed.

We could, of course, not use a table but simply test each bit of PORTA in turn and use GOTO statements to obtain data about which note should be played and which note reset value is needed. Instead, though, for the sake of demonstration, a different technique is used, converting the 4-bit PORTA value to a 2-bit value, covering four possible combinations rather than 16.

PORTA's value is ANDed with B'00001111' and copied into STORE, and a loop set for a maximum of four operations. Up to four rotate left (RLF) actions can then be called in routine ROTATE, and the value of STORE bit 3 tested. Each bit of STORE corresponds to a separate switch, so the rotation allows all four switches to be tested. If a 1 is found during the rotation, the value of the loop

corresponds to the switch in question and a jump is made to the play routine. If a zero is found, then no switches are pressed and no note play action occurs.

In the PLAY routine, the loop value (LOOPA) is decremented while being moved into W (the loop value will be between 4 and 1 but for program ease we need a value between 3 and 0). The value of W is added to the address of NOTE1 and the answer is put into the indirect address register FSR. The note now pointed to by FSR is decremented via INDF and if the result is not zero, a return to GETKEY is made.

A zero result causes the value of LOOPA to again be decremented into W and then the table is called, returning with the reset value for the note in use, which is put into it via INDF. As we have seen before, the output value at PORTA RA4 is then incremented and a jump back to GETKEY occurs.

Had this whole operation been programmed as separate routines for each note, its length would have been considerably greater, as in the previous example of TK3TUT18; indirect addressing, bit rotation and a table have changed that. (Consider the length that would have been required if we were using eight switches for eight notes – a situation that would have also brought up the problem of a table that was greater than 256 commands!) We shall use indirect addressing again later.

EXERCISE 16

16.1. In Program TK3TUT19, priority has been given to switches in descending order (test bit 3). How would you rewrite to give priority in ascending order?

16.2. If you wanted one of the switches to be ignored, what extra command(s) would be needed, and where? When considering where, think of the number of times the situation has to be checked for between each input of PORTA's value, remembering that each command processed wastes valuable time.

16.3. Is the AND command at the head of the table actually necessary?

16.4. As the program stands, there is one extra file name used than needs to be; which file could be used in two situations?

16.5. Also, with careful thought, parts of the program could be slightly rewritten to save at least seven commands. Can you spot how this could be done? Question all aspects, from initialisation downwards (see also the full listing).

(Whilst the SETUP routine could be heavily rewritten to save four of these commands, in a real programming situation, unless you are short of program space, it is better to concentrate on saving commands in routines that are being called frequently, so significantly increasing the speed of operation. SETUP is only used once, and so has no affect on the loop speed.)

TUTORIAL 17

CONCEPTS EXAMINED

Command XORLW
Command XORWF
Command IORLW
Command IORWF
Tone modulation

LISTING 20 – PROGRAM TK3TUT20

```
GETKEY   MOVF PORTA,W
         ANDLW B'00001111'
         XORLW B'11111010'
         MOVWF PORTB
         GOTO GETKEY
```

CONNECTIONS NEEDED

All Port B to all l.e.d.s.

Port A RA0-RA3 to switches SW0-SW3 (via CP19-CP16)

Port A RA4 connected as in Fig.3 (audio connection)

CP21 to +5V OUT

CP20 to 0V OUT

Preset VR1 set to minimum resistance (fully clockwise)

In a moment, we shall come down to a somewhat simpler audio program, in which we illustrate how two tones can be created, one modulated, the other fixed. Both tones could find use in, for example, a simple intruder alarm. Also to be illustrated is how the combined status of two or more switches on a port can be tested using the XOR (Exclusive-OR) command. This allows us to take one action only if all the specified switches are on simultaneously, otherwise taking another action. First, let's examine the XOR command on its own.

COMMAND XOR

The command XOR checks for "equality" between two numbers. There are two commands, XORLW (XOR Literal with W) and XORWF (XOR W with value in specified File). The latter is followed by the file name, a comma, and the destination (W or F), e.g. XORWF STORE,W and XORWF STORE,F.

Probably you know that in electronics there are XOR gates included in the digital logic chip families, and you will no doubt have read descriptions of truth tables relating to just two inputs of an XOR gate (two bits):

0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

As far as a PIC's XOR function is concerned, the result of XORing two bytes of eight bits is the condition being checked. It is easier here to show the principle by means of switches and l.e.d.s rather than by truth tables. To do this we should really use eight switches on one port and eight l.e.d.s on the other. However, since PORTA has only four switches connected to it, we shall just use a 4-bit number to illustrate the principle, via four l.e.d.s on PORTB.

The basic program we shall use is shown in Listing 20. Run program TK3TUT20.HEX and play with PORTA's switches. You will find that when no switches are pressed, PORTB l.e.d.s LD0 and LD2 are off, and LD1 and LD3 are on, as are LD4 to LD7. When switch SW1 and SW3 are pressed, they turn off their respective l.e.d.s (LD1 and LD3). Switches SW0 and SW2 turn on their l.e.d.s (LD0 and LD2) when pressed.

In this listing, the value on PORTA is input as usual. The next command (ANDLW B'00001111') is necessary to this demonstration since we only want to use the first four bits of PORTA. If PORTA had eight bits that could be used, the AND command would be omitted. The status of each switch is being XORed with the respective bit in the statement XORLW B'11111010'; switch 0 with bit 0, switch 1 with bit 1, etc.

If any bit of PORTA is equal to that of the same bit in the XOR command, the same bit in the W register will be cleared. Thus two zeros will produce a 0, and two 1s will produce a 0. If the bits are dissimilar (1 and 0) the W bit is set (1). The reason that the four lefthand l.e.d.s are on is that bits 4 to 7 from the AND command and bits 4 to 7 from the XOR command have resulted in four non-equalities.

Suppose that the switches produce binary number 0111, the ANDed result in W is 00000111, the sequence of events is:

```
MOVF PORTA,W       answer = xxxx0111
ANDLW B'00001111'  answer = 00000111
XORLW                        11111010
                   answer = 11111101
```

Bits that are equal to their counterparts have their corresponding l.e.d.s turned off, those that are *not* equal have their l.e.d.s turned on. Take another example:

```
MOVF PORTA,W       answer = xxxx0010
ANDLW B'00001111'  answer = 00000010
XORLW                        00000010
                   answer = 00000000
```

Here each bit is equal to its counterpart, therefore all l.e.d.s are turned off, i.e. a zero result has occurred and, importantly, the Zero flag will have been set accordingly. Therefore, we can check for equality by checking the Zero flag following an XOR command. Non-equality clears the flag, equality sets it. Consequently, following an XOR command you simply check STATUS,Z and route accordingly.

LISTING 21 – PROGRAM TK3TUT21

```
GETKEY   MOVF PORTA,W
         ANDLW B'00001111'
         XORLW B'00001010'
         MOVWF PORTB
         BTFSC STATUS,Z
         BSF PORTB,7
         GOTO GETKEY
```

Let's use l.e.d. LD7 to illustrate this, turning it on if equality exists, turning it off if it doesn't. Any bit between 0 and 3 which is equal to the same XOR bit will have its corresponding l.e.d. turned off, otherwise its l.e.d. will be on. Load TK3TUT21.HEX and press PORTA switches SW0 to SW3 to observe this in action. Pressing SW3 and SW1 together causes LB7 to come on. The commands are shown in Listing 21.

COMMAND IOR

Although we shall not meet it until later (Tutorial 21), it is opportune to mention now that there is an "ordinary" OR command available. It is more correctly termed "Inclusive-OR" (as opposed to Exclusive-OR). It has two versions, IORLW

LISTING 22 – PROGRAM TK3TUT22

```
ENTRY     MOVLW 80
          MOVWF NOTE
          MOVWF FREQ
          MOVLW 128
          MOVWF MODLAT
          MOVLW 64
          MOVWF DELAY
GETKEY    MOVF PORTA,W
          ANDLW B'00000011'
          XORLW B'00000011'
          BTFSC STATUS,Z
          GOTO GETKEY
          DECFSZ NOTE,F
          GOTO GETKEY
          MOVF FREQ,W
          BTFSC PORTA,1
          GOTO OUTPUT
          DECFSZ DELAY,F
          GOTO GK2
          BSF DELAY,6
          DECFSZ MODLAT,F
          GOTO GK2
          BSF MODLAT,7
GK2       ADDWF MODLAT,W
OUTPUT    MOVWF NOTE
          MOVLW B'00010000'
          ADDWF PORTA,F
          GOTO GETKEY
```

(Inclusive OR Literal with W) and IORWF (IOR W with value in specified File). The latter is followed by the file name, a comma, and the destination (W or F), e.g. IORWF STORE,W and IORWF STORE,F.

MODULATION

The use of XOR in a practical situation is illustrated in Listing 22. Temporarily swap over the connections to CP20 and CP21 so that RA0 to RA3 are biased normally high, going low when pressed. Reconnect the audio output.

Load TK3TUT22.HEX, press any switches SW0 to SW3, but principally use switches SW0 and SW1 since these are the ones coded to be active.

Listening to the output from PORTA, you will find that switch SW0 controls a static tone and SW1 controls a modulated (ramped) tone. As you will have heard, the tone starts at a low pitch, ascends and then jumps back low again, repeatedly. Adjust VR1 until this fact is more obvious. All other switches are ignored. Look at the program's listing.

As with earlier tone generation examples, a starting value is loaded into NOTE and FREQ, then a modulation starting value is loaded into MODLAT, and a delay value into DELAY, after which the GETKEY loop is entered. Here the switch settings on PORTA are read and ANDed with 00000011 to extract the status of switches SW0 and SW1. The answer is XORed with the same value to check for equality. If neither switch is pressed, no further action is required and the routine jumps back to GETKEY.

We are looking for the situation in which either of the two switches is pressed. We could do it simply by bit testing (indeed, it would be easier!), but part of the aim of this demo is to show a use of XOR. When either switch is pressed, NOTE is decremented and checked for zero and reset as appropriate, as before.

When zero is encountered, if switch SW1 is pressed, the DELAY counter is decremented, if it is zero, DELAY is then reset to 64 (BSF DELAY,6), the value of MODLAT is added to the NOTE reset value and the value of MODLAT itself is then decremented. When MODLAT reaches zero, it is reset to 128 (BSF MODLAT,7). The OUTPUT routine is common to both switch routings.

Note how bit values of MODLAT and DELAY are set to reset these counters to their original values. This works because both values are known to have reached zero.

EXERCISE 17

17.1. Experiment with different settings for FREQ, DELAY and MODLAT

17.2. How would you change the coding to respond to two other switches instead, e.g. SW2 and SW3?

17.3. How would you reverse the ramp to create a rising tone rather than a falling one?

17.4. The addition of a third switch would allow tones to be switched for rising, falling or fixed. Can you write the program for it?

17.5. Can you add another routine which would create a triangular modulation pattern (rising tone, followed by falling, followed by rising, and so on)?

TUTORIAL 18

CONCEPTS EXAMINED

OPTION register
INTCON register
TMR0 register
Command OPTION_REG
Command INTCON

LISTING 23 – PROGRAM TK3TUT23

```
          CLRF PORTA
          CLRF PORTB
          BANK1
          CLRF TRISA
          CLRF TRISB
          MOVLW B'10000000'
          MOVWF OPTION_REG
          BANK0
          CLRF RATE
          MOVLW 8
          MOVWF COUNT
          BCF INTCON,2
MAIN      BTFSS INTCON,2
          GOTO MAIN
          BCF INTCON,2
          MOVLW B'00010000'
          ADDWF PORTB,F
          BTFSS STATUS,C
          GOTO MAIN
          DECFSZ COUNT,F
          GOTO MAIN
          BSF COUNT,3
          INCF RATE,W
          ANDLW 7
          MOVWF RATE
          MOVWF PORTB
          BANK1
          IORLW B'10000000'
          MOVWF OPTION_REG
          BANK0
          GOTO MAIN
```

Command TMR0
Use of internal timer

CONNECTIONS NEEDED

All Port B to all l.e.d.s.
Port A RA0-RA3 to switches SW0-SW3 (via CP19-CP16)
CP20 to +5V OUT
CP21 to 0V OUT
Preset VR1 set to maximum resistance (fully anti-clockwise)

The PIC16F84 has one special register reserved for use as an 8-bit timer, TMR0 (Timer 0). It divides its input frequency by 256 and can be both written to and read from. In most situations, though, it is unlikely that you will need to use the read/write facility, but note that if TMR0 is written to, the timer is inhibited from counting for two clock cycles.

Probably more useful than writing to TMR0 is to use its output as it occurs naturally at the 1:256 division rate, and then to use the prescaler to subdivide that rate as required. The prescaler divides its input pulses by presettable powers of two. There are eight possible division ratios which are set via bits 0, 1 and 2 of the OPTION register. When used with TMR0, the prescaler division ratios are 1:2, 1:4, 1:8, 1:16, 1:32, 1:64, 1:128 and 1:256.

The prescaler can alternatively be allocated for use with the Watchdog Timer (WDT), in which mode each of these ratios is halved (minimum is thus 1:1 and maximum is 1:128) – more on this later.

OPTION NAMING

Note that the OPTION register should not be equated as such since Microchip previously had a command actually named OPTION and use of this term in an ASM file assembled by MPASM causes an error condition. Consequently it is preferable that the register should be equated as OPTION_REG (Microchip's equated term in their INC files). You may still sometimes come across the equated name OPTION, or even OPSHUN instead.

(It should also be noted that bit 7 of the OPTION register controls the PIC's Light-pullups facility and should be set high to turn it off, as in Listing 23 – this facility is discussed separately later.)

We commented earlier that the PIC effectively runs at one quarter of the input clock frequency at pin 16 (OSC1/CLKIN). When TMR0 is used as an internal timer, the pulses it counts also occur at one quarter of the clock frequency. So, if the clock frequency (set by a crystal oscillator, perhaps) is running at 3·2768MHz, TMR0 will count at 819200Hz and its 1:256 roll-over rate will be 3200Hz. This rate is then divided by the ratio set into the prescaler. If we divide by 32, for example, we obtain the convenient rate of 100Hz.

In TMR0 mode, when the prescaler rolls-over to zero, a flag is set in the INTCON register, at bit 2. The setting of this bit can be used as an interrupt (see Part 3) which automatically routes the program to another specified routine, irrespective of which routine is currently being processed, returning to the same point after the interrupt procedure has been finished. The interrupt can also be turned off and INTCON bit 2 read by the

program to establish its status, taking action accordingly.

TIMER SUB-DIVISION

Using the timer and the prescaler, you can specify that some actions will only be performed at specified sub-divided values of the clock frequency. Amongst other things, this allows the PIC to be used as a real-time clock, a function towards which we now progress.

First, let's illustrate the effect of setting different prescaler ratios and, using the l.e.d.s on PORTB, show what happens. Load TK3TUT23.HEX and run it. Set VR1 to full anti-clockwise rotation (slowest rate). In this program we read the status of INTCON bit 2 rather than using the interrupt facility (discussed in Tutorial 27).

Initially, you will see a fairly fast binary count occurring on PORTB's l.e.d.s LD3 to LD7. It is created with the timer "in-circuit" with the prescaler set for a minimum division ratio of 1:2. This is because OPTION_REG bits 0 to 2 are set to 000, a value which is shown on LD0, LD1 and LD2 – all off initially.

This rate of counting continues for eight cycles of 32 increments (incrementing PORTB's count in steps of eight). The ratio is then set at 1:4 (prescaler value 001), and again another eight cycles occur. Similarly, the other ratios are set. The difference in the resulting l.e.d. count rates will be obvious.

Adjust the setting of preset VR1 if the slowness becomes tedious in later ratios. After the eight ratios, the whole cycle restarts from 1:2.

Looking at Listing 23, you will see that the TMR0 rate is set into the OPTION_REG register while in BANK1 mode, along with the port direction registers.

EXERCISE 18

18.1. Study TK3TUT23.ASM, note the comments and see if you understand what is happening at each stage. Note the detection and resetting of the INTCON,2 flag and the need to go via BANK1 when changing the prescaler rate.

TUTORIAL 19

CONCEPT EXAMINED

BCD (Binary Coded Decimal) counting

CONNECTIONS NEEDED

All Port B to all l.e.d.s.

Port A RA0-RA3 to switches SW0-SW3 (via CP19-CP16)

CP20 to +5V OUT

CP21 to 0V OUT

Preset VR1 set to maximum resistance (fully anti-clockwise)

Having established the use of the timer, we now work towards its use as the pulse source for a real-time clock. There are a few bridges to be crossed yet, though. The first is counting in decimal rather than binary, facilitating the eventual output to a 7-segment l.e.d. or a liquid crystal display. We could keep the counted units in one byte, tens in another, hundreds in another, and so on, but, to conserve precious byte space, it is equally possible to use each byte as two 4-bit nibbles, keeping units in bits 0 to 3, and tens in bits 4 to 7. Hundreds units and tens would be treated similarly in a second byte.

LISTING 24 – PROGRAM TK3TUT24

```
MAIN    BTFSS INTCON,2
        GOTO MAIN
        BCF INTCON,2
        INCF COUNT,F
        MOVF COUNT,W
        ADDLW 6
        BTFSS STATUS,DC
        GOTO OUTPUT
        MOVWF COUNT
        ADDLW 96
        BTFSC STATUS,C
        CLRF COUNT
OUTPUT  MOVF COUNT,W
        MOVWF PORTB
        GOTO MAIN
```

For simplicity now, we concentrate on counting up to 99, first considering the use of two bytes. In 8-bit binary, a value of decimal 9 is expressed as 00001001, decimal 10 is 00001010, decimal 16 is 00010000. It is obvious that with decimal values we have no single symbol for a number greater than nine. When a value one greater than nine occurs, what we do is reset the units digit to 0 and add one to the next digit, i.e. ten is written as 10.

While counting in binary coded decimal (BCD), we can do a similar thing. When the byte holding the units reaches ten, we reset that byte to zero and add one to the next byte. In 8-bit BCD and at a count of nine, the two bytes would read 00000000 (tens) and 00001001 (units). At the count of ten, the bytes become 00000001 (tens) and 00000000 (units).

When using two nibbles of an 8-bit byte (instead of the above two bytes), a BCD value of nine reads as 00001001, but a BCD value of ten reads as 00010000. And, for example, a BCD value of 37 reads as 00110111, i.e. the lefthand nibble (MSN – Most Significant Nibble) holds a value of 3 and the righthand nibble (LSN – Least Significant Nibble) holds 7. A value of 99 is expressed as 10011001. For a value of 100, both nibbles are reset to zero (00000000) and if there is a byte for hundreds and tens of hundreds, its righthand nibble (LSN) would be incremented, and so on.

Thus, when counting in BCD, we have to check the four bits of the LSN on their own and see if their value is greater than nine. If it is, that nibble is reset and the MSN incremented. The MSN is then taken on its own as a 4-bit value and checked if it is greater than nine. If so, this nibble is reset and the LSN of the next byte incremented accordingly.

CHECKING FOR EXCESS VALUES

There are (as in many programming matters) several ways of checking the nibbles for excess values, of which we shall describe one: an additive checking routine. We said earlier (Tutorial 7) that there is a Digit Carry (DC) flag which signals if the binary value of the LSN has become greater than 15 following an addition. We can use this fact by adding a number to the LSN which will make the answer greater than 15 if the basic value of the LSN is greater than 9.

The number to be added is 6, e.g. 10 + 6 = 16 with DC flag set; 9 + 6 = 15 with DC flag clear. Therefore, to check if an LSN value is greater than 9, we temporarily add 6 to it and check the DC flag. If the flag is clear, the LSN is left as it is. If the flag is set, we increment the MSN and clear the LSN.

There is a short cut to doing this, taking advantage of the fact that 10 + 6 = 16, being 00010000 in binary. If you look at this answer, the LSN is now zero, while the MSN has been incremented automatically, thus representing decimal 10 in BCD. Thus, when we add 6 to the byte as a whole, if the DC flag is clear, no further action on that byte is needed (or on any subsequent bytes for that matter). If, though, the DC flag is set, we simply replace the existing value in the byte with the value now stored temporarily. These commands do the job:

```
INCF COUNT,F      ; increment file
                    value
MOVLW 6           ; move 6 into W
ADDWF COUNT,W     ; add it to new file
                    value but keep
                    answer in W
BTFSC STATUS,DC   ; is the Digit
                    Carry flag clear?
MOVWF COUNT       ; no, it's set so
                    move W into file,
                    replacing previ-
                    ous value
(next command)
```

The above check is done in respect of LSN, but when the DC flag is set, the resulting action changes the value of the MSN, which then has to be checked to see if it (as a 4-bit nibble) is greater than 9, i.e. is the BCD value of the whole byte now equal to or greater than decimal 100?

Again there is an easy additive technique. If we translate the binary value of BCD 100 (10100000) the decimal answer is 160. If we temporarily add 96 (256 – 160) to the whole byte, we can then check the Carry flag (C) to see if it has been set, which it will be if the binary answer has rolled over beyond 255. As before, if the flag is clear, the byte can remain as is; if the flag is set, we replace the value with the temporary one.

(Note that the DC and Carry flags are unaffected by an INCF or INCFSZ command.)

Here's the extended routine. Note the inverted logic for checking Digit Carry and Carry flags, BTFSS STATUS,DC in the first instance, BTFSC STATUS,C in the second.

```
        INCF COUNT,F
        MOVLW 6
        ADDWF COUNT,W
        BTFSS STATUS,DC
        GOTO ENDADD
        MOVWF COUNT
        MOVLW 96
        ADDWF COUNT,W
        BTFSC STATUS,C
        MOVWF COUNT
ENDADD  (program continues)
```

Let's look at the BCD additive technique in practice, triggering it from the timer

routine. In Listing 24, note the use of CLRF COUNT before OUTPUT at the end. This can be used here since we know that adding 1 to the count is occurring, rather than adding values of 2 or greater. In the latter instance, the resulting temporary answer must be MOVed into COUNT, as in the above examples.

Load TK3TUT24.HEX and observe the count incrementing on the l.e.d.s. The prescaler is now run at a fixed ratio of 1:128. Try adjusting VR1 so that an l.e.d. count rate of one per second (1Hz) occurs. (The tolerance of VR1 and the in-circuit capacitance may not allow you to set the rate quite this slow without also amending the OPTION_REG timing value.)

EXERCISE 19

19.1. Suppose our counting system was not decimal but quinary, i.e. no digit greater than 5, rather than no digit greater than 9. How would you change the additive values in the above examples (you can use decimal, binary or hexadecimal for those!).

19.2. Checking for excess BCD values can be done using an XOR technique which is valid if the count is being incremented rather than added to. Adding to the BCD value cannot be used with XOR since the answer could be to either side of the equality being checked for. Can you write an XORed BCD incrementing program?

TUTORIAL 20

CONCEPTS EXAMINED

Real-time timing at 1/25th second
Counting seconds 0 to 60

LISTING 25 – PROGRAM TK3TUT25

```
        CLRF PORTA
        CLRF PORTB
        BANK1
        CLRF TRISA
        CLRF TRISB
        MOVLW B'10000110'
        MOVWF OPTION_REG
        BANK0
        MOVLW 25
        MOVWF CLKCNT
        CLRF CLKSEC
        BCF INTCON,2

MAIN    BTFSS INTCON,2
        GOTO MAIN
        BCF INTCON,2
        DECFSZ CLKCNT,F
        GOTO MAIN
        MOVLW 25
        MOVWF CLKCNT
        INCF CLKSEC,F
        MOVF CLKSEC,W
        ADDLW 6
        BTFSS STATUS,DC
        GOTO OUTPUT
        MOVWF CLKSEC
        MOVLW B'01100000'
        XORWF CLKSEC,W
        BTFSC STATUS,Z
        CLRF CLKSEC

OUTPUT  MOVF CLKSEC,W
        MOVWF PORTB
        GOTO MAIN
```

CONNECTIONS NEEDED

All Port B to all l.e.d.s.
Port A RA0-RA3 to switches SW0-SW3 (via CP19-CP16)
CP20 to +5V OUT
CP21 to 0V OUT
Crystal oscillator

Moving on from decade counting between 0 and 99, it is an easy step to count in BCD from 0 to 59, accurately simulating the seconds count of a real-time clock. In doing so, though, it can be useful to actually increase the count rate available via the prescaler from 1Hz to 25Hz, 50Hz or even 100Hz. Indeed, if a crystal oscillator running at the convenient rate of 3·2768MHz is used, it is actually easier to work with one of these three rates. This is due to the sub-division values available from a crystal of this frequency which can be used in conjunction with the TMR0. Prescaler division ratios of 1:128, 1:64 or 1:32 respectively produce these rates.

So now go over to crystal control on *TK3*'s p.c.b. Go into *TK3*'s PIC Configura-tion option and select crystal XT instead of the previous RC mode. Leave all other settings as they are. Send the configuration to the PIC. Set *TK3*'s switch S2 to crystal mode. It is assumed that the crystal on your p.c.b. is 3·2768MHz. Crystals having a different frequency may be used but the clock timings shown on your l.c.d. will differ accordingly.

All of the programs you have used so far, with the exception of TK3TUT2, can be run under crystal control. Consequently, if you want to go back and look at some of them again, you do not need to reset the PIC for RC mode.

Load TK3TUT25.HEX and observe PORTB's l.e.d.s. You will see them incrementing at a rate of one per second, and the twin-nibble BCD count will be seen to progressively step from zero to BCD 59 (01011001), then restart again at zero, just as would an ordinary seconds clock and, indeed, it should take one minute for the full cycle to occur.

In this program the prescaler rate has been set for 1:128, providing an INTCON,2 pulse rate of 1/25th of second. A counter, CLKCNT, counts down from 25 in response to the pulses. When it reaches zero, it is reset to 25 and a seconds counter, CLKSEC is incremented in BCD.

Checking for the BCD count becoming ten is performed by the additive (+6) technique we have already shown. However, checking for the count being at BCD 60 is done using the XOR equality testing method (XOR 01100000 = BCD 60). If equality exists, the CLKSEC counter is reset to zero.

EXERCISE 20

20.1 There are three commands associated with the XOR check. What XOR coding would be needed to lose one of them?

TUTORIAL 21

CONCEPTS EXAMINED

Using 7-segment l.e.d. displays
Showing hours, minutes and seconds
Command IORLW (usage)

CONNECTIONS NEEDED

7-segment display as in Fig.6
CP20 to +5V OUT

LISTING 26 – PROGRAM TK3TUT26

```
COMCATHODE
        ADDWF PCL,F
        RETLW B'00111111'       ; 0
        RETLW B'00000110'       ; 1
        RETLW B'01011011'       ; 2
        RETLW B'01001111'       ; 3
        RETLW B'01100110'       ; 4
        RETLW B'01101101'       ; 5
        RETLW B'01111100'       ; 6
        RETLW B'00000111'       ; 7
        RETLW B'01111111'       ; 8
        RETLW B'01100111'       ; 9
        ; common cathode codes

COMANODE
        ADDWF PCL,F
        RETLW B'11000000'       ; 0
        RETLW B'11111001'       ; 1
        RETLW B'10100100'       ; 2
        RETLW B'10110000'       ; 3
        RETLW B'10011001'       ; 4
        RETLW B'10010010'       ; 5
        RETLW B'10000011'       ; 6
        RETLW B'11111000'       ; 7
        RETLW B'10000000'       ; 8
        RETLW B'10011000'       ; 9
        ; common anode codes

MAIN
        BTFSS INTCON,2
        GOTO MAIN
        BCF INTCON,2
        DECFSZ CLKCNT,F
        GOTO MAIN
        MOVLW 25
        MOVWF CLKCNT
        INCF CLKSEC,F
        MOVF CLKSEC,W
        ADDLW 6
        BTFSC STATUS,DC
        CLRF CLKSEC

OUTPUT
        MOVF CLKSEC,W
        ANDLW B'00001111'
        CALL COMCATHODE
        MOVWF PORTB
        GOTO MAIN
```

CP21 to 0V OUT
Crystal oscillator

Obviously it is not feasible to show hours, minutes and seconds by just using BCD formatted values on individual l.e.d.s. We need a display which is more suited to being understood. Such a display could be via alphanumeric liquid crystal displays (l.c.d.s) and a typical routine using them will be shown later on.

Another choice is the use of 7-segment l.e.d. displays, and that is the route we now take. First, though, we must examine how the output from PORTB needs to be coded to drive a single 7-segment common cathode l.e.d. display. We shall then extend the principle to multiplexing four such displays to show a full 24-hour clock.

As Tutorial 21 is the only section to use 7-segment displays, you may prefer not to purchase one at this time, and to just read about using them, for future reference. Don't skip reading this section, though, as other concepts are examined.

Each segment of a 7-segment l.e.d. display has to be controlled by individual PIC

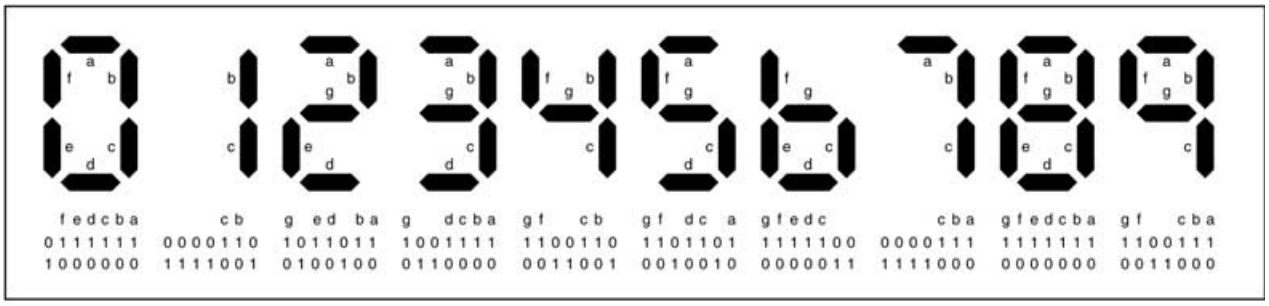

Fig.4. Numerals 0 to 9 on a 7-segment l.e.d. display, plus controlling binary codes for common cathode (middle line) and common anode (bottom line).

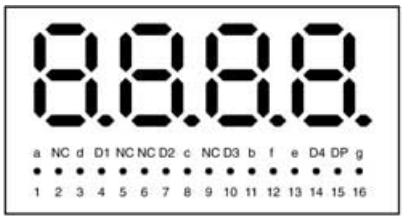

Fig.5. Pinouts for a typical 4-digit multiplexed 7-segment l.e.d. display.

data lines. It does not matter in which order the data lines are connected to the display since the way that they are activated can be set from within the PIC's controlling program. For convenience, here we use PORTB lines RB0 to RB6 connected in their natural order to segments A to G.

In Fig.4 are shown the segments and code letters required to form the ten numerals 0 to 9. Also shown are two lines of binary code. The first one shows the bits which need to be taken high if a common cathode display is used. The second is for a common anode display, each line being taken low to turn on the segment. It is a common cathode display that we use here; its pinouts are shown in Fig.5. Connect it to the p.c.b. as shown in Fig.6, ensuring that the 330Ω resistors do not short between each other. Also connect RA0-RA3 to TR2-TR5 via CP8-CP11. The program keeps transistor TR2 turned on constantly.

Load TK3TUT26.HEX. You will see the individual numerals being shown on the left-hand digit on a cyclic basis from 0 to 9. Using the crystal oscillator selected for the previous Tutorial, the rate of display will be at one unit per second. In other words, it can be regarded as being a seconds counter.

Referring to Listing 26, you will see that the counting routine is very similar to that in Listing 25, but only dealing with units of seconds. Now, though, instead of the count being sent to individual l.e.d.s it is converted in the COMCATHODE table to the required 7-segment code for that numeral when used with a common cathode display. Since we know that the value held in W when the table is called can never be greater than nine, an AND command is not needed with this table.

LISTING 27 – PROGRAM TK3TUT27

```
MAIN      CALL DIGSEL
          BTFSS INTCON,2
          GOTO MAIN
          BCF INTCON,2
          DECFSZ CLKCNT,F
          GOTO MAIN
          MOVLW 25
          MOVWF CLKCNT
          NCF CLKSEC,F
          MOVF CLKSEC,W
          ADDLW 6
          BTFSS STATUS,DC
          GOTO ENDTIM
          MOVWF CLKSEC
          MOVLW B'01100000'
          XORWF CLKSEC,W
          BTFSC STATUS,Z
          CLRF CLKSEC
ENDTIM    GOTO MAIN
SECTEN    SWAPF CLKSEC,W
          GOTO OUTPUT
SECONE    MOVF CLKSEC,W
OUTPUT    ANDLW B'00001111'
          CALL COMCATHODE
          MOVWF PORTB
          INCF DIGIT,W
          MOVWF PORTA
          RETURN
DIGSEL    INCF DIGIT,W
          ANDLW 1
          MOVWF DIGIT
          ADDWF PCL,F
          GOTO SECTEN
          GOTO SECONE
```

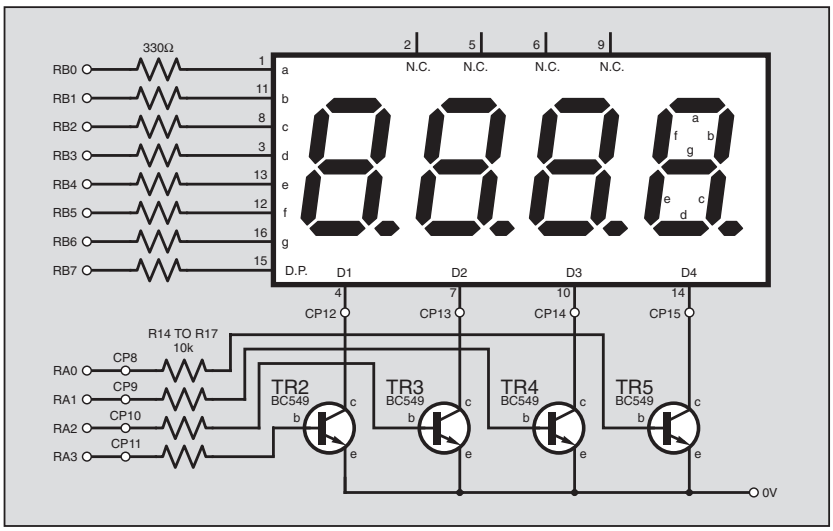

Fig.6. Connections required to drive a 4-digit 7-segment common cathode l.e.d. module.

MULTIPLEXING

Obviously, to show the tens of seconds as well we need a second 7-segment display. However, it is not possible, of course, to use the same PORTB data lines to control both displays simultaneously. Nor can we use PORTA for the second display, it hasn't enough lines. What we can do, though, is to connect PORTB to the segments of both display digits and then alternate the data being output between units and tens values, turning on each digit (via their common cathode pins) only when the relevant data is being sent to them. If this is done at a fast enough rate, the eye is fooled into thinking that both displays are on simultaneously – persistence of vision.

This technique is known as multiplexing, and what we do in this instance is to put the common cathode of each display under control of two separate data lines on PORTA, RA0 and RA1. However, the pins cannot supply sufficient current to adequately drive 7-segment displays (PIC pins can handle around 20mA to 25mA – see PIC datasheets). To provide enough current to drive them, the port lines are buffered by transistors TR2 and TR3 configured as current sinks. The emitters are connected to the 0V line and the collectors are connected to the common cathodes of the displays. The displays are turned on when PORTA lines go high.

Note that for this example the active digits are the righthand two (controlled by TR4 and TR5) in Fig.6. The other two digits are ignored.

The program which is now required to drive the two displays is shown in Listing 27. Load TK3TUT27.HEX.

Studying Listing 27, first note that (for the sake of demo) an XOR command is used to check for a count value equal to 60 (BCD). Next, and significantly for two displays, digit-alternating commands have been introduced. At label MAIN the command CALL DIGSEL is given. In DIGSEL, a digit counter (DIGIT) is incremented, ANDed with 1, and the result of this increment is carried by W into the

LISTING 28 – PROGRAM TK3TUT28

```
DIGSEL    INCF DIGIT,W
          ANDLW B'00000011'
          MOVWF DIGIT
          BTFSS PORTA,4
          ADDLW 2
          ADDWF PCL,F
          GOTO HRSTEN
          GOTO HRSONE
          GOTO MINTEN
          GOTO MINONE
          GOTO SECTEN
          GOTO SECONE
DIGSHW    MOVF DIGIT,W
          ADDWF PCL,F
          RETLW 1
          RETLW 2
          RETLW 4
          RETLW 8
MAIN      CALL DIGSEL
          BTFSS INTCON,2
          GOTO MAIN
          BCF INTCON,2
          CALL CLKADD
          GOTO MAIN
CLKADD    DECFSZ CLKCNT,F
          RETURN
          MOVLW 25
          MOVWF CLKCNT
SECCLK    INCF CLKSEC,F
          MOVLW 6
          ADDWF CLKSEC,W
          BTFSS STATUS,DC
          RETURN
          MOVWF CLKSEC
          XORLW B'01100000'
          BTFSS STATUS,Z
          RETURN
          CLRF CLKSEC
```

table that immediately follows. There are only two jumps in this table, GOTO SECTEN and GOTO SECONE.

Note that the table is still within the 256 block permitted for tables. If this were not the case, the table would need to be placed separately within that block.

Routine SECTEN extracts the tens of units value. Command SWAPF CLKSEC,W swaps the nibbles of the seconds and holds the result in W, putting the tens of seconds into the LSN position. The routine then jumps to OUTPUT, where command ANDLW B'00001111' isolates that nibble, zeroing the MSN bits now in W.

Next, the COMCATHODE table (as in Listing 26) is called to obtain the 7-segment code for that number, which is output to PORTB. Now the digit counter value is obtained (INCF DIGIT,W) and output to PORTA to turn on that digit of the display. The INCF command is used because DIGIT only alternates between 0 and 1, whereas PORTA needs to be alternated between 1 and 2 (binary 01 and 10).

Routine SECONE is similar, dealing with the units of seconds. Here we can simply get the LSN by using MOVF CLKSEC,W, ANDing it with B'00001111' at OUTPUT. The rate of alternation between the two digits is several kilohertz, slowing down briefly each time a time-out is detected.

24 HOURS

Whilst one would like to use six digits in order to display a full 24-hour clock showing hours, minutes and seconds simultaneously, this is not convenient since we only have five lines on PORTA which can control individual digits. Therefore, we must compromise and continue to use a 4-digit display but which can now have its data sources changed when a switch is pressed. In this way, we can show either hours and minutes together, or minutes and seconds. The program which does this is TK3TUT28, part of which is shown in Listing 28. Load TK3TUT28.HEX then look at listing 28.

Each time the seconds roll over to zero from 59, the minutes need incrementing; each time they roll over to zero from 59, the hours need incrementing. The hours, though, need to roll over to zero from 23. As far as incrementing each of the three counters is concerned, the easiest thing to do (but not the shortest) is to use three separate BCD routines – as we do in TK3TUT28 (see full listing). The minutes routine is the same as the seconds one, both requiring a count from 0 to 59, with routines to check for 10 and 60. The hours routine, though, requires slight alteration.

With the hours, we need to check when counts of 10 and 20 occur (+6 check), and also when 24 occurs (BCD = 00100100). This check cannot be done in the same way as for the BCD 60 check. With the latter, the check is made at the same time as the tens are incremented. For 24 hours, the simplest test is to check on each hourly digit increment:

```
HRSCLK    INCF CLKHRS,F
          MOVLW 6
          ADDWF CLKHRS,W
          BTFSC STATUS,DC
          MOVWF CLKHRS
          XORLW B'00100100'
          BTFSC STATUS,Z
          CLRF CLKHRS
```

The activating of the decimal point, when required, is done by setting the correct bit in the code once the table has been called (BSF PORTB,7), as seen in the OUTPUT routine:

```
OUTPUT    ANDLW B'00001111'
          CALL COMCATHODE
          CLRF PORTA
          MOVWF PORTB
          CALL DIGSHW
          MOVWF PORTA
          MOVF DIGIT,W
          XORLW 1
          BTFSC STATUS,Z
          BSF PORTB,7
          RETURN
```

Minutes and seconds values are dealt with in the same manner. Minutes units, though, are accompanied by the decimal point bit. Seconds are processed similarly, but without any additional bit setting for colons or points. In Tutorial 24 we shall show how a similar result can be achieved by using fewer commands. A loop plays an active role and a table is used when checking the roll-over values for the time.

In Listing 28, when switch SW4 is not pressed (checked by BTFSS PORTA,4), a value of 2 is added to effective value of DIGIT, to cause the table jumps within DIGSEL to be to the minutes and seconds display routines. Pressing SW3 results in hours and minutes being shown.

You will observe that the brilliance of the display is less than that previously seen, due to the multiplexing. In a real clock situation, the use of a high brightness display would probably be preferable.

Note that ANDing with B'00001111' (as we have done several times in this section)

LISTING 29 – PROGRAM TK3TUT29

```
TABLCD    ADDWF PCL,F
          RETLW B'00110011'
          RETLW B'00110011'
          RETLW B'00110010'
          RETLW B'00101100'
          RETLW B'00000110'
          RETLW B'00001100'
          RETLW B'00000001'
          RETLW B'00000010'
MESSAG    ADDWF PCL,F
          RETLW 'R'
          RETLW 'E'
          RETLW 'A'
          RETLW 'D'
          RETLW ' '
          RETLW 'E'
          RETLW 'P'
          RETLW 'E'

SETUP     CALL PAUSIT
LCDSET    CLRF LOOP
          CLRF RSLINE
LCDST2    MOVF LOOP,W
          CALL TABLCD
          CALL LCDOUT
          INCF LOOP,F
          BTFSS LOOP,3
          GOTO LCDST2
          CALL PAUSIT
LCDMSG    CLRF LOOP
          BSF RSLINE,4
LCDMS2    MOVF LOOP,W
          CALL MESSAG
          CALL LCDOUT
          INCF LOOP,F
          BTFSS LOOP,3
          GOTO LCDMS2
NOMORE    GOTO NOMORE

LCDOUT    MOVWF STORE
          MOVLW 50
          MOVWF LOOPA
DELAY     DECFSZ LOOPA,F
          GOTO DELAY
          CALL SENDIT
          CALL SENDIT
          RETURN
SENDIT    SWAPF STORE,F
          MOVF STORE,W
          ANDLW 15
          IORWF RSLINE,W
          MOVWF PORTB
          BSF PORTA,5
          BCF PORTA,5
          RETURN

PAUSIT    MOVLW 5
          MOVWF CLKCNT
          CLRF INTCON
PAUSE     BTFSS INTCON,2
          GOTO PAUSE
          BCF INTCON,2
          DECFSZ CLKCNT,F
          GOTO PAUSE
          RETURN
```

is a common requirement and it is actually easier to type in using its decimal equivalent of 15, so the ANDLW 15 command could be used instead.

You will see in Listing 26 that a COMANODE table is provided as well. In other applications using common anode displays this table would be called from the OUTPUT routine rather than COMCATHODE. In this instance the transistors would have their collectors connected to the +5V line and their emitters to the anode control pins of the display. Common cathode displays cannot be used with TK3's p.c.b. as the transistor emitters are connected to the 0V line.

EXERCISE 21

21.1. You may have noticed "ghost" images on the "off" segments for the active digits in TK3TUT27, but not in TK3TUT28. Study TK3TUT28's full listing and amend TK3TUT27 similarly to eliminate the "ghosts".

21.2. Create a table that holds all 16 conversions for a hexadecimal count (i.e. 0 to 9 and A to F) to be shown on a 4-digit common anode display. Write a simple counting routine which makes use of it. What compromise might you have to accept?

21.3. Extend the routine from 21.2 so that it blanks the display of any leading zeros (i.e. don't show 0007, but just show 7 on its own).

TUTORIAL 22

CONCEPTS EXAMINED

Using intelligent l.c.d.s
Setting l.c.d. contrast
Initialising the l.c.d.s
Sending a message to the l.c.d.

CONNECTIONS NEEDED

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

Having established how 7-segment displays can be driven by the PIC, we now show how an alphanumeric l.c.d. can be used to achieve not only the same result, but one that has additional facilities as well. The coding required is not especially complex, although minimum timing factors for some aspects of sending data to an l.c.d. have to be observed.

The first requirement is to show the basics of how data is output to an l.c.d. from the PIC. We shall not cover the l.c.d. itself in any great detail – you are referred to manufacturer's datasheets for more information.

Here, we first show how the l.c.d. is initialised for 4-bit data transfer from the PIC, using two control lines, RS and E. Line RS sets the l.c.d. for inputting either character data or control data. Line E tells the l.c.d. to act on the data output to it.

Disconnect the l.e.d. module and connect the l.c.d. to the p.c.b. as in Fig.7. Typical l.c.d. pinouts are shown in Fig.8. Load TK3TUT29.HEX and run it. While reading these next paragraphs, refer to Listing 29 as appropriate.

The first time the l.c.d. is used, the Contrast control VR2 should be adjusted until the display is clearly visible.

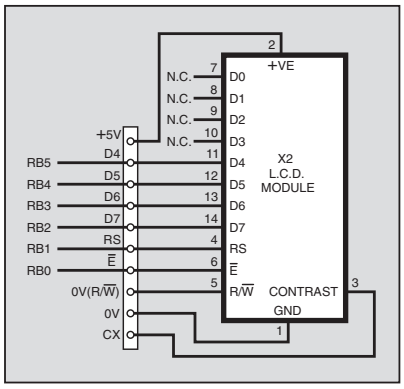

Fig.7. Connection between l.c.d. and TK3 p.c.b.

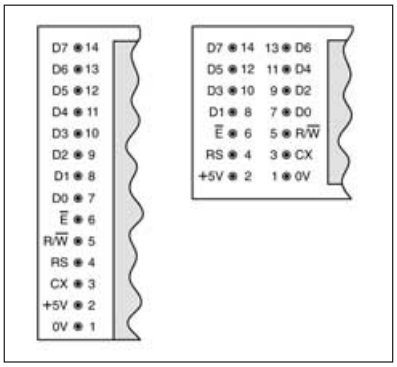

Fig.8. The two "standard" l.c.d. module pinout arrangements.

DELAYED START

When the l.c.d. is under high speed control from a device such as the PIC, it is necessary to allow a minimum of 1/5th of a second between the circuit being switched on and any data being sent to the l.c.d. So, after the PIC's initialisation, the program jumps to the routine at SETUP which, via sub-routine PAUSIT, creates this delay by making use of the prescaler.

The prescaler has been set for an INTCON,2 pulse every 1/25th of a second, so a loop beginning at PAUSE is used to wait for five of these pulses to be completed, i.e. 1/5th of a second. Then a series of commands is sent to set the l.c.d. into the required 4-bit mode. (There are other command routines possible which achieve a similar result.) The commands are held in the table TABLCD, which is accessed from the routine at label LCDSET. The first command here clears the loop counter and the byte (RSLINE) which holds the RS-controlling bit.

Bit 4 of RSLINE is used to inform the l.c.d. what type of data is being sent to it. The bit is cleared for control data, and set for character data. Now, in the manner of table use which was demonstrated earlier, the control commands from TABLCD are sent to the l.c.d. via the LCDST2 routine.

The loop counter is then cleared, RSLINE bit 4 is set and used to inform the l.c.d. that the next commands being sent to it are character data.

Then the message held in the table MESSAG is sent via the routine headed LCDMSG. The l.c.d. displays this message on its first line, starting at the left. Now, for the sake this demo, the perpetual loop at NOMORE is entered and no more actions occur. To replay the routine, use the Reset switch.

In both data sending routines, the l.c.d. output routine is called by the command CALL LCDOUT. The entire block between the start of LCDOUT and the final RETURN at the end of SENDIT is responsible for sending each byte of 8-bit data to the LCD as two 4-bit nibbles, to which control data is then ORed to expand them to a full 8-bit byte. Nibble data is held in the LSN of this byte, control data (in this instance just that for lines RSLINE and E) is held in its MSN.

On entry into LCDOUT, the data brought in on W is copied into a temporary file, STORE. Now a delay loop is entered. The l.c.d. can only handle bytes of data coming to it at a rate which allows previous data received to be processed fully. Details of the delay required are stated in manufacturer's datasheets. In theory, the delay depends on the type of data and command being sent, but on a practical level, a fixed delay of so many PIC commands can be used. In this example, LOOPA is set for 50 and then decremented until zero, as performed by the instructions:

```
        MOVLW 50
        MOVWF LOOPA
DELAY   DECFSZ LOOPA,F
        GOTO DELAY
```

In the author's experience with many programs, this delay is satisfactory for a PIC running at up to about 5MHz. Too short a delay will result in erratic behaviour of the l.c.d., probably accompanied by erroneous display results. The most likely result of this is that the display will not enter 2-line mode, characterised by a line of dark pixels on the upper line, but none on the lower, which will remain blank. If this occurs, the delay loop value should be increased.

Following the delay, there is a call to SENDIT. In SENDIT, the MS nibble of data is retrieved from STORE with the commands:

```
SWAPF STORE,F
MOVF STORE,W
ANDLW 15
```

The first command swaps the two nibbles within STORE, the second copies STORE into W, and then W is ANDed with 15 to isolate bits 0 to 3. The result is ORed with the RSLINE bit and the byte is then output to the l.c.d. via PORTB. The E line is taken high and immediately low again, telling the l.c.d. to process the data on its data inputs. A return to the calling point occurs and then SENDIT is again called. This time, the LSN is extracted from STORE and sent to the l.c.d. in the same way. After two returns, the program returns to the original calling point.

It is important to note that the port bits which are used in these routines to control the Data, RSLINE and E lines reflect the physical connections between the PIC and the l.c.d. as shown in Fig.7. It is permissible to use other PIC port lines for this purpose, but the controlling bits of the software must be changed accordingly.

EXERCISE 22

22.1. There are two commands in the LCDOUT to SENDIT routine which, while being perfectly legitimate, are actually unnecessary. What are they and why are they not needed? (Think "default".)

22.2. When the l.c.d. is first initialised, it is possible (though not definite) that all its character positions (cells) will show as black squares. Sending the message will correct that situation for the first eight cells. How could you ensure that the remaining eight cells on the top line are set to "clear" blanks? There are two methods; try both.

22.3. How would you now set the lower line to all blanks?

TUTORIAL 23

CONCEPTS EXAMINED

Coding hours, minutes and seconds for an alphanumeric l.c.d.
Shortened clock monitoring code
Command SUBLW
Command SUBWF

CONNECTIONS NEEDED

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

Having shown how the l.c.d. can have data written to it, we now show how the method can be extended in order to display 24-hour clock data.

Load TK3TUT30.HEX then glance at the display from time to time while you read on here.

COMMANDS SUBLW AND SUBWF

Rather late on perhaps, in the program we are about to display we illustrate the first use of subtraction. PICs have two subtraction commands, SUBLW (Subtract W from Literal) and SUBWF (Subtract W from File). The latter command is used with either the F or the W suffix, e.g. SUBWF (FILE),F and SUBWF (FILE),W.

One might reasonably have expected that SUBLW would actually mean Subtract Literal from W. This is not the case, the subtraction is that of W from the Literal. Consequently, unless you keep your wits about you, this is a command that you could quite easily use incorrectly.

In the following code, the value in the file named DEMO is subtracted from 30 and the result put back into DEMO (the first two lines are just to put an initial value into DEMO):

```
MOVLW 20
MOVWF DEMO
MOVF DEMO,W
SUBLW 30
MOVWF DEMO
```

In this case, the answer is 10 (30 – 20), even though instinctively we might have expected 30 to be subtracted from 20. In this next example, to illustrate SUBWF, again it is the value already in W which is subtracted from the value in file DEMO, the result being returned to DEMO. This is more logical. (Once more the first two commands are just to put an initial value into DEMO.)

LISTING 30 – PROGRAM TK3TUT30

```
MAIN    BTFSS INTCON,2
        GOTO MAIN
        BCF INTCON,2
        CALL CLKADD
        GOTO MAIN
CLKADD  DECFSZ CLKCNT,F
        RETURN
        MOVLW 25
        MOVWF CLKCNT
        MOVLW CLKSEC
        MOVWF FSR
        MOVLW 3
        MOVWF LOOP
        CLRF STORE1
ADDCLK  INCF INDF,F
        MOVLW 6
        ADDWF INDF,W
        BTFSC STATUS,DC
        MOVWF INDF
ADDCL2  MOVF STORE1,W
        CALL CHKVAL
        MOVWF STORE2
        MOVF INDF,W
        SUBWF STORE2,F
        BTFSC STATUS,C
        GOTO CLKSHW
        CLRF INDF
        INCF STORE1,F
        INCF FSR,F
        DECFSZ LOOP,F
        GOTO ADDCLK
CLKSHW  MOVLW B'11000000'
        CALL LCDLIN
        MOVF CLKHRS,W
        CALL LCDFRM
        MOVLW ':'
        CALL LCDOUT
        MOVF CLKMIN,W
        CALL LCDFRM
        MOVLW '.'
        CALL LCDOUT
        MOVF CLKSEC,W
        CALL LCDFRM RETURN
LCDFRM  MOVWF STORE2
        SWAPF STORE2,W
        ANDLW 15
        IORLW 48
        CALL LCDOUT
        MOVF STORE2,W
        ANDLW 15
        IORLW 48
        CALL LCDOUT
        RETURN
LCDLIN  BCF RSLINE,4
        CALL LCDOUT
        BSF RSLINE,4
        RETURN
```

```
MOVLW 20
MOVWF DEMO
MOVLW 5
SUBWF DEMO,F
```

The answer put back into DEMO is, of course, 15 (20 – 5).

In these two examples, the value subtracted is less than the value from which it is being subtracted. What happens if the opposite is true?

For a start, if the value subtracted is greater than the value from which it is being subtracted, the byte simply "rolls-over". We have already shown that decrementing a value of zero results in an answer of 255. Decrementing, of course, is simply a subtraction of 1 from a number and we could, therefore, consider the 0 – 1 situation as being expressed (256 + 0) – 1 = 255.

What we have done by using the addition of 256, is to "borrow" the 256 in order to achieve the correct 8-bit result. The same roll-over situation applies to subtraction of numbers greater than 1. Thus subtracting 20 from 10 produces an answer of 246 (256 + 10 – 20 = 246).

We are quite used to "borrowing" in normal arithmetic, so the concept should be familiar to you, although we express the result of subtracting 20 from 10 as equalling –10.

The difference with PICs (and other digital devices) is that we cannot produce a negative answer as such. What we can do, however, is to use a flag to indicate that a borrow or negative answer situation has occurred. With the PIC, the Carry bit is used for this purpose. In a subtraction operation we simply test the Carry bit to establish whether or not there has been a borrow.

This, though, is where another "inverted" concept has to be applied to SUB commands. Whereas with the ADD commands the Carry bit is Set if a carry result occurs, with the SUB commands the Carry bit is Cleared if a borrow occurs, and it is Set if a borrow does *not* occur.

You could, perhaps, regard the Carry bit as being the bit which is available to be "borrowed" for the subtraction, hence it remaining set if a borrow is not needed, and cleared if it is.

The following are examples of routines which test the Carry bit in a subtraction operation:

```
MOVLW 30
MOVWF DEMO
MOVF DEMO,W
SUBLW 20
MOVWF DEMO
BTFSS STATUS,C
INCF STORE,F
RETURN
```

The above example will cause STORE to be incremented since a borrow will occur when 30 is subtracted from 20. The next example, 30 – 20, does not result in a borrow, so STORE remains at its previous value:

```
MOVLW 20
MOVWF DEMO
MOVF DEMO,W
SUBLW 30
MOVWF DEMO
BTFSS STATUS,C
INCF STORE,F
RETURN
```

You will see the use of SUBWF and the subsequent testing of the Carry bit for the occurrence of a borrow in TK3TUT30.

NEXT MONTH

In the final part of this series we move on to some of a PIC's "sophisticated" operations. You might also care to obtain a PIC16F877 (although this is optional) as we also illustrate some advanced programming techniques that can be used with this device family, as listed on page 3 of Part 1.