

# Introduction to PIC Programming

## Midrange Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

### Lesson 12: Driving 7-Segment Displays

Up until now, we have only used a small number of individual LEDs as outputs. LEDs are adequate as simple indicators, but many applications need to be able to display information in numeric, alphanumeric or graphical form. Although LCD and OLED displays are becoming more common, there is still a place, when displaying numeric (or sometimes hexadecimal) information, for 7-segment LED displays.

This lesson revisits the material from [baseline lesson 8](#), showing how techniques such as *lookup tables* and interrupt-driven *multiplexing* can be used with a PIC16F684 to drive up to four 7-segment displays.

In summary, this lesson covers:

- Driving a single 7-segment display
- Using lookup tables
- Using interrupt-driven multiplexing to drive multiple displays
- Binary-coded decimal (BCD)

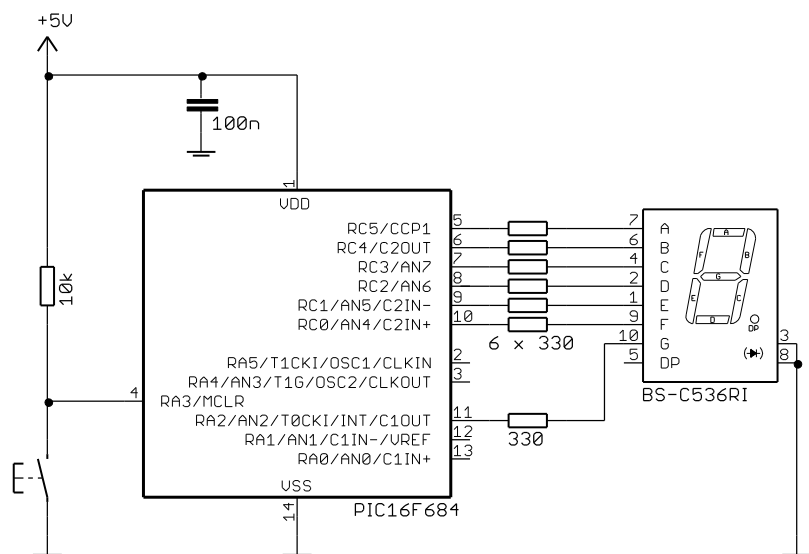
#### Driving a 7-segment LED Display

A 7-segment LED display is simply a collection of LEDs, typically one per segment (but often having two or more LEDs per segment for large displays), arranged in a “figure 8” pattern. 7-segment display modules also commonly include one or two LEDs for decimal points.

7-segment LED display modules typically come in one of two varieties: common-anode or common-cathode.

In a common-cathode module, the cathodes belonging to each segment are wired together within the module, and brought out through one or two (or sometimes more) pins. The anodes for each segment are brought out separately, each to its own pin. Typically, each segment would be connected to a separate output pin on the PIC, as shown in the circuit diagram on the right.

The common cathode pins are connected together and grounded.



To light a given segment in a common-cathode display, the corresponding PIC output is set high. Current flows from the output and through the given segment (limited by a series resistor) to ground.

In a common-anode module, this is reversed; the anodes for each segment are wired together and the cathodes are connected separately. In that case, the common anode pins are connected to the positive supply and each cathode is connected to a separate PIC output. To light a segment in a common-anode display, the corresponding PIC output is set low; current flows from the positive supply, through the segment and into the PIC's output.

Although a single pin can source or sink up to 25 mA, the combined maximum for all the pins on the 16F684 is 90 mA and since all segments may be lit at once (when displaying '8'), we need to limit the current per pin to  $90 \text{ mA} \div 6 = 15 \text{ mA}$ . The  $330 \Omega$  resistors limit the current to 10 mA, well within spec while giving a bright display, and allowing some overhead for adding LEDs or other outputs on the unused pins, in future.

The examples in this section assume a common-cathode display, as shown in the circuit diagram above. If you have a common-anode display, you will need to wire it correctly and make appropriate changes to the code presented here, but the techniques for driving the display are essentially the same.

7-segment displays come in a variety of modules in a range of sizes from a number of manufacturers; yours will very likely have a different pin-out to that shown above. So don't follow the pin numbering shown; be careful to connect your module so that segment A connects to RC5, segment B connects to RC4, etc. If you are using Microchip's Low Pin Count Demo Board, these connections can be made via the 14-pin header on that board.

Or, you could connect your module in a way that simplifies the wiring, and instead change the lookup tables in the code (see section below) to reflect your wiring. You'll find when you design circuit boards for your project or product, that board layout and pin assignments go hand in hand; it's common to change pin assignments to simplify the board layout, in a process that may go through a number of iterations.

### Lookup tables

To display each digit, a corresponding pattern of segments must be lit, as follows:

Segment:	A	B	C	D	E	F	G
Pin:	RC5	RC4	RC3	RC2	RC1	RC0	RA2
0	on	on	on	on	on	on	off
1	off	on	on	off	off	off	off
2	on	on	off	on	on	off	on
3	on	on	on	on	off	off	on
4	off	on	on	off	off	on	on
5	on	off	on	on	off	on	on
6	on	off	on	on	on	on	on
7	on	on	on	off	off	off	off
8	on	on	on	on	on	on	on
9	on	on	on	on	off	on	on

We need a way to determine the pattern corresponding to the digit to be displayed, and that is most effectively done using a *lookup table*.

The midrange PIC architecture is similar to the baseline architecture in that the most common method of implementing lookup tables is to use a computed jump into a table of ‘retlw’ instructions, as explained in [baseline lesson 8](#).

In that lesson, to return the binary pattern to be applied to PORTC, corresponding to the digit in W, we used the following subroutine:

```
get7sC   addwf   PCL, f
         retlw  b'111111'   ; 0
         retlw  b'011000'   ; 1
         retlw  b'110110'   ; 2
         retlw  b'111100'   ; 3
         retlw  b'011001'   ; 4
         retlw  b'101101'   ; 5
         retlw  b'101111'   ; 6
         retlw  b'111000'   ; 7
         retlw  b'111111'   ; 8
         retlw  b'111101'   ; 9
```

The ‘addwf f, d’ (“**add W to file register**”, which places the result of the addition in the register if the destination is ‘, f’, or in W if the destination is ‘, w’) instruction is used here to add the *offset* held in the W register to the current value of the *program counter* (PC). This is called a *computed GOTO* operation, and has the effect of skipping program execution forward by the offset amount, jumping to the table entry corresponding to the offset.

The program counter is a 13-bit register holding the full address of the next instruction to be executed. But, because the midrange PICs are 8-bit devices, they cannot update a 13-bit register, such as PC, all at once. Only the lower byte (PC<7:0>) is directly accessible, through the PCL register. If you change the contents of PCL, you change the program counter – affecting which instruction will be executed next.

As mentioned in [lesson 2](#), two bits of the PCLATH register (PCLATH<4:3>) define the current program memory page; these two bits are updated by the pagesel directive, and are copied to PC<12:11> whenever a call or goto instruction is executed.

However, whenever an instruction, such as movwf or addwf, writes to PCL, the upper five bits of the program counter is loaded from all five PCLATH bits: PCLATH<4:0> is copied to PC<12:8>.

This means that, when you perform a computed GOTO on midrange PICs, you must ensure that the correct value is loaded into PCLATH, before you write to PCL. Otherwise, if PCLATH is pointing to some other part of memory, your code will jump there, and not into your table, when you update PCL.

You can’t use pagesel to do this, because pagesel only affects PCLATH<4:3> - you must ensure that every bit of PCLATH is correct.

This can be done by using the ‘HIGH’ assembler directive, to load the “high” part (bits 8 to 13) of the destination address into PCLATH, before accessing (‘calling’, since it is actually a subroutine) the lookup table, as follows:

```
        movlw  HIGH get7sC    ; load high bits of table base address
        movwf  PCLATH         ; into PCLATH
        movf   digit, w       ; get digit to display
        call   get7sC         ; lookup pattern for port C
        movwf  PORTC         ; then output it
```

(assuming that the digit to be displayed is stored in a variable called ‘digit’)

Note that there is no need to use pagesel before the call instruction, because all pagesel does is to ensure that PCLATH<4:3> is correct, but we have already loaded the whole of PCLATH with the correct value, to jump to the base of the lookup table.

Unfortunately, there is still a potential problem.

Recall that, in [baseline lesson 8](#), it was explained that, in the baseline PIC architecture, lookup tables must be wholly contained within the first 256 addresses of a program memory page.

If you are happy with that type of restriction, you can carefully place each of your tables entirely between 256-word program memory boundaries, and the above code will work.

However, the midrange architecture is more flexible than that. A lookup table can be placed anywhere in program memory, and can be any length – but that flexibility comes at a cost. You must be careful, in case your table crosses any 256-word boundaries, to ensure that PCLATH is updated correctly.

Consider the above code again, but this time with the table located such that it crosses a 256-word boundary:

```
TABLES CODE 0x02FC
get7sC addwf PCL, f
      retlw b'111111' ; 0
      retlw b'011000' ; 1
      retlw b'110110' ; 2
      retlw b'111100' ; 3
      retlw b'011001' ; 4
      retlw b'101101' ; 5
      retlw b'101111' ; 6
      retlw b'111000' ; 7
      retlw b'111111' ; 8
      retlw b'111101' ; 9
```

This places the base of the table (the ‘`addwf PCL, f`’ instruction) at 0x02FC – just before the end of a 256-word boundary.

The first `retlw` instruction will be located at 0x02FD, the second at 0x02FE, the third at 0x02FF, and the fourth at 0x0300.

If we run:

```
movlw HIGH get7sC ; load high bits of table base address
movwf PCLATH ; into PCLATH
```

PCLATH will be loaded with 0x02, since that is the high byte of the table base address (0x02FC).

If we then call the table with  $W = 0$ , the ‘`addwf PCL, f`’ instruction will add 0 to PCL, which contains 0xFD (lower 8 bits of the address of the next instruction), and 0xFD will be written back to PCL (and hence PC<7:0>). At the same time, the upper five bits of the program counter will be loaded from PCLATH, ending up with PC = 0x02FD. That’s the address of the first `retlw` instruction, which will therefore be executed next – exactly the behaviour we want.

Suppose instead that we call the table with  $W = 3$ . Adding 3 to 0xFD gives 0x100, but that’s a 9-bit number. The result cannot fit into the 8-bit PCL register (we say that an *overflow* has taken place), so the value written back is 0x00. But PCLATH still holds 0x02, so the value loaded into the program counter is 0x0200. Therefore the next instruction to be executed will be whatever happens to be at address 0x0200! That’s not part of the table; your program will be executing unintended instructions, and will probably crash. At the very least, it won’t work correctly.

To avoid this problem, you must test for an overflow, or *carry*, condition, and increment PCLATH if the computed address is across the 256-word boundary.

As we saw in [lesson 4](#), this can be done by testing the C (carry) bit in the STATUS register, following an addition.

If you wish to implement a single table longer than 255 entries, you must perform a full 16-bit addition to compute the table entry address.

But if the table has fewer than 256 entries (most do), meaning that the offset is an 8-bit number, the table lookup can be performed as follows:

```

movlw   HIGH (get7sC+1)   ; load PCLATH with base of table
movwf   PCLATH
movlw   LOW (get7sC+1)    ; add offset of digit to display
addwf   digit,w
btfsc   STATUS,C          ; if overflow
incf    PCLATH,f          ; increment PCLATH
call    get7sC            ; lookup pattern for port C
movwf   PORTC            ; then output it

```

The addition is now being done before the table is called. So now, instead of beginning the table with an 'addwf' instruction, we use 'movwf' to load the calculated address into PCL:

```

get7sC  movwf   PCL
        retlw  b'111111'   ; 0
        retlw  b'011000'   ; 1
        retlw  b'110110'   ; 2
        retlw  b'111100'   ; 3
        retlw  b'011001'   ; 4
        retlw  b'101101'   ; 5
        retlw  b'101111'   ; 6
        retlw  b'111000'   ; 7
        retlw  b'111111'   ; 8
        retlw  b'111101'   ; 9

```

Because we are now moving the calculated address into PCL, and no longer adding to the current contents of the program counter (which points to the next instruction to be executed, not the current instruction), the address of the first table entry (the first 'retlw' instruction, corresponding to an offset of 0) has to be used as the base of the lookup address calculation. This is why 'get7sC+1' is used here; it is the address of the first table entry.

Since a table lookup is an operation we will want to perform a number of times (twice in this example; we need to look up patterns for both PORTA and PORTC), it would be useful to express the code above, using a more generic, re-usable macro (see [lesson 5](#)):

```

; Perform table lookup
;
; parameters: table = address of table base ('movwf PCL' instruction)
;              off_var = 8-bit variable holding offset into table
;                  (assumes correct bank already selected)
;
; Table entry at given offset is returned in W
; (first entry is at offset 0)
;
Lookup  MACRO   table,off_var

        movlw  HIGH (table+1)   ; load PCLATH with base of table
        movwf  PCLATH
        movlw  LOW (table+1)    ; add offset to base
        addwf  off_var,w
        btfsc  STATUS,C        ; if overflow
        incf   PCLATH,f        ; increment PCLATH
        call   table           ; perform table lookup
        ENDM

```

This macro assumes that the table offset is stored in a variable, which will normally be the case, as in this example, where we are using the 'digit' variable.

We can then use the macro to perform the table lookups, as follows:

```
Lookup  get7sA,digit      ; lookup pattern for port A
banksel PORTA            ; then output it
movwf   PORTA
Lookup  get7sC,digit      ; repeat for port C
movwf   PORTC
```

### **The define table directive**

Since lookup tables are very useful, and commonly used, the MPASM assembler provides a shorthand way to define them: the 'dt' (short for "define table") directive. Its syntax is:

```
[label] dt      expr1[,expr2,...,exprN]
```

where each expression is an 8-bit value. This generates a series of `retlw` instructions, one for each expression. The directive is equivalent to:

```
[label] retlw   expr1
        retlw   expr2
        ...
        retlw   exprN
```

Thus, we could express the table above as:

```
get7sC  movwf   PCL
        dt      b'111111',b'011000',b'110110',b'111100',b'011001' ; 0,1,2,3,4
        dt      b'101101',b'101111',b'111000',b'111111',b'111101' ; 5,6,7,8,9
```

or it could even be written as:

```
get7sC  movwf   PCL
        dt      0x3F,0x18,0x36,0x3C,0x19,0x2D,0x2F,0x38,0x3F,0x3d ; 0-9
```

Of course, the `dt` directive is more appropriate in some circumstances than others. Your table may be easier to understand if you use only one expression per line, as in this example, where it is probably clearer to simply use `retlw`.

A special case where 'dt' makes your code much more readable is with text strings. For example:

```
dt      "Hello world",0
```

is equivalent to:

```
retlw   'H'
retlw   'e'
retlw   'l'
retlw   'l'
retlw   'o'
retlw   ' '
retlw   'w'
retlw   'o'
retlw   'r'
retlw   'l'
retlw   'd'
retlw   0
```

The 'dt' form is clearly preferable in this case.

**Complete program**

The following program uses the code fragments presented above, and code (e.g. macros) and techniques from previous lessons, to implement a simple seconds counter. It counts repeatedly from 0 to 9 on the single 7-segment display, with 1 second between each count.

```

;*****
; Description: Lesson 12, example 1a *
; *
; Demonstrates use of lookup tables to drive 7-segment display *
; *
; Single digit 7-segment LED display counts repeating 0 -> 9 *
; 1 second per count, with timing derived from int 4MHz oscillator *
; *
;*****
; Pin assignments: *
; RA2, RC0-5 - 7-segment display (common cathode) *
; *
;*****

list p=16F684
#include <p16F684.inc>

#include <stdmacros-mid.inc> ; DelayMS - delay in milliseconds

errorlevel -302 ; no "register not in bank 0" warnings
errorlevel -312 ; no "page or bank selection not needed" messages

radix dec

EXTERN delay10 ; W x 10ms delay

;***** CONFIGURATION
; ext reset, no code or data protect, no brownout detect,
; no watchdog, power-up timer, int clock with I/O,
; no failsafe clock monitor, two-speed start-up disabled
__CONFIG _MCLRE_ON & _CP_OFF & _CPD_OFF & _BOD_OFF & _WDT_OFF & _PWRTE_ON
& _INTOSCIO & _FCMEN_OFF & _IESO_OFF

;***** MACROS

; Perform table lookup
;
; parameters: table = address of table base ('movwf PCL' instruction)
; off_var = 8-bit variable holding offset into table
; (assumes correct bank already selected)
;
; Table entry at given offset is returned in W
; (first entry is at offset 0)
;
Lookup MACRO table,off_var

    movlw HIGH (table+1) ; load PCLATH with base of table
    movwf PCLATH
    movlw LOW (table+1) ; add offset to base
    addwf off_var,w
    btfsc STATUS,C ; if overflow
    incf PCLATH,f ; increment PCLATH
    call table ; perform table lookup
    ENDM

```

```

;***** VARIABLE DEFINITIONS
        UDATA_SHR
digit   res 1          ; digit to be displayed

;*****
RESET   CODE    0x0000      ; processor reset vector

;***** MAIN PROGRAM

;***** Initialisation
        banksel TRISA      ; configure PORTA and PORTC as all outputs
        clrf   TRISA
        clrf   TRISC

        clrf   digit      ; start with digit = 0

;***** Main loop
count
        ; display digit
        Lookup tb7segA,digit ; lookup pattern for port A
        banksel PORTA      ; then output it
        movwf  PORTA
        Lookup tb7segC,digit ; repeat for port C
        movwf  PORTC

        DelayMS 1000      ; delay 1s

        incf   digit,f    ; increment digit
        movlw  .10
        xorwf  digit,w    ; if digit = 10
        btfsc STATUS,Z
        clrf   digit      ; reset it to 0

        pagesel count    ; repeat forever
        goto  count

;***** LOOKUP TABLES
TABLES  CODE

; Digit to pattern lookup table for 7 segment display on port A
; RA2 = G
tb7segA movwf  PCL
        retlw  b'000000'   ; 0
        retlw  b'000000'   ; 1
        retlw  b'000100'   ; 2
        retlw  b'000100'   ; 3
        retlw  b'000100'   ; 4
        retlw  b'000100'   ; 5
        retlw  b'000100'   ; 6
        retlw  b'000000'   ; 7
        retlw  b'000100'   ; 8
        retlw  b'000100'   ; 9

; Digit to pattern lookup table for 7 segment display on port C
; RC5:0 = ABCDEF
tb7segC movwf  PCL
        retlw  b'111111'   ; 0
        retlw  b'011000'   ; 1
        retlw  b'110110'   ; 2

```

```

retlw  b'111100'    ; 3
retlw  b'011001'    ; 4
retlw  b'101101'    ; 5
retlw  b'101111'    ; 6
retlw  b'111000'    ; 7
retlw  b'111111'    ; 8
retlw  b'111101'    ; 9

END
    
```

### Interrupt-driven Multiplexing

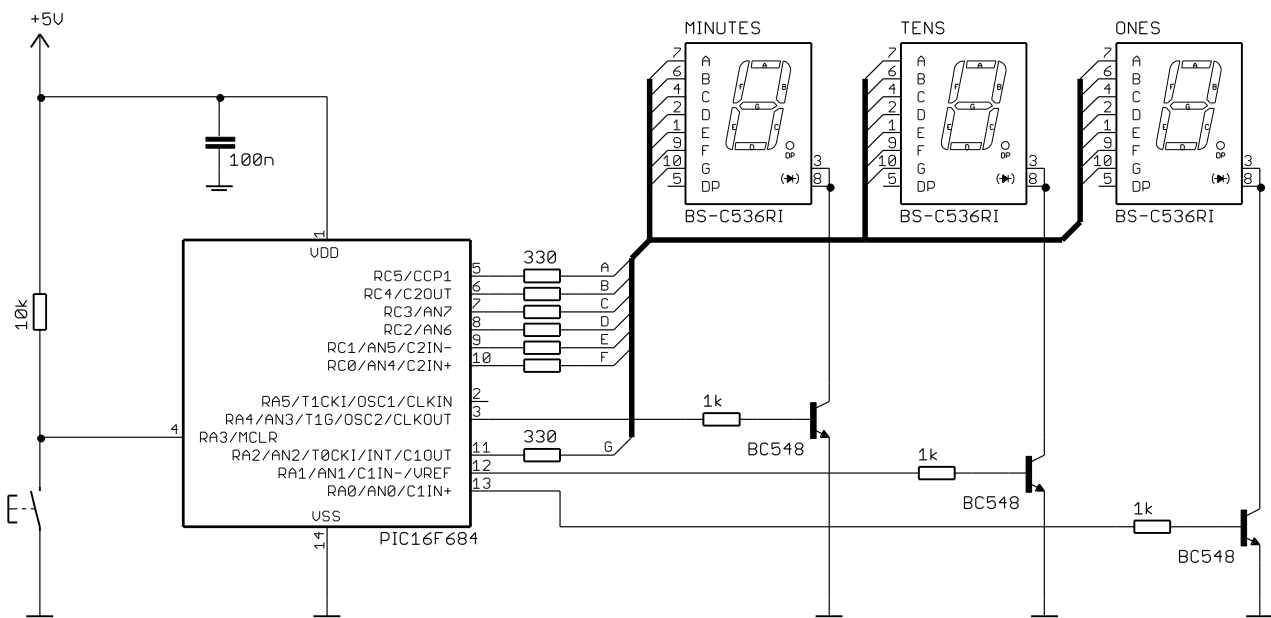
To display multiple digits, as in (say) a digital clock, the obvious approach is to extend the method used above for a single digit. That is, where one digit requires 7 outputs, two digits would apparently need 14 outputs; four digits would need 28 outputs, etc. At that rate, you would very quickly run out of output pins, even on the bigger PICs!

A technique commonly used to conserve pins is to *multiplex* a number of displays (and/or inputs – a topic we’ll look at another time).

Display multiplexing relies on speed, and human persistence of vision, to create an illusion that a number of displays are on at once, whereas in fact they are being lit rapidly in sequence, so quickly that it appears that they are on continuously.

To multiplex 7-segment displays, it is usual to connect each display in parallel, so that one set of output pins on the PIC drives every display at once, the connections between the modules and to the PIC forming a *bus*. If the common cathodes are all grounded, every module would display the same digit (feebly, since the output current would be shared between them).

To enable a different digit to be displayed on each module, the individual displays need to be switched on or off under software control, and for that, transistors are usually used, as illustrated below:



Note that it is not possible to connect the common cathodes directly to the PIC’s outputs; the combined current from all the segments in a module will be up to 70 mA – too high for a single pin to sink. Instead, the output pin is used to switch a transistor on or off.

Almost any NPN transistor<sup>1</sup> could be used for this, as is it not a demanding application. It's also possible to use FETs; for example, MOSFETs are usually used to switch high-power devices.

When the output pin is set 'high', the transistor's base is pulled high, turning it 'on'. The 1 kΩ resistors are used to limit the base current to around 4.4 mA – enough to saturate the transistor, effectively grounding the module's common cathode connection, allowing the display connected to that transistor to light.

These transistors are then used to switch each module on, in sequence, for a short time, while the pattern for that digit is output on the display bus. This is repeated for each digit in the display, quickly enough to avoid visible flicker (preferably at least 70 times per second).

The approach taken in the single-digit example above – set the outputs and then delay for 1 s – won't work, since the display multiplexing has to continue throughout the delay.

Ideally the display multiplexing would be a "background task"; one that continues steadily while the main program is free to perform tasks such as responding to changing inputs. As we saw in [lesson 6](#), that's an ideal application for timer-based interrupts.

A timer, such as Timer0, would be used to trigger a regular interrupt. The interrupt service routine displays each digit, one at a time, in succession. If the interrupt is triggered at 1 ms intervals, one digit would be displayed for 1 ms, then the next digit for another 1 ms, and so on. If there are three digits, as in the circuit above, each digit will be on for 1 ms, then off for 2 ms, with the whole display being refreshed every 3 ms.

But before going all the way to a three-digit, multiplexed example, let's start by converting the previous single-digit example, so that a timer-based interrupt is used to maintain the display, with the "main loop" doing nothing more than incrementing the 'digit' variable, which the ISR will then display<sup>2</sup>.

We'll use the digit labelled 'ONES' in the circuit above. To enable that digit, we need to raise RA0.

Note, however, that when we write a looked-up value to PORTA, the whole of PORTA is overwritten, not only the bits used for the display. That means that the following approach **will not work**:

```
bsf      PORTA,0          ; raise RA0 to enable display

Lookup  tb7segA,digit    ; lookup pattern for PORTA
banksel PORTA           ; then output it
movwf   PORTA
```

This is because the final 'movwf PORTA' overwrites every bit of PORTA, including RA0.

If you want the 'bsf PORTA, 0' to stick, you have to put it after the write to PORTA:

```
Lookup  tb7segA,digit    ; lookup pattern for PORTA
banksel PORTA           ; then output it
movwf   PORTA

bsf      PORTA,0          ; raise RA0 to enable display
```

This will work correctly.

---

<sup>1</sup> If you had common-anode displays, you would normally use PNP transistors as high-side switches (between VDD and each common anode), instead of the NPN low-side switches shown here.

<sup>2</sup> It is usually a good idea, when developing an application, to build it up one piece at a time, so that you can test each section, concept or algorithm, as you go.

Placing this code into an interrupt service routine (see [lesson 6](#)), and using shadow registers to avoid any potential read-modify-write issues<sup>3</sup>, we have:

```

; *** Service Timer0 interrupt
;   TMR0 overflows every 2.048 ms
;
bcf      INTCON,T0IF      ; clear interrupt flag

; display digit (using shadow registers)
Lookup  tb7segA,digit    ; lookup pattern for PORTA
movwf   sPORTA          ; then output it
Lookup  tb7segC,digit    ; repeat for PORTC
movwf   sPORTC
bsf     sDISPLAY        ; enable display

; copy shadow regs to ports
banksel PORTA
movf    sPORTA,w
movwf   PORTA
movf    sPORTC,w
movwf   PORTC

```

The symbol 'sDISPLAY' is defined by:

```
#define sDISPLAY    sPORTA,0      ; display enable (shadow)
```

Timer0 is set up to overflow (generating an interrupt) every 2.048 ms, and the timer interrupt enabled, by:

```

; configure timer
movlw   b'11000010'      ; configure Timer0:
;   --0-----          timer mode (TOCS = 0)
;   ----0----          prescaler assigned to Timer0 (PSA = 0)
;   -----010         prescale = 8 (PS = 010)
banksel OPTION_REG      ; -> increment TMR0 every 8 us
movwf   OPTION_REG      ; -> TMR0 overflows every 2.048 ms
; configure interrupts
movlw   1<<GIE|1<<T0IE  ; enable Timer0 and global interrupts
movwf   INTCON

```

With this interrupt code running in the background, taking care of displaying the current contents of the 'digit' variable, all the main loop code has to do is update the value of 'digit' every second:

```

count   ; delay 1 sec
DelayMS 1000          ; delay 1s

; increment digit (displayed by ISR)
incf    digit,f
movlw   .10
xorwf   digit,w      ; if digit = 10
btfsc  STATUS,Z
clrf    digit        ; reset it to 0

; repeat forever
pagesel count
goto   count

```

---

<sup>3</sup> This is not strictly necessary for PORTC, in this example, because PORTC is only ever written as a whole. A shadow register is only used here for consistency with PORTA.

That's one of the main advantages of using a timer-based "background" interrupt to maintain the display; your main code only has to update the display contents, without worrying about the mechanics of how it is displayed, making the main code easier to follow.

### **Example application**

To demonstrate display multiplexing, we'll extend the example above so that the first digit will count minutes and the next two digits will count seconds (00 to 59).

Whenever you use display (or other) multiplexing, you must decide on a multiplexing rate – in this case, how quickly the digits are updated, corresponding to the *tick* speed (how often then timer-based interrupt runs). Making the rate faster (shorter ticks), means less perceptible flicker, but the faster the multiplexing rate, the greater the proportion of time spent updating the display. So you need to find a sensible balance – generally speaking it's a good idea to update the display as slowly as you can get away with, to maximise the time available for other tasks.

In this case, if each of three digits is updated at a rate of 2 ms per digit, the whole 3-digit display is updated every 6 ms, so the display rate is  $1 \div 6 \text{ ms} = 167 \text{ Hz}$  – fast enough to avoid perceptible flicker.

Since the exact display refresh rate doesn't matter, we can use an "easy to generate" 2.048 ms tick, as in the single-digit example above.

To store the count, the simplest approach is to use three variables, to store the minutes, tens and ones digits separately.

Each time the interrupt is triggered, the next digit in sequence is displayed, so we need a way to keep track of which digit to display – a variable which is incremented each time the ISR runs.

So we need the following variables to keep track of the current count and the digit to display next:

```
GENVAR      UDATA                ; general variables
mpx_cnt     res 1                 ; multiplex counter
mins       res 1                 ; current count: minutes
tens       res 1                 ; tens
ones       res 1                 ; ones
```

This "multiplex counter" is used in the ISR to select which digit to display.

We could express this in pseudo code as:

```
; display next digit in sequence
; (determined by current value of mpx_cnt)
if mpx_cnt = 0
    display ones digit
if mpx_cnt = 1
    display tens digit
if mpx_cnt = 2
    display minutes digit
; increment mpx_cnt, to select next digit for next time
mpx_cnt = mpx_cnt + 1
if mpx_cnt = 3          ; reset count if at end of digit sequence
    mpx_cnt = 0
```

We've seen that it is possible to use XOR to test whether a register equals a certain value, for example:

```
movf      mpx_cnt,w
xorlw    1                ; if mpx_cnt = 1
btfsc   STATUS,Z
call     dsp_tens         ; display tens digit
```

Note that, to test whether a register is zero, no XOR operation is necessary; the ‘movf’ instruction updates the Z (zero) flag, depending on the contents of the register being moved.

We could therefore directly translate the above pseudo code into assembler as:

```

; display next digit in sequence
; (determined by current value of mpx_cnt)
movf    mpx_cnt,w          ; if mpx_cnt = 0
btfsc  STATUS,Z
call   dsp_ones           ; display ones digit
movf    mpx_cnt,w
xorlw   1                 ; if mpx_cnt = 1
btfsc  STATUS,Z
call   dsp_tens           ; display tens digit
movf    mpx_cnt,w
xorlw   2                 ; if mpx_cnt = 2
btfsc  STATUS,Z
call   dsp_mins           ; display minutes digit

; increment mpx_cnt, to select next digit for next time
incf    mpx_cnt,f         ; mpx_cnt = mpx_cnt + 1
; reset count if at end of digit sequence
movf    mpx_cnt,w
xorlw   3                 ; if mpx_cnt = 3
btfsc  STATUS,Z
clrf   mpx_cnt

```

However, since we are testing for successively higher values of mpx\_cnt, we can shorten this code by taking a different approach – subtract one from a copy of mpx\_cnt (held in W), and compare the result with zero, for each successive test.

This works best if mpx\_cnt is incremented first, so that when we first subtract one from it, using ‘addlw -1’, the result is zero (which we can test for, because ‘addlw’ affects the Z flag).

So we have:

```

; Display current count on 3 x 7-segment displays
; mpx_cnt determines current digit to display
;
banksel mpx_cnt
incf    mpx_cnt,f         ; increment mpx_cnt for next digit
movf    mpx_cnt,w        ; and copy to W
; determine current mpx_cnt by successive subtraction
addlw   -1
btfsc  STATUS,Z         ; if current mpx_cnt = 0
goto   dsp_ones         ; display ones digit
addlw   -1
btfsc  STATUS,Z         ; if current mpx_cnt = 1
goto   dsp_tens         ; display tens digit
clrf   mpx_cnt          ; else mpx_cnt = 2, so reset to 0
goto   dsp_mins         ; and display minutes digit

```

That’s a lot shorter.

Note the use of ‘goto’ here, instead of ‘call’. This is because, when a subroutine returns, execution continues with the next instruction – which we don’t want in this case. We’d actually need a ‘goto’ after each ‘call’, to avoid falling through to the rest of this routine – and that gets messy.

Of course, we'll still need a 'goto' at the end of each display routine, as we'll see below, so in a sense it's messy either way – it's difficult to make assembler as “clean” as a higher-level language, such as 'C'.

Note also that, at the end of the routine we know that the digit to display is “minutes”, and that we've come to the end of the digit sequence, meaning that multiplex counter can be reset; there is no need to test it.

The individual digit display routines are:

```
dsp_ones
    ; display ones digit (using shadow registers)
    Lookup  tb7segA,ones    ; lookup ones pattern for PORTA
    movwf  sPORTA          ; then output it
    Lookup  tb7segC,ones    ; repeat for PORTC
    movwf  sPORTC
    bsf    sONES           ; enable ones display
    goto   dsp_end

dsp_tens
    ; display tens digit
    Lookup  tb7segA,tens    ; output tens digit
    movwf  sPORTA
    Lookup  tb7segC,tens    ; repeat for PORTC
    movwf  sPORTC
    bsf    sTENS           ; enable tens display
    goto   dsp_end

dsp_mins
    ; display minutes digit
    Lookup  tb7segA,mins    ; output minutes digit
    movwf  sPORTA
    Lookup  tb7segC,mins    ; repeat for PORTC
    movwf  sPORTC
    bsf    sMINS           ; enable minutes display

dsp_end
```

There is no need to explicitly turn off the previous digit, since, whenever a new digit pattern is written to PORTA, bits RA0, RA1 and RA4 are always cleared (because the digit pattern tables contain '0's for every bit in PORTA, other than RA2). Thus, all the displays are blanked whenever a new digit is output.

Finally, since we're using shadow registers, we need to copy them to the ports, as before:

```
    ; copy shadow regs to ports
    banksel PORTA
    movf   sPORTA,w
    movwf  PORTA
    movf   sPORTC,w
    movwf  PORTC
```

With all this display code being in the interrupt service routine, all that the main loop has to do is increment the time counters, once per second:

```
main_loop
    ; delay 1 sec
    DelayMS 1000          ; delay 1s
```

```

; increment counters
banksel ones
incf    ones,f          ; increment ones
movlw   .10
xorwf   ones,w         ; if ones overflow,
btfss   STATUS,Z
goto    inc_end
clrf    ones           ; reset ones to 0
incf    tens,f         ; and increment tens
movlw   .6
xorwf   tens,w        ; if tens overflow,
btfss   STATUS,Z
goto    inc_end
clrf    tens           ; reset tens to 0
incf    mins,f         ; and increment minutes
movlw   .10
xorwf   mins,w        ; if minutes overflow,
btfsc   STATUS,Z
clrf    mins           ; reset minutes to 0
inc_end

; repeat forever
pagesel main_loop
goto   main_loop

```

It's simply a matter of incrementing the 'ones' digit as was done for a single digit, checking for overflows and incrementing the higher digits accordingly. The overflow (or *carry*) from seconds to minutes is done by testing for "tens = 6". If you wanted to make this purely a seconds counter, counting from 0 to 999 seconds, you'd simply change this to test for "tens = 10", instead.

After incrementing the time counters, the main loop begins again, with the ISR continuing to display the updated time, in the background.

### **Complete program**

Here is the complete program, incorporating the above code fragments.

One point to note is that TMR0 is never initialised; there's no need, as it simply means that there may be a delay of up to 2 ms before the display begins for the first time, which isn't at all noticeable.

```

;*****
; Description: Lesson 12, example 2 *
; *
; Demonstrates use of timer-based interrupt-driven multiplexing *
; to drive multiple 7-seg displays *
; *
; 3 digit 7-segment LED display: 1 digit minutes, 2 digit seconds *
; counts in seconds 0:00 to 9:59 then repeats, *
; with timing derived from int 4 MHz oscillator *
; *
;*****
; Pin assignments: *
; RA2, RC0-5 - 7-segment display bus (common cathode) *
; RA4 - minutes enable (active high) *
; RA1 - tens enable *
; RA0 - ones enable *
; *
;*****

list          p=16F684
#include      <p16F684.inc>

```

```

#include    <stdmacros-mid.inc>        ; DelayMS - delay in milliseconds

errorlevel -302    ; no "register not in bank 0" warnings
errorlevel -312    ; no "page or bank selection not needed" messages

radix      dec

EXTERN     delay10        ; W x 10ms delay

;***** CONFIGURATION
                ; ext reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer, int clock with I/O,
                ; no failsafe clock monitor, two-speed start-up disabled
__CONFIG    _MCLRE_ON & _CP_OFF & _CPD_OFF & _BOD_OFF & _WDT_OFF & _PWRTE_ON
& _INTOSCIO & _FCMEN_OFF & _IESO_OFF

; pin assignments
#define SMINS    SPORTA,4        ; minutes enable (shadow)
#define STENS    SPORTA,1        ; tens enable
#define SONES    SPORTA,0        ; ones enable

;***** MACROS

; Perform table lookup
;
; parameters: table = address of table base ('movwf PCL' instruction)
;               off_var = 8-bit variable holding offset into table
;                   (assumes correct bank already selected)
;
; Table entry at given offset is returned in W
; (first entry is at offset 0)
;
Lookup MACRO    table,off_var

    movlw    HIGH (table+1)    ; load PCLATH with base of table
    movwf    PCLATH
    movlw    LOW (table+1)    ; add offset to base
    addwf    off_var,w
    btfsc    STATUS,C        ; if overflow
    incf    PCLATH,f        ; increment PCLATH
    call    table        ; perform table lookup
    ENDM

;***** VARIABLE DEFINITIONS
CONTEXT    UDATA_SHR        ; variables used for context saving
cs_W      res 1
cs_STATUS  res 1

SHADOW    UDATA_SHR        ; shadow registers
SPORTA    res 1            ; PORTA
SPORTC    res 1            ; PORTC

GENVAR    UDATA            ; general variables
mpx_cnt   res 1            ; multiplex counter
mins      res 1            ; current count: minutes
tens      res 1            ; tens
ones      res 1            ; ones

```

```

;*****
RESET   CODE    0x0000           ; processor reset vector
        pagesel Start
        goto    Start

;***** INTERRUPT SERVICE ROUTINE
ISR     CODE    0x0004
        ; *** Save context
        movwf   cs_W             ; save W
        movf    STATUS,w        ; save STATUS
        movwf   cs_STATUS

        ; *** Service Timer0 interrupt
        ; TMR0 overflows every 2.048 ms
        ; (only Timer0 interrupts are enabled)
        ;
        bcf     INTCON,T0IF      ; clear interrupt flag

        ; Display current count on 3 x 7-segment displays
        ; mpx_cnt determines current digit to display
        ;
        banksel mpx_cnt
        incf    mpx_cnt,f        ; increment mpx_cnt for next digit
        movf    mpx_cnt,w        ; and copy to W
        ; determine current mpx_cnt by successive subtraction
        addlw   -1
        btfsc   STATUS,Z         ; if current mpx_cnt = 0
        goto    dsp_ones        ; display ones digit
        addlw   -1
        btfsc   STATUS,Z         ; if current mpx_cnt = 1
        goto    dsp_tens        ; display tens digit
        clrf    mpx_cnt         ; else mpx_cnt = 2, so reset to 0
        goto    dsp_mins        ; and display minutes digit

dsp_ones
        ; display ones digit (using shadow registers)
        Lookup  tb7segA,ones     ; lookup ones pattern for PORTA
        movwf   sPORTA          ; then output it
        Lookup  tb7segC,ones     ; repeat for PORTC
        movwf   sPORTC
        bsf     sONES            ; enable ones display
        goto    dsp_end

dsp_tens
        ; display tens digit
        Lookup  tb7segA,tens     ; output tens digit
        movwf   sPORTA
        Lookup  tb7segC,tens     ; repeat for PORTC
        movwf   sPORTC
        bsf     sTENS            ; enable tens display
        goto    dsp_end

dsp_mins
        ; display minutes digit
        Lookup  tb7segA,mins     ; output minutes digit
        movwf   sPORTA
        Lookup  tb7segC,mins     ; repeat for PORTC
        movwf   sPORTC
        bsf     sMINS            ; enable minutes display

dsp_end

```

```

    ; copy shadow regs to ports
    banksel PORTA
    movf    sPORTA,w
    movwf   PORTA
    movf    sPORTC,w
    movwf   PORTC

isr_end ; *** Restore context then return
    movf    cs_STATUS,w      ; restore STATUS
    movwf   STATUS
    swapf   cs_W,f          ; restore W
    swapf   cs_W,w
    retfie

;***** MAIN PROGRAM
MAIN    CODE

;***** Initialisation
Start   ; configure ports
    banksel TRISA           ; configure PORTA and PORTC as all outputs
    clrf   TRISA
    clrf   TRISC
    ; configure timer
    movlw  b'11000010'     ; configure Timer0:
    ; --0-----           timer mode (T0CS = 0)
    ; ----0----           prescaler assigned to Timer0 (PSA = 0)
    ; -----010         prescale = 8 (PS = 010)
    banksel OPTION_REG     ; -> increment TMR0 every 8 us
    movwf  OPTION_REG     ; -> TMR0 overflows every 2.048 ms
    ; initialise variables
    banksel mins           ; start with count = 0:00
    clrf   mins
    clrf   tens
    clrf   ones
    clrf   mpx_cnt        ; display ones digit first
    ; enable interrupts
    movlw  1<<GIE|1<<T0IE ; enable Timer0 and global interrupts
    movwf  INTCON

;***** Main loop
main_loop
    ; delay 1 sec
    DelayMS 1000          ; delay 1s

    ; increment counters
    banksel ones
    incf   ones,f         ; increment ones
    movlw  .10
    xorwf  ones,w         ; if ones overflow,
    btfss  STATUS,Z
    goto   inc_end
    clrf   ones           ; reset ones to 0
    incf   tens,f        ; and increment tens
    movlw  .6
    xorwf  tens,w        ; if tens overflow,
    btfss  STATUS,Z
    goto   inc_end
    clrf   tens          ; reset tens to 0
    incf   mins,f        ; and increment minutes
    movlw  .10

```

```

        xorwf    mins,w          ;      if minutes overflow,
        btfsc   STATUS,Z        ;
inc_end  clrf    mins           ;      reset minutes to 0

        ; repeat forever
        pagesel main_loop
        goto   main_loop

;***** LOOKUP TABLES
TABLES  CODE

; Digit to pattern lookup table for 7 segment display on port A
; RA2 = G
tb7segA movwf   PCL
        retlw  b'000000'      ; 0
        retlw  b'000000'      ; 1
        retlw  b'000100'      ; 2
        retlw  b'000100'      ; 3
        retlw  b'000100'      ; 4
        retlw  b'000100'      ; 5
        retlw  b'000100'      ; 6
        retlw  b'000000'      ; 7
        retlw  b'000100'      ; 8
        retlw  b'000100'      ; 9

; Digit to pattern lookup table for 7 segment display on port C
; RC5:0 = ABCDEF
tb7segC movwf   PCL
        retlw  b'111111'      ; 0
        retlw  b'011000'      ; 1
        retlw  b'110110'      ; 2
        retlw  b'111100'      ; 3
        retlw  b'011001'      ; 4
        retlw  b'101101'      ; 5
        retlw  b'101111'      ; 6
        retlw  b'111000'      ; 7
        retlw  b'111111'      ; 8
        retlw  b'111101'      ; 9

        END

```

## Binary-Coded Decimal

In the previous example, each digit in the time count was stored in its own 8-bit register.

Since a single digit can only have values from 0 to 9, while an 8-bit register can store any integer from 0 to 255, it is apparent that storing each digit in a separate variable is an inefficient use of storage space. That can be an issue on devices with such a small amount of data memory – only 128 bytes on the 16F684.

The most space-efficient way to store integers is to use a purely binary representation. For example, the number '183' would be stored in one byte as b'10110111' (or 0xB7). That's three digits in a single byte. Of course, 3-digit numbers larger than 255 need two bytes, but any 4-digit number can be stored in two bytes, as can any 5-digit number less than 65536.

The problem with such "efficient" binary representation is that it's difficult (i.e. time consuming) to unpack into decimal; necessary so that it can be displayed.

Consider how you would convert a number such as 0xB7 into decimal.

First, determine how many hundreds are in it. PIC's don't have a "divide" instruction; the simplest approach is to subtract 100, check to see if there is a borrow, and subtract 100 again if there wasn't (keeping track of the number of hundreds subtracted; this number of hundreds is the first digit):

$$0xB7 - 100 = 0x53$$

Now continue to subtract 10 from the remainder (0x53) until a borrow occurs, keeping track of how many tens were successfully subtracted, giving the second digit:

$$0x53 - (8 \times 10) = 0x03$$

The remainder (0x03) is of course the third digit.

Not only is this a complex routine, and takes a significant time to run (up to 12 subtractions are needed for a single conversion), it also requires storage; intermediate results such as "remainder" and "tens count" need to be stored somewhere.

Sometimes converting from pure binary into decimal is unavoidable, perhaps for example when dealing with quantities resulting from an analog to digital conversion (which we'll look at in the [next lesson](#)). But often, when storing numbers which will be displayed in decimal form, it makes sense to store them using *binary-coded decimal* representation.

In binary-coded decimal, or *BCD*, two digits are *packed* into each byte – one in each nybble (or "nibble").

For example, the BCD representation of 56 is 0x56. That is, each decimal digit corresponds directly to a hex digit when converted to BCD.

All eight bits in the byte are used, although not as efficiently as for binary. But BCD is far easier to work with for decimal operations, as we'll see.

### **Example application**

To demonstrate the use of BCD, we'll modify the previous example to store "seconds" as a BCD variable.

So only two variables for the time count are now needed, instead of three:

```
GENVAR      UDATA                ; general variables
mpx_cnt     res 1                 ; multiplex counter
mins       res 1                 ; current count: minutes
secs       res 1                 ; seconds (BCD)
```

To display minutes is the same as before (since minutes is still being stored in its own variable).

The ones digit is stored as the low nybble (bits 0-3) of 'secs'.

It can be extracted through a technique called *masking*. It relies on the fact that any bit ANDed with '1' remains unchanged, while any bit ANDed with '0' is cleared to '0'. That is:

$$n \text{ AND } 1 = n$$

$$n \text{ AND } 0 = 0$$

If a byte is ANDed with binary 00001111, the high nybble will be cleared, leaving the low nybble unchanged.

This means that we can extract the ones digit by:

```
movf      secs,w                ; get ones digit
andlw    0x0F                  ; from low nybble of seconds
```

The ‘andlw 0x0F’ instruction<sup>4</sup> masks off the high nybble, leaving only the tens digit left in W.

But there’s a problem. Our ‘Lookup’ macro relies on the table offset being stored in a variable, not passed in W. We could modify the table lookup code, so that it accepts the table offset in W, but it would then need to store the offset in a temporary variable, because it is not possible to copy the base address of the table to PCLATH, without overwriting the value in W. Either way, we need to introduce another variable.

The simplest course of action is to keep the existing table lookup code, and use a variable to store the digit extracted from ‘secs’:

```
dsp_ones
    ; display ones digit (using shadow registers)
    movf    secs,w          ; get ones digit
    andlw   0x0F           ; from low nybble of seconds
    movwf   digit
    Lookup  tb7segA,digit   ; lookup ones pattern for PORTA
    movwf   sPORTA        ; then output it
    Lookup  tb7segC,digit   ; repeat for PORTC
    movwf   sPORTC
    bsf     sONES          ; enable ones display
    goto    dsp_end
```

The tens digit is stored as the high nybble of ‘secs’.

To move the contents of bits 4-7 (the high nybble) into bits 0-3 (the low nybble) of a register, you could use four ‘rrf’ instructions, to shift the contents of the register four bits to the right.

But we have already seen an instruction which can do this in a single operation – ‘swapf’, introduced in [lesson 6](#), in the context (excuse the pun) of context saving in interrupt service routine. It is actually a very useful instruction for working with BCD data, since it swaps the nybbles in a register.

So to extract the tens digit, we can use ‘swapf’ to copy the high nybble of ‘secs’ to the low nybble of W, and then mask off the high nybble of W, as before:

```
dsp_tens
    ; display tens digit
    swapf   secs,w         ; get tens digit
    andlw   0x0F           ; from high nybble of seconds
    movwf   digit
    Lookup  tb7segA,digit   ; output tens digit
    movwf   sPORTA
    Lookup  tb7segC,digit
    movwf   sPORTC
    bsf     sTENS          ; enable tens display
    goto    dsp_end
```

Given that we are storing seconds in BCD format, the time count has to be incremented differently, as follows:

```
    ; increment counters
    banksel secs
    incf    secs,f         ; increment seconds
```

---

<sup>4</sup> The *bit mask* is expressed in hexadecimal (0x0F) instead of binary (b’00001111’) here because, when working with BCD values, hexadecimal notation seems clearer.

```

    movf    secs,w           ; if ones overflow,
    andlw  0x0F
    xorlw  .10
    btfss  STATUS,Z
    goto   inc_end
    movlw  .6               ; BCD adjust seconds
    addwf  secs,f
    movlw  0x60
    xorwf  secs,w          ; if seconds = 60,
    btfss  STATUS,Z
    goto   inc_end
    clrf   secs            ; reset seconds to 0
    incf   mins,f          ; and increment minutes
    movlw  .10
    xorwf  mins,w          ; if minutes overflow,
    btfsc  STATUS,Z
    clrf   mins            ; reset minutes to 0
inc_end

```

To check to see whether the ‘ones’ digit has been incremented past 9, it is extracted (by masking) and tested to see if it equals 10. If it does, then we need to reset the ‘ones’ digit to 0, and increment the ‘tens’ digit. But remember that BCD digits are essentially hexadecimal digits. The ‘tens’ digit is really counting by 16s, as far as the PIC is concerned, which operates purely on binary numbers, regardless of whether we consider them to be in BCD format. If the ‘ones’ digit is equal to 10, then adding 6 to it would take it to 16, which would overflow, leaving ‘ones’ cleared to 0, and incrementing ‘tens’.

Putting it another way, you could say that adding 6 adjusts for BCD digit overflow. Some microprocessors provide a “decimal adjust” instruction, that performs this adjustment. The PIC doesn’t, so we do it manually.

Finally, note that to check for seconds overflow, the test is not for “seconds = 60”, but “seconds = 0x60”, i.e. the value to be compared is expressed in hexadecimal, because seconds is stored in BCD format. Forgetting to express the seconds overflow test in hex would be an easy mistake to make...

The rest of the code is exactly the same as before, so won’t be repeated here (although the source files for all the examples are of course available for download from [www.gooligum.com.au](http://www.gooligum.com.au)).

In this example, although we saved one byte by storing seconds in a single byte, there was no net gain, because we had to introduce a variable to store the extracted digit; the code was made more complex for no apparent benefit. However, when you need to work with more data than this, you will find that you can use data memory space more efficiently if you use BCD representation, compared with storing each digit in a separate register, by expanding on the techniques presented above.

Now that we have developed an ability to display numeric values, we have the basic display capability we need to demonstrate analog-to-digital conversion, to be introduced in the [next lesson](#).