Introduction to PIC Programming

Baseline Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 8: Driving 7-Segment Displays

The applications we've looked at so far have used only one or two LEDs as outputs. That's enough for 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.

To drive a single 7-segment display, in a straightforward manner, we need seven outputs. That rules out the PIC12F509 we've been examining so far – but its bigger brother, the 14-pin 16F505, is quite suitable. In fact, the 16F505 can be made to drive up to four 7-segment displays, using a technique known as *multiplexing*. But to display even a single digit, that digit has to be translated into a specific pattern of segments in the display. That translation is normally done through *lookup tables*.

In summary, this lesson covers:

- The PIC16F505 MCU
- Driving a single 7-segment display
- Using lookup tables
- Using multiplexing to drive multiple displays
- Binary-coded decimal (BCD)

Introducing the PIC16F505

The 16F505 is a larger, faster variant of the 12F508 and 12F509 MCUs described in the early lessons.

Lesson 1 included the following table:

Device	Program Memory (words)	Data Memory (bytes)	Package	I/O pins	Clock rate (maximum)
12F508	512	25	8-pin	6	4 MHz
12F509	1024	41	8-pin	6	4 MHz
16F505	1024	72	14-pin	12	20 MHz

Although the 16F505 is architecturally very similar to the 12F508/509, it has more data memory, more I/O pins (11 I/O and 1 input-only), a higher maximum clock speed and wider range of clock options.

The expanded capabilities of the 16F505 are detailed in the following sections.

Additional clock options

In addition to a higher clock rate, the 16F505 supports an expanded range of clock options, selected by three FOSC bits in the configuration word:

Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-	-	-	-	-	-	MCLRE	$\overline{\text{CP}}$	WDTE	FOSC2	FOSC1	FOSC0

The three FOSC bits allow the selection of eight clock options (twice the number available in the 12F508/509), as in the table below.

FOSC<2:0>	Standard symbol	Oscillator configuration
000	`_LP_OSC`	LP oscillator
001	`_XT_OSC`	XT oscillator
010	`_HS_OSC`	HS oscillator
011	`_EC_RB4EN`	EC oscillator + RB4
100	`_IntRC_OSC_RB4EN`	Internal RC oscillator + RB4
101	`_IntRC_OSC_CLKOUTEN`	Internal RC oscillator + CLKOUT
110	`_ExtRC_OSC_RB4EN`	External RC oscillator + RB4
111	`_ExtRC_OSC_CLKOUTEN`	External RC oscillator + CLKOUT

The 'LP' and 'XT' oscillator options are exactly the same as described in lesson 7: 'LP' mode being typically used to drive crystals with a frequency less than 200 kHz, and 'XT' mode being intended for crystals or resonators with a frequency between 200 kHz and 4 MHz.

The 'HS' ("high speed") mode extends this to 20 Mhz. The crystal or resonator, with appropriate loading capacitors, is connected between the OSC1 and OSC2 pins in exactly the same way as for the 'LP' or 'XT' modes.

As explained in lesson 7, the 'LP' and 'XT' (and indeed 'HS') modes can be used with an external clock signal, driving the OSC1, or CLKIN, pin. The downside to using the "crystal" modes with an external clock is that the OSC2 pin remains unused, wasting a potentially valuable I/O pin.

The 'EC' oscillator mode addresses this problem. It is designed for use with an external clock signal driving the CLKIN pin, the same as is possible in the crystal modes, but with the significant advantage that the "OSC2 pin", pin 3 on the 16F505, is available for digital I/O as pin 'RB4'.

The internal RC oscillator on the 16F505 runs at a nominal 4 MHz, the same as that on the 12F508/509, but there are now two options. In the '`_IntRC_OSC_RB4EN`' mode, pin 3 is available for digital I/O as RB4.

The second internal RC option, '`_IntRC_OSC_CLKOUTEN`', assigns pin 3 as 'CLKOUT' instead of RB4. In this mode, the instruction clock (which runs at one quarter the speed of the processor clock, i.e. a nominal 1 MHz), is output on the CLKOUT pin. This output clock signal can be used to provide a clock signal to external devices, or for synchronising other devices with the PIC.

Lesson 7 showed how an external RC oscillator can be used with the 12F508/509. Although this mode usefully allows for low cost, low power operation, it has the same drawback as the externally-clocked "crystal" modes: pin 3 (OSC2) cannot be used for anything.

The external RC oscillator modes on the 16F505 overcome this drawback. In the first option, '`_ExtRC_OSC_RB4EN`', pin 3 is available for digital I/O as RB4.

The other external RC option, '`_ExtRC_OSC_CLKOUTEN`', assigns pin 3 to CLKOUT, with the instruction clock appearing as an output signal, running at one quarter the rate of the external RC oscillator (FOSC/4).

In summary, the expanded range of clock options provides for higher speed operation, more usable I/O pins, or a clock output to allow for external device synchronisation.

Additional I/O pins

The 16F505 provides twelve I/O pins (one being input-only), compared with the six (with one being input-only) available on the 12F508/509.

Twelve is too many pins to represent in a single 8-bit register, so instead of a single port named GPIO, the 16F505 has two ports, named PORTB and PORTC.

Six I/O pins are allocated to each port:

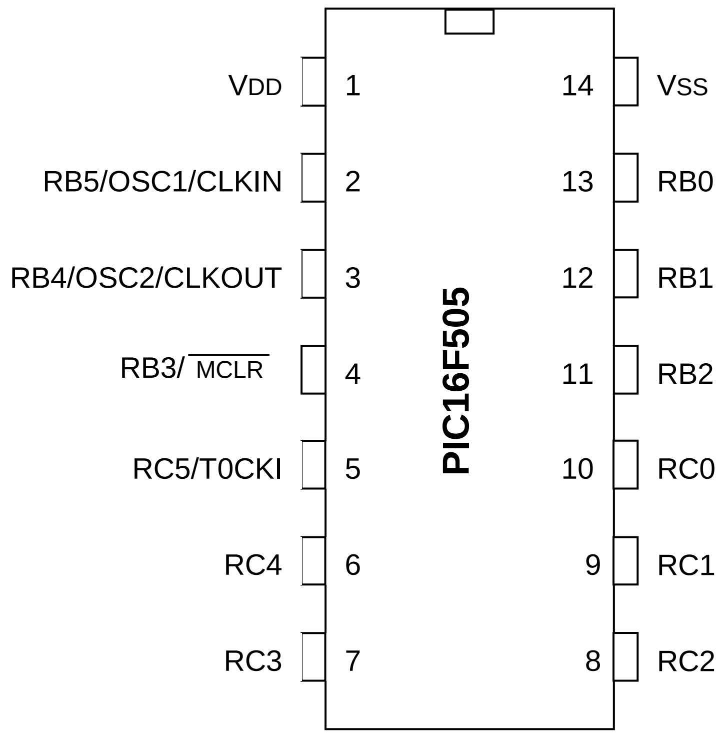
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PORTB			RB5	RB4	RB3	RB2	RB1	RB0
PORTC			RC5	RC4	RC3	RC2	RC1	RC0

The direction of each I/O pin is controlled by corresponding TRIS registers:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TRISB			RB5	RB4		RB2	RB1	RB0
TRISC			RC5	RC4	RC3	RC2	RC1	RC0

As in the 12F508/509, the TRIS registers are not mapped into data memory and can only be accessed through the '`tris`' instruction, with an operand of 6 (or '`PORTB`') to load TRISB, or an operand of 7 (or '`PORTC`') to load TRISC.

RB3 is input only and, like GP3 on the 12F508/509, it shares a pin with $\overline{\text{MCLR}}$; the pin assignment being controlled by the MCLRE bit in the configuration word.



The 16F505 comes in a 14-pin package; the pin diagram is shown on the left.

Note that RC5 and T0CKI (the Timer0 external clock input) share the same pin.

We have seen that on the 12F508/509, T0CKI shares a pin with GP2, and to use GP2 as an output you must first disable T0CKI by clearing the T0CS bit in the OPTION register.

In the same way, to use RC5 as an output on the 16F505, you must first disable T0CKI by clearing T0CS.

Additional data memory

The data memory, or register file, on the 16F505 is arranged in four banks, as follows:

PIC16F505 Registers

	Bank 0		*Bank 1*		*Bank 2*		*Bank 3*
00h	INDF	20h	INDF	40h	INDF	60h	INDF
01h	TMR0	21h	TMR0	41h	TMR0	61h	TMR0
02h	PCL	22h	PCL	42h	PCL	62h	PCL
03h	STATUS	23h	STATUS	43h	STATUS	63h	STATUS
04h	FSR	24h	FSR	44h	FSR	64h	FSR
05h	OSCCAL	25h	OSCCAL	45h	OSCCAL	65h	OSCCAL
06h	PORTB	26h	PORTB	46h	PORTB	66h	PORTB
07h	PORTC	27h	PORTC	47h	PORTC	67h	PORTC
08h 0Fh	General Purpose Registers	28h 2Fh	Map to Bank 0 08h – 0Fh	48h 4Fh	Map to Bank 0 08h – 0Fh	68h 6Fh	Map to Bank 0 08h – 0Fh
10h 1Fh	General Purpose Registers	30h 3Fh	General Purpose Registers	50h 5Fh	General Purpose Registers	70h 7Fh	General Purpose Registers

There are 8 shared data registers (08h – 0Fh), which are mapped into all four banks.

In addition, there are 4 × 16 = 64 non-shared (*banked*) data registers, filling the top half of each bank.

Thus, the 16F505 has a total of 8 + 64 = 72 general purpose data registers.

The bank is selected by the FSR<6:5> bits, as was explained in lesson 3. Although an additional bank selection bit is used, compared with the single bit in the 12F509, you don't need to be aware of that; simply use the `banksel` directive in the usual way.

Other than the differences outlined above, the 16F505 is identical to the 12F508/509[1].

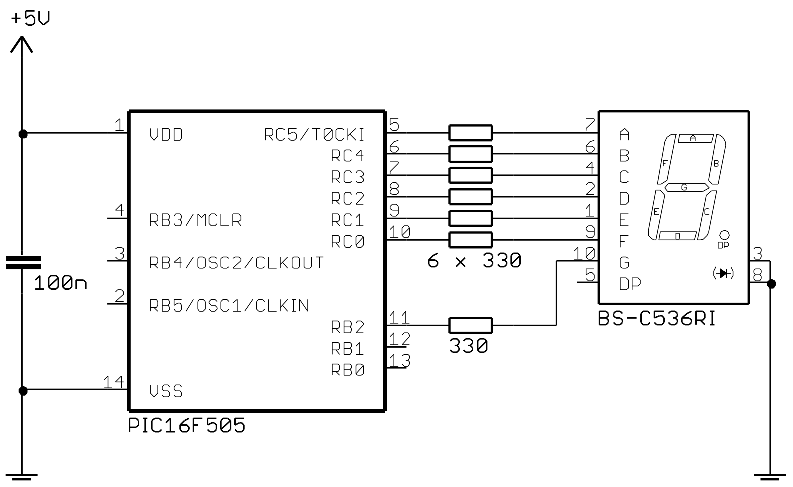
Driving a 7-segment LED Display

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 the "figure 8" pattern we are familiar with from numeric digital displays. 7-segment display modules also commonly include one or two LEDs for decimal points.

[1] For full details, you should of course consult the data sheet

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 following circuit diagram:



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.

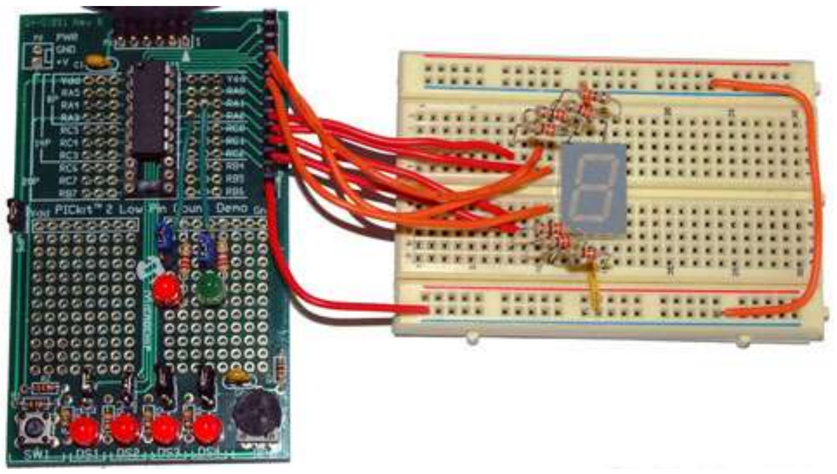
Although a single pin can source or sink up to 25 mA, the maximum for a port is 75 mA and since all segments may be lit at once (when displaying '8'), we need to limit the current per pin to 75 mA ÷ 6 = 12.5 mA. The 330 Ω resistors limit the current to 10 mA, well within spec while giving a bright display.

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. 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.

In the prototype, the display module and resistors were bread-boarded and connected to the 14-pin header on the LPC Demo Board, as illustrated on the right.

Note that the header pins corresponding to the "RB" pins on the 16F505 are labelled "RA" on the demo board, reflecting the PIC16F690 it is supplied with, not the 16F505 used here.



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	RB2
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 with a *lookup table*.

The most common method of implementing lookup tables in the baseline PIC architecture is to use a computed jump into a table of '`retlw`' instructions.

For example, to return the binary pattern to be applied to PORTC, corresponding to the digit in W, we could use 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
```

Baseline PICs have a single addition instruction: '`addwf f,d`' – "**add W** to **f**ile register", placing the result in the register if the destination is '`,f`', or in W if the destination is '`,w`'.

As mentioned in lesson 3, the program counter (PC) is a 12-bit register holding the full address of the next instruction to be executed. The lower eight bits of the program counter (PC<7:0>) are mapped into the PCL register. If you change the contents of PCL, you change the program counter – affecting which instruction will be executed next. For example, if you add 2 to PCL, the program counter will be advanced by 2, skipping the next two instructions.

In the code above, the first instruction adds the table index, or offset (the digit being looked up), in W to PCL, writing the result back to PCL.

If W contains '0', 0 is added to PCL, leaving the program counter unchanged, and the next instruction is executed as normal: the first '`retlw`', returning the pattern for digit '0' in W.

But consider what happens if the subroutine is called with W containing ‘4’. PCL is incremented by 4, advancing the program counter by 4, so the next four instructions will be skipped. The fifth ‘`retlw`’ instruction will be executed, returning the pattern for digit ‘4’ in W.

This lookup table could then be used (‘called’, since it is actually a subroutine) as follows:

```
        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`’)

A second lookup table, called the same way, would be used to lookup the pattern to be output on PORTB.

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 write the code above as:

```
get7sC  addwf   PCL,f
        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  addwf   PCL,f
        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, in which case it is 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.

Lookup table address limitation

A significant limitation of the baseline PIC architecture is that, when any instruction modifies PCL, bit 8 of the program counter (PC<8>) is cleared. That means that, whatever the result of the table offset addition, when PCL is updated, the program counter will be left pointing at an address in the first 256 words of the current program memory page (PC<9> is updated from the PA0 bit, in the same way as for a `goto` or `call` instruction; see lesson 3.)

This is very similar to the address limitation, discussed in lesson 3, which applies to subroutines on baseline PICs. But the constraint on lookup tables is even more limiting – since it is the result of the offset addition that that must be within the first 256 words of a page, not just the start of the table, the whole table has to fit within the first 256 words of a page.

*In the baseline PIC architecture, **lookup tables must be wholly contained within the first 256 locations of a program memory page.***

We have seen that a workaround for the limitation on subroutine addressing is to use a vector table, but no such workaround is possible for lookup tables. Therefore you must take care to ensure that any lookup tables are located toward the beginning of a program memory page. A simple way to do that is to place the lookup tables in a separate code section, located explicitly at the start of a page, by specifying its address with the `CODE` directive.

For example:

```
;***** LOOKUP TABLES
TABLES  CODE    0x200           ; locate at beginning of a page

; Lookup pattern for 7 segment display on port B
get7sB  addwf   PCL,f
        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
```

This places the tables explicitly at the beginning of page 1 (the 16F505 has two program memory pages), out of the way of the start-up code located at the beginning of page 0 (0x000).

This means of course that you need to use the `pagesel` directive if calling these lookup tables from a different code section.

To display a digit, we need to lookup and then write the correct patterns for ports B and C, meaning two table lookups for each digit displayed.

Ideally we’d have a single routine which, given the digit to be displayed, performs the table lookups and writes the patterns to the I/O ports. To avoid the need for multiple `pagesel` directives, this “display digit” subroutine can be located on the same page as the lookup tables.

For example:

```
;***** LOOKUP TABLES
TABLES  CODE    0x200           ; locate at beginning of a page

; Lookup pattern for 7 segment display on port B
get7sB  addwf   PCL,f
        retlw   b'000000'       ; 0

        ...

        retlw   b'000100'       ; 9

; Lookup pattern for 7 segment display on port C
get7sC  addwf   PCL,f
        retlw   b'111111'       ; 0

        ...

        retlw   b'111101'       ; 9

; Display digit passed in 'digit' variable on 7-segment display
set7seg_R
        movf    digit,w         ; get digit to display
        call    get7sB          ; lookup pattern for port B
        movwf   PORTB           ;   then output it
        movf    digit,w         ; repeat for port C
        call    get7sC
        movwf   PORTC
        retlw   0
```

Then to display a digit, it is simply a matter of writing the value into the '`digit`' variable (assumed to be in a shared data segment to avoid the need for banking), and calling the '`set7seg_R`' routine.

Note that it's assumed that the '`set7seg_R`' routine is called through a vector in page 0 labelled '`set7seg`', so that the subroutine doesn't have to be in the first 256 words of page 1; it can be anywhere on page 1 and we still avoid the need for a '`pagesel`' when calling the lookup tables from it.

So, given these lookup tables and a subroutine that will display a selected digit, what to do with them? We've been blinking LEDs at 1 Hz, so counting seconds seems appropriate.

Complete program

The following program incorporates the code fragments presented above, and code (e.g. macros) and techniques from previous lessons, to count repeatedly from 0 to 9, with 1 s between each count.

```
;************************************************************************
;   Description:    Lesson 8, 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:                                                    *
;       RB2, RC0-5 - 7-segment display (common cathode)                 *
;                                                                       *
;************************************************************************
```

```
    list        p=16F505
    #include    <p16F505.inc>
    #include    <stdmacros-base.inc> ; DelayMS - delay in milliseconds

    radix       dec


;***** EXTERNAL LABELS
    EXTERN      delay10_R       ; W x 10ms delay


;***** CONFIGURATION
                ; ext reset, no code protect, no watchdog, 4MHz int clock
    __CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _IntRC_OSC_RB4EN


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


;***** RESET VECTOR *****************************************************
RESET   CODE    0x000           ; effective reset vector
        movwf   OSCCAL          ; update OSCCAL with factory cal value
        pagesel start
        goto    start           ; jump to main program

;***** Subroutine vectors
delay10                         ; delay W x 10ms
        pagesel delay10_R
        goto    delay10_R
set7seg                         ; display digit on 7-segment display
        pagesel set7seg_R
        goto    set7seg_R


;***** MAIN PROGRAM *****************************************************
MAIN    CODE

;***** Initialisation
start
        clrw                    ; configure PORTB and PORTC as all outputs
        tris    PORTB
        tris    PORTC
        movlw   ~(1<<T0CS)      ; disable T0CKI input
        option                  ;   -> RC5 usable

        clrf    digit           ; start with digit = 0

;***** Main loop
count
        pagesel set7seg         ; display digit
        call    set7seg

        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    0x200           ; locate at beginning of a page

; Lookup pattern for 7 segment display on port B
get7sB  addwf   PCL,f
        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

; Lookup pattern for 7 segment display on port C
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

; Display digit passed in 'digit' variable on 7-segment display
set7seg_R
        movf    digit,w         ; get digit to display
        call    get7sB          ; lookup pattern for port B
        movwf   PORTB           ;   then output it
        movf    digit,w         ; repeat for port C
        call    get7sC
        movwf   PORTC
        retlw   0


        END
```

Multiplexing

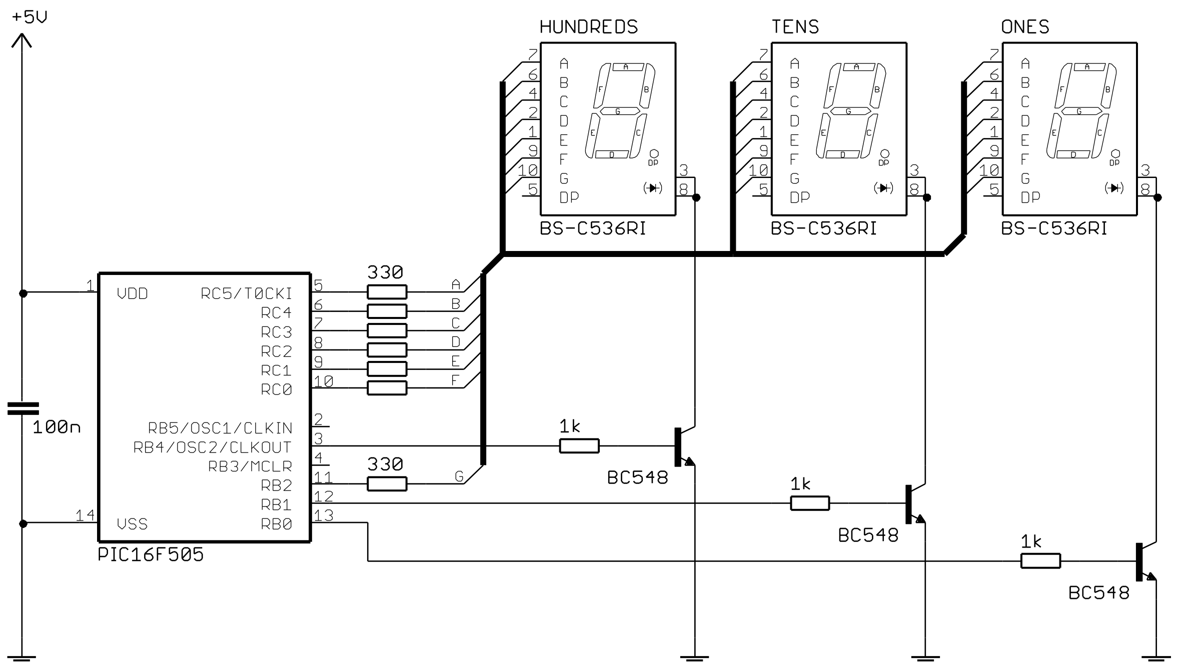
To display multiple digits, as in (say) a digital clock, the obvious approach is to extend the method used above for a single digit. That is, where one digit requires 7 outputs, two digits would apparently need 14 outputs; four digits would need 28 outputs, etc. At that rate, you would very quickly run 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[2] 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. That's an ideal application for timer-based interrupts – a feature available on more advanced PICs (as we will see in midrange lesson 12), but not baseline devices like the 16F505.

[2] 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.

But a timer can still be used to good advantage when implementing multiplexing on a baseline PIC. It would be impractical to try to use programmed delays while multiplexing; there's too much going on. But Timer0 can provide a steady *tick* that we can base our timing on – displaying each digit for a single tick, and then counting ticks to decide when a certain time (e.g. 1 s) has elapsed and we need to perform an action (such as incrementing counters).

If the tick period is too short, there may not be enough time to complete all the program logic needed between ticks, but if it's too long, the display will flicker.

Example application

To demonstrate display multiplexing, we'll extend the example above to count seconds, but instead of counting to 999 seconds, the first digit will count minutes (despite being labelled "hundreds" in the circuit diagram above) and the next two digits will count seconds (00 to 59).

Many PIC developers use a standard 1 ms tick, but to simplify the task of counting in seconds, an (approximately) 2 ms tick is used in this example. 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 ÷ 6 ms = 167 Hz – fast enough to avoid perceptible flicker.

To generate an approximately 2 ms tick, we can use Timer0 in timer mode (based on the 1 MHz instruction clock), with a prescale ratio of 1:256. Bit 2 of Timer0 (TMR0<2>) will then be changing with a period of 2048 µs.

In pseudo-code, the multiplexing technique used here is:

```
time = 0:00
loop
      tick count = 0
      repeat
            display minutes digit for 1 tick (2 ms)
            display tens digit for 1 tick
            display ones digit for 1 tick
      until tick count = #ticks in 1 second

      increment time by 1 second
goto loop
```

To store the time, the simplest approach is to use three variables, to store the minutes, tens and ones digits separately. Setting the time to zero then means clearing each of these variables.

To display a single digit, such as minutes, the code becomes:

```
        ; display minutes for 2.048ms
w60_hi  btfss   TMR0,2          ; wait for TMR0<2> to go high
        goto    w60_hi
        movf    mins,w          ; output minutes digit
        pagesel set7seg
        call    set7seg
        pagesel $
        bsf     MINUTES         ; enable minutes display
w60_lo  btfsc   TMR0,2          ; wait for TMR<2> to go low
        goto    w60_lo
```

This routine begins by waiting for TMR0<2> to go high, then displays the minutes digit (with the others turned off), and finally waits for TMR0<2> to go low again.

The routine to display the tens digit also begins with a wait for TMR0<2> to go high:

```
        ; display tens for 2.048ms
w10_hi  btfss   TMR0,2          ; wait for TMR0<2> to go high
        goto    w10_hi
        movf    tens,w          ; output tens digit
        pagesel set7seg
        call    set7seg
        pagesel $
        bsf     TENS            ; enable tens display
w10_lo  btfsc   TMR0,2          ; wait for TMR<2> to go low
        goto    w10_lo
```

There is no need to explicitly turn off the minutes digit, since, whenever a new digit pattern is output by the ‘`set7seg`’ routine, RB0, RB1 and RB4 are always cleared (because the digit pattern tables contain ‘0’s for every bit in PORTB, other than RB2). Thus, all the displays are blanked whenever a new digit is output.

The ones digit is then displayed in the same way:

```
        ; display ones for 2.048ms
w1_hi   btfss   TMR0,2          ; wait for TMR0<2> to go high
        goto    w1_hi
        movf    ones,w          ; output ones digit
        pagesel set7seg
        call    set7seg
        pagesel $
        bsf     ONES            ; enable ones display
w1_lo   btfsc   TMR0,2          ; wait for TMR<2> to go low
        goto    w1_lo
```

By waiting for TMR0<2> high at the start of each digit display routine, we can be sure that each digit is displayed for exactly 2.048 ms (or, as close as the internal RC oscillator allows, which is only accurate to 1% or so…).

Note that the ‘`set7seg`’ subroutine has been modified to accept the digit to be displayed as a parameter passed in W, instead of placing it a shared variable; it shortens the code a little to do it this way.

Note also the ‘`pagesel $`’ after the subroutine call. It is necessary to ensure that the current page is selected before the ‘`goto`’ commands are executed.

After TMR0<2> goes low at the end of the ‘ones’ display routine, there is approximately 1 ms before it will go high again, when the ‘minutes’ display will be scheduled to begin again. That means that there is a “spare” 1 ms, after the end of the ‘ones’ routine, in which to perform the program logic of counting ticks and incrementing the time counters; 1 ms is 1000 instruction cycles – plenty of time!

The following code construct continues multiplexing the digit display until 1 second has elapsed:

```
; multiplex display for 1 sec
        movlw   1000000/2048/3  ; display each of 3 digits for 2.048ms each
        movwf   mpx_cnt         ;   repeat multiplex loop for 1 second

mplex_loop
        ; display minutes for 2.048ms

        ; display tens for 2.048ms

        ; display ones for 2.048ms

        decfsz  mpx_cnt,f       ; continue to multiplex display
        goto    mplex_loop      ;   until 1s has elapsed
```

Since there are three digits displayed in the loop, and each is displayed for 2 ms (approx.), the total time through the loop is 6 ms, so the number of iterations until 1 second has elapsed is 1 s ÷ 6 ms = 167, small enough to fit into a single 8-bit counter, which is why a tick period of approximately 2 ms was chosen.

Note that, even if the internal RC oscillator was 100% accurate, giving an instruction clock of exactly 1 MHz, the time taken by this loop will be 162 × 3 × 2.048 ms = 995.3 ms. Hence, this "clock" is guaranteed to be out by at least 0.5%. But accuracy isn't the point of this exercise.

After displaying the current time for (close to) 1 second, we need to increment the time counters, and that can be done as follows:

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

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, displaying the updated time.

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 8, example 2                                 *
;                                                                       *
;   Demonstrates use of 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 4MHz oscillator                        *
;                                                                       *
;************************************************************************
;   Pin assignments:                                                    *
;       RB2, RC0-5 - 7-segment display bus (common cathode)             *
;       RB4 - minutes enable (active high)                              *
;       RB1 - tens enable                                               *
;       RB0 - ones enable                                               *
;                                                                       *
;************************************************************************
```

```
    list        p=16F505
    #include    <p16F505.inc>

    radix       dec


;***** CONFIGURATION
                ; ext reset, no code protect, no watchdog, 4 MHz int clock
    __CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _IntRC_OSC_RB4EN

; pin assignments
    #define MINUTES PORTB,4     ; minutes enable
    #define TENS    PORTB,1     ; tens enable
    #define ONES    PORTB,0     ; ones enable


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

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


;***** RESET VECTOR *****************************************************
RESET   CODE    0x000           ; effective reset vector
        movwf   OSCCAL          ; update OSCCAL with factory cal value
        pagesel start
        goto    start           ; jump to main program

;***** Subroutine vectors
set7seg                         ; display digit on 7-segment display
        pagesel set7seg_R
        goto    set7seg_R


;***** MAIN PROGRAM *****************************************************
MAIN    CODE

;***** Initialisation
start
        clrw                    ; configure PORTB and PORTC as all outputs
        tris    PORTB
        tris    PORTC
        movlw   b'11010111'     ; configure Timer0:
                ; --0-----        timer mode (T0CS = 0) -> RC5 usable
                ; ----0---        prescaler assigned to Timer0 (PSA = 0)
                ; -----111        prescale = 256 (PS = 111)
        option                  ;   -> increment every 256 us
                                ;      (TMR0<2> cycles every 2.048ms)

        banksel mins            ; start with count=0
        clrf    mins
        clrf    tens
        clrf    ones
```

```
;***** Main loop
main_loop

; multiplex display for 1 sec
        movlw   1000000/2048/3  ; display each of 3 digits for 2.048ms each
        movwf   mpx_cnt         ;   repeat multiplex loop for approx 1 second

mplex_loop
        ; display minutes for 2.048ms
w60_hi  btfss   TMR0,2          ; wait for TMR0<2> to go high
        goto    w60_hi
        movf    mins,w          ; output minutes digit
        pagesel set7seg
        call    set7seg
        pagesel $
        bsf     MINUTES         ; enable minutes display
w60_lo  btfsc   TMR0,2          ; wait for TMR<2> to go low
        goto    w60_lo

        ; display tens for 2.048ms
w10_hi  btfss   TMR0,2          ; wait for TMR0<2> to go high
        goto    w10_hi
        movf    tens,w          ; output tens digit
        pagesel set7seg
        call    set7seg
        pagesel $
        bsf     TENS            ; enable tens display
w10_lo  btfsc   TMR0,2          ; wait for TMR<2> to go low
        goto    w10_lo

        ; display ones for 2.048ms
w1_hi   btfss   TMR0,2          ; wait for TMR0<2> to go high
        goto    w1_hi
        movf    ones,w          ; output ones digit
        pagesel set7seg
        call    set7seg
        pagesel $
        bsf     ONES            ; enable ones display
w1_lo   btfsc   TMR0,2          ; wait for TMR<2> to go low
        goto    w1_lo

        decfsz  mpx_cnt,f       ; continue to multiplex display
        goto    mplex_loop      ;   until 1 sec has elapsed

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

        goto    main_loop       ; repeat forever


;***** LOOKUP TABLES ****************************************************
TABLES  CODE    0x200           ; locate at beginning of a page

; Lookup pattern for 7 segment display on port B
get7sB  addwf   PCL,f
        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

; Lookup pattern for 7 segment display on port C
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

; Display digit passed in W on 7-segment display
set7seg_R
        movwf   digit           ; save digit
        call    get7sB          ; lookup pattern for port B
        movwf   PORTB           ;   then output it
        movf    digit,w         ; get digit
        call    get7sC          ;   then repeat for port C
        movwf   PORTC
        retlw   0


        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 72 bytes on the 16F505.

The most space-efficient way to store integers is to use pure binary representation. E.g. the number '183' would be stored in a single 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. Baseline PIC's do not 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 × 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 lesson 10). 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", as Microchip spells it).

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:

```
        UDATA
mpx_cnt res 1                   ; multiplex counter
mins    res 1                   ; time 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), but to display the tens digit, we must first extract the digit from the high nybble, as follows:

```
        ; display tens for 2.048ms
w10_hi  btfss   TMR0,2          ; wait for TMR0<2> to go high
        goto    w10_hi
        swapf   secs,w          ; get tens digit
        andlw   0x0F            ;   from high nybble of seconds
        pagesel set7seg
        call    set7seg         ;   then output it
        pagesel $
```

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 the baseline PICs provide a very useful instruction for working with BCD: '`swapf f,d`' – "**swap** nybbles in **f**ile register". As usual, '`f`' is the register supplying the value to be swapped, and '`d`' is the destination: '`,f`' to write the swapped result back to the register, or '`,w`' to place the result in W.

Having gotten the tens digit into the lower nybble (in W, since we don't want to change the contents of the '`secs`' variable), the upper nybble has to be cleared, so that only the tens digit is passed to the '`set7seg`' routine.

This is done 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 AND 1 = n

n AND 0 = 0

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

So far we've only seen the exclusive-or instructions, but the baseline PICs provide equivalent instructions for the logical "and" and "or" operations, including '`andlw`', which ANDs a literal value with the contents of W, placing the result in W – "**and l**iteral with **W**".

So the '`andlw 0x0F`' instruction masks off the high nybble, leaving only the tens digit left in W, to be passed to the '`set7seg`' routine. And why express the *bit mask* in hexadecimal (0x0F) instead of binary (b'00001111')? Simply because, when working with BCD values, hexadecimal notation seems clearer.

Extracting the ones digit is simply a masking operation, as the ones digit is already in the lower nybble:

```
        ; display ones for 2.048ms
w1_hi   btfss   TMR0,2          ; wait for TMR0<2> to go high
        goto    w1_hi
        movf    secs,w          ; get ones digit
        andlw   0x0F            ;   from low nybble of seconds
        pagesel set7seg
        call    set7seg         ;   then output it
        pagesel $
```

The only other routine that has to be done differently, due to storing seconds in BCD format, is incrementing the time count, as follows:

```
; increment counters
        incf    secs,f          ; increment seconds
        movf    secs,w          ; if ones overflow,
        andlw   0x0F
        xorlw   .10
        btfss   STATUS,Z
        goto    end_inc
        movlw   .6              ;   BCD adjust seconds
        addwf   secs,f
        movlw   0x60
        xorwf   secs,w          ;   if seconds = 60,
        btfss   STATUS,Z
        goto    end_inc
        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
end_inc
```

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).

That completes our survey of digital I/O with the baseline PIC devices. More is possible, of course, but to go much further in digital I/O, it is better to make the jump to the midrange architecture.

But before doing so, we'll take a look at analog inputs, using comparators (lesson 9) and analog to digital conversion (lesson 10), and for that it's worth staying with the baseline architecture for one more device, the PIC16F506[3].

[3] If you're following this tutorial series and are concerned about the number of different devices being used, that you would need to buy to try the examples, remember that to learn PIC programming and applications effectively, you need to build on these tutorials by designing and building your own projects. For that, you'll end up using a number of PICs; any devices purchased to follow these tutorials are unlikely to be wasted by a keen new PIC developer!