Introduction to PIC Programming

Programming Baseline PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 5: Driving 7-Segment Displays

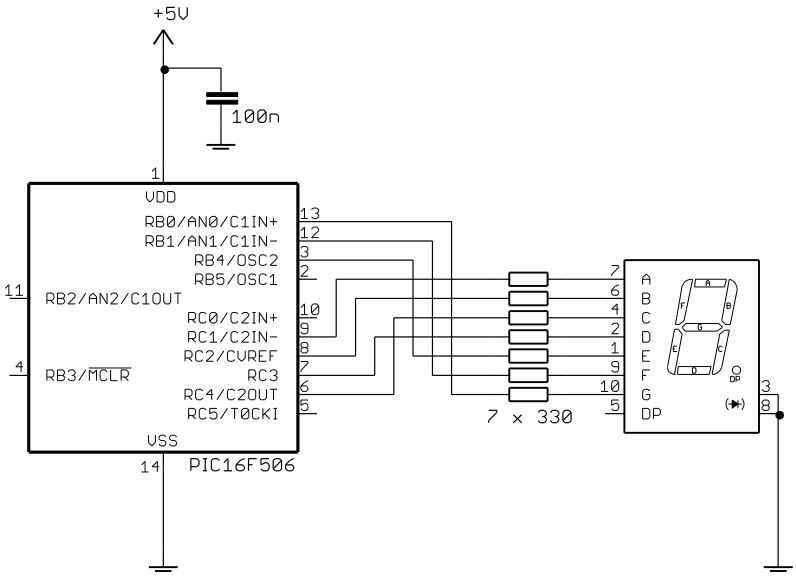
We saw in baseline assembler lesson 8 how to drive 7-segment LED displays, using lookup-tables and multiplexing techniques implemented in assembly language. This lesson shows how C can be used to apply those techniques to drive multiple 7-segment displays, using the Microchip's XC8 (running in "Free mode") and CCS' PCB[1] compilers to re-implement the examples.

In summary, this lesson covers:

- Using lookup tables to drive a single 7-segment display
- Using multiplexing to drive multiple displays

Lookup Tables and 7-Segment Displays

To demonstrate how to drive a single 7-segment display, we will use the circuit from baseline assembler lesson 8, as shown below.



It uses a 16F506 which, as was explained in that lesson, is a 14-pin baseline PIC, with analog inputs (comparators and ADC), more oscillator modes and more data memory, but is otherwise similar to the 12F509 used in the earlier lessons. It provides two 6-pin ports: PORTB and PORTC.

A common-cathode 7-segment LED module is used here. The common-cathode connection is grounded. Each segment is driven, via a 330 Ω resistor, directly from one of the output pins. To light a given segment, the corresponding output is set high.

If a common-anode module is used instead, the anode connection is connected to VDD and the pins become active-low (cleared to zero to make the connected segment light) – you would need to make appropriate changes to the examples below.

[1] XC8 is available as a free download from www.microchip.com, and CCS PCB is bundled for free with MPLAB 8

If you are using the Gooligum baseline training board, you can implement this circuit by:

- placing shunts (six of them) across every position in jumper block JP4, connecting segments A-D, F and G to pins RB0-1 and RC1-4
- placing a single shunt in position 1 ("RA/RB4") of JP5, connecting segment E to pin RB4
- placing a shunt across pins 1 and 2 ("GND") of JP6, connecting digit 1 to ground.

All other shunts should be removed.

As we saw in baseline assembler lesson 8, lookup tables on baseline PICs are normally implemented as a computed jump into a sequence of '`retlw`' instructions, each returning a value corresponding to its position in the table. Care has to be taken to ensure that the table is wholly contained within the first 256 words of a program memory page, and that the page selection bits are set correctly before accessing (calling) the table.

The example program in that lesson implemented a simple seconds counter, displaying each digit from 0 to 9, then repeating, with a 1 s delay between each count.

XC8

In C, a lookup table would usually be implemented as an initialised array. For example:

```
uint8_t days[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

The problem with such a declaration for XC8 is that the compiler has no way to know whether the array contents will change, so it is forced to place such an array in data memory (which even in larger 8-bit PICs is a very limited resource) and add code to initialise the array on program start-up – wasteful of both data and program space.

If, instead, the array is declared as '`const`', the compiler knows that the contents of the array will never change, and so can be placed in ROM (program memory), as a lookup table of `retlw` instructions.

So to create lookup tables equivalent to those in the assembler example in baseline lesson 8, we can write:

```
// pattern table for 7 segment display on port B
const uint8_t pat7segB[10] = {
    // RB4 = E, RB1:0 = FG
    0b010010,   // 0
    0b000000,   // 1
    0b010001,   // 2
    0b000001,   // 3
    0b000011,   // 4
    0b000011,   // 5
    0b010011,   // 6
    0b000000,   // 7
    0b010011,   // 8
    0b000011    // 9
};

// pattern table for 7 segment display on port C
const uint8_t pat7segC[10] = {
    // RC4:1 = CDBA
    0b011110,   // 0
    0b010100,   // 1
    0b001110,   // 2
    0b011110,   // 3
    0b010100,   // 4
    0b011010,   // 5
    0b011010,   // 6
    0b010110,   // 7
```

```
    0b011110,   // 8
    0b011110    // 9
};
```

Looking up the display patterns is easy; the digit to be displayed is used as the array index.

To set the port pins for a given digit, we then have:

```
        PORTB = pat7segB[digit];    // lookup port B and C patterns
        PORTC = pat7segC[digit];
```

This is quite straightforward, and certainly simpler than the assembler version.

However, the assembler example used two tables, one for PORTB, the other for PORTC, to simplify the code for writing the appropriate pattern to each port. In C, it is easier to write more complex expressions, without having to be as concerned by (or even aware of) such implementation details.

In this case, if you were writing the C program for this example from scratch, instead of converting an existing assembler program, it may seem more natural to use a single lookup table with patterns specifying all seven segments of the display, and to then extract the parts of each pattern corresponding to various pins.

For example:

```
// pattern table for 7 segment display on ports B and C
const uint8_t pat7seg[10] = {
    // RC4:1,RB4,RB1:0 = CDBAEFG
    0b1111110,  // 0
    0b1010000,  // 1
    0b0111101,  // 2
    0b1111001,  // 3
    0b1010011,  // 4
    0b1101011,  // 5
    0b1101111,  // 6
    0b1011000,  // 7
    0b1111111,  // 8
    0b1111011   // 9
};
```

Bits 6:3 of each pattern provide the PORTC bits 4:1, so to get the value for PORTC, shift the pattern two bits to the right, and mask off bit 0:

```
        PORTC = (pat7seg[digit] >> 2) & 0b011110;
```

Extracting the bits for PORTB is a little more difficult.

Pattern bit 2 gives the value for RB4. To extract that bit (by ANDing with a single-bit mask) and shift it to position 4 (corresponding to RB4), we can use the expression:

```
        (pat7seg[digit] & 1<<2) << 2
```

Pattern bits 1:0 give the values of PORTB bits 1:0 (RB1 and RB0). We don't need to do any shifting; the bit positions already align, so to extract these bits, we can simply AND them with a mask:

```
        (pat7seg[digit] & 0b00000011)
```

Finally, we need to OR these two expressions together, to build the value to load into PORTB:

```
            PORTB = (pat7seg[digit] & 1<<2) << 2 |
                    (pat7seg[digit] & 0b00000011);
```

Whether you would choose to do this in practice (it seems a bit clumsy here) is partly a matter of personal style, and also a question of whether the space savings, from using only one pattern array, are worth it.

Because we are now using a PIC16F506, instead of the simpler 12F509, there are a couple of differences from our earlier XC8 programs, in configuration and port initialisation, to be aware of.

The 16F506 includes an analog-to-digital converter and two analog comparators. As explained in baseline assembler lesson 8, the analog inputs associated with these peripherals must be disabled before those pins can be used for digital I/O, and this can be done by clearing the ADCON0 register (to deselect all of the ADC inputs) and the C1ON and C2ON bits (to disable the two comparators).

This can be done in XC8 by:

```
    ADCON0 = 0;                     // disable AN0, AN1, AN2 inputs
    CM1CON0bits.C1ON = 0;           //     and comparator 1 -> RB0,RB1 digital
    CM2CON0bits.C2ON = 0;           // disable comparator 2 -> RC1 digital
```

We also saw that the 16F506 supports a wider range of clock options than the 12F509. Since we want to use the internal RC oscillator, with RB4 available for I/O, we need to use the '`OSC_IntRC_RB4EN`' symbol, instead of '`OSC_IntRC`', and include '`IOSCFS_OFF`' (to configure the internal oscillator for 4 MHz operation) in the `__CONFIG()` macro, as follows:

```
// ext reset, no code protect, no watchdog, 4 MHz int clock
__CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & IOSCFS_OFF & OSC_IntRC_RB4EN);
```

For the full list of configuration symbols for the 16F506, see the "`pic16f506.h`" file in the XC8 include directory.

Complete program

Here is the complete single-lookup-table version of this example, for XC8:

```
/************************************************************************
*   Description:    Lesson 5, example 1b                                *
*                                                                       *
*   Demonstrates use of lookup tables to drive a 7-segment display      *
*                                                                       *
*   Single digit 7-segment display counts repeating 0 -> 9              *
*   1 second per count, with timing derived from int 4 MHz oscillator   *
*   (single pattern lookup array)                                       *
*                                                                       *
************************************************************************
*   Pin assignments:                                                    *
*       RB0-1,RB4, RC1-4 = 7-segment display bus (common cathode)       *
*                                                                       *
************************************************************************/

#include <xc.h>
#include <stdint.h>

#define _XTAL_FREQ  4000000     // oscillator frequency for delay functions


/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, 4 MHz int clock
__CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & IOSCFS_OFF & OSC_IntRC_RB4EN);


/***** LOOKUP TABLES *****/

// pattern table for 7 segment display on ports B and C
const uint8_t pat7seg[10] = {
    // RC4:1,RB4,RB1:0 = CDBAEFG
    0b1111110,  // 0
    0b1010000,  // 1
```

```
    0b0111101,  // 2
    0b1111001,  // 3
    0b1010011,  // 4
    0b1101011,  // 5
    0b1101111,  // 6
    0b1011000,  // 7
    0b1111111,  // 8
    0b1111011   // 9
};


/***** MAIN PROGRAM *****/
void main()
{
    uint8_t     digit;              // digit to be displayed

    //*** Initialisation

    // configure ports
    TRISB = 0;                      // configure PORTB and PORTC as all outputs
    TRISC = 0;
    ADCON0 = 0;                     // disable AN0, AN1, AN2 inputs
    CM1CON0bits.C1ON = 0;           //     and comparator 1 -> RB0,RB1 digital
    CM2CON0bits.C2ON = 0;           // disable comparator 2 -> RC1 digital

    //*** Main loop
    for (;;)
    {
        // display each digit from 0 to 9 for 1 sec
        for (digit = 0; digit < 10; digit++)
        {
            // display digit by extracting pattern bits for all pins
            PORTB = (pat7seg[digit] & 1<<2) << 2 |      // RB4
                    (pat7seg[digit] & 0b00000011);      // RB0-1
            PORTC = (pat7seg[digit] >> 2) & 0b011110;   // RC1-4

            // delay 1 sec
            __delay_ms(1000);
        }
    }
}
```

CCS PCB

Like XC8, the CCS PCB compiler also places initialised arrays in program memory, as a table of `retlw` instructions, if the array is declared with the '`const`' qualifier.

Hence, the pattern lookup array is defined in the same way as for XC8.

The expressions for extracting the pattern bits are also the same, since they are standard ANSI syntax. But of course, the statements for assigning those patterns to the port pins are different, because CCS PCB uses built-in functions:

```
            output_b((pat7seg[digit] & 1<<2) << 2 |         // RB4
                     (pat7seg[digit] & 0b00000011));        // RB0-1
            output_c((pat7seg[digit] >> 2) & 0b011110);     // RC1-4
```

As we did in the XC8 version, we need to make some changes to the configuration and port initialisation code, to reflect the fact that we're using a 16F506 instead of a 12F509.

To disable the analog inputs, making all pins available for digital I/O, we can use the built-in functions `setup_comparator()` and `setup_adc_ports()`. We'll see how to use them in lessons 6 and 7, but for now we can simply use:

```
    setup_adc_ports(NO_ANALOGS);    // disable all analog and comparator inputs
    setup_comparator(NC_NC_NC_NC);  // -> RB0, RB1, RC0, RC1 digital
```

We also need to update the `#fuses` statement to use the internal RC oscillator, with RB4 available for I/O, by using the '`INTRC_IO`' symbol instead of '`INTRC`', and to run at 4 MHz, by including the symbol '`IOSC4`', as follows:

```
// ext reset, no code protect, no watchdog, 4 MHz int clock
#fuses MCLR,NOPROTECT,NOWDT,IOSC4,INTRC_IO
```

Note also that to define these symbols, you must include the correct header file for the target PIC – in this case it is "`16F506.h`" (located in the CCS PCB "Devices" directory), where you will find the full list of configuration symbols for the 16F506.

Finally, now that we're using a device with a port B, there is no need for the `#define` statements we used in the 12F509 examples to define GP pin labels.

Complete program

Here is the complete single-table-lookup version of the program, for CCS PCB:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 5, example 1b                                *
*                                                                       *
*   Demonstrates use of lookup tables to drive a 7-segment display      *
*                                                                       *
*   Single digit 7-segment display counts repeating 0 -> 9              *
*   1 second per count, with timing derived from int 4 MHz oscillator   *
*   (single pattern lookup array)                                       *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       RB0-1,RB4, RC1-4 = 7-segment display bus (common cathode)       *
*                                                                       *
************************************************************************/

#include <16F506.h>

#use delay (clock=4000000)      // oscillator frequency for delay_ms()


/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, 4 MHz int clock
#fuses MCLR,NOPROTECT,NOWDT,IOSC4,INTRC_IO


/***** LOOKUP TABLES *****/

// pattern table for 7 segment display on ports B and C
const int8 pat7seg[10] = {
    // RC4:1,RB4,RB1:0 = CDBAEFG
    0b1111110,  // 0
    0b1010000,  // 1
    0b0111101,  // 2
    0b1111001,  // 3
    0b1010011,  // 4
```

```
    0b1101011,  // 5
    0b1101111,  // 6
    0b1011000,  // 7
    0b1111111,  // 8
    0b1111011   // 9
};


/***** MAIN PROGRAM *****/
void main()
{
    unsigned int8   digit;              // digit to be displayed

    //*** Initialisation
    // configure ports
    setup_adc_ports(NO_ANALOGS);    // disable all analog and comparator inputs
    setup_comparator(NC_NC_NC_NC);  // -> RB0, RB1, RC0, RC1 digital


    //*** Main loop
    while (TRUE)
    {
        // display each digit from 0 to 9 for 1 sec
        for (digit = 0; digit < 10; digit++)
        {
            // display digit by extracting pattern bits for all pins
            output_b((pat7seg[digit] & 1<<2) << 2 |         // RB4
                     (pat7seg[digit] & 0b00000011));        // RB0-1
            output_c((pat7seg[digit] >> 2) & 0b011110);     // RC1-4

            // delay 1 sec
            delay_ms(1000);
        }
    }
}
```

Comparisons

The following table summarises the source code length and resource usage for the "single-digit seconds counter" assembly and C example programs, for the versions (assembly and C) with two lookup tables with direct port updates, and the C versions that use a single combined lookup array with more complex pattern extraction for each port.

Count_7seg_x1

Assembler / Compiler	Lookup tables	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	per-port	67	73	4
XC8 (Free mode)	per-port	38	104	4
CCS PCB	per-port	34	92	7
XC8 (Free mode)	combined	27	125	4
CCS PCB	combined	23	98	10

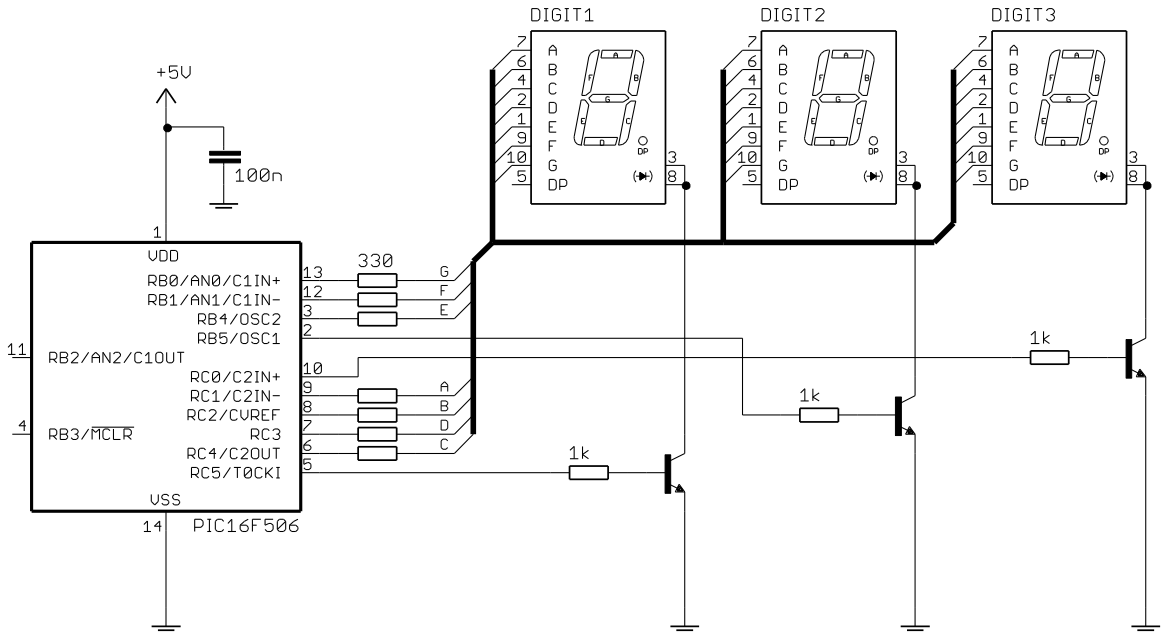
As you can see, the table-per-port C versions are much shorter than the assembler equivalent – and the versions using a single combined lookup table are even shorter. But even with only one table in memory, the C compilers still generate larger code than the two-table version – due to the instructions needed to extract the patterns from each array entry.

In this case, the added complexity of the code needed to extract bit patterns from an combined lookup array isn't worth it – the generated code becomes bigger overall, and the CCS compiler used a lot of extra data memory (presumably to store intermediate results during the extraction process). And the combined-table version is arguably harder to understand. Nevertheless, if the lookup tables were much longer (say 50 entries instead of 10), it would be a different story – the space saved by storing only a single table in memory would more than make up for the extra instructions needed to decode it. Sometimes you simply need to try both ways, to see what's best.

Multiplexing

As explained in more detail in baseline assembler lesson 8, *multiplexing* can used to drive mutiple displays, using a minimal number of output pins. Each display is lit in turn, one at a time, so rapidly that it appears to the human eye that each display is lit continuously.

We'll use the example circuit from that lesson, shown below, to demonstrate how to implement this technique, using C.



To implement this circuit using the Gooligum baseline training board:

- keep the six shunts in every position of jumper block JP4, connecting segments A-D, F and G to pins RB0-1 and RC1-4
- keep the shunt in position 1 ("RA/RB4") of JP5, connecting segment E to pin RB4
- move the shunt in JP6 to across pins 2 and 3 ("RC5"), connecting digit 1 to the transistor controlled by RC5
- place shunts in jumpers JP8, JP9 and JP10, connecting pins RC5, RB5 and RC0 to their respective transistors

All other shunts should be removed.

Each 7-segment display is enabled (able to be lit when its segment inputs are set high) when the NPN transistor connected to its cathode pins is turned on (by pulling the base high), providing a path to ground.

To multiplex the display, each transistor is turned on (by setting high the pin connected to its base) in turn, while outputting the pattern corresponding to that digit on the segment pins, which are wired as a bus.

To ensure that the displays are lit evenly, a timer should be used to ensure that each display is enabled for the same period of time. In the assembler example, this was done as follows:

```
        ; display minutes for 2.048 ms
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
```

Timer0 is used to time the display sequencing; it is configured such that bit 2 cycles every 2.048 ms, providing a regular *tick* to base the multiplex timing on.

Since each display is enabled for 2.048 ms, and there are three displays, the output is refreshed every 6.144 ms, or about 162 times per second – fast enough to appear continuous.

The assembler example implemented a minutes and seconds counter, so the output refresh process was repeated for 1 second (i.e. 162 times), before incrementing the count.

This approach is not 100% accurate (the prototype had a measured accuracy of 0.3% over ten minutes), but given that the timing is based on the internal RC oscillator, which is only accurate to within 1% or so, that's not really a problem.

XC8

In the assembler version of this example (baseline lesson 8, example 2), the time count digits were stored as a separate variables:

```
        UDATA
mins    res 1                   ; current count: minutes
tens    res 1                   ;   tens
ones    res 1                   ;   ones
```

This was done to simplify the assembler code, which, at the end of the main loop, incremented the "ones" variable, and if it overflowed from 9 to 0, incremented "tens" (and on a "tens" overflow from 5 to 0, incremented "minutes").

The next example (baseline lesson 8, example 3) then showed how the seconds value could be stored in a single value, using BCD format to simplify the process of extracting each digit for display:

```
        UDATA
mins    res 1                   ; time count: minutes
secs    res 1                   ;   seconds (BCD)
```

For example, to extract and display the tens digit, we had:

```
        swapf   secs,w          ; get tens digit
        andlw   0x0F            ;   from high nybble of seconds
        pagesel set7seg
        call    set7seg         ;   then output it
```

However, in C it is far more natural to simply store minutes and seconds as ordinary integer variables:

```
uint8_t     mins, secs;         // time counters
```

And then the tens digit would be extracted by dividing seconds by ten, and displayed, as follows:

```
PORTB = pat7segB[secs/10];      // output tens digit
PORTC = pat7segC[secs/10];      //   on display bus
```

Similarly, the ones digit is returned by the simple expression '`secs%10`', which gives the remainder after dividing seconds by ten.

Or course we need some code round that, to wait for TMR0<2> to go high and then low, and to enable the appropriate display module:

```
// display tens for 2.048 ms
while (!(TMR0 & 1<<2))          // wait for TMR0<2> to go high
    ;
PORTB = 0;                      // disable displays
PORTC = 0;
PORTB = pat7segB[secs/10];      // output tens digit
PORTC = pat7segC[secs/10];      //   on display bus
TENS = 1;                       // enable tens display only
while (TMR0 & 1<<2)             // wait for TMR0<2> to go low
    ;
```

This code assumes that the symbol '`TENS`' has been defined:

```
// Pin assignments
#define MINUTES PORTCbits.RC5   // minutes enable
#define TENS    PORTBbits.RB5   // tens enable
#define ONES    PORTCbits.RC0   // ones enable
```

The block of code to display the tens digit has to be repeated with only minor variations for the minutes and ones digits.

This repetition can be reduced in a couple of ways.

The expression '`TMR0 & 1<<2`', used to access TMR0<2>, is a little unwieldy. Since it is used six times in the program (twice for each digit), it makes sense to define it as a macro:

```
#define TMR0_2  (TMR0 & 1<<2)   // access to TMR0<2>
```

The loop which waits for TMR0<2> to go high can then be written more simply as:

```
while (!TMR0_2)         // wait for TMR0<2> to go high
    ;
```

and to wait for it to go low:

```
while (TMR0_2)          // wait for TMR0<2> to go low
    ;
```

More significantly, the code which outputs the digit patterns can be implemented as a function:

```
void set7seg(uint8_t digit)
{
    // pattern table for 7 segment display on port B
    const uint8_t pat7segB[10] = {
        // RB4 = E, RB1:0 = FG
        0b010010,   // 0
        0b000000,   // 1
        0b010001,   // 2
        0b000001,   // 3
        0b000011,   // 4
        0b000011,   // 5
        0b010011,   // 6
        0b000000,   // 7
        0b010011,   // 8
        0b000011    // 9
    };

    // pattern table for 7 segment display on port C
    const uint8_t pat7segC[10] = {
        // RC4:1 = CDBA
        0b011110,   // 0
        0b010100,   // 1
        0b001110,   // 2
        0b011110,   // 3
        0b010100,   // 4
        0b011010,   // 5
        0b011010,   // 6
        0b010110,   // 7
        0b011110,   // 8
        0b011110    // 9
    };

    // Disable displays
    PORTB = 0;                  // clear all digit enable lines on PORTB
    PORTC = 0;                  //  and PORTC

    // Output digit pattern
    PORTB = pat7segB[digit];    // lookup and output port B and C patterns
    PORTC = pat7segC[digit];
}
```

It makes sense to include the pattern table definition within the function, so that the function is self-contained – only the function needs to "know" about the pattern table; it is never accessed directly from other parts of the program. This is very similar to what was done in the assembler examples.

It also makes sense to include the code to disable the displays, prior to outputting a new pattern on the segment bus, within this function, since otherwise it would have to be repeated for each digit.

Displaying the tens digit then becomes:

```
        // display tens for 2.048 ms
        while (!TMR0_2)         // wait for TMR0<2> to go high
            ;
        set7seg(secs/10);       // output tens digit
        TENS = 1;               // enable tens display
        while (TMR0_2)          // wait for TMR0<2> to go low

            ;
```

This is much more concise than before.

To display all three digits of the current count for 1 second, we then have:

```
        // for each time count, multiplex display for 1 second
        // (display each of 3 digits for 2.048 ms each,
        //  so repeat 1000000/2048/3 times to make 1 second)
        for (mpx_cnt = 0; mpx_cnt < 1000000/2048/3; mpx_cnt++)
        {
            // display minutes for 2.048 ms
            while (!TMR0_2)         // wait for TMR0<2> to go high
                ;
            set7seg(mins);          // output minutes digit
            MINUTES = 1;            // enable minutes display
            while (TMR0_2)          // wait for TMR0<2> to go low
                ;

            // display tens for 2.048 ms
            while (!TMR0_2)         // wait for TMR0<2> to go high
                ;
            set7seg(secs/10);       // output tens digit
            TENS = 1;               // enable tens display
            while (TMR0_2)          // wait for TMR0<2> to go low
                ;

            // display ones for 2.048 ms
            while (!TMR0_2)         // wait for TMR0<2> to go high
                ;
            set7seg(secs%10);       // output ones digit
            ONES = 1;               // enable ones display
            while (TMR0_2)          // wait for TMR0<2> to go low
                ;
        }
```

Finally, instead of taking the assembler approach of incrementing all the counters (checking for and reacting to overflows) at the end of an endless loop, it seems much more natural in C to use nested `for` loops:

```
    //*** Main loop
    for (;;)
    {
        // count in seconds from 0:00 to 9:59
        for (mins = 0; mins < 10; mins++)
        {
            for (secs = 0; secs < 60; secs++)
            {
                // for each time count, multiplex display for 1 second

                // display multiplexing loop goes here
            }
        }
    }
```

Complete program

Fitting all this together, including function prototypes, we have:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 5, 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 4 MHz oscillator                       *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       RB0-1,RB4,RC1-4 = 7-segment display bus (common cathode)        *
*       RC5             = minutes enable (active high)                  *
*       RB5             = tens enable                                   *
*       RC0             = ones enable                                   *
*                                                                       *
************************************************************************/

#include <xc.h>
#include <stdint.h>


/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, 4 MHz int clock
__CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & IOSCFS_OFF & OSC_IntRC_RB4EN);

// Pin assignments
#define MINUTES PORTCbits.RC5   // minutes enable
#define TENS    PORTBbits.RB5   // tens enable
#define ONES    PORTCbits.RC0   // ones enable


/***** PROTOTYPES *****/
void set7seg(uint8_t digit);    // display digit on 7-segment display


/***** MACROS *****/
#define TMR0_2  (TMR0 & 1<<2)   // access to TMR0<2>


/***** MAIN PROGRAM *****/
void main()
{
    uint8_t     mpx_cnt;            // multiplex counter
    uint8_t     mins, secs;         // time counters

    //*** Initialisation

    // configure ports
    TRISB = 0;                      // configure PORTB and PORTC as all outputs
    TRISC = 0;
    ADCON0 = 0;                     // disable AN0, AN1, AN2 inputs
    CM1CON0bits.C1ON = 0;           //     and comparator 1 -> RB0,RB1 digital
    CM2CON0bits.C2ON = 0;           // disable comparator 2 -> RC1 digital

    // configure timer
    OPTION = 0b11010111;            // configure Timer0:
             //--0-----                  timer mode (T0CS = 0) -> RC5 usable
             //----0---                  prescaler assigned to Timer0 (PSA = 0)
             //-----111                  prescale = 256 (PS = 111)
             //                          -> increment every 256 us
             //                             (TMR0<2> cycles every 2.048 ms)
```

```
    //*** Main loop
    for (;;)
    {
        // count in seconds from 0:00 to 9:59
        for (mins = 0; mins < 10; mins++)
        {
            for (secs = 0; secs < 60; secs++)
            {
                // for each time count, multiplex display for 1 second
                // (display each of 3 digits for 2.048 ms each,
                //  so repeat 1000000/2048/3 times to make 1 second)
                for (mpx_cnt = 0; mpx_cnt < 1000000/2048/3; mpx_cnt++)
                {
                    // display minutes for 2.048 ms
                    while (!TMR0_2)         // wait for TMR0<2> to go high
                        ;
                    set7seg(mins);          // output minutes digit
                    MINUTES = 1;            // enable minutes display
                    while (TMR0_2)          // wait for TMR0<2> to go low
                        ;

                    // display tens for 2.048 ms
                    while (!TMR0_2)         // wait for TMR0<2> to go high
                        ;
                    set7seg(secs/10);       // output tens digit
                    TENS = 1;               // enable tens display
                    while (TMR0_2)          // wait for TMR0<2> to go low
                        ;

                    // display ones for 2.048 ms
                    while (!TMR0_2)         // wait for TMR0<2> to go high
                        ;
                    set7seg(secs%10);       // output ones digit
                    ONES = 1;               // enable ones display
                    while (TMR0_2)          // wait for TMR0<2> to go low
                        ;
                }
            }
        }
    }
}

/***** FUNCTIONS *****/

/***** Display digit on 7-segment display *****/
void set7seg(uint8_t digit)
{
    // pattern table for 7 segment display on port B
    const uint8_t pat7segB[10] = {
        // RB4 = E, RB1:0 = FG
        0b010010,   // 0
        0b000000,   // 1
        0b010001,   // 2
        0b000001,   // 3
        0b000011,   // 4
        0b000011,   // 5
        0b010011,   // 6
        0b000000,   // 7
        0b010011,   // 8
        0b000011    // 9
    };
```

```
    // pattern table for 7 segment display on port C
    const uint8_t pat7segC[10] = {
        // RC4:1 = CDBA
        0b011110,   // 0
        0b010100,   // 1
        0b001110,   // 2
        0b011110,   // 3
        0b010100,   // 4
        0b011010,   // 5
        0b011010,   // 6
        0b010110,   // 7
        0b011110,   // 8
        0b011110    // 9
    };

    // Disable displays
    PORTB = 0;                  // clear all digit enable lines on PORTB
    PORTC = 0;                  //  and PORTC

    // Output digit pattern
    PORTB = pat7segB[digit];    // lookup and output port B and C patterns
    PORTC = pat7segC[digit];
}
```

CCS PCB

Converting this program for the CCS compiler isn't difficult; it supports the same program structures, such as functions, as the XC8 compiler, and no new features are needed.

Using the `get_timer0()` function, the macro for accessing TMR0<2> would be written as:

```
#define TMR0_2  (get_timer0() & 1<<2)   // access to TMR0<2>
```

Alternatively, as we saw in lesson 3, TMR0<2> could be accessed through a bit variable, declared as:

```
#bit TMR0_2 = 0x01.2                    // access to TMR0<2>
```

The main problem with this approach is that it's not portable – you shouldn't assume that TMR0 will always be at address 01h; if you migrate your code to another PIC, you may have to remember to change this line. On the other hand, the `get_timer0()` function will always work.

Complete program

Most of the code is very similar to the XC8 version, with register accesses replaced with their CCS built-in function equivalents:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 5, 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 4 MHz oscillator                       *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       RB0-1,RB4,RC1-4 = 7-segment display bus (common cathode)        *
```

```
*       RC5             = minutes enable (active high)                  *
*       RB5             = tens enable                                   *
*       RC0             = ones enable                                   *
*                                                                       *
************************************************************************/

#include <16F506.h>


/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, 4 MHz int clock
#fuses MCLR,NOPROTECT,NOWDT,IOSC4,INTRC_IO

// Pin assignments
#define MINUTES PIN_C5              // minutes enable
#define TENS    PIN_B5              // tens enable
#define ONES    PIN_C0              // ones enable


/***** PROTOTYPES *****/
void set7seg(unsigned int8 digit);      // display digit on 7-segment display


/***** MACROS *****/
#define TMR0_2  (get_timer0() & 1<<2)   // access to TMR0<2>


/***** MAIN PROGRAM *****/
void main()
{
    unsigned int8    mpx_cnt;       // multiplex counter
    unsigned int8    mins, secs;    // time counters

    //*** Initialisation

    // configure ports
    setup_adc_ports(NO_ANALOGS);    // disable all analog and comparator inputs
    setup_comparator(NC_NC_NC_NC);  // -> RB0, RB1, RC0, RC1 digital

    // configure Timer0
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_256); // timer mode, prescale = 256
                                               // -> bit 2 cycles every 2.048 ms


    //*** Main loop
    while (TRUE)
    {
        // count in seconds from 0:00 to 9:59
        for (mins = 0; mins < 10; mins++)
        {
            for (secs = 0; secs < 60; secs++)
            {
                // for each time count, multiplex display for 1 second
                // (display each of 3 digits for 2.048 ms each,
                //  so repeat 1000000/2048/3 times to make 1 second)
                for (mpx_cnt = 0; mpx_cnt < 1000000/2048/3; mpx_cnt++)
                {
                    // display minutes for 2.048 ms
                    while (!TMR0_2)         // wait for TMR0<2> to go high
                        ;
                    set7seg(mins);          // output minutes digit
```

```
                    output_high(MINUTES);   // enable minutes display
                    while (TMR0_2)          // wait for TMR0<2> to go low
                        ;

                    // display tens for 2.048 ms
                    while (!TMR0_2)         // wait for TMR0<2> to go high
                        ;
                    set7seg(secs/10);       // output tens digit
                    output_high(TENS);      // enable tens display
                    while (TMR0_2)          // wait for TMR0<2> to go low
                        ;

                    // display ones for 2.048 ms
                    while (!TMR0_2)         // wait for TMR0<2> to go high
                        ;
                    set7seg(secs%10);       // output ones digit
                    output_high(ONES);      // enable ones display
                    while (TMR0_2)          // wait for TMR0<2> to go low
                        ;
                }
            }
        }
    }
}


/***** FUNCTIONS *****/

/***** Display digit on 7-segment display *****/
void set7seg(unsigned int8 digit)
{
    // pattern table for 7 segment display on port B
    const int8 pat7segB[10] = {
        // RB4 = E, RB1:0 = FG
        0b010010,   // 0
        0b000000,   // 1
        0b010001,   // 2
        0b000001,   // 3
        0b000011,   // 4
        0b000011,   // 5
        0b010011,   // 6
        0b000000,   // 7
        0b010011,   // 8
        0b000011    // 9
    };

    // pattern table for 7 segment display on port C
    const int8 pat7segC[10] = {
        // RC4:1 = CDBA
        0b011110,   // 0
        0b010100,   // 1
        0b001110,   // 2
        0b011110,   // 3
        0b010100,   // 4
        0b011010,   // 5
        0b011010,   // 6
        0b010110,   // 7
        0b011110,   // 8
        0b011110    // 9
    };
```

```
    // Disable displays
    output_b(0);                    // clear all digit enable lines on PORTB
    output_c(0);                    //  and PORTC

    // Output digit pattern
    output_b(pat7segB[digit]);      // lookup and output port B and C patterns
    output_c(pat7segC[digit]);
}
```

Comparisons

Here is the resource usage summary for the 3-digit time count example programs, including the BCD version of the assembler example from baseline assembler lesson 8:

Count_7seg_x3

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	118	104	4
XC8 (Free mode)	60	484	11
CCS PCB	56	180	10

Although the C source code is much shorter than the assembler program, the code generated by the C compilers is much bigger than the hand-written assembler version – the CCS version being nearly twice as large, despite having full optimisation enabled. This is mainly because of the apparently simple division and modulus operations used in the C examples. Something may be very easy to express (leading to shorter source code), but be inefficient to implement – and mathematical operations, even simple integer arithmetic, are a classic example.

And without any optimisation, the XC8 compiler (running in 'Free mode') generates very poor code indeed, in this example – nearly five times as big as the assembler version!

Summary

We have seen in this lesson that lookup tables can be effectively implemented in C as initialised arrays qualified as '`const`'. We also saw that C bit-manipulation expressions make it reasonably easy to extract more than one segment display pattern from a single table entry, making it seem natural to use a single lookup table – but that the extra instructions that the compiler generates to perform this pattern extraction may not be worth the savings in lookup table size.

Similarly, we saw that it was quite straightforward to use multiplexing to implement a multi-digit display, without needing to be as concerned (as we were with assembly language) about how to store the values being displayed. This allowed us to use simple arithmetic expressions such as '`secs/10`' and as '`secs%10`', but at a significant cost in generated code size – demonstrating that what seems easy or natural in C, is not always the most efficient way to do something.

Although it would be possible to re-write the C programs so that the compilers can generate more efficient code, to some extent that misses the point of programming in C – it's all about being able to save valuable time.

Of course it is useful, when using C, to be aware of which program structures use more memory or need more instructions to implement than others (such as including floating point calculations when it is not necessary). But if you really need efficiency, as you often will with these small devices, it's difficult to do beat assembler.

The next lesson ventures into analog territory, covering comparators and programmable voltage references (revisiting material from baseline assembler lesson 9).