Introduction to PIC Programming

Programming Baseline PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 4: Driving 7-Segment Displays

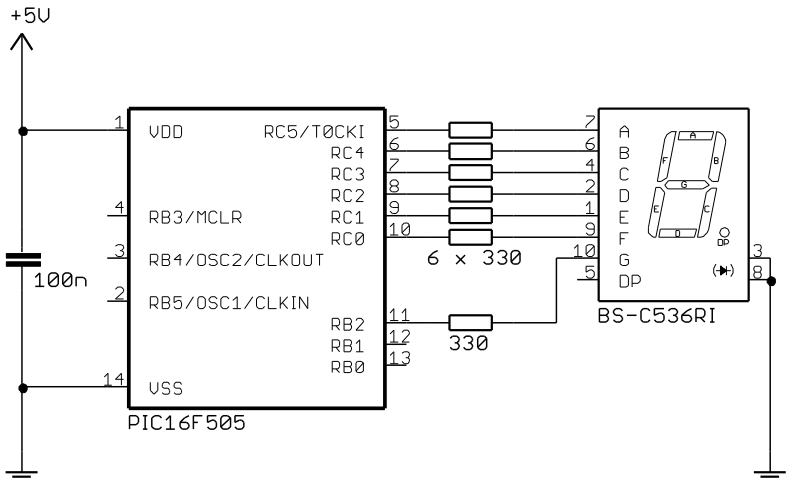
We saw in baseline lesson 8 how to drive 7-segment LED displays, using lookup-tables and multiplexing techniques implemented in assembly language. This lesson shows how C can be used to apply those techniques to drive multiple 7-segment displays, using the free HI-TECH C[1] (in "Lite" mode), PICC-Lite and CCS PCB 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 lesson 8, shown on the right.



It uses a 16F505 which, as was explained in that lesson, is a 14-pin variant of 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. Most will have a different pin-out to that shown, but are all connected to the PIC in the same way. Each segment is driven, via a 330 Ω resistor, directly from one of the output pins. The whole of PORTC is used, plus RB2 from PORTB.

The common-cathode connection is grounded. 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] PICC-Lite was bundled with versions of MPLAB up to 8.10. HI-TECH C (earlier known as "HI-TECH C PRO") was bundled with MPLAB 8.15 and later, although you should download the latest version from www.htsoft.com.

As we saw in baseline 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.

HI-TECH C or PICC-Lite

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

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

The problem with such a declaration for HI-TECH C 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 large 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:

```
// Lookup pattern for 7 segment display on port B
const char pat7segB[10] = {
    // RB2 = G
    0b000000,   // 0
    0b000000,   // 1
    0b000100,   // 2
    0b000100,   // 3
    0b000100,   // 4
    0b000100,   // 5
    0b000100,   // 6
    0b000000,   // 7
    0b000100,   // 8
    0b000100    // 9
};

// Lookup pattern for 7 segment display on port C
const char pat7segC[10] = {
    // RC5:0 = ABCDEF
    0b111111,   // 0
    0b011000,   // 1
    0b110110,   // 2
    0b111100,   // 3
    0b011001,   // 4
    0b101101,   // 5
    0b101111,   // 6
    0b111000,   // 7
    0b111111,   // 8
    0b111101    // 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 being as concerned by (or even aware of) 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 would probably 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:

```
// Lookup pattern for 7 segment display on ports B and C
const char pat7seg[10] = {
    // RC5:0,RB2 = ABCDEFG
    0b1111110,   // 0
    0b0110000,   // 1
    0b1101101,   // 2
    0b1111001,   // 3
    0b0110011,   // 4
    0b1011011,   // 5
    0b1011111,   // 6
    0b1110000,   // 7
    0b1111111,   // 8
    0b1111011    // 9
};
```

Bits 6:1 of each pattern provide the PORTC bits 5:0, so to get the value for PORTC, shift the pattern one bit to the right:

```
        PORTC = pat7seg[digit] >> 1;
```

Pattern bit 0 gives the value for RB2. To derive that value, the pattern is ANDed with a mask, leaving only bit 0:

```
        RB2 = pat7seg[digit] & 0b0000001;
```

There is one other difference from our earlier HI-TECH C programs to be aware of. This example uses the PIC16F505, which, as described in baseline lesson 8, supports a wider range of clock options than the 12F508/509. That means a change in the configuration word setting.

We want to use the internal RC oscillator, with RB4 available for I/O. That means using the '`INTRCRB4`' symbol, instead of '`INTRC`', in the `__CONFIG()` macro.

For the full list of configuration symbols for the 16F505, see the "pic16505.h" file in the PICC-Lite include directory.

Complete program

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

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 4, 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 RC oscillator      *
*   (single pattern lookup array)                                       *
*                                                                       *
************************************************************************
```

```
*   Pin assignments:                                                    *
*       RB2, RC0-5 = 7-segment display (common cathode)                 *
*                                                                       *
************************************************************************/

#include <htc.h>

#define XTAL_FREQ   4MHZ    // oscillator frequency for delay functions
#include "stdmacros-PCL.h"  // DelayS() - delay in seconds


/***** CONFIGURATION *****/
// Config: ext reset, no code protect, no watchdog, 4MHz int clock
__CONFIG(MCLREN & UNPROTECT & WDTDIS & INTRCRB4);


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

// Lookup pattern for 7 segment display on ports B and C
const char pat7seg[10] = {
    // RC5:0,RB2 = ABCDEFG
    0b1111110,   // 0
    0b0110000,   // 1
    0b1101101,   // 2
    0b1111001,   // 3
    0b0110011,   // 4
    0b1011011,   // 5
    0b1011111,   // 6
    0b1110000,   // 7
    0b1111111,   // 8
    0b1111011    // 9
};


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

    // Initialisation
    TRISB = 0;                      // configure PORTB and PORTC as all outputs
    TRISC = 0;
    PORTB = 0;                      // make all PORTB pins low
    OPTION = ~T0CS;                 // disable T0CKI input (enables RC5 output)

    // Main loop
    for (;;) {
        // display each digit from 0 to 9 for 1s
        for (digit = 0; digit < 10; digit++) {
            // display digit
            RB2 = pat7seg[digit] & 0b0000001;   // extract pattern bits
            PORTC = pat7seg[digit] >> 1;        // for each port

            // delay 1 sec
            DelayS(1);
        }
    }
}
```

This makes use of the `DelayS()` macro developed in lesson 2, defined in the external "`stdmacros-PCL.h`" file.

The HI-TECH C PRO version is the same, except that, to make use of its built-in delay functions, we substitute:

```
#define _XTAL_FREQ  4000000     // oscillator frequency for delay functions
#include "stdmacros-HTC.h"      // DelayS() - delay in seconds
```

CCS PCB

Like HI-TECH C, 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 exactly the same way as for HI-TECH C.

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_bit(PIN_B2,pat7seg[digit] & 0b0000001);  // RB2
output_c(pat7seg[digit] >> 1);                  // PORTC
```

Note that pin 2 of PORTB is referred to as '`PIN_B2`'. Although this is different from Microchip's data sheets, which call it RB2, it is nevertheless clear which pin is being referenced, so there is no need for the `#define` statements we used in the 12F509 examples to define GP pin labels.

Note also that to define these symbols, you must include the correct header file for the target PIC – in this case it is "16F505.h".

Timer0 must be placed into timer mode, to disable the T0CKI external counter input, so that pin RC5 can be used as an output. This was done in the HI-TECH C code by clearing the T0CS bit in the OPTION register. But since CCS PCB does not provide direct access to OPTION, we must use a built-in function:

```
setup_timer_0(RTCC_INTERNAL);
```

This selects the internal instruction clock for Timer0, disabling the T0CKI input, without selecting any prescaler or other options.

Finally, we need to update the `#fuses` statement to use the internal RC oscillator, with RB4 available for I/O, which means using the '`INTRC_IO`' symbol instead of '`INTRC`'.

For the full list of configuration symbols for the 16F505, see the "16F505.h" file in the CCS PCB "Devices" directory.

Complete program

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

```
/************************************************************************
*   Description:    Lesson 4, 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 RC oscillator      *
*   (single pattern lookup array)                                       *
*                                                                       *
************************************************************************
*   Pin assignments:                                                    *
*       RB2, RC0-5 = 7-segment display (common cathode)                 *
*                                                                       *
```

```
************************************************************************/

#include <16F505.h>

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


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


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

// Lookup pattern for 7 segment display on ports B and C
const char pat7seg[10] = {
    // RC5:0,RB2 = ABCDEFG
    0b1111110,   // 0
    0b0110000,   // 1
    0b1101101,   // 2
    0b1111001,   // 3
    0b0110011,   // 4
    0b1011011,   // 5
    0b1011111,   // 6
    0b1110000,   // 7
    0b1111111,   // 8
    0b1111011    // 9
};


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

    // Initialisation
    output_b(0);                    // make all PORTB pins low
    setup_timer_0(RTCC_INTERNAL);   // disable T0CKI input (enables RC5 output)

    // Main loop
    while (TRUE) {
        // display each digit from 0 to 9 for 1s
        for (digit = 0; digit < 10; digit++) {
            // display digit by extracting pattern bits for all pins
            output_bit(PIN_B2,pat7seg[digit] & 0b0000001);  // RB2
            output_c(pat7seg[digit] >> 1);                  // PORTC

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

Comparisons

The following table summarises the resource usage for the "single-digit seconds counter" assembler and C example programs. Note however that the assembler example uses two lookup tables with direct port updates, while the C programs use a single lookup array with more complex pattern extraction for each port. You could argue that such a comparison is not valid. However, the purpose of these tutorials is to show how a task would typically be done in each language; different ways to approach a problem may seem more

natural in one language or another. The idea here is to show how each example might typically be implemented in C, without being constrained by what is simplest in assembler.

Count_7seg_x1

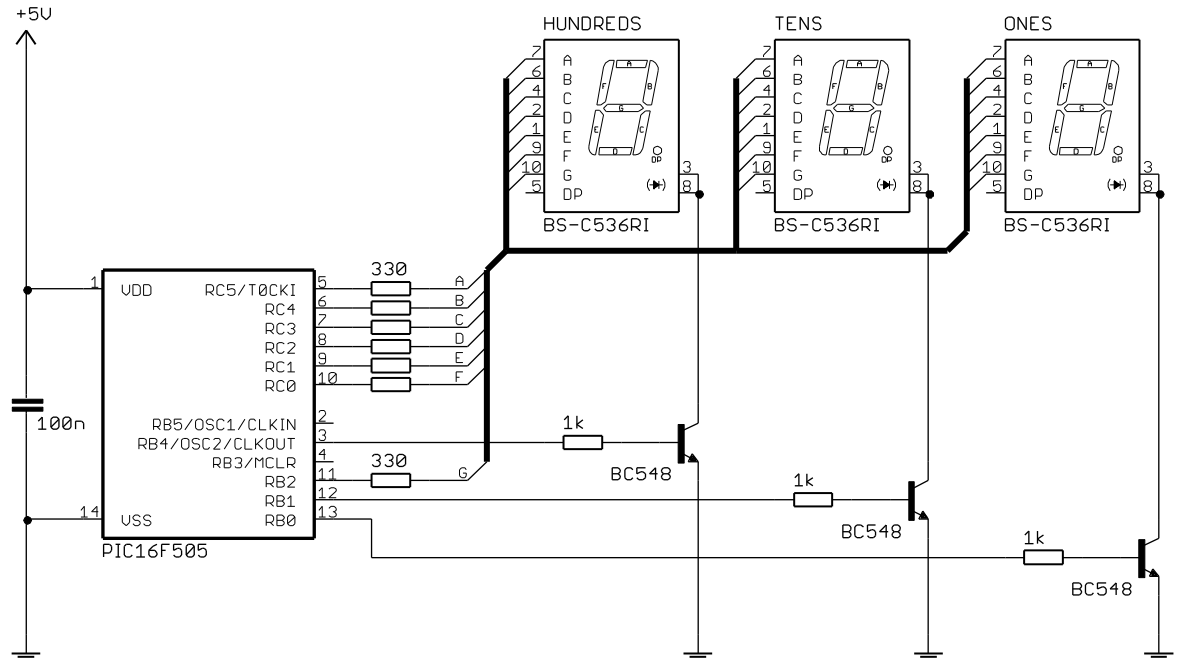
Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	64	71	4
HI-TECH PICC-Lite	26	96	11
HI-TECH C PRO Lite	26	122	4
CCS PCB	23	86	7

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

Multiplexing

As explained in more detail in baseline 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 lesson 8, shown below, to demonstrate how to implement this technique, using C.



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. [For common-anode displays, PNP transistors would be used between the anodes connections and VDD.]

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 example in lesson 8 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 2%, that's not really a problem.

HI-TECH C PRO or PICC-Lite

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:

```
unsigned char    mins, secs;    // time counters
```

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

```
RB2 = pat7seg[secs/10] & 0b0000001;   // output tens digit
PORTC = pat7seg[secs/10] >> 1;        //   on segment 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 all displays
RB2 = pat7seg[secs/10] & 0b0000001;   // output tens digit
PORTC = pat7seg[secs/10] >> 1;        //   on segment 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 RB4             // minutes enable
#define TENS    RB1             // tens enable
#define ONES    RB0             // 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 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(char digit)
{
    // Lookup pattern table for 7-segment display on ports B and C
    const char pat7seg[10] = {
        // RC5:0,RB2 = ABCDEFG
        0b1111110,   // 0
        0b0110000,   // 1
        0b1101101,   // 2
        0b1111001,   // 3
        0b0110011,   // 4
        0b1011011,   // 5
        0b1011111,   // 6
        0b1110000,   // 7
        0b1111111,   // 8
        0b1111011    // 9
    };

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

    // Extract pattern bits and write to segment bus pins
    RB2 = pat7seg[digit] & 0b0000001;
    PORTC = pat7seg[digit] >> 1;
}
```

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 significantly 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 2.048ms each,
        //  so repeat 1000000/2048/3 times to make 1 second)
        for (mpxcnt = 0; mpxcnt < 1000000/2048/3; mpxcnt++) {

            // 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 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 4, 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 (active high)               *
*       RB4         = minutes enable (active high)                      *
*       RB1         = tens enable                                       *
*       RB0         = ones enable                                       *
*                                                                       *
************************************************************************/
```

```
#include <htc.h>


/***** CONFIGURATION *****/
// Config: ext reset, no code protect, no watchdog, 4MHz int clock
__CONFIG(MCLREN & UNPROTECT & WDTDIS & INTRCRB4);

// Pin assignments
#define MINUTES RB4             // minutes enable
#define TENS    RB1             // tens enable
#define ONES    RB0             // ones enable


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


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


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

    // Initialisation
    TRISB = 0;                      // configure PORTB and PORTC as all outputs
    TRISC = 0;
    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 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(char digit)
{
    // Lookup pattern table for 7-segment display on ports B and C
    const char pat7seg[10] = {
        // RC5:0,RB2 = ABCDEFG
        0b1111110,   // 0
        0b0110000,   // 1
        0b1101101,   // 2
        0b1111001,   // 3
        0b0110011,   // 4
        0b1011011,   // 5
        0b1011111,   // 6
        0b1110000,   // 7
        0b1111111,   // 8
        0b1111011    // 9
    };

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

    // Extract pattern bits and write to segment bus pins
    RB2 = pat7seg[digit] & 0b0000001;
    PORTC = pat7seg[digit] >> 1;
```

CCS PCB

Converting this program for the CCS compiler isn't difficult; it supports the same program structures, such as functions, as the HI-TECH 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 2, TMR0<2> could be accessed through a bit variable, declared as:

```
#bit TMR0_2 = 0x01.7                    // 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 HI-TECH version, with register accesses replaced with their CCS built-in function equivalents:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 4, 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 (active high)               *
*       RB4         = minutes enable (active high)                      *
*       RB1         = tens enable                                       *
*       RB0         = ones enable                                       *
*                                                                       *
************************************************************************/

#include <16F505.h>


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

// Pin assignments
#define MINUTES PIN_B4          // minutes enable
#define TENS    PIN_B1          // tens enable
#define ONES    PIN_B0          // ones enable


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


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


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

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

```
    // Main loop
    while (TRUE)
    {
        // count 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 2.048ms 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(char digit)
{
    // Lookup pattern table for 7-segment display on ports B and C
    const char pat7seg[10] = {
        // RC5:0,RB2 = ABCDEFG
        0b1111110,   // 0
        0b0110000,   // 1
        0b1101101,   // 2
        0b1111001,   // 3
        0b0110011,   // 4
        0b1011011,   // 5
        0b1011111,   // 6
        0b1110000,   // 7
        0b1111111,   // 8
        0b1111011    // 9
    };
```

```
    // Disable displays
    output_b(0);                    // clear all enable lines on PORTB

    // Extract pattern bits and write to display bus pins
    output_bit(PIN_B2,pat7seg[digit] & 0b0000001);  // RB2
    output_c(pat7seg[digit] >> 1);                  // PORTC
}
```

Comparisons

Here is the resource usage summary for the 3-digit time count example programs, including both the separate-variable-per-digit and BCD assembler versions:

Count_7seg_x3

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM (non-BCD)	110	96	5
Microchip MPASM (BCD)	111	98	4
HI-TECH PICC-Lite	44	185	13
HI-TECH C PRO Lite	44	425	12
CCS PCB	42	164	11

The two assembler versions are very similar – storing the seconds count in BCD format saved one byte of data memory, at the expense of using two more words of program memory. But this difference pales in comparison with the C programs.

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 PICC-Lite 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 HI-TECH C PRO compiler (running in 'lite' mode) generates very poor code indeed, in this example – more than four 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`', and that through the use of C expressions is it simple to extract more than one segment display pattern from a single table entry, making it seem natural to use a single lookup table.

We also 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 assembler) about how to store the values being displayed, using simple arithmetic expressions such as '`secs/10`' and as '`secs%10`'.

Thus, both examples could be expressed very succinctly in C, using either compiler, compared with the assembler versions (the BCD assembler version is the basis for comparison in example 2), as shown on the table on next page:

Source code (lines)

Assembler / Compiler	Count_7seg_x1	Count_7seg_x3
Microchip MPASM	64	111
HI-TECH PICC-Lite	26	44
HI-TECH C PRO Lite	26	44
CCS PCB	23	42

Now that the examples are becoming a little more complex, the C source code is becoming very significantly shorter than the assembler versions – less than half the length. Although the CCS code is slightly shorter (as usual) than that for HI-TECH C, there is very little difference.

However, there is a potential cost associated with writing what seems to be short, simple code in C. For example, it is easy to write an expression like '`secs/10`', without appreciating that this means that the compiler has to generate code to perform a division, which is not very efficient. So we're now seeing a very clear trade-off between ease of coding (shorter source code) and resource usage efficiency:

Program memory (words)

Assembler / Compiler	Count_7seg_x1	Count_7seg_x3
Microchip MPASM	71	98
HI-TECH PICC-Lite	96	185
HI-TECH C PRO Lite	122	425
CCS PCB	86	164

Data memory (bytes)

Assembler / Compiler	Count_7seg_x1	Count_7seg_x3
Microchip MPASM	4	4
HI-TECH PICC-Lite	11	13
HI-TECH C PRO Lite	4	12
CCS PCB	7	11

The optimising C compilers are generating code up to 87% larger than the assembler version, for the 3-digit example, and using up to three times as much data memory.

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. 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 lesson 9).

Since we will need a device with analog inputs, we'll once again use the 14-pin PIC16F506.