Introduction to PIC Programming

Programming Midrange PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 7: Driving 7-Segment Displays

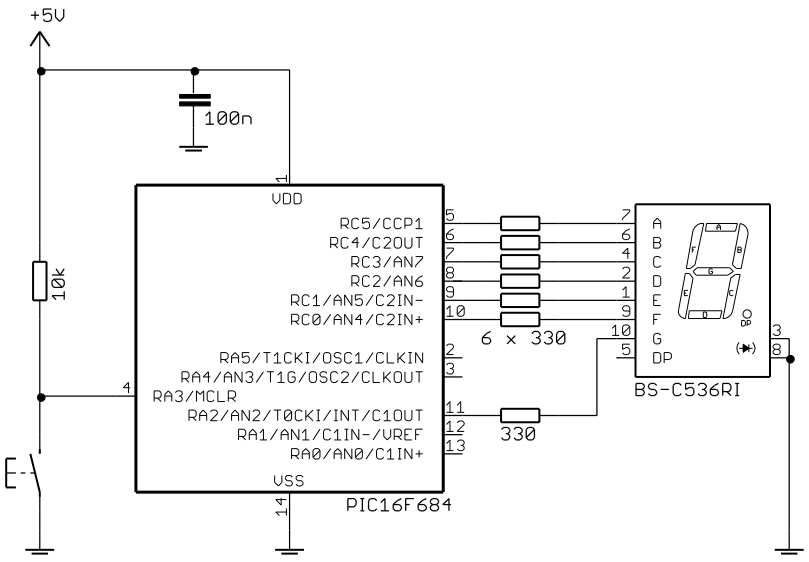
We saw in midrange lesson 12 how to drive 7-segment LED displays, using lookup-tables and multiplexing techniques implemented in assembly language. This lesson shows how C can be used to apply those techniques to drive multiple 7-segment displays, using the free HI-TECH C[1] (in "Lite" mode) and PICC-Lite compilers to re-implement the examples from the assembler lesson.

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 midrange lesson 12, using a PIC16F684, as shown here.



This circuit uses a common-cathode 7-segment LED module. 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 RA2 from PORTA.

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 midrange lesson 12, lookup tables on midrange PICs are normally implemented as a *computed goto* into a sequence of '`retlw`' instructions, each returning a value corresponding to its offset within 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 PRO 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 midrange lesson 12, we can write:

```
// Lookup pattern for 7 segment display on port A
const char pat7segA[10] = {
    // RA2 = 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:

```
PORTA = pat7segA[digit];    // lookup port A and C patterns
PORTC = pat7segC[digit];
```

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

However, the assembler example used two tables, one for PORTA, 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 A and C
const char pat7seg[10] = {
    // RC5:0,RA2 = 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 RA2. To derive that value, the pattern is ANDed with a mask, leaving only bit 0:

```
        RA2 = pat7seg[digit] & 0b0000001;
```

Complete program

Here is the complete single-lookup-table version of this example, for HI-TECH C PRO:

```
/************************************************************************
*   Description:    Lesson 7, 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:                                                    *
*       RA2, RC0-5 = 7-segment display (common cathode)                 *
*                                                                       *
************************************************************************/

#include <htc.h>

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

```
/***** CONFIGURATION *****/
//  ext reset, no code or data protect, no brownout detect,
//  no watchdog, power-up timer, int clock with I/O,
//  no failsafe clock monitor, two-speed start-up disabled
__CONFIG(MCLREN & UNPROTECT & BORDIS & WDTDIS & PWRTEN & INTIO & FCMDIS &
IESODIS);


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

// Lookup pattern for 7 segment display on ports A and C
const char pat7seg[10] = {
    // RC5:0,RA2 = 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
    TRISA = 0;                      // configure PORTA and PORTC as all outputs
    TRISC = 0;
    PORTA = 0;                      // make all PORTA pins low

    // Main loop
    for (;;) {
        // display each digit from 0 to 9 for 1 sec
        for (digit = 0; digit < 10; digit++) {
            // display digit
            RA2 = 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-HTC.h`" file.

The PICC-Lite version is the same, except that, to make use the delay functions it is supplied with, we substitute:

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

Comparisons

The following table summarises the resource usage for the "single-digit seconds counter" assembler and C example, along with the baseline (PIC16F505) versions of this example, from baseline C lesson 4, for comparison. Note however that the assembler example uses two lookup tables, while the C versions use a single lookup array with more complex pattern extraction. 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)	
	16F684	16F505	16F684	16F505	16F684	16F505
Microchip MPASM	60	66	66	72	4	4
HI-TECH PICC-Lite	25	26	78	96	8	11
HI-TECH C PRO Lite	25	26	107	122	6	4

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.

Note that the 16F684 versions are all smaller than their 16F505 equivalents, demonstrating that this type of application can be implemented more efficiently using the midrange PIC architecture.

Interrupt-driven Multiplexing

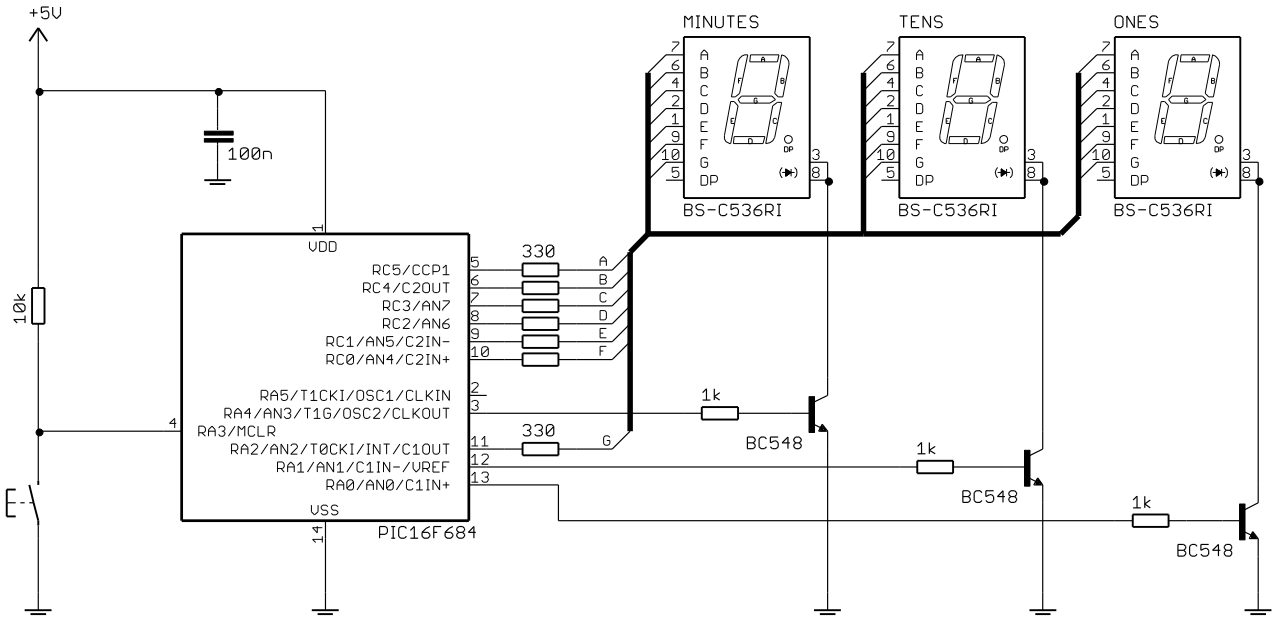
As explained in more detail in midrange lesson 12, *multiplexing* can used to drive mutiple displays, using a minimal number of output pins. Each display is lit in turn, one at a time, so rapidly that it appears to the human eye that each display is lit continuously.

Ideally the display multiplexing would be a "background task"; one that continues steadily while the main program is free to perform tasks such as responding to changing inputs. As we saw in lesson 3, that's an ideal application for timer-based interrupts. The interrupt service routine displays each digit, one at a time, in succession. If the interrupt is triggered at 1 ms intervals, each digit would be displayed for 1 ms, then the next digit for another 1 ms, and so on.

We'll use the example circuit from midrange lesson 12, shown at the top of the next page, to demonstrate how to implement this interrupt-driven multiplexing technique, using C.

Each 7-segment display is enabled 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 raising 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.



The multiplexing example in midrange lesson 12 was built a piece at a time. First, the single-digit example was re-implemented, using only the 'ONES' digit (enabled by raising RA0), with the display maintained by a timer-driven interrupt, running every 2.048 ms.

This was then expanded to three digits, counting minutes and seconds.

The interrupt service routine displayed a different digit on each 2.048 ms *tick*.

A "multiplex counter" variable was used in the ISR to keep track of which digit to display.

This approach was expressed this in pseudo code as:

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

However, the assembler code actually used was structured a little differently than this, so that it could be implemented more efficiently.

HI-TECH C PRO or PICC-Lite

As we did in midrange lesson 12, we can start by re-implementing the single-digit example, setting up a timer-based interrupt, running every 2.048 ms, to maintain the display.

In the assembler example we used a macro to perform the table lookups. In C, it is more natural to implement this as a function, called from the interrupt service routine:

```
void set7seg(char digit);       // display digit on 7-segment display (shadow)
```

This function would be responsible for outputting the pattern corresponding to the digit passed to it.

For consistency with the assembler examples, and because it is good practice (avoiding potential read-modify-write issues), we will use shadow port registers, with the `set7seg()` function updating the shadow registers, not the ports directly.

It makes sense to include the pattern table definitions within the function, so that the function is self-contained – only the function needs to "know" about the pattern tables; they are never accessed directly from other parts of the program.

So we have:

```
/***** Display digit on 7-segment display (shadow) *****/
void set7seg(char digit)
{
    // Lookup pattern table for 7 segment display on PORTA
    const char pat7segA[10] = {
        // RA2 = G
        0b000000,   // 0
        0b000000,   // 1
        0b000100,   // 2
        0b000100,   // 3
        0b000100,   // 4
        0b000100,   // 5
        0b000100,   // 6
        0b000000,   // 7
        0b000100,   // 8
        0b000100    // 9
    };

    // Lookup pattern table for 7 segment display on PORTC
    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
    };

    // lookup pattern bits and write to shadow registers
    sPORTA = pat7segA[digit];
    sPORTC = pat7segC[digit];
}
```

Note that we've gone back to using two pattern tables. Because we're writing the whole of PORTA (actually, the shadow copy of PORTA) in this function, and the display enable lines are connected to PORTA, it has the side effect of blanking the display by clearing all the display enable lines.

So, after calling this function, we must enable the display:

```
    // display digit (using shadow registers)
    set7seg(digit);                 // output digit
    sPORTA |= 1 << nDISPLAY;        // enable display
```

You can see that this will be easy to extend to multiple digits; all we need do is enable different displays, after outputting the appropriate digit pattern on the 7-segment bus.

We can then wrap this within an interrupt service routine:

```
void interrupt isr(void)
{
    // *** Service Timer0 interrupt
    //  TMR0 overflows every 2.048 ms
    //  (only Timer0 interrupts are enabled)
    //
    T0IF = 0;                       // clear interrupt flag

    // display digit (using shadow registers)
    set7seg(digit);                 // output digit
    sPORTA |= 1 << nDISPLAY;        // enable display

    // copy shadow regs to ports
    PORTA = sPORTA;
    PORTC = sPORTC;
}
```

And configure Timer0 to generate an interrupt (running this ISR) every 2.048 ms:

```
    // setup Timer0
    OPTION = 0b11000010;            // configure Timer0:
             //--0-----                 timer mode (T0CS = 0)
             //----0---                 prescaler assigned to Timer0 (PSA = 0)
             //-----010                 prescale = 8 (PS = 010)
                                    //  -> increment every 8 us
                                    //  -> TMR0 overflows every 2.048 ms
    // configure interrupts
    T0IE = 1;                       // enable Timer0 interrupt
    ei();                           // enable global interrupts
```

With this interrupt code running in the background, taking care of displaying the current contents of the '`digit`' variable (which now has to be declared as a global variable, so that the ISR can access it), all the main loop code has to do is update the value of '`digit`' every second:

```
    // Main loop
    for (;;)
    {
        // display each digit from 0 to 9 for 1 sec
        for (digit = 0; digit < 10; digit++)
        {
            DelayS(1);              // delay 1 sec
        }
    }
```

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

In the first three-digit assembler example (midrange lesson 12, example 2), the time count digits were stored as a separate variables:

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

This 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".

This was followed by an example showing how the seconds value could be stored in a single value, using binary-coded decimal (BCD) format to save data memory, while keeping the process of extracting each digit relatively simple:

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

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

```
unsigned char   mins = 0;           // time counters (displayed by ISR)
unsigned char   secs = 0;
```

(initialised to ensure that they hold defined, legal values when the ISR, which references them, first runs)

And then to extract the tens digit (by dividing seconds by ten) and display it, using the function developed above, we can simply write:

```
        set7seg(secs/10);               // output tens digit
        sPORTA |= 1 << nTENS;           // enable tens display
```

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

```
        set7seg(secs%10);               // output ones digit
        sPORTA |= 1 << nONES;           // enable ones display
```

This code assumes that the symbols '`nTENS`' and '`nONES`' have been defined:

```
// Pin assignments
#define nMINS   4           // minutes enable on RA4
#define nTENS   1           // tens enable on RA1
#define nONES   0           // ones enable on RA0
```

Within the interrupt service routine, we need to keep track, from one invocation of the ISR to the next, of which digit to display next. So we need to declare a static variable within the ISR, to be used for this:

```
    static unsigned char    mpx_cnt = 0;    // multiplex counter
```

It is then straightforward to translate the pseudo code presented above into C:

```
    if (mpx_cnt == 0) {
        set7seg(secs%10);               // output ones digit
        sPORTA |= 1 << nONES;           // enable ones display
    }
    if (mpx_cnt == 1) {
        set7seg(secs/10);               // output tens digit
        sPORTA |= 1 << nTENS;           // enable tens display
    }
    if (mpx_cnt == 2) {
        set7seg(mins);                  // output minutes digit
        sPORTA |= 1 << nMINS;           // enable minutes display
    }
    // Increment mpx_cnt, to select next digit for next time
    mpx_cnt++;
    if (mpx_cnt == 3)       // reset count if at end of digit sequence
        mpx_cnt = 0;
```

However, when selecting between blocks of code, based on various possible values of a single variable, it is more normal to use the C '`switch`' statement:

```
switch (mpx_cnt)
{
    case 0:
        set7seg(secs%10);               // output ones digit
        sPORTA |= 1 << nONES;           // enable ones display
        break;
    case 1:
        set7seg(secs/10);               // output tens digit
        sPORTA |= 1 << nTENS;           // enable tens display
        break;
    case 2:
        set7seg(mins);                  // output minutes digit
        sPORTA |= 1 << nMINS;           // enable minutes display
        break;
}
// Increment mpx_cnt, to select next digit for next time
mpx_cnt++;
if (mpx_cnt == 3)       // reset count if at end of digit sequence
    mpx_cnt = 0;
```

Note that the multiplex count is updated in a separate increment and test (in case the end of the sequence has been reached) operation, after the '`switch`' statement.

An alternative is to update `mpx_cnt` within the `switch` statement:

```
switch (mpx_cnt)
{
    case 0:
        set7seg(secs%10);               // output ones digit
        sPORTA |= 1 << nONES;           // enable ones display
        mpx_cnt = 1;                    // display tens next
        break;
    case 1:
        set7seg(secs/10);               // output tens digit
        sPORTA |= 1 << nTENS;           // enable tens display
        mpx_cnt = 2;                    // display minutes next
        break;
    case 2:
        set7seg(mins);                  // output minutes digit
        sPORTA |= 1 << nMINS;           // enable minutes display
        mpx_cnt = 0;                    // display ones next
        break;
}
```

This is in the form of a *state machine*, where for each current state, we explicitly state what the next state will be. It is particularly appropriate when the states are not purely sequential, as they are here.

This method is also shorter, when there are only a couple of cases. In this example, both code fragments are the same length, and PICC-Lite generates the same size (optimised) code in both cases, so there is no advantage one way or another – it becomes a matter of personal style.

After updating the shadow registers, we have to copy them to PORTA and PORTC, as before:

```
// copy shadow regs to ports
PORTA = sPORTA;
PORTC = sPORTC;
```

Once again, with the display being updated by the interrupt code, in the background, the main loop code has nothing to do except increment the counters, once per second:

```
    for (;;)
    {
        // count minutes:seconds from 0:00 to 9:59
        // (displayed by ISR)
        for (mins = 0; mins < 10; mins++)
        {
            for (secs = 0; secs < 60; secs++)
            {
                DelayS(1);              // delay 1 sec
            }
        }
    }
```

Complete program

Fitting all this together, we have, for PICC-Lite:

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

#include <htc.h>

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


/***** CONFIGURATION *****/
//  ext reset, no code or data protect, no brownout detect,
//  no watchdog, power-up timer, int clock with I/O,
//  no failsafe clock monitor, two-speed start-up disabled
__CONFIG(MCLREN & UNPROTECT & BORDIS & WDTDIS & PWRTEN & INTIO & FCMDIS &
IESODIS);

// Pin assignments
#define nMINS   4           // minutes enable on RA4
#define nTENS   1           // tens enable on RA1
#define nONES   0           // ones enable on RA0


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

```
/***** GLOBAL VARIABLES *****/
unsigned char   sPORTA;         // shadow registers: PORTA
unsigned char   sPORTC;         //                   PORTC
unsigned char   mins = 0;           // time counters (displayed by ISR)
unsigned char   secs = 0;


/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation
    // setup ports
    TRISA = 0;                      // configure PORTA and PORTC as all outputs
    TRISC = 0;
    // setup Timer0
    OPTION = 0b11000010;            // configure Timer0:
             //--0-----                 timer mode (T0CS = 0)
             //----0---                 prescaler assigned to Timer0 (PSA = 0)
             //-----010                 prescale = 8 (PS = 010)
                                    //  -> increment every 8 us
                                    //  -> TMR0 overflows every 2.048 ms
    // configure interrupts
    T0IE = 1;                       // enable Timer0 interrupt
    ei();                           // enable global interrupts

    // Main loop
    for (;;)
    {
        // count minutes:seconds from 0:00 to 9:59
        // (displayed by ISR)
        for (mins = 0; mins < 10; mins++)
        {
            for (secs = 0; secs < 60; secs++)
            {
                DelayS(1);              // delay 1 sec
            }
        }
    }
}


/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{
    static unsigned char    mpx_cnt = 0;    // multiplex counter

    // *** Service Timer0 interrupt
    //  TMR0 overflows every 2.048 ms
    //  (only Timer0 interrupts are enabled)
    //
    T0IF = 0;                       // clear interrupt flag

    // Display current count on 3 x 7-segment displays
    //   mpx_cnt determines current digit to diplay
    //
    switch (mpx_cnt)
    {
        case 0:
            set7seg(secs%10);               // output ones digit
            sPORTA |= 1 << nONES;           // enable ones display
            break;
```

```
        case 1:
            set7seg(secs/10);               // output tens digit
            sPORTA |= 1 << nTENS;           // enable tens display
            break;
        case 2:
            set7seg(mins);                  // output minutes digit
            sPORTA |= 1 << nMINS;           // enable minutes display
            break;
    }
    // Increment mpx_cnt, to select next digit for next time
    mpx_cnt++;
    if (mpx_cnt == 3)       // reset count if at end of digit sequence
        mpx_cnt = 0;

    // copy shadow regs to ports
    PORTA = sPORTA;
    PORTC = sPORTC;
}


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

/***** Display digit on 7-segment display (shadow) *****/
void set7seg(char digit)
{
    // Lookup pattern table for 7 segment display on PORTA
    const char pat7segA[10] = {
        // RA2 = G
        0b000000,   // 0
        0b000000,   // 1
        0b000100,   // 2
        0b000100,   // 3
        0b000100,   // 4
        0b000100,   // 5
        0b000100,   // 6
        0b000000,   // 7
        0b000100,   // 8
        0b000100    // 9
    };

    // Lookup pattern table for 7 segment display on PORTC
    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
    };

    // lookup pattern bits and write to shadow registers
    sPORTA = pat7segA[digit];
    sPORTC = pat7segC[digit];
}
```

As always, the HI-TECH C PRO version is the same, except for substituting:

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

Comparisons

Here is the resource usage summary for all of the 3-digit time count example programs:

Count_7seg_x3

Assembler / Compiler	Source code (lines)		Program memory (words)		Data memory (bytes)	
	16F684	16F505	16F684	16F505	16F684	16F505
Microchip MPASM (non-BCD)	136	113	150	97	11	5
Microchip MPASM (BCD)	142	114	156	99	11	4
HI-TECH PICC-Lite	68	44	214	185	23	13
HI-TECH C PRO Lite	68	44	456	425	20	12

You can see that using interrupts to drive the display multiplexing in the midrange versions has added to the program complexity, increasing both the source code length and program memory size, compared with the baseline versions. You may conclude from this that using interrupts isn't worth the trouble, but if you compare the relative simplicity of the "main loop" code in the midrange, interrupt-driven examples with the baseline versions, you can see that we now have a platform that is easily built upon – the display code, in the ISR, can remain the same, while program complexity increases.

Otherwise, the patterns we saw for the baseline examples in baseline C lesson 4 remain the same.

The C source code is less than half as long as either of the assembler version, demonstrating how much simpler it is to implement the display multiplexing algorithm in C.

However – even the optimised code generated by the PICC-Lite compiler is significantly (37%) larger than the hand-written assembler version. As in the baseline example, this is due to the use of the apparently innocuous '`/`' and '`%`' arithmetic operations in the C version.

And again, the non-optimised code generated by the HI-TECH C PRO compiler (running in 'lite' mode) is terrible – around three times as big as either of the assembler versions!

Summary

We have seen in this lesson that lookup tables can be effectively implemented in C as initialised arrays qualified as '`const`', and that by using 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 timer-based interrupt-driven multiplexing to implement a multi-digit display, without needing to be as concerned (as we were in the assembler versions) about how to store the values being displayed.

Thus, both examples could be expressed very succinctly in C, using either compiler, compared with the assembler versions (the non-BCD assembler version is the basis for comparison for example 2):

Source code (lines)

Assembler / Compiler	Count_7seg_x1	Count_7seg_x3
Microchip MPASM	60	136
HI-TECH PICC-Lite	25	68
HI-TECH C PRO Lite	25	68

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.

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	66	150
HI-TECH PICC-Lite	78	214
HI-TECH C PRO Lite	107	456

Data memory (bytes)

Assembler / Compiler	Count_7seg_x1	Count_7seg_x3
Microchip MPASM	4	11
HI-TECH PICC-Lite	8	23
HI-TECH C PRO Lite	6	20

The optimising PICC-Lite compilers is generating code up to 43% larger than the assembler version, for the 3-digit example, and using more than twice 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 makes use of our new ability to display numeric values, covering analog-to-digital conversion and simple arithmetic and arrays (revisiting material from midrange lessons 13 and 14).