Introduction to PIC Programming

Programming Midrange PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 8: Analog-to-Digital Conversion and Simple Filtering

Midrange lesson 13 explained how to use the 10-bit analog-to-digital converter (ADC) module available on midrange PICs, such as the PIC16F684, using assembly language. This lesson demonstrates how to use C to control and access the ADC, re-implementing the examples using the free HI-TECH C[1] (in "Lite" mode) and PICC-Lite compilers.

It then shows how a simple moving-average filter, as described in midrange lesson 14, can be implemented in C. The final example implements a simple light meter, with the light level smoothed, scaled and shown as two decimal digits, using 7-segment LED displays.

In summary, this lesson covers:

- Using the ADC module to read analog inputs
- ADC operation in sleep mode
- ADC interrupts
- Hexadecimal output on 7-segment displays
- Working with arrays
- Calculating a moving average to implement a simple filter

with examples for HI-TECH C and PICC-Lite.

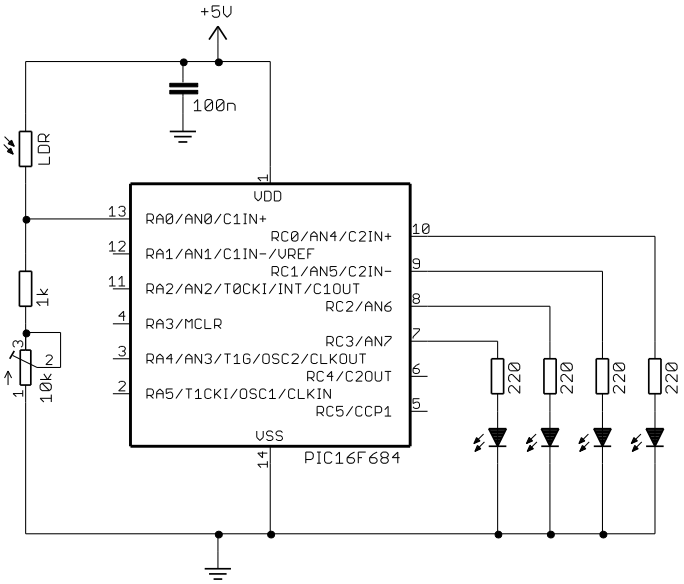
Analog-to-Digital Converter

As explained in more detail in midrange lesson 13, the *analog-to-digital converter (ADC)* module on midrange PICs allows analog input voltages to be measured, with a resolution of ten bits: 0 corresponds to VSS, and 1023 corresponds to either VDD or a reference voltage on the VREF pin.

The ADC module on the 16F684 has eight external inputs, or *channels*: AN0 to AN7. But, since there is only a single ADC module, only one channel can be selected at one time, meaning that only one input can be read (or *converted*) at once.

A simple example in midrange lesson 13 demonstrated basic ADC operation, using the circuit on the next page – similar to that used in the previous lesson on comparators, but using four LEDs connected to RC0 – RC3 (labeled 'DS1' – 'DS4' if you are using the Low Pin Count Demo Board).

[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.



The LDR/resistor voltage divider presents a voltage on AN0 which increases with light level. This voltage is continually *sampled*, with the most significant four bits of the result being displayed on the LEDs, forming a 4-bit binary display indicating (in a crude way) the light level.

The analog inputs share pins with RA0-2, RA4 and RC0-3. By default (after a power-on reset), the analog inputs are enabled. To use a pin for digital I/O, any analog function on that pin must first be disabled.

Whether a pin is configured for analog input is controlled by the corresponding bit in the ANSEL register; setting a bit in ANSEL places the corresponding analog input into analog mode, while clearing it makes the corresponding pin available for digital I/O.

You can also see that a quick way to disable all of the analog inputs is to clear ANSEL, although, as we saw in midrange lesson 12, you must also disable the comparators, by selecting comparator mode 7, to be able to use the comparator input pins (RA0, RA1, RC0 and RC1 on the 16F684) for digital I/O.

So to make all pins available for digital I/O, we have (for HI-TECH C):

```
// enable all digital inputs
CMCON0 = 7;    // disable comparators  (CM = 7 -> both comparators off)
ANSEL = 0;     // deselect all analog inputs
```

But in this example, we will be using AN0 (only) as an analog input, so we should set ANSEL<0> and leave the rest of ANSEL clear.

Having configured one or more pins as analog inputs, you must select which of those input channels to read, or *sample*, using the CHS<2:0> bits in the ADCON0 register:

CHS<2:0>	ADC channel
000	AN0
001	AN1
010	AN2
011	AN3

CHS<2:0>	ADC channel
100	AN4
101	AN5
110	AN6
111	AN7

In this example, AN0 has to be selected as the ADC channel, specified by CHS<2:0> = '000'.

An appropriate ADC conversion clock must be selected, so that the bit conversion period, TAD, is at least 1.6 µs – as explained in midrange lesson 13.

The conversion clock is selected by the ADCS<2:0> bits in ADCON1, as shown in the table on the right.

Given the default 4 MHz processor clock rate, the FOSC/8 option (ADCS<2:0> = '001') is best, giving TAD = 2.0 µs.

The ADC's internal RC clock option, FRC (ADCS<2:0> = '001'), is typically only used when the ADC needs to operate in sleep mode, as we'll see later.

ADCS<2:0>	conversion clock	TAD
000	FOSC / 2	TCY / 2
001	FOSC / 8	TCY × 2
010	FOSC / 32	TCY × 8
011	FRC	1.6 µs – 9 µs
100	FOSC / 4	TCY
101	FOSC / 16	TCY × 4
110	FOSC / 64	TCY × 16
111	FRC	1.6 µs – 9 µs

The 10-bit result is presented in the ADRESL and ADRESH registers, and the ADFM bit in ADCON0 selects either of two ways to split the 10-bit value between these two 8-bit registers:

If ADFM = 0, the result is *left-justified*, with the most significant eight bits of the result in ADRESH, and the least significant two bits of the result in the upper two bits of ADRESL.

If ADFM = 1, the result is *right-justified*, with the least significant eight bits of the result in ADRESL, and the most significant two bits of the result in the lower two bits of ADRESH.

Since we want to use only the top four bits of the result in this example, it is easier to work with them if they are all in the same register, so it's best to select the left-justified result format (ADFM = 0); the top four bits of the 10-bit result will be held in the high nybble (top four bits) of ADRESH.

The positive reference voltage is selected by the VCFG bit[2] in ADCON0:

VCFG = 0 selects VREF = VDD

VCFG = 1 means that VREF is taken from the VREF pin (shared with RA1)

Finally, the ADC module is turned on, by setting the ADON bit (in ADCON0) to '1'.

In the first example in midrange lesson 13, the ADC was configured with the above options using:

```
movlw   b'00010000'
        ; -001----          Tad = 8*Tosc (ADCS = 001)
banksel ADCON1          ;   -> Tad = 2.0 us (with Fosc = 4 MHz)
movwf   ADCON1
movlw   b'00000001'
        ; 0-------          MSB of result in ADRESH<7> (ADFM = 0)
        ; -0------          voltage reference is Vdd (VCFG = 0)
        ; ---000--          select channel AN0 (CHS = 000)
        ; -------1          turn ADC on (ADON = 1)
banksel ADCON0
movwf   ADCON0
```

[2] On newer midrange PICs, such as the 16F887, the ADC's negative reference is also selectable, between VSS and an external voltage input, using a second VCFG bit.

Before beginning a conversion, the ADC holding capacitor must be given enough time (T_{ACQ} – known as the *acquisition* or *sampling* time) to charge.

The device data sheets include formulas you can use to calculate the minimum T_{ACQ} – one of the main variables is the source impedance of the input being sampled, although temperature also plays a role. However, assuming that the source impedance is less than the recommended maximum of 10 kΩ, an acquisition time of 10 µs is adequate for the 16F684 and most modern midrange PICs.

After delaying for the required acquisition time, the conversion is then initiated by setting the GO/$\overline{\text{DONE}}$ bit in ADCON0 to '1'.

Your code then needs to wait until the GO/$\overline{\text{DONE}}$ bit has been cleared to '0', which indicates that the conversion is complete. You can then read the conversion result from the ADRESH and ADRESL registers.

You should copy the result before beginning the next conversion, so that it isn't overwritten during the conversion process.

HI-TECH C implementation

Since HI-TECH C makes the special function registers directly accessible through variables defined in the device-specific header files, the code to configure RC0 – RC3 as outputs, and AN0 as an analog input, is simply:

```
// configure ports
TRISC = 0;                  // PORTC is all outputs
ANSEL = 1<<0;               // AN0 (only) is analog
CMCON0 = 7;                 // disable comparators  (CM = 7)
```

Configuring the ADC module can then be done by writing to ADCON0 and ADCON1:

```
// configure ADC
ADCON1 = 0b00010000;
         //-001----          Tad = 8*Tosc (ADCS = 001)
         //                      = 2.0 us (with Fosc = 4 MHz)
ADCON0 = 0b00000001;
         //0-------          MSB of result in ADRESH<7> (ADFM = 0)
         //-0------          voltage reference is Vdd (VCFG = 0)
         //---000--          select channel AN0 (CHS = 000)
         //-------1          turn ADC on (ADON = 1)
```

Before starting the conversion, we must wait the required minimum acquisition time, which can be done by:

```
__delay_us(10);         // wait 10 us for acquisition time
```

using the delay function and `__delay_us()` macro built into HI-TECH C, or:

```
DelayUs(10);            // wait 10 us for acquisition time
```

using the `DelayUs()` macro provided with PICC-Lite.

To begin the conversion, set the GO/$\overline{\text{DONE}}$ bit:

```
GODONE = 1;             // start conversion
```

We then wait until the GO/$\overline{\text{DONE}}$ bit is clear (something that can be done quite succinctly in C):

```
        while (GODONE)          // wait until done
            ;
```

The upper eight bits of the result of the conversion is available in ADRESH, accessible through the 'ADRESH' variable.

We need to copy the upper four bits of the result to the lower four bits of PORTC (where the LEDs are connected). This means shifting the result four bits to the right, so we can write simply:

```
        PORTC = ADRESH >> 4;    // copy high four bits of result
                                //   to low nybble of output port
```

Complete program

Here is how the above code fragments fit together, for HI-TECH C:

```
/*****************************************************************
*                                                                *
*   Description:    Lesson 8, example 1                          *
*                                                                *
*   Demonstrates basic use of ADC                                *
*                                                                *
*   Continuously samples analog input, copying value to 4 x LEDs *
*                                                                *
*****************************************************************
*                                                                *
*   Pin assignments:                                             *
*       AN0     - voltage to be measured (e.g. pot output or LDR)*
*       RC0-3   - output LEDs (RC3 is MSB)                       *
*                                                                *
*****************************************************************/

#include <htc.h>

#define _XTAL_FREQ  4000000     // oscillator frequency for __delay_us()

/***** 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);


/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation

    // configure ports
    TRISC = 0;                  // PORTC is all outputs
    ANSEL = 1<<0;               // AN0 (only) is analog
    CMCON0 = 7;                 // disable comparators  (CM = 7)

    // configure ADC
    ADCON1 = 0b00010000;
             //-001----          Tad = 8*Tosc (ADCS = 001)
             //                      = 2.0 us (with Fosc = 4 MHz)
```

```
    ADCON0 = 0b00000001;
             //0-------          MSB of result in ADRESH<7> (ADFM = 0)
             //-0------          voltage reference is Vdd (VCFG = 0)
             //---000--          select channel AN0 (CHS = 000)
             //-------1          turn ADC on (ADON = 1)

    // Main loop
    for (;;)
    {
        // sample analog input
        __delay_us(10);         // wait 10 us for acquisition time
        GODONE = 1;             // start conversion
        while (GODONE)          // wait until done
            ;

        // display result on 4 x LEDs
        PORTC = ADRESH >> 4;    // copy high four bits of result
                                //   to low nybble of output port
    }
}
```

Comparisons

The following table summarises the resource usage for the "simple ADC demo" assembler and C examples, along with the baseline (PIC16F506) versions of this example, from baseline C lesson 6, for comparison.

ADC_4LEDs

Assembler / Compiler	Source code (lines)		Program memory (words)		Data memory (bytes)	
	16F684	16F506	16F684	16F506	16F684	16F506
Microchip MPASM	32	20	31	16	0	0
HI-TECH PICC-Lite	15	12	27	18	3	4
HI-TECH C (Lite)	14	12	44	40	7	4

The PICC-Lite compiler in this case generates extremely efficient code, even shorter than the hand-written assembler version, from source code less than half the length of the assembler source.

Note that the 16F684 versions are all larger than their 16F506 equivalents, reflecting the need for additional initialisation instructions (because there are more settings to configure), code required for the acquisition time delay (part of the conversion process in the baseline ADC module), and the additional bank selection instructions necessary in the midrange architecture.

ADC Operation in Sleep Mode

To save power, the PIC can be placed into sleep mode after the AD conversion has started. When the conversion is complete, the device will wake, with the result in ADRESL and ADRESH as normal.

As with any other event able to wake a midrange PIC from sleep mode, the corresponding interrupt source must be enabled, which, for the ADC module, is done by setting the ADIE bit in the PIE1 register, and, because the ADC module is a peripheral, also setting the PEIE bit in INTCON.

If you do not wish to actually generate an interrupt when the AD conversion completes (see the next section), you should ensure that the GIE bit in INTCON is clear.

The ADIF flag in the PIR1 register must be cleared before the conversion begins. This flag will be set when the conversion is complete, waking the device from sleep mode; if it has not been cleared, the PIC will wake immediately, before the conversion is complete, and the result will be incorrect.

It is also important to select the ADC's internal oscillator, FRC, as the conversion clock source. This is because the processor clock is stopped while in sleep mode – using the ADC's internal clock allows it to continue to operate, even while the rest of the PIC is stopped.

HI-TECH C implementation

To show how to the ADC module can be used in sleep mode, we can modify the previous example, so that the device enters sleep immediately after the AD conversion begins.

We need to modify the ADC configuration so that the ADC's internal oscillator has to be selected as the conversion clock source:

```
// configure ADC
ADCON1 = 0b00110000;
         //--11----          internal oscillator, Frc (ADCS = x11)
         //                   -> operation in sleep mode possible
ADCON0 = 0b00000001;
         //0-------          MSB of result in ADRESH<7> (ADFM = 0)
         //-0------          voltage reference is Vdd (VCFG = 0)
         //---000--          select channel AN0 (CHS = 000)
         //-------1          turn ADC on (ADON = 1)
```

We also need to enable the ADC and peripheral interrupts, and this is done through single-bit variables providing access to the interrupt enable flags, in much the same way as in previous lessons:

```
// enable ADC interrupt (for wake on completion)
ADIE = 1;                   // enable ADC interrupt
PEIE = 1;                   // and peripheral interrupts
```

Within the main loop, we now need to clear the ADC interrupt flag (ADIF), accessible via the single bit variable '`ADIF`', before initiating the conversion.

After the conversion has been done, the device can then be placed into sleep mode, using the `SLEEP()` macro, instead of polling the GO/$\overline{\text{DONE}}$ flag:

```
// Main loop
for (;;)
{
    // sample analog input
    __delay_us(10);         // wait 10 us for acquisition time
    ADIF = 0;               // clear ADC interrupt flag
    GODONE = 1;             // start conversion

    SLEEP();                // sleep until done

    // display result on 4 x LEDs
    PORTC = ADRESH >> 4;    // copy high four bits of result
                            //   to low nybble of output port
}
```

Note, however, that if we were serious about saving power, we'd turn off the LEDs before entering sleep mode. With the LEDs left on, the power saved by using sleep mode is minimal.

ADC Interrupts

As mentioned earlier, the ADC module can be configured to generate an interrupt when the analog-to-digital conversion process is complete. If the ADC interrupt is enabled, your code does not have to sit in a loop, polling the GO/$\overline{\text{DONE}}$ flag. Instead, some of that the AD conversion time can be spent on with other tasks, until the conversion is complete. The ADC interrupt will then be triggered, and your interrupt service routine (ISR) can immediately read the conversion result.

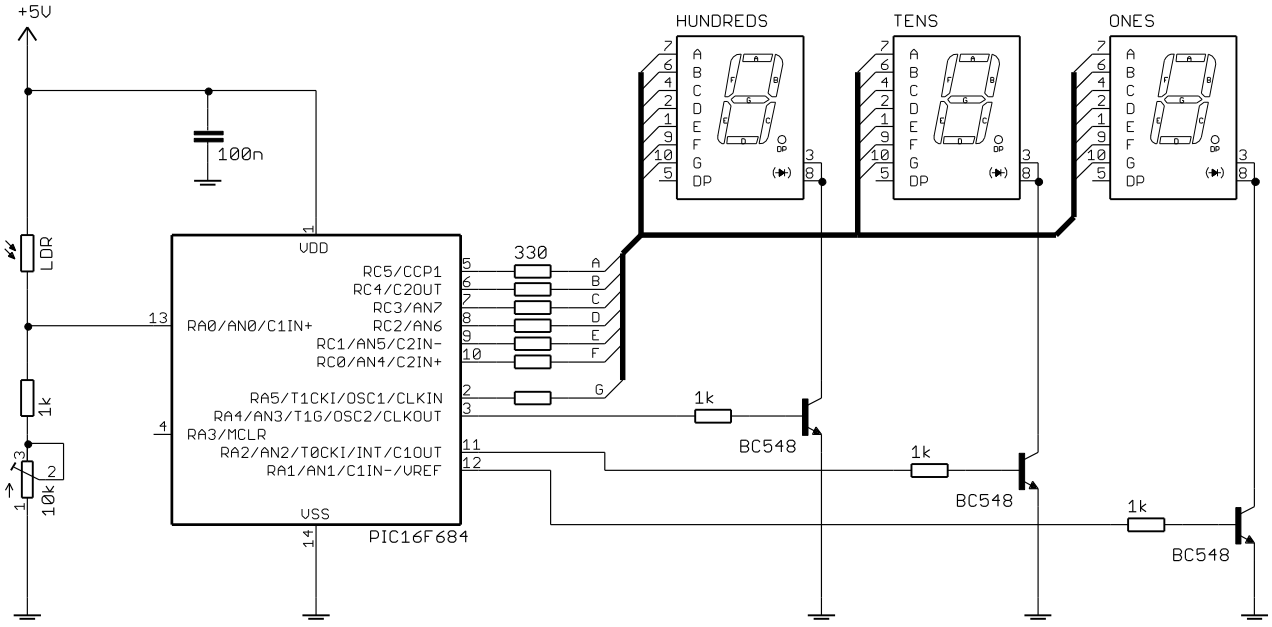
Note, however, that if your program has to perform a number of tasks, it doesn't really make sense to initiate a new AD conversion as soon as the last one completes – that approach leaves no time to do anything else. It is often more appropriate to perform the AD conversions at a steady rate – more slowly than the ADC module is actually capable of. An ideal way to do that is to use a timer-based interrupt (see lesson 3) to initiate the conversions. This means that other tasks can be completed in between AD conversions; it also means that there is no need for any additional, acquisition delay before initiating each conversion – we know that, with the conversions spaced apart, ample acquisition time has elapsed between each conversion.

If you are using a timer interrupt to initiate the AD conversions, it then makes sense to use an ADC interrupt to process the conversion result. This avoids placing a polling loop within the ISR; something to be avoided if at all possible. Interrupt service routines should be made as short and sharp as possible, so that other events can be responded to as quickly as possible.

To illustrate this, we'll use a multiplexed 7-segment LED display to output the value of an analog signal, with the AD conversion being done within the main loop, polling the GO/$\overline{\text{DONE}}$ flag. Then we'll re-implement the same thing, using an ADC interrupt, instead of polling.

Example 3: Hexadecimal Output

To add a more useful, human-readable output to the ADC demo, the third example in midrange lesson 13 implemented a three-digit hexadecimal display, based on the multiplexed 7-segment display circuit from midrange lesson 12, as shown below:



The source code was adapted from the timer interrupt-driven 7-segment display multiplexing routines presented in midrange lesson 12, with the only important differences being that, instead of a time count, the value to be displayed was now the 10-bit result of an analog-to-digital conversion, and that the pattern

lookup table for the 7-segment display was extended from 10 to 16 entries, to include representations of the letters 'A' to 'F'.

HI-TECH C implementation

Firstly we setup all of PORTC and four of the PORTA pins as digital outputs, with AN0 configured as an analog input:

```
// configure ports
TRISC = 0;                  // PORTC is all outputs
TRISA = 1<<0;               // configure RA0/AN0 (only) as an input
ANSEL = 1<<0;               // make only AN0 analog
CMCON0 = 7;                 // disable comparators (CM = 7)
```

The ADC is setup much as before, but this time the result is right-justified (ADFM = 1), to make it easier to extract the hex digits from the result:

```
// configure ADC
ADCON1 = 0b00010000;
         //-001----          Tad = 8*Tosc (ADCS = 001)
         //                      = 2.0 us (with Fosc = 4 MHz)
ADCON0 = 0b10000001;
         //1-------          LSB of result in ADRESL<0> (ADFM = 1)
         //-0------          voltage reference is Vdd (VCFG = 0)
         //---000--          select channel AN0 (CHS = 000)
         //-------1          turn ADC on (ADON = 1)
```

Most of the rest of the code is adapted from that presented in lesson 7.

For example, setting up the timer interrupt:

```
// configure 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();                           //  and global interrupts
```

A set of global variables are used to communicate with the ISR:

```
unsigned char   hundreds = 0;   // current ADC result (in hex): "hundreds"
unsigned char   tens = 0;       //  "tens"
unsigned char   ones = 0;       //  "ones"
```

(explicitly initialised to ensure that they hold defined values, within the expected 0 – 15 range, when the ISR, which references them, first runs)

The main loop updates these display variables, which hold the three digits which the ISR then displays on the 3 × 7-segment displays.

Note that since the value displayed is in hexadecimal, "hundreds" stores the number of 0x100s in the result, not 100s, and "tens" stores 0x10s, not 10s...

The main loop then has the job of performing the analog-to-digital conversion:

```
        // sample analog input
        __delay_us(10);         // wait 10 us for acquisition time
        GODONE = 1;             // start conversion
        while (GODONE)          // wait until done
            ;
```

and extracting the hexadecimal digits from the result into the display variables[3]:

```
        // copy result to variables
        //  (to be displayed by ISR)
        ones = ADRESL & 0x0F;   // get "ones" digit from low nybble of ADRESL
        tens = ADRESL >> 4;     // get "tens" digit from high nybble of ADRESL
        hundreds = ADRESH;      // get "hundreds" digit from ADRESH
```

The content of these variables is then displayed by the ISR, using code adapted from lesson 7:

```
    // Display current ADC result (in hex) on 3 x 7-segment displays
    //   mpx_cnt determines current digit to diplay
    //
    switch (mpx_cnt)
    {
        case 0:
            set7seg(ones);                  // output ones digit
            sPORTA |= 1 << nONES;           // enable ones display
            break;
        case 1:
            set7seg(tens);                  // output tens digit
            sPORTA |= 1 << nTENS;           // enable tens display
            break;
        case 2:
            set7seg(hundreds);              // output hundreds digit
            sPORTA |= 1 << nHUNDREDS;       // enable hundreds display
            break;
    }
```

The '`set7seg()`' function is much the same as that presented in lesson 7, extracting the pattern bits from a lookup array (now extended to 16 entries) for each port and writing to the corresponding shadow register:

```
/***** Display digit on 7-segment display (shadow) *****/
void set7seg(char digit)
{
    // Lookup pattern table for 7 segment display on PORTA
    const char pat7segA[16] = {
        // RA5 = G
        0b000000,   // 0
        [patterns 1-9 go here]
        0b100000,   // A
        0b100000,   // b
        0b000000,   // C
        0b100000,   // d
        0b100000,   // E
        0b100000    // F
    };
```

[3] This is an example of where C expressions can be much more succinct than the assembler equivalent; these three statements were implemented as fourteen lines of assembler in the corresponding example in midrange lesson 13.

```
    // Lookup pattern table for 7 segment display on PORTC
    const char pat7segC[16] = {
        // RC5:0 = ABCDEF
        0b111111,   // 0
        [patterns 1-9 go here]
        0b111011,   // A
        0b001111,   // b
        0b100111,   // C
        0b011110,   // d
        0b100111,   // E
        0b100011    // F
    };

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

Alternatively, you could use a single pattern array, instead of having a separate one for each port, and extract the bits for each port from it, as was done in lesson 7:

```
/***** Display digit on 7-segment display (shadow) *****/
void set7seg(char digit)
{
    // Lookup pattern table for 7 segment display on ports A and C
    const char pat7seg[16] = {
        // RC5:0,RA5 = ABCDEFG
        0b1111110,   // 0
        0b0110000,   // 1
        0b1101101,   // 2
        0b1111001,   // 3
        0b0110011,   // 4
        0b1011011,   // 5
        0b1011111,   // 6
        0b1110000,   // 7
        0b1111111,   // 8
        0b1111011,   // 9
        0b1110111,   // A
        0b0011111,   // b
        0b1001110,   // C
        0b0111101,   // d
        0b1001111,   // E
        0b1000111    // F
    };

    // lookup pattern bits and write to shadow registers
    sPORTA = (pat7seg[digit] & 0b0000001) << 5;     // update shadow RA5
    sPORTC = pat7seg[digit] >> 1;                   //  and PORTC
}
```

Note that the value to be written to RA5 is held in the least significant bit of the values in the pattern array; this is extracted by ANDing the looked-up value with binary 0000001, to mask out all the other bits. But because we're using shadow registers, we can't update RA5 directly with:

```
    RA5 = pat7seg[digit] & 0b0000001;
```

Instead, we have to shift this value 5 binary digits to the right (since it corresponds to RA5), before writing it to the shadow register for PORTA. This has the side-effect of clearing every other bit in PORTA, disabling whichever display is currently lit. The appropriate segment is then enabled in the ISR.

Using a single pattern array is certainly shorter, but the more complex extraction process means that the C compilers will generate longer code. The code generated by PICC-Lite occupies 153 words when separate pattern arrays are used, versus 156 words for the single, combined pattern array. There is very little difference, and it is clear that, as lookup tables become longer, it will become more memory efficient to combine them to save array storage memory – but in this case, using separate arrays is still (slightly) more efficient – and a little easier to understand.

Another change you could make would be to have the ISR read the ADC result directly, instead of using global display variables:

```
// Display current ADC result (in hex) on 3 x 7-segment displays
//   mpx_cnt determines current digit to diplay
//
switch (mpx_cnt)
{
    case 0:
        set7seg(ADRESL & 0x0F);         // output low nybble of ADRESL
        sPORTA |= 1 << nONES;           // enable ones display
        break;
    case 1:
        set7seg(ADRESL >> 4);           // output high nybble of ADRESL
        sPORTA |= 1 << nTENS;           // enable tens display
        break;
    case 2:
        set7seg(ADRESH);                // output ADRESH
        sPORTA |= 1 << nHUNDREDS;       // enable hundreds display
        break;
}
```

The main loop then does nothing more than continually perform analog-to-digital conversions:

```
// Main loop
for (;;)
{
    // sample analog input
    DelayUs(10);            // wait 10 us for acquisition time
    GODONE = 1;             // start conversion
    while (GODONE)          // wait until done
        ;
}
```

This is a valid approach (it works, and the code is shorter, and in some ways easier to understand), but goes against the principle of keeping the ISR code as short as possible. Extracting the hex digits from the ADC result does not involve a lot of processing; nevertheless, it is normally considered "better practice" to minimise the amount of processing performed within an ISR, so that it finishes as quickly as possible.

Complete program

Here is the complete HI-TECH C version of the "ADC demo with hexadecimal output" program, using separate pattern arrays and global display variables:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 8, example 3a                                *
*                                                                       *
*   Displays ADC output in hexadecimal on 7-segment LED displays        *
*                                                                       *
```

```
*   Continuously samples analog input,                                  *
*   displaying result as 3 x hex digits on multiplexed 7-seg displays   *
*   (one pattern lookup array per port)                                 *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       AN0         = voltage to be measured (e.g. pot or LDR)          *
*       RA5, RC0-5  = 7-segment display bus (common cathode)            *
*       RA4         = "hundreds" enable (active high)                   *
*       RA2         = "tens" enable                                     *
*       RA1         = "ones" enable                                     *
*                                                                       *
************************************************************************/

#include <htc.h>

#define _XTAL_FREQ  4000000     // oscillator frequency for __delay_us()


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

// Pin assignments
#define nHUNDREDS   4           // "hundreds" enable on RA4
#define nTENS       2           // "tens" enable on RA2
#define nONES       1           // "ones" enable on RA1


/***** 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   hundreds = 0;   // current ADC result (in hex): "hundreds"
unsigned char   tens = 0;       //  "tens"
unsigned char   ones = 0;       //  "ones"


/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation

    // configure ports
    TRISC = 0;                  // PORTC is all outputs
```

```
    TRISA = 1<<0;               // configure RA0/AN0 (only) as an input
    ANSEL = 1<<0;               // make only AN0 analog
    CMCON0 = 7;                 // disable comparators (CM = 7)

    // configure 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 ADC
    ADCON1 = 0b00010000;
             //-001----          Tad = 8*Tosc (ADCS = 001)
             //                      = 2.0 us (with Fosc = 4 MHz)
    ADCON0 = 0b10000001;
             //1-------          LSB of result in ADRESL<0> (ADFM = 1)
             //-0------          voltage reference is Vdd (VCFG = 0)
             //---000--          select channel AN0 (CHS = 000)
             //-------1          turn ADC on (ADON = 1)


    // configure interrupts
    T0IE = 1;                       // enable Timer0 interrupt
    ei();                           //  and global interrupts


    // Main loop
    for (;;)
    {
        // sample analog input
        __delay_us(10);         // wait 10 us for acquisition time
        GODONE = 1;             // start conversion
        while (GODONE)          // wait until done
            ;

        // copy result to variables
        //  (to be displayed by ISR)
        ones = ADRESL & 0x0F;   // get "ones" digit from low nybble of ADRESL
        tens = ADRESL >> 4;     // get "tens" digit from high nybble of ADRESL
        hundreds = ADRESH;      // get "hundreds" digit from ADRESH
    }
}


/***** 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 ADC result (in hex) on 3 x 7-segment displays
    //   mpx_cnt determines current digit to diplay
    //
    switch (mpx_cnt)
    {
        case 0:
            set7seg(ones);                  // output ones digit
            sPORTA |= 1 << nONES;           // enable ones display
            break;
        case 1:
            set7seg(tens);                  // output tens digit
            sPORTA |= 1 << nTENS;           // enable tens display
            break;
        case 2:
            set7seg(hundreds);              // output hundreds digit
            sPORTA |= 1 << nHUNDREDS;       // enable hundreds 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[16] = {
        // RA5 = G
        0b000000,   // 0
        0b000000,   // 1
        0b100000,   // 2
        0b100000,   // 3
        0b100000,   // 4
        0b100000,   // 5
        0b100000,   // 6
        0b000000,   // 7
        0b100000,   // 8
        0b100000,   // 9
        0b100000,   // A
        0b100000,   // b
```

```
        0b000000,   // C
        0b100000,   // d
        0b100000,   // E
        0b100000    // F
    };

    // Lookup pattern table for 7 segment display on PORTC
    const char pat7segC[16] = {
        // RC5:0 = ABCDEF
        0b111111,   // 0
        0b011000,   // 1
        0b110110,   // 2
        0b111100,   // 3
        0b011001,   // 4
        0b101101,   // 5
        0b101111,   // 6
        0b111000,   // 7
        0b111111,   // 8
        0b111101,   // 9
        0b111011,   // A
        0b001111,   // b
        0b100111,   // C
        0b011110,   // d
        0b100111,   // E
        0b100011    // F
    };

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

Comparisons

Here is the resource usage for the "ADC demo with hexadecimal output" assembler and C examples:

ADC_hex-out

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	150	165	8
HI-TECH PICC-Lite	88	153	14
HI-TECH C (Lite)	87	189	19

Once again, the PICC-Lite compiler generates extremely efficient code, continuing to be even shorter than the hand-written assembler version, from source code around half the length of the assembler source.

Note that it would not be valid to compare this example directly with the corresponding example in baseline C lesson 6, because that example was for only two digits (the baseline devices having only an 8-bit ADC).

Example 4: ADC Interrupts

Next we'll modify the last example, to use an ADC interrupt, instead of polling the GO/$\overline{\text{DONE}}$ flag.

This is done by enabling ADC interrupts, and getting the timer interrupt to initiate the conversion. When the conversion is complete, an interrupt is triggered, and the ADC interrupt handler copies the result into the display variables, to be displayed by the timer interrupt, as before.

Given that there are three digits, the overall display is refreshed every third timer interrupt. It is pointless to sample the input faster than the result can be displayed, so we should only initiate a conversion on every third timer interrupt. That is easily done, by tying the AD conversion to a specific digit.

It makes most sense to initiate the conversion after the "hundreds" digit has been displayed, because that is the last digit to be displayed in each refresh cycle.

HI-TECH C implementation

Most of the code is the same as in the last example.

To enable the ADC interrupt, we expand the interrupt initialisation code:

```
// configure interrupts
                            // enable ADC interrupt
ADIF = 0;                   //   clear interrupt flag
ADIE = 1;                   //   set enable bit
T0IE = 1;                   // enable Timer0 interrupt
PEIE = 1;                   // enable peripheral
ei();                       //   and global interrupts
```

This is much the same as in the wake-up from sleep example, except that now, because global interrupts are being enabled, as interrupt will actually be triggered as soon as the ADIF flag is set, which happens whenever a conversion completes.

Note that, to avoid the ADC interrupt triggering prematurely, ADIF is cleared before the interrupts are enabled[4].

The code to initiate the conversion is moved into the Timer0 ISR, just after the "hundreds" digit has been displayed:

```
// display "hundreds" digit
set7seg(hundreds);              // output hundreds digit
sPORTA |= 1 << nHUNDREDS;       // enable hundreds display

// get next analog sample
GODONE = 1;                     // start conversion
break;
```

Since we now have two interrupt sources, we need to add some code to the ISR, to execute the appropriate interrupt handler, depending on which interrupt flag has been set. Note that, because more than one interrupt flag may be set (more than one interrupt source may have triggered since the last time we entered the ISR), you should test the interrupt flags in priority.

---

[4] You may have noticed that we haven't been clearing the Timer0 interrupt flag, T0IF, before enabling the Timer0 interrupt. This is because the Timer0 interrupt occurs asynchronously with the rest of the program; that is, it can happen at any time. So it doesn't really matter if it triggers prematurely. In fact, because the initial value of TMR0 is undefined, the Timer0 interrupt could occur before your code is ready for it, even if you clear T0IF. If that's a concern, you should load TMR0 with a known value, and clear T0IF, before enabling the Timer0 interrupt.

In this case, we want the display to keep an even brightness, and a delay in reading the conversion result of a couple of milliseconds won't be noticeable, so we'll give priority to the timer interrupt:

```
// Service all triggered interrupt sources

if (T0IF)
{
    // *** Timer0 interrupt
}

if (ADIF)
{
    // *** ADC interrupt
}
```

The ADC interrupt handler then consists only of the code to copy the conversion result into the display variables:

```
    // *** ADC interrupt
    //
    ADIF = 0;                       // clear interrupt flag

    // copy ADC result to display variables
    //  (to be displayed by ISR)
    ones = ADRESL & 0x0F;   // get "ones" digit from low nybble of ADRESL
    tens = ADRESL >> 4;     // get "tens" digit from high nybble of ADRESL
    hundreds = ADRESH;      // get "hundreds" digit from ADRESH
```

This leaves the main loop with nothing to do, which becomes simply:

```
// Main loop
for (;;)
    ;                       // do nothing
```

All the regular analog input sampling and display updating is being done by the interrupt handlers, in the background, leaving the main loop to respond to external events, or a timer, or whatever.

Complete interrupt service routine

Since most of the code is the same as in the previous example, with the changes detailed above, there is no need to list the complete program here – but it's worth seeing the new ISR:

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

    // Service all triggered interrupt sources

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

        // Display current ADC result (in hex) on 3 x 7-segment displays
        //   mpx_cnt determines current digit to diplay
```

```
        //
        switch (mpx_cnt)
        {
            case 0:
                // display "ones" digit
                set7seg(ones);                  // output ones digit
                sPORTA |= 1 << nONES;           // enable ones display
                break;
            case 1:
                // display "tens" digit
                set7seg(tens);                  // output tens digit
                sPORTA |= 1 << nTENS;           // enable tens display
                break;
            case 2:
                // display "hundreds" digit
                set7seg(hundreds);              // output hundreds digit
                sPORTA |= 1 << nHUNDREDS;       // enable hundreds display

                // get next analog sample
                GODONE = 1;                     // start conversion
                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;
    }

    if (ADIF)
    {
        // *** ADC interrupt
        //
        ADIF = 0;                       // clear interrupt flag

        // copy ADC result to display variables
        //  (to be displayed by ISR)
        ones = ADRESL & 0x0F;   // get "ones" digit from low nybble of ADRESL
        tens = ADRESL >> 4;     // get "tens" digit from high nybble of ADRESL
        hundreds = ADRESH;      // get "hundreds" digit from ADRESH
    }
}
```

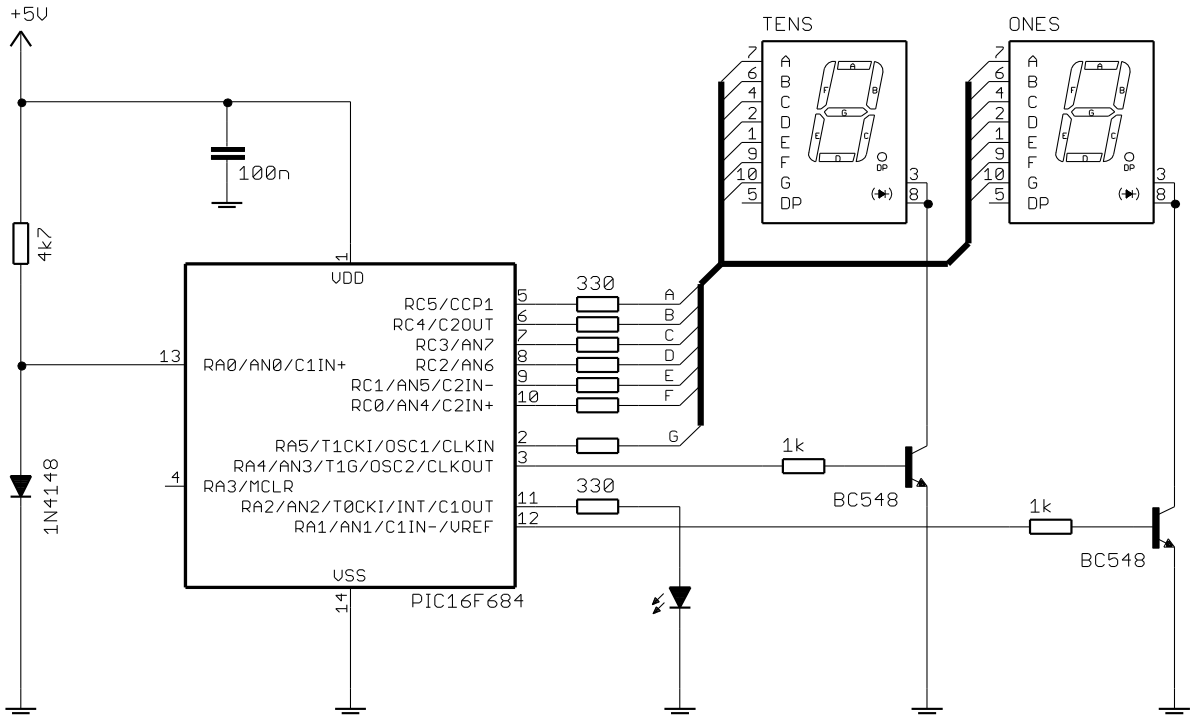
Measuring Supply Voltage

In some PICs, such as the 16F506 used in baseline C lesson 6, an internal fixed (or *absolute*) 0.6 V reference can be selected as an ADC input channel. This makes is possible to infer the supply voltage (effectively V_{DD}, given that in most cases V_{SS} = 0 V), since the 0.6 V reference will read as 0.6 V ÷ V_{DD} × 255.

For V_{DD} = 5.0 V, the expected ADC result = 30.

As V_{DD} falls, the ADC reading corresponding to 0.6 V rises. This provides a means to check that the power supply voltage (perhaps from a battery) is adequate, and to shut down the device and/or provide a warning if it falls too low.

Although the PIC16F684 does not provide an internal fixed voltage reference, we can achieve the same effect, using the forward voltage drop across a silicon diode as an external reference, as shown:



To demonstrate this, we will show the ADC result (now representing the value of the 0.6 V reference) in hex on the 7-segment displays, and to indicate low voltage, the LED on RA2 will be lit if VDD falls below a threshold of approximately 3.5 V.

Since we had already used all the available output pins on the 16F684, we have to drop one of the 7-segment displays, to accommodate the warning LED on RA2.

HI-TECH C implementation

Most of the program code is the same as that in example 3 (to make the code easier to follow, we won't use the ADC interrupt). We only need to reduce the number of displays from three to two, and add some code to turn on the warning LED when the value read on AN0 is above the threshold.

Since we are now displaying only two hex digits (the most significant eight bits of the result), it makes more sense to left-justify the result, using ADRESH as our 8-bit result register, and ignoring the two LSBs in ADRESL.

The ADC configuration becomes:

```
// configure ADC
ADCON1 = 0b00010000;
         //-001----          Tad = 8*Tosc (ADCS = 001)
         //                      = 2.0 us (with Fosc = 4 MHz)
ADCON0 = 0b00000001;
         //0-------          MSB of result in ADRESH<7> (ADFM = 0)
         //-0------          voltage reference is Vdd (VCFG = 0)
         //---000--          select channel AN0 (CHS = 000)
         //-------1          turn ADC on (ADON = 1)
```

The display routine in the ISR is reduced to two digits, becoming:

```
// Display current ADC result (in hex) on 3 x 7-segment displays
//   mpx_cnt determines current digit to diplay
//
switch (mpx_cnt)
{
    case 0:
        set7seg(ones);                  // output ones digit
        sPORTA |= 1 << nONES;           // enable ones display
        break;
    case 1:
        set7seg(tens);                  // output tens digit
        sPORTA |= 1 << nTENS;           // enable tens display
        break;
}
// Increment mpx_cnt, to select next digit for next time
mpx_cnt++;
if (mpx_cnt == 2)       // reset count if at end of digit sequence
    mpx_cnt = 0;
```

The input is sampled within the main loop, in the same way as before, but we now need to add some code to test for the under-voltage condition (V_{DD} < 3.5 V).

In the assembler example, the minimum allowable V_{DD} was defined as a constant at the beginning of the program, so that it could be easily changed later:

```
constant MINVDD=3500            ; Minimum Vdd (in mV)
```

It was necessary to express this as an integer, because MPASM does not support floating-point expressions. Thus, the expression to convert this minimum V_{DD} value to a constant which could be used to compare the ADC result with also had to be written using integers only:

```
constant VRMAX=255*600/MINVDD   ; Threshold for 0.6V ref measurement
```

Since C does support floating-point expressions, it is tempting to define the minimum V_{DD} as a floating-point constant:

```
#define MINVDD  3.5             // minimum Vdd (Volts)
```

and to then write the ADC comparison as:

```
        // test for low Vdd (measured 0.6 V > threshold)
        if (ADRESH > 0.6/MINVDD*255)    // if measured 0.6 V > threshold
        {
            sPORTA |= 1 << nWARN;       //   light warning LED
        }
```

Writing it that way makes the code very clear, because we normally refer to the internal reference as 0.6 V, not 600 mV, and it is natural to express the minimum V_{DD} as 3.5 V, not 3500 mV.

But there is a big problem with this – and it is a very easy mistake to make, when using C with small microcontrollers. The compiler sees '`0.6/MINVDD*255`' as a floating-point expression (which, of course, it is), and implements the comparison as a floating-point operation. To do so, it links a number of floating-point routines into the code, and generates code to convert `ADRESH` into floating-point form, passing it to a floating-point comparison routine. This greatly increases the size of the generated code, with the PICC-Lite version blowing out to 361 words of program memory[5]. Compare this with example 3, which uses three 7-

[5] Using PICC-Lite v9.60PL2, with default options.

segment displays and lacks only this ADC comparison routine, but required only 153 words of program memory. You wouldn't expect that adding such a simple routine would more than double the size of the generated program! And normally it wouldn't; the only reason the generated code is so large is that floating-point routines have been inadvertently, and unnecessarily, included into it.

> *Note: The inadvertent use of floating-point expressions in C programs can lead the C compiler to unnecessarily link floating-point routines into the object code, significantly increasing the size of the generated code.*

There are a number of ways to overcome this problem, including the use of integer-only expressions, but surely the simplest method, while maintaining clarity, is to explicitly *cast* the expression as an integer:

```
if (ADRESH > (int)(0.6/MINVDD*255))    // if measured 0.6V > threshold
{
    sPORTA |= 1 << nWARN;              //   light warning LED
}
```

This simple change prevents the compiler from including floating-point code, reducing the size of the generated code from 361 to only 145 words of program memory!

Finally, now that the shadow registers are being updated within the main loop (when the warning LED is turned on), it is necessary to move the port update routine, where the shadow registers are written to the ports, from the timer ISR to the main loop, to make changes made within the main loop visible. This is explained in more detail in midrange lesson 13.

Complete program

Although the changes from example 3 are not extensive, some of the structure of the program has changed (such as moving the port update routine out of the ISR), so it is worth looking at the complete source for the HI-TECH C version, to see how it all fits together:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 8, example 5b                                *
*                                                                       *
*   Demonstrates use of fixed reference with ADC to test supply voltage *
*                                                                       *
*   Continuously samples external 0.6V reference,                       *
*   displaying result as 2 x hex digits on multiplexed 7-seg displays   *
*   Turns on warning LED if measurement > threshold                     *
*   (using integer comparison)                                          *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       AN0         = external 0.6 V reference                          *
*       RA5, RC0-5  = 7-segment display bus (common cathode)            *
*       RA4         = "tens" enable (active high)                       *
*       RA1         = "ones" enable                                     *
*       RA2         = warning LED                                       *
*                                                                       *
************************************************************************/

#include <htc.h>

#define _XTAL_FREQ  4000000     // oscillator frequency for __delay_us()
```

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

// Pin assignments
#define nTENS       4           // "tens" enable on RA4
#define nONES       1           // "ones" enable on RA1
#define nWARN       2           // warning LED on RA2


/***** CONSTANTS *****/
#define MINVDD  3.5             // minimum Vdd (Volts)


/***** 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   tens = 0;       // current ADC result (in hex): "tens"
unsigned char   ones = 0;       //  "ones"


/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation

    // configure ports
    TRISC = 0;                  // PORTC is all outputs
    TRISA = 1<<0;               // configure RA0/AN0 (only) as an input
    ANSEL = 1<<0;               // make only AN0 analog
    CMCON0 = 7;                 // disable comparators  (CM = 7)

    // configure 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 ADC
    ADCON1 = 0b00010000;
             //-001----          Tad = 8*Tosc (ADCS = 001)
             //                      = 2.0 us (with Fosc = 4 MHz)
    ADCON0 = 0b00000001;
             //0-------          MSB of result in ADRESH<7> (ADFM = 0)
             //-0------          voltage reference is Vdd (VCFG = 0)
             //---000--          select channel AN0 (CHS = 000)
             //-------1          turn ADC on (ADON = 1)

    // configure interrupts
    T0IE = 1;                       // enable Timer0 interrupt
    ei();                           //  and global interrupts
```

```
    // Main loop
    for (;;)
    {
        // sample 0.6 V reference on AN0
        __delay_us(10);         // wait 10 us for acquisition time
        GODONE = 1;             // start conversion
        while (GODONE)          // wait until done
            ;

        // test for low Vdd (measured 0.6 V > threshold)
        if (ADRESH > (int)(0.6/MINVDD*255)) // if measured 0.6 V > threshold
        {
            sPORTA |= 1 << nWARN;           //   light warning LED
        }

        // copy result to variables
        //  (to be displayed by ISR)
        ones = ADRESH & 0x0F;   // get "ones" digit from low nybble of ADRESH
        tens = ADRESH >> 4;     // get "tens" digit from high nybble of ADRESH

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


/***** 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 ADC result (in hex) on 3 x 7-segment displays
    //   mpx_cnt determines current digit to diplay
    //
    switch (mpx_cnt)
    {
        case 0:
            set7seg(ones);                  // output ones digit
            sPORTA |= 1 << nONES;           // enable ones display
            break;
        case 1:
            set7seg(tens);                  // output tens digit
            sPORTA |= 1 << nTENS;           // enable tens display
            break;
    }
    // Increment mpx_cnt, to select next digit for next time
    mpx_cnt++;
    if (mpx_cnt == 2)       // reset count if at end of digit sequence
        mpx_cnt = 0;
}
```

```
/***** FUNCTIONS *****/

/***** Display digit on 7-segment display (shadow) *****/
void set7seg(char digit)
{
    // Lookup pattern table for 7 segment display on PORTA
    const char pat7segA[16] = {
        // RA5 = G
        0b000000,   // 0
        0b000000,   // 1
        0b100000,   // 2
        0b100000,   // 3
        0b100000,   // 4
        0b100000,   // 5
        0b100000,   // 6
        0b000000,   // 7
        0b100000,   // 8
        0b100000,   // 9
        0b100000,   // A
        0b100000,   // b
        0b000000,   // C
        0b100000,   // d
        0b100000,   // E
        0b100000    // F
    };

    // Lookup pattern table for 7 segment display on PORTC
    const char pat7segC[16] = {
        // RC5:0 = ABCDEF
        0b111111,   // 0
        0b011000,   // 1
        0b110110,   // 2
        0b111100,   // 3
        0b011001,   // 4
        0b101101,   // 5
        0b101111,   // 6
        0b111000,   // 7
        0b111111,   // 8
        0b111101,   // 9
        0b111011,   // A
        0b001111,   // b
        0b100111,   // C
        0b011110,   // d
        0b100111,   // E
        0b100011    // F
    };

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

Comparisons

Here is the resource usage comparison for the "VDD measure" example, including the floating-point and integer arithmetic versions of the C programs, along with the baseline (PIC16F506) versions of this example, from baseline C lesson 6, for comparison:

ADC_Vdd-measure

Assembler / Compiler	Arithmetic	Source code (lines)		Program memory (words)		Data memory (bytes)	
		16F684	16F506	16F684	16F506	16F684	16F506
Microchip MPASM	integer	142	99	144	87	7	1
HI-TECH PICC-Lite	float	85	50	361	297	31	15
HI-TECH PICC-Lite	integer	85	50	145	109	13	7
HI-TECH C (Lite)	float	84	50	458	552	42	34
HI-TECH C (Lite)	integer	84	50	184	162	18	7

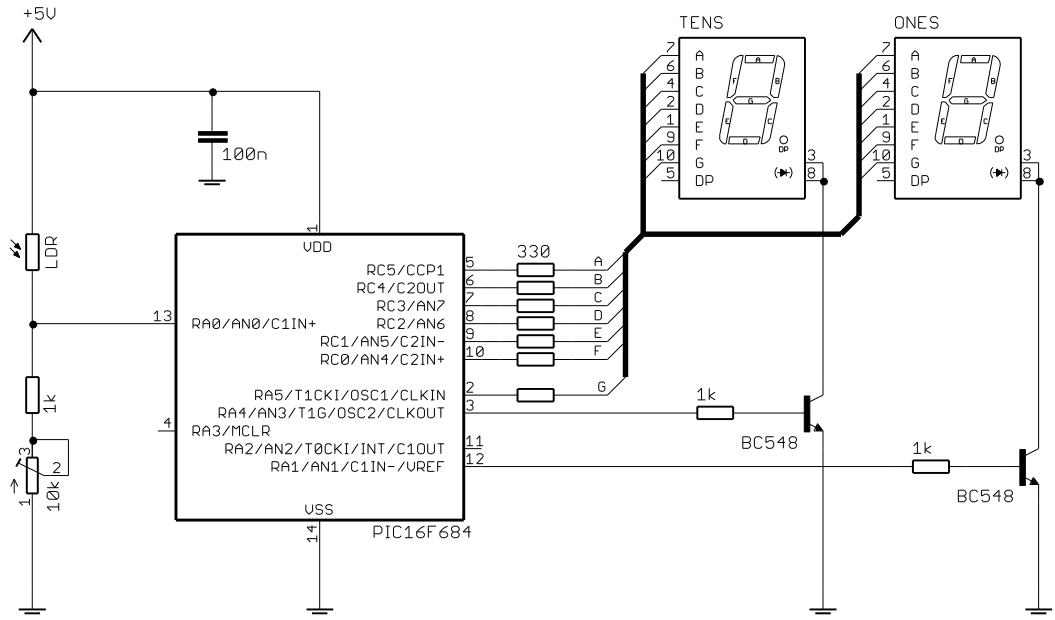
There's a lot of information in this table, but we can see that the C source code is around 40% shorter than the assembler source, while the PICC-Lite compiler continues to generate efficient code – only a few words larger than the assembler version. It's also apparent that the 16F684 versions are much larger than their 16F506 equivalents – mainly due to the use of timer interrupts to drive the display.

The real story here, however, is how very inefficient the floating-point versions are, in comparison with the integer versions, showing that floating-point operations should be avoided wherever possible.

Decimal Output

Most people would find it easier to read the output of the light meter presented above if the display was in decimal, not hex, with a scale from 0 – 99 instead of 0 – 3FFh.

To demonstrate how to do this, we'll use a 2-digit version of the circuit, as shown below.



This example was implemented in assembler in midrange lesson 14, where the main focus of the lesson was on integer arithmetic, including multi-byte addition and subtraction, and 8- and 16-bit multiplication. Since the C compiler takes care of the implementing arithmetic operations, we don't need to be concerned with those details here.

To scale the full 10-bit ADC output from 0 – 1023 to 0 – 99, it should be multiplied by 99/1023. That can be done easily in C, but it is more difficult to do in assembler. In the assembler example, the ADC result was limited to the most significant eight bits and then multiplied by 100/256, which is much easier to implement and is "close enough", in that the full-scale ADC output of 1023 will result in the full-scale value of 99 being displayed.

So that the C examples are comparable to the assembler version, we will use only use the most significant eight bits of the ADC result and scale it by 100/256 here, as well.

HI-TECH C implementation

Most of the HI-TECH C program code can be re-used from the hexadecimal example, above.

After sampling the analog input, we need to scale the ADC result to 0 – 99, and this scaled result is then referenced twice; once for each digit. So it makes sense to store the scaled result in a variable, which would normally be declared as:

```
unsigned char   adc_dec;        // scaled ADC output (0-99)
```

because this value will always be small enough (≤ 99) to represent using 8 bits.

However, the PICC-Lite compiler generates smaller code if we declare it as:

```
unsigned int    adc_dec;        // scaled ADC output (0-99)
```

C compilers usually *promote* smaller integral types (such as 'char') to type 'int' when they are included in integer arithmetic calculations. In fact, this behaviour is required by the ANSI C standard. By declaring this variable as type 'int', the PICC-Lite compiler does not need to promote it during arithmetic operations., and can therefore generate a smaller program.

Note that integer promotion is not mandatory in all situations, and C compilers will generally avoid it in situations where they can conclude that the result will be the same if promotion doesn't occur. So it's not a good idea to assume that a particular change, like this, will make your code smaller – it depends on the specific compiler and its optimisation settings.

In this case, HI-TECH C v9.70, running in "lite" mode, generates smaller code if we declare `adc_dec` as an `unsigned char`, as in the first declaration, above.

After sampling the input, the ADC result is scaled as follows:

```
// scale result to 0-99
adc_dec = ADRESH * 100/256;
```

Both HI-TECH compilers generate smaller code if this is written as:

```
adc_dec = (unsigned)ADRESH * 100/256;
```

That is, the 8-bit ADC result in ADRESH is cast as an unsigned integer.

This allows the compilers to optimise their code generation, because they can avoid promoting the ADC result to a signed integer and using signed multiplication and division routines; unsigned arithmetic is simpler and therefore requires less code to implement.

We then need to extract each digit of the scaled result for display. As we saw in lesson 7, this can be done using the integer division (/) and modulus (%) operators:

```
// extract digits of scaled result for ISR to display
//  (to be displayed by ISR)
ones = adc_dec%10;
tens = adc_dec/10;
```

However, HI-TECH C v9.70 (in "lite" mode) can generate smaller code if we write this as:

```
ones = (unsigned)adc_dec%10;
tens = (unsigned)adc_dec/10;
```

But this change makes no difference when using PICC-Lite.

Again, source code changes which will optimise code generation is very much compiler-specific.

As we did in the assembler example in midrange lesson 14, it is best to disable interrupts while the display variables are being updated. We don't know how the compiler implements these calculations, and hence we cannot know whether the variables will at any point hold values outside the range that the ISR is designed to handle.

So we will place these statements, along with the code to copy the shadow register variables to the ports, within a pair of directives (macros provided by HI-TECH C) to disable and then re-enable interrupts:

```
// disable interrupts during display variable update
// (stops ISR attempting to display out-of-range intermediate results)
di();

    // extract digits of scaled result for ISR to display
    //  (to be displayed by ISR)
    ones = (unsigned)adc_dec%10;
    tens = (unsigned)adc_dec/10;

    // copy shadow regs (updated by ISR) to ports
    PORTA = sPORTA;
    PORTC = sPORTC;

// re-enable interrupts
ei();
```

Main program listing

Here is the new `main()` function, showing where these additions fit in:

```
/***** MAIN PROGRAM *****/
void main()
{
    unsigned char    adc_dec;        // scaled ADC output (0-99)

    // Initialisation

    // configure ports
    TRISC = 0;                  // PORTC is all outputs
    TRISA = 1<<0;               // configure RA0/AN0 (only) as an input
    ANSEL = 1<<0;               // make only AN0 analog
    CMCON0 = 7;                 // disable comparators  (CM = 7)

    // configure 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 ADC
    ADCON1 = 0b00010000;
            //-001----          Tad = 8*Tosc (ADCS = 001)
            //                      = 2.0 us (with Fosc = 4 MHz)
    ADCON0 = 0b00000001;
            //0-------          MSB of result in ADRESH<7> (ADFM = 0)
            //-0------          voltage reference is Vdd (VCFG = 0)
            //---000--          select channel AN0 (CHS = 000)
            //-------1          turn ADC on (ADON = 1)


    // configure interrupts
    T0IE = 1;                       // enable Timer0 interrupt
    ei();                           //  and global interrupts


    // Main loop
    for (;;)
    {
        // sample analog input
        __delay_us(10);         // wait 10 us for acquisition time
        GODONE = 1;             // start conversion
        while (GODONE)          // wait until done
            ;

        // scale 8-bit ADC result to 0-99
        adc_dec = (unsigned)ADRESH * 100/256;

        // disable interrupts during display variable update
        // (stops ISR attempting to display out-of-range intermediate results)
        di();

            // extract digits of scaled result for ISR to display
            //  (to be displayed by ISR)
            ones = (unsigned)adc_dec%10;
            tens = (unsigned)adc_dec/10;

            // copy shadow regs (updated by ISR) to ports
            PORTA = sPORTA;
            PORTC = sPORTC;

        // re-enable interrupts
        ei();
    }
}
```

Comparisons

The table on the next page summarises the resource usage of the "ADC demo with decimal output" examples, along with the baseline (PIC16F506) versions from baseline C lesson 6, for comparison:

ADC_dec-out

Assembler / Compiler	Source code (lines)		Program memory (words)		Data memory (bytes)	
	16F684	16F506	16F684	16F506	16F684	16F506
Microchip MPASM	144	111	146	100	11	7
HI-TECH PICC-Lite	73	42	272	238	29	14
HI-TECH C (Lite)	72	42	388	529	26	20

In this example, where integer arithmetic is involved, the pros and cons of assembler versus C become very apparent. The assembler source is twice as long as the C versions, reflecting the need to explicitly code the arithmetic operations in assembler. On the other hand, the assembler version generates significantly smaller code – around half the size of the optimised PICC-Lite C version.

And again, the 16F684 versions are all much larger than their 16F506 equivalents.

Using an Array to Implement a Moving Average

A problem with the decimal-output example above (and the previous hexadecimal-output example) is that that the display can become unreadable in flickering light, such as that produced by fluorescent lamps. These flicker at 50 or 60 Hz – too fast for the human eye to notice, but not too quickly for our simple light meter, which displays the changing light level 244 times per second.

As we saw in midrange lesson 14, this problem can be effectively overcome by smoothing, or *filtering*, the raw results before displaying them. Although more advanced (and efficient and effective) filtering algorithms exist, one that is simple to implement is the *simple moving average* (or *box filter*), which averages the last N samples (where N is a fixed number, referred to as the *window* size), giving the same weight to each sample.

To implement this filter, we need to sample the input at a fixed rate (say, every 2 ms), which can be done by using a timer interrupt to initiate the conversions, as was done in the ADC interrupt example, above. We also need to store the last N samples, in an array of size N. Every time a new light level is sampled, the array is updated, with the oldest sample value being overwritten with the new one. Note that is it not necessary to calculate the sum of values in the array every time it is updated; we can instead maintain a running total by subtracting the oldest value and adding the new value to it.

Since the largest contiguous block of data memory in the 16F684 is 80 bytes, and given that a single, simple array[6] has to be contiguous, in a 16F684 no single array can be larger than 80 bytes. In practice, the maximum size will be less than this, because, in the 16F684, the largest contiguous block of data memory is located in bank 0 (see midrange lessons 10 and 14), and C compilers will normally need to allocate storage for at least some of the variables or working space in bank 0.

If we sample the input every 2 ms, we need to store at least ten samples (10 × 2 ms = 20 ms) to smooth out a 50 Hz flicker, since a 50 Hz signal has a period of 20 ms. But it wouldn't hurt to try to store enough samples to average across a few power cycles, to be sure of removing visible flicker.

In the assembler example in midrange lesson 14, we worked with the full 10-bit ADC output, maintaining more resolution than the display is able to show, mainly to demonstrate how to work with 16-bit quantities.

[6] as opposed to a more elaborate structure such as a linked list

This means that, if each10-bit sample is stored as a 16-bit value, we need to allocate two bytes for each sample in the array, or *buffer*. For example, storing sixteen samples would require thirty-two bytes of storage – which shouldn't be a problem, even after allowing for C compiler overheads.

PICC-Lite implementation

Storing the samples and maintaining the running total has to be done within the ADC interrupt handler, because it handles the ADC results as they become available.

In theory, calculating the scaled average from the running total could be done in the main loop, outside the ISR. That's a valid option and would often be the best choice, because calculations take time and it's usually best to keep the ISR as short as possible, so that there is enough time to perform other tasks. But in this example, there are no other interrupts or events for the program to respond to; it doesn't matter if a lot of time is spent within an ISR, performing calculations. Since it simplifies the code a little, we'll perform the scaling calculation within the ADC interrupt handler in this example.

That means that the running total, scaled average, and of course the sample buffer itself as well as the index used to reference the current sample can be declared as local variables within the ISR:

```
// variables used by ADC interrupt to calculate moving average
static unsigned int     smp_buf[NSAMPLES];  // sample buffer
                                            // cleared by startup code)
static unsigned char    s = 0;              // index into sample array
static unsigned int     adc_sum = 0;        // sum of samples in buffer
unsigned char           adc_dec;            // scaled avg ADC output (0-99)
```

where '`NSAMPLES`' is a symbol defined toward the start of the code, to make it easier to modify the number of samples stored in the array:

```
#define NSAMPLES    16          // size of sample array
```

As `NSAMPLES` is increased, at some point the `smp_buf[]` array will become too big to fit into memory alongside the other variables and the compiler's working space, and you will see error messages such as:

```
can't find 0x1 words (0x1 withtotal) for psect "intsave" in egment "BANK0"
can't find 0x1 words (0x1 withtotal) for psect "xtemp" in segment "BANK0"
```

Using PICC-Lite v9.60PL2 with the PIC16F684 and the code presented below, the maximum array size is 28 samples; much smaller than the assembler example in midrange lesson 14, where we were able to use the whole of bank 0 to store 40 samples.

Unfortunately, the PICC-Lite compiler limits data memory use on the 16F684 to a single bank: bank 0. On some midrange devices, such as the 16F690 and 16F887, PICC-Lite is able to use two banks, and the '`bank1`' type qualifier can then be used to explicitly place specific variables into the second bank. However, even if PICC-Lite did allow us to relocate some variables into bank 1 on the 16F684, it wouldn't be enough to make it possible to use the whole of bank 0 for the sample array; PICC-Lite will always allocate storage for interrupt context saving, parameter passing and temporary storage (such as immediate results) within bank 0. We simply have to accept that the sample buffer cannot be as large as in the assembler version.

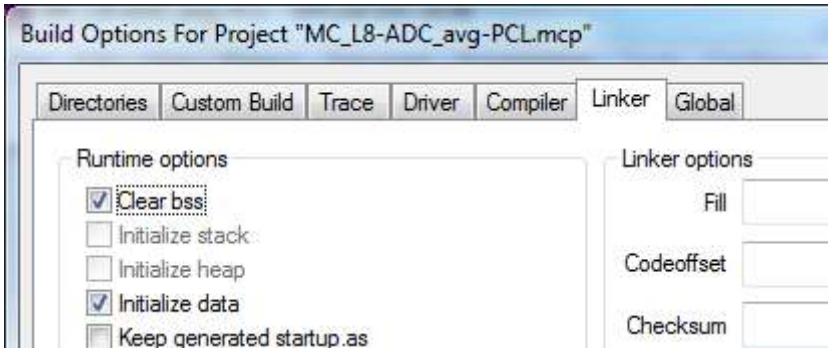
Note that the sample buffer, index and running total are declared as `static`, because they have to retain their values from one interrupt to the next.

Note also that the index and running total are cleared to zero as part of the variable declaration. The sample array also has to be cleared before it can be used, so that the running total is correct (if the running total is initially zero, the array elements must initially sum to zero – easiest to ensure if they are all initially equal to zero). But there is no need to include explicit code to clear the array.

By default, the PICC_Lite compiler adds runtime code which, among other things, clears all uninitialised non-auto (global or static) variables, including arrays.

You can check that this option is selected by looking at the "Linker" tab in the project's build options (Project → Build Options… → Project), as shown on the right.

If "Clear bss" is selected, the compiler-provided runtime code will clear all the variables.



Within the ADC interrupt handler, we must store the new sample and update the running total, as follows:

```
// store current ADC result and update running total
adc_sum -= smp_buf[s];              // subtract old sample from total
smp_buf[s] = ADRESH<<8 | ADRESL;    // save new sample
adc_sum += smp_buf[s];              // and add it to running total
```

Note the expression used to access the current ADC result: `ADRESH<<8 | ADRESL`.

Since the lower two bits of ADRESH hold the upper two bits of the 10-bit ADC result, we need to shift those bits eight places to the left, before ORing them with the lower eight bits held in ADRESL.

Another way to write this would be as '`ADRESH*256 + ADRESL`', but the first form more clearly expresses how we are assembling the 10-bit result from the two 8-bit registers.

The above code assumes that the sample buffer index ('`s`') is pointing to the current sample. This will be true when the program starts, because '`s`' is initialised to zero, but having processed the current sample, we need to increment '`s`' to reference the next sample, ready for the next time the ADC handler runs:

```
// advance index to reference next sample
if (++s == NSAMPLES) s = 0;
```

Equivalently, this could have been written as:

```
s++;                    // increment sample index
if (s == NSAMPLES)      // if end of buffer is reached
    s = 0;              //   reset index to start of buffer
```

The compiler will actually generate the same code in both cases, but the first form is shorter and just as easily understood when you are familiar with C.

Next, we need to calculate the scaled average.

The average is equal to the running total divided by the number of samples in the buffer, and could be calculated as:

```
adc_avg = adc_sum / NSAMPLES
```

This average value will be in the range 0 – 1023.

To scale it to the range 0 – 99, we can multiply it by 100 / 1024:

```
adc_dec = adc_avg * 100 / 1024
```

But there is no need to actually declare and use an intermediary variable such as ‘adc_avg’.

Instead, the averaging and scaling operations could be combined in a single expression, such as:

```
adc_dec = (adc_sum / NSAMPLES) * 100 / 1024;
```

or:

```
adc_dec = adc_sum * 100 / (1024 * NSAMPLES);
```

The second form is preferable, if we are striving to preserve as much resolution as possible, because if the division is done first, only the integer part of the quotient is preserved and any remainder is lost, because we are using integer arithmetic. Order of evaluation can be very important in integer arithmetic, and this can be source of errors and confusion.

For example, suppose adc_sum = 103 and NSAMPLES = 10.

Then:

```
adc_dec = (adc_sum / NSAMPLES) * 100 / 1024
        = (103 / 10) * 100 / 1024
        = (10) * 100 / 1024
        = 1000 / 1024
        = 0
```

because 103/10 evaluates to 10 using integer arithmetic (since the remainder is thrown away) and 1000/1024 evaluates to 0.

But:

```
adc_dec = adc_sum * 100 / (1024 * NSAMPLES)
        = 103 * 100 / (1024 * 10)
        = 10300 / 10240
        = 1
```

Using floating point (real number) arithmetic, the correct answer is 1.006. As you can see, the second integer expression gives a more accurate result, because we’re not losing information when the division is done.

But there is still a trap for the unwary:

Given that adc_sum is an ‘unsigned int’, and that by default PICC-Lite will perform calculations using 16-bit ‘int’ types, the constant ‘100’ is treated as an ‘int’ and the expression ‘adc_sum*100’ is evaluated as a 16-bit integer calculation.

That’s a problem if ‘adc_sum’ is greater than 655 (which it could easily be, given that individual samples range up to 1023). If adc_sum = 656, then adc_sum*100 = 65600, which is too big to be expressed as a 16-bit integer, and the result will *overflow*.

The result will be incorrect, but the worst part is that the compiler will not warn you that this could happen. Everything will appear to be ok, but your results will be very wrong!

To avoid this, we need to cast adc_sum as a long (32-bit) integer, so that the expression will not overflow:

```
adc_dec = (long)adc_sum * 100 / (1024 * NSAMPLES);
```

Alternatively, the constant ‘100’ can be specified as a 32-bit quantity by appending an ‘L’ to it:

```
adc_dec = adc_sum * 100L / (1024 * NSAMPLES);
```

However, as we saw in the decimal output example, the compiler generates smaller code if it can use unsigned arithmetic routines.

So finally we have:

```
// scale running total to 0-99 for display
adc_dec = (unsigned long)adc_sum * 100 / (1024 * NSAMPLES);
```

Note that PICC-Lite generates more efficient code to implement this expression if `NSAMPLES` is a power of two (such as 16), because division can then be performed by a series of right-shifts.

We can then extract the decimal digits of the scaled average for display, as before:

```
// extract digits of scaled result for Timer0 handler to display
ones = adc_dec%10;
tens = adc_dec/10;
```

The rest of the program is essentially the same as in the ADC interrupt example, above.

HI-TECH C (v9.70 'Lite mode') implementation

The HI-TECH C compiler, as of version 9.70, does not restrict the amount of usable program or data memory on any of the midrange PICs, including the 16F684, and therefore is able to utilise bank 1, even when running in the free 'Lite' mode. Further, HI-TECH C will automatically manage the allocation of storage across banks as appropriate.

So you could reasonably expect that it would be possible to declare a larger array, storing more samples, than we were able to in the PICC-Lite version.

Unfortunately, when running in 'Lite mode', HI-TECH C performs very little code optimisation, and this appears to extend to the way it uses data memory; it is actually quite inefficient.

Using HI-TECH C v9.70 in 'Lite mode' with the PIC16F684 and the same variable types as the PICC-Lite version, the maximum array size is only 24 samples; even less than PICC-Lite was able to support.

But like PICC-Lite, HI-TECH C generates smaller code for the scaled average calculation if `NSAMPLES` is a power of two.

Another effect of the 'Lite mode' code not being optimised is that it runs more slowly.

This has a serious impact in this example: the code used to calculate the scaled moving average is unable to complete within 2 ms. That's a real problem when the interrupt, running that code, is being triggered every 2 ms – there isn't enough time, between interrupts, for the calculations to finish!

The solution is to change the timer initialisation code, so that the interrupt is triggered every 4 ms, giving the calculations enough time to run:

```
// configure Timer0
OPTION = 0b11000011;
         //--0-----
         //----0---
         //-----011
```

```
// configure Timer0:
    timer mode (T0CS = 0)
    prescaler assigned to Timer0 (PSA = 0)
    prescale = 16 (PS = 011)
//  -> increment every 16 us
//  -> TMR0 overflows every 4.096 ms
```

Luckily this is still fast enough that the multiplexed display, which will now have a 125 Hz refresh rate, doesn't appear to flicker.

Finally, as we saw in the decimal output example, the HI-TECH C compiler generates smaller code if we change the digit extraction code to:

```
        // extract digits of scaled result for Timer0 handler to display
        ones = (unsigned)adc_dec%10;
        tens = (unsigned)adc_dec/10;
```

Complete program

Here is the complete source code for the HI-TECH C version of the "ADC demo with averaged decimal output" program, showing where these code fragments fit in:

```
/************************************************************************
*   Description:    Lesson 8, example 7                                 *
*                                                                       *
*   Displays smoothed ADC output in decimal on 2x7-segment LED displays *
*                                                                       *
*   Continuously samples analog input, averages last 16 samples,        *
*   scales result to 0 - 99 and displays as 2 x decimal digits          *
*   on multiplexed 7-seg displays                                       *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       AN0         = voltage to be measured (e.g. pot or LDR)          *
*       RA5, RC0-5  = 7-segment display bus (common cathode)            *
*       RA4         = tens enable (active high)                         *
*       RA1         = ones enable                                       *
*                                                                       *
************************************************************************/

#include <htc.h>


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

// Pin assignments
#define nTENS_EN    4           // tens enable on RA4
#define nONES_EN    1           // ones enable on RA1


/***** CONSTANTS *****/
#define NSAMPLES    16         // size of sample array


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


/***** GLOBAL VARIABLES *****/
unsigned char   sPORTA;         // shadow registers: PORTA
unsigned char   sPORTC;         //                   PORTC

                                // current result in decimal (displayed by ISR):
unsigned char   tens = 0;       //   tens
unsigned char   ones = 0;       //   ones
```

```
/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation

    // configure ports
    TRISC = 0;                  // PORTC is all outputs
    TRISA = 1<<0;               // configure RA0/AN0 (only) as an input
    ANSEL = 1<<0;               // make only AN0 analog
    CMCON0 = 7;                 // disable comparators  (CM = 7)

    // configure Timer0
    OPTION = 0b11000011;            // configure Timer0:
             //--0-----                  timer mode (T0CS = 0)
             //----0---                  prescaler assigned to Timer0 (PSA = 0)
             //-----011                  prescale = 16 (PS = 011)
                                    //  -> increment every 16 us
                                    //  -> TMR0 overflows every 4.096 ms

    // configure ADC
    ADCON1 = 0b00010000;
             //-001----          Tad = 8*Tosc (ADCS = 001)
             //                      = 2.0 us (with Fosc = 4 MHz)
    ADCON0 = 0b10000001;
             //1-------          LSB of result in ADRESL<0> (ADFM = 1)
             //-0------          voltage reference is Vdd (VCFG = 0)
             //---000--          select channel AN0 (CHS = 000)
             //-------1          turn ADC on (ADON = 1)


    // configure interrupts
                                // enable ADC interrupt
    ADIF = 0;                   //   clear interrupt flag
    ADIE = 1;                   //   set enable bit
    T0IE = 1;                   // enable Timer0 interrupt
    PEIE = 1;                   // enable peripheral
    ei();                       //   and global interrupts


    // Main loop
    for (;;)
    {
        // copy shadow regs (updated by ISR) to ports
        PORTA = sPORTA;
        PORTC = sPORTC;
    }
}


/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{
    // variables used by timer0 interrupt
    static unsigned char    mpx_cnt = 0;        // multiplex counter

    // variables used by ADC interrupt to calculate moving average
    static unsigned int     smp_buf[NSAMPLES];  // sample buffer
                                                // cleared by startup code)
    static unsigned char    s = 0;              // index into sample array
    static unsigned int     adc_sum = 0;        // sum of samples in buffer
    unsigned char           adc_dec;            // scaled avg ADC output (0-99)
```

```
    // Service all triggered interrupt sources

    if (T0IF)
    {
        // *** Timer0 interrupt
        //  TMR0 overflows every 4.096 ms
        //  (only Timer0 interrupts are enabled)
        //
        T0IF = 0;                       // clear interrupt flag

        // Display current ADC result (in hex) on 3 x 7-segment displays
        //   mpx_cnt determines current digit to diplay
        //
        switch (mpx_cnt)
        {
            case 0:
                // display "ones" digit
                set7seg(ones);                  // output ones digit
                sPORTA |= 1 << nONES_EN;        // enable ones display
                break;
            case 1:
                // display "tens" digit
                set7seg(tens);                  // output tens digit
                sPORTA |= 1 << nTENS_EN;        // enable tens display
                break;
        }
        // Increment mpx_cnt, to select next digit for next time
        mpx_cnt++;
        if (mpx_cnt == 2)       // reset count if at end of digit sequence
            mpx_cnt = 0;

        // start next analog conversion
        GODONE = 1;
    }

    if (ADIF)
    {
        // *** ADC interrupt
        //
        ADIF = 0;                       // clear interrupt flag

        // store current ADC result and update running total
        adc_sum -= smp_buf[s];              // subtract old sample from total
        smp_buf[s] = ADRESH<<8 | ADRESL;    // save new sample
        adc_sum += smp_buf[s];              // and add it to running total

        // advance index to reference next sample
        if (++s == NSAMPLES) s = 0;

        // scale running total to 0-99 for display
        adc_dec = (unsigned long)adc_sum * 100 / (1024 * NSAMPLES);

        // extract digits of scaled result for Timer0 handler to display
        ones = (unsigned)adc_dec%10;
        tens = (unsigned)adc_dec/10;
    }
}


/***** 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] = {
        // RA5 = G
        0b000000,   // 0
        0b000000,   // 1
        0b100000,   // 2
        0b100000,   // 3
        0b100000,   // 4
        0b100000,   // 5
        0b100000,   // 6
        0b000000,   // 7
        0b100000,   // 8
        0b100000,   // 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];
}
```

Comparisons

Here is the resource usage for the "ADC demo with averaged decimal output" midrange and baseline assembler and C examples, also showing the sample buffer size used in each version:

ADC_avg

Assembler / Compiler	Number of Samples	Source code (lines)		Program memory (words)		Data memory (bytes)	
		16F684	16F506	16F684	16F506	16F684	16F506
Microchip MPASM	40	200	-	193	-	95	-
Microchip MPASM	16	-	146	-	133	-	26
HI-TECH PICC-Lite	16	79	57	379	254	72	34
HI-TECH PICC-Lite	28	79	-	500	-	98	-
HI-TECH C (Lite)	16	79	49	514	521	66	38
HI-TECH C (Lite)	24	79	-	641	-	96	-

In this example, the differences between C and assembler are even more pronounced. The midrange (16F684) assembler source is more than twice as long as the HI-TECH C versions, while the assembled code is only around half the size of the "optimised" code generated by PICC-Lite for the 16-sample version.

More significantly, neither of the free HI-TECH C compilers allowed us to implement a sample array anywhere near as large as that possible with hand-written assembler: only 24 samples with HI-TECH C (in 'Lite mode') and 28 samples with PICC-Lite, compared with 40 samples with assembler. And in increasing the buffer size from 16 to 24 samples in the PICC-Lite version, the generated code becomes 32% bigger, making it nearly 160% larger than the assembler version, even though the assembler version is able to work with 40% more samples.

We have reached a point where the limitations of the free HI-TECH C compilers are getting in the way of making full use of the PIC's capability. The PIC16F684 has enough data memory to store and work with 40 samples, but only if we program in assembler; the free versions, at least, of the HI-TECH C compilers are too inefficient.

Summary

The examples in this lesson demonstrate that it is possible to effectively perform analog to digital conversion on midrange PICs, such as the PIC16F684, using free HI-TECH C compilers. But we have also seen that, using either PICC-Lite or HI-TECH C operating in 'Lite mode'[7], we were forced to implement buffers (or arrays) smaller than those we were able to implement using assembler.

The C source code continues to be significantly shorter than the assembler equivalent (typically around half as many lies) for each example. This difference is especially pronounced when array handing and arithmetic expressions, which can be written succinctly in C, are heavily used, as in the final example:

Source code (lines)

Assembler / Compiler	ADC_4LEDs	ADC_hex_out	Vdd_measure	ADC_dec_out	ADC_avg
Microchip MPASM	32	150	142	144	200
HI-TECH PICC-Lite	15	88	85	73	79
HI-TECH C (Lite)	14	87	84	72	79

An optimising C compiler, such as PICC-Lite, can sometimes generate code that is smaller than hand-written assembler. This is apparent in the table below, for the the first three examples. But, although arithmetic expressions can be expressed succinctly in C, the compilers usually generate code which is significantly larger than the corresponding assembler versions; the last two examples, using integer arithmetic, being significantly than the assembler versions; more than twice the size, in the final example:

Program memory (words)

Assembler / Compiler	ADC_4LEDs	ADC_hex_out	Vdd_measure	ADC_dec_out	ADC_avg
Microchip MPASM	31	165	144	146	193
HI-TECH PICC-Lite	27	153	145	272	500
HI-TECH C (Lite)	44	189	184	388	641

[7] using the versions available at the time of writing (April 2010): PICC-Lite v9.60PL2 and HI-TECH C v9.70

Data memory usage has not been a big concern until now, because there has always been plenty of data memory available. However, this became significant in the final example, where inefficient data memory use meant that we were unable to make as effective use of the 16F684's resources; we were forced to implement much smaller sample buffers using C, than were able to in assembler.

In the table below, the data memory use for the ADC averaging example is shown along with the corresponding number of samples we were able to store within that memory, in brackets:

Data memory (bytes)

Assembler / Compiler	ADC_4LEDs	ADC_hex_out	Vdd_measure	ADC_dec_out	ADC_avg
Microchip MPASM	0	8	7	11	95 (40)
HI-TECH PICC-Lite	3	14	13	29	98 (28)
HI-TECH C (Lite)	7	19	18	26	96 (24)

There is no doubt that it is much easier to express complex routines in C than assembler, which is reflected in the C code being significantly shorter source than the corresponding assembler source code.

On the other hand, we have seen that it is very important to be aware of the impact of variable and expression types on code generation, and the need to use type casting appropriately, to allow the compiler to generate more efficient code or indeed, as we saw in the last example, to produce correct results.

So although it is very easy to write arithmetic expressions in C, you have to be very careful when doing so.

It also appears that, in the last example, when implementing a "large" sample buffer, we were starting to reach the limit of what can be achieved with either the free HI-TECH C compilers on a device as small as the PIC16F684. To make the most of a PIC's memory resources, it may be necessary to write at least parts of the program in assembler. Or – use a bigger PIC. Or pay for an optimising C compiler without limitations (which is nevertheless unlikely to use memory as efficiently as hand-written assembler in many cases).

We have now gone as far as the baseline C tutorial series did, but of course the midrange PIC architecture has a lot more to offer.

Even for something as apparently simple as timers, we have only just scratched the surface, having only described Timer0 so far.

In the next lesson, we'll revisit the material from midrange lesson 15, introducing a 16-bit timer: Timer1.