Introduction to PIC Programming

Programming Baseline PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 7: Analog-to-Digital Conversion and Simple Filtering

Baseline assembler lesson 10 explained how to use the analog-to-digital converter (ADC) available on baseline PICs, such as the PIC16F506, using assembly language. This lesson demonstrates how to use C to control and access the ADC, re-implementing the examples using Microchip's XC8 (running in "Free mode") and CCS' PCB compilers[1].

It then shows how a simple moving-average filter, as described in baseline assembler lesson 11, 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:

- Configuring the ADC peripheral
- Reading analog inputs
- Hexadecimal output on 7-segment displays
- Working with arrays
- Accessing more than one bank of data memory
- Calculating a moving average to implement a simple filter

with examples for XC8 and CCS PCB.

Analog-to-Digital Converter

As explained in more detail in baseline assembler lesson 10, the *analog-to-digital converter (ADC)* peripheral on baseline PICs allows analog input voltages to be measured, with a resolution of eight bits: 0 corresponds to VSS, and 255 corresponds to VDD.

The ADC module on the 16F506 has three external inputs, or *channels*: AN0, AN1 and AN2. Since there is only one ADC module, only one channel can be selected at one time, meaning that only one input can be read (*sampled* or *converted*) at once.

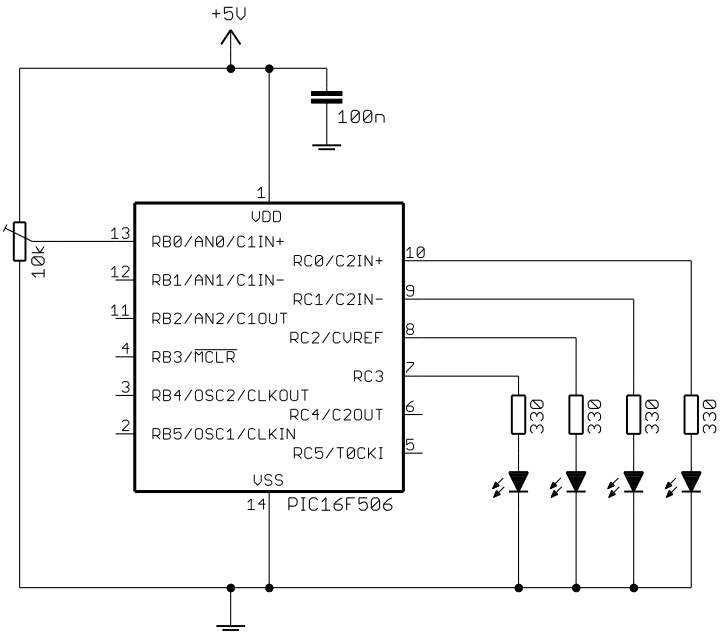
A simple example in baseline lesson 10 demonstrated basic ADC operation, using use a potentiometer to provide a variable voltage to an analog input, and four LEDs to show a 4-bit binary representation of that value, using the circuit shown on the next page.

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

To implement it using the Gooligum baseline training board, place a shunt across pins 1 and 2 ('POT') of JP24, connecting the 10 kΩ pot (RP2) to AN0, and shunts in JP16-19, enabling the LEDs on RC0-3.

If you are using Microchip's Low Pin Count Demo Board, the onboard pot and LEDs are already connected to AN0 and RC0 – RC3. You only need to ensure that jumpers JP1-5 are closed.

The voltage on AN0 is continually sampled, with the most significant four bits of the result being displayed on the LEDs, forming a 4-bit binary display.



The analog inputs share pins with RB0, RB1 and RB2. 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 ANS<1:0> bits in the ADCON0 register, as shown on in the table on the right.

The pins cannot be configured independently; only the listed combinations are possible.

A quick way to disable the analog inputs is to clear ADCON0, since clearing ANS<1:0> deselects all the analog inputs.

ANS<1:0>	Pins configured as analog inputs
00	none
01	AN2 only
10	AN0 and AN2
11	AN0, AN1 and AN2

In this example, only AN0 has to be configured as an analog input; either of the combinations which include AN0 could be used – in this case, the "AN0 and AN2" option, selected by ANS<1:0> = '10', is used.

The appropriate ADC input channel must also be selected. This is controlled by the CHS<1:0> bits in ADCON0, as shown on the right.

Note that, in addition to the three external analog inputs, the 0.6 V fixed voltage reference is selectable as an ADC channel. We'll use this feature in a later example.

In this example, AN0 has to be selected as the ADC channel, specified by CHS<1:0> = '00'.

CHS<1:0>	ADC channel
00	analog input AN0
01	analog input AN1
10	analog input AN2
11	0.6 V internal voltage reference

An appropriate ADC conversion clock source must be selected, specified by the ADCS<1:0> bits in ADCON0. As explained in baseline assembler lesson 10, the INTOSC/4 clock option (ADCS<1:0> = '11') is a safe option which will always work, so that option is used here.

Finally, the ADC peripheral must be turned on, by setting the ADON bit (in ADCON0) to '1'.

In the example in baseline assembler lesson 10, the ADC was configured with the above options with:

```
        movlw   b'10110001'     ; configure ADC:
                ; 10------          AN0, AN2 analog (ANS = 10)
                ; --11----          clock = INTOSC/4 (ADCS = 11)
                ; ----00--          select channel AN0 (CHS = 00)
                ; -------1          turn ADC on (ADON = 1)
        movwf   ADCON0          ;   -> AN0 ready for sampling
```

To begin a conversion, the GO/$\overline{\text{DONE}}$ bit (in ADCON0) is set:

```
        bsf     ADCON0,GO       ; start conversion
```

It is then necessary to wait until the GO/$\overline{\text{DONE}}$ bit is clear:

```
w_adc   btfsc   ADCON0,NOT_DONE ; wait until done
        goto    w_adc
```

The result of the conversion is then available in the ADRES register:

```
        swapf   ADRES,w         ; copy high nybble of result
        movwf   PORTC           ;   to low nybble of output port (LEDs)
```

Note that, in this example, the most significant four bits of the result are copied to the least four significant bits of PORTC, because the LEDs are connected to RC0 – RC3.

We saw in baseline assembler lesson 10 that, to use RC0 and RC1 for digital I/O, the C2IN+ and C2IN- inputs must be disabled. This was done by clearing CM2CON0:

```
        clrf    CM2CON0         ; disable comparator 2 -> RC0, RC1 digital
```

We also saw that, to use RC2 for digital I/O, the CVREF output has to be disabled. Although the programmable voltage reference module is disabled by default, it was explicitly turned off in the example, by clearing VRCON:

```
        clrf    VRCON           ; disable CVref -> RC2 usable
```

XC8

Since XC8 makes the special function registers directly accessible through variables defined in the device-specific header files, the code to configure RC0 – RC3 as outputs is simply:

```
    // configure ports
    TRISC = 0b110000;           // configure RC0-RC3 as outputs
    CM2CON0 = 0;                // disable comparator 2 -> RC0, RC1 digital
    VRCON = 0;                  // disable CVref -> RC2 usable
```

Configuring the ADC module could then be done in the same way, by assigning a value to ADCON0:

```
    // configure ADC
    ADCON0 = 0b10110001;
             //10------          AN0, AN2 analog (ANS = 10)
             //--11----          clock = INTOSC/4 (ADCS = 11)
             //----00--          select channel AN0 (CHS = 00)
             //-------1          turn ADC on (ADON = 1)
```

However, as we have seen in the earlier lessons, the XC8 header files define most special function registers, including ADCON0, as unions of structures containing bit-fields corresponding to that register's bits.

Thus, we can configure the ADC module with:

```
// configure ADC
ADCON0bits.ADCS = 0b11;     // clock = INTOSC/4
ADCON0bits.ANS  = 0b10;     // AN0, AN2 analog
ADCON0bits.CHS  = 0b00;     // select channel AN0
ADCON0bits.ADON = 1;        // turn ADC on
                            // -> AN0 ready for sampling
```

Although this approach involves more statements, leading to a longer program and a larger executable, it has the advantage of clarity, is less prone to errors, and seems more "natural" when programming in C – so it's the method we'll use in the examples in this lesson. But as ever, which approach you use is a question of personal programming style – they're both valid.

Like MPASM, the XC8 device headers define more than one symbol for the GO/$\overline{\text{DONE}}$ bit.

In fact you can access it as any of:

```
ADCON0bits.GO_nDONE
ADCON0bits.GO
ADCON0bits.nDONE
```

As we did in baseline assembler lesson 10, we'll use the "GO" bit-field when starting the conversion:

```
ADCON0bits.GO = 1;          // start conversion
```

and we'll use the "nDONE" version of the bit-field when waiting for the conversion to finish:

```
while (ADCON0bits.nDONE)    // wait until done
    ;
```

even though they are referring to the same bit – the intent of the code is clearer this way.

The result of the conversion is available in ADRES, accessible through the 'ADRES' 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:

```
LEDS = ADRES >> 4;      // copy high nybble of result to LEDs
```

(having defined 'LEDS' as an alias for 'PORTC')

Complete program

Here is how the above code fragments fit together:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 7, 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)             *
*       RC0-3   = output LEDs (RC3 is MSB)                             *
*                                                                      *
************************************************************************/

#include <xc.h>


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

// Pin assignments
#define LEDS    PORTC           // output LEDs on RC0-RC3


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

    // configure ports
    TRISC = 0b110000;           // configure RC0-RC3 as outputs
    CM2CON0 = 0;                // disable comparator 2 -> RC0, RC1 digital
    VRCON = 0;                  // disable CVref -> RC2 usable

    // configure ADC
    ADCON0bits.ADCS = 0b11;     // clock = INTOSC/4
    ADCON0bits.ANS  = 0b10;     // AN0, AN2 analog
    ADCON0bits.CHS  = 0b00;     // select channel AN0
    ADCON0bits.ADON = 1;        // turn ADC on
                                // -> AN0 ready for sampling


    //*** Main loop
    for (;;)
    {
        // sample analog input
        ADCON0bits.GO = 1;          // start conversion
        while (ADCON0bits.nDONE)    // wait until done
            ;

        // display result on 4 x LEDs
        LEDS = ADRES >> 4;      // copy high nybble of result to LEDs
    }
}
```

CCS PCB

We saw in the lesson 6 that the CCS compiler provides a built-in function, '`setup_comparator()`', which can be used to disable comparator 2 (so that we can use RC0 and RC1 as digital outputs):

```
    setup_comparator(NC_NC_NC_NC);  // disable comparators -> RC0, RC1 digital
```

Note that this command actually disables both comparators, but since comparator 1 is not used in this example, there is no reason to enable it.

Similarly, the '`setup_vref()`' function can be used to disable the CVREF output, making RC2 usable:

```
setup_vref(FALSE);              // disable CVref -> RC2 usable
```

A number of built-in functions are used to configure the ADC module.

The '`setup_adc_ports()`' function is used to select which ports are configured as analog inputs.

It is called with one of the symbols defined in the device's header file. For example, "16F506.h" contains:

```
// Constants used in SETUP_ADC_PORTS() are:
#define AN0_AN1_AN2                     0xc0   // A0 A1 A2
#define AN0_AN2                         0x80   // A0 A2
#define AN2                             0x40   // A2
#define NO_ANALOGS                         0   // None
```

In this case, we want the AN0 and AN2 configuration, so we use:

```
setup_adc_ports(AN0_AN2);       // configure AN0 and AN2 for analog input
```

Note that, if you wanted to disable all the analog inputs, you would use:

```
setup_adc_ports(NO_ANALOGS);    // no analog inputs (all digital)
```

The '`setup_adc()`' function is used to select the ADC clock source, or to turn the ADC module off (useful for saving power in sleep mode).

It is also called with a symbol defined in the device's header file. For example, "16F506.h" contains:

```
// Constants used for SETUP_ADC() are:
#define ADC_OFF                 0          // ADC Off
#define ADC_CLOCK_DIV_32    0x00
#define ADC_CLOCK_DIV_16    0x10
#define ADC_CLOCK_DIV_8     0x20
#define ADC_CLOCK_INTERNAL  0x30           // Internal 2-6us
```

In this case we want the internal clock source, so we use:

```
setup_adc(ADC_CLOCK_INTERNAL);  // select INTOSC/4 clock and turn ADC on
```

Note that the ADC is implicitly being turned on by this function. If you don't want it turned on, you need to explicitly turn it off, with:

```
setup_adc(ADC_OFF);             // turn ADC module off
```

The '`set_adc_channel()`' function is used to select the ADC input channel.

The parameter corresponds to the value of the CHS channel selection bits, as defined in the device data sheet (and, for the 16F506, in the table above).

In this case, we want channel 0, corresponding to AN0, so we use:

```
set_adc_channel(0);             // ADC channel = AN0
```

Initiating the conversion, waiting for it to complete, then returning the result can be done with a single built-in function: '`read_adc()`'.

It can optionally be passed one of the symbols defined in the header file, for example:

```
// Constants used in READ_ADC() are:
#define ADC_START_AND_READ     7  // This is the default if nothing is specified
#define ADC_START_ONLY         1
#define ADC_READ_ONLY          6
```

This means that you can start a conversion with:

```
    read_adc(ADC_START_ONLY);       // start ADC conversion
```

and do something else while waiting for the conversion to complete (indicated by the '`adc_done()`' built-in function), and then read the result with something like:

```
    result = read_adc(ADC_READ_ONLY);  // read ADC result
```

In this case, we want to initiate the conversion and then read the result in a single operation, so to sample the input and place the upper four bits of the result in the lower four bits of PORTC, we can write:

```
    output_c(read_adc()>>4);  // read ADC and copy high nybble of result to LEDs
```

Note that there is no need to specify '`ADC_START_AND_READ`' as the parameter to '`read_adc()`', since it is the default if nothing is specified.

Complete program

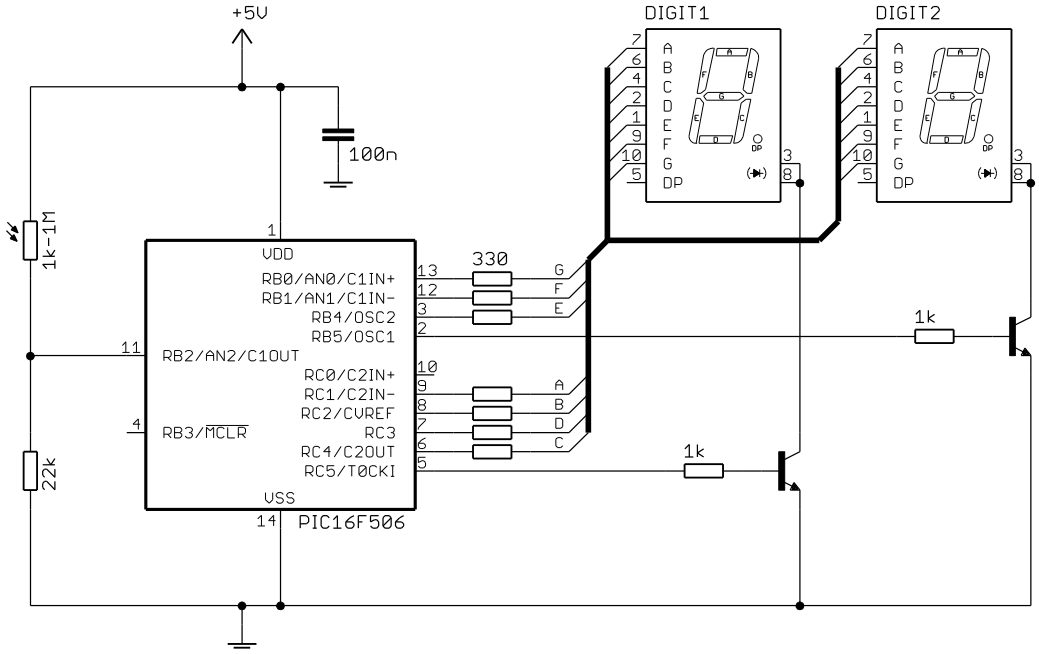
Here is how these code fragments fit together in the CCS version of the "4 LEDs ADC demo" program:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 7, 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 <16F506.h>


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


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

    // configure ports
    setup_comparator(NC_NC_NC_NC);  // disable comparators -> RC0, RC1 digital
    setup_vref(FALSE);              // disable CVref -> RC2 usable
```

```
    // configure ADC
    setup_adc(ADC_CLOCK_INTERNAL);  // clock = INTOSC/4, turn ADC on
    setup_adc_ports(AN0_AN2);       // AN0, AN2 analog
    set_adc_channel(0);             // select channel AN0
                                    // -> AN0 ready for sampling


    //*** Main loop
    while (TRUE)
    {
        // sample and display analog input
        output_c(read_adc() >> 4);  // read ADC and copy result to LEDs
    }
}
```

Hexadecimal Output

To add a more useful, human-readable output to the ADC demo, the second example in baseline assembler lesson 10 implemented a two-digit hexadecimal display, based on the multiplexed 7-segment display circuit from baseline assembler lesson 8, dropping one digit, and adding a photocell and resistor to supply a voltage that increases with light level, as shown below:



To implement this circuit using the Gooligum baseline training board, place shunts:

- across every position (all six of them) of jumper block JP4, connecting segments A-D, F and G to pins RB0-1 and RC1-4
- in position 1 ('RA/RB4') of JP5, connecting segment E to pin RB4
- across pins 2 and 3 ('RC5') of JP6, connecting digit 1 to the transistor controlled by RC5
- in jumpers JP8 and JP9, connecting pins RC5 and RB5 to their respective transistors
- in position 1 ('AN2') of JP25, connecting photocell PH2 to AN2.

All other shunts should be removed.

The source code was also adapted from the timer-based 7-segment display multiplexing routines presented in baseline assembler lesson 8, with the only important differences being:

- the value to be displayed was now the result of an analog-to-digital conversion, performed using the code from the first example (above), instead of a time count;
- 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';

XC8

The previous example included initialisation code to disable comparator 2 and the programmable voltage reference. Extending this to also disable comparator 1 is simply:

```
CM1CON0 = 0;                    // disable comparator 1 -> RB0, RB1 digital
CM2CON0 = 0;                    // disable comparator 2 -> RC0, RC1 digital
VRCON = 0;                      // disable CVref -> RC2 usable
```

We also need to configure the ADC, but this time with AN2 as the only analog input:

```
// configure ADC
ADCON0bits.ADCS = 0b11;         // clock = INTOSC/4
ADCON0bits.ANS  = 0b01;         // AN2 (only) analog
ADCON0bits.CHS  = 0b10;         // select channel AN2
ADCON0bits.ADON = 1;            // turn ADC on
                                // -> AN2 ready for sampling
```

The ADC input is sampled, using code from the previous example:

```
// sample input
ADCON0bits.GO = 1;              // start conversion
while (ADCON0bits.nDONE)        // wait until done
    ;
```

Then the result is displayed, using code adapted from lesson 5:

```
// display high nybble for 2.048 ms
while (!TMR0_2)             // wait for TMR0<2> to go high
    ;
set7seg(ADRES >> 4);        // output high nybble of result
TENS_EN = 1;                // enable "tens" digit
while (TMR0_2)              // wait for TMR0<2> to go low
    ;

// display low nybble for 2.048 ms
while (!TMR0_2)             // wait for TMR0<2> to go high
    ;
set7seg(ADRES & 0x0F);      // output low nybble of result
ONES_EN = 1;                // enable ones digit
while (TMR0_2)              // wait for TMR0<2> to go low
    ;
```

The '`set7seg()`' function is much the same as that presented in lesson 5, but with the pattern arrays (lookup tables) now extended from 10 to 16 entries, adding the 7-segment representations of the letters 'A' to 'F'.

Complete program

Here is the complete XC8 version of the "ADC demo with hexadecimal output" program, showing how these code fragments – mostly adapted from previous programs – fit together:

```
/************************************************************************
*   Description:    Lesson 7, example 2                                 *
*                                                                       *
*   Displays ADC output in hexadeximal on 7-segment LED displays        *
*                                                                       *
*   Continuously samples analog input,                                  *
*   displaying result as 2 x hex digits on multiplexed 7-seg displays   *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       AN2             = voltage to be measured (e.g. pot or LDR)      *
*       RB0-1,RB4,RC1-4 = 7-segment display bus (common cathode)        *
*       RC5             = "tens" digit enable (active high)             *
*       RB5             = ones digit enable                             *
*                                                                       *
************************************************************************/

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


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

// Pin assignments
#define TENS_EN     PORTCbits.RC5   // "tens" (high nybble) digit enable
#define ONES_EN     PORTBbits.RB5   // ones digit enable


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


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


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

    // configure ports
    TRISB = 0;                      // configure PORTB and PORTC as all outputs
    TRISC = 0;
    CM1CON0 = 0;                    // disable comparator 1 -> RB0, RB1 digital
    CM2CON0 = 0;                    // disable comparator 2 -> RC0, RC1 digital
    VRCON = 0;                      // disable CVref -> RC2 usable

    // configure ADC
    ADCON0bits.ADCS = 0b11;         // clock = INTOSC/4
    ADCON0bits.ANS  = 0b01;         // AN2 (only) analog
    ADCON0bits.CHS  = 0b10;         // select channel AN2
    ADCON0bits.ADON = 1;            // turn ADC on
                                    // -> AN2 ready for sampling
```

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


    //*** Main loop
    for (;;)
    {
        // sample input
        ADCON0bits.GO = 1;          // start conversion
        while (ADCON0bits.nDONE)    // wait until done
            ;

        // display high nybble for 2.048 ms
        while (!TMR0_2)             // wait for TMR0<2> to go high
            ;
        set7seg(ADRES >> 4);        // output high nybble of result
        TENS_EN = 1;                // enable "tens" digit
        while (TMR0_2)              // wait for TMR0<2> to go low
            ;

        // display low nybble for 2.048 ms
        while (!TMR0_2)             // wait for TMR0<2> to go high
            ;
        set7seg(ADRES & 0x0F);      // output low nybble of result
        ONES_EN = 1;                // enable ones digit
        while (TMR0_2)              // wait for TMR0<2> to go low
            ;
    }
}


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

/***** Display digit on 7-segment display *****/
void set7seg(uint8_t digit)
{
    // pattern table for 7 segment display on port B
    const uint8_t pat7segB[16] = {
        // RB4 = E, RB1:0 = FG
        0b010010,   // 0
        0b000000,   // 1
        0b010001,   // 2
        0b000001,   // 3
        0b000011,   // 4
        0b000011,   // 5
        0b010011,   // 6
        0b000000,   // 7
        0b010011,   // 8
        0b000011,   // 9
        0b010011,   // A
        0b010011,   // b
        0b010010,   // C
        0b010001,   // d
        0b010011,   // E
        0b010011    // F
    };
```

```
    // pattern table for 7 segment display on port C
    const uint8_t pat7segC[16] = {
        // RC4:1 = CDBA
        0b011110,   // 0
        0b010100,   // 1
        0b001110,   // 2
        0b011110,   // 3
        0b010100,   // 4
        0b011010,   // 5
        0b011010,   // 6
        0b010110,   // 7
        0b011110,   // 8
        0b011110,   // 9
        0b010110,   // A
        0b011000,   // b
        0b001010,   // C
        0b011100,   // d
        0b001010,   // E
        0b000010    // F
    };

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

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

CCS PCB

Since the built-in '`setup_comparator()`' function can be used to disable both comparators with a single call, the code to disable the comparators and the voltage reference is the same as in the first example, above:

```
    setup_comparator(NC_NC_NC_NC);  // disable comps -> RB0-1, RC0-1 digital
    setup_vref(FALSE);              // disable CVref -> RC2 usable
```

In this example, the ADC has to be configured with AN2 as the only analog input:

```
    setup_adc(ADC_CLOCK_INTERNAL);  // clock = INTOSC/4, turn ADC on
    setup_adc_ports(AN2);           // AN2 (only) analog
    set_adc_channel(2);             // select channel AN2
```

Because we need to access the ADC result twice (once for each digit in the display), it makes sense to sample the input and store the result in a variable, for later reference:

```
    adc_res = read_adc();
```

This result is then displayed, using code adapted from lesson 5:

```
        // display high nybble for 2.048 ms
        while (!TMR0_2)           // wait for TMR0<2> to go high
            ;
        set7seg(adc_res >> 4);    // output high nybble of result
        output_high(TENS_EN);     // enable "tens" digit
```

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

        // display low nybble for 2.048 ms
        while (!TMR0_2)           // wait for TMR0<2> to go high
            ;
        set7seg(adc_res & 0x0F);  // output low nybble of result
        output_high(ONES_EN);     // enable ones digit
        while (TMR0_2)            // wait for TMR0<2> to go low
            ;
```

Note that, instead of storing the ADC result in a variable, we could have written:

```
    // display high nybble for 2.048 ms
    while (!TMR0_2)           // wait for TMR0<2> to go high
        ;
    set7seg(read_adc() >> 4); // sample input, then
                              //   output high nybble of result
    output_high(TENS_EN);     // enable "tens" digit
    while (TMR0_2)            // wait for TMR0<2> to go low
        ;

    // display low nybble for 2.048 ms
    while (!TMR0_2)           // wait for TMR0<2> to go high
        ;
    set7seg(read_adc(ADC_READ_ONLY) & 0x0F);  // output low nybble of result
    output_high(ONES);        // enable ones digit
    while (TMR0_2)            // wait for TMR0<2> to go low
        ;
```

This uses the '`read_adc()`' function to sample the input as part of the first digit display routine, and then uses the '`read_adc(ADC_READ_ONLY)`' form of the function to return the already-sampled result, when displaying the second digit. However, although this approach saves a line of code and avoids the need to allocate a variable, it seems a little unwieldy. Again, it's really a question of personal style.

As in the XC8 example, the '`set7seg()`' function is much the same as that presented in lesson 5, but with the pattern arrays extended from 10 to 16 entries.

Complete program

Here is the complete CCS version of the "ADC demo with hexadecimal output" program, showing how these code fragments – again mostly adapted from previous programs – fit together:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 7, example 2                                 *
*                                                                       *
*   Displays ADC output in hexadeximal on 7-segment LED displays        *
*                                                                       *
*   Continuously samples analog input,                                  *
*   displaying result as 2 x hex digits on multiplexed 7-seg displays   *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       AN2             = voltage to be measured (e.g. pot or LDR)      *
*       RB0-1,RB4,RC1-4 = 7-segment display bus (common cathode)        *
```

```
*       RC5             = "tens" digit enable (active high)             *
*       RB5             = ones digit enable                             *
*                                                                       *
************************************************************************/

#include <16F506.h>


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

// Pin assignments
#define TENS_EN     PIN_C5          // "tens" (high nybble) enable
#define ONES_EN     PIN_B5          // ones enable


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


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


/***** MAIN PROGRAM *****/
void main()
{
    unsigned int8   adc_res;        // result of ADC conversion

    //*** Initialisation

    // configure ports
    setup_comparator(NC_NC_NC_NC);  // disable compss -> RB0-1, RC0-1 digital
    setup_vref(FALSE);              // disable CVref -> RC2 usable

    // configure ADC
    setup_adc(ADC_CLOCK_INTERNAL);  // clock = INTOSC/4, turn ADC on
    setup_adc_ports(AN2);           // AN2 (only) analog
    set_adc_channel(2);             // select channel AN2
                                    // -> AN2 ready for sampling

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

    //*** Main loop
    while (TRUE)
    {
        // sample input
        adc_res = read_adc();

        // display high nybble for 2.048 ms
        while (!TMR0_2)           // wait for TMR0<2> to go high
            ;
        set7seg(adc_res >> 4);    // output high nybble of result
        output_high(TENS_EN);     // enable "tens" digit
        while (TMR0_2)            // wait for TMR0<2> to go low
            ;

        // display low nybble for 2.048 ms
```

```
        while (!TMR0_2)           // wait for TMR0<2> to go high
            ;
        set7seg(adc_res & 0x0F);  // output low nybble of result
        output_high(ONES_EN);     // enable ones digit
        while (TMR0_2)            // wait for TMR0<2> to go low
            ;
    }
}


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

/***** Display digit on 7-segment display *****/
void set7seg(unsigned int8 digit)
{
    // pattern table for 7 segment display on port B
    const int8 pat7segB[16] = {
        // RB4 = E, RB1:0 = FG
        0b010010,   // 0
        0b000000,   // 1
        0b010001,   // 2
        0b000001,   // 3
        0b000011,   // 4
        0b000011,   // 5
        0b010011,   // 6
        0b000000,   // 7
        0b010011,   // 8
        0b000011,   // 9
        0b010011,   // A
        0b010011,   // b
        0b010010,   // C
        0b010001,   // d
        0b010011,   // E
        0b010011    // F
    };

    // pattern table for 7 segment display on port C
    const int8 pat7segC[16] = {
        // RC4:1 = CDBA
        0b011110,   // 0
        0b010100,   // 1
        0b001110,   // 2
        0b011110,   // 3
        0b010100,   // 4
        0b011010,   // 5
        0b011010,   // 6
        0b010110,   // 7
        0b011110,   // 8
        0b011110,   // 9
        0b010110,   // A
        0b011000,   // b
        0b001010,   // C
        0b011100,   // d
        0b001010,   // E
        0b000010    // F
    };

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

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

Comparisons

Here is the resource usage 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	96	86	1
XC8 (Free mode)	68	161	2
CCS PCB	63	135	8

Despite the different approaches of the two C compilers (direct register access versus built-in functions), the source code written for XC8 is much the same length as that for CCS PCB, and around two thirds the length of the assembler source. On the other hand, the optimised code generated by the CCS compiler is more than 50% larger than the assembler version.

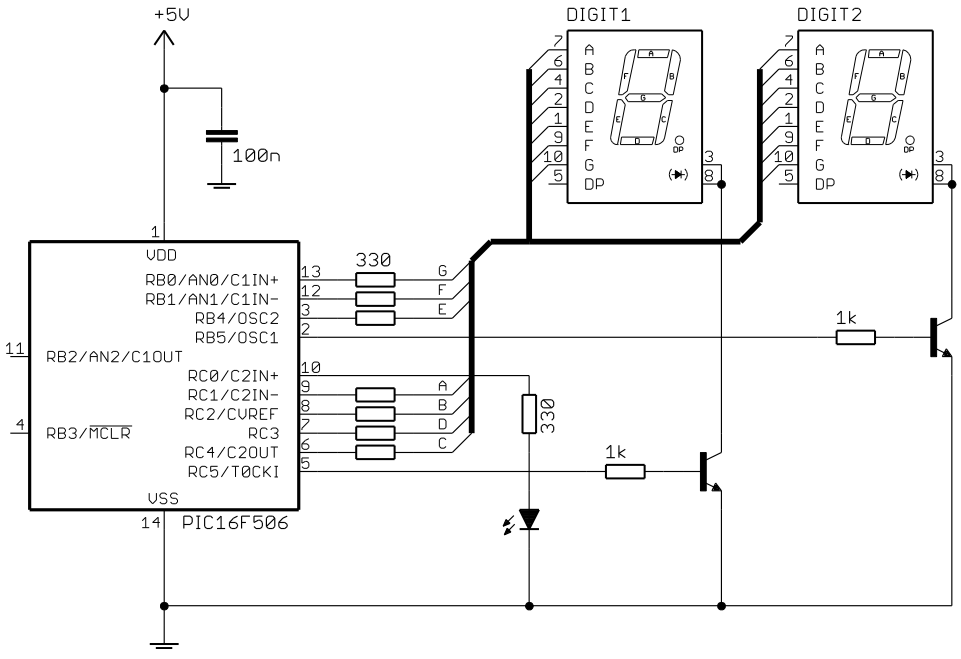
Measuring Supply Voltage

The fact that the absolute 0.6 V reference can be selected as an ADC input channel means that it can be used 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 is 0.6 V ÷ 5.0 V × 255 = 30.

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

The circuit shown below was used in baseline assembler lesson 10 to demonstrate this.



If you are using the Gooligum baseline training board, you should set it up as in the last example, but remove the shunt from JP25 (disconnecting the photocell from AN2) and close JP16 (connecting the LED on RC0).

As in the last example, the ADC result (now representing the value of the 0.6 V reference) is displayed in hex on the 7-segment displays, but to indicate low voltage, the LED on RC0 is lit if VDD falls below 3.5 V.

XC8

Most of the program code is the same as that in the previous example, but because we are now sampling the internal 0.6 V reference instead of AN0, the ADC has to be configured differently:

```
// configure ADC
ADCON0bits.ADCS = 0b11;         // clock = INTOSC/4
ADCON0bits.ANS  = 0b00;         // no analog inputs -> RB0-2 digital
ADCON0bits.CHS  = 0b11;         // select 0.6 V reference
ADCON0bits.ADON = 1;            // turn ADC on
                                // -> 0.6 V reference ready for sampling
```

The code to sample the ADC and output the result on the 7-segment displays is the same as before, but we need to add some code to test for the under-voltage condition (VDD < 3.5 V).

In the assembler example, the minimum allowable VDD 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 VDD value to a constant which could be used to compare the ADC result with also had to be written using only integers:

```
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 VDD as a floating-point constant:

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

and to then write the ADC comparison as:

```
        if (ADRES > 0.6/MINVDD*255)     // if measured 0.6V > threshold
            WARN = 1;                   //   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 VDD 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 being 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 `ADRES` into floating-point form, passing it to a floating-point comparison routine. This greatly increases the size of the generated code, blowing out to 508 words of program memory[2]! Compare this with the previous example, which is almost identical –

[2] using XC8 v1.01 running in 'Free mode'

lacking only this comparison routine – but required only 161 words of program memory. You wouldn't expect that adding such a simple routine would more than triple 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 (ADRES > (int)(0.6/MINVDD*255))  // if measured 0.6 V > threshold
    WARN = 1;                       //   light warning LED
```

This simple change prevents the compiler from including floating-point code, reducing the size of the generated code from 508 to only 165 words of program memory!

Program listing

The only change to the program setup (device configuration, function prototypes etc.) from the previous example is the addition of the following constant definition:

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

Most of the rest of the source code is identical to the previous example, but it is worth looking at the main program code, so that you can see the new ADC configuration and how the comparison code fits into the sample and display loop:

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

    // configure ports
    TRISB = 0;                      // configure PORTB and PORTC as all outputs
    TRISC = 0;
    CM1CON0 = 0;                    // disable comparator 1 -> RB0, RB1 digital
    CM2CON0 = 0;                    // disable comparator 2 -> RC0, RC1 digital
    VRCON = 0;                      // disable CVref -> RC2 usable

    // configure ADC
    ADCON0bits.ADCS = 0b11;         // clock = INTOSC/4
    ADCON0bits.ANS  = 0b00;         // no analog inputs -> RB0-2 digital
    ADCON0bits.CHS  = 0b11;         // select 0.6 V reference
    ADCON0bits.ADON = 1;            // turn ADC on
                                    // -> 0.6 V reference ready for sampling

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

```
    //*** Main loop
    for (;;)
    {
        // sample 0.6 V reference
        ADCON0bits.GO = 1;          // start conversion
        while (ADCON0bits.nDONE)    // wait until done
            ;

        // test for low Vdd
        if (ADRES > (int)(0.6/MINVDD*255))  // if measured 0.6 V > threshold
            WARN = 1;                       //   light warning LED

        // display high nybble for 2.048 ms
        while (!TMR0_2)         // wait for TMR0<2> to go high
            ;
        set7seg(ADRES >> 4);    // output high nybble of result
        TENS_EN = 1;            // enable "tens" digit
        while (TMR0_2)          // wait for TMR0<2> to go low
            ;

        // display low nybble for 2.048 ms
        while (!TMR0_2)         // wait for TMR0<2> to go high
            ;
        set7seg(ADRES & 0x0F);  // output low nybble of result
        ONES_EN = 1;            // enable ones digit
        while (TMR0_2)          // wait for TMR0<2> to go low
            ;
    }
}
```

CCS PCB

The initialisation code is much the same as in the previous example, except that we must now select the 0.6 V reference as the ADC input channel, instead of AN2:

```
    // configure ADC:
    setup_adc(ADC_CLOCK_INTERNAL);  // clock = INTOSC/4, turn ADC on
    setup_adc_ports(NO_ANALOGS);    // no analog inputs -> RB0-2 digital
    set_adc_channel(3);             // select 0.6 V reference
                                    // -> 0.6 V reference ready for sampling
```

The main sample and display loop is reused from the previous example, but, again, we need to insert some code to check that V_{DD} is above the minimum allowed value.

The minimum allowable V_{DD} can be defined as:

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

and the ADC result tested, in a similar way to how it was initially written using XC8, above:

```
        // test for low Vdd
        if (adc_res > 0.6/MINVDD*255)       // if measured 0.6 V > threshold
            output_high(WARN);              //   light warning LED
```

Just as in the XC8 example, the use of the floating-point expression '`0.6/MINVDD*255`' in the comparison causes the compiler to incorporate floating-point routines, making the generated code significantly larger

than it needs to be – 258 words of program memory[3], compared with only 135 words for the previous hexadecimal output example.

In the same way as was done with XC8, the unnecessary use of floating-point code can be avoided by casting the expression as an integer:

```
        if (adc_res > (int)(0.6/MINVDD*255))    // if measured 0.6 V > threshold
            output_high(WARN);                  //   light warning LED
```

Without the floating-point code, the size of the generated program is reduced to only 145 words of program memory.

Program listing

As in the XC8 version, the only change to the program setup (device configuration, function prototypes etc.) from the previous example is the addition of the following constant definition:

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

And again, most of the rest of the source code is the same as in the previous example, but it is worth listing the main program code, to see the new ADC configuration and how the comparison code fits in:

```
/***** MAIN PROGRAM *****/
void main()
{
    unsigned int8   adc_res;        // result of ADC conversion

    //*** Initialisation

    // configure ports
    setup_comparator(NC_NC_NC_NC);  // disable comps -> RB0-1, RC0-1 digital
    setup_vref(FALSE);              // disable CVref -> RC2 usable

    // configure ADC:
    setup_adc(ADC_CLOCK_INTERNAL);  // clock = INTOSC/4, turn ADC on
    setup_adc_ports(NO_ANALOGS);    // no analog inputs -> RB0-2 digital
    set_adc_channel(3);             // select 0.6 V reference
                                    // -> 0.6 V reference ready for sampling

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

    //*** Main loop
    while (TRUE)
    {
        // sample 0.6 V reference
        adc_res = read_adc();

        // test for low Vdd
        if (adc_res > (int)(0.6/MINVDD*255))    // if measured 0.6 V > threshold
            output_high(WARN);                  //   light warning LED

        // display high nybble for 2.048 ms
        while (!TMR0_2)           // wait for TMR0<2> to go high
            ;
        set7seg(adc_res >> 4);    // output high nybble of result
        output_high(TENS_EN);     // enable "tens" digit
```

[3] using CCS PCB v4.073

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

        // display low nybble for 2.048 ms
        while (!TMR0_2)           // wait for TMR0<2> to go high
            ;
        set7seg(adc_res & 0x0F);  // output low nybble of result
        output_high(ONES_EN);     // enable ones digit
        while (TMR0_2)            // wait for TMR0<2> to go low
            ;
    }
}
```

Comparisons

Here is the resource usage comparison for the "VDD measure" example, including the floating-point and integer arithmetic versions of the C programs:

ADC_Vdd-measure

Assembler / Compiler	Arithmetic	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	integer	104	90	1
XC8 (Free mode)	float	72	508	20
XC8 (Free mode)	integer	72	165	2
CCS PCB	float	67	258	15
CCS PCB	integer	67	145	9

The C source code continues to be significantly shorter than the assembly language version source, and the optimised code generated by the CCS compiler is still more than 50% larger than the assembly version. The real story here, however, is how very inefficient the floating-point versions are, in comparison with integer arithmetic, showing that floating-point operations should be avoided wherever possible.

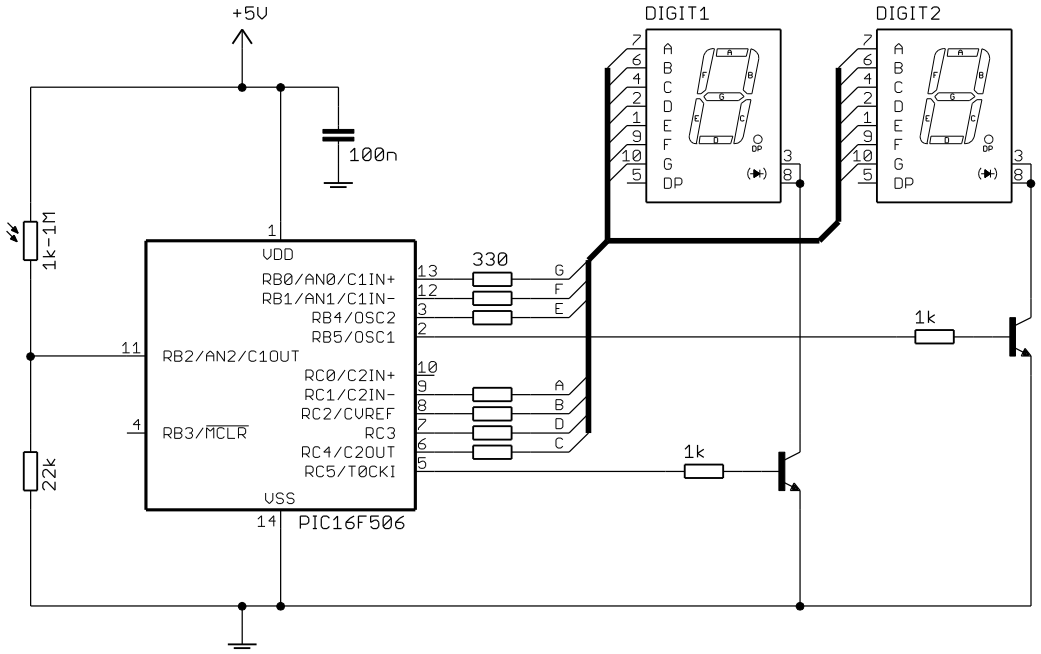
Decimal Output

The light meter presented earlier would be more useful if the light level was represented as a decimal value, instead of hexadecimal. Although we could add a third digit, so that the ADC output between 0 and 255 can be displayed directly in decimal, it would be more meaningful to most people if the result was scaled to a 2-digit result, with the full range being 0 – 99.

The circuit from the hexadecimal output example (shown again on the next page) can be re-used for this. If you are using the Gooligum baseline training board, you should set it up the same way as in that example.

This example was implemented in assembly language in baseline assembler lesson 11, where the main focus of the lesson was on integer arithmetic, including multi-byte addition and subtraction, and 8-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 ADC output from 0 – 255 to 0 – 99, it should be multiplied by 99/255. That can be done easily in C, but it is more difficult to do in assembler. In the assembler example, the ADC result was multiplied by



100/256, which is much easier to implement and is only "out" by 0.6%; not really significant, given that the ADC is only accurate to within 0.8%, in any case.

So that the C examples are comparable to the assembler version, we will use the scaling factor of 100/256 here, as well.

XC8

Most of the XC8 program code can be re-used from the hexadecimal output example.

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 we can declare as:

```
uint8_t     adc_dec;            // scaled ADC output (0-99)
```

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

To scale the ADC result, we could use:

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

However, the XC8 compiler generates smaller code if this is written as:

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

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

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.

The reason for this "integral promotion" is clear, when we consider how this expression might be evaluated. If the compiler calculates '`ADRES * 100`' first, it is likely to evaluate to a value greater than 255, which would overflow an 8-bit calculation, leading to incorrect results. Using 16-bit integers to perform these intermediate calculations avoids such problems.

However, C compilers will generally avoid integral promotion in situations where they can conclude that the result will be the same if promotion doesn't occur.

In this case, casting ADRESH as an unsigned integer allows the compiler to optimise its code generation, because it 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.

Note though that you can't simply assume that a particular change, like this, will make your code smaller – it depends on the specific compiler and its optimisation settings. Sometimes you need to try a number of combinations of type declarations and casting, if you want to generate the smallest possible code.

We then need to extract each digit of the scaled result for display. As we saw in lesson 5, this can be done using the integer division (/) and modulus (%) operators.

This is best shown in context, within the complete sample and display loop:

```
    //*** Main loop
    for (;;)
    {
        // sample input
        ADCON0bits.GO = 1;              // start conversion
        while (ADCON0bits.nDONE)        // wait until done
            ;

        // scale result to 0-99
        adc_dec = (unsigned)ADRES * 100/256;

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

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

Again, the `adc_dec` variable has been cast as an unsigned integer in each expression, to optimise code generation.

Finally, because only the decimal digits (0-9) need to be displayed, the additional hexadecimal digits (A-F) can be removed from the lookup tables in the digit display function:

```
/***** Display digit on 7-segment display *****/
void set7seg(uint8_t digit)
{
    // pattern table for 7 segment display on port B
    const uint8_t pat7segB[10] = {
        // RB4 = E, RB1:0 = FG
        0b010010,   // 0
        0b000000,   // 1
        0b010001,   // 2
```

```
        0b000001,   // 3
        0b000011,   // 4
        0b000011,   // 5
        0b010011,   // 6
        0b000000,   // 7
        0b010011,   // 8
        0b000011    // 9
    };

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

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

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

CCS PCB

In the CCS version of the hexadecimal example, the result of the ADC conversion was stored in a variable:

```
adc_res = read_adc();
```

Instead of scaling this value and storing the result in another variable, it makes more sense to sample the analog input and scale the result in a single operation, such as:

```
adc_dec = read_adc()*100/256;
```

where the variable, '`adc_dec`', has been declared in the same way as '`adc_res`' had been:

```
unsigned int8   adc_dec;        // scaled ADC output (0-99)
```

However, you will find that this doesn't work! This code, as written, always sets '`adc_dec`' equal to zero.

This happens because the CCS compiler does not perform automatic integral promotion, in the same way that the XC8 compiler does. The '`read_adc()`' function returns an 8-bit result, and the expression '`read_adc()*100/256`' is evaluated using 8-bit arithmetic operations. Any 8-bit quantity divided by 256 (equivalent to right-shifting it eight times) will always be equal to zero, which is the result we see here.

You might expect that this problem could be overcome by defining '`adc_dec`' as a 16-bit '`int16`' or '`long`' type, but unfortunately that doesn't affect how the expression '`read_adc()*100/256`' is evaluated; it is still performed using 8-bit arithmetic, regardless of the type of variable it is assigned to.

The answer is to cast the result of the '`read_adc()`' function as a 16-bit type:

```
adc_dec = (int16)read_adc()*100/256;
```

This generates the correct result.

This type of problem can be quite difficult to find. You need to be careful in case intermediate values in integer expressions overflow – especially when using the CCS compiler, which, unlike the XC8 compiler, does not automatically promote small integers into larger types.

As in the XC8 version, the digits of the scaled result can be extracted using the integer division (/) and modulus (%) operators.

Again, this is best shown in context, within the complete sample and display loop:

```
// Main loop
while (TRUE)
{
    // sample input and scale to 0-99
    adc_dec = (int16)read_adc()*100/256;

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

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

And finally, the additional hexadecimal digits (A-F) can be removed from the lookup tables in the digit display function:

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

```
    // pattern table for 7 segment display on port C
    const int8 pat7segC[10] = {
        // RC4:1 = CDBA
        0b011110,   // 0
        0b010100,   // 1
        0b001110,   // 2
        0b011110,   // 3
        0b010100,   // 4
        0b011010,   // 5
        0b011010,   // 6
        0b010110,   // 7
        0b011110,   // 8
        0b011110    // 9
    };
    // disable displays
    output_b(0);                    // clear all digit enable lines on PORTB
    output_c(0);                    //  and PORTC

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

Comparisons

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

ADC_dec-out

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	115	103	7
XC8 (Free mode)	58	423	8
CCS PCB	51	185	15

In this example, where integer arithmetic is involved, the pros and cons of assembler versus C become very apparent. The assembly source is around 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 – only 56% the size of the optimised CCS version. It is also clear that the XC8 compiler, when running in 'Free mode', generates very inefficient code in this example.

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 output 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 samples and displays the changing light level 244 times per second.

As we saw in baseline assembler lesson 11, 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 easy 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 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 data memory in the PIC16F506 is divided into four banks of 16 registers (plus three shared registers), the largest array that can be allocated as a single object is 16 bytes. That is, we can only easily store the last 16 samples. Since the input is sampled every 4 ms, our filter's window is 16 × 4 ms = 64 ms. This is more than enough to smooth out a 50 Hz flicker, since a 50 Hz signal has a period of only 20 ms.

XC8

To start with, we need to declare the sample array:

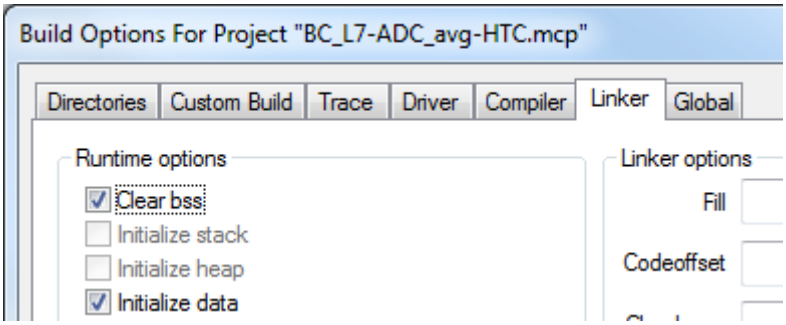
```
#define NSAMPLES    16              // size of sample array

uint8_t smp_buf[NSAMPLES];          // array of samples for moving average
```

Defining the constant, '`NSAMPLES`', toward the start of the program, makes it easier to change the number of samples from 16 later, if desired.

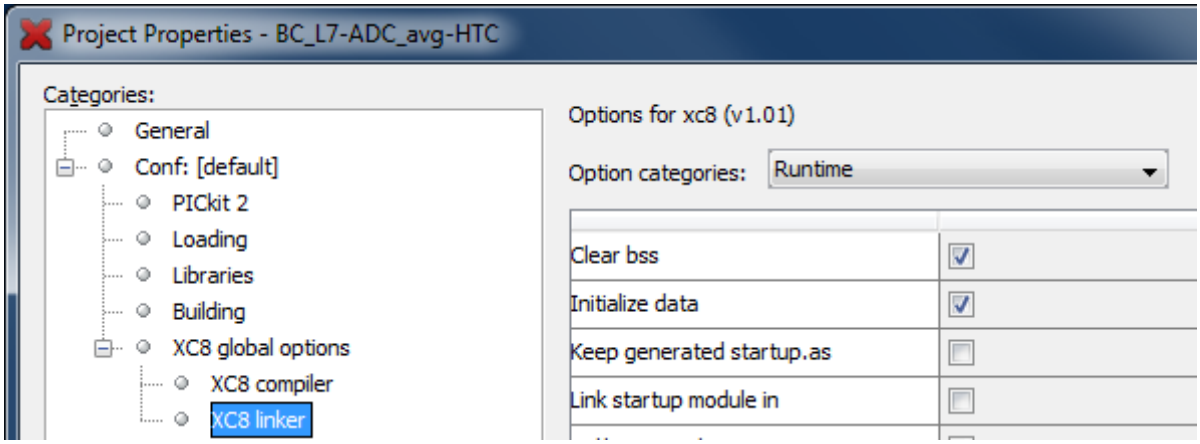
The sample array has to be cleared before it can be used, so that the running total is correct (if the running total is initially zero, the array elements must initially sum to zero; this is easiest to ensure if they are all initially equal to zero). But there is no need to include explicit code to clear the array. All we need to do is to make it a global variable, by declaring it outside any function, including `main()`.

By default, XC8 adds runtime code which, among other things, clears all uninitialized global and static variables, including arrays.

You can check that this option is selected in MPLAB 8 by looking at the "Linker" tab in the project's build options (Project → Build Options… → Project), as shown on the right.



Or, if you are using MPLAB X, you will find the equivalent option within the "Linker" category of the project properties (File → Project Properties, or click on the Project Properties button on the left side of the project dashboard), as shown below:



Whichever version of MPLAB you are using, if the "Clear bss" linker option is selected, the compiler-provided runtime code will clear all the variables.

In addition to the '`adc_dec`' variable from the last example, we will need variables to store the running total and to keep track of the current sample (used as an index into the sample array):

```
uint16_t    sum = 0;            // running total of ADC samples
uint8_t     adc_dec;            // scaled average (0-99)
uint8_t     s;                  // index into sample array
```

The running total (`sum`) is declared as an unsigned16-bit integer because it needs to be able to hold values up to 16 × 255 = 4080, which is too large for an 8-bit variable.

Note that it is zeroed as part of the variable declaration; this saves a line of code later.

The body of the sample and display loop has to be placed within a "`for`" loop (using '`s`' as the loop counter), so that each array element is accessed in turn:

```
for (s = 0; s < NSAMPLES; s++)
{
    // sample input
    ...
    // calculate moving average
    ...
    // display digits
}
```

Within the loop, after sampling the input, we update the running total and calculate the average, as follows:

```
// update running total
sum += ADRES - smp_buf[s];  // add new value and subtract old
smp_buf[s] = ADRES;         // update buffer with new value

// calculate average and scale to 0-99
adc_dec = sum / NSAMPLES * 100/256;
```

Complete program

Here is the complete source code for the XC8 version of the "ADC demo with averaged decimal output" program, showing where these code fragments fit in:

```
/************************************************************************
*   Description:    Lesson 7, example 5                                 *
*                                                                       *
*   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:                                                    *
*       AN2             = voltage to be measured (e.g. pot or LDR)      *
*       RB0-1,RB4,RC1-4 = 7-segment display bus (common cathode)        *
*       RC5             = tens digit enable (active high)               *
*       RB5             = ones digit enable                             *
*                                                                       *
************************************************************************/
```

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


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

// Pin assignments
#define TENS_EN     PORTCbits.RC5   // tens digit enable
#define ONES_EN     PORTBbits.RB5   // ones digit enable


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

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

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


/***** GLOBAL VARIABLES *****/
uint8_t smp_buf[NSAMPLES];          // array of samples for moving average


/***** MAIN PROGRAM *****/
void main()
{
    uint16_t    sum = 0;            // running total of ADC samples
    uint8_t     adc_dec;            // scaled average (0-99)
    uint8_t     s;                  // index into sample array

    //*** Initialisation

    // configure ports
    TRISB = 0;                      // configure PORTB and PORTC as all outputs
    TRISC = 0;
    CM1CON0 = 0;                    // disable comparator 1 -> RB0, RB1 digital
    CM2CON0 = 0;                    // disable comparator 2 -> RC0, RC1 digital
    VRCON = 0;                      // disable CVref -> RC2 usable

    // configure ADC
    ADCON0bits.ADCS = 0b11;         // clock = INTOSC/4
    ADCON0bits.ANS  = 0b01;         // AN2 (only) analog
    ADCON0bits.CHS  = 0b10;         // select channel AN2
    ADCON0bits.ADON = 1;            // turn ADC on
                                    // -> AN2 ready for sampling

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


    //*** Main loop
    for (;;)
```

```
    {
        for (s = 0; s < NSAMPLES; s++)
        {
            // sample input
            ADCON0bits.GO = 1;              // start conversion
            while (ADCON0bits.nDONE)        // wait until done
                ;

            // update running total
            sum += ADRES - smp_buf[s];      // add new value and subtract old
            smp_buf[s] = ADRES;             // update buffer with new value

            // calculate average and scale to 0-99
            adc_dec = sum / NSAMPLES * 100/256;

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

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

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

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

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

```
        0b010100,   // 4
        0b011010,   // 5
        0b011010,   // 6
        0b010110,   // 7
        0b011110,   // 8
        0b011110    // 9
    };

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

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

CCS PCB

By default, the CCS PCB compiler will only place variables (and arrays) in bank 0.

To instruct the compiler to use the other register banks, place a '`#device *=8`' directive near the start of the program:

```
#device *=8                         // allow variable placement in banks 1-3
```

Once this has been done, variables and arrays can be declared as usual, with the compiler automatically handling their placement.

We can then declare the sample buffer array as:

```
int8  smp_buf[NSAMPLES];            // array of samples for moving average
```

Unlike XC8, the CCS PCB compiler does not automatically clear uninitialized global variables, so it does not matter whether this array is made global or declared within `main()`. Regardless of where it is declared, we need to include a routine, as part of the program initialisation code, to clear the sample array:

```
    int8    s;                      // index into sample array


    // clear sample buffer
    for (s = 0; s < NSAMPLES; s++)
        smp_buf[s] = 0;
```

We also need to declare the variables needed for the moving average calculation:

```
    int8    adc_res;                // result of ADC conversion
    int16   sum = 0;                // running total of ADC samples
    int8    adc_dec;                // scaled average (0-99)
```

Note that '`sum`' has to be declared as an '`int16`' (or '`long`'), as this needs to be a 16-bit value. The other variables could be declared as '`char`' or '`int`', because CCS PCB defines both to be 8-bit types.

As we did in the XC8 example, we need to place the body of the sample and display loop within a "for" loop, to retrieve and update each array element in turn:

```
    for (s = 0; s < NSAMPLES; s++)
    {
        // sample ADC, calculate moving average, scale and display
    }
```

In theory, it should be possible to update the running total and then calculate and scale the moving average as follows:

```
// update running total
sum += (int16)adc_res - smp_buf[s]; // add new value and subtract old
smp_buf[s] = adc_res;               // update buffer with new value

// calculate average and scale to 0-99
adc_dec = sum / NSAMPLES * 100/256;
```

Unfortunately, this does not work! **The array is not written to correctly – apparently due to a bug in version 4.073 (and earlier) of the CCS PCB compiler.**

Until CCS releases, and makes freely available, a version of the PCB compiler which corrects this problem, we need to find another way to implement our 16-byte sample buffer.

Luckily, the PCB compiler provides two built-in functions, intended to allow efficient access to registers outside bank 0: '`read_bank()`' and '`write_bank()`'.

They are most useful in applications where an array would otherwise be used, such as implementing a buffer.

But before using these bank-access functions, we must ensure that the compiler will only use bank 0 by removing the '`#device *=8`' directive, so that there is no risk of overwriting registers used by the compiler.

Assuming that we will use bank 1 for the sample buffer, we first have to clear it:

```
// clear sample buffer
    for (s = 0; s < NSAMPLES; s++)
        write_bank(1,s,0);
```

The function '`write_bank(1,s,0)`' writes the value '0' to the register at address offset '`s`' in bank 1, where address offset = 0 is the start of the bank (address 0x30 for bank 1).

The code to update the running total then becomes:

```
// update running total
sum += (int16)adc_res - read_bank(1,s); // add new val and subtract old
write_bank(1,s,adc_res);                // update buffer with new value
```

The function '`read_bank(1,s)`' returns the value in the register at address offset '`s`' in bank 1.

As you can see, the '`read_bank()`' and '`write_bank()`' functions can be substituted quite easily for array reads and writes.

Complete program

Here is the complete source code for the CCS version of the "ADC demo with averaged decimal output" program, using the direct bank-access functions, showing where these code fragments fit within the program:

```
/************************************************************************
*   Description:    Lesson 7, example 5b                                *
*                                                                       *
*   Displays smoothed ADC output in decimal on 2x7-seg 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-segment displays.                                  *
```

```
*   Uses bank read and write functions to implement sample buffer       *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       AN2             = voltage to be measured (e.g. pot or LDR)      *
*       RB0-1,RB4,RC1-4 = 7-segment display bus (common cathode)        *
*       RC5             = tens digit enable (active high)               *
*       RB5             = ones digit enable                             *
*                                                                       *
************************************************************************/

#include <16F506.h>


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

// Pin assignments
#define TENS_EN     PIN_C5          // tens digit enable
#define ONES_EN     PIN_B5          // ones digit enable


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


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


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


/***** MAIN PROGRAM *****/
void main()
{
    int8    adc_res;                // result of ADC conversion
    int16   sum = 0;                // running total of ADC samples
    int8    adc_dec;                // scaled average (0-99)
    int8    s;                      // index into sample buffer

    //*** Initialisation

    // configure ports
    setup_comparator(NC_NC_NC_NC);  // disable comps -> RB0-1, RC0-1 digital
    setup_vref(FALSE);              // disable CVref -> RC2 usable

    // configure ADC
    setup_adc(ADC_CLOCK_INTERNAL);  // clock = INTOSC/4, turn ADC on
    setup_adc_ports(AN2);           // AN2 (only) analog
    set_adc_channel(2);             // select channel AN2
                                    // -> AN2 ready for sampling

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

    // clear sample buffer
```

```
    for (s = 0; s < NSAMPLES; s++)
        write_bank(1,s,0);


    //*** Main loop
    while (TRUE)
    {
        for (s = 0; s < NSAMPLES; s++)
        {
            // sample input
            adc_res = read_adc();

            // update running total
            sum += (int16)adc_res - read_bank(1,s); // add new, subtract old
            write_bank(1,s,adc_res);                // update buffer with new

            // calculate average and scale to 0-99
            adc_dec = sum / NSAMPLES * 100/256;

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

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


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

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

    // pattern table for 7 segment display on port C
    const int8 pat7segC[10] = {
```

```
        // RC4:1 = CDBA
        0b011110,   // 0
        0b010100,   // 1
        0b001110,   // 2
        0b011110,   // 3
        0b010100,   // 4
        0b011010,   // 5
        0b011010,   // 6
        0b010110,   // 7
        0b011110,   // 8
        0b011110    // 9
    };

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

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

Comparisons

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

ADC_avg

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	150	136	26
XC8 (Free mode)	65	502	27
CCS PCB	61	257	35

In this example, the differences between C and assembly are even more pronounced. The assembly source is more than twice as long as the XC8 and CCS versions, while the assembled version is only around half the size of the optimised code generated by the CCS PCB compiler.

But it's also clear that, given the problems with compiler bugs and limitations encountered when implementing this example in C, we are hitting the limits of what can be achieved using C compilers on these small baseline devices – something that was not apparent when developing the assembly version.

Summary

The examples in this lesson demonstrate that it is possible to effectively perform analog to digital conversion on baseline PICs, such as the PIC16F506, using either of the XC8 or CCS C compilers. But we have also seen that, although all these compilers make it possible to implement buffers in memory outside bank 0, only the XC8 compiler is able to effectively work directly with "large" (16 byte) arrays.

As expected, source code written for the CCS compiler is consistently the shortest, due to the use of its built-in functions. However, the differences between the CCS and XC8 compilers are dwarfed by that between

assembler and C source, especially for more sophisticated programs, particularly when arithmetic expressions, which can be written succinctly in C, are heavily used:

Source code (lines)

Assembler / Compiler	ADC_hex_out	Vdd_measure	ADC_dec_out	ADC_avg
Microchip MPASM	96	104	115	150
XC8 (Free mode)	68	72	58	65
CCS PCB	63	67	51	61

But again, both C compilers generate code which is significantly larger than the corresponding hand-written assembler versions; the most complex programs being around twice the size of the assembler version, even for the CCS PCB compiler, with "optimised" code generation:

Program memory (words)

Assembler / Compiler	ADC_hex_out	Vdd_measure	ADC_dec_out	ADC_avg
Microchip MPASM	86	90	103	136
XC8 (Free mode)	161	165	423	502
CCS PCB	135	145	185	257

Data memory (bytes)

Assembler / Compiler	ADC_hex_out	Vdd_measure	ADC_dec_out	ADC_avg
Microchip MPASM	1	1	7	26
XC8 (Free mode)	2	2	8	27
CCS PCB	8	9	15	35

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

On the other hand, it certainly 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 CCS or XC8 compilers, on a device as small as the PIC16F506. The CCS PCB compilers had a problem with its implementation of banked array access, suggesting that the baseline PIC architecture just isn't well suited to the use of C for this type of application. Simple LED flashing and responding to key presses is fine, but when it comes to a moderately sophisticated application, involving analog to digital conversion, with simple digital filtering and scaling, while driving a multiplexed 7-segment display, we appear to have pushed the C compilers nearly as far as they will go. It seems that, to get the most from these baseline PICs, to reach their full potential, we need to use assembler. Or you could pay for the full (optimising) version of XC8, which did not require any workarounds to implement the moving average example, but, with optimisation disabled, generated code which used more than half the memory available on the 16F506.

For anything beyond the simplest applications, instead of trying to fit the solution into the baseline architecture, it often makes more sense to spend a little extra on the microcontroller in order to simplify the programming problem, by moving up to Microchip's "Mid-Range" PIC architecture.

These larger, more flexible microcontrollers are covered in the "Mid-Range PIC Architecture and Assembly Language" tutorial series, which introduces the mid-range PIC architecture, starting with the PIC12F629. We'll go back to flashing LEDs and responding to pushbutton switches, but we'll see how it can be done, using assembler, on a midrange device.

This is then followed up in the "Programming Mid-range PICs in C" tutorial series, where we cover the same ground again, using C.