

Introduction to PIC Programming

Programming Baseline PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 6: Analog-to-Digital Conversion and Simple Filtering

[Baseline 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 the free HI-TECH C¹ (in “Lite” mode), PICC-Lite and CCS PCB compilers (introduced in [lesson 1](#)).

It then shows how a simple moving-average filter, as described in [baseline 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 HI-TECH C, PICC-Lite and CCS PCB.

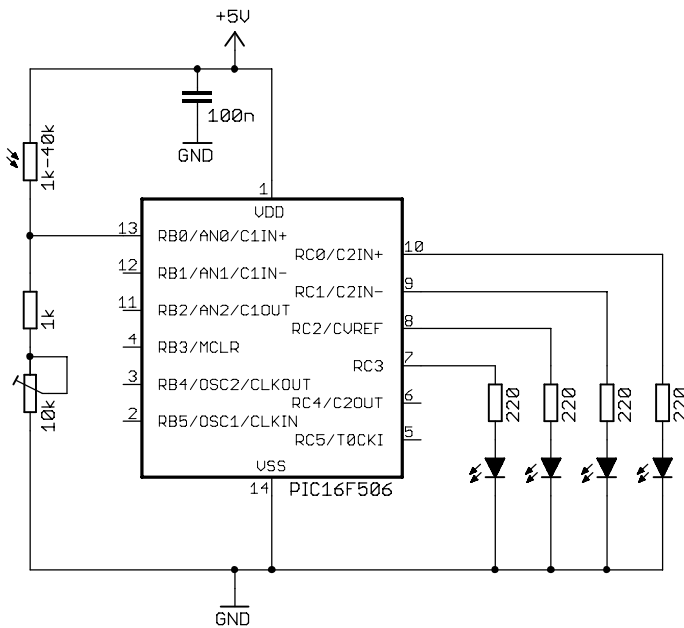
Analog-to-Digital Converter

As explained in more detail in [baseline 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 (or *converted*) at once.

A simple example in [baseline lesson 10](#) demonstrated basic ADC operation, using the circuit on the next page – similar to that used in [lesson 5](#) on comparators, but using four LEDs connected to RC0 – RC3 (labelled ‘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 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.

As you can see, the pins cannot be configured independently; only the listed combinations are possible.

You can also see that a quick way to disable the analog inputs is to clear ADCON0, since clearing ANS<1:0> deselects all the analog inputs.

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.

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

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 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 lesson 10](#), the ADC was configured with the above options with:

```

; configure ADC
movlw    b'10110001'
; 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

```

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

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

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

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

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 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` (which also disables `C2OUT`, making `RC4` usable):

```
clrf     CM2CON0      ; disable Comparator 2 (RC0, RC1, RC4 usable)
```

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)
```

HI-TECH C or PICC-Lite

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 is simply:

```

// Initialisation
TRISC = 0b110000;      // configure RC0-RC3 as outputs
CM2CON0 = 0;          // disable C2 inputs (RC0, RC1, RC4 usable)
VRCON = 0;            // disable CVref (RC2 usable)

```

Configuring the ADC module can then be done in the same way, by writing 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)

```

Alternatively, the individual bits in `ADCON0` can be written to, as we did with the comparator control bits in the [previous lesson](#):

```
// configure ADC
ANS0 = 0;           // AN0, AN2 analog (ANS = 10)
ANS1 = 1;
ADCS0 = 1;         // clock = INTOSC/4 (ADCS = 11)
ADCS1 = 1;
CHS0 = 0;         // select channel AN0 (CHS = 00)
CHS1 = 0;
ADON = 1;         // turn ADC on
```

This has the advantage of clarity, but it is seven lines of source code, translating to seven processor instructions, compared with just one or two instructions if `ADCON0` is loaded with a single value. Again, it is a question of personal programming style.

However, to begin the conversion, it is clearly best to access the `GO/DONE` bit directly:

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

We then wait until the `GO/DONE` bit is clear (something that can be done quite succinctly in C):

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

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 6, 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>

/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, 8 MHz int clock
__CONFIG(MCLREN & UNPROTECT & WDTDIS & INTOSC8 & INTIO);

```

```

// 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 C2 inputs (RC0, RC1, RC4 usable)
    VRCON = 0;                  // disable CVref (RC2 usable)

    // 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)

    // Main loop
    for (;;)
    {
        // sample analog input
        GODONE = 1;              // start conversion
        while (GODONE)           // 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 5](#) 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 usable)
```

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 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, to make the CCS version of the "4 LEDs ADC demo" program:

```

/*****
*
* Description: Lesson 6, 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, 8 MHz int clock
#fuses MCLR,NOPROTECT,NOWDT,INTRC_IO,IOSC8

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

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

    // configure ADC
    setup_adc_ports(AN0_AN2); // configure AN0 and AN2 for analog input
    setup_adc(ADC_CLOCK_INTERNAL); // select INTOSC/4 clock and turn ADC on
    set_adc_channel(0); // ADC channel = AN0

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

```

Comparisons

The following table summarises the resource usage for the “simple ADC demo” assembler and C examples:

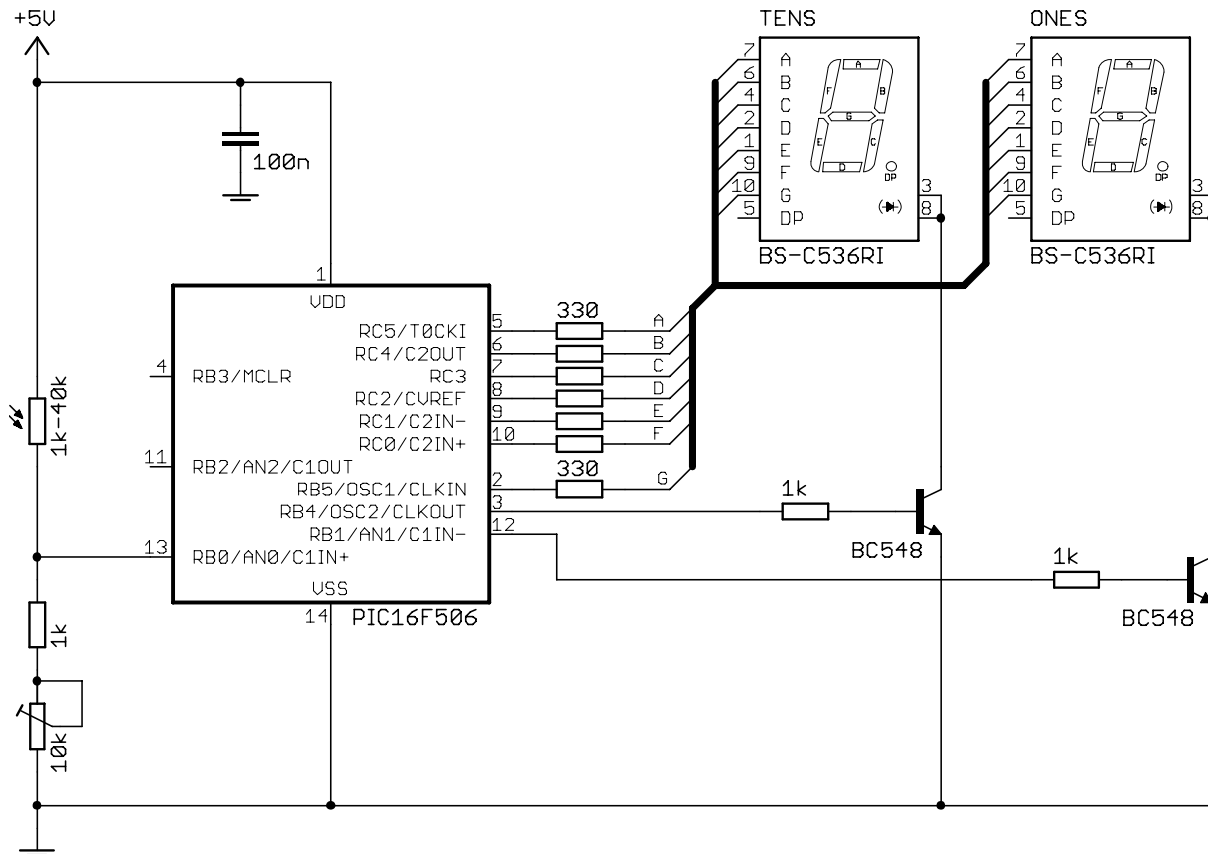
ADC_4LEDs

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	20	16	0
HI-TECH PICC-Lite	12	18	4
HI-TECH C (Lite)	12	40	4
CCS PCB	10	37	5

The PICC-Lite compiler in this case generates extremely efficient code, almost as short as the hand-written assembler version, from source code only around half the length of the assembler source. The code generated by the CCS compiler is much less efficient in this example – as is the unoptimised code generated by HI-TECH C running in “Lite” mode (as is to be expected).

Hexadecimal Output

To add a more useful, human-readable output to the ADC demo, the second example in [baseline lesson 10](#) implemented a two-digit hexadecimal display, based on the multiplexed 7-segment display circuit from [baseline lesson 8](#), shown below:



The source code was also adapted from the timer-based 7-segment display multiplexing routines presented in [baseline 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';
- since the processor was now a 16F506, with analog features, instead of the digital-only 16F505, those analog features (both comparators and the programmable voltage reference) had to be disabled to allow all pins, except AN0, AN2 and RB3, to be used as digital outputs to drive the display.

HI-TECH C or PICC-Lite

The example above 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, RB2 usable)
CM2CON0 = 0;           // disable comparator 2 (RC0, RC1, RC4 usable)
VRCON = 0;             // disable CVref (RC2 usable)
```

We also need to configure the ADC, as in the previous example:

```
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)
```

Most of the rest of the code is adapted from that presented in [lesson 4](#).

For example, setting up the 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)
```

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

```
// sample input
GODONE = 1;           // start conversion
while (GODONE)       // wait until done
;
```

Then the result is displayed, using code adapted from [lesson 4](#):

```
// 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 on segment bus
TENS = 1;            // enable tens digit to display it
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 on segment bus
    ONES = 1;             // enable ones digit to display it
    while (TMR0_2)       // wait for TMR0<2> to go low
        ;

```

The ‘set7seg()’ function is much the same as that presented in [lesson 4](#), extracting the pattern bits from the lookup array (now extended to 16 entries) and writing to the “segment bus” output pins:

```

// Extract pattern bits and write to segment bus pins
RB5 = pat7seg[digit] & 0b0000001;
PORTC = pat7seg[digit] >> 1;

```

Complete program

Here is the complete HI-TECH C version of the “ADC demo with hexadecimal output” program, showing how these code fragments – mostly adapted from previous programs – fit together:

```

/*****
 *
 * Description: Lesson 6, example 2
 *
 * Displays ADC output in hexadecimal on 7-segment LED displays
 *
 * Continuously samples analog input,
 * displaying result as 2 x hex digits on multiplexed 7-seg displays
 *
 *****/
 *
 * Pin assignments:
 * AN0 = voltage to be measured (e.g. pot or LDR)
 * RB5, RC0-5 = 7-segment display bus (active high)
 * RB4 = "tens" (high nybble) enable (active high)
 * RB1 = ones enable
 *
 *****/

#include <htc.h>

/***** CONFIGURATION *****/
// Config: ext reset, no code protect, no watchdog, 4MHz int clock
__CONFIG(MCLREN & UNPROTECT & WDTDIS & INTOSC4 & INTIO);

// Pin assignments
#define TENS RB4 // "tens" (high nybble) enable
#define ONES RB1 // ones enable

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

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

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

```

```

// configure ports
TRISB = 0;           // configure PORTB and PORTC as all outputs
TRISC = 0;
CM1CON0 = 0;        // disable comparator 1 (RB0, RB1, RB2 usable)
CM2CON0 = 0;        // disable comparator 2 (RC0, RC1, RC4 usable)
VRCON = 0;          // disable CVref (RC2 usable)
// 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)
// 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
    GODONE = 1;           // start conversion
    while (GODONE)       // 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 on segment bus
    TENS = 1;           // enable tens digit to display it
    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 on segment bus
    ONES = 1;           // enable ones digit to display it
    while (TMR0_2)      // wait for TMR0<2> to go low
        ;
}
}

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

/***** Display digit on 7-segment display *****/
void set7seg(char digit)
{
    // Lookup pattern table for 7-segment display on ports B and C
    const char pat7seg[16] = {
        // RC5:0, RB5 = 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
};

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

// Extract pattern bits and write to segment bus pins
RB5 = pat7seg[digit] & 0b0000001;
PORTC = pat7seg[digit] >> 1;
}

```

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 comparators (RB0-2, RC0,1,4 usable)
setup_vref(FALSE); // disable CVref (RC2 usable)

```

The ADC also has to be configured, the same as before:

```

setup_adc_ports(AN0_AN2); // configure AN0 and AN2 as analog (RB1 usable)
setup_adc(ADC_CLOCK_INTERNAL); // select INTOSC/4 clock and turn ADC on
set_adc_channel(0); // ADC channel = AN0

```

As in the HI-TECH version, much of the code can be adapted from that presented in [lesson 4](#).

For example, setting up the timer:

```

setup_timer_0(RTCC_INTERNAL|RTCC_DIV_256); // timer mode, prescale = 256
// -> TMR0<2> cycles every 2.048ms

```

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 4](#):

```

// 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 on segment bus
output_high(TENS); // enable tens digit to display it
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 on segment bus
output_high(ONES);      // enable ones digit to display it
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 on segment bus
output_high(TENS);      // enable tens digit to display it
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
                        // on segment bus
output_high(ONES);      // enable ones digit to display it
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.

The ‘`set7seg()`’ function is also much the same as that presented in [lesson 4](#), extracting the pattern bits from the lookup array (now extended to 16 entries) and writing to the “segment bus” output pins:

```

// Extract pattern bits and write to segment bus pins
output_bit(PIN_B5, pat7seg[digit] & 0b0000001); // RB5
output_c(pat7seg[digit] >> 1); // PORTC

```

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 6, example 2
*
* Displays ADC output in hexadecimal on 7-segment LED displays
*
* Continuously samples analog input,
* displaying result as 2 x hex digits on multiplexed 7-seg displays
*
*****
*
* Pin assignments:
* AN0 = voltage to be measured (e.g. pot or LDR)
*

```

```

*      RB5, RC0-5  = 7-segment display bus (active high)      *
*      RB4        = "tens" (high nybble) enable (active high) *
*      RB1        = ones 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    PIN_B4           // "tens" (high nybble) enable
#define ONES    PIN_B1           // ones enable

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

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

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

    // Initialisation

    // configure ports
    setup_comparator(NC_NC_NC_NC); // disable comps (RB0-2, RC0,1,4 usable)
    setup_vref(FALSE);             // disable CVref (RC2 usable)

    // configure ADC:
    setup_adc_ports(AN0_AN2);      // config AN0 and AN2 as analog (RB1 usable)
    setup_adc(ADC_CLOCK_INTERNAL); // select INTOSC/4 clock and turn ADC on
    set_adc_channel(0);            // ADC channel = AN0

    // configure timer:
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_256); // timer mode, prescale = 256
                                                //     -> TMR0<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 on segment bus
        output_high(TENS);     // enable tens digit to display it
        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 on segment bus
    output_high(ONES);      // enable ones digit to display it
    while (TMR0_2)         // wait for TMR0<2> to go low
        ;
    }
}

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

/***** Display digit on 7-segment display *****/
void set7seg(char digit)
{
    // Lookup pattern table for 7-segment display on ports B and C
    const char pat7seg[16] = {
        // RC5:0,RB5 = 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
    };

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

    // Extract pattern bits and write to segment bus pins
    output_bit(PIN_B5, pat7seg[digit] & 0b0000001); // RB5
    output_c(pat7seg[digit] >> 1); // PORTC
}

```

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	92	83	1
HI-TECH PICC-Lite	46	105	7
HI-TECH C (Lite)	46	155	7
CCS PCB	45	119	9

HI-TECH C or PICC-Lite

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:

```
ADCON0 = 0b00111101;
//00-----          no analog inputs (ANS = 00) -> RB0-2 usable
//--11-----        clock = INTOSC/4 (ADCS = 11)
//----11--          select 0.6 V reference (CHS = 11)
//-----1          turn ADC on (ADON = 1)
```

The code to sample the ADC and output the result on the 7-segment displays is the same as above, but we 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 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 V_{DD} 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 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 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, with the PICC-Lite version blowing out to 297 words of program memory²! Compare this with the previous example, which is almost identical – lacking only this ADC comparison routine – but required only 105 words of program memory. You wouldn’t expect that adding such a simple routine would almost 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.

² using PICC-Lite v9.60PL2. Earlier versions of PICC-Lite contained a bug which prevented this code from working.

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 0.6V > threshold
{
    WARN = 1; // light warning LED
}
```

This simple change prevents the compiler from including floating-point code, reducing the size of the PICC-Lite generated code from 297 to only 109 words of program memory!

Complete program

Although the program is almost identical to the previous example, it is worth looking at the complete source, so that you can see where the ADC comparison code fits into the sample and display loop:

```

/*****
*
* Description: Lesson 6, example 3b
*
* Demonstrates use of 0.6 V ref with ADC to test supply voltage
*
* Continuously samples 0.6 V internal reference,
* displaying result as 2 x hex digits on multiplexed 7-seg displays
* If measurement > threshold, turn on warning LED
* (using integer comparison)
*
*****/
*
* Pin assignments:
* AN0 = voltage to be measured (e.g. pot or LDR)
* RB5, RC0-5 = 7-segment display bus (active high)
* RB4 = "tens" (high nybble) enable (active high)
* RB1 = ones enable
* RB2 = warning LED
*
*****/

#include <htc.h>

/***** CONFIGURATION *****/
// Config: ext reset, no code protect, no watchdog, 4 MHz int clock
__CONFIG(MCLREN & UNPROTECT & WDTDIS & INTOSC4 & INTIO);

// Pin assignments
#define TENS RB4 // "tens" (high nybble) enable
#define ONES RB1 // ones enable
#define WARN RB2 // warning LED

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

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

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

```

```

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

    // configure ports
    TRISB = 0;           // configure PORTB and PORTC as all outputs
    TRISC = 0;
    CM1CON0 = 0;        // disable comparator 1 (RB0, RB1, RB2 usable)
    CM2CON0 = 0;        // disable comparator 2 (RC0, RC1, RC4 usable)
    VRCON = 0;          // disable CVref (RC2 usable)
    // configure ADC
    ADCON0 = 0b00111101;
        //00-----    no analog inputs (ANS = 00) -> RB0-2 usable
        //--11-----    clock = INTOSC/4 (ADCS = 11)
        /----11--     select 0.6 V reference (CHS = 11)
        /-----1     turn ADC on (ADON = 1)
    // 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
        GODONE = 1;           // start conversion
        while (GODONE)       // 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 on segment bus
        TENS = 1;             // enable tens digit to display it
        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 on segment bus
        ONES = 1;            // enable ones digit to display it
        while (TMR0_2)       // wait for TMR0<2> to go low
            ;
    }
}

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

```

```

/***** Display digit on 7-segment display *****/
void set7seg(char digit)
{
    // Lookup pattern table for 7-segment display on ports B and C
    const char pat7seg[16] = {
        // RC5:0, RB5 = 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
    };

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

    // Extract pattern bits and write to segment bus pins
    RB5 = pat7seg[digit] & 0b0000001;
    PORTC = pat7seg[digit] >> 1;
}

```

CCS PCB

The initialisation code is much the same as in the previous example, except that we must disable all the analog inputs (so that RB1 and RB2 can be used as digital outputs), with:

```
setup_adc_ports(NO_ANALOGS); // disable all analog inputs (RB0-2 usable)
```

and the 0.6 V reference must be selected as the ADC input channel, instead of AN0:

```
set_adc_channel(3); // ADC channel = 0.6 V reference
```

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

The minimum allowable VDD can be defined:

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

and the ADC result tested, in a similar way to how it was initially written using PICC-Lite, 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 HI-TECH C 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 – 242 words of program memory, compared with only 119 words for the previous hexadecimal output example.

In the same way as was done with HI-TECH C, 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 129 words of program memory.

Complete program

Here is the complete CCS version of the “VDD measure” program, showing how the ADC comparison code fits into the sample and display loop:

```

/*****
*
*   Description:      Lesson 6, example 3b
*
*   Demonstrates use of 0.6 V ref with ADC to test supply voltage
*
*   Continuously samples 0.6 V internal reference,
*   displaying result as 2 x hex digits on multiplexed 7-seg displays
*   If measurement > threshold, turn on warning LED
*   (using integer comparison)
*
*****/
*
*   Pin assignments:
*
*   AN0              = voltage to be measured (e.g. pot or LDR)
*   RB5, RC0-5      = 7-segment display bus (active high)
*   RB4              = "tens" (high nybble) enable (active high)
*   RB1              = ones enable
*   RB2              = warning LED
*
*****/

#include <16F506.h>

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

// Pin assignments
#define TENS    PIN_B4           // "tens" (high nybble) enable
#define ONES    PIN_B1           // ones enable
#define WARN    PIN_B2           // warning LED

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

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

```

```

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

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

    // Initialisation

    // configure ports
    setup_comparator(NC_NC_NC_NC);    // disable comps (RB0-2, RC0,1,4 usable)
    setup_vref(FALSE);                // disable CVref (RC2 usable)

    // configure ADC:
    setup_adc_ports(NO_ANALOGS);      // disable all analog inputs (RB0-2 usable)
    setup_adc(ADC_CLOCK_INTERNAL);    // select INTOSC/4 clock and turn ADC on
    set_adc_channel(3);               // ADC channel = 0.6 V reference

    // configure timer:
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_256); // timer mode, prescale = 256
                                                // -> TMR0<2> cycles
                                                // every 2.048 ms

    // Main loop
    while (TRUE)
    {
        // sample 0.6V 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 on segment bus
        output_high(TENS);                 // enable tens digit to display it
        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 on segment bus
        output_high(ONES);                 // enable ones digit to display it
        while (TMR0_2)                    // wait for TMR0<2> to go low
            ;
    }
}

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

/***** Display digit on 7-segment display *****/
void set7seg(char digit)
{

```

```

// Lookup pattern table for 7-segment display on ports B and C
const char pat7seg[16] = {
    // RC5:0,RB5 = 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
};

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

// Extract pattern bits and write to segment bus pins
output_bit(PIN_B5, pat7seg[digit] & 0b0000001); // RB5
output_c(pat7seg[digit] >> 1); // PORTC
}

```

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	99	87	1
HI-TECH PICC-Lite	float	50	297	15
HI-TECH PICC-Lite	integer	50	109	7
HI-TECH C (Lite)	float	50	552	34
HI-TECH C (Lite)	integer	50	162	7
CCS PCB	float	49	242	15
CCS PCB	integer	49	129	9

Once again, the C source code is around half the length of the assembler source, and the PICC-Lite compiler continues to generate most efficient code – although still 25% larger than the assembler 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.

We then need to extract each digit of the scaled result for display. As we saw in [lesson 4](#), 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
    GODONE = 1;           // start conversion
    while (GODONE)       // wait until done
        ;

    // scale result to 0-99
    adc_dec = ADRES * 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 on segment bus
    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(adc_dec%10); // output ones digit of result on segment bus
    ONES_EN = 1;        // enable ones digit display
    while (TMR0_2)     // wait for TMR0<2> to go low
        ;
}
}
```

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

```
/***** Display digit on 7-segment display *****/
void set7seg(char digit)
{
    // Lookup pattern table for 7-segment display on ports B and C
    const char pat7seg[10] = {
        // RC5:0, RB5 = ABCDEFG
        0b1111110, // 0
        0b0110000, // 1
        0b1101101, // 2
        0b1111001, // 3
        0b0110011, // 4
        0b1011011, // 5
        0b1011111, // 6
        0b1110000, // 7
        0b1111111, // 8
        0b1111011 // 9
    };

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

    // Extract pattern bits and write to segment bus pins
    RB5 = pat7seg[digit] & 0b0000001;
    PORTC = pat7seg[digit] >> 1;
}
}
```

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 as:

```
unsigned int    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 type promotion, in the same way that the HI-TECH 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.

In CCS PCB, the 'int' type defines an 8-bit quantity³, the same as a 'char'. You might expect that this problem could be overcome by defining 'adc_dec' as a 16-bit '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 'long' type:

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

This generates the correct result.

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

As in the HI-TECH C 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 and scale input
    adc_dec = (long)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 on segment bus
    output_high(TENS_EN);    // enable tens digit display
    while (TMR0_2)           // wait for TMR0<2> to go low
    ;
}
```

³ unlike HI-TECH C, where an 'int' is 16-bit

```

    // 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 on segment bus
    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 table in the digit display function:

```

/***** Display digit on 7-segment display *****/
void set7seg(char digit)
{
    // Lookup pattern table for 7-segment display on ports B and C
    const char pat7seg[10] = {
        // RC5:0,RB5 = ABCDEFG
        0b1111110, // 0
        0b0110000, // 1
        0b1101101, // 2
        0b1111001, // 3
        0b0110011, // 4
        0b1011011, // 5
        0b1011111, // 6
        0b1110000, // 7
        0b1111111, // 8
        0b1111011 // 9
    };

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

    // Extract pattern bits and write to segment bus pins
    output_bit(PIN_B5, pat7seg[digit] & 0b0000001); // RB5
    output_c(pat7seg[digit] >> 1); // PORTC
}

```

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	111	100	7
HI-TECH PICC-Lite	42	238	14
HI-TECH C (Lite)	42	529	20
CCS PCB	39	175	15

In this example, where integer arithmetic is involved, the pros and cons of assembler versus C become very apparent. The assembler source is nearly three times 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 – less than half the size of the PICC-Lite version, and only 57% the size of the CCS version. It is also clear that the CCS compiler generates significantly smaller code than the HI-TECH PICC-Lite compiler, in this case.

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 lesson 10](#), 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 it is 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 $16 \times 4 \text{ ms} = 64 \text{ ms}$. This is more than enough to smooth out a 50 Hz flicker, since a 50 Hz signal has a period of only 20 ms.

HI-TECH C

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

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

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

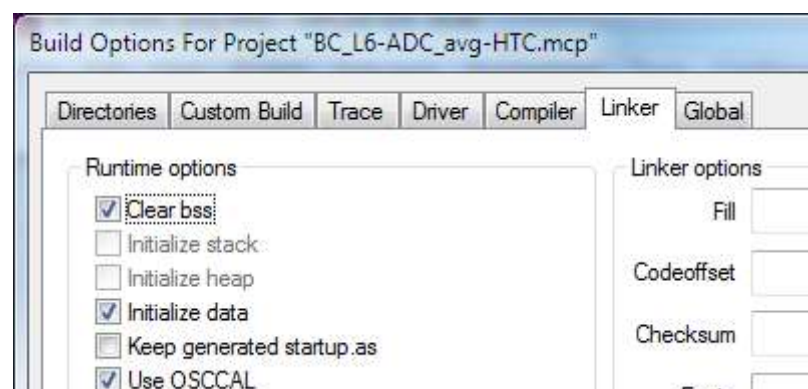
Defining the constant, 'NSAMPLES', toward the start of the program, makes it easier to change the number of samples from 16 later, if desired.

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, the HI-TECH C compiler adds runtime code which, among other things, clears all uninitialized global and 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.



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

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

Note that the running total 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 HI-TECH C version of the “ADC demo with averaged decimal output” program, showing where these code fragments fit in:

```
/*
 * Description: Lesson 6, 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:
 * AN0 = voltage to be measured (e.g. pot or LDR)
 * RB5, RC0-5 = 7-segment display bus (active high)
 * RB4 = tens enable (active high)
 * RB1 = ones enable
 */
#include <htc.h>

/* ***** CONFIGURATION ***** */
// Config: ext reset, no code protect, no watchdog, 4MHz int clock
__CONFIG(MCLRREN & UNPROTECT & WDTDIS & INTOSC4 & INTIO);
```

```

// Pin assignments
#define TENS_EN      RB4           // tens enable
#define ONES_EN     RB1           // ones enable

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

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

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

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

/***** MAIN PROGRAM *****/
void main()
{
    unsigned int    sum = 0;       // running total of ADC samples
    unsigned int    adc_dec;      // scaled average (0-99)
    unsigned char   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, RB2 usable)
    CM2CON0 = 0;                 // disable comparator 2 (RC0, RC1, RC4 usable)
    VRCON = 0;                   // disable CVref (RC2 usable)
    // 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)
    // 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
            GODONE = 1;           // start conversion
            while (GODONE)       // 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(adc_dec/10); // output tens digit of result on segment bus
    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(adc_dec%10); // output ones digit of result on segment bus
    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(char digit)
{
    // Lookup pattern table for 7-segment display on ports B and C
    const char pat7seg[10] = {
        // RC5:0,RB5 = ABCDEFG
        0b1111110, // 0
        0b0110000, // 1
        0b1101101, // 2
        0b1111001, // 3
        0b0110011, // 4
        0b1011011, // 5
        0b1011111, // 6
        0b1110000, // 7
        0b1111111, // 8
        0b1111011 // 9
    };

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

    // Extract pattern bits and write to segment bus pins
    RB5 = pat7seg[digit] & 0b0000001;
    PORTC = pat7seg[digit] >> 1;
}

```

HI-TECH PICC-Lite

If you try to use PICC-Lite to compile the source code from the HI-TECH C version, the compiler will complain that it can't allocate enough data memory, with an error message such as:

```
can't find 0x18 words for psect "rbss_0" in segment "BANK0"
```

This happens because PICC-Lite places all variables into bank 0, unless told otherwise. Unlike HI-TECH C, it will not automatically use the other three banks of data memory.

If there are too many variables and/or arrays to all fit into bank 0, one or more of the variables or arrays will need to be explicitly placed into another bank.

In this case, because the sample array is 16 bytes long, it will fill an entire bank, and therefore has to be located in a bank on its own.

To place a variable or array into bank 1, 2 or 3, the `bank1`, `bank2` and `bank3` type qualifiers are used. For example, to place the sample array into bank 1, it is declared as:

```
bank1 unsigned char smp_buf[NSAMPLES]; // array of samples for moving average
```

Note that the bank qualifiers can only be used with global (or external) or static variables. This means that the declaration must be placed outside the program's functions, or, if declared within a function (including `main()`), it must be declared as `static`.

For example:

```
bank1 char abc[12]; // global scope - ok

main()
{
    static bank2 char xyz[10]; // local scope - static, so ok
}
```

You will find that, with the sample array explicitly placed in bank 1, the code will now compile.

Unfortunately, there is still a problem – due to a bug in PICC-Lite version 9.60 PL2 (also present in version 9.60 PL1, as supplied with MPLAB 8.10): the statement `"smp_buf[s] = ADRES;"` doesn't work!

That is, **the array is not written to correctly, when it is placed in bank 1, as in the declaration above**. If the size of the array is reduced until it can fit into bank 0 (about 8 elements), the code works ok. The problem relates to writing to arrays located in a bank other than bank 0, on baseline PICs.

Unless HI-TECH Software releases a version of PICC-Lite which corrects this problem⁴, we need to work around it by inserting an assembler routine into the code, to perform the array write operation.

This is simplified if we first declare a pointer, to reference the array element to be updated:

```
char *smp_ptr; // pointer to sample to be updated
```

This declaration must be global (outside any functions), or made static, so that the address of the pointer variable is fixed. This allows it to be referenced easily from within an inline assembler routine.

We can then point it at the sample to be updated:

```
smp_ptr = smp_buf + s; // get pointer to smp_buf[s]
```

⁴ unlikely, since it is no longer under development...

This address calculation could be done in the assembler routine, avoiding the need to declare the ‘smp_ptr’ pointer variable, but since we are programming in C, we may as well use C where possible...

Finally, the inline assembler routine is added by placing it between ‘#asm’ and ‘#endasm’ directives, as follows:

```
// (implement write to smp_buf[s] in assembler, due to compiler bug)
#asm
    movf    _smp_ptr,w      ; copy pointer to FSR
    movwf  _FSR
    movf    _ADRES,w      ; copy ADRES to INDF
    movwf  0              ; (indirect access to smp_buf[s])
    clrf   _FSR          ; reset to bank 0
#endasm
```

Note that symbols defined in the C source, such as ‘smp_ptr’, and register names defined in the C header files, such as ‘FSR’ and ‘ADRES’, can be referenced in the assembler source, by pre-pending a ‘_’ to them. But since no ‘INDF’ symbol is defined in the PICC-Lite headers, we have to use the numeric address of the INDF register; hence the “movwf 0” line.

To understand what this assembler code is doing, including the use of the FSR and INDF registers to write to an array, see the discussion in [baseline lesson 10](#).

The final assembler instruction, “clrf _FSR”, is included to set the bank selection bits in FSR to point to bank 0. This is necessary because the PICC-Lite compiler cannot know what our assembler code has done, and will assume that bank 0 is still selected.

Here is the new main loop, showing where this assembler routine fits in:

```
// Main loop
for (;;)
{
    for (s = 0; s < NSAMPLES; s++)
    {
        // sample input
        GODONE = 1;           // start conversion
        while (GODONE)       // 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
        // (DOES NOT WORK!)

        // (implement write to smp_buf[s] in assembler, due to compiler bug)
        smp_ptr = smp_buf + s;    // get pointer to smp_buf[s]
        #asm
            movf    _smp_ptr,w      ; copy pointer to FSR
            movwf  _FSR
            movf    _ADRES,w      ; copy ADRES to INDF
            movwf  0              ; (indirect access to smp_buf[s])
            clrf   _FSR          ; reset to bank 0
        #endasm

        // 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 on segment bus
        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(adc_dec%10); // output ones digit of result on segment bus
        ONES_EN = 1;      // enable ones digit display
        while (TMR0_2)    // wait for TMR0<2> to go low
            ;
    }
}
}

```

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; there is no need to specify which bank to use (unlike PICC-Lite).

So we can declare the sample buffer array as:

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

Unlike HI-TECH C, 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:

```

int 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:

```

char adc_res; // result of ADC conversion
long sum = 0; // running total of ADC samples
int adc_dec; // scaled average (0-99)

```

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

As we did in the PICC-Lite 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 += (long)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 += (long)adc_res - read_bank(1,s); // add new value 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 6, 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:                                               *
*       AN0           = voltage to be measured (e.g. pot or LDR)   *
*       RB5, RC0-5    = 7-segment display bus (active high)       *
*       RB4           = tens enable (active high)                 *
*       RB1           = ones enable                               *
*                                                                     *
*****/

#include <16F506.h>

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

// Pin assignments
#define TENS_EN      PIN_B4           // tens enable
#define ONES_EN      PIN_B1           // ones enable

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

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

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

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

    // Initialisation

    // configure ports
    setup_comparator(NC_NC_NC_NC); // disable comps (RB0-2, RC0,1,4 usable)
    setup_vref(FALSE);            // disable CVref (RC2 usable)

    // configure ADC:
    setup_adc_ports(AN0_AN2);     // config AN0 and AN2 as analog (RB1 usable)
    setup_adc(ADC_CLOCK_INTERNAL); // select INTOSC/4 clock and turn ADC on
    set_adc_channel(0);           // ADC channel = AN0

    // configure timer:
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_256); // timer mode, prescale = 256
                                                // -> TMR0<2> period = 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 += (long)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 on segment bus
        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 on segment bus
        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(char digit)
{
    // Lookup pattern table for 7-segment display on ports B and C
    const char pat7seg[10] = {
        // RC5:0, RB5 = ABCDEFG
        0b1111110, // 0
        0b0110000, // 1
        0b1101101, // 2
        0b1111001, // 3
        0b0110011, // 4
        0b1011011, // 5
        0b1011111, // 6
        0b1110000, // 7
        0b1111111, // 8
        0b1111011 // 9
    };

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

```

```

// Extract pattern bits and write to segment bus pins
output_bit(PIN_B5, pat7seg[digit] & 0b0000001); // RB5
output_c(pat7seg[digit] >> 1); // PORTC
}

```

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	146	133	26
HI-TECH PICC-Lite	57	254	34
HI-TECH C (Lite)	49	521	38
CCS PCB	49	247	35

In this example, the differences between C and assembler are even more pronounced. The assembler source is three times as long as the HI-TECH C and CCS versions, while the assembled version is only around half the size of the code generated by the “optimising” C compilers.

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 assembler 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 HI-TECH 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 HI-TECH C compiler is able to effectively work directly with “large” (16 byte) arrays. This means that, at the time of writing (February 2010), if you wish to use a “free” compiler which generates optimised code, you need to use workarounds if you need to work with arrays on baseline PICs.

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 HI-TECH 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_4LEDs	ADC_hex_out	Vdd_measure	ADC_dec_out	ADC_avg
Microchip MPASM	20	92	99	111	146
HI-TECH PICC-Lite	12	46	50	42	57
HI-TECH C (Lite)	12	46	50	42	49
CCS PCB	10	45	49	39	49

But again, both C compilers generate code which is significantly larger than the corresponding hand-written assembler versions; the most complex programs being more than twice the size of the assembler version:

Program memory (words)

Assembler / Compiler	ADC_4LEDs	ADC_hex_out	Vdd_measure	ADC_dec_out	ADC_avg
Microchip MPASM	16	83	87	100	133
HI-TECH PICC-Lite	18	105	109	238	254
HI-TECH C (Lite)	40	155	162	529	521
CCS PCB	37	119	129	175	247

Data memory (bytes)

Assembler / Compiler	ADC_4LEDs	ADC_hex_out	Vdd_measure	ADC_dec_out	ADC_avg
Microchip MPASM	0	1	1	7	26
HI-TECH PICC-Lite	4	7	7	14	34
HI-TECH C (Lite)	4	7	7	20	38
CCS PCB	5	9	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 HI-TECH C compilers, on a device as small as the PIC16F506. The PICC-Lite and CCS PBC compilers each 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 as far as they will go. It seems that, to get the most from these baseline PICs, to reach their potential, we need to use assembler. Or you could pay for the full (optimising) version of HI-TECH C, 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 “midrange” PIC architecture.

These larger, more flexible microcontrollers are covered in the “[Midrange PIC Architecture and Assembly Language](#)” tutorial series, which introduces the midrange 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 Midrange PICs in C](#)” tutorial series, where we cover the same ground again, using C.

