Introduction to PIC Programming

Baseline Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 11: Integer Arithmetic and Arrays

In the last lesson, we saw how to read an analog input and display the "raw" result. But in most cases the raw values aren't directly usable; normally they will need to be processed in some way before being displayed or used in decision making. While advanced signal processing is beyond the capabilities of baseline and even midrange PICs, this lesson demonstrates that basic post-processing, such as integer scaling and simple filtering, can be readily accomplished with even the lowest-end PICs.

This lesson introduces some of the basic integer arithmetic operations. For more complete coverage of this topic, refer to Microchip's application notes *AN526: "PIC16C5X / PIC16CXXX Math Utility Routines"*, and *AN617: "Fixed Point Routines"*, available at www.microchip.com.

We'll also see how to use indirect addressing to implement arrays, illustrated by a simple moving average routine, used to filter noise from an analog signal.

In summary, this lesson covers:

- Multi-byte (including 16-bit and 32-bit) addition and subtraction
- Two's complement representation of negative numbers
- 8-bit unsigned multiplication
- Using indirect addressing to work with arrays
- Calculating a moving average

Integer Arithmetic

At first sight, the baseline PICs seem to have very limited arithmetic capabilities: just a single 8-bit addition instruction (`addwf`) and a single 8-bit subtraction instruction (`subwf`).

However, addition and subtraction can be extended to arbitrarily large numbers by using the carry flag (C, in the STATUS register), which indicates when a result cannot be represented in a single 8-bit byte.

The `addwf` instruction sets the carry flag if the result *overflows* a single byte, i.e. is greater than 255.

And as explained in lesson 5, the carry flag acts as a "not borrow" in a subtraction: the `subwf` instruction clears C if a borrow occurs, i.e. the result is negative.

The carry flag allows us to cascade addition or subtraction operations when working with long numbers.

Multi-byte variables

To store values larger than 8-bits, you need to allocate multiple bytes of memory to each, for example:

```
        UDATA
a       res 2                   ; 16-bit variables "a" and "b"
b       res 2
```

You must then decide how to order the bytes within the variable – whether to place the least significant byte at the lowest address in the variable (known as *little-endian* ordering) or the highest (*big-endian*).

For example, to store the number 0x482C in variable "a", the bytes 0x48 and 0x2C would be placed in memory as shown:

	a	a+1
Little-endian	0x2C	0x48
Big-endian	0x48	0x2C

Big-endian ordering has the advantage of making values easy to read in a hex dump, where increasing addresses are presented left to right. On the other hand, little-endian ordering makes a certain sense, because increasing addresses store increasingly significant bytes.

Which ordering you chose is entirely up to you; both are valid. This tutorial uses little-endian ordering, but the important thing is to be consistent.

16-bit addition

The following code adds the contents of the two 16-bit variables, "a" and "b", so that b = b + a, assuming little-endian byte ordering:

```
movf    a,w         ; add LSB
addwf   b,f
btfsc   STATUS,C    ; increment MSB if carry
incf    b+1,f
movf    a+1,w       ; add MSB
addwf   b+1,f
```

After adding the least significant bytes (LSB's), the carry flag is checked, and, if the LSB addition overflowed, the most significant byte (MSB) of the result is incremented, before the MSB's are added.

Multi-byte (including 32-bit) addition

It may appear that this approach would be easily extended to longer numbers by testing the carry after the final '`addwf`', and incrementing the next MSB of the result if carry was set. But there's a problem. What if the LSB addition overflows, while (`b+1`) contains $FF? The '`incf b+1,f`' instruction will increment (`b+1`) to $00, which should result in a "carry", but it doesn't, since '`incf`' does not affect the carry flag.

By re-ordering the instructions, it is possible to use the '`incfsz`' instruction to neatly avoid this problem:

```
movf    a,w         ; add LSB
addwf   b,f
movf    a+1,w       ; get MSB(a)
btfsc   STATUS,C    ; if LSB addition overflowed,
incfsz  a+1,w       ;   increment copy of MSB(a)
addwf   b+1,f       ; add to MSB(b), unless MSB(a) is zero
```

On completion, the carry flag will now be set correctly, allowing longer numbers to be added by repeating the final four instructions. For example, for a 32-bit add:

```
movf    a,w         ; add byte 0 (LSB)
addwf   b,f
movf    a+1,w       ; add byte 1
btfsc   STATUS,C
incfsz  a+1,w
addwf   b+1,f
movf    a+2,w       ; add byte 2
btfsc   STATUS,C
incfsz  a+2,w
addwf   b+2,f
movf    a+3,w       ; add byte 3 (MSB)
btfsc   STATUS,C
incfsz  a+3,w
addwf   b+3,f
```

Multi-byte (including 16-bit and 32-bit) subtraction

Long integer subtraction can be done using a very similar approach.

For example, to subtract the contents of the two 16-bit variables, "a" and "b", so that b = b – a, assuming little-endian byte ordering:

```
movf    a,w         ; subtract LSB
subwf   b,f
movf    a+1,w       ; get MSB(a)
btfss   STATUS,C    ; if borrow from LSB subtraction,
incfsz  a+1,w       ;   increment copy of MSB(a)
subwf   b+1,f       ; subtract MSB(b), unless MSB(a) is zero
```

This approach is readily extended to longer numbers, by repeating the final four instructions.

For a 32-bit subtraction, we have:

```
movf    a,w         ; subtract byte 0 (LSB)
subwf   b,f
movf    a+1,w       ; subtract byte 1
btfss   STATUS,C
incfsz  a+1,w
subwf   b+1,f
movf    a+2,w       ; subtract byte 2
btfss   STATUS,C
incfsz  a+2,w
subwf   b+2,f
movf    a+3,w       ; subtract byte 3 (MSB)
btfss   STATUS,C
incfsz  a+3,w
subwf   b+3,f
```

Two's complement

Microchip's application note *AN526* takes a different approach to subtraction.

Instead of subtracting a number, it is *negated* (made negative), and then added. That is, b – a = b + (–a).

Negating a binary number is also referred to as taking its *two's complement*, since the operation is equivalent to subtracting it from a power of two.

The two's complement of an n-bit number, "a", is given by the formula $2^n - a$.

For example, the 8-bit two's complement of 10 is $2^8 - 10 = 256 - 10 = 246$.

The two's complement of a number acts the same as a negative number would, in fixed-length binary addition and subtraction.

For example, 10 + (–10) = 0 is equivalent to 10 + 246 = 256, since in an 8-bit addition, the result (256) overflows, giving an 8-bit result of 0.

Similarly, 10 + (–9) = 1 is equivalent to 10 + 247 = 257, which overflows, giving an 8-bit result of 1.

And 10 + (–11) = –1 is equivalent to 10 + 245 = 255, which is the two's complement of 1.

Thus, two's complement is normally used to represent negative numbers in binary integer arithmetic, because addition and subtraction continue to work the same way. The only thing that needs to change is how the numbers being added or subtracted, and the results, are interpreted.

For unsigned quantities, the range of values for an n-bit number is from 0 to 2^n-1.

For signed quantities, the range is from -2^{n-1} to $2^{n-1}-1$.

For example, 8-bit signed numbers range from –128 to 127.

The usual method used to calculate the two's complement of a number is to take the ones' complement (flip all the bits) and then add one.

This method is used in the 16-bit negate routine provided in *AN526*:

```
neg_A   comf    a,f         ; negate a ( -a -> a )
        incf    a,f
        btfsc   STATUS,Z
        decf    a+1,f
        comf    a+1,f
```

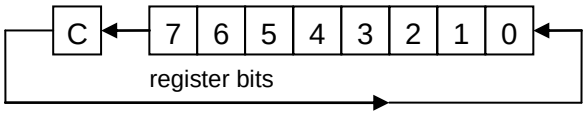
There is a new instruction here: '`comf f,d`' – "**com**plement register **f**ile", which calculates the ones' complement of register '`f`', placing the result back into the register if the destination is '`,f`', or in W if the destination is '`,w`'.

One reason you may wish to negate a number is to display it, if it is negative.

To test whether a two's complement signed number is negative, check its most significant bit, which acts as a sign bit: '1' indicates a negative number, '0' indicates non-negative (positive or zero).

Unsigned multiplication

It may seem that baseline PICs have no multiplication or division instructions, but that's not quite true: the "rotate left" instruction (`rlf`) can be used to shift the contents of a register one bit to the left, which has the effect of multiplying it by two:



Since the `rlf` instruction rotates bit 7 into the carry bit, and carry into bit 0, these instructions can be cascaded, allowing arbitrarily long numbers to be shifted left, and hence multiplied by two.

For example, to multiply the contents of 16-bit variable "`a`" by two, assuming little-endian byte ordering:

```
; left-shift 'a' (multiply by 2)
bcf     STATUS,C        ; clear carry
rlf     a,f             ; left shift LSB
rlf     a+1,f           ; then MSB (LSB<7> -> MSB<0> via carry)
```

[Although we won't consider division here (see *AN526* for details), a similar sequence of "rotate right" instructions (`rrf`) can be used to shift an arbitrarily long number to the right, dividing it by two.]

You can see, then, that it is quite straightforward to multiply an arbitrarily long number by two. Indeed, by repeating the shift operation, multiplying or dividing by any power of two is easy to implement.

But that doesn't help us if we want to multiply by anything other than a power of two – or does it? Remember that every integer is composed of powers of two; that is how binary notation works

For example, the binary representation of 100 is 01100100 – the '1's in the binary number corresponding to powers of two:

$$100 = 64 + 32 + 4 = 2^6 + 2^5 + 2^2.$$

Thus, $100 \times N = (2^6 + 2^5 + 2^2) \times N = 2^6 \times N + 2^5 \times N + 2^2 \times N$

In this way, multiplication by any integer can be broken down into a series of multiplications by powers of two (repeated left shifts) and additions.

The general multiplication algorithm, then, consists of a series of shifts and additions, an addition being performed for each '1' bit in the multiplier, indicating a power of two that has to be added.

See *AN526* for a flowchart illustrating the process.

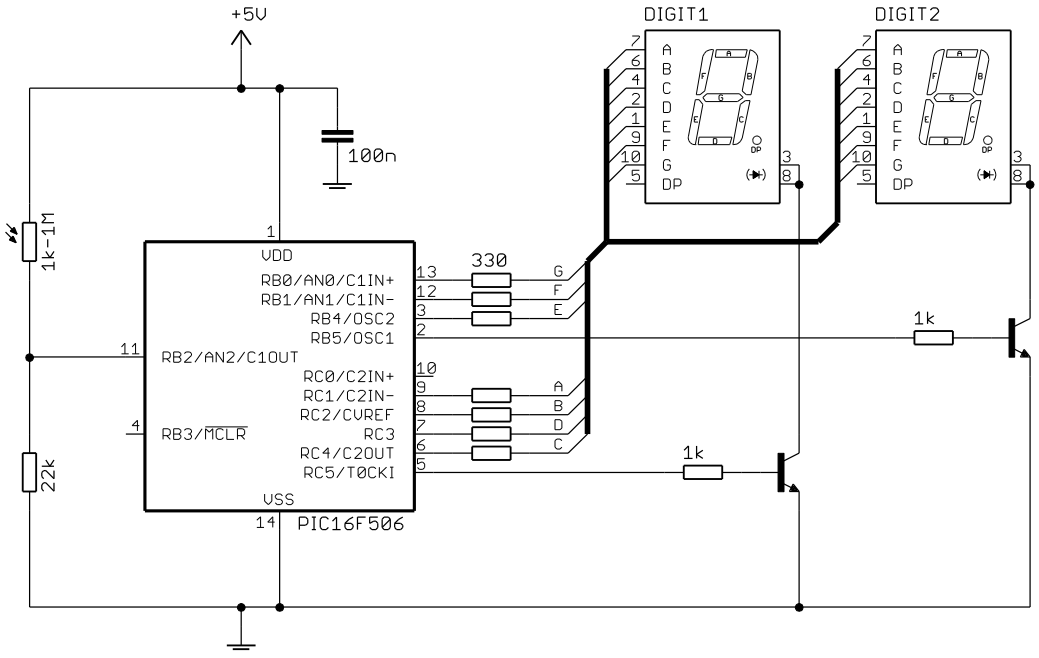
Here is the 8-bit unsigned multiplication routine from *AN526*:

```
; Variables:
;   mulcnd - 8 bit multiplicand
;   mulplr - 8 bit multiplier
;   H_byte - High byte of the 16 bit result
;   L_byte - Low byte of the 16 bit result
;   count  - loop counter
;
; ***************************** Begin Multiplier Routine
mpy_S   clrf    H_byte          ; start with result = 0
        clrf    L_byte
        movlw   8               ;   count = 8
        movwf   count
        movf    mulcnd,w        ;   multiplicand in W
        bcf     STATUS,C        ;   and carry clear
loop    rrf     mulplr,f        ; right shift multiplier
        btfsc   STATUS,C        ; if low-order bit of multiplier was set
        addwf   H_byte,f        ;   add multiplicand to MSB of result
        rrf     H_byte,f        ; right shift result
        rrf     L_byte,f
        decfsz  count,f         ; repeat for all 8 bits
        goto    loop
```

It may seem strange that `rrf` is being used here, instead of `rlf`. This is because the multiplicand is being added to the MSB of the result, before being right shifted. The multiplier is processed starting from bit 0. Suppose that bit 0 of the multiplier is a '1'. The multiplicand will be added to the MSB of the result in the first loop iteration. After all eight iterations, it will have been shifted down (right) into the LSB. Subsequent multiplicand additions, corresponding to higher multiplier bits, won't be shifted down as far, so their contribution to the final result is higher. You may need to work an example on paper to see how it works…

Example 1: Light meter with decimal output

Lesson 10 included a simple light meter based on a light-dependent resistor, which displayed the 8-bit ADC output as a two-digit hexadecimal number, using 7-segement LED displays, as shown below:



That's adequate for demonstrating the operation of the ADC module, but it's not a very good light meter. Most people would find it easier to read the display if it was in decimal, not hex, with a scale from 00 – 99 instead of 00h – FFh.

To scale the ADC output from 0 – 255 to 0 – 99, it has to be multiplied by 99/255.

Multiplying by 99 isn't difficult, but dividing by 255 is.

The task is made much easier by using an approximation: instead of multiplying by 99/255, multiply by 100/256. That's a difference of 0.6%; not really significant, given that the ADC is only accurate to ±2 lsb (2/256, or 0.8%) in any case.

Dividing by 256 is trivial – to divide a 16-bit number by 256, the result is already there – it's simply the most significant byte, with the LSB being the remainder. That gives a result which is always rounded down; if you want to round "correctly", increment the result if the LSB is greater than 127 (LSB<7> = 1) . For example:

```
; Variables:
;   a = 16-bit value (little endian)
;   b = a / 256 (rounded)
        movf    a+1,w           ; result = MSB
        btfsc   a,7             ; if LSB<7> = 1
        incf    a+1,w           ;   result = MSB+1
        movwf   b               ; write result
```

Note that, if MSB = 255 and LSB > 127, the result will "round" to zero; probably not what you want.

And in this example, since we're scaling the output to 0 – 99, we wouldn't want to round the result up to 100, since it couldn't be displayed in two digits. We could check for that case and handle it, but it's easiest to simply ignore rounding, and that's valid, because the numbers displays on the light meter don't correspond to any "real" units which would need to be accurately measured. In other words, the display is in arbitrary units; regardless of the rounding, it will display higher numbers in brighter light, and that's all we're trying to do.

To multiply the raw ADC result by 100, we can adapt the routine from *AN526*:

```
        ; scale to 0-99: adc_dec = adc_out * 100
        ;   -> MSB of adc_dec = adc_out * 100 / 256
        clrf    adc_dec         ; start with adc_dec = 0
        clrf    adc_dec+1
        movlw   .8              ;   count = 8
        movwf   mpy_cnt
        movlw   .100            ;   multiplicand (100) in W
        bcf     STATUS,C        ;   and carry clear
l_mpy   rrf     adc_out,f       ; right shift multiplier
        btfsc   STATUS,C        ; if low-order bit of multiplier was set
        addwf   adc_dec+1,f     ;   add multiplicand (100) to MSB of result
        rrf     adc_dec+1,f     ; right shift result
        rrf     adc_dec,f
        decfsz  mpy_cnt,f       ; repeat for all 8 bits
        goto    l_mpy
```

The 16-bit variable '`adc_dec`' now holds the raw ADC result multiplied by 100.

This means that most significant byte of '`adc_dec`' (the value stored in the memory location '`adc_dec+1`') is equal to the raw ADC result × 100/256.

After scaling the ADC result, we need to extract the "tens" and "ones" digits from it.

That can be done by repeated subtraction; the "tens" digit is determined by continually subtracting 10 from the original value, counting the subtractions until the remainder is less than 10. The "ones" digit is then simply the remainder:

```
        ; extract digits of result
        movf    adc_dec+1,w     ; start with scaled result
        movwf   ones            ;   in ones digit
        clrf    tens            ; and tens clear
l_bcd   movlw   .10             ; subtract 10 from ones
        subwf   ones,w
        btfss   STATUS,C        ; (finish if < 10)
        goto    end_bcd
        movwf   ones
        incf    tens,f          ; increment tens
        goto    l_bcd           ; repeat until ones < 10
end_bcd
```

The '`ones`' and '`tens`' variables now hold the two digits to be displayed.

Complete program

The rest of the program is essentially the same as the hexadecimal-output example from lesson 10. Here is how the scaling and digit extraction routines fit in:

```
;*****************************************************************************
;                                                                            *
;   Description:    Lesson 11, example 1                                     *
;                                                                            *
;   Displays ADC output in decimal on 2-digit 7-segment LED display          *
;                                                                            *
;   Continuously samples analog input, scales result to 0 - 99               *
;   and displays as 2 x dec 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                                  *
;                                                                            *
;*****************************************************************************

    list        p=16F506
    #include    <p16F506.inc>

    radix       dec


;***** CONFIGURATION
                ; ext reset, no code protect, no watchdog, 4 MHz int clock
    __CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _IOSCFS_OFF & _IntRC_OSC_RB4EN

; pin assignments
    #define TENS_EN     PORTC,5     ; tens digit enable
    #define ONES_EN     PORTB,5     ; ones digit enable
```

```
;***** VARIABLE DEFINITIONS
        UDATA
adc_out res 1                   ; raw ADC output
adc_dec res 2                   ; scaled ADC output (LE 16 bit, 0-99 in MSB)
mpy_cnt res 1                   ; multiplier count
                                ; digits to be displayed:
tens    res 1                   ;   tens
ones    res 1                   ;   ones

temp    res 1                   ; (temp storage used by set7seg)


;***** RC CALIBRATION
RCCAL   CODE    0x3FF           ; processor reset vector
        res 1                   ; holds internal RC cal value, as a movlw k

;***** RESET VECTOR ****************************************************
RESET   CODE    0x000           ; effective reset vector
        movwf   OSCCAL          ; apply internal RC factory calibration
        pagesel start
        goto    start           ; jump to main code

;***** SUBROUTINE VECTORS
set7seg                         ; display digit on 7-segment display
        pagesel set7seg_R
        goto    set7seg_R


;***** MAIN PROGRAM ****************************************************
MAIN    CODE

;***** Initialisation
start
        ; configure ports
        clrw                    ; configure PORTB and PORTC as all outputs
        tris    PORTB
        tris    PORTC
        clrf    CM1CON0         ; disable comparator 1 -> RB0, RB1 digital
        clrf    CM2CON0         ; disable comparator 2 -> RC0, RC1 digital
        clrf    VRCON           ; disable CVref -> RC2 usable

        ; configure ADC
        movlw   b'01111001'     ; configure ADC:
                ; 01------          AN2 (only) analog (ANS = 01)
                ; --11----          clock = INTOSC/4 (ADCS = 11)
                ; ----10--          select channel AN2 (CHS = 10)
                ; -------1          turn ADC on (ADON = 1)
        movwf   ADCON0          ;   -> AN2 ready for sampling

        ; configure timer
        movlw   b'11010111'     ; configure Timer0:
                ; --0-----          timer mode (T0CS = 0) -> RC5 usable
                ; ----0---          prescaler assigned to Timer0 (PSA = 0)
                ; -----111          prescale = 256 (PS = 111)
        option                  ;   -> increment every 256 us
                                ;      (TMR0<2> cycles every 2.048ms)

;***** Main loop
main_loop
        ; sample input
        bsf     ADCON0,GO       ; start conversion
```

```
w_adc   btfsc   ADCON0,NOT_DONE ; wait until conversion complete
        goto    w_adc
        movf    ADRES,w         ; save ADC result in adc_out
        banksel adc_out
        movwf   adc_out

        ; scale to 0-99: adc_dec = adc_out * 100
        ;   -> MSB of adc_dec = adc_out * 100 / 256
        clrf    adc_dec         ; start with adc_dec = 0
        clrf    adc_dec+1
        movlw   .8              ;   count = 8
        movwf   mpy_cnt
        movlw   .100            ;   multiplicand (100) in W
        bcf     STATUS,C        ;   and carry clear
l_mpy   rrf     adc_out,f       ; right shift multiplier
        btfsc   STATUS,C        ; if low-order bit of multiplier was set
        addwf   adc_dec+1,f     ;   add multiplicand (100) to MSB of result
        rrf     adc_dec+1,f     ; right shift result
        rrf     adc_dec,f
        decfsz  mpy_cnt,f       ; repeat for all 8 bits
        goto    l_mpy

        ; extract digits of result
        movf    adc_dec+1,w     ; start with scaled result
        movwf   ones            ;   in ones digit
        clrf    tens            ; and tens clear
l_bcd   movlw   .10             ; subtract 10 from ones
        subwf   ones,w
        btfss   STATUS,C        ; (finish if < 10)
        goto    end_bcd
        movwf   ones
        incf    tens,f          ; increment tens
        goto    l_bcd           ; repeat until ones < 10
end_bcd

        ; display tens digit for 2.048 ms
w10_hi  btfss   TMR0,2          ; wait for TMR0<2> to go high
        goto    w10_hi
        movf    tens,w          ; output tens digit
        pagesel set7seg
        call    set7seg
        pagesel $
        bsf     TENS_EN         ; enable tens display
w10_lo  btfsc   TMR0,2          ; wait for TMR<2> to go low
        goto    w10_lo

        ; display ones digit for 2.048 ms
w1_hi   btfss   TMR0,2          ; wait for TMR0<2> to go high
        goto    w1_hi
        banksel ones            ; output ones digit
        movf    ones,w
        pagesel set7seg
        call    set7seg
        pagesel $
        bsf     ONES_EN         ; enable ones display
w1_lo   btfsc   TMR0,2          ; wait for TMR<2> to go low
        goto    w1_lo

        ; repeat forever
        goto    main_loop
```

```
;***** LOOKUP TABLES ************************************************
TABLES  CODE    0x200           ; locate at beginning of a page

; pattern table for 7 segment display on port B
;   RB4 = E, RB1:0 = FG
get7sB  addwf   PCL,f
        retlw   b'010010'       ; 0
        retlw   b'000000'       ; 1
        retlw   b'010001'       ; 2
        retlw   b'000001'       ; 3
        retlw   b'000011'       ; 4
        retlw   b'000011'       ; 5
        retlw   b'010011'       ; 6
        retlw   b'000000'       ; 7
        retlw   b'010011'       ; 8
        retlw   b'000011'       ; 9

; pattern table for 7 segment display on port C
;   RC4:1 = CDBA
get7sC  addwf   PCL,f
        retlw   b'011110'       ; 0
        retlw   b'010100'       ; 1
        retlw   b'001110'       ; 2
        retlw   b'011110'       ; 3
        retlw   b'010100'       ; 4
        retlw   b'011010'       ; 5
        retlw   b'011010'       ; 6
        retlw   b'010110'       ; 7
        retlw   b'011110'       ; 8
        retlw   b'011110'       ; 9

; Display digit passed in W on 7-segment display
set7seg_R
        ; disable displays
        clrf    PORTB           ; clear all digit enable lines on PORTB
        clrf    PORTC           ;   and PORTC

        ; output digit pattern
        banksel temp
        movwf   temp            ; save digit
        call    get7sB          ; lookup pattern for port B
        movwf   PORTB           ;   then output it
        movf    temp,w          ; get digit
        call    get7sC          ;   then repeat for port C
        movwf   PORTC
        retlw   0


        END
```

Moving Averages, Indirect Addressing and Arrays

Moving averages

A problem with the light meter, as developed so far, is that the display can become unreadable in fluorescent light, because fluorescent lights flicker (too fast for the human eye to notice), and since the meter reacts very quickly (244 samples per second), the display changes too fast to follow.

One solution would be to reduce the sampling rate, to say one sample per second, so that the changes become slow enough for a human to see. But that's not a good solution; the display would still jitter significantly, since some samples would be taken when the illumination was high and others when it was low.

Instead of using a single raw sample, it is often better to smooth the results by implementing a *filter* based on a number of samples over time (a *time series*). Many filter algorithms exist, with various characteristics.

One that is particularly easy to implement is the *simple moving average*, also known as a *box filter*. This is simply the mean value of the last N samples. It is important to average enough samples to produce a smooth result, and to maintain a fast response time, a new average should be calculated every time a new sample is read. For example, you could keep the last ten samples, and then to calculate the simple moving average by adding all the sample values and then dividing by ten. Whenever a new sample is read, it is added to the list, the oldest sample is discarded, and the calculation is repeated. In fact, it is not necessary to repeat all the additions; it is only necessary to subtract the oldest value (the sample being discarded) and to add the new sample value.

Sometimes it makes more sense to give additional weight to more recent samples, so that the moving average more closely tracks the most recent input. A number of forms of *weighting* can be used, including arithmetic and exponential, which require more calculation. But a simple moving average is sufficient for our purpose here.

Indirect addressing and arrays

Instead of talking about a "list" of samples, we'd normally call it an *array*.

An array is a contiguous set of variables which can be accessed through a numeric index.

For example, to calculate an average in C, you might write something like:

```
int s[10];      /* array of samples */
int avg;        /* sample average */
int i;

avg = 0;
for (i = 0; i < 10; i++)    /* add all the samples */
    avg = avg + s[i];
avg = avg / 10;             /* divide by 10 to calculate average */
```

But how could we do that in PIC assembler?

You could define a series of variables: s0, s1, s2, … , s9, but there is then no way to add them in a loop, since each variable would have to be referred to by its own block of code. That would make for a long, and difficult to maintain program.

There is of course a way: the baseline PICs support *indirect addressing* (making array indexing possible), through the FSR and INDF registers.

The INDF (**ind**irect **f**ile) "register" acts as a window, through which the contents of any other register can be accessed.

The FSR (**f**ile **s**elect **r**egister) holds the address of the register which will be accessed through INDF.

For example, if FSR = 08h, INDF accesses the register at address 08h, which is CM1CON0 on the PIC16F506.

So, on the PIC16F506, if FSR = 08h, reading or writing INDF is the same as reading or writing CM1CON0.

Recall that the bank selection bits form the upper bits of the FSR register.

When you write a value into FSR, INDF will access the register at the address given by that value, irrespective of banking. That is, indirect addressing allows linear, un-banked access to the register file.

For example, if FSR = 54h, INDF will access the register at address 54h; this happens to be in bank 2, but that's not a consideration when using indirect addressing.

> *Note: When FSR is updated for indirect register access, the bank selection bits will be overwritten.*

The PIC12F510/16F506 data sheet includes the following code to clear registers 10h – 1Fh:

```
        movlw   0x10      ; initialize pointer to RAM
        movwf   FSR
next    clrf    INDF      ; indirectly clear register (pointed to by FSR)
        incf    FSR,f     ; inc pointer
        btfsc   FSR,4     ; all done?
        goto    next      ; NO, clear next
continue
                          ; YES, continue
```

The '`clrf INDF`' instruction clears the register pointed to by FSR, which is incremented from 10h to 1Fh.

Note that at the test at the end of the loop, '`btfsc FSR,4`', finishes the loop when the end of bank 0 (1Fh) has been reached. In fact, this test can be used for the end of any bank, not just bank 0.

Example 2: Light meter with smoothed decimal output

To effectively smooth the light meter's output, so that it doesn't jitter under fluorescent lighting, a simple moving average is quite adequate – assuming that the sample *window* (the time that samples are averaged over) is longer than the variations to be smoothed.

The electricity supply, and hence the output of most A/C lighting, cycles at 50 or 60 Hz in most places. A 50 Hz cycle is 20 ms long, so the sample window needs to be longer than that. Our example light meter program samples every 4 ms, so at least five samples need to be averaged (5 x 4 ms = 20 ms) to smooth a 50 Hz cycle. But a longer window would be better; two or three times the cycle time would ensure that cyclic variations are smoothed out.

We have seen that the data memory on any baseline PIC with multiple data memory banks is not contiguous. The 16F506 has four banked 16-byte general purpose register (GPR) regions (forming the "top half" of each of the four banks), plus one 3-byte non-banked (or shared) GPR region. Thus, the largest contiguous block of memory that can be allocated on the 16F506 is 16 bytes. It is easiest to implement arrays if they are contiguous, so the largest single array we can easily define is 16 bytes – which happens to be a good size for the sample array (or *buffer*) for this application.

Since each data section has to fit within a single data memory region, and the largest available data memory region on a PIC16F506 is 16 bytes, if you try something like:

```
        UDATA
adc_dec res 2                   ; scaled ADC output (LE 16 bit, 0-99 in MSB)
mpy_cnt res 1                   ; multiplier count
smp_buf res 16                  ; array of samples for moving average
```

you will get a "`'.udata' can not fit the section`" error from the linker, because we have tried to reserve a total of 19 bytes in a single `UDATA` section. Unnamed `UDATA` sections are given the default name '`.udata`', so the error message is telling us that this section, which is named '`.udata`', is too big.

So we need to split the variable definitions into two (or more) `UDATA` sections, with no more than 16 bytes in each section. To declare more than one `UDATA` section, they have to have different names, for example:

```
VARS1   UDATA
adc_dec res 2                   ; scaled ADC output (LE 16 bit, 0-99 in MSB)
mpy_cnt res 1                   ; multiplier count

ARRAY1  UDATA
smp_buf res 16                  ; array of samples for moving average
```

Although we don't know which bank the array will be placed in, we do know that it will fill the whole of one of the 16-byte banked GPR memory regions, forming the top half of whichever bank it is in.

That means that to clear the array, we can adapt the code from the data sheet:

```
        ; clear sample buffer
        movlw   smp_buf
        movwf   FSR
l_clr   clrf    INDF            ; clear each byte
        incf    FSR,f
        btfsc   FSR,4           ; until end of bank is reached
        goto    l_clr
```

This approach wouldn't work if the array was any smaller than 16 bytes, in which case we would need to use a subtraction or XOR to test for FSR reaching the end of the array.

Since the 16-byte array uses all the banked data space in one bank, there is no additional room in that bank to store any other variables we may need to access while working with the array, such as the running total of sample values in the array. In the baseline architecture, accessing variables in other banks is very awkward when using indirect memory access, because selecting another bank means changing FSR, which is being used to access the array.

To reduce the number of bank selection changes necessary, and the need to save/restore FSR after each one, it makes sense to place variables associated with the array in shared memory, wherever possible.

For example:

```
SHR1    UDATA_SHR
adc_sum res 2                   ; sum of samples (LE 16-bit), for average
adc_avg res 1                   ; average ADC output
```

It was ok to work directly with FSR in the "clear sample buffer" loop above, since it is short and no bank selection occurs within it. But it's not practical to remove the need for banking altogether throughout the sampling loop, where we read a sample, update the moving average calculation, scale the result, convert it to decimal and then display it, before moving on to the next sample. So we need to save the pointer to the "current" sample in a variable ('`smp_idx`') which will not be overwritten when a bank is selected.

Updating and calculating the total of the samples (stored in a 16-bit variable called '`adc_sum`') is done as follows:

```
        banksel smp_idx
        movf    smp_idx,w       ; set FSR to current sample buffer index
        movwf   FSR
        movf    INDF,w          ; subtract old sample from running total
        subwf   adc_sum,f
        btfss   STATUS,C
        decf    adc_sum+1,f
        movf    ADRES,w         ; save new sample (ADC result)
        movwf   INDF
```

```
addwf   adc_sum,f       ; and add to running total
btfsc   STATUS,C
incf    adc_sum+1,f
```

This total then has to be divided by 16 (the number of samples) to give the moving average.

As we've seen, dividing by any power of two can be simply done through a series of right-shifts. In this case, since we need to keep '`adc_sum`' intact from one loop iteration to the next (to maintain the running total), we would need to take a copy of it and right-shift the copy four times (to divide by 16). Since '`adc_sum`' is a 16-bit quantity, both the MSB and LSB would have to be right-shifted, so we'd need eight right-shifts in total, plus a few instructions to copy '`adc_sum`' – around a dozen instructions in total.

But since we need to right-shift by four bits, and the `swapf` instruction swaps the nybbles (four bits) in a byte, shifting the upper nybble right by four bits, we can use it to divide by 16 more efficiently.

Suppose the running total in '`adc_sum`' is 0ABCh. (The upper nybble will always be zero because the result of adding 16 eight-bit numbers is a twelve-bit number; the sum can never be more than 0FF0h).

The result we want (0ABCh divided by 16, or right-shifted four times) is ABh.

Swapping the nybbles in the LSB gives CBh. Next we need to clear the high nybble to remove the 'C', which as we saw in lesson 8, can be done through a masking operation, using AND, leaving 0Bh.

Swapping the nybbles in the MSB gives A0h.

Finally we need to combine the upper nybble in the MSB (A0h) with the lower nybble in the LSB (0Bh).

This can be done with an inclusive-or, since any bit ORed with '0' remains unchanged, while any bit ORed with '1' is set to '1'. That is:

n OR 0 = n

n OR 1 = 1

So, for example, A0h OR 0Bh = ABh. (In binary, 1010 0000 OR 0000 1011 = 1010 1011.)

The baseline PICs provide two "inclusive-or" instructions:

`iorwf` – "**i**nclusive-**or W** with register **f**ile"

`iorlw` – "**i**nclusive-**or l**iteral with **W**"

These are used in the same way as the exclusive-or instructions we've seen before.

For completeness, the baseline PICs provide one more logic instruction we haven't covered so far:

`andwf` – "**and W** with register **f**ile"

We can use '`swapf`' to rearrange the nybbles, '`andlw`' to mask off the unwanted nybble, and '`iorwf`' to combine the bytes, creating an efficient "divide by 16" routine, as follows:

```
swapf   adc_sum,w       ; divide total by 16
andlw   0x0F
movwf   adc_avg
swapf   adc_sum+1,w
iorwf   adc_avg,f
```

The result is the moving average, which can be scaled, converted to decimal and displayed as before.

Complete program

Although much of this code is the same as in the previous example, here is the complete "light meter with smoothed decimal display" program, showing how all the parts fit together:

```
;*****************************************************************
;                                                                *
;   Description:    Lesson 11, example 2                         *
;                                                                *
;   Demonstrates use of indirect addressing                      *
;   to implement a simple moving average filter                  *
;                                                                *
;   Displays ADC output in decimal on 2-digit 7-segment LED display *
;                                                                *
;   Continuously samples analog input, averages last 16 samples, *
;   scales result to 0 - 99 and displays as 2 x dec 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                      *
;                                                                *
;*****************************************************************

    list        p=16F506
    #include    <p16F506.inc>

    radix       dec


;***** CONFIGURATION
                ; ext reset, no code protect, no watchdog, 4 MHz int clock
    __CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _IOSCFS_OFF & _IntRC_OSC_RB4EN

; pin assignments
    #define TENS_EN     PORTC,5     ; tens digit enable
    #define ONES_EN     PORTB,5     ; ones digit enable


;***** VARIABLE DEFINITIONS
VARS1   UDATA
adc_dec res 2                   ; scaled ADC output (LE 16 bit, 0-99 in MSB)
mpy_cnt res 1                   ; multiplier count
smp_idx res 1                   ; index into sample array
                                ; digits to be displayed:
tens    res 1                   ;   tens
ones    res 1                   ;   ones

temp   res 1                    ; (temp storage used by set7seg)

ARRAY1  UDATA
smp_buf res 16                  ; array of samples for moving average

SHR1    UDATA_SHR
adc_sum res 2                   ; sum of samples (LE 16-bit), for average
adc_avg res 1                   ; average ADC output
```

```
;***** RC CALIBRATION
RCCAL   CODE    0x3FF           ; processor reset vector
        res 1                   ; holds internal RC cal value, as a movlw k


;***** RESET VECTOR ****************************************************
RESET   CODE    0x000           ; effective reset vector
        movwf   OSCCAL          ; apply internal RC factory calibration
        pagesel start
        goto    start           ; jump to main code

;***** SUBROUTINE VECTORS
set7seg                         ; display digit on 7-segment display
        pagesel set7seg_R
        goto    set7seg_R


;***** MAIN PROGRAM ****************************************************
MAIN    CODE

;***** Initialisation
start
        ; configure ports
        clrw                    ; configure PORTB and PORTC as all outputs
        tris    PORTB
        tris    PORTC
        clrf    CM1CON0         ; disable comparator 1 -> RB0, RB1 digital
        clrf    CM2CON0         ; disable comparator 2 -> RC0, RC1 digital
        clrf    VRCON           ; disable CVref -> RC2 usable

        ; configure ADC
        movlw   b'01111001'     ; configure ADC:
                ; 01------          AN2 (only) analog (ANS = 01)
                ; --11----          clock = INTOSC/4 (ADCS = 11)
                ; ----10--          select channel AN2 (CHS = 10)
                ; -------1          turn ADC on (ADON = 1)
        movwf   ADCON0          ;   -> AN2 ready for sampling

        ; configure timer
        movlw   b'11010111'     ; configure Timer0:
                ; --0-----          timer mode (T0CS = 0) -> RC5 usable
                ; ----0---          prescaler assigned to Timer0 (PSA = 0)
                ; -----111          prescale = 256 (PS = 111)
        option                  ;   -> increment every 256 us
                                ;      (TMR0<2> cycles every 2.048ms)

        ; initialise variables
        clrf    adc_sum         ; sample buffer total = 0
        clrf    adc_sum+1

        ; clear sample buffer
        movlw   smp_buf
        movwf   FSR
l_clr   clrf    INDF            ; clear each byte
        incf    FSR,f
        btfsc   FSR,4           ; until end of bank is reached
        goto    l_clr


;***** Main loop
main_loop
```

```
        ; set index to start of sample buffer
        movlw   smp_buf
        banksel smp_idx
        movwf   smp_idx

; *** repeat for each sample in buffer
l_smp_buf

        ; sample input
        bsf     ADCON0,GO       ; start conversion
w_adc   btfsc   ADCON0,NOT_DONE ; wait until conversion complete
        goto    w_adc

        ; calculate moving average
        banksel smp_idx
        movf    smp_idx,w       ; set FSR to current sample buffer index
        movwf   FSR
        movf    INDF,w          ; subtract old sample from running total
        subwf   adc_sum,f
        btfss   STATUS,C
        decf    adc_sum+1,f
        movf    ADRES,w         ; save new sample (ADC result)
        movwf   INDF
        addwf   adc_sum,f       ; and add to running total
        btfsc   STATUS,C
        incf    adc_sum+1,f
        swapf   adc_sum,w       ; divide total by 16
        andlw   0x0F
        movwf   adc_avg
        swapf   adc_sum+1,w
        iorwf   adc_avg,f

        ; scale to 0-99: adc_dec = adc_avg * 100
        ;   -> MSB of adc_dec = adc_avg * 100 / 256
        banksel adc_dec
        clrf    adc_dec         ; start with adc_dec = 0
        clrf    adc_dec+1
        movlw   .8              ;   count = 8
        movwf   mpy_cnt
        movlw   .100            ;   multiplicand (100) in W
        bcf     STATUS,C        ;   and carry clear
l_mpy   rrf     adc_avg,f       ; right shift multiplier
        btfsc   STATUS,C        ; if low-order bit of multiplier was set
        addwf   adc_dec+1,f     ;   add multiplicand (100) to MSB of result
        rrf     adc_dec+1,f     ; right shift result
        rrf     adc_dec,f
        decfsz  mpy_cnt,f       ; repeat for all 8 bits
        goto    l_mpy

        ; extract digits of result
        movf    adc_dec+1,w     ; start with scaled result
        movwf   ones            ;   in ones digit
        clrf    tens            ; and tens clear
l_bcd   movlw   .10             ; subtract 10 from ones
        subwf   ones,w
        btfss   STATUS,C        ; (finish if < 10)
        goto    end_bcd
        movwf   ones
        incf    tens,f          ; increment tens
        goto    l_bcd           ; repeat until ones < 10
end_bcd
```

```
        ; display tens digit for 2.048 ms
w10_hi  btfss   TMR0,2          ; wait for TMR0<2> to go high
        goto    w10_hi
        movf    tens,w          ; output tens digit
        pagesel set7seg
        call    set7seg
        pagesel $
        bsf     TENS_EN         ; enable tens display
w10_lo  btfsc   TMR0,2          ; wait for TMR<2> to go low
        goto    w10_lo

        ; display ones digit for 2.048 ms
w1_hi   btfss   TMR0,2          ; wait for TMR0<2> to go high
        goto    w1_hi
        banksel ones            ; output ones digit
        movf    ones,w
        pagesel set7seg
        call    set7seg
        pagesel $
        bsf     ONES_EN         ; enable ones display
w1_lo   btfsc   TMR0,2          ; wait for TMR<2> to go low
        goto    w1_lo

        ; end sample buffer loop
        banksel smp_idx         ; increment sample buffer index
        incf    smp_idx,f
        btfsc   smp_idx,4       ; repeat loop until end of buffer
        goto    l_smp_buf

        ; repeat main loop forever
        goto    main_loop


;***** LOOKUP TABLES
TABLES  CODE    0x200           ; locate at beginning of a page

; pattern table for 7 segment display on port B
;   RB4 = E, RB1:0 = FG
get7sB  addwf   PCL,f
        retlw   b'010010'       ; 0
        retlw   b'000000'       ; 1
        retlw   b'010001'       ; 2
        retlw   b'000001'       ; 3
        retlw   b'000011'       ; 4
        retlw   b'000011'       ; 5
        retlw   b'010011'       ; 6
        retlw   b'000000'       ; 7
        retlw   b'010011'       ; 8
        retlw   b'000011'       ; 9

; pattern table for 7 segment display on port C
;   RC4:1 = CDBA
get7sC  addwf   PCL,f
        retlw   b'011110'       ; 0
        retlw   b'010100'       ; 1
        retlw   b'001110'       ; 2
        retlw   b'011110'       ; 3
        retlw   b'010100'       ; 4
        retlw   b'011010'       ; 5
        retlw   b'011010'       ; 6
        retlw   b'010110'       ; 7
```

```
        retlw   b'011110'       ; 8
        retlw   b'011110'       ; 9

; Display digit passed in W on 7-segment display
set7seg_R
        ; disable displays
        clrf    PORTB           ; clear all digit enable lines on PORTB
        clrf    PORTC           ;   and PORTC

        ; output digit pattern
        banksel temp
        movwf   temp            ; save digit
        call    get7sB          ; lookup pattern for port B
        movwf   PORTB           ;   then output it
        movf    temp,w          ; get digit
        call    get7sC          ;   then repeat for port C
        movwf   PORTC
        retlw   0


        END
```

You should find that the resulting display is stable, even under fluorescent lighting, and yet still responds quickly to changing light levels.

Conclusion

This tutorial series has now introduced every baseline PIC instruction and every special function register (except those associated with EEPROM access on those few baseline PICs with EEPROMs).

That concludes our introduction to the baseline PIC architecture and assembly programming.

The material in these lessons is revisited in a tutorial series on programming baseline PICs in C.

In that series it becomes apparent that some tasks are more easily expressed in C than assembler, especially the most recent topic of arithmetic and arrays, but that C can be relatively inefficient. It is also seen that different C compilers take different approaches – with pros and cons that become apparent as the various examples are implemented in each.

Now that you have a basic understanding of programming baseline PICs in assembler (and C, if you go through the baseline C tutorial series), you may wish to move on to the midrange PIC architecture and assembler tutorials, where you will be introduced to the more flexible and capable midrange PIC core, and some of its diverse range of peripherals. These lessons are also followed up by a series on programming midrange PICs in C.

Enjoy!