

# Introduction to PIC Programming

## Midrange Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

### Lesson 14: 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; they normally need to be processed in some way before being displayed or used in decision making. While advanced signal processing is beyond the capabilities of midrange PICs, this lesson demonstrates that simpler post-processing, such as integer scaling and implementing a simple filter, such as a moving average, can be readily accomplished with even the smallest midrange PICs.

This lesson revisits material from [baseline lesson 11](#), introducing some 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](http://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 midrange PICs seem to have very limited arithmetic capabilities: only two 8-bit addition instructions (`addwf` and `addlw`) and two 8-bit subtraction instructions (`subwf` and `sublw`).

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` and `addlw` instructions set the carry flag if the result of the addition *overflows* a single byte (that is, the result is greater than 255).

And as explained in [lesson 4](#), the carry flag acts as a “not borrow” in a subtraction: the `subwf` and `sublw` instructions clear `C` to ‘0’ if a borrow occurs (that is, if 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 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 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 example, for a 32-bit subtraction:

```

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
```

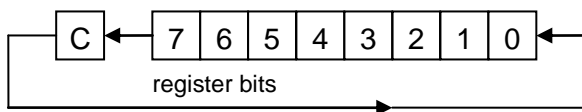
There is a new instruction here: 'comf f,d' – “**complement register file**”, which calculates the ones' complement of register 'f', placing the result back into the register if the destination is 'f', or in W if the destination is 'w'.

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 might seem that the midrange 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:

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

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

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:

```

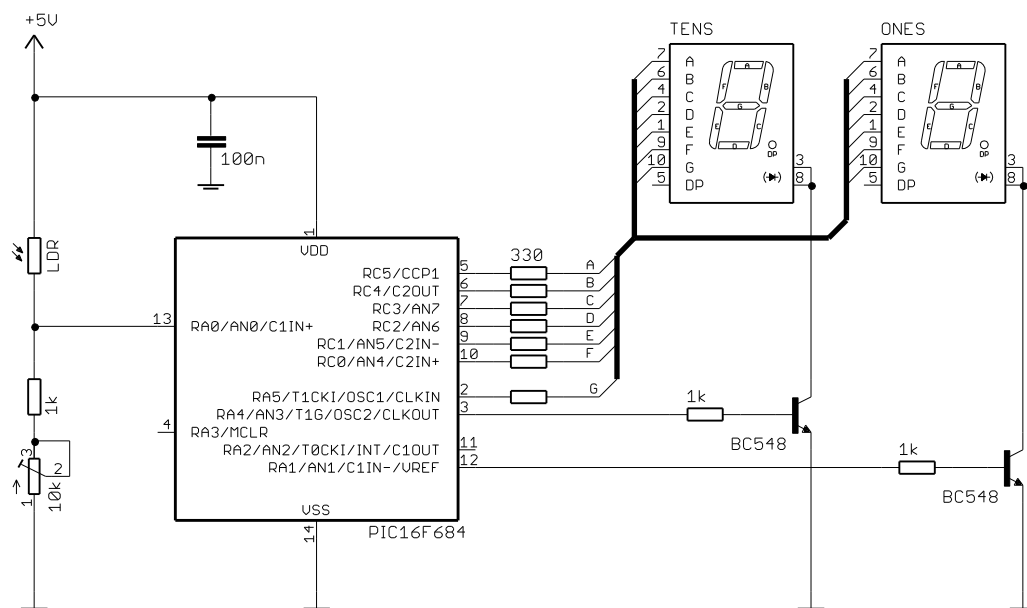
; Variables:
; mulcnd - 8 bit multiplicand
; mulplr - 8 bit multiplier
; H_byte - High byte of the 16 bit result
; L_byte - Low byte of the 16 bit result
; count - loop counter
;
; *****_***** Begin Multiplier Routine
mpy_S   clrfs   H_byte   ; start with result = 0
        clrfs   L_byte
        movlw   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 13](#) included a simple light meter based on a light-dependent resistor, which displayed the 10-bit ADC output as a three-digit hexadecimal number, using 7-segment LED displays. 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 0 – 99 instead of 0 – 3FFh.

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



To scale the ADC output from 0 – 1023 to 0 – 99, multiply the 10-bit result by 99/1023.

Multiplying by 99 isn't difficult, but dividing by 1023 is.

The task is made much simpler by using an approximation: instead of multiplying by 99/1023, multiply by 100/1024. That's a difference of 0.9%; close enough for our purpose here.

As mentioned above, dividing by a power of two, such as 1024 ( $= 2^{10}$ ) is easy, requiring only a sequence of right-shift operations – to divide by 1024, right-shift the result 10 times.

We can make the task even easier by treating the ADC result as an 8-bit number, by configuring the ADC to left-justify the result, and using only the eight most-significant-bits in ADRESH (discarding the two LSBs in ADRESL); when the output consists of only two digits, there is no need to retain full 10-bit ADC resolution.

This means that, to scale the result to 0 – 99, we can multiply the 8-bit ADC result by 100/256.

To divide a 16-bit number by 256, we don't need to actually perform eight right shifts, because 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.

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, such as lumens, 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 8-bit ADC result (copied from ADRESH to the variable 'adc\_out') 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 8-bit 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 8-bit ADC result  $\times 100/256$ .

And this is equal to the full 10-bit result  $\times 100/1024$ .

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 for ISR to display
        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.

There is, however, a problem with this. While this code within the main loop is executing, interrupts are continuing to run in the background; the ISR, which displays the current value of the ones and tens variables, can run *at any time* – including the point when the scaled binary result has been copied into the ones variable, at the start of this decimal conversion routine.

If the ISR was to run at that time, it is likely to attempt to display a 'ones' value that is greater than nine. Since the value in 'ones' is used as an index into the segment pattern lookup tables, with no bounds testing, there is a strong chance that the code will attempt to lookup a value past the end of the table. Given that the table lookup works through a "computed goto", executing a 'retlw' instruction at a given displacement from the start of the table, if your code tries to lookup a value past the end of the table, it will attempt to execute an instruction sitting somewhere beyond the table. This could be another part of your code, or it could be unused program memory space. It doesn't matter; the effect is the same – your program will crash!

To avoid this, there are a number of possible approaches.

Instead of displaying the 'ones' and 'tens' variables directly, the ISR could display some other variables, say 'ones\_dsp' and 'tens\_dsp'. Your code would calculate 'ones' and 'tens', as above, and then copy the final, correct (within bounds) values into the 'ones\_dsp' and 'tens\_dsp' variables for display.

That's ok, but it means allocating more data memory for variables. One way to avoid that is to change the decimal extraction routine to operate on 'adc\_dec+1' directly, and copy the result to 'ones' only at the end of the routine. That's ok if you don't mind losing the value in 'adc\_dec' (in this example, it doesn't matter if it is overwritten), but it's a little ugly.

A straightforward alternative is to simply disable interrupts before the digit extraction routine:

```

        ; disable interrupts during calculation
        ; (stops ISR trying to display out-of-range intermediate results)
        bcf    INTCON,GIE

```

and then re-enable them after the calculation is complete, when the 'ones' and 'tens' variables hold values suitable for display:

```
    ; re-enable interrupts
    bsf     INTCON,GIE
```

It's not always appropriate to disable interrupts, but there is no problem doing so in this case, so that's the approach we'll take here.

In fact, since we are disabling interrupts during part of the loop, it makes sense to keep them disabled while copying the shadow registers (which are updated by the ISR) to the ports, to ensure that the values written to PORTA and PORTC are always consistent with each other.

### Complete program

The rest of the program is essentially the same as the hexadecimal-output example from [lesson 13](#). Here is how the scaling and digit extraction routines, and the disabling and re-enabling of interrupts, presented above, fit in:

```

;*****
;
; Description:      Lesson 14, example 1
;
; Displays ADC output in decimal on 2x7-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:
; AN0              = voltage to be measured (e.g. pot or LDR)
; RA5, RC0-5      = 7-segment display bus (common cathode)
; RA4              = tens enable (active high)
; RA1              = ones enable
;
;*****

list          p=16F684
#include      <p16F684.inc>

#include      <stdmacros-mid.inc>          ; Delay10us          - 10 us delay
                                           ; Lookup table,off_var
                                           ; - lookup table entry
                                           ; at offset held in off_var

errorlevel   -302      ; no "register not in bank 0" warnings
errorlevel   -312      ; no "page or bank selection not needed" messages

radix        dec

;***** CONFIGURATION
                ; ext reset, no code or data protect, no brownout detect,
                ; no watchdog, no power-up timer, int clock with I/O,
                ; no failsafe clock monitor, two-speed start-up disabled
__CONFIG      _MCLRE_ON & _CP_OFF & _CPD_OFF & _BOD_OFF & _WDT_OFF &
_PWRTE_OFF & _INTOSCIO & _FCMEN_OFF & _IESO_OFF

```

```

; pin assignments
    #define STENS_EN    SPORTA,4    ; tens enable (shadow)
    #define SONES_EN   SPORTA,1    ; ones enable

;***** VARIABLE DEFINITIONS
CONTEXT    UDATA_SHR           ; variables used for context saving
cs_W      res 1
cs_STATUS  res 1

SHADOW     UDATA_SHR           ; shadow registers
SPORTA     res 1                ; PORTA
SPORTC     res 1                ; PORTC

GENVAR     UDATA                ; general variables
mpx_cnt    res 1                ; multiplex counter
adc_out    res 1                ; raw ADC output (8-bit)
adc_dec    res 2                ; scaled ADC output (LE 16 bit, 0-99 in MSB)
mpy_cnt    res 1                ; multiplier count
                                ; result in decimal (displayed by ISR):
tens       res 1                ; tens
ones       res 1                ; ones

;***** RESET VECTOR *****
RESET     CODE    0x0000        ; processor reset vector
          pagesel Start
          goto    Start

;***** INTERRUPT SERVICE ROUTINE *****
ISR       CODE    0x0004
          ; *** Save context
          movwf   cs_W           ; save W
          movf    STATUS,w       ; save STATUS
          movwf   cs_STATUS

          ; *** Service Timer0 interrupt
          ; TMR0 overflows every 2.048 ms
          ; (only Timer0 interrupts are enabled)
          ;
          bcf     INTCON,T0IF     ; clear interrupt flag

          ; Display scaled ADC result on 2 x 7-segment displays
          ; mpx_cnt determines current digit to display
          ;
          banksel mpx_cnt
          incf   mpx_cnt,f        ; increment mpx_cnt for next digit
          movf   mpx_cnt,w        ; and copy to W
          ; determine current mpx_cnt by successive subtraction
          addlw  -1
          btfsc  STATUS,Z        ; if current mpx_cnt = 0
          goto   dsp_ones        ; display ones digit
          clrf   mpx_cnt         ; else mpx_cnt = 1, so reset to 0
          goto   dsp_tens        ; and display tens digit

dsp_ones
          ; display ones digit (using shadow registers)
          Lookup tb7segA,ones     ; lookup ones pattern for PORTA
          movwf  SPORTA          ; then output it
          Lookup tb7segC,ones     ; repeat for PORTC
          movwf  SPORTC

```

```

        bsf      sONES_EN          ; enable ones display
        goto    dsp_end

dsp_tens
        ; display tens digit
        Lookup  tb7segA,tens      ; output tens digit
        movwf   sPORTA
        Lookup  tb7segC,tens
        movwf   sPORTC
        bsf     sTENS_EN          ; enable tens display
dsp_end

isr_end ; *** Restore context then return
        movf    cs_STATUS,w       ; restore STATUS
        movwf   STATUS
        swapf   cs_W,f           ; restore W
        swapf   cs_W,w
        retfie

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

;***** Initialisation
Start   ; configure ports
        banksel TRISC           ; configure PORTC as all outputs
        clrf   TRISC
        movlw  1<<0             ; configure RA0/AN0 (only) as an input
        movwf  TRISA
        banksel ANSEL          ; make only AN0 analog
        movwf  ANSEL
        movlw  .7               ; disable comparators
        banksel CMCON0         ; CM = 7 -> both comparators off
        movwf  CMCON0

        ; configure timer
        movlw  b'11000010'      ; configure Timer0:
                ; --0-----   timer mode (TOCS = 0)
                ; ----0----   prescaler assigned to Timer0 (PSA = 0)
                ; -----010   prescale = 8 (PS = 010)
        banksel OPTION_REG      ; -> increment TMR0 every 8 us
        movwf  OPTION_REG       ; -> TMR0 overflows every 2.048 ms

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

        ; initialise variables
        banksel tens           ; clear display variables
        clrf   tens            ; to ensure that out-of-range values
        clrf   ones            ; are not displayed
        clrf   mpx_cnt         ; mpx_cnt = 0 -> display ones digit first

```

```

; enable interrupts
movlw 1<<GIE|1<<T0IE ; enable Timer0 and global interrupts
movwf INTCON

;***** Main loop
main_loop
; sample analog input
Delay10us ; wait 10 us for acquisition time
banksel ADCON0 ; start conversion
bsf ADCON0,GO
waitadc btfsc ADCON0,NOT_DONE ; wait until done
goto waitadc
banksel ADRESH ; save high 8 bits of ADC result
movf ADRESH,w
banksel adc_out ; to 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

; disable interrupts during calculation
; (stops ISR trying to display out-of-range intermediate results)
bcf INTCON,GIE

; extract digits of result for ISR to display
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

; copy shadow registers (updated by ISR) to ports
banksel PORTA
movf sPORTA,w
movwf PORTA
movf sPORTC,w
movwf PORTC

; re-enable interrupts
bsf INTCON,GIE

```

```

; repeat forever
pagesel main_loop
goto    main_loop

;***** LOOKUP TABLES *****
TABLES  CODE

; Digit to pattern lookup table for 7 segment display on port A
;   RA5 = G
tb7segA movwf  PCL
        retlw  b'000000'    ; 0
        retlw  b'000000'    ; 1
        retlw  b'100000'    ; 2
        retlw  b'100000'    ; 3
        retlw  b'100000'    ; 4
        retlw  b'100000'    ; 5
        retlw  b'100000'    ; 6
        retlw  b'100000'    ; 7
        retlw  b'000000'    ; 8
        retlw  b'100000'    ; 9

; Digit to pattern lookup table for 7 segment display on port C
;   RC5:0 = ABCDEF
tb7segC movwf  PCL
        retlw  b'111111'    ; 0
        retlw  b'011000'    ; 1
        retlw  b'110110'    ; 2
        retlw  b'111100'    ; 3
        retlw  b'011001'    ; 4
        retlw  b'101101'    ; 5
        retlw  b'101111'    ; 6
        retlw  b'111000'    ; 7
        retlw  b'111111'    ; 8
        retlw  b'111101'    ; 9

END

```

## Moving Averages, Indirect Addressing and Arrays

### ***Moving averages***

We saw in [lesson 13](#) that 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 (“continuous” sampling with 244 display updates per second), the display changes too quickly 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 ideal; 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 here.

### **Indirect addressing and arrays**

A “list of samples” would normally be described as an *array*, or *buffer*.

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: midrange PICs support *indirect addressing* (making array indexing possible), through the FSR and INDF registers.

The INDF (**indirect file**) “register” acts as a window, through which the contents of any register can be accessed.

The FSR (**file select register**) holds the address of the register which will be accessed through INDF.

For example, if FSR = 07h, INDF accesses the register at address 07h, which is PORTC on the PIC16F684. So, on the PIC16F684, if FSR = 07h, reading or writing INDF is the same as reading or writing PORTC.

FSR is an 8-bit register, so it is able to address 256 data memory locations.

In the midrange PIC architecture, data memory is banked, with each bank consisting of 128 registers – as described in [lesson 1](#). Thus, FSR can be used to access any register in the first two banks.

Since the 16F684 has only two register banks, any register can be addressed, through INDF, by writing that register’s address into FSR, irrespective of the RP0 and RP1 bank selection bits in the STATUS register. That is, indirect addressing allows linear, un-banked access to the entire register file, on midrange PICs with two banks of data memory<sup>1</sup>.

---

<sup>1</sup> In midrange PICs with four banks of data memory, such as the 16F690 or 16F887, the IRP bit in the STATUS register is used to select which pair of banks (0 and 1 or 2 and 3) is accessed through INDF.

IRP acts as if it was the 9<sup>th</sup> bit of the FSR register. E.g. if IRP = 1 and FSR = 05h, INDF will access the register at 105h, in bank 3. If we then clear IRP, INDF will access the register at 05h, in bank 0.

For example, if `FSR = 91h`, `INDF` will access the register at address 91h; this happens to be in bank 1, but that's not a consideration when using indirect addressing on the 16F684.

The PIC16F684 data sheet includes the following code to clear registers 20h – 2Fh:

```

        movlw    0x20        ; initialize pointer to RAM
        movwf   FSR
next    clrfs   INDF        ; indirectly clear register (pointed to by FSR)
        incf   FSR, f      ; inc pointer
        btfss  FSR, 4      ; all done?
        goto   next       ; NO, clear next
continue        ; YES, continue

```

The `'clrfs INDF'` instruction clears the register pointed to by `FSR`, which is incremented from 20h to 2Fh.

Note that an instruction like `'btfss FSR, 4'` can be an efficient way of testing for the final value of a loop counter, but it's only applicable in cases where a particular single bit in the counter will only change when the end of the loop is reached (`FSR<4>` in this case).

### **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; the sample window needs to be longer than that, so that a series of raw samples are averaged across at least a whole cycle.

To limit the number of samples needed, it is best to sample in the input at a steady rate, such as once per millisecond. This can be done by using a timer interrupt, and, as we saw in [lesson 13](#), since we are already using a timer interrupt to drive the display, we can use that interrupt to initiate the AD conversions as well. An ADC interrupt is then used to process the conversion result, instead of polling the `GO/DONE` flag.

In the example in [lesson 13](#), the AD conversion was initiated only when every digit of the display had been updated, instead of every time the timer interrupt ran, because there was no advantage gained by sampling the input faster than the result could be displayed.

When the ADC results will be stored and averaged (or filtered in some other way), this reasoning no longer applies. Instead, it can be useful to *oversample* the input, sampling it more often than it can be displayed (or logged or processed in some other way), and averaging the results to effectively provide a higher-resolution result, equivalent to using an ADC module with more output bits<sup>2</sup>.

We've been using a timer interrupt running every 2 ms (approx), and if we use this interrupt to initiate the conversions, we will need to average at least ten samples ( $10 \times 2 \text{ ms} = 20 \text{ ms}$ ) 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. So averaging a minimum of twenty, and preferably thirty or more (if we have enough data memory to store them) samples, spaced 2 ms apart, seems to be a reasonable choice.

In the previous example, we discarded the lower two bits of the ADC's 10-bit result, because we were only displaying an integer from 0 to 99 and dealing with 8-bit numbers made the calculations easier. But in this

---

<sup>2</sup> This actually depends on the presence of appropriately-distributed noise in the input; search for "oversampling" on the Internet if you wish to learn more...

example, we'll work with the full 10-bit results, maintaining more resolution than the display is able to show, simply to demonstrate how it can be done.

To store a 10-bit result, it is easiest to use two bytes of data memory, with the lower eight bits in one byte, and the upper two bits in another, treating them as 16-bit variables, and performing normal 16-bit addition.

Therefore, we will allocate two bytes of data memory for each result to be stored in the sample buffer.

Recall (see [lesson 10](#)) that the PIC16F684 has 80 general purpose registers (GPRs) in bank 0, another 32 GPRs in bank 1, and 16 shared GPRs mapped into both banks.

This means that the largest contiguous block of data memory on the 16F684 is 80 bytes, and given that a simple, single array<sup>3</sup> has to be contiguous, the largest single array we can define on the 16F684 is 80 bytes. Hence, the largest sample buffer we can implement on a 16F684 is  $40 \times 16$ -bit samples.

Note that, on a larger PIC, such as the 16F690, with more data memory, it would be possible to implement a larger sample buffer by using two arrays, each in a separate bank; one for the least significant bytes (LSBs) and a separate array storing the most significant bytes (MSBs) of each sample. But the 16F684 doesn't have enough memory in bank 1 to make that approach worthwhile and besides, using a single array to hold the sample buffer is more straightforward.

To make it easier to change the number of samples later, without having to find all the references in the code, we'll define the numbers of samples (and hence, buffer size) as constants:

```
constant NUM_SAMPLES=40          ; number of samples in moving avg buffer
constant BUF_SIZE=NUM_SAMPLES*2 ; moving avg buffer size (16 bits/sample)
```

Since each data section has to fit within a single data memory region, and the largest contiguous data memory region on a PIC16F684 is 80 bytes, if we try something like:

```
                UDATA          ; general variables
adc_sum        res 2           ; sum of ADC samples (LE 16-bit)
adc_dec        res 3           ; scaled ADC output (LE 24-bit, 0-99 in MSB)
mpy_cnt        res 1           ; multiplier count

smp_buf        res BUF_SIZE    ; array of samples for moving average
```

we will get a `“.udata” can not fit the section` error from the linker, because we have tried to reserve a total of 86 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 80 bytes in the largest section, for the linker to place within available data memory. To declare more than one UDATA section, they have to have different names, for example:

```
GENVAR         UDATA          ; general variables
adc_sum        res 2           ; sum of ADC samples (LE 16-bit)
adc_dec        res 3           ; scaled ADC output (LE 24-bit, 0-99 in MSB)
mpy_cnt        res 1           ; multiplier count

ARRAY1         UDATA          ; array of samples for moving average
smp_buf        res BUF_SIZE    ; NUM_SAMPLES x 16-bit (little endian)
```

---

<sup>3</sup> One that can be accessed by using FSR as a simple index, as opposed to a more elaborate structure such as a linked list

This will now fit within the 16F684's memory, as long as the buffer size is no more than 40 samples.

Before the sample buffer is used, it has to be cleared.

We cannot use the method presented above, because it will only work in certain specific cases. Instead, we'll use the more general method of using XOR to test for the final value at the end of the loop:

```

        ; clear sample buffer
        movlw   smp_buf
        movwf   FSR
l_clr   clrfs   INDF           ; clear each byte
        incf   FSR,f
        movlw   smp_buf+BUF_SIZE ; until end of buffer is reached
        xorwf   FSR,w
        btfss  STATUS,Z
        goto   l_clr

```

With the samples in the buffer all cleared to zero, we know that the sum of those samples must also be zero. So if we also zero the running total ('adc\_sum'), we can be sure that the initial value of the running total is equal to the initial sum of the samples in the buffer, because they are all equal to zero:

```

        ; initialise variables
        banksel tens           ; clear display variables
        clrfs   tens           ; to ensure that out-of-range values
        clrfs   ones           ; are not displayed
        clrfs   mp_x_cnt       ; mp_x_cnt = 0 -> display ones digit first
        clrfs   adc_sum        ; sample buffer total = 0
        clrfs   adc_sum+1

```

Then, as we replace samples in the buffer, subtracting the old sample from the running total before adding the new sample, the running total will continue to accurately reflect the contents of the sample buffer.

Since we will not be using FSR for anything other than as the index into the sample buffer, there is no need to define a separate variable to act as an index; we can simply use FSR directly. But before the interrupts begin, we need to ensure that FSR is pointing to the start of the array:

```

        ; initialise sample buffer index
        movlw   smp_buf        ; FSR is only used to access the sample buffer
        movwf   FSR           ; so point it to the start of the buffer

```

Storing each sample in the buffer and calculating the moving average is done by the ADC interrupt handler.

First, we need to subtract the oldest sample in the buffer from the running total, 'adc\_sum'.

Since the samples and running total are 16-bit quantities, we can adapt the 16-bit subtraction routine presented earlier:

```

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

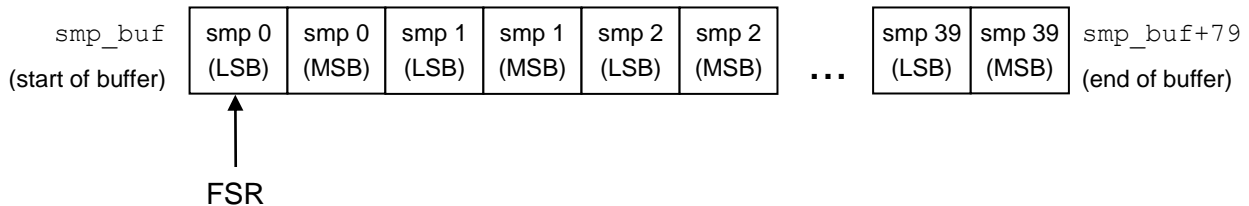
```

Variable 'a' corresponds to the sample being subtracted and variable 'b' corresponds to the running total. So, for 'b' we can simply substitute 'adc\_sum'.

Referencing the sample to be subtracted is another story, because we need to be able to reference a different location in the sample buffer, each time the ADC interrupt runs. We can't simply use the fixed address of a normal variable, like 'a', because we need to be able to access a variable in a different memory location each time. So we have to use indirect addressing.

FSR already points to the current sample, because we initialised it to point to the first sample in the buffer, and when we finish processing this sample we'll advance FSR to point to the next sample.

Assuming that the samples are stored using little-endian ordering (see above), the least significant byte (LSB) or each sample is stored at the lowest address, with the most significant byte (MSB) or each sample at the next address:



This means that FSR is actually pointing to the LSB of the current sample.

So instead of writing:

```
movf    a, w           ; subtract LSB
subwf   b, f
```

we use:

```
movf    INDF, w       ; subtract LSB
subwf   adc_sum, f
```

because INDF accesses the memory location pointed to by FSR.

Next we need to subtract the MSB, which is stored in the next memory location.

Instead of writing:

```
movf    a+1, w        ; get MSB(a)
```

we use:

```
incf    FSR, f        ; advance index to access MSB of sample
movf    INDF, w
```

Note that it would be wrong to use:

```
movf    INDF+1, w     ; get MSB of sample (note: this will not work!)
```

### **This approach simply won't work!**

'INDF' is a register at address 00h, and is also mapped into bank 1, at 81h.

'INDF+1' corresponds to the register after INDF, at address 01h or 80h, depending on which bank happens to be selected, which, on the PIC16F684, is either TMR0 or OPTION\_REG. It **does not** correspond to the memory location beyond whatever address FSR is pointing to.

If you want to access the location other than the specific one that FSR is pointing to, you must first change FSR, to point to the new location, as we have done here.

The subtraction routine then becomes:

```

; (FSR points to LSB of next sample buffer entry)
banksel adc_sum
movf   INDF,w           ; subtract old sample from running total
subwf  adc_sum,f       ; subtract LSB
incf   FSR,f           ; advance index to access MSB of sample
movf   INDF,w
btfss  STATUS,C        ; check for borrow from LSB subtraction
incfsz INDF,w
subwf  adc_sum+1,f     ; subtract MSB

```

Next we must store the new sample, available in ADRESL and ADRESH, and add it to the running total.

We could write two separate routines to do this, one to store the sample and another to perform the 16-bit addition, but it's more efficient to combine these two steps into one:

```

banksel ADRESL           ; save LSB of new sample (ADC result)
movf   ADRESL,w         ; get LSB of result
decf   FSR,f           ; decrement index to access LSB of sample
movwf  INDF             ; save as LSB of new sample
banksel adc_sum         ; add LSB to running total
addwf  adc_sum,f

banksel ADRESH           ; save MSB of new sample (ADC result)
movf   ADRESH,w        ; get MSB of result
incf   FSR,f           ; advance index to access MSB of sample
movwf  INDF             ; save as MSB of new sample
banksel adc_sum
btfsc  STATUS,C        ; check for borrow from LSB addition
incfsz INDF,w
addwf  adc_sum+1,f     ; add MSB to running total

```

This is an adaptation of the 16-bit addition routine presented earlier, modified for indirect memory access in the same way as the 16-bit subtraction routine was, and interspersed with instructions for storing the new sample into the buffer.

We then need to advance FSR to point to the next sample in the buffer, ready for the next time the ADC interrupt runs. If the end of the buffer has been reached, FSR must be reset to point back to the start of the buffer, and this end of buffer condition is detected using the XOR test used above:

```

; advance index to point to next sample
incf   FSR,f           ; increment FSR
movlw  smp_buf+BUF_SIZE
xorwf  FSR,w           ; if end of buffer is reached
movlw  smp_buf
btfsc  STATUS,Z
movwf  FSR             ; reset FSR to start of buffer

```

Finally, we need to calculate the average sample value and scale it for display.

The average is simply the running total divided by the number of samples.

Since the samples are 10-bit values ranging from 0 – 1023, the average will also range up to a maximum of 1023. The average could be a fractional value, but always in the range 0 – 1023.

It can be scaled to a 0 – 99 range for display by multiplying by 100 and dividing by 1024.

Therefore, the scaled average =  $\text{adc\_sum} \times 100 / 1024 / \text{NUM\_SAMPLES}$ .

That's equivalent to calculating the scaled average as:  $\text{adc\_sum} \times 6400 / \text{NUM\_SAMPLES} / 65536$ .

Suppose we let the 24-bit variable  $\text{adc\_dec} = \text{adc\_sum} \times 6400 / \text{NUM\_SAMPLES}$ .

This is equal to the scaled average multiplied by 65536.

Therefore, the scaled average =  $\text{adc\_dec} / 65536$ .

Dividing by 65536 is easy, since  $65536 = 2^{16}$ . It's equivalent to performing 16 right-shifts, but we don't actually need to do that: if  $\text{adc\_dec}$  is a 24-bit quantity, to divide it by  $2^{16}$  all we need to do is throw away the least significant 16 bits, and the result is sitting there already – as the most significant byte of  $\text{adc\_dec}$ .

So to calculate the scaled average, all we really need to do is multiply the running total,  $\text{adc\_sum}$ , by a value equal to 6400 divided by the number of samples ( $\text{NUM\_SAMPLES}$ ).

Of course, this works best if  $\text{NUM\_SAMPLES}$  divides evenly into 6400. And the multiplication routine is made simpler if we multiply  $\text{adc\_sum}$  by an 8-bit number, meaning that  $6400/\text{NUM\_SAMPLES}$  must evaluate to less than 256. Since the 16F684 cannot accommodate more than 40 samples, these constraints mean that only two buffer sizes are suitable: 32 and 40. We'll choose  $\text{NUM\_SAMPLES} = 40$ .

It is quite straightforward to modify the 8-bit routine used in the first example to work with a 16-bit multiplier and a 24-bit result. There is however one important addition we must make. In the original routine from AN526, the content of the variable holding the multiplier is lost, as it is repeatedly right-shifted. Since we have to preserve the value of the running total,  $\text{adc\_sum}$ , from one ADC interrupt to the next, we must not lose that value by allowing the multiplication routine to operate on it directly. Instead, we must make a copy of  $\text{adc\_sum}$ , and use the copy as the multiplier, instead.

Here is the modified 16-bit multiplication routine:

```

; scale running total to 0-99 for display:
;   scaled average = adc_sum * 100 / 1024 / NUM_SAMPLES
;                   = adc_sum * 6400 / NUM_SAMPLES / 65536
;   let adc_dec (24_bit) = 6400/NUM_SAMPLES * adc_sum (16-bit)
;   then scaled average (0-99) = MSB of adc_dec
movf   adc_sum,w           ; copy adc_sum
movwf  adc_sum_x          ; to temp variable
movf   adc_sum+1,w       ; so that original is not overwritten
movwf  adc_sum_x+1
clrf   adc_dec           ; start with adc_dec = 0
clrf   adc_dec+1
clrf   adc_dec+2
movlw  .16               ; count = 16
movwf  mpy_cnt
movlw  6400/NUM_SAMPLES  ; multiplicand (6400/NUM_SAMPLES) in W
bcf    STATUS,C          ; and carry clear
l_mpy  rrf   adc_sum_x+1,f ; right shift multiplier (adc_sum_x)
      rrf   adc_sum_x,f
      btfsc STATUS,C      ; if low-order bit of multiplier was set
      addwf adc_dec+2,f   ; add multiplicand to MSB of result
      rrf   adc_dec+2,f   ; right shift 24-bit result (adc_dec)
      rrf   adc_dec+1,f
      rrf   adc_dec,f
      decfsz mpy_cnt,f    ; repeat for all 16 bits of multiplier
      goto  l_mpy

```

Note the way that the 16- or 24-bit right-shifts can be performed by repeated `rrf` instructions, as discussed earlier.

The moving average, scaled to the range 0 – 99, can now be found in the most significant byte of `adc_dec`, at location `adc_dec+2`.

All we need do now is extract the decimal digits for display, as before:

```

        ; extract digits of scaled average (in MSB of adc_dec)
        ; to display variables (to be displayed by TMR0 handler)
        movf    adc_dec+2,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

```

### Complete program

Here is the complete “light meter with smoothed decimal display” program, showing how all these parts fit together (rearranged with the main program code presented first, now that the ISR has become so much longer:

```

;*****
;
; Description:    Lesson 14, example 2
;
; Demonstrates use of indirect addressing
; to implement a simple moving average filter
;
; Displays ADC output in decimal on 2x7-segment LED display
;
; Samples analog input every 2 ms, averages last 32 samples,
; scales result to 0 - 99 and displays as 2 x dec digits
; on multiplexed 7-seg displays
;
;*****
;
; Pin assignments:
; AN0           = voltage to be measured (e.g. pot or LDR)
; RA5, RC0-5   = 7-segment display bus (common cathode)
; RA4           = tens enable (active high)
; RA1           = ones enable
;
;*****

list      p=16F684
#include   <p16F684.inc>

#include   <stdmacros-mid.inc>      ; Lookup table,off_var
                                       ; - lookup table entry
                                       ; at offset held in off_var

errorlevel -302    ; no "register not in bank 0" warnings
errorlevel -312    ; no "page or bank selection not needed" messages

radix     dec

```

```

;***** CONFIGURATION
                ; ext reset, no code or data protect, no brownout detect,
                ; no watchdog, no power-up timer, int clock with I/O,
                ; no failsafe clock monitor, two-speed start-up disabled
        _CONFIG    _MCLRE_ON & _CP_OFF & _CPD_OFF & _BOD_OFF & _WDT_OFF &
        _PWRTE_OFF & _INTOSCIO & _FCMEN_OFF & _IESO_OFF

; pin assignments
        #define sTENS_EN    sPORTA,4    ; tens enable (shadow)
        #define sONES_EN    sPORTA,1    ; ones enable

;***** CONSTANTS
        constant NUM_SAMPLES=40          ; number of samples in moving avg buffer
        constant BUF_SIZE=NUM_SAMPLES*2 ; moving avg buffer size (16 bits/sample)

;***** VARIABLE DEFINITIONS
CONTEXT    UDATA_SHR    ; variables used for context saving
cs_W      res 1
cs_STATUS res 1

SHADOW    UDATA_SHR    ; shadow registers
sPORTA    res 1        ; PORTA
sPORTC    res 1        ; PORTC

GENVAR    UDATA        ; general variables
mpx_cnt   res 1        ; multiplex counter
adc_sum   res 2        ; sum of ADC samples (LE 16-bit)
adc_sum_x res 2        ; temp copy of ADC sum (used in mult routine)
adc_dec   res 3        ; scaled ADC output (LE 24-bit, 0-99 in MSB)
mpy_cnt   res 1        ; multiplier count
                ; result in decimal (displayed by ISR):
tens      res 1        ; tens
ones      res 1        ; ones

ARRAY1    UDATA        ; array of samples for moving average
smp_buf   res BUF_SIZE ; NUM_SAMPLES x 16-bit (little endian)

;***** RESET VECTOR *****
RESET     CODE    0x0000    ; processor reset vector
                pagesel Start
                goto    Start

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

;***** Initialisation
Start    ; configure ports
                banksel TRISC    ; configure PORTC as all outputs
                clrf    TRISC
                movlw  1<<0    ; configure RA0/AN0 (only) as an input
                movwf  TRISA
                banksel ANSEL    ; make only AN0 analog
                movwf  ANSEL
                movlw  .7        ; disable comparators
                banksel CMCON0   ; CM = 7 -> both comparators off
                movwf  CMCON0

```

```

; configure timer
movlw    b'11000010'      ; configure Timer0:
; --0-----            timer mode (TOCS = 0)
; ----0---             prescaler assigned to Timer0 (PSA = 0)
; -----010           prescale = 8 (PS = 010)
banksel  OPTION_REG      ; -> increment TMR0 every 8 us
movwf    OPTION_REG      ; -> TMR0 overflows every 2.048 ms

; configure ADC
movlw    b'00010000'      Tad = 8*Tosc (ADCS = 001)
; -001-----            ; -> Tad = 2.0 us (with Fosc = 4 MHz)
banksel  ADCON1
movwf    ADCON1
movlw    b'10000001'      LSB of result in ADRESL<0> (ADFM = 1)
; 1-----             voltage reference is Vdd (VCFG = 0)
; -0-----             select channel AN0 (CHS = 000)
; ---000--             turn ADC on (ADON = 1)
; -----1
banksel  ADCON0
movwf    ADCON0

; initialise variables
banksel  tens              ; clear display variables
clrf    tens              ; to ensure that out-of-range values
clrf    ones              ; are not displayed
clrf    mpx_cnt           ; mpx_cnt = 0 -> display ones digit first
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
movlw    smp_buf+BUF_SIZE ; until end of buffer is reached
xorwf    FSR,w
btfss   STATUS,Z
goto    l_clr

; initialise sample buffer index
movlw    smp_buf          ; FSR is only used to access the sample buffer
movwf    FSR              ; so point it to the start of the buffer

; enable interrupts
banksel  PIR1              ; clear ADC interrupt flag
bcf     PIR1,ADIF
banksel  PIE1              ; enable ADC interrupt
bsf     PIE1,ADIE
movlw    1<<GIE|1<<PEIE|1<<T0IE ; enable Timer0, peripheral
movwf    INTCON           ; and global interrupts

;***** Main loop
main_loop
; copy shadow registers (updated by ISR) to ports
banksel  PORTA
movf     sPORTA,w
movwf    PORTA
movf     sPORTC,w
movwf    PORTC

```

```

; repeat forever
pagesel main_loop
goto    main_loop

;***** INTERRUPT SERVICE ROUTINE *****
ISR     CODE    0x0004
; *** Save context
movwf  cs_W           ; save W
movf   STATUS,w      ; save STATUS
movwf  cs_STATUS

; *** Identify interrupt source
btfsc  INTCON,T0IF   ; Timer0
goto   t0_int
banksel PIR1         ; ADC
btfsc  PIR1,ADIF
goto   adc_int
goto   isr_end       ; none of the above, so exit

t0_int ; *** Service Timer0 interrupt
; TMR0 overflows every 2.048 ms
; (only Timer0 interrupts are enabled)
;
bcf    INTCON,T0IF   ; clear interrupt flag

; Display current ADC result (in hex) on 3 x 7-segment displays
; mp_x_cnt determines current digit to display
;
banksel mp_x_cnt
incf   mp_x_cnt,f    ; increment mp_x_cnt for next digit
movf   mp_x_cnt,w    ; and copy to W
; determine current mp_x_cnt by successive subtraction
addlw  -1
btfsc  STATUS,Z      ; if current mp_x_cnt = 0
goto   dsp_ones      ; display ones digit
clrf   mp_x_cnt      ; else mp_x_cnt = 1, so reset to 0
goto   dsp_tens      ; and display tens digit

dsp_ones
; display ones digit (using shadow registers)
Lookup tb7segA,ones  ; lookup ones pattern for PORTA
movwf  sPORTA        ; then output it
Lookup tb7segC,ones  ; repeat for PORTC
movwf  sPORTC
bsf    sONES_EN      ; enable ones display
goto   dsp_end

dsp_tens
; display tens digit
Lookup tb7segA,tens  ; output tens digit
movwf  sPORTA
Lookup tb7segC,tens  ; repeat for PORTC
movwf  sPORTC
bsf    sTENS_EN      ; enable tens display

dsp_end

; Start next analog conversion
banksel ADCON0

```

```

    bsf      ADCON0,GO

    goto    isr_end

adc_int ; *** Service ADC interrupt
;
    bcf      PIR1,ADIF          ; clear interrupt flag

; Calculate moving average from ADC result

; store current ADC result and update running total
; (FSR points to LSB of next sample buffer entry)
    banksel adc_sum
    movf    INDF,w              ; subtract old sample from running total
    subwf   adc_sum,f          ; subtract LSB
    incf    FSR,f              ; advance index to access MSB of sample
    movf    INDF,w
    btfss   STATUS,C           ; check for borrow from LSB subtraction
    incfsz  INDF,w
    subwf   adc_sum+1,f        ; subtract MSB

    banksel ADRESL             ; save LSB of new sample (ADC result)
    movf    ADRESL,w           ; get LSB of result
    decf    FSR,f              ; decrement index to access LSB of sample
    movwf   INDF               ; save as LSB of new sample
    banksel adc_sum            ; add LSB to running total
    addwf   adc_sum,f

    banksel ADRESH             ; save MSB of new sample (ADC result)
    movf    ADRESH,w           ; get MSB of result
    incf    FSR,f              ; advance index to access MSB of sample
    movwf   INDF               ; save as MSB of new sample
    banksel adc_sum
    btfsc   STATUS,C           ; check for borrow from LSB addition
    incfsz  INDF,w
    addwf   adc_sum+1,f        ; add MSB to running total

; advance index to point to next sample
    incf    FSR,f              ; increment FSR
    movlw   smp_buf+BUF_SIZE
    xorwf   FSR,w              ; if end of buffer is reached
    movlw   smp_buf
    btfsc   STATUS,Z
    movwf   FSR                ; reset FSR to start of buffer

; scale running total to 0-99 for display:
;   scaled average = adc_sum * 100 / 1024 / NUM_SAMPLES
;                   = adc_sum * 6400 / NUM_SAMPLES / 65536
;   let adc_dec (24_bit) = 6400/NUM_SAMPLES * adc_sum (16-bit)
;   then scaled average (0-99) = MSB of adc_dec
    movf    adc_sum,w          ; copy adc_sum
    movwf   adc_sum_x          ; to temp variable
    movf    adc_sum+1,w        ; so that original is not overwritten
    movwf   adc_sum_x+1
    clrf    adc_dec            ; start with adc_dec = 0
    clrf    adc_dec+1
    clrf    adc_dec+2
    movlw   .16                ; count = 16
    movwf   mpy_cnt
    movlw   6400/NUM_SAMPLES   ; multiplicand (6400/NUM_SAMPLES) in W

```

```

l_mpy    bcf      STATUS,C           ; and carry clear
        rrf      adc_sum_x+1,f      ; right shift multiplier (adc_sum_x)
        rrf      adc_sum_x,f
        btfsc   STATUS,C           ; if low-order bit of multiplier was set
        addwf   adc_dec+2,f        ; add multiplicand to MSB of result
        rrf      adc_dec+2,f        ; right shift 24-bit result (adc_dec)
        rrf      adc_dec+1,f
        rrf      adc_dec,f
        decfsz  mpy_cnt,f          ; repeat for all 16 bits of multiplier
        goto    l_mpy

        ; extract digits of scaled average (in MSB of adc_dec)
        ; to display variables (to be displayed by TMR0 handler)
        movf    adc_dec+2,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

        goto    isr_end

isr_end ; *** Restore context then return
        movf    cs_STATUS,w       ; restore STATUS
        movwf   STATUS
        swapf   cs_W,f           ; restore W
        swapf   cs_W,w
        retfie

```

```

;***** LOOKUP TABLES *****
TABLES  CODE

```

```

; Digit to pattern lookup table for 7 segment display on port A

```

```

; RA5 = G

```

```

tb7segA movwf   PCL
        retlw   b'000000'        ; 0
        retlw   b'000000'        ; 1
        retlw   b'100000'        ; 2
        retlw   b'100000'        ; 3
        retlw   b'100000'        ; 4
        retlw   b'100000'        ; 5
        retlw   b'100000'        ; 6
        retlw   b'000000'        ; 7
        retlw   b'100000'        ; 8
        retlw   b'100000'        ; 9

```

```

; Digit to pattern lookup table for 7 segment display on port C

```

```

; RC5:0 = ABCDEF

```

```

tb7segC movwf   PCL
        retlw   b'111111'        ; 0
        retlw   b'011000'        ; 1
        retlw   b'110110'        ; 2
        retlw   b'111100'        ; 3

```

```
retlw    b'011001'    ; 4
retlw    b'101101'    ; 5
retlw    b'101111'    ; 6
retlw    b'111000'    ; 7
retlw    b'111111'    ; 8
retlw    b'111101'    ; 9
```

```
END
```

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

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

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

In the [next lesson](#), we'll introduce a 16-bit timer: Timer1.