Introduction to PIC Programming

Midrange Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 13: Analog-to-Digital Conversion

We saw in lessons 9 and 11 that a comparator can be used to respond to an analog signal being above or below a specific threshold. In other cases, the precise value of the input is important and you need to measure, or *digitise* it, so that your code can process a digital representation of the signal's value.

This lesson explains how to use the analog-to-digital converter (*ADC*), available on most midrange PICs, to read analog inputs, converting them to digital values you can operate on.

To display these values, we'll make use of the 7-segment displays introduced in lesson 12.

In summary, this lesson covers:

- Using the midrange ADC module to read analog inputs
- ADC operation in sleep mode
- ADC interrupts
- Hexadecimal output on 7-segment displays

Analog-to-Digital Converter

The analog-to-digital converter (ADC) on the 16F684 allows you to measure analog input voltages to a resolution of 10 bits. An input level of 0 V[1] will read as 0, while an input level equal to VDD[2] corresponds to the full-scale reading of 1023.

Eight analog input pins are available: AN0 to AN7, shared with digital pins RA0-2, RA4 and RC0-3. But, since there is only one ADC module, only one input can be read (or *converted*) at once.

The analog-to-digital conversion is performed through a process known as successive approximation, in which the result is determined one bit at a time. This process is driven by a conversion clock, and the time to complete each bit conversion is referred to as TAD. The complete 10-bit conversion requires 11 TAD periods (one to start the conversion process, and then one for each bit of the result).

For example, if TAD = 2 µs, the full conversion will be completed in 11 × 2 µs = 22 µs.

The conversion clock is derived from either the processor clock (FOSC) or a dedicated internal RC oscillator (FRC)[3].

[1] or VSS, if VSS is not at 0 V

[2] alternatively, a voltage applied to the VREF pin can be selected as the positive reference voltage

[3] This is separate from the HFINTOSC oscillator, available as a processor clock source, described in lesson 10.

The ADC conversion clock is selected by the ADCS<2:0> bits in the ADCON1 register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ADCON1	–	ADCS2	ADCS1	ADCS0	–	–	–	–

In the table on the right, FOSC is the processor clock speed, and TCY is the instruction cycle period.

For accurate conversions, the conversion clock must be selected such that TAD is at least 1.6 µs.

One way to achieve that would be to always select the ADC's internal RC clock, FRC.

That's ok – it's a safe option – but notice how variable FRC can be. If you select the internal RC clock, Tad could be anything up to 9 µs.

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

That means that the conversion could take almost 100 µs (11 × 9 µs = 99 µs).

Normally, you want to complete the conversions as quickly as possible, so choosing a conversion clock which makes Tad as small as possible, but not below 1.6 µs, is usually preferable. For this reason, the ADC's internal RC clock should only be used if you want to perform conversions while the device is in sleep mode, when the main processor clock (FOSC) is turned off. More on this later.

For most of the examples in this tutorial series, the processor has been clocked at 4 MHz, so for those examples, FOSC = 4 MHz and TCY = 1 µs. In that case, the best choice is ADCS = 001, giving a conversion clock of FOSC / 8 and TAD = 2 × TCY = 2 µs:

```
movlw   b'00010000'
        ; -001----          Tad = 8*Tosc (ADCS = 001)
banksel ADCON1          ;   -> Tad = 2.0 us (with Fosc = 4 MHz)
movwf   ADCON1
```

That is the closest we can come to the minimum of 1.6 µs, given a 4 MHz processor clock, but if the processor was clocked at 20 MHz, you could choose the FOSC / 32 option, for TAD = 1.6 µs.

But remember, if in doubt you can choose FRC (ADCS = 011 or 111). It won't give you the fastest conversions, and conversion times won't be consistent, but it will always work.

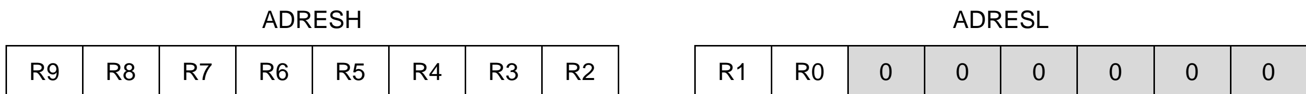
The other features of the analog-to-digital converter are controlled by the ADCON0 register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ADCON0	ADFM	VCFG	–	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	ADON

Since the result of the conversion is a 10-bit value, it won't fit into a single 8-bit register. Therefore, the result is stored in two registers: ADRESH (holding the high part of the result) and ADRESL (the low part).

There is more than one way to store a 10-bit value in two 8-bit registers, and the 16F684's ADC module provides two ways to do it, selected by the ADFM bit.

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

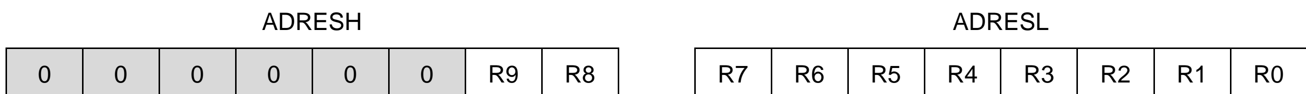


ADRESH								ADRESL							
R9	R8	R7	R6	R5	R4	R3	R2	R1	R0	0	0	0	0	0	0

The unused six bits in ADRESL read as '0'.

This format is useful when you are not concerned with the full 10-bit resolution; you can simply treat ADRESH as holding an 8-bit result, and ignore the least significant two bits held in ADRESL.

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



ADRESH								ADRESL							
0	0	0	0	0	0	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0

The unused six bits in ADRESH read as '0'.

The right-justified format is useful when you want to perform calculations using the full 10-bit result, using normal 16-bit arithmetic operations, as we will see in the next lesson.

The 10-bit result is linearly and proportionally related to the analog input being measured, with a maximum error of ±1 lsb (least significant bit), which means an accuracy of better than 0.1 %.

The minimum analog input is 0 V (or VSS, if VSS is not at 0 V), which converts to a result of 0.

The maximum or full-scale input voltage is defined as the positive reference voltage, VREF. An input equal to VREF will read as 1023.

The positive reference voltage is either VDD or an external voltage reference, depending on VCFG[4]:

> VCFG = 0 selects VREF = VDD
>
> VCFG = 1 means that VREF is taken from the VREF pin (shared with RA1)

An external voltage reference may be more accurate and stable than VDD, improving the accuracy of the ADC result. Or it may be that the input you are measuring is, by nature, some fraction of another voltage, and you need to read the fraction instead of the absolute value.

Note that, on the 16F684, VREF must be at least 2.7 V for correct ADC operation.

Before an input can be selected as an input channel for the ADC, it must first be configured as an analog input.

---

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

As we've seen before, all pins that can be configured as analog inputs will be configured as analog inputs at power-on, and you must explicitly disable the analog configuration on a pin if you wish to use it for digital I/O. This because, if a pin is configured as a digital input, it will draw excessive current if the input voltage is not at a digital "high" or "low" level, i.e. somewhere in-between. Thus, the safe, low-current option is to default to analog behaviour and to leave it up to the user program to enable digital inputs only on those pins known to be digital.

Since the default is that all analog inputs are in analog mode at power-on, we don't really have to select which inputs will be analog – it's more a case of turning off analog mode for any pins which you need to use for digital I/O[5]. It's good practice to get into the habit of turning off analog mode on any pin you're not using as an analog input (whether for a comparator or ADC); it makes it easier to avoid problems later.

Unlike the baseline architecture, analog pins in many of the midrange PICs can be independently configured as analog inputs, using the ANSEL register[6]:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ANSEL	ANS7	ANS6	ANS5	ANS4	ANS3	ANS2	ANS1	ANS0
analog input	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0
digital I/O pin	RC3	RC2	RC1	RC0	RA4	RA2	RA1	RA0

Setting a bit in ANSEL places the corresponding analog input into analog mode.

Clearing a bit in ANSEL enables the corresponding digital I/O pin.

Having selected an appropriate ADC clock rate, voltage reference, and result format, and configured the analog input pins, you must select which input channel to read, or *sample*, using the CHS<2:0> bits:

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

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

The ADON bit turns the ADC module on or off: '1' to turn it on, '0' to turn it off.

The ADC module is off (ADON = 0) by default, at power-on.

Note: To minimise power consumption, the ADC module should be turned off before entering sleep mode – unless a conversion is being performed in sleep mode.

[5] It's not strictly necessary to disable analog mode when using a pin as a digital output, but doing so avoids potential problems due to that pin reading as a '0' when other pins in the same port are updated.

[6] Midrange PICs with more than eight analog inputs may have more than one ANSEL register.

The ADC module includes a holding capacitor, CHOLD, which has to be allowed to charge to within ½ LSB (i.e. 1/2048) of the input voltage, before the conversion commences. During the conversion process, this capacitor is disconnected from the input (which may be changing) and is used as a fixed "copy" of the input being sampled, during the successive approximation process.

Therefore, before starting the conversion, you must give this capacitor enough time to charge[7]. This is referred to as the *acquisition* or *sampling* time, TACQ.

The device data sheets include formulas you can use to calculate the minimum TACQ – one of the main variables is the source impedance of the input being sampled, although temperature also plays a role.

However, assuming that the source impedance is less than the recommended maximum of 10 kΩ, an acquisition time of 10 µs is adequate for the 16F684 and most modern midrange PICs.

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

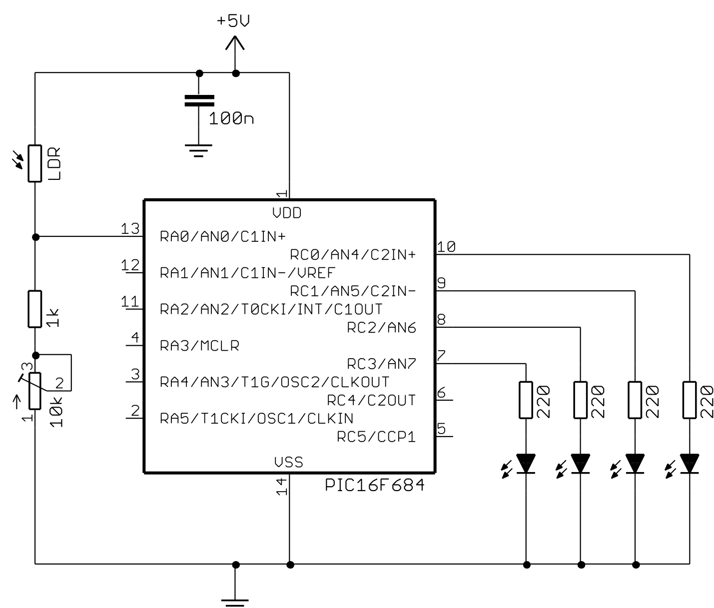
Your code then needs to wait until the GO/$\overline{\text{DONE}}$ bit has been cleared to '0', which indicates that the conversion is complete. You can then read the conversion result from the ADRESH and ADRESL registers.

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

An example will hopefully make these steps clearer.

Example 1: Binary Output

As a simple demonstration of how to use the ADC, we can use the four LEDs on the Low Pin Count Demo Board, which are connected to RC0 – RC3, as shown on the right.



The four LEDs can be used to show a 4-bit binary ADC result.

To make the display meaningful (i.e. a binary representation of the input voltage, corresponding to sixteen input levels), the top four bits of the ADC result should be copied to the four LEDs.

The bottom six bits of the ADC result are thrown away; they are not significant when we only have four output bits.

---

[7] This differs from the baseline architecture, where the sampling time is incorporated into the conversion process, being the first four TAD periods after the conversion begins

We'll configure the device as usual, with the internal RC oscillator providing the default 4 MHz clock:

```
                ; ext reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer off, 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
```

Only AN0 is being used as an analog input, so all the other pins should be configured for digital I/O, and the comparators disabled (since we are not using them):

```
        ; configure ports
        banksel TRISC           ; configure PORTC as all outputs
        clrf    TRISC
        movlw   b'00000001'     ; configure AN0 (only) as analog
        banksel ANSEL
        movwf   ANSEL
        movlw   .7              ; disable comparators
        banksel CMCON0          ;   CM = 7 -> both comparators off
        movwf   CMCON0
```

Given the 4 MHz processor clock, the best choice of ADC clock is FOSC/8:

```
        ; configure ADC
        movlw   b'00010000'
                ; -001----          Tad = 8*Tosc (ADCS = 001)
        banksel ADCON1          ;   -> Tad = 2.0 us (with Fosc = 4 MHz)
        movwf   ADCON1
```

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

We're not using a separate reference voltage, so we need to select VREF = VDD.

We also need to select AN0 as the ADC input channel, and turn the ADC on, so we have:

```
        movlw   b'00000001'
                ; 0-------          MSB of result in ADRESH<7> (ADFM = 0)
                ; -0------          voltage reference is Vdd (VCFG = 0)
                ; ---000--          select channel AN0 (CHS = 000)
                ; -------1          turn ADC on (ADON = 1)
        banksel ADCON0
        movwf   ADCON0
```

Having configured the ports and the ADC, we can begin the main loop.

It takes the form:

```
    wait required acquisition time (10 us delay)
    begin conversion
        wait until conversion is complete
    display high four bits of result on LEDs

    repeat
```

To generate the 10 µs delay needed for the acquisition time, we can re-use the 'Delay10us' macro introduced in lesson 9.

The main loop is then:

```
mainloop
        ; 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

        ; display result on 4 x LEDs
        banksel ADRESH              ; copy high four bits of result
        swapf   ADRESH,w
        banksel PORTC               ;   to low nybble of output port
        movwf   PORTC

        ; repeat forever
        goto    mainloop

        END
```

You'll see that two symbols are used for the GO/$\overline{\text{DONE}}$ bit, depending on the context: when setting the bit to start the conversion, it is referred to as "GO", but when using it as a flag to check whether the conversion is complete, it is referred to as "NOT_DONE".

Using the appropriate symbol for the context makes the intent of the code clearer, even though both symbols refer the same bit.

Finally, note the use of the 'swapf' instruction. The output bits we need to copy are in the high nybble of ADRESH, while the output LEDs (RC0 – RC3) form the low nybble of PORTC, making 'swapf' a neat solution; much shorter than using four right-shifts.

ADC Operation in Sleep Mode

To save power, it is possible to place the PIC into sleep mode, immediately after initiating the AD conversion. The device will wake when the conversion is complete; the result can then be accessed in ADRESL and ADRESH as normal.

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

If you only intend to wake the device from sleep, and do not wish to actually generate an interrupt when the AD conversion completes (see the next section), you should ensure that the GIE bit in INTCON is clear, as usual.

Before starting the conversion, you should clear the ADIF flag in the PIR1 register. This will be set when the conversion is complete, waking the device from sleep mode. If you fail to clear ADIF, the PIC will wake immediately, before the conversion is complete, and the conversion result will be incorrect.

Finally, as noted above, the ADC will only operate in sleep mode if the ADC's internal oscillator, FRC, is selected as the conversion clock source. This is because the processor clock is stopped while the device is in sleep mode – and we need the ADC to continue to operate.

Example 2: Sleep Mode

To illustrate how to use the ADC module in sleep mode, we'll modify the previous example, so that the device enters sleep immediately after the AD conversion is started.

Recall that, to operate in sleep mode, the ADC's internal oscillator must be selected as the conversion clock source, so the ADC configuration instructions become:

```
        ; configure ADC
        movlw   b'00110000'
                ; --11----          internal oscillator, Frc (ADCS = x11)
        banksel ADCON1            ;  -> operation in sleep mode possible
        movwf   ADCON1
        movlw   b'00000001'
                ; 0-------          MSB of result in ADRESH<7> (ADFM = 0)
                ; -0------          voltage reference is Vdd (VCFG = 0)
                ; ---000--          select channel AN0 (CHS = 000)
                ; -------1          turn ADC on (ADON = 1)
        banksel ADCON0
        movwf   ADCON0
```

We also need to enable the ADC interrupt (but not global interrupts), so the device will wake from sleep when the AD conversion is complete:

```
        ; enable ADC interrupt (for wake on completion)
        banksel PIE1              ; enable ADC interrupt
        bsf     PIE1,ADIE
        bsf     INTCON,PEIE       ; and peripheral interrupts
```

With the internal conversion clock selected and the ADC interrupt enabled, we can use a 'sleep' instruction in the main loop, instead of polling the GO/$\overline{\text{DONE}}$ flag – but only after clearing the ADC interrupt flag:

```
        ; sample analog input
        Delay10us                 ; wait 10 us for acquisition time
        banksel PIR1              ; clear ADC interrupt flag
        bcf     PIR1,ADIF
        banksel ADCON0            ; start conversion
        bsf     ADCON0,GO

        sleep                     ; sleep until done

        ; display result on 4 x LEDs
```

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

Complete program

Here is how these fragments fit into the complete "ADC in sleep mode" example program:

```
;************************************************************************
;                                                                       *
;   Description:    Lesson 13, example 2                                *
;                                                                       *
;   Demonstrates use of ADC in sleep mode                               *
;                                                                       *
;   Continuously samples analog input, sleeping during conversion,      *
;   then copying high 4 bits of result to 4 x LEDs                      *
```

```
;                                                                          *
;***************************************************************************
;                                                                          *
;   Pin assignments:                                                       *
;       AN0   - voltage to be measured (e.g. pot output or LDR)            *
;       RC0-3 - output LEDs                                                *
;                                                                          *
;***************************************************************************

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

    #include    <stdmacros-mid.inc>     ; Delay10us - 10 us delay

    errorlevel  -302            ; no warnings about registers not in bank 0

    radix       dec


;***** CONFIGURATION
                ; ext reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer off, 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


;***** RESET VECTOR *****************************************************
RESET   CODE    0x0000              ; processor reset vector


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

;***** Initialisation
        ; configure ports
        banksel TRISC               ; configure PORTC as all outputs
        clrf    TRISC
        movlw   b'00000001'         ; configure AN0 (only) as analog
        banksel ANSEL
        movwf   ANSEL
        movlw   .7                  ; disable comparators
        banksel CMCON0              ;   CM = 7 -> both comparators off
        movwf   CMCON0
        ; configure ADC
        movlw   b'00110000'
                ; --11----            internal oscillator, Frc (ADCS = x11)
        banksel ADCON1              ;  -> operation in sleep mode possible
        movwf   ADCON1
        movlw   b'00000001'
                ; 0-------            MSB of result in ADRESH<7> (ADFM = 0)
                ; -0------            voltage reference is Vdd (VCFG = 0)
                ; ---000--            select channel AN0 (CHS = 000)
                ; -------1            turn ADC on (ADON = 1)
        banksel ADCON0
        movwf   ADCON0
        ; enable ADC interrupt (for wake on completion)
        banksel PIE1                ; enable ADC interrupt
        bsf     PIE1,ADIE
        bsf     INTCON,PEIE         ; and peripheral interrupts
```

```
;***** Main loop
mainloop
        ; sample analog input
        Delay10us                       ; wait 10 us for acquisition time
        banksel PIR1                    ; clear ADC interrupt flag
        bcf     PIR1,ADIF
        banksel ADCON0                  ; start conversion
        bsf     ADCON0,GO

        sleep                           ; sleep until done

        ; display result on 4 x LEDs
        banksel ADRESH                  ; copy high four bits of result
        swapf   ADRESH,w
        banksel PORTC                   ;   to low nybble of output port
        movwf   PORTC

        ; repeat forever
        goto    mainloop

        END
```

ADC Interrupts

As mentioned in the section on sleep mode, above, the ADC module can be configured to generate an interrupt when the analog-to-digital conversion process is complete.

But this isn't only useful for waking the PIC from sleep; ADC interrupts are an alternative to polling.

Polling the GO/$\overline{\text{DONE}}$ flag is ok if your program has nothing else to do, but if reading analog inputs is only one of a number of tasks, polling may be a waste of valuable instruction cycles.

For example, with a 20 MHz processor clock and the FOSC/32 conversion clock selected, T_{AD} = 1.6 µs and the conversion completes in 11 × 1.6 µs = 17.6 µs = 88 instruction cycles. That is, at higher processor clock rates, there can be enough time to execute 88 instructions, or more, while the AD conversion completes.

Instead of doing nothing but poll the GO/$\overline{\text{DONE}}$ flag, it is possible to spend at least some of that conversion time doing something more useful, by enabling the ADC interrupt, initiating the AD conversion, and then going on with other tasks until the conversion is complete. The ADC interrupt will then be triggered, and your interrupt service routine (ISR) can immediately read the result.

One possible approach is to have the ISR set a flag, which is polled within the main program loop. When the main loop detects that the ISR has handled the ADC interrupt, a new conversion is initiated, after waiting the required acquisition delay. But although this method features in a Microchip application note, it is really no improvement over polling GO/$\overline{\text{DONE}}$ directly; most of the processor time is still spent polling a flag.

If your program has to perform other tasks while also reading one or more analog inputs, it doesn't really make sense to initiate a new AD conversion as soon as the last one completes – that approach leaves no time to do anything else. It is often more appropriate to perform the AD conversions at a steady rate – more slowly than the ADC module is actually capable of. An ideal way to do that is to use a timer-based interrupt (see lesson 6) to initiate the conversions, for example once every millisecond. Not only does this mean that other tasks can be completed in between AD conversions, it also means that there is no need for any additional, explicit, acquisition delay before initiating each conversion – because we know that, with the conversions spaced apart, ample acquisition time has elapsed between each conversion.

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

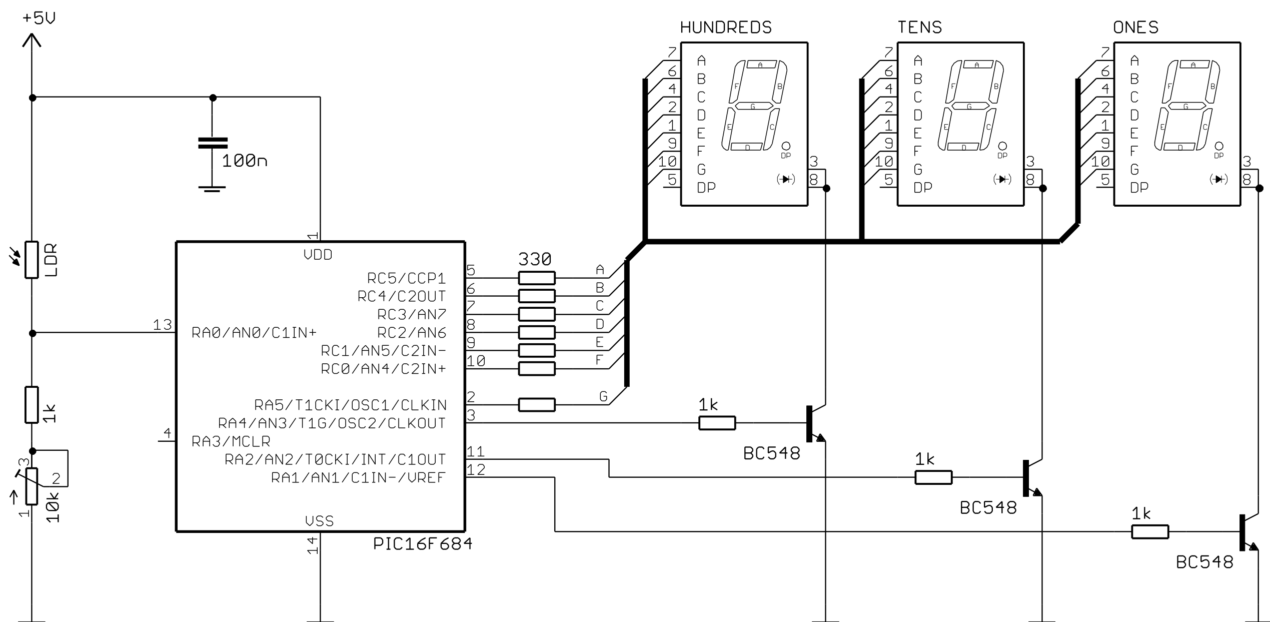
other interrupts, or conditions which the main loop is polling for, can be responded to as quickly as possible. Your program will be more responsive overall, if you can minimise the time spent within ISRs.

If a timer-based interrupt is already being used for display multiplexing (see lesson 12), it makes sense to use this same time base to initiate the AD conversions – especially if the result of the conversion is being output on the multiplexed display; the sample rate can then be synchronised with the display rate.

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

Example 3: Hexadecimal Output

A binary LED display, as in example 1, is not a very useful form of output. To create a more human-readable output, we can modify the 7-segment LED circuit from lesson 12, as shown below:



To display the hexadecimal value, we can adapt the multiplexed 7-segment display code from lesson 12.

First, to drive the displays using RC0-RC5 and RA1, RA2 and RA4, we need to disable the comparators, and configure only AN0 as an analog input:

```
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
```

Note that, in addition to setting ANSEL<0> to put the RA0/AN0 pin into analog mode, it is also necessary to set TRISA<0>, to make that pin an input. This is different from the baseline architecture, where the analog setting on a pin overrides the TRIS bit; in midrange PICs, you have to set them both.

We also need to set up the ADC:

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

In this case, it is easiest to access the 10-bit result, for display as three hexadecimal digits, if it is right-justified (ADFM = 1). The highest hex digit of the result will appear as the lower two bits of ADRESH, and the bottom two hex digits will be the lower eight bits of the result, in ADRESL.

The three-digit display can be maintained in the same way as was done in lesson 12, with Timer0 configured to generate an interrupt approximately every 2 ms, and the ISR displaying each digit in turn, using a "multiplex counter" variable, '`mpx_cnt`', to keep track of which digit to display. The lookup tables have to be extended to include 7-segment representations of the letters 'A' to 'F', but the lookup code can remain the same – re can re-use the '`Lookup`' macro developed in lesson 12.

Although the ISR could read the ADC result directly, the code is more flexible if we continue to store the digits to be displayed in a set of variables, which can be updated elsewhere. In lesson 12, these variables were named '`mins`', '`tens`' and '`ones`', but since this is not a time display any more, it makes more sense to name the variables the more generic '`hundreds`', '`tens`' and '`ones`'. Of course, these are hexadecimal "hundreds" and "tens" (0x100s and 0x10s, not 100s and 10s)... We can then update these variables in the main loop, using the current ADC reading, which will then be displayed by the ISR.

So in the main loop, we first perform an analog-to-digital conversion:

```
        ; 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
```

and then extract the hexadecimal digits from the result, to be displayed:

```
        ; copy result to variables for ISR to display
        banksel ADRESL
        movf    ADRESL,w            ; get "ones" digit
        andlw   0x0F                ;   from low nybble of ADRESL
        banksel ones
        movwf   ones
        banksel ADRESL              ; get "tens" digit
        swapf   ADRESL,w            ;   from high nybble of ADRESL
        andlw   0x0F
        banksel tens
        movwf   tens
        banksel ADRESH              ; get "hundreds" digit
        movf    ADRESH,w            ;   from ADRESH
        banksel hundreds
        movwf   hundreds
```

This is similar to the way in which digits were extracted from BSD representation, in lesson 12.

Note that there is no need to mask the result copied from ADRESH, because the unused upper six bits of ADRESH are guaranteed to read a zero – so, in the right-justified result format, ADRESH will only ever contain a number between 0 and 3.

Complete program

Here is the complete "hexadecimal light meter" (or potentiometer hex readout, if you're not using an LDR), so that you can see how the various program fragments fit together:

```
;************************************************************************
;   Description:    Lesson 13, example 3                                *
;                                                                       *
;   Displays ADC output in hexadecimal on 7-segment LED displays        *
;                                                                       *
;   Continuously samples analog input,                                  *
;   displaying result as 3 x hex digits on multiplexed 7-seg displays   *
;                                                                       *
;************************************************************************
;   Pin assignments:                                                    *
;       AN0         = voltage to be measured (e.g. pot or LDR)          *
;       RA5, RC0-5  = 7-segment display bus (common cathode)            *
;       RA4         = "hundreds" enable (active high)                   *
;       RA2         = "tens" enable                                     *
;       RA1         = "ones" enable                                     *
;                                                                       *
;************************************************************************

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

; pin assignments
    #define sHUNDREDS   sPORTA,4    ; "hundreds" enable (shadow)
    #define sTENS       sPORTA,2    ; "tens" enable
    #define sONES       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
hundreds    res 1               ; current ADC result (in hex): "hundreds"
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 current ADC result (in hex) on 3 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
        addlw   -1
        btfsc   STATUS,Z            ; if current mpx_cnt = 1
        goto    dsp_tens            ;   display "tens" digit
        clrf    mpx_cnt             ; else mpx_cnt = 2, so reset to 0
        goto    dsp_hundreds        ;   and display "hundreds" 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               ; 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               ; enable "tens" display
```

```
        goto    dsp_end

dsp_hundreds
        ; display "hundreds" digit
        Lookup  tb7segA,hundreds    ; output "hundreds" digit
        movwf   sPORTA
        Lookup  tb7segC,hundreds
        movwf   sPORTC
        bsf     sHUNDREDS           ; enable "hundreds" display
dsp_end

        ; copy shadow regs to ports
        banksel PORTA
        movf    sPORTA,w
        movwf   PORTA
        movf    sPORTC,w
        movwf   PORTC

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 (T0CS = 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'
                ; -001----          Tad = 8*Tosc (ADCS = 001)
        banksel ADCON1          ;   -> Tad = 2.0 us (with Fosc = 4 MHz)
        movwf   ADCON1
        movlw   b'10000001'
                ; 1-------          LSB of result in ADRESL<0> (ADFM = 1)
                ; -0------          voltage reference is Vdd (VCFG = 0)
                ; ---000--          select channel AN0 (CHS = 000)
                ; -------1          turn ADC on (ADON = 1)
        banksel ADCON0
        movwf   ADCON0
```

```
        ; initialise variables
        banksel hundreds        ; clear display variables
        clrf    hundreds        ;   to ensure that out-of-range values
        clrf    tens            ;   are not displayed
        clrf    ones
        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

        ; copy result to variables for ISR to display
        banksel ADRESL
        movf    ADRESL,w            ; get "ones" digit
        andlw   0x0F                ;   from low nybble of ADRESL
        banksel ones
        movwf   ones
        banksel ADRESL              ; get "tens" digit
        swapf   ADRESL,w            ;   from high nybble of ADRESL
        andlw   0x0F
        banksel tens
        movwf   tens
        banksel ADRESH              ; get "hundreds" digit
        movf    ADRESH,w            ;   from ADRESH
        banksel hundreds
        movwf   hundreds

        ; 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'000000'       ; 7
        retlw   b'100000'       ; 8
        retlw   b'100000'       ; 9
        retlw   b'100000'       ; A
        retlw   b'100000'       ; b
        retlw   b'000000'       ; C
```

```
        retlw   b'100000'       ; d
        retlw   b'100000'       ; E
        retlw   b'100000'       ; F

; 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
        retlw   b'111011'       ; A
        retlw   b'001111'       ; b
        retlw   b'100111'       ; C
        retlw   b'011110'       ; d
        retlw   b'100111'       ; E
        retlw   b'100011'       ; F


        END
```

Of course, most people are more comfortable with a decimal output, perhaps scaled to 0 - 99, instead of hexadecimal 000 - 3FF.

And you'll find, if you build this as a light meter, using an LDR, that although the output is quite stable when lit by daylight, the least significant digits jitter quite badly when the LDR is lit by incandescent and, in particular, fluorescent lighting. This is because these lights flicker at 50 or 60 Hz (depending on where you live), too fast for your eyes to detect, but not too fast for this light meter to react to, since it is sampling and updating the display 244 times per second.

So some obvious improvements to the design would be to scale and display the output as 0-99, decimal, and to smooth or filter noise, such as that caused by fluorescent lighting.

But those are topics for lesson 14. First, we'll see how to use ADC interrupts.

Example 4: ADC Interrupts

As discussed, it is often sensible to use a timer interrupt to initiate regular analog-to-digital conversions, especially if one is already used to perform "background" tasks such as display multiplexing. In that case, instead of waiting within the timer interrupt service routine for the AD conversion to complete, it is usually better to enable ADC interrupts, and allow an ADC interrupt handler to process the ADC result.

To illustrate this, we'll modify the previous example to use ADC interrupts.

Firstly, we need to enable ADC interrupts, as we did when configuring the ADC for wake-up from sleep:

```
        ; 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
```

This time, because the GIE bit is also being set, an interrupt will be triggered as soon as the ADIF flag is set, which happens whenever a conversion completes.

To avoid premature triggering, it is a good idea to clear all enabled interrupt flags, before enabling interrupts. In this case, the ADIF flag is being cleared to ensure that the ADC interrupt will not be triggered until an analog-to-digital conversion had completed – although it wouldn't really matter in this example if it did run prematurely. Clearing the interrupt flags before enabling interrupts is simply a good habit to get into, because most of the time you don't want an ISR to run before you're ready for it.

Similarly, nothing bad will happen if Timer0 interrupt triggers immediately – but in any case the T0IF flag is implicitly cleared when INTCON is initialised.

That is, we could have written:

```
movlw   1<<GIE|1<<PEIE|1<<T0IE|0<<T0IF ; clear T0IF and enable Timer0,
movwf   INTCON                         ;  peripheral and global interrupts
```

but that is actually the same as above. Perhaps this longer expression is clearer if it is important to note explicitly that T0IF is being cleared, but since it really makes no difference in this example, why bother?

Next we need to make the Timer0 interrupt initiate the AD conversions.

Because there are three digits, with a different digit being displayed each time the Timer0 ISR runs, the complete display is only refreshed every third interrupt. Since the value being displayed is read from the ADC, there is no point performing the conversions any faster than the display is being refreshed, i.e. every third Timer0 interrupt.

So, instead of initiating the AD conversion every time the Timer0 ISR runs, it makes more sense to initiate the conversion after all three digits have been updated.

That means adding the code to begin the conversion, within the Timer0 ISR, just after the "hundreds" digit has been displayed:

```
; display "hundreds" digit
Lookup  tb7segA,hundreds    ; output "hundreds" digit
movwf   sPORTA
Lookup  tb7segC,hundreds
movwf   sPORTC
bsf     sHUNDREDS           ; enable "hundreds" display

; get next analog sample
banksel ADCON0              ; start conversion
bsf     ADCON0,GO
```

Since we now have two interrupt sources, we also need to add some code to the start of the ISR, to jump to the appropriate interrupt handler, depending on which interrupt flag has been set:

```
; *** 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
```

The ADC interrupt handler then consists of the code to read the conversion result, and write it into the display variables. It is exactly the same code as in the polled example, above, except (as in any interrupt handler) it begins by clearing the interrupt flag, to show that this interrupt has been processed:

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

        ; copy ADC result to display variables
        ;   (to be displayed by TMR0 handler)
        banksel ADRESL
        movf    ADRESL,w            ; get "ones" digit
        andlw   0x0F                ;   from low nybble of ADRESL
        banksel ones
        movwf   ones
        banksel ADRESL              ; get "tens" digit
        swapf   ADRESL,w            ;   from high nybble of ADRESL
        andlw   0x0F
        banksel tens
        movwf   tens
        banksel ADRESH              ; get "hundreds" digit
        movf    ADRESH,w            ;   from ADRESH
        banksel hundreds
        movwf   hundreds

        goto    isr_end
```

With the AD conversion being initiated by the Timer0 interrupt handler, and the conversion result being processed by the ADC interrupt hander, there is literally nothing left for the "main loop" to do:

```
main_loop
        ; do nothing
        goto    main_loop
```

And that, of course, is the point of using interrupts – all the display and regular input sampling activity runs in the background, leaving the main loop to perform whatever "foreground" tasks remain. In a real example, there would be some other inputs to respond to...

Complete interrupt service routine

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

```
;***** INTERRUPT SERVICE ROUTINE ***************************************
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
        ;   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
        addlw   -1
        btfsc   STATUS,Z            ; if current mpx_cnt = 1
        goto    dsp_tens            ;   display "tens" digit
        clrf    mpx_cnt             ; else mpx_cnt = 2, so reset to 0
        goto    dsp_hundreds        ;   and display "hundreds" 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               ; 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               ; enable "tens" display
        goto    dsp_end

dsp_hundreds
        ; display "hundreds" digit
        Lookup  tb7segA,hundreds    ; output "hundreds" digit
        movwf   sPORTA
        Lookup  tb7segC,hundreds
        movwf   sPORTC
        bsf     sHUNDREDS           ; enable "hundreds" display

        ; get next analog sample
        banksel ADCON0              ; start conversion
        bsf     ADCON0,GO

dsp_end
        ; copy shadow regs to ports
        banksel PORTA
        movf    sPORTA,w
        movwf   PORTA
        movf    sPORTC,w
        movwf   PORTC

        goto    isr_end
```

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

        ; copy ADC result to display variables
        ;   (to be displayed by TMR0 handler)
        banksel ADRESL
        movf    ADRESL,w            ; get "ones" digit
        andlw   0x0F                ;   from low nybble of ADRESL
        banksel ones
        movwf   ones
        banksel ADRESL              ; get "tens" digit
        swapf   ADRESL,w            ;   from high nybble of ADRESL
        andlw   0x0F
        banksel tens
        movwf   tens
        banksel ADRESH              ; get "hundreds" digit
        movf    ADRESH,w            ;   from ADRESH
        banksel hundreds
        movwf   hundreds

        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
```

Example 5: Measuring Supply Voltage

Baseline lesson 10, which covered the 8-bit ADC module in the 16F506 and other baseline PICs, showed how the device's internal 0.6 V fixed voltage reference could be sampled by the ADC to infer changes in the power supply voltage, VDD (actually VDD – VSS, but to keep this simple we'll assume VSS = 0V).

Assuming that VDD = 5.0 V and VSS = 0 V, the 0.6 V reference should read as:

$$0.6V \div 5.0V \times 255 = 30$$

Now if VDD was to fall to, say, 3.5 V, the 0.6 V reference would then read as:

$$0.6V \div 3.5V \times 255 = 43$$

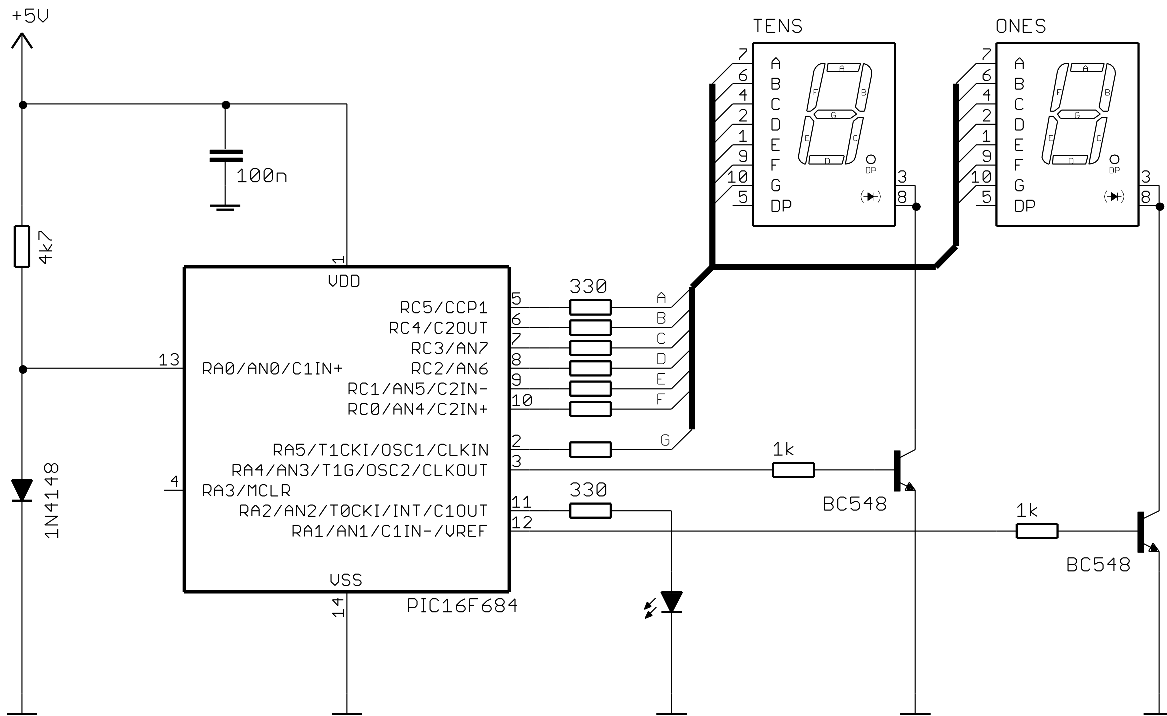
As VDD falls, reading the 0.6 V reference gives a larger ADC result, since it remains constant as VDD decreases.

So to check for the power supply voltage falling too low, the value returned by sampling the 0.6 V reference can be compared with a threshold, allowing a warning to be displayed, or perhaps the device could be shut down cleanly before power falls too low.

The PIC16F684 does not include a fixed, or absolute, voltage reference, so this technique cannot be applied directly. However, more advanced midrange devices, such as the 16F887, do provide an absolute voltage reference, selectable as an ADC input channel, in much the same way as in the 16F506.

So it is worth seeing how this technique can be applied, using an external fixed reference on the 16F684. The code change, if you are using a device with an internal reference, is trivial – simply select the internal reference as the ADC input channel, instead of the external input pin (AN0) used here. The rest of the code, and the concepts, are the same. You will of course need to consult the device's data sheet, but you should be doing that anyway.

There are a number of ways to implement an external fixed reference. You could use a precision voltage reference, or a voltage regulator. Less expensive is to use a Zener diode (with a current-limiting resistor). But for this application, it is perfectly adequate to use the approximately 0.6 V forward voltage drop across a normal silicon diode, as shown in the circuit below:



It is true that this forward voltage is not really "fixed"; it drops as the diode current, and hence the power supply voltage falls. However, forward voltage across the diode (appearing on AN0) falls much more slowly than the supply voltage does, meaning that, as the supply voltage falls, the voltage on AN0 rises as a proportion of VDD, in the same way as described for the absolute reference, above.

So, if we configure the ADC with VDD as the positive reference, we can infer the value of VDD by sampling AN0, and turn on the LED on RA2 when the value read on AN0 goes above a threshold, to indicate a low power supply voltage.

Note that, because of the limited number of pins available on the 16F684, to accommodate the warning LED we need to remove one of the 7-segment displays. But that's ok – two digits is enough to give a good indication of the value of the voltage reference on AN0, as a fraction of VDD.

To implement this, the code from example 3 can be used with very little modification (to make the code easier to follow, we won't use the ADC interrupt). We only need to reduce the number of displays from three to two, and add some code to turn on the warning LED when the value read on AN0 is above the threshold.

To make the code easier to maintain, we can define the voltage threshold as a constant:

```
constant MINVDD=3500            ; Minimum Vdd (in mV)
constant VRMAX=255*600/MINVDD   ; Threshold for 0.6 V ref measurement
```

Note that, since MPASM only supports integer expressions, “MINVDD” has to be expressed in millivolts instead of volts (so that fractions of a volt can be specified).

The initialisation code can remain the same as in example 3, except that, since we are only displaying eight bits of the ADC result, it makes more sense to left-justify the result, using ADRESH as our 8-bit result register, and ignoring the two LSBs in ADRESL, so the ADC configuration becomes:

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

Since there is no longer a “hundreds” digit, we can drop the ‘hundreds’ variable, and also the “dsp_hundreds” display routine in the ISR.

The digit selection code becomes:

```
; Display current ADC result (in hex) 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
```

With only two digits, this is a bit clunky. Instead of incrementing the ‘mpx_cnt’ variable and then resetting it to 0 after it reaches 1, it would be simpler and neater to simply toggle a single bit. But since we’re modifying existing code instead of writing new code from scratch, it’s easier to leave it like this, and we know it works – even if it’s not elegant. That’s a common experience when adapting old code to a new purpose – it might be “cleaner” to start again, but it often makes sense to build on what’s tried and true.

In the main loop, after sampling the 0.6 V reference input, we can test for VDD too low by comparing the conversion result (ADRESH, since we are only considering eight bits) with the threshold (VRMAX):

```
; test for low Vdd (measured 0.6 V > threshold)
movlw   VRMAX
banksel ADRESH
subwf   ADRESH,w            ; if ADRESH > VRMAX
btfsc   STATUS,C
bsf     sWARN               ;   turn on warning LED
```

There's a big problem with this approach though.

We're only updating the shadow register, which is only written to the port (making the change visible) after the digits have been displayed, within the ISR. That update, to output the pattern for a given digit, will overwrite whatever changes we've made to shadow registers in the main loop. So – we'll never see the warning LED go on.

The solution is to move the port refresh (copying the current contents of the shadow registers to the ports) from the ISR to the main loop.

This means that, when the shadow bit corresponding to the warning LED is set in the main loop, that change will be copied to the port and we'll see the LED go on.

There's still a problem though. When the Timer0 ISR runs, the digits are updated and the new pattern looked up for PORTA will overwrite whatever the warning LED on RA2 is set to; the LED will go out.

Ideally the lookup code wouldn't write a whole new value to PORTA; it should update only the bits corresponding to the pins driving the 7-segment displays, and preserve the values of any other bits, such as RA2. In practice though, it's not a significant problem; the main loop cycles much faster than the timer interrupt tick rate, so in practice the 7-segment display update only turns off the warning LED for a short time before the code in the main loop turns it on again. The LED flashes on and off, but too quickly to notice.

Complete program

Although the changes from example 3 are not extensive, the structure of the program has changed enough to make it worth including the full listing here:

```
;************************************************************************
;   Description:    Lesson 13, example 5                                *
;                                                                       *
;   Demonstrates use of fixed reference with ADC to test supply voltage *
;                                                                       *
;   Continuously samples external 0.6V reference,                       *
;   displaying result as 2 x hex digits on multiplexed 7-seg displays   *
;   Turns on warning LED if measurement > threshold                     *
;                                                                       *
;************************************************************************
;   Pin assignments:                                                    *
;       AN0         = external 0.6 V reference                          *
;       RA5, RC0-5  = 7-segment display bus (common cathode)            *
;       RA4         = "tens" enable (active high)                       *
;       RA1         = "ones" enable                                     *
;       RA2         = warning LED                                       *
;************************************************************************

    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
                ; int reset, no code or data protect, no brownout detect,
                ; no watchdog, no power-up timer, int clock with I/O,
```

```
                ; no failsafe clock monitor, two-speed start-up disabled
    __CONFIG    _MCLRE_OFF & _CP_OFF & _CPD_OFF & _BOD_OFF & _WDT_OFF &
_PWRTE_OFF & _INTOSCIO & _FCMEN_OFF & _IESO_OFF

; pin assignments
    #define sTENS       sPORTA,4    ; "tens" enable (shadow)
    #define sONES       sPORTA,1    ; "ones" enable
    #define sWARN       sPORTA,2    ; warning LED


;***** CONSTANTS
    constant MINVDD=3500            ; Minimum Vdd (in mV)
    constant VRMAX=255*600/MINVDD   ; Threshold for 0.6 V ref measurement


;***** 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
tens        res 1               ; current ADC result (in hex): "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 current ADC result (in hex) 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               ; 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               ; 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 (T0CS = 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'
                ; -001----          Tad = 8*Tosc (ADCS = 001)
        banksel ADCON1          ;   -> Tad = 2.0 us (with Fosc = 4 MHz)
        movwf   ADCON1
        movlw   b'00000001'
                ; 0-------          MSB of result in ADRESH<7> (ADFM = 0)
                ; -0------          voltage reference is Vdd (VCFG = 0)
                ; ---000--          select channel AN0 (CHS = 000)
                ; -------1          turn ADC on (ADON = 1)
        banksel ADCON0
        movwf   ADCON0
```

```
        ; 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 0.6 V reference on AN0
        Delay10us                   ; wait 10 us for acquisition time
        banksel ADCON0              ; start conversion
        bsf     ADCON0,GO
waitadc btfsc   ADCON0,NOT_DONE     ; wait until done
        goto    waitadc

        ; test for low Vdd (measured 0.6 V > threshold)
        movlw   VRMAX
        banksel ADRESH
        subwf   ADRESH,w            ; if ADRESH > VRMAX
        btfsc   STATUS,C
        bsf     sWARN               ;   turn on warning LED

        ; copy result to variables for ISR to display
        banksel ADRESH
        movf    ADRESH,w            ; get "ones" digit
        andlw   0x0F                ;   from low nybble of ADRESH
        banksel ones
        movwf   ones
        banksel ADRESH              ; get "tens" digit
        swapf   ADRESH,w            ;   from high nybble of ADRESH
        andlw   0x0F
        banksel tens
        movwf   tens

        ; copy shadow registers to ports
        banksel PORTA
        movf    sPORTA,w
        movwf   PORTA
        movf    sPORTC,w
        movwf   PORTC

        ; 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
```

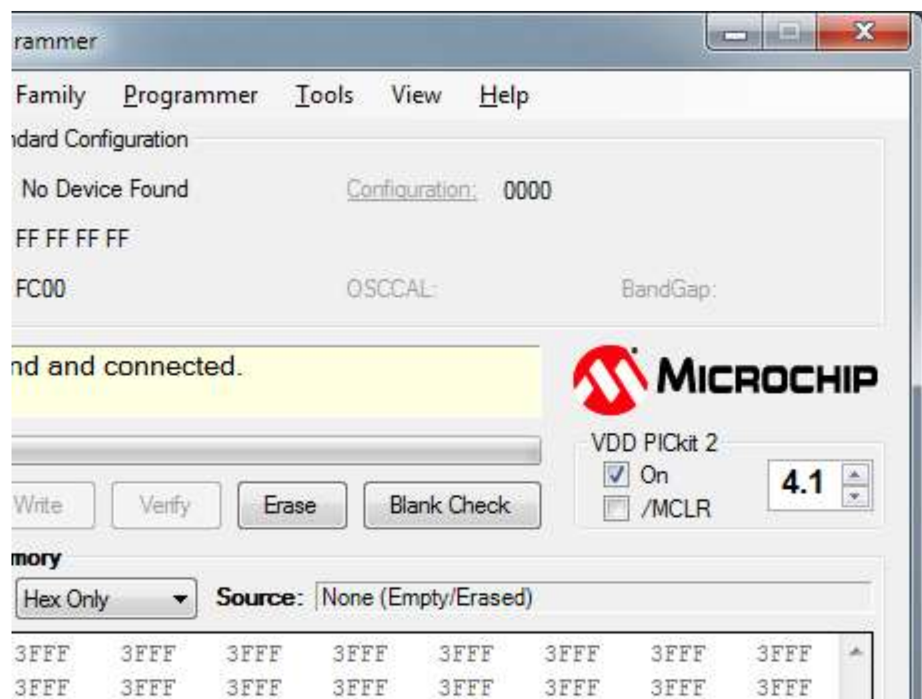
```
        retlw   b'100000'       ; 4
        retlw   b'100000'       ; 5
        retlw   b'100000'       ; 6
        retlw   b'000000'       ; 7
        retlw   b'100000'       ; 8
        retlw   b'100000'       ; 9
        retlw   b'100000'       ; A
        retlw   b'100000'       ; b
        retlw   b'000000'       ; C
        retlw   b'100000'       ; d
        retlw   b'100000'       ; E
        retlw   b'100000'       ; F

; 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
        retlw   b'111011'       ; A
        retlw   b'001111'       ; b
        retlw   b'100111'       ; C
        retlw   b'011110'       ; d
        retlw   b'100111'       ; E
        retlw   b'100011'       ; F


        END
```

To test this application, you need to be able to vary VDD. If you are using a PICkit 2 to power your circuit, you can use the PICkit 2 application (as shown) to vary VDD, while the circuit is powered.

You should first exit MPLAB, so that you don't have two applications trying to control the PICkit 2 at once. The PICkit 2 application may not recognise the PIC, because of the diode on RA0 (to program the PIC, you first need to remove the diode, then put it back, after programming, to test).



That doesn't matter; we only need to apply a voltage, so click 'On' in the 'VDD PICkit 2' box.

Your circuit should now be powered on, and, assuming the supply voltage is 5.0V, the display should show "1E" (hexadecimal), or something close to that.

You can now start to decrease VDD, by clicking on the down arrow next to the voltage display, 0.1 V at a time. When you get to around 3.5 V, the display should read "2b", and the warning LED should light.

As mentioned earlier, the light meter project would be more useful if the output was converted to a range of 0-99 and displayed in decimal, and if the results were filtered to smooth out short term fluctuations.

The next lesson will demonstrate how to perform some simple arithmetic operations, including calculating a moving average and working with arrays, to implement these suggested improvements.