Introduction to PIC Programming

Baseline Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 10: Analog-to-Digital Conversion

We saw in the last lesson how a comparator can be used to respond to an analog signal being above or below a specific threshold. In other cases, the 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 a number of baseline 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 used in lesson 8.

In summary, this lesson covers:

- Using an ADC module to read analog inputs
- Hexadecimal output on 7-segment displays

Analog-to-Digital Converter

The analog-to-digital converter (ADC) on the 16F506 allows you to measure analog input voltages to a resolution of 8 bits. An input of 0 V (or VSS, if VSS is not at 0 V) will read as 0, while an input of VDD corresponds to the full-scale reading of 255.

Three analog input pins are available: AN0, AN1 and AN2. But, since there is only one ADC module, only one input can be read (or *converted*) at once.

The analog-to-digital converter is controlled by the ADCON0 register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ADCON0	ANS1	ANS0	ADCS1	ADCS0	CHS1	CHS0	GO/$\overline{\text{DONE}}$	ADON

Before a pin can be selected as an input channel for the ADC, it must first be configured as an analog input, using the ANS<1:0> bits:

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

Note that pins cannot be independently configured as analog inputs.

If only one analog input is needed, it has to be AN2. If any analog inputs are configured, AN2 must be one of them.

If only two analog inputs are needed, they must be AN0 and AN2.

By default, following power-on, ANS<1:0> is set to '11', configuring AN0, AN1 and AN2 as analog inputs.

This is the default behaviour for all PICs; 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. The reason for this is that, 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.

All of the analog inputs can be disabled (enabling digital I/O on the RB0, RB1 and RB2 pins) by clearing ADCON0, which clears ANS<1:0> to '00'.

The ADON bit turns the ADC module on or off: '1' to turn it on, '0' to turn it off. The ADC module is turned on (ADON = 1) by default, at power-on.

Note: To minimise power consumption, the ADC module should be turned off before entering sleep mode.

Hence, clearing ADCON0 will also clear ADON to '0', disabling the ADC module, conserving power.

However, disabling the ADC module is not enough to disable the analog inputs; the ANS<1:0> bits must be used to configure analog pins for digital I/O, regardless of the value of ADON.

The analog-to-digital conversion process is driven by a clock, which is derived from either the processor clock (FOSC) or the internal RC oscillator (INTOSC). For accurate conversions, the ADC clock rate must be selected such that the ADC conversion clock period, TAD, be between 500 ns and 50 µs.

The ADC conversion clock is selected by the ADCS<1:0> bits:

ADCS<1:0>	ADC conversion clock
00	FOSC/16
01	FOSC/8
10	FOSC/4
11	INTOSC/4

Note that, if the internal RC oscillator is being used as the processor clock, the INTOSC/4 and FOSC/4 options are the same.

But whether you are using a high-speed 20 MHz crystal, a low-power 32 kHz watch crystal, or a low-speed external RC oscillator, the INTOSC/4 ADC clock option (ADCS<1:0> = '11') will always work, giving accurate conversions.

INTOSC/4 is always a safe choice.

Each analog-to-digital conversion requires 13 TAD periods to complete.

If you are using the INTOSC/4 ADC clock option, and the internal RC oscillator is running at 4 MHz, INTOSC/4 = 1 MHz and TAD = 1 µs. Each conversion will then take a total of 13 µs.

If the internal RC oscillator is running at 8 MHz, TAD = 500 ns (the shortest period allowed, making this the fastest conversion rate possible), each conversion will take 13 × 500 ns = 6.5 µs.

Having turned on the ADC, selected the ADC conversion clock, and configured the analog input pins, the next step is to select an input (or *channel*) to be converted.

CHS<1:0> selects the ADC channel:

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

Note that, in addition to the three analog input pins, AN0 to AN2, the 0.6V internal voltage reference can be selected as an ADC input channel.

Why measure the 0.6V absolute reference voltage, if it never changes?

That's the point – it never changes (except for some drift with temperature). But if you're not using a regulated power supply (e.g. running direct from batteries), VDD will vary as the power supply changes. Since the full scale range of the ADC is VSS to VDD, your analog measurements are a fraction of VDD and will vary as VDD varies. By regularly measuring (*sampling*) the 0.6 V absolute reference, it is possible to achieve greater measurement accuracy by correcting for changes in VDD. It is also possible to use the 0.6 V reference to indirectly measure VDD, and hence battery voltage, as we'll see in a later example.

Having set up the ADC and selected an input channel to be sampled, the final step is to begin the conversion, by setting the GO/$\overline{\text{DONE}}$ bit 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 ADRES register.

You should copy the result from ADRES before beginning the next conversion, so that it isn't overwritten during the conversion process[1].

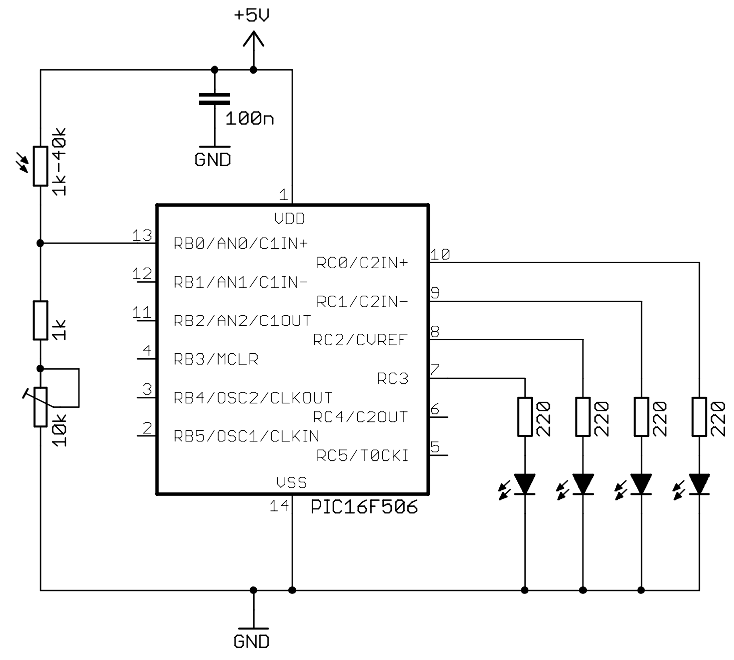
Also for best results, the source impedance of the input being sampled should be no more than 10 kΩ.

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 (in ADRES) should be copied to the four LEDs.



[1] The result actually remains in ADRES for the first four TAD periods after the conversion begins. This is the sampling period, and for best results the input signal should not be changing rapidly during this period.

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

To use RC0 and RC1 as digital outputs, we need to disable the C2IN+ and C2IN- inputs, which can be done by disabling comparator 2:

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

This also disables C2OUT, making RC4 available for digital I/O, even though it isn't used in this example.

To use RC2 as a digital output, CVREF output has to be disabled. By default, on power-up, the voltage reference module is disabled, including the CVREF output. But it doesn't hurt to explicitly disable it as part of your initialisation code:

```
        clrf    VRCON           ; disable CVref (RC2 usable)
```

AN0 has to be configured as an analog input. It's not possible to configure AN0 as an analog input without AN2, so for the minimal number of analog inputs, set ANS<1:0> = '10' (AN0 and AN2 analog).

It makes sense to choose INTOSC/4 as the conversion clock (ADCS<1:0> = '11'), as a safe default, although in fact any of the ADC clock settings will work when the processor clock is 4 MHz or 8 MHz.

AN0 has to be selected as the ADC input channel: CHS<1:0> = '00'.

And of course the ADC module has to be enabled: ADON = '1'.

So we have:

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

Alternatively the '`movlw`' could be written as:

```
        movlw   b'10'<<ANS0|b'11'<<ADCS0|b'00'<<CHS0|1<<ADON
```

But that's unwieldy, and if anything harder to understand.

Having configured the LED outputs (PORTC) and the ADC, the main loop is quite straightforward:

```
mainloop
        ; sample analog input
        bsf     ADCON0,GO       ; start conversion
waitadc btfsc   ADCON0,NOT_DONE ; wait until done
        goto    waitadc

        ; display result on 4 x LEDs
        swapf   ADRES,w         ; copy high nybble of result
        movwf   LEDS            ;   to low nybble of output port (LEDs)

        ; repeat forever
        goto    mainloop
```

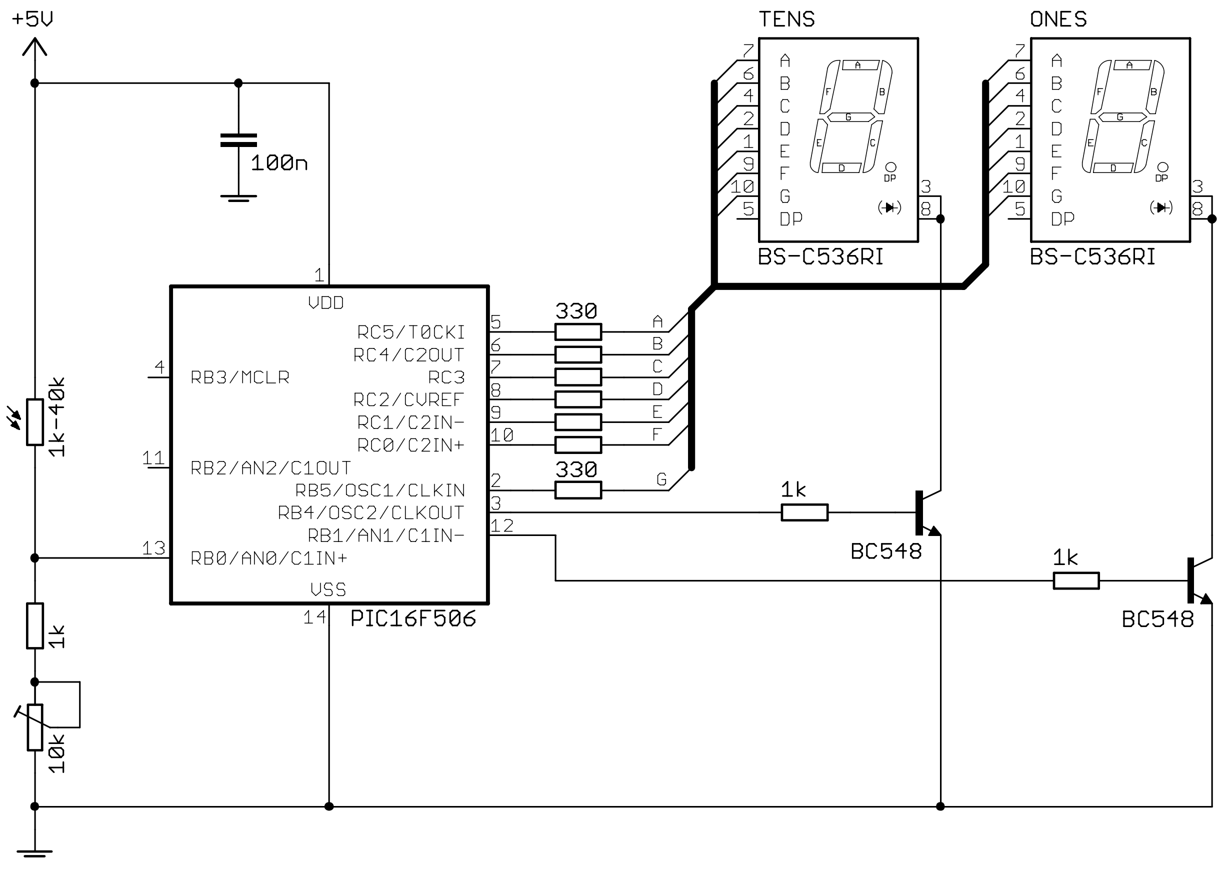
You'll see that two symbols are used for the GO/$\overline{\text{DONE}}$ bit, depending on the context: when setting the bit to start the conversion, it is referred to as "`GO`", but when using it as a flag to check whether the conversion is complete, it is referred to as "`NOT_DONE`".

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 ADRES, while the output LEDs (RC0 – RC3) form the low nybble of PORTC, making '`swapf`' a neat solution; much shorter than using four right-shifts.

Example 2: 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 8, as shown below:



To display the hexadecimal value, we can adapt the multiplexed 7-segment display code from lesson 8.

First, to drive the displays using RC0-RC5 and RB1, RB4 and RB5, we need to disable the comparators, comparator outputs, and voltage reference output:

```
        ; configure ports
        clrw                    ; configure PORTB and PORTC as all outputs
        tris    PORTB
        tris    PORTC
        clrf    CM1CON0         ; disable Comparator 1 (RB0, RB1, RB2 usable)
        clrf    CM2CON0         ; disable Comparator 2 (RC0, RC1, RC4 usable)
        clrf    VRCON           ; disable CVref (RC2 usable)
```

To configure RB1 for digital I/O, it is also necessary to deselect AN1 as an analog input.

Since we are using AN0 as an analog input in this example, we need to select ANS = 10 when initialising the ADC, as in the last example:

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

As we did in lesson 8, the timer is used to provide a ~2 ms tick to drive the display multiplexing:

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

This assumes a 4 MHz clock, not 8 MHz, so the configuration directive needs to include '`_IOSCFS_OFF`':

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

The lookup tables have to be extended to include 7-segment representations of the letters 'A' to 'F', but the lookup code remains the same as in lesson 8.

Since each digit is displayed for 2 ms, and the analog to digital conversion only takes around 13 µs, the ADC read can be completed well within the time spent waiting to begin displaying the next digit (~1 ms), without affecting the display multiplexing.

The main loop, then, simply consists of reading the analog input and displaying each digit of the result, then repeating that quickly enough for the display to appear to be continuous:

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

        ; display high nybble for 2.048ms
w10_hi  btfss   TMR0,2          ; wait for TMR0<2> to go high
        goto    w10_hi
        swapf   ADRES,w         ; get "tens" digit
        andlw   0x0F            ;   from high nybble of ADC result
        [code to display "tens" digit then wait for TMR<2> low goes here]

        ; display ones for 2.048ms
w1_hi   btfss   TMR0,2          ; wait for TMR0<2> to go high
        goto    w1_hi
        movf    ADRES,w         ; get ones digit
        andlw   0x0F            ;   from low nybble of ADC result
        [code to display "ones" digit then wait for TMR<2> low goes here]

        ; repeat forever
        goto    main_loop
```

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 in:

```
;************************************************************************
;                                                                       *
;   Description:    Lesson 10, example 2                                *
;                                                                       *
;   Displays ADC output in hexadecimal on 7-segment LED displays        *
;                                                                       *
;   Continuously samples analog input,                                  *
;   displaying result as 2 x hex digits on multiplexed 7-seg displays   *
;                                                                       *
;************************************************************************
;                                                                       *
;   Pin assignments:                                                    *
;       AN0         - voltage to be measured (e.g. pot or LDR)          *
;       RB5, RC0-5 - 7-segment display bus (common cathode)             *
;       RB4         - tens enable (active high)                         *
;       RB1         - ones enable                                       *
;                                                                       *
;************************************************************************

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

    radix       dec


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

; pin assignments
    #define TENS    PORTB,4     ; tens enable
    #define ONES    PORTB,1     ; ones enable


;***** VARIABLE DEFINITIONS
        UDATA_SHR
digit   res 1                   ; digit to be displayed


;***** RESET VECTOR *****************************************************
RESET   CODE    0x000           ; effective reset vector
        movwf   OSCCAL          ; update OSCCAL with factory cal value
        pagesel start
        goto    start           ; jump to main program

;***** Subroutine vectors
set7seg                         ; display digit on 7-segment display
        pagesel set7seg_R
        goto    set7seg_R


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

;***** Initialisation
start
        ; configure ports
        clrw                    ; configure PORTB and PORTC as all outputs
        tris    PORTB
        tris    PORTC
        clrf    CM1CON0         ; disable Comparator 1 (RB0, RB1, RB2 usable)
        clrf    CM2CON0         ; disable Comparator 2 (RC0, RC1, RC4 usable)
        clrf    VRCON           ; disable CVref (RC2 usable)
        ; configure ADC
        movlw   b'10110001'
                ; 10------        AN0, AN2 analog (ANS = 10)
                ; --11----        clock = INTOSC/4 (ADCS = 11)
                ; ----00--        select channel AN0 (CHS = 00)
                ; -------1        turn ADC on (ADON = 1)
        movwf   ADCON0
        ; configure timer
        movlw   b'11010111'     ; configure Timer0:
                ; --0-----          timer mode (T0CS = 0) -> RC5 usable
                ; ----0---          prescaler assigned to Timer0 (PSA = 0)
                ; -----111          prescale = 256 (PS = 111)
        option                  ;   -> increment every 256 us
                                ;      (TMR0<2> cycles every 2.048ms)

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

        ; display high nybble for 2.048ms
w10_hi  btfss   TMR0,2          ; wait for TMR0<2> to go high
        goto    w10_hi
        swapf   ADRES,w         ; get "tens" digit
        andlw   0x0F            ;   from high nybble of ADC result
        pagesel set7seg
        call    set7seg         ;   then output it
        pagesel $
        bsf     TENS            ; enable "tens" display
w10_lo  btfsc   TMR0,2          ; wait for TMR<2> to go low
        goto    w10_lo

        ; display ones for 2.048ms
w1_hi   btfss   TMR0,2          ; wait for TMR0<2> to go high
        goto    w1_hi
        movf    ADRES,w         ; get ones digit
        andlw   0x0F            ;   from low nybble of ADC result
        pagesel set7seg
        call    set7seg         ;   then output it
        pagesel $
        bsf     ONES            ; enable ones display
w1_lo   btfsc   TMR0,2          ; wait for TMR<2> to go low
        goto    w1_lo

        ; repeat forever
        goto    main_loop


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

; Lookup pattern for 7 segment display on port B
; RB5 = G
```

```
get7sB  addwf   PCL,f
        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

; Lookup pattern for 7 segment display on port C
; RC5:0 = ABCDEF
get7sC  addwf   PCL,f
        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

; Display digit passed in W on 7-segment display
set7seg_R
        movwf   digit           ; save digit
        call    get7sB          ; lookup pattern for port B
        movwf   PORTB           ;   then output it
        movf    digit,w         ; get digit
        call    get7sC          ;   then repeat for port C
        movwf   PORTC
        retlw   0


        END
```

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

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 digit jitters 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 quickly 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 high-frequency noise, such as that caused by fluorescent lighting.

We'll make those improvements in lesson 11. But first we'll look at one last example.

Example 3: Measuring Supply Voltage

As mentioned above, the 0.6 V absolute voltage reference can be sampled by the ADC, and this provides a way to infer the supply voltage (actually VDD – VSS, but to keep this simple we'll assume VSS = 0 V).

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

$$0.6 \text{ V} \div 5.0 \text{ V} \times 255 = 30$$

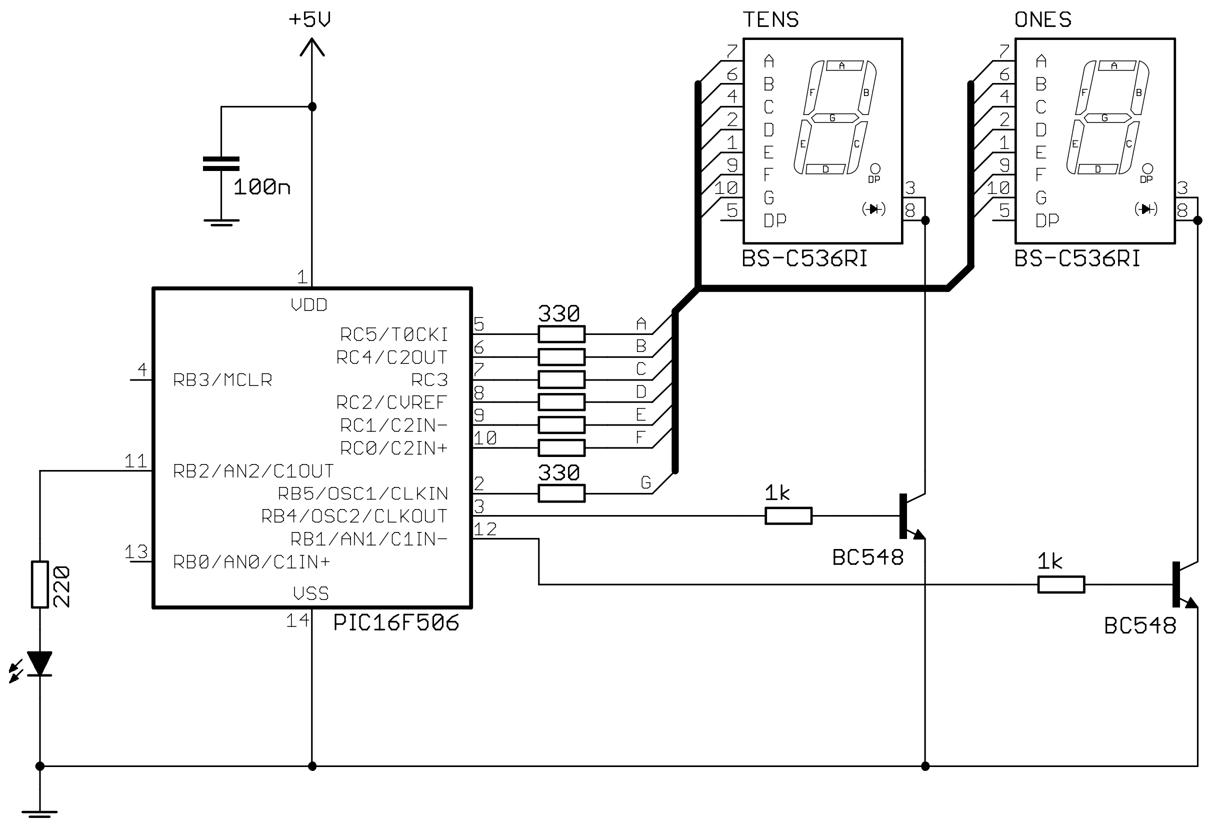
Now if VDD was to fall to, say, 3.5 V, the 0.6 V reference will read as:

$$0.6 \text{ V} \div 3.5 \text{ V} \times 255 = 43$$

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

So to check for the power supply falling too low, the value returned by sampling the 0.6 V reference can be compared with a threshold. For example, a value above 43 indicates that VDD < 3.5 V, and perhaps a warning should be displayed, or the device shut down before power falls too low.

To illustrate this, we can use adapt the circuit and program from example 2, displaying the ADC reading corresponding to the 0.6 V reference as two hex digits, and light a "low voltage warning" LED attached to RB2 (as shown below) if VDD falls below some threshold.



To implement this, the code from example 2 can be used with very little modification.

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 0.6V ref measurement
```

Note that, because 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 remains the same, except that the ADC configuration is changed to disable all the analog inputs (allowing RB2 to be used as a digital output) and selecting the internal 0.6 V reference as the ADC input channel:

```
        ; configure ADC
        movlw   b'00111101'
                ; 00------        no analog inputs (ANS = 00) -> RB0-2 usable
                ; --11----        clock = INTOSC/4 (ADCS = 11)
                ; ----11--        select 0.6V reference (CHS = 11)
                ; -------1        turn ADC on (ADON = 1)
```

After sampling the 0.6 V input, we can test for VDD being too low by comparing the conversion result (ADRES) with the threshold (VRMAX):

```
        ; sample 0.6 V reference
        bsf     ADCON0,GO       ; start conversion
w_adc   btfsc   ADCON0,NOT_DONE ; wait until conversion complete
        goto    w_adc

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

        ; display high nybble for 2.048ms
        [wait for TMR0<2> high then display "tens" digit]
```

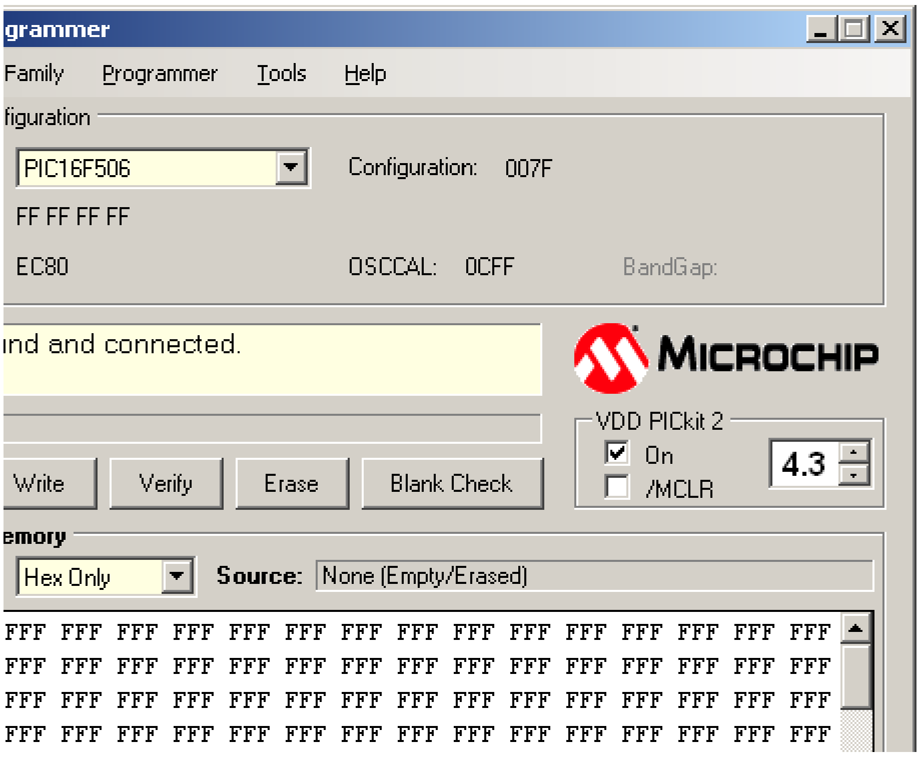
There's a slight problem with this approach: as soon as a digit is displayed, the LED on RB2 will be extinguished, since the lookup table for PORTB always returns a '0' for bit 2. To avoid that problem, the 'set7seg_R' routine would need to be modified to include logical masking operations so that RB2 is not overwritten. But it's not really a significant problem; the LED will remain lit for ~1 ms, while the "display high nybble" routine waits for TMR0<2> to go high, out of a total multiplex cycle time of ~4 ms. That is, when the LED is 'lit', it will actually be on for ~25% of the time, and that's enough to make it visible.

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 (shown on the next page) to vary VDD, while the circuit is powered.

But first, you should exit MPLAB, so that you don't have two applications trying to control the PICkit 2 at once.

In the PICkit 2 application, select the device (16F506) and then click ‘On’ in the ‘VDD PICkit 2’ box.



PIC16F506   Configuration: 007F

FF FF FF FF

EC80   OSCCAL: 0CFF   BandGap:

ınd and connected.

[✓ On] 4.3

[ ] /MCLR

Hex Only   Source: None (Empty/Erased)

```
FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF
FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF
FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF
FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF FFF
```

Your circuit should now be powered on, and, assuming the supply voltage is 5.0V, the display should show “1E” (30 in hexadecimal), or something close to that.

You can now start to decrease VDD, by clicking on the down arrow next to the voltage display, 0.1V at a time. When you get to 3.5V, the display should read “2b” (43 in hex), although the prototype displays “29” at 3.5V. Below this, 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 complete our overview of baseline PIC assembler, by demonstrating how to perform some simple arithmetic operations, including moving averages and working with arrays, to implement these suggested improvements.