Introduction to PIC Programming

Midrange Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 7: Interrupt-on-change, Sleep Mode and the Watchdog Timer

One of the most useful features of modern microcontrollers (including PICs) is their ability to enter a power-saving "sleep" mode, where power drain may be less than a microwatt, facilitating the design of low-powered devices without traditional on-off switches – the device can turn itself "off". For example, the Gooligum Christmas Star, based on a PIC12F683, runs on a pair of N-cell batteries, but will remain "shut off", with no significant battery drain, for a year or more, coming to life as soon as a pushbutton is pressed.

The latter feature relies on the PIC's "interrupt-on-change" facility, which is often used to wake the device from sleep. As we shall see, it can also (as the name suggests) be used to trigger an interrupt in response to a changing input; similar to the external interrupt facility introduced in lesson 6.

Another facility usually found in modern microcontrollers (including PICs) is a "watchdog timer", intended to make a device more robust by providing a means of detecting situations where the program appears to be hung, and then resetting the processor so that the system can recover.

But as we'll see in this lesson, the watchdog timer can also be used to periodically wake the PIC from sleep, a facility which makes it possible to design devices which spend most of their time sleeping, drawing very little current (and hence power) on average.

In summary, this lesson covers:

- Interrupt-on-change
- Sleep mode (power down)
- Wake-up on change (power up)
- The watchdog timer
- Periodic wake from sleep

Interrupt-on-change

In lesson 6, we saw that the 12F629's external interrupt facility can used to trigger an interrupt on each rising or falling transition on the INT (GP2) pin; useful when we need to respond to an external digital signal more quickly than using a timer interrupt to poll the input every millisecond or so, without having to tie up the processor with a tight polling loop. Instead, the processor can be going about other tasks, but still be able to service the external event within microseconds of it occurring.

Note that the external interrupt is triggered on either a rising or falling edge (selectable by the INTEDG bit), but not both. If rising edges are selected, falling edges will be ignored. This tends to simplify code, since we are normally only interested in one type of transition.

The midrange PIC architecture only supports a single external interrupt pin. However, the interrupt-on-change facility can be used if you need to respond quickly to a number of digital signal sources.

GP port change interrupts are enabled by setting the GPIE bit in the INTCON register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
INTCON	GIE	PEIE	T0IE	INTE	GPIE	T0IF	INTF	GPIF

As always, for any interrupts to occur, the global interrupt enable bit, GIE, must also be set.

Every pin in the GPIO port can be enabled independently for interrupt-on-change.

This is controlled by the IOC register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
IOC	-	-	IOC5	IOC4	IOC3	IOC2	IOC1	IOC0

If a bit in the IOC register is set, the corresponding GPIO pin will be enabled for interrupt-on-change.

For example, to enable interrupt-on-change for GP2, we would set IOC2 = 1.

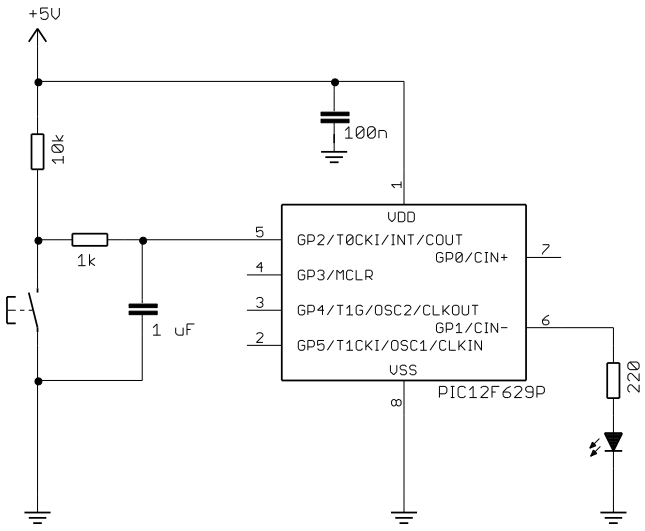
If a pin is enabled for interrupt on change, any change in the state of that pin (since the last time the port was read or written) will create a *mismatch condition* and set the GPIF flag in the INTCON register. If GP port change interrupts are also enabled (GPIE = 1 and GIE = 1), an interrupt will be triggered.

This means that you should read or write GPIO immediately before enabling the port change interrupt, to end any existing mismatch condition, avoiding the interrupt being triggered the moment it is enabled. Similarly, in the interrupt service routine it is important to read or write GPIO, to end the mismatch before clearing the GPIF flag. If you do not end the mismatch, you will not be able to clear GPIF, and the interrupt will re-trigger, as soon as the ISR ends.

Note also that, unlike external interrupts, any change – whether a rising or falling transition – will trigger a port change interrupt.

Example 1: Interrupt-on-change (single input)

We'll start by demonstrating how to use interrupt-on-change to respond to a single input, using the circuit from the external interrupt example in lesson 6 (shown on the right), where a pushbutton is connected to GP2 via a simple RC filter.



As we did in that example, we'll toggle the LED on GP1 whenever the pushbutton is pressed.

GP2 was used because, on the 12F629, it has a Schmitt-trigger input, allowing the simple RC filter to provide effective hardware debouncing, as explained in baseline lesson 4.

This is necessary because, although the switch debouncing could be implemented in software, it is difficult to do so while responding quickly to changes.

Firstly, in our initialisation code, we need to enable interrupt-on-change on GP2:

```
        banksel IOC             ; enable interrupt-on-change
        bsf     IOC,nBUTTON     ;   on pushbutton input
```

(where '`BUTTON`' is a constant which has been set to '2')

Then we can enable port change interrupts, after having written to GPIO to clear any existing port mismatch condition:

```
        banksel GPIO            ;   (write to GPIO will clear any mismatch)
        clrf    GPIO            ; start with all LEDs off
        clrf    sGPIO           ;   update shadow
        ; configure interrupts
        movlw   1<<GIE|1<<GPIE  ; enable port change and global interrupts
        movwf   INTCON
```

In the interrupt handler, we must clear the port mismatch condition which triggered this interrupt and (as with all interrupts) clear the interrupt flag:

```
        banksel GPIO
        movf    GPIO,w              ; clear mismatch condition
        bcf     INTCON,GPIF         ; clear interrupt flag
```

Since the port change interrupt is triggered by any change, the ISR will be run on both button press and button release. This is different from the external interrupt example in lesson 6, where the ISR only had to handle button press events.

Therefore, we must check whether the button had been pressed or released:

```
        ; toggle LED only on button press
        btfsc   GPIO,nBUTTON        ; is button down?
        goto    isr_end
```

If the button was pressed, we can toggle the LED on GP1, as we've done before:

```
        movlw   1<<nB_LED           ; if so, toggle indicator LED
        xorwf   sGPIO,f             ;   using shadow register
```

(where '`nB_LED`' is a constant which has been set to '1')

Otherwise, the code, including processor context save and restore, is essentially the same as that in the examples from lesson 6.

Complete program

Here is how these pieces fit together, along with interrupt code framework introduced in lesson 6:

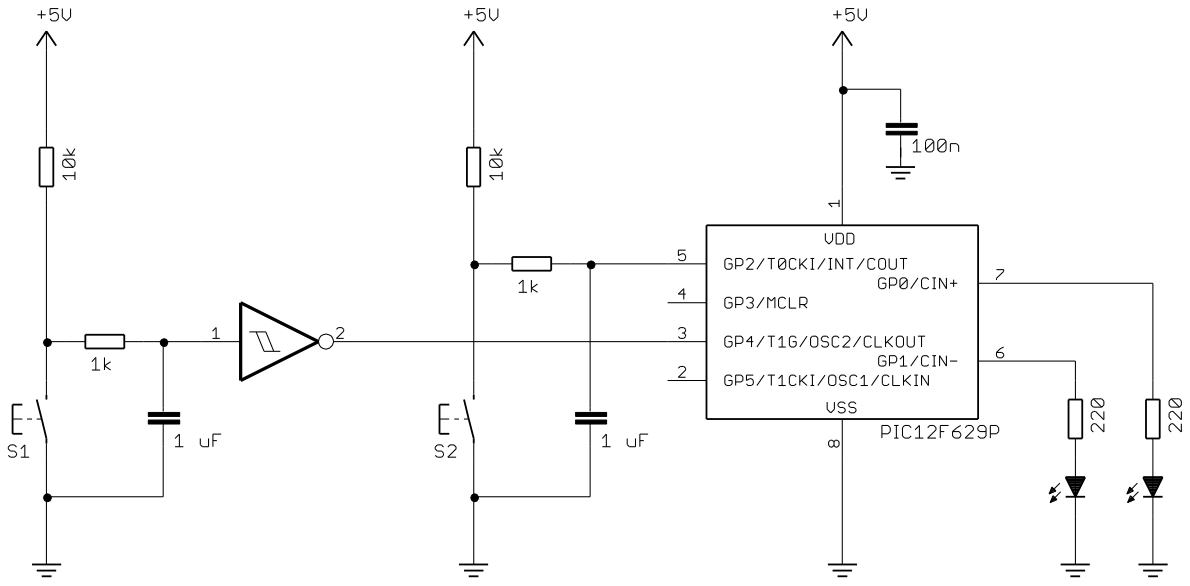
```
;*****************************************************************
;   Description:    Lesson 7 example 1                           *
;                                                                *
;   Demonstrates use of interrupt-on-change interrupts           *
;   (without software debouncing)                                *
;   Toggles LED on GP1                                           *
;   when pushbutton on GP2 is pressed (high -> low transition)   *
;*****************************************************************
;   Pin assignments:                                             *
;       GP1 - indicator LED                                      *
;       GP2 - pushbutton (externally debounced, active low)      *
;*****************************************************************
```

```
    list        p=12F629
    #include    <p12F629.inc>

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


;***** CONFIGURATION
                ; ext reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer, 4Mhz int clock
    __CONFIG    _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

; pin assignments
    constant    nB_LED=1            ; "button pressed" indicator LED on GP1
    constant    nBUTTON=2           ; externally debounced pushbutton on GP2


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

GENVAR      UDATA_SHR       ; general variables
sGPIO       res 1               ; shadow copy of GPIO


;*********************************************************************
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 interrupt-on-change
        ;   Triggered on any transition on IOC-enabled input pin
        ;   caused by externally debounced pushbutton press
        ;
        banksel GPIO
        movf    GPIO,w              ; clear mismatch condition
        bcf     INTCON,GPIF         ; clear interrupt flag
        ; toggle LED only on button press
        btfsc   GPIO,nBUTTON        ; is button down?
        goto    isr_end
        movlw   1<<nB_LED           ; if so, toggle indicator LED
        xorwf   sGPIO,f             ;   using shadow register

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
Start   ; calibrate internal RC oscillator
        call    0x03FF          ; retrieve factory calibration value
        banksel OSCCAL          ;   then update OSCCAL
        movwf   OSCCAL

;***** Initialisation
        ; configure port
        movlw   ~(1<<nB_LED)    ; configure LED pin as output
        banksel TRISIO
        movwf   TRISIO
        ; initialise and configure port
        banksel IOC             ; enable interrupt-on-change
        bsf     IOC,nBUTTON     ;   on pushbutton input
        banksel GPIO            ;   (write to GPIO will clear any mismatch)
        clrf    GPIO            ; start with all LEDs off
        clrf    sGPIO           ;   update shadow
        ; configure interrupts
        movlw   1<<GIE|1<<GPIE  ; enable port change and global interrupts
        movwf   INTCON

;***** Main loop
loop
        ; continually copy shadow GPIO to port
        banksel GPIO
        movf    sGPIO,w
        movwf   GPIO

        ; repeat forever
        goto    loop


        END
```

Example 2: Interrupt-on-change (multiple inputs)

This example demonstrates how to handle the situation where interrupt-on-change is enabled on more than one input pin, using the circuit shown below:



Each pushbutton toggles an LED: S1 controls the LED on GP1, and S2 controls the LED on GP0.

Once again, both buttons are debounced using hardware, to avoid messy software debounce routines (which would miss the point of using interrupt-on-change; if we were going to implement software debouncing, we'd be better off using a timer interrupt to poll the inputs, as we did in lesson 6). For effective hardware debouncing, the simple RC filters need to be coupled with Schmitt-trigger inputs, and since the only available Schmitt-trigger GP input on the 12F629 is GP2, a Schmitt-trigger inverter (such as a 74HC14) is used to drive GP4.

Thus, the operation of S1 is inverted, with respect to S2; GP4 is driven high when S1 is pressed, while GP2 is pulled low when S2 is pressed. We will have to take this difference into account.

The basic difficulty with having interrupt-on-change enabled for more than one input is that there are no flags to indicate which input changed; the GPIF flag can tell you that a port change has happened, but not which pin changed.

So when a port change interrupt occurs, we need to deduce which pin(s) have changed, by reading GPIO and comparing it to the last recorded state. And then, before exiting the ISR, we need to update our "last state" record, ready for next time.

Hence, we need some variables to store this information:

```
GENVAR      UDATA_SHR       ; general variables
sGPIO       res 1               ; shadow copy of GPIO
lGPIO       res 1               ; last state of GPIO (for change detection)
cGPIO       res 1               ; current state of GPIO (used by IOC ISR)
```

In the initialisation code, we need to enable interrupt-on-change for both inputs, and update the "last state" variable when initialising the port, so that everything is in sync:

```
; initialise and configure port
banksel IOC             ; enable interrupt-on-change
movlw   1<<nPB1|1<<nPB2 ;   on pushbuttons 1 and 2
movwf   IOC
banksel GPIO
clrf    GPIO            ; start with all LEDs off
clrf    sGPIO           ;   update shadow
movf    GPIO,w          ; read GPIO to clear any IOC mismatch condition
movwf   lGPIO           ;   update last state (for pin change detection)
```

Then, when handling the port change interrupt in the ISR, we need to determine which pins have changed. This can be done by XORing the current state of GPIO with the last recorded state. Since an XOR operation only results in a '1' where the inputs differ, this is a means of detecting which bits have changed:

```
bcf     INTCON,GPIF         ; clear interrupt flag

; determine which pins have changed
banksel GPIO                ; read GPI0
movf    GPIO,w              ;   to clear mismatch condition
movwf   cGPIO               ;   and save current state
xorwf   lGPIO,f             ; XOR with last state to detect changes
```

Note that the result of the XOR was written back to `lGPIO`, which now contains '0's in bit positions where the current state matches the last state and'1's where they differ. That is, if a pin has changed, the corresponding bit in `lGPIO` will be set to '1'.

Next we need to check each bit in `lGPIO` corresponding to the interrupt-on-change inputs, and toggle the appropriate LED if that input has changed:

```
        ; toggle LED 1 only on button 1 press (active low)
        btfss   lGPIO,nPB1          ; has button 1 changed?
        goto    ioc_pb2             ;   check next button if not
        btfsc   cGPIO,nPB1          ; is button down (low)?
        goto    ioc_pb2             ;   check next button if not
        movlw   1<<nB1_LED          ; if so, toggle LED 1
        xorwf   sGPIO,f             ;   using shadow register

ioc_pb2 ; toggle LED 2 only on button 2 press (active high)
        btfss   lGPIO,nPB2          ; has button 2 changed?
        goto    ioc_end             ;   finish IOC if not
        btfss   cGPIO,nPB2          ; is button down (high)?
        goto    ioc_end             ;   finish IOC if not
        movlw   1<<nB2_LED          ; if so, toggle LED 2
        xorwf   sGPIO,f             ;   using shadow register
```

Note that the test for "button press" button 2 is opposite to that for button 1, because of the external inverter on the GP4 input, as discussed above.

Finally, we need to record the current GPIO state, for reference as the "last state", the next time a port change interrupt occurs:

```
ioc_end ; update last GPIO state (for next time)
        movf    cGPIO,w             ; copy current state of GPIO
        movwf   lGPIO               ;   to last state
```

Complete program

Here is how these pieces fit into the framework used in the first example, to form the complete "interrupt-on-change with multiple inputs" program:

```
;*****************************************************************
;   Description:    Lesson 7 example 2                          *
;                                                               *
;   Demonstrates handling of multiple interrupt-on-change interrupts    *
;   (without software debouncing)                               *
;                                                               *
;   Toggles LED on GP0 when pushbutton on GP2 is pressed        *
;   (high -> low transition)                                    *
;   and LED on GP1 when pushbutton on GP4 is pressed            *
;   (low -> high transition)                                    *
;                                                               *
;*****************************************************************
;                                                               *
;   Pin assignments:                                            *
;       GP0 - indicator LED 1                                   *
;       GP1 - indicator LED 2                                   *
;       GP2 - pushbutton 1 (externally debounced, active low)   *
;       GP4 - pushbutton 2 (externally debounced, active high)  *
;                                                               *
;*****************************************************************

    list        p=12F629
    #include    <p12F629.inc>

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

```
;***** CONFIGURATION
                ; ext reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer, 4Mhz int clock
    __CONFIG    _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

; pin assignments
    constant    nB1_LED=0           ; "button 1 pressed" indicator LED on GP0
    constant    nB2_LED=1           ; "button 2 pressed" indicator LED on GP1
    constant    nPB1=2              ; pushbutton 1 (ext debounce, active low) on
GP2
    constant    nPB2=4              ; pushbutton 2 (ext debounce, active high)
on GP4


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

GENVAR      UDATA_SHR       ; general variables
sGPIO       res 1               ; shadow copy of GPIO
lGPIO       res 1               ; last state of GPIO (for change detection)
cGPIO       res 1               ; current state of GPIO (used by IOC ISR)


;*********************************************************************
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 interrupt-on-change
        ;   Triggered on any transition on IOC-enabled input pins
        ;   caused by externally debounced pushbutton press
        ;
        bcf     INTCON,GPIF         ; clear interrupt flag

        ; determine which pins have changed
        banksel GPIO                ; read GPI0
        movf    GPIO,w              ;   to clear mismatch condition
        movwf   cGPIO               ;   and save current state
        xorwf   lGPIO,f             ; XOR with last state to detect changes

        ; toggle LED 1 only on button 1 press (active low)
        btfss   lGPIO,nPB1          ; has button 1 changed?
        goto    ioc_pb2             ;   check next button if not
        btfsc   cGPIO,nPB1          ; is button down (low)?
        goto    ioc_pb2             ;   check next button if not
        movlw   1<<nB1_LED          ; if so, toggle LED 1
        xorwf   sGPIO,f             ;   using shadow register
```

```
ioc_pb2 ; toggle LED 2 only on button 2 press (active high)
        btfss   lGPIO,nPB2          ; has button 2 changed?
        goto    ioc_end             ;   finish IOC if not
        btfss   cGPIO,nPB2          ; is button down (high)?
        goto    ioc_end             ;   finish IOC if not
        movlw   1<<nB2_LED          ; if so, toggle LED 2
        xorwf   sGPIO,f             ;   using shadow register

ioc_end ; update last GPIO state (for next time)
        movf    cGPIO,w             ; copy current state of GPIO
        movwf   lGPIO               ;   to last state


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
Start   ; calibrate internal RC oscillator
        call    0x03FF          ; retrieve factory calibration value
        banksel OSCCAL          ;   then update OSCCAL
        movwf   OSCCAL

;***** Initialisation
        ; configure port
        movlw   ~(1<<nB1_LED|1<<nB2_LED)    ; configure LED pins as outputs
        banksel TRISIO
        movwf   TRISIO
        ; initialise and configure port
        banksel IOC             ; enable interrupt-on-change
        movlw   1<<nPB1|1<<nPB2 ;   on pushbuttons 1 and 2
        movwf   IOC
        banksel GPIO
        clrf    GPIO            ; start with all LEDs off
        clrf    sGPIO           ;   update shadow
        movf    GPIO,w          ; read GPIO to clear any IOC mismatch condition
        movwf   lGPIO           ;   update last state (for pin change detection)
        ; configure interrupts
        movlw   1<<GIE|1<<GPIE  ; enable port change and global interrupts
        movwf   INTCON

;***** Main loop
loop
        ; continually copy shadow GPIO to port
        banksel GPIO
        movf    sGPIO,w
        movwf   GPIO

        ; repeat forever
        goto    loop


        END
```

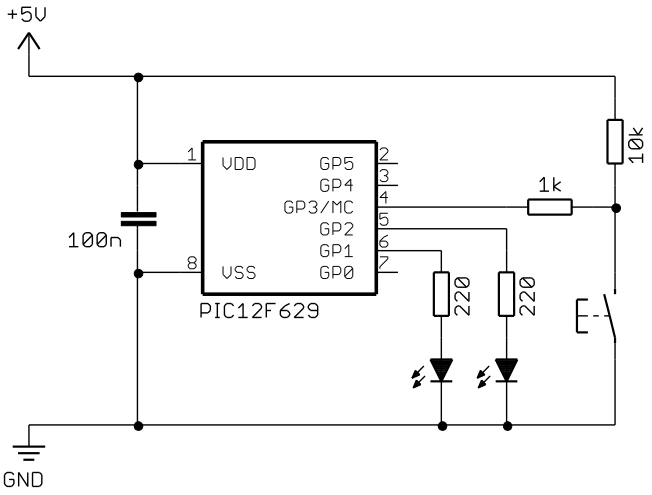
Sleep Mode

As mentioned earlier, midrange PICs are able to enter a standby, or *sleep* mode, to save power.

In this mode, the PIC12F629 will typically draw less than 3 nA (down to only 1 nA when the power supply is reduced to 2 V), when all of the power-consuming facilities (such as the watchdog timer; see later) have been disabled and the output pins are not supplying any current.

To demonstrate how it is used, we'll use the circuit from lesson 6, shown on the right.

It consists of a PIC12F629, LEDs on GP1 and GP2, and a pushbutton switch on GP3. It can be readily built on Microchip's LPC Demo Board, as described in baseline lesson 1. But if you want to demonstrate to yourself that power consumption really is reduced when the PIC enters sleep mode, you will need to build the circuit such that you can place a multimeter in line with the power supply (or use a power supply with a current display), so that you can measure the supply current. You could, for example, easily build this circuit on prototyping breadboard.



The instruction for placing the PIC into standby mode is '`sleep`' – "enter **sleep** mode".

To illustrate the use of the sleep instruction, consider the following fragment of code. It turns on the LED on GP1, waits for the button to be pressed, and then enters sleep mode:

```
        movlw   ~(1<<GP1)       ; configure LED pin as output
        banksel TRISIO
        movwf   TRISIO

        banksel GPIO
        bsf     GPIO,GP1        ; turn on LED

waitlo  btfsc   GPIO,GP3        ; wait for button press (low)
        goto    waitlo

        sleep                   ; enter sleep mode

        goto    $               ; (this instruction should never run)
```

Note that the final '`goto $`' instruction (an endless loop) will never be executed, because '`sleep`' will halt the processor; any instructions after '`sleep`' will never be reached.

When you run this program, the LED will turn on and then, when you press the button, nothing will appear to happen! The LED stays on. Shouldn't it turn off? What's going on?

The current supplied from a 5 V supply, before pressing the button, with the LED on, was measured to be 11.27 mA. After pressing the button, the measured current dropped to 10.57 mA, a fall of only 0.70 mA.

This happens because, when the PIC goes into standby mode, the PIC stops executing instructions, saving some power (0.70 mA × 5 V = 3.5 mW in this case), but the I/O ports remain in the state they were in, before the '`sleep`' instruction was executed.

> *Note: For low power consumption in standby mode, the I/O ports must be configured to stop sourcing or sinking current, before entering SLEEP mode.*

In this case, the fix is simple – turn off the LED before entering sleep mode, as follows:

```
        movlw   ~(1<<GP1)       ; configure LED pin as output
        banksel TRISIO
        movwf   TRISIO

        banksel GPIO
        bsf     GPIO,GP1        ; turn on LED

waitlo  btfsc   GPIO,GP3        ; wait for button press (low)
        goto    waitlo

        bcf     GPIO,GP1        ; turn off LED

        sleep                   ; enter sleep mode

        goto    $               ; (this instruction should never run)
```

When this program is run, the LED will turn off when the button is pressed.

The current measured in the prototype with the PIC in standby and the LED off was less than 0.1 µA – too low to register on the multimeter used! That was with the unused pins tied to V_{DD} or V_{SS} (whichever is most convenient on the circuit board), as floating CMOS inputs can lead to unnecessary current draw.

> *Note: To minimise power in standby mode, configure all unused pins as inputs, and tie them V_{DD} or V_{SS} through 10 kΩ resistors. Do not connect them directly to V_{DD} or V_{SS}, as the PIC may be damaged if these pins are inadvertently configured as outputs.*

For clarity, tying the unused inputs to V_{DD} or V_{SS} was not shown in the circuit diagram above.

Wake-up from sleep

Sleep mode would not be useful if there was no way to wake up from it – there has to be a way to turn the device "on" when needed (perhaps in response to an event, such as a button press), after it has been turned "off".

Midrange PICs provide a number of ways to wake from sleep mode:

- Any device reset, such as an external reset signal on the $\overline{\text{MCLR}}$ pin (if enabled)
- Watchdog timer timeout (see the section on the watchdog timer, later in this lesson)
- Any enabled interrupt source which can set its interrupt flag while in sleep mode

Some interrupt sources cannot be used wake the device from sleep, because, in sleep mode, the PIC's clock, or oscillator, is not running. For example, the Timer0 interrupt cannot be used for wake-up from sleep, because TMR0 does not increment while the PIC is in sleep mode.

However, external (INT pin) and port change interrupts can be used for wake-up on midrange PICs, as well as some other interrupt sources, such as Timer1 and comparators, that we will examine in later lessons.

In this lesson, we'll look at how to use the port change interrupt to wake a PIC from sleep mode; the method for using an external interrupt is essentially the same, but is of course limited to only the INT pin.

Example 4: Using interrupt-on-change for wake-up from sleep

In baseline lesson 7, we saw that a "wake-up on change" facility is available in the baseline architecture on a handful of pins, but that it is an all or nothing affair; either all of the available pins are enabled for wake-up on change, or none of them are.

The midrange equivalent to wake-up on change is the interrupt-on-change facility introduced above. It is more flexible, in that interrupt-on-change can be enabled independently on each pin. And on the 12F629, interrupt-on-change is available on every pin in GPIO.

"Interrupt-on-change" can be used to wake the device from sleep, even if interrupts are not enabled. If port change interrupts are enabled (GPIE = 1), but global interrupts are disabled (GIE = 0), then the device will wake from sleep when an IOC-enabled input changes, but no interrupt will occur. Program execution simply continues with the instruction following the `sleep` instruction.

> *Note: in the midrange PIC architecture, on wake-up from sleep, program execution continues with the instruction following `sleep`. No device reset occurs (unless a reset event, such as a reset signal on $\overline{MCLR}$ caused the wake-up). This is different from the baseline architecture.*

If port change interrupts are enabled (GPIE = 1) and global interrupts are enabled (GIE = 1), if a change occurs on an IOC-enabled input while the PIC is in sleep mode, the device will wake from sleep, execute the instruction following `sleep`, and then enter the interrupt service routine.

If you want the PIC to execute the ISR immediately after it wakes from sleep, you need to enable interrupts and place a `nop` ("do nothing") instruction immediately following the `sleep` instruction.

If you are using other interrupts in your program, and don't want to execute the ISR when the PIC wakes from sleep, simply disable interrupts (clear GIE) before entering sleep mode.

In any case, if GPIE = 1, the PIC will wake if the value of any IOC-enabled input changes while it is in sleep mode.

It is important to clear the GPIF flag before entering sleep mode, or else the PIC will wake immediately.

> *Note: You should read the input pins configured for interrupt-on-change just prior to entering sleep mode, and clear GPIF. Otherwise, if the value at an IOC-enabled pin had changed since the last time it was read, the PIC will wake immediately upon entering sleep mode, as the input value would be seen to be different from that last read.*

It is also important to ensure that any input which will be used to trigger a wake-up is stable before entering sleep mode. Consider what would happen if interrupt-on-change was enabled in the program above. As soon as the button is pressed, the LED will turn off and the PIC will enter standby mode, as intended. But on the first switch bounce, the input would be seen to have changed, and the PIC would wake.

Even if the circuit included hardware debouncing, there's still a problem: the LED will go off and the PIC will enter standby as soon as the button is pressed, but when the button is subsequently released, it will be seen as a change, and the PIC will wake up! To successfully use the pushbutton to turn the circuit (PIC and LED) "off", it is necessary to wait for the button to be released and remain stable (debounced) before entering sleep mode.

But there's still a potential problem. Assume that, in this example, we want to wake-up the PIC and turn the LED on when the button is pressed. PICs are fast, and human fingers are slow – if, as soon as the PIC waits from sleep, the program immediately checks for a "turn off" button press, the button will still be down, as part of the button press which woke the PIC from sleep, and the LED will immediately turn off again. To avoid this, we must wait for the button to be in a stable "up" state before checking that it is "down".

So the necessary sequence is:

```
loop
    turn on LED
    wait for stable button high
    wait for button low
    turn off LED
    wait for stable button high
    clear GPIF
    sleep
    goto loop  ; repeat from the beginning
```

The following code, which makes use of the debounce macro defined in lesson 5, implements this:

```
;***** Initialisation
        ; configure port
        movlw   ~(1<<nLED)      ; configure LED pin as output
        banksel TRISIO
        movwf   TRISIO
        ; configure Timer0 (for DbnceHi macro)
        movlw   b'11000111'     ; configure Timer0:
                ; --0-----          timer mode (T0CS = 0)
                ; ----0---          prescaler assigned to Timer0 (PSA = 0)
                ; -----111          prescale = 256 (PS = 111)
        banksel OPTION_REG      ;   -> increment TMR0 every 256 us
        movwf   OPTION_REG
        ; configure interrupt-on-change
        banksel IOC             ; enable interrupt-on-change
        bsf     IOC,nBUTTON     ;   on pushbutton input
        bsf     INTCON,GPIE     ; enable wake-up (interrupt) on port change

;***** Main loop
loop
        banksel GPIO            ; turn on LED
        bsf     GPIO,nLED

        DbnceHi GPIO,nBUTTON    ; wait for stable button high
                                ;   (in case restarted after button press)

waitlo  btfsc   GPIO,nBUTTON    ; wait for button press (low)
        goto    waitlo

        bcf     GPIO,nLED       ; turn off LED

        DbnceHi GPIO,nBUTTON    ; wait for stable button release

        bcf     INTCON,GPIF     ; clear port change interrupt flag

        sleep                   ; enter sleep mode

        goto    loop            ; repeat forever
```

(the labels '`nLED`' and '`nBUTTON`' are defined earlier in the program)

This code does essentially the same thing as the "toggle an LED" programs developed in lesson 3, except that in this case, when the LED is off, the PIC is drawing negligible power.

Watchdog Timer

In the real world, computer programs sometimes "crash"; they will stop responding to input, stuck in a continuous loop they can't get out of, and the only way out is to reset the processor (e.g. Ctrl-Alt-Del on Windows PCs – and even that sometimes won't work, and you need to power cycle a PC to bring it back). Microcontrollers are not immune to this. Their programs can become stuck because some unforseen sequence of inputs has occurred, or perhaps because an expected input signal never arrives. Or, in the electrically noisy industrial environment in which microcontrollers are often operating, power glitches and EMI on signal lines can create an unstable environment, perhaps leading to a crash.

Crashes present a special problem for equipment which is intended to be reliable, operating autonomously, in environments where user intervention isn't an option.

One of the major functions of a *watchdog timer* is to automatically reset the microcontroller in the event of a crash. It is simply a free-running timer (running independently of any other processor function, including sleep) which, if allowed to overflow, will reset the PIC. In normal operation, an instruction which clears the watchdog timer is regularly executed – often enough to prevent the timer ever overflowing. This instruction is often placed in the "main loop" of a program, where it would normally be expected to be executed often enough to prevent watchdog timer overflows. If the program crashes, the main loop presumably won't complete; the watchdog timer won't be cleared, and the PIC will be reset. Hopefully, when the PIC restarts, whatever condition led to the crash will have gone away, and the PIC will resume normal operation.

The instruction for clearing the watchdog timer is '`clrwdt`' – "**cl**ea**r** **w**atch**d**og **t**imer".

The watchdog timer has a nominal time-out period of 18 ms. If that's not long enough, it can be extended by using a prescaler.

As we saw in lesson 4, a single prescaler is shared between Timer0 and the watchdog timer – it can be assigned to one or the other, but not both.

It is configured using a number of bits in the OPTION register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OPTION_REG	$\overline{\text{GPPU}}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0

To assign the prescaler to the watchdog timer, set the PSA bit to '1'.

When assigned to the watchdog timer, the prescale ratio is set by the PS<2:0> bits, as shown in the following table:

PS<2:0> bit value	WDT prescale ratio	WDT period (nominal)
000	1 : 1	18 ms
001	1 : 2	36 ms
010	1 : 4	72 ms
011	1 : 8	144 ms
100	1 : 16	288 ms
101	1 : 32	576 ms
110	1 : 64	1.15 s
111	1 : 128	2.30 s

Note that the prescale ratios are one half of those that apply when the prescaler is assigned to Timer0.

For example, if PSA = 1 (assigning the prescaler to the watchdog timer) and PS<2:0> = '011' (selecting a ratio of 1:8), the watchdog time-out period will be 8 × 18 ms = 144 ms.

With the maximum prescale ratio, the watchdog time-out period is 128 × 18 ms = 2.3 s.

The watchdog timer is controlled by the WDTE bit in the configuration word:

Bit 13	12	11	10	9	8	7	6	5	4	3	2	1	Bit 0
BG1	BG0	-	-	-	$\overline{\text{CPD}}$	$\overline{\text{CP}}$	BODEN	MCLRE	$\overline{\text{PWRTE}}$	WDTE	FOSC2	FOSC1	FOSC0

Setting WDTE to '1' enables the watchdog timer.

To set WDTE, use the symbol '_WDT_ON' instead of '_WDT_OFF' in the __CONFIG directive.

Since the configuration word cannot be accessed by programs running on the PIC (it can only be written to when the PIC is being programmed), **the watchdog timer cannot be enabled or disabled at runtime**. It can only be configured to be 'on' or 'off' when the PIC is programmed.

Example 5a: Watchdog Timer

To show how the watchdog timer allows the PIC to recover from a crash, we'll use a simple program which turns on an LED for 1.0 s, turns it off again, and then enters an endless loop (simulating a crash).

If the watchdog timer is disabled, the loop will never exit and the LED will remain off. But if the watchdog timer is enabled, with a period of 2.3 s, the program should restart itself after 2.3s, and the LED will flash: on for 1.0 s and off for 1.3 s (approximately).

To make it easy to test configurations with the watchdog timer on or off, you can use a construct such as:

```
#define     WATCHDOG        ; define to enable watchdog timer

IFDEF WATCHDOG
                ; ext reset, no code or data protect, no brownout detect,
                ; watchdog, power-up timer, 4Mhz int clock
    __CONFIG    _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_ON &
                _PWRTE_ON & _INTRC_OSC_NOCLKOUT
ELSE
                ; ext reset, no code or data protect, no brownout detect,
                ; no watchdog, power-up timer, 4Mhz int clock
    __CONFIG    _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
                _PWRTE_ON & _INTRC_OSC_NOCLKOUT
ENDIF
```

Note that these __CONFIG directives enable external reset ('_MCLRE_ON'), allowing the pushbutton switch connected to pin 4, to reset the PIC. That's useful because, with this program going into an endless loop, having to power cycle the PIC to restart it would be annoying; pressing the button is much more convenient.

The prescaler is set to 1:128 and assigned to the watchdog timer by:

```
    movlw   1<<PSA | b'111' ; assign prescaler to WDT (PSA = 1)
                            ; prescale = 128 (PS = 111)
    banksel OPTION_REG      ; -> WDT timeout = 2.3 s
    movwf   OPTION_REG
```

The code to flash the LED once and then enter an endless loop is simple, making use of the 'DelayMS' macro introduced in lesson 5:

```
    banksel GPIO            ; turn on LED
    bsf     GPIO,nLED

    DelayMS 1000            ; delay 1s

    banksel GPIO            ; turn off LED
    bcf     GPIO,nLED

    goto    $               ; wait forever
```

Complete program

If you build and run this program, with '`#define WATCHDOG`' commented out (place a '`;`' in front of it), the LED will light once, and then remain off. But if you define '`WATCHDOG`', the LED will continue to flash:

```
;*****************************************************************
;                                                                *
;   Description:    Lesson 7, example 5a                         *
;                                                                *
;   Demonstrates use of watchdog timer                           *
;                                                                *
;   Turn on LED for 1s, turn off, then enter endless loop        *
;   LED stays off if watchdog not enabled, flashes if WDT set to 2.3s   *
;                                                                *
;*****************************************************************
;                                                                *
;   Pin assignments:                                             *
;       GP1 - indicator LED                                      *
;                                                                *
;*****************************************************************

    list        p=12F629
    #include    <p12F629.inc>

    #include    <stdmacros-mid.inc>     ; DelayMS - delay in milliseconds

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

    radix       dec

    EXTERN      delay10         ; W x 10ms delay


;***** CONFIGURATION
    #define     WATCHDOG        ; define to enable watchdog timer

    IFDEF WATCHDOG
                    ; ext reset, no code or data protect, no brownout detect,
                    ; watchdog, power-up timer, 4Mhz int clock
        __CONFIG    _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_ON &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT
    ELSE
                    ; ext reset, no code or data protect, no brownout detect,
                    ; no watchdog, power-up timer, 4Mhz int clock
        __CONFIG    _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT
    ENDIF

; pin assignments
    constant    nLED=1          ; indicator LED on GP1


;*****************************************************************
RESET   CODE    0x0000          ; processor reset vector
        ; calibrate internal RC oscillator
        call    0x03FF          ; retrieve factory calibration value
        banksel OSCCAL          ;   then update OSCCAL
        movwf   OSCCAL
```

```
;***** Initialisation
        ; configure port
        movlw   ~(1<<nLED)      ; configure LED pin as output
        banksel TRISIO
        movwf   TRISIO
        ; configure watchdog timer
        movlw   1<<PSA | b'111' ; assign prescaler to WDT (PSA = 1)
                                ; prescale = 128 (PS = 111)
        banksel OPTION_REG      ; -> WDT timeout = 2.3 s
        movwf   OPTION_REG

;***** Main code
        banksel GPIO            ; turn on LED
        bsf     GPIO,nLED

        DelayMS 1000            ; delay 1s

        banksel GPIO            ; turn off LED
        bcf     GPIO,nLED

        goto    $               ; wait forever

        END
```

Example 5b: Detecting a WDT time-out reset

Since, when the watchdog timer times out, the PIC is reset, your program is restarted, in the same way that is was when power was first applied, or after an $\overline{\text{MCLR}}$ reset.

But you may want your program to behave differently, depending on why it was restarted. In particular, if a WDT time-out reset has occurred, you may wish to reset some external equipment to a known state, or perhaps simply turn on an alarm indicator to show that something has gone wrong.

Watchdog timer resets are indicated by the $\overline{\text{TO}}$ bit in the STATUS register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
STATUS	IRP	RP1	RP0	$\overline{\text{TO}}$	$\overline{\text{PD}}$	Z	DC	C

The $\overline{\text{TO}}$ (time-out) bit is cleared to '0' by a WDT time-out reset.

It is set to '1' at power-on, or by entering sleep mode, or execution of the '`clrwdt`' instruction.

Thus, if $\overline{\text{TO}}$ has been cleared, it means that a WDT time-out reset has occurred.

To demonstrate how the $\overline{\text{TO}}$ flag is used, the previous example can be modified, to light a second LED when a watchdog timer reset has occurred, but not when the PIC is first powered on, as follows:

```
;***** Initialisation
        ; configure port
        movlw   ~(1<<nF_LED|1<<nW_LED)  ; configure LED pins as outputs
        banksel TRISIO
        movwf   TRISIO
        ; configure watchdog timer
        movlw   1<<PSA | b'111' ; assign prescaler to WDT (PSA = 1)
                                ; prescale = 128 (PS = 111)
        banksel OPTION_REG      ; -> WDT timeout = 2.3 s
        movwf   OPTION_REG
```

```
;***** Main code
        banksel GPIO
        btfss   STATUS,NOT_TO   ; if WDT timeout has occurred,
        bsf     GPIO,nW_LED     ;   turn on "WDT" LED

        bsf     GPIO,nF_LED     ; turn on "flashing" LED

        DelayMS 1000            ; delay 1s

        banksel GPIO            ; turn off "flashing" LED
        bcf     GPIO,nF_LED

        goto    $               ; wait forever
```

Note that, if, after the watchdog timer has reset the PIC, and the "WDT" LED has been lit, you use the reset button to restart the program, the "WDT" LED will remain lit. This is because a $\overline{\text{MCLR}}$ reset does not affect the $\overline{\text{TO}}$ bit.

Example 5c: Using the `clrwdt` instruction

Of course, normally you will want to avoid WDT time-out resets.

As discussed earlier, to prevent the watchdog timer timing out, simply place a '`clrwdt`' instruction within the main loop.

A watchdog timer period should be selected which is long enough to ensure that the watchdog timer never expires within the loop, unless something is wrong. For example, if your main loop normally completes within 10 ms, but can sometimes take up to 40 ms, you would select a watchdog period of 72 ms (prescale ratio = 1:4) or perhaps 144 ms (prescale = 1:8) to be sure.

To demonstrate that the '`clrwdt`' instruction really does stop the watchdog expiring (if executed often enough), simply include it in the endless loop at the end of the code:

```
loop    clrwdt                  ; clear watchdog timer
        goto    loop            ;   repeat forever
```

If you replace the '`goto $`' line with this "clear watchdog timer" loop, you will find that, after flashing once, the LED will remain off – regardless of the watchdog timer setting.

Example 6: Periodic wake from sleep

The watchdog timer can also be used to wake the PIC from sleep mode.

This is useful in situations where inputs do not need to be responded to instantly, but can be checked periodically. To minimise power consumption, the PIC can sleep most of the time, waking up every so often (say, once per second), checking inputs and, if there is nothing to do, going back to sleep.

Note that a periodic wake-up can be combined with wake-up on pin change; you may for example wish to periodically log the value of a sensor, but also respond immediately to button presses.

If the watchdog timer expires while the PIC is in sleep mode, the device wakes from sleep, and program execution continues with the instruction following `sleep`. No device reset occurs (unlike the WDT wake from sleep in the baseline architecture).

The `sleep` instruction clears the watchdog timer and prescaler. This means that the device will sleep for however long the watchdog timer period is set to (unless another event wakes it before the WDT expires).

To demonstrate how this works, we can simply convert the main code in example 5a into a loop, incorporating a '`sleep`' instruction:

```
loop
        banksel GPIO            ; turn on LED
        bsf     GPIO,nLED

        DelayMS 1000            ; delay 1s

        banksel GPIO            ; turn off LED
        bcf     GPIO,nLED

        sleep                   ; enter sleep mode (until WDT time-out)

        goto    loop            ; repeat forever
```

If you enable the watchdog timer, you'll find that the LED turns on for 1 s, and is then off for around 2 s, before turning on again. And if you measure the current drawn by the PIC, you will find that very little power is consumed while the LED is off, because the PIC is in sleep mode.

On the other hand, if you disable the watchdog timer, the LED will turn on for 1 s, but then turn off forever, because, with the watchdog disabled, the PIC never wakes from sleep.

Complete program

Here is how this new main loop fits into the program presented in example 5a:

```
;*****************************************************************
;                                                                *
;   Description:    Lesson 7, example 6                          *
;                                                                *
;   Demonstrates periodic wake from sleep, using the watchdog timer     *
;                                                                *
;   Turn on LED for 1s, turn off, then sleep                     *
;       LED stays off if watchdog not enabled,                   *
;       flashes (1s on, 2.3s off) if WDT enabled                 *
;                                                                *
;*****************************************************************
;                                                                *
;   Pin assignments:                                             *
;       GP1 - indicator LED                                      *
;                                                                *
;*****************************************************************

    list        p=12F629
    #include    <p12F629.inc>

    #include    <stdmacros-mid.inc>     ; DelayMS - delay in milliseconds

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

    radix       dec

    EXTERN      delay10         ; W x 10ms delay


;***** CONFIGURATION
    #define     WATCHDOG        ; define to enable watchdog timer
```

```
    IFDEF WATCHDOG
                    ; ext reset, no code or data protect, no brownout detect,
                    ; watchdog, power-up timer, 4Mhz int clock
        __CONFIG    _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_ON &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT
    ELSE
                    ; ext reset, no code or data protect, no brownout detect,
                    ; no watchdog, power-up timer, 4Mhz int clock
        __CONFIG    _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT
    ENDIF

; pin assignments
    constant    nLED=1          ; indicator LED on GP1


;*****************************************************************************
RESET   CODE    0x0000          ; processor reset vector
        ; calibrate internal RC oscillator
        call    0x03FF          ; retrieve factory calibration value
        banksel OSCCAL          ;   then update OSCCAL
        movwf   OSCCAL

;***** Initialisation
        ; configure port
        movlw   ~(1<<nLED)      ; configure LED pin as output
        banksel TRISIO
        movwf   TRISIO
        ; configure watchdog timer
        movlw   1<<PSA | b'111' ; assign prescaler to WDT (PSA = 1)
                                ; prescale = 128 (PS = 111)
        banksel OPTION_REG      ; -> WDT timeout = 2.3 s
        movwf   OPTION_REG

;***** Main loop
loop
        banksel GPIO            ; turn on LED
        bsf     GPIO,nLED

        DelayMS 1000            ; delay 1s

        banksel GPIO            ; turn off LED
        bcf     GPIO,nLED

        sleep                   ; enter sleep mode (until WDT time-out)

        goto    loop            ; repeat forever


        END
```

So far in this tutorial series we've focussed on programming and the internal architecture of midrange PICs, but in the next lesson we'll dive into hardware, taking a look at features related to the power supply, such as brown-out detection and the power-up timer, and the available oscillator (clock) options.