Introduction to PIC Programming

Programming Mid-Range PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 4: Interrupt-on-change, Sleep Mode and the Watchdog Timer

This lesson revisits material from mid-range lesson 7, looking at the mid-range PIC architecture's power-saving sleep mode, its ability to generate interrupts and/or wake from sleep when an input changes state and the watchdog timer – generally used to automatically restart a crashed program, but also useful for periodically waking the PIC from sleep, for low-power operation.

One again, the examples are re-implemented using Microchip's XC8 compiler[1] (running in "Free mode"), introduced in lesson 1.

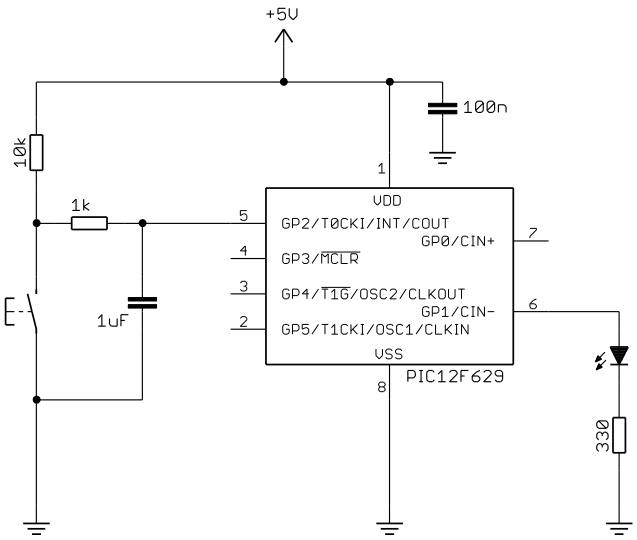
In summary, this lesson covers:

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

Interrupt-on-change

As we saw in mid-range lesson 7, mid-range PICs provide a port change interrupt facility, which, on the 12F629, is available on every GPIO pin.

This feature is similar to the external interrupt facility covered in lesson 3, except that a port change interrupt will be triggered by any change (not just one type of transition) on any of the pins for which it is enabled. This makes it more flexible (being available on more pins), but also more difficult to deal with correctly, as we shall see in the examples in this section.



The first example uses the circuit on the right to demonstrate how to use interrupt-on-change to respond to a single, externally debounced input.

If you are using the Gooligum training board, close jumpers JP3, JP7 and JP12 to enable the 10 kΩ pull-up resistors on $\overline{\text{MCLR}}$ (not shown here)

[1] Available as a free download from www.microchip.com.

and GP2 and the LED on GP1. You can add the 1 µF capacitor (supplied with the board) between GP2 and ground via pins 13 ('GP/RA/RB2') and 16 ('GND') on the 16-pin expansion header. There should be no need to use the solderless breadboard – simply plug the capacitor directly into these header pins.

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

To enable a pin for interrupt-on-change, the corresponding bit must be set in the IOC register. This was done in mid-range lesson 7 by:

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

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

Before actually enabling port change interrupts, it is necessary to either read or write to the port to clear any existing *mismatch condition*, to prevent any false triggering.

The port change interrupt can then enabled by setting the GPIE bit in the INTCON register.

This was done in assembler by:

```
; enable interrupts
movlw   1<<GIE|1<<GPIE  ; enable port change and global interrupts
movwf   INTCON
```

(note that global interrupts are also being enabled here, by setting the GIE bit)

In the interrupt handler, we must clear the port mismatch condition which triggered this interrupt.

In mid-range lesson 7 this was done by reading the port. And (as for all interrupts), we must also clear the corresponding interrupt flag, GPIF:

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

Since we want to toggle the LED on GP1 each time the pushbutton is pressed, but not when it is released, we need to check whether the switch is up or down (this is different from the situation with external interrupts, which are only triggered on one type of transition).

This was implemented in assembler as:

```
; 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
```

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

The shadow register was copied to GPIO in the main loop, as in the earlier examples.

XC8 implementation

Implementing these steps using XC8 is quite straightforward, using techniques we have seen before.

Enabling interrupt-on-change on GP2 is simply:

```
IOCbits.IOC2 = 1;               // enable IOC on GP2 input
```

To enable the port change and global interrupts, we have:

```
// enable interrupts
INTCONbits.GPIE = 1;            // enable port change interrupt
ei();                           // enable global interrupts
```

In the interrupt handler, it is best to explicitly clear the mismatch condition (by reading GPIO) at the start of the routine, instead of relying on this occurring as a side-effect of statements in the body of the handler, which may be changed later.

This can be done by:

```
GPIO;                           // read GPIO to clear mismatch condition
```

"GPIO" is an expression which evaluates to the value of the contents of GPIO, but does nothing with it.

In general, the compiler's optimiser will discard any such "do nothing" statements.

However, GPIO is declared as a 'volatile' variable in the processor header file. We've seen that this qualifier warns the compiler that the value of this variable may change at any time, to prevent the optimiser from eliminating apparently redundant references to it. It also ensures that, when the variable's name is used on its own in this way, the compiler will generate code which reads the variable's memory location and discards the result, which is exactly what we want.

We must also clear the port interrupt flag, to indicate that this interrupt has been serviced:

```
INTCONbits.GPIF = 0;            // clear interrupt flag
```

Finally, we need to check the status of the pushbutton, and toggle the LED only if the button is pressed (meaning that GP2 is low):

```
// toggle LED only on button press
if (!BUTTON)                    // if button is down
    sB_LED = ~sB_LED;           //   toggle LED (via shadow register)
```

Complete program

Here is how these code fragments fit into a working program:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 4, example 1                                 *
*                                                                       *
*   Demonstrates use of interrupt-on-change interrupts                  *
*   (without software debouncing)                                       *
*                                                                       *
*   Toggles LED when pushbutton is pressed (high -> low transition)     *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       GP1 = indicator LED                                             *
*       GP2 = pushbutton (externally debounced, active low)             *
*                                                                       *
************************************************************************/

#include <xc.h>
#include <stdint.h>
```

```
/***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

// Pin assignments
#define sB_LED  sGPIO.GP1           // indicator LED (shadow)
#define BUTTON  GPIObits.GP2        // pushbutton


/***** GLOBAL VARIABLES *****/
volatile union {                    // shadow copy of GPIO
    uint8_t         port;
    struct {
        unsigned    GP0     : 1;
        unsigned    GP1     : 1;
        unsigned    GP2     : 1;
        unsigned    GP3     : 1;
        unsigned    GP4     : 1;
        unsigned    GP5     : 1;
    };
} sGPIO;


/***** MAIN PROGRAM *****/
void main()
{
    //*** Initialisation

    // configure port
    GPIO = 0;                       // start with all LEDs off
    sGPIO.port = 0;                 //   update shadow
    TRISIO = ~(1<<1);               // configure GP1 (only) as an output
    IOCbits.IOC2 = 1;               // enable IOC on GP2 input

    // enable interrupts
    INTCONbits.GPIE = 1;            // enable port change interrupt
    ei();                           // enable global interrupts


    //*** Main loop
    for (;;)
    {
        // continually copy shadow GPIO to port
        GPIO = sGPIO.port;

    }   // repeat forever
}


/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{
    //*** Service port change interrupt
    //
    //  Triggered on any transition on IOC-enabled input pin
    //  caused by externally debounced pushbutton press
    //
    //  Toggles LED on every high -> low transition
    //
```

```
    //  (only port change interrupts are enabled)
    //
    GPIO;                           // read GPIO to clear mismatch condition
    INTCONbits.GPIF = 0;            // clear interrupt flag

    // toggle LED only on button press
    if (!BUTTON)                    // if button is down
        sB_LED = ~sB_LED;           //   toggle LED (via shadow register)
}
```

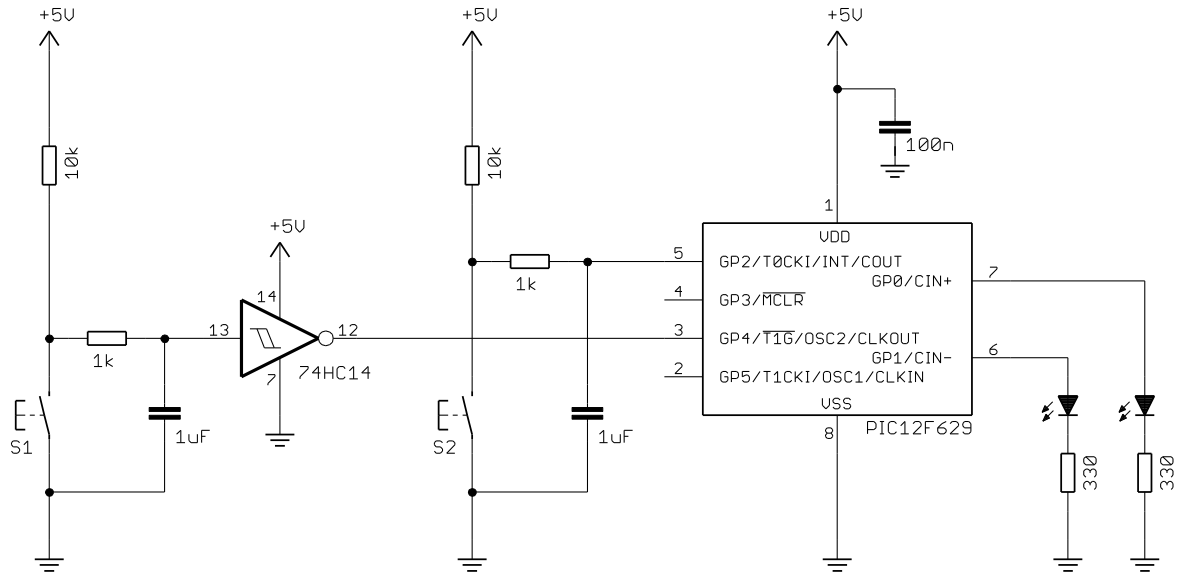
Example 2: Interrupt-on-change (multiple inputs)

This example demonstrates how to handle the situation where interrupt-on-change is enabled on more than one input pin.

The basic difficulty with handling this situation is that there are no flags to indicate which input has changed; the GPIF flag can tell you that at least one pin enabled for IOC has changed, but not which pin it was.

So when a port change interrupt occurs, we need to deduce which pin(s) have changed, by reading GPIO and comparing the current state to the last recorded state. That means that the ISR (where the port change interrupt is handled) needs to keep track of the state of GPIO, and update that "last state" record, every time a change is detected, to be ready for the next time.

We'll use the circuit from the corresponding example in mid-range lesson 7, shown (with the reset switch and pull-up omitted for clarity) below:



If you have the Gooligum training board, you can use the additional components supplied with your board to build this circuit on the solderless breadboard, connecting them to signals on the 16-pin header: GP4 input on pin 3 ('GP/RA/RB4'), GP2 input on pin 13 ('GP/RA/RB2') and ground and +5 V on pins15 ('+V') and 16 ('GND') – see the illustration in mid-range lesson 7. You should also close JP3, JP7, JP11 and JP12 to enable the pull-up resistors on $\overline{\text{MCLR}}$ (not shown here) and GP2 and the LEDs on GP0 and GP1.

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

To simplify the software, both buttons are externally debounced, and since the only Schmitt-trigger GP input on the 12F629 is GP2, an external Schmitt-trigger inverter is used to drive GP4.

Thus, the operation of S1 is inverted with respect to S2; the software will have to take this difference into account.

XC8 implementation

As in the last example, to make the code more maintainable, it is good practice to define symbols represent the pins being used:

```
// Pin assignments
#define sB1_LED sGPIO.GP0   // "button 1 pressed" indicator LED (shadow)
#define sB2_LED sGPIO.GP1   // "button 2 pressed" indicator LED (shadow)
#define nPB1    2           // pushbutton 1 (ext debounce, active low) on GP2
#define nPB2    4           // pushbutton 2 (ext debounce, active high) on GP4
```

Note that the pushbutton pins have been defined as numeric constants, representing pin numbers, because it simplifies the change detection code (see below).

Given that we must keep track of the "last state" of GPIO, to compare with the current state when a port change is detected, and that this state will need to be initialised in the main code, but accessed and updated in the ISR, we need to declare it as a volatile global variable (along with the shadow copy of GPIO, which is also accessed in both the ISR and the main code):

```
/***** GLOBAL VARIABLES *****/
volatile union {                    // shadow copy of GPIO
    uint8_t         port;
    struct {
        unsigned    GP0     : 1;
        unsigned    GP1     : 1;
        unsigned    GP2     : 1;
        unsigned    GP3     : 1;
        unsigned    GP4     : 1;
        unsigned    GP5     : 1;
    };
} sGPIO;

volatile uint8_t    lGPIO;          // last state of GPIO (for change detection)
```

In the initialisation code, we need to initialise and configure the port, before updating the "last state" variable, so that everything is in sync:

```
    // configure port
    GPIO = 0;               // start with all LEDs off
    sGPIO.port = 0;         //   update shadow
    TRISIO = 0b111100;      // configure GP0 and GP1 (only) as outputs
    lGPIO = GPIO;           // update last port state (for pin change detection)
```

Why bother reading GPIO, when we just cleared it?

Why not just write:

```
    GPIO = 0;                   // start with all LEDs off
    sGPIO = 0;                  //   update shadow
    lGPIO = 0;                  //   and last state (NOTE: THIS WILL NOT WORK!)
    TRISIO = 0b111100;      // configure GP0 and GP1 (only) as outputs
```

This approach will not, in general, work, because the value read from an input pin depends on the external signal applied to the pin; if an input pin is being held high externally, clearing the port register (GPIO) won't have any effect – it will still read as a '1'.

So the only way to be sure of the current state of GPIO is to read it.

Having done so, we can enable interrupt-on-change for both inputs, with:

```
IOC = 1<<nPB1|1<<nPB2;  // enable interrupt-on-change on pushbuttons 1 and 2
```

A useful side-effect of reading GPIO is that it clears any existing IOC mismatch condition, so we can now safely go ahead and enable the port change interrupt:

```
// enable interrupts
INTCONbits.GPIE = 1;        // enable port change interrupt
ei();                       // enable global interrupts
```

As usual, the main loop does nothing more than continually update GPIO from the shadow register:

```
//*** Main loop
for (;;)
{
    // continually copy shadow GPIO to port
    GPIO = sGPIO.port;

}   // repeat forever
```

Meanwhile, the ISR updates the shadow copy of GPIO, whenever a port change occurs (triggering an interrupt).

Within the ISR, it is best to take a "snapshot" of the current state of GPIO, and use this to determine which pins have changed, instead of referring back continually to GPIO itself, in case an input changes while the interrupt handler is running (leading to inconsistent results).

So we declare a variable within the ISR function, so hold this current state:

```
void interrupt isr(void)
{
    uint8_t     cGPIO;      // current state of GPIO (used by IOC handler)

    // IOC handler goes here…
}
```

When servicing the port change interrupt, we begin by clearing the interrupt flag, as usual:

```
INTCONbits.GPIF = 0;            // clear interrupt flag
```

There is no need to explicitly clear the IOC mismatch here, because GPIO is read in the very next statement:

```
cGPIO = GPIO;                   // save current GPIO state
```

Next we need to determine which pin(s) have changed, by comparing the current state of GPIO with the last recorded state.

This can be done by XORing the current and last states. Since an XOR results in a '1' only where the inputs differ, the result will be all '0's, except for those bits corresponding to any pins which have changed.

We could write this as:

```
changes = lGPIO ^ cGPIO     // XOR current with last state to detect changes
```

but there is actually no need to introduce another variable; the only time we need to reference the last state (`lGPIO`) is here, to deduce which pins have changed, and, having done so, there is no need to refer back to `lGPIO` again, until it is updated at the end of the ISR.

So, to save data memory, it is possible to write the XOR result back to `lGPIO` with:

```
lGPIO ^= cGPIO;             // XOR with last state to detect changes
```

The `lGPIO` variable will now contain '1's only in bit positions where the current state differs from the last state, corresponding to pins that have changed.

We can then use this to check whether each pushbutton input has changed, for example:

```
if (lGPIO & 1<<nPB1)            // if button 1 changed
{
    // handle button 1 input change…
}
```

But since we only want to toggle the LED when the pushbutton has pressed, we must check not only that the pushbutton input has changed, but that the button is down, which we can do with nested `if` statements:

```
// toggle LED 1 only on button 1 press (active low)
if (lGPIO & 1<<nPB1)            // if button 1 changed
    if (!(cGPIO & 1<<nPB1))     //  and if button 1 is down (low)
    {
        sB1_LED = ~sB1_LED;     //   toggle LED 1 (via shadow register)
    }
```

Alternatively, this can be written as a single `if` statement, using a logical AND expression:

```
// toggle LED 1 only on button 1 press (active low)
if ((lGPIO & 1<<nPB1)           // if button 1 changed
    && (!(cGPIO & 1<<nPB1)))    //  and button 1 is down (low)
    {
        sB1_LED = ~sB1_LED;     //   toggle LED 1 (via shadow register)
    }
```

Either form is acceptable; both generate the same (efficient) code, so which you use only a question of personal programming style.

Looking at these logical expressions, you may conclude that it would also be possible to replace the logical AND ('`&&`') with a bitwise AND ('`&`') and condense the expression to:

```
if (lGPIO & cGPIO & 1<<nPB1)    // if button 1 changed and down
{
    sGPIO ^= 1<<nB1_LED;        //   toggle LED 1 using shadow register
}
```

However, although this works, in terms of program logic (the expressions are, after all, logically the same), it is less clear and generates less efficient code. This is a case where writing more obscure code is counter-productive – it's simply a bad idea.

We can then write a very similar construct for the second pushbutton, but with the logic for testing "button down" inverted because this signal is active high, not low:

```
// toggle LED 2 only on button 2 press (active high)
if ((lGPIO & 1<<nPB2)           // if button 2 changed
    && (cGPIO & 1<<nPB2))       //  and button 2 is down (high)
    {
        sB2_LED = ~sB2_LED;     //   toggle LED 2 (via shadow register)
    }
```

Finally, before exiting the interrupt handler, we must save the current state of GPIO as the new "last state", so that the next input change can be properly detected:

```
    // update last GPIO state (for next time)
    lGPIO = cGPIO;                  // new "last state" = current
```

Complete program

Here is how these code fragments fit together, to form the complete "interrupt-on-change with multiple inputs" example program:

```
/************************************************************************
*   Description:    Lesson 4, 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)          *
*                                                                       *
************************************************************************/

#include <xc.h>
#include <stdint.h>


/***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

// Pin assignments
#define sB1_LED sGPIO.GP0   // "button 1 pressed" indicator LED (shadow)
#define sB2_LED sGPIO.GP1   // "button 2 pressed" indicator LED (shadow)
#define nPB1    2           // pushbutton 1 (ext debounce, active low) on GP2
#define nPB2    4           // pushbutton 2 (ext debounce, active high) on GP4


/***** GLOBAL VARIABLES *****/
volatile union {                    // shadow copy of GPIO
    uint8_t         port;
    struct {
        unsigned    GP0     : 1;
        unsigned    GP1     : 1;
        unsigned    GP2     : 1;
        unsigned    GP3     : 1;
        unsigned    GP4     : 1;
        unsigned    GP5     : 1;
    };
} sGPIO;
```

```
volatile uint8_t    lGPIO;          // last state of GPIO (for change detection)


/***** MAIN PROGRAM *****/
void main()
{
    //*** Initialisation

    // configure port
    GPIO = 0;               // start with all LEDs off
    sGPIO.port = 0;         //   update shadow
    TRISIO = 0b111100;      // configure GP0 and GP1 (only) as outputs
    lGPIO = GPIO;           // update last port state (for pin change detection)
    IOC = 1<<nPB1|1<<nPB2;  // enable interrupt-on-change on pushbuttons 1 and 2

    // enable interrupts
    INTCONbits.GPIE = 1;        // enable port change interrupt
    ei();                       // enable global interrupts


    //*** Main loop
    for (;;)
    {
        // continually copy shadow GPIO to port
        GPIO = sGPIO.port;

    }   // repeat forever
}


/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{
    uint8_t     cGPIO;      // current state of GPIO (used by IOC handler)

    //*** Service port change interrupt
    //
    //  Triggered on any transition on IOC-enabled input pin
    //  caused by externally debounced pushbutton press
    //
    //  Toggles LED1 on every high -> low transition of PB1
    //      and LED2 on every low -> high transition of PB2
    //
    //  (only port change interrupts are enabled)
    //
    INTCONbits.GPIF = 0;            // clear interrupt flag

    // determine which pins have changed
    cGPIO = GPIO;                   // save current GPIO state
                                    //   (GPIO read clears mismatch condition)
    lGPIO ^= cGPIO;                 // XOR with last state to detect changes

    // toggle LED 1 only on button 1 press (active low)
    if ((lGPIO & 1<<nPB1)           // if button 1 changed
        && (!(cGPIO & 1<<nPB1)))    //  and button 1 is down (low)
        {
            sB1_LED = ~sB1_LED;     //   toggle LED 1 (via shadow register)
        }

    // toggle LED 2 only on button 2 press (active high)
    if ((lGPIO & 1<<nPB2)           // if button 2 changed
```

```
        && (cGPIO & 1<<nPB2))       //  and button 2 is down (high)
        {
            sB2_LED = ~sB2_LED;     //   toggle LED 2 (via shadow register)
        }

    // update last GPIO state (for next time)
    lGPIO = cGPIO;                  // new "last state" = current
}
```

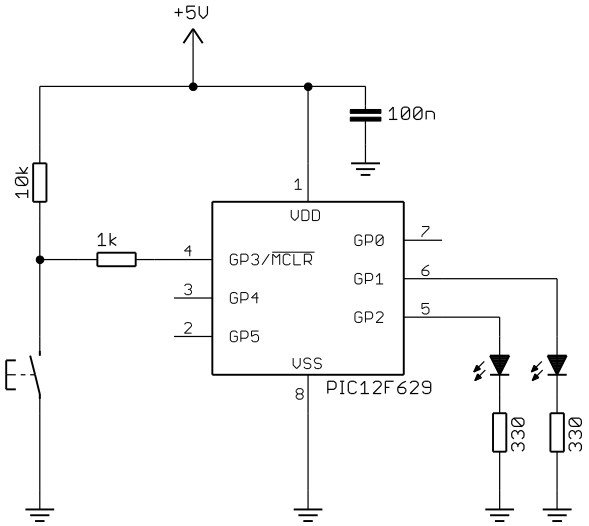
Sleep Mode

As explained in mid-range lesson 7, the mid-range PICs can be placed into a power-saving standby, or sleep mode, using the assembler instruction '`sleep`'.

In this mode, the PIC12F629 will typically draw only a few nanoamps (or less), when all of the power-consuming facilities have been disabled and the output pins are not supplying any current.

This was demonstrated using the circuit on the right.

To implement it using the Gooligum training board, close jumpers JP3, JP12 and JP13 to enable the pull-up resistor on GP3 and the LEDs on GP1 and GP2.



To demonstrate to yourself that power consumption really is reduced when the PIC enters sleep mode, you would have to use an external power supply, instead of using your PICkit 2 or PICkit 3 to power the circuit. You can then place a multimeter in-line with the power supply, to measure the supply current.

The LED on GP1 is initially turned on, and then when the pushbutton is pressed, the LED is turned off (reducing power consumption) before placing the PIC permanently into sleep mode (effectively shutting it down).

The following assembler code was used:

```
        ; turn on LED
        banksel GPIO
        bsf     LED

        ; wait for button press
wait_lo btfsc   BUTTON          ; wait until button low
        goto    wait_lo

        ; go into standby mode
        sleep                   ; enter sleep mode

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

(where '`BUTTON`' and '`LED`' are symbols representing GP3 and GP1 respectively)

XC8 implementation

To place the PIC into sleep mode, XC8 provides a '`SLEEP()`' macro.

It is defined in the "`pic.h`" header file (called from the "`xc.h`" file we've included at the start of each program), as:

```
#define     SLEEP()           asm("sleep")
```

'`asm()`' is a XC8 statement which embeds a single assembler instruction, in-line, in the C source code. But since '`SLEEP()`' is provided as a standard macro, it makes sense to use it, instead of the '`asm()`' statement.

Complete program

The following program shows how the XC8 '`SLEEP()`' macro is used:

```
/************************************************************************
*   Description:    Lesson 4, example 3                                 *
*                                                                       *
*   Demonstrates sleep mode                                             *
*                                                                       *
*   Turn on LED, wait for button pressed, turn off LED, then sleep      *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       GP1 = indicator LED                                             *
*       GP3 = pushbutton (active low)                                   *
*                                                                       *
************************************************************************/

#include <xc.h>


/***** CONFIGURATION *****/
// int reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCIO);

// Pin assignments
#define LED     GPIObits.GP1    // indicator LED on GP1
#define nLED    1               //   (port bit 1)
#define BUTTON  GPIObits.GP3    // pushbutton on GP3 (active low)


/***** MAIN PROGRAM *****/
void main()
{
    //***** Initialisation

    // configure port
    TRISIO = ~(1<<nLED);        // configure LED pin (only) as output


    //***** Main code

    // turn on LED
    LED = 1;

    // wait for button press
    while (BUTTON == 1)         // wait until button low
        ;
```

```
    // go into standby (low power) mode
    LED = 0;                    // turn off LED
    SLEEP();                    // enter sleep mode

    for (;;)                    // (this loop should never execute)
        ;
}
```

Wake-up from sleep

As discussed in mid-range lesson 7, mid-range PICs can be woken from sleep mode in a number of ways:

- 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

Since the PIC's oscillator (clock) does not run in sleep mode, interrupt sources which require the clock to function, such as Timer0, cannot be used wake the device from sleep. However, external (INT pin) and port change interrupts (and others that we will see in later lessons) can be used to wake up a mid-range PIC.

The following example looks 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 the INT pin.

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

In baseline assembler lesson 7, we saw that the baseline architecture includes a "wake-up on change" feature. Its mid-range equivalent is the interrupt-on-change facility, introduced above.

"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, or, if using XC8, the statement following the '`SLEEP()`' macro.

If port change interrupts are enabled (GPIE = 1) and global interrupts are enabled (GIE = 1), if an IOC-enabled input changes 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" – available in XC8 as a '`NOP()`' macro) instruction immediately following the `sleep` instruction.

If you are using other interrupts (such as Timer0) in your program, but don't want to have to deal with executing the ISR as the device wakes from sleep, simply disable interrupts (clear GIE – which can be done in XC8 with the '`di()`' macro) 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.

This means that any switch used as a "soft" on/off switch must be debounced both as soon as the PIC has been restarted (in case the switch is still bouncing) and prior to entering sleep mode (in case a bounce causes the PIC to wake).

In this example, we want to wake-up the PIC and turn on an LED when the button is pressed, and then turn off the LED and place the PIC into sleep mode when the button is pressed again.

The necessary sequence is:

```
do
    turn on LED
    wait for stable button high
    wait for button low
    turn off LED
    wait for stable button high
    clear GPIF
    sleep
forever     // repeat from the beginning
```

XC8 implementation

The following code, which uses the debounce macro defined in lesson 2, implements the sequence of steps given above:

```
    /*** Initialisation ***/

    // configure port
    TRISIO = ~(1<<nLED);        // configure LED pin (only) as output

    // configure Timer0 (for DbnceHi() macro)
    OPTION_REGbits.T0CS = 0;        // select timer mode
    OPTION_REGbits.PSA = 0;         // assign prescaler to Timer0
    OPTION_REGbits.PS = 0b111;      // prescale = 256
                                    // -> increment every 256 us

    // configure interrupt-on-change
    IOC |= 1<<nBUTTON;          // enable IOC on pushbutton input
    INTCONbits.GPIE = 1;        // enable wake-up (interrupt) on port change


    /*** Main loop ***/
    for (;;)
    {
        // turn on LED
        LED = 1;

        // wait for stable button high
        // (in case it is still bouncing after wakeup)
        DbnceHi(BUTTON);

        // wait for button press
        while (BUTTON == 1)         // wait until button low
            ;

        // go into standby (low power) mode
        LED = 0;                    // turn off LED
        DbnceHi(BUTTON);            // wait for stable button release
```

```
        INTCONbits.GPIF = 0;        // clear port change interrupt flag
        SLEEP();                    // enter sleep mode
    }
}
```

(the labels '`LED`', '`nLED`', '`BUTTON`' and '`nBUTTON`' are defined earlier in the program, as usual)

This code does essentially the same thing as the "toggle an LED" programs developed in lessons 1 and 2, except that in this case, when the LED is off, the PIC is drawing negligible power.

Watchdog Timer

As described in mid-range lesson 7, the watchdog timer is free-running counter which, if enabled, operates independently of the program running on the PIC. It is typically used to avoid program crashes, where your application enters a state it will never return from, such as a loop waiting for a condition that will never occur. If the watchdog timer overflows, the PIC is reset, restarting your program – hopefully allowing it to recover and operate normally.

To avoid this "WDT reset" from occurring, your program must periodically reset, or clear, the watchdog timer before it overflows.

This watchdog time-out period on the mid-range PICs is nominally 18 ms, but can be extended to a maximum of 2.3 seconds by assigning the prescaler to the watchdog timer (in which case the prescaler is no longer available for use with Timer0).

The watchdog timer can also be used to regularly wake the PIC from sleep mode, perhaps to sample and log an environmental input (say a temperature sensor), for low power operation.

Example 5a: Enabling the watchdog timer and detecting WDT resets

To illustrate 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).

We saw in mid-range lesson 7 that the watchdog timer is controlled by the WDTE bit in the processor configuration word: setting WDTE to '1' enables the watchdog timer.

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.

The assembler examples in that lesson included the following construct, to make it easy to select whether the watchdog timer is enabled or disabled when the code is built:

```
    #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
```

To set the watchdog time-out period to the maximum of 2.3 seconds, the prescaler was assigned to the watchdog timer, with a prescale ratio of 1:128 (18 ms × 128 = 2.3 s), 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
```

If you want your program to behave differently when restarted by a watchdog time-out, test the $\overline{TO}$ flag in the STATUS register: it is cleared to '0' only when a WDT reset has occurred.

The example in mid-range lesson 7 used this approach to turn on an "error" LED, to indicate if a restart was due to a WDT reset:

```
;***** Initialisation
        ; configure port
        movlw   ~(1<<nF_LED|1<<nW_LED)  ; configure LED pins as outputs
        banksel TRISIO
        movwf   TRISIO
        ; configure watchdog timer prescaler
        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
```

XC8 implementation

Since the watchdog timer is controlled by a configuration bit, the only change we need to make to enable it is to use a different `__CONFIG()` statement, with the symbol '`WDTE_ON`' replacing '`WDTE_OFF`'.

A construct very similar to that in the assembler example can be used to select between processor configurations:

```
#ifdef WATCHDOG
    // ext reset, no code or data protect, no brownout detect,
    // watchdog, power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
         WDTE_ON & PWRTE_OFF & FOSC_INTRCIO);
#else
    // ext reset, no code or data protect, no brownout detect,
    // no watchdog, power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
         WDTE_OFF & PWRTE_OFF & FOSC_INTRCIO);
#endif
```

Assigning the prescaler to the watchdog timer and selecting a prescale ratio of 128:1 is done by:

```
    OPTION_REGbits.PSA = 1;             // assign prescaler to WDT
    OPTION_REGbits.PS = 0b111;          // prescale = 128
                                        // -> WDT timeout = 2.3 s
```

To check for a WDT timeout reset, the $\overline{TO}$ flag can be tested directly, using:

```
    if (!STATUSbits.nTO)        // if WDT timeout has occurred,
        W_LED = 1;              //   turn on "error" LED
```

Note that the test condition is inverted, using '!', since this flag is "active" when clear.

Complete program

Here is the complete program, showing how the above code fragments are used:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 4, example 5a                                *
*                                                                       *
*   Demonstrates watchdog timer                                         *
*       plus differentiation of WDT time-out from POR reset             *
*                                                                       *
*   Turn on LED for 1s, turn off, then enter endless loop               *
*   If WDT enabled, processor resets after 2.3s                         *
*   Turns on WDT LED to indicate WDT reset                              *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       GP1 = flashing LED                                              *
*       GP2 = WDT-reset indicator LED                                   *
*                                                                       *
************************************************************************/

#include <xc.h>

#define _XTAL_FREQ  4000000     // oscillator frequency for _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 enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
         WDTE_ON & PWRTE_OFF & FOSC_INTRCIO);
#else
    // ext reset, no code or data protect, no brownout detect,
    // no watchdog, power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
         WDTE_OFF & PWRTE_OFF & FOSC_INTRCIO);
#endif

// Pin assignments
#define F_LED   GPIObits.GP1    // "flashing" LED on GP1
#define nF_LED  1               //   (port bit 1)
#define W_LED   GPIObits.GP2    // WDT LED to indicate WDT time-out reset
#define nW_LED  2               //   (port bit 2)
```

```
/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation ***/

    // configure port
    TRISIO = ~(1<<nF_LED|1<<nW_LED);    // configure LED pins as outputs

    // configure watchdog timer
    OPTION_REGbits.PSA = 1;             // assign prescaler to WDT
    OPTION_REGbits.PS = 0b111;          // prescale = 128
                                        // -> WDT timeout = 2.3 s


    /*** Main code ***/

    // test for WDT-timeout reset
    if (!STATUSbits.nTO)        // if WDT timeout has occurred,
        W_LED = 1;              //   turn on "error" LED

    // flash LED
    F_LED = 1;                  // turn on "flash" LED
    __delay_ms(1000);           // delay 1 sec
    F_LED = 0;                  // turn off "flash" LED

    // wait forever
    for (;;)
        ;
}
```

Example 5b: Clearing the watchdog timer

Normally, you will want to prevent watchdog timer overflows; a WDT reset should only happen when something has gone wrong.

To avoid WDT resets, the watchdog timer has to be regularly cleared. This is typically done by inserting a '`clrwdt`' instruction within the program's "main loop", and within any subroutine which may, in normal operation, not complete within the watchdog timer period.

To demonstrate the effect of clearing the watchdog timer, a '`clrwdt`' instruction was added into the endless loop in the example in mid-range lesson 7:

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

        DelayMS 1000            ; delay 1 sec

        banksel GPIO            ; turn off LED
        bcf     GPIO,LED

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

With the '`clrwdt`' instruction in place, the watchdog timer never overflows, so the PIC is never restarted by a WDT reset, and the LED remains turned off (until the power is cycled), whether the watchdog timer is enabled or not.

XC8 implementation

XC8 provides a '`CLRWDT()`' macro, defined in the "`pic.h`" header file as:

```
#define     CLRWDT()    asm("clrwdt")
```

That is, the '`CLRWDT()`' macro simply inserts a '`clrwdt`' instruction into the code.

Using this macro, the assembler code above can be implemented with XC8 as follows:

```
    LED = 1;                    // turn on LED

    __delay_ms(1000);           // delay 1 sec

    LED = 0;                    // turn off LED

    for (;;)                    // repeatedly clear watchdog timer forever
        CLRWDT();
```

Example 6: Periodic wake from sleep

As explained in mid-range lesson 7, the watchdog timer is can also be used to periodically wake the PIC from sleep mode, typically to read some inputs, take some action and then return to sleep mode, saving power. This can be combined with wake-up on pin change, allowing immediate response to some inputs, such as a button press, while periodically checking others.

To illustrate this, the example in mid-range lesson 7 converted the main code in the first watchdog timer example into a loop, incorporating the '`sleep`' instruction:

```
main_loop
        banksel GPIO            ; turn on LED
        bsf     LED

        DelayMS 1000            ; delay 1 sec

        banksel GPIO            ; turn off LED
        bcf     LED

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

        goto    main_loop       ; repeat forever
```

With the watchdog timer enabled, with a period of 2.3 s, the LED is on for 1 s, and then off for 1.3 s, as in the earlier example. But this time the PIC is in sleep mode while the LED is off, conserving power.

XC8 implementation

In a similar way, we can convert the main code in example 5, above, into a loop – dropping the WDT timeout test, and adding a `SLEEP()` macro:

```
    for (;;)
    {
        LED = 1;                    // turn on LED

        __delay_ms(1000);           // delay 1 sec

        LED = 0;                    // turn off LED

        SLEEP();                    // enter sleep mode (until WDT time-out)
    }
```

Complete program

Here is how this new main loop fits into the code:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 4, 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                                             *
*                                                                       *
************************************************************************/

#include <xc.h>

#define _XTAL_FREQ  4000000     // oscillator frequency for _delay_ms()


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

#ifdef WATCHDOG
    // ext reset, no code or data protect, no brownout detect,
    // watchdog, power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
         WDTE_ON & PWRTE_OFF & FOSC_INTRCIO);
#else
    // ext reset, no code or data protect, no brownout detect,
    // no watchdog, power-up timer enabled, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF &
         WDTE_OFF & PWRTE_OFF & FOSC_INTRCIO);
#endif

// Pin assignments
#define LED     GPIObits.GP1    // indicator LED on GP1
#define n_LED   1               //   (port bit 1)


/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation ***/

    // configure port
    TRISIO = ~(1<<n_LED);           // configure LED pin (only) as output

    // configure watchdog timer
    OPTION_REGbits.PSA = 1;             // assign prescaler to WDT
    OPTION_REGbits.PS = 0b111;          // prescale = 128
                                        // -> WDT timeout = 2.3 s


    /*** Main loop ***/
    for (;;)
```

```
    {
        LED = 1;                    // turn on LED

        __delay_ms(1000);           // delay 1 sec

        LED = 0;                    // turn off LED

        SLEEP();                    // enter sleep mode (until WDT time-out)
    }
}
```

Summary

We have seen in this lesson that the interrupt-on-change, sleep mode, wake-up on change, and watchdog timer features of the mid-range PIC architecture can be configured and used effectively in C programs, using the XC8 compiler.

The next lesson revisits material from mid-range lesson 8, briefly covering some of the hardware-related features of the 12F629 (and most other mid-range PICs), such as brown-out detection and the available oscillator (clock) options.