Introduction to PIC Programming

Programming Baseline PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 4: Sleep Mode and the Watchdog Timer

Continuing the series on C programming, this lesson revisits material from baseline lesson 7, which examined the baseline PIC architecture's power-saving sleep mode, its ability to wake from sleep when an input changes 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. As before, selected examples from that lesson are re-implemented using Microchip's XC8 compiler (running in "Free mode") and CCS PCB[1], introduced in lesson 1.

Baseline assembler lesson 7 also described the various clock, or oscillator, configurations available for the PIC12F508/509 – a topic which does not really need a separate treatment for C, since the programming techniques needed to implement the examples from that lesson have already been covered in lessons 1 to 3. Only the PIC configuration is different, so this lesson includes a table listing the corresponding configuration word settings between MPASM, XC8 and CCS PCB.

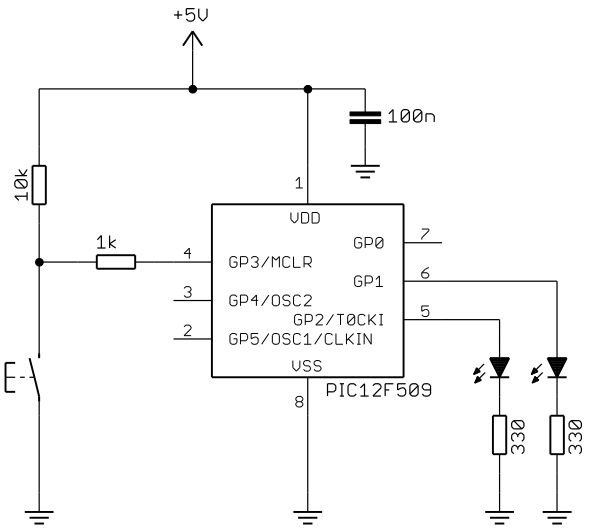
In summary, this lesson covers:

- Sleep mode (power down)
- Wake-up on change (power up on input change)
- The watchdog timer, including periodic wake from sleep
- Configuration word settings

with examples for XC8 and CCS PCB.

Circuit Diagram

The examples in this lesson use the circuit shown on the right, consisting of a PIC12F509 and 100 nF bypass capacitor, with LEDs on GP1 and GP2, and a pushbutton switch on GP3.



If you have the Gooligum baseline training board, connect jumpers JP3, JP12 and JP13 to enable the pull-up resistor on GP3 and the LEDs on GP1 and GP2. Or, if you are using Microchip's Low Pin Count Demo Board, you will need to connect LEDs to GP1 and GP2, as described in baseline lesson 1.

However, if you want to be able to see how the power consumption is reduced when the PIC is placed into sleep mode, you should use an external power supply,

[1] XC8 is available as a free download from www.microchip.com, and CCS PCB is bundled for free with MPLAB 8

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.

Sleep Mode

As explained in baseline lesson 7, the assembler instruction for placing the PIC into sleep mode is '`sleep`'.

This was demonstrated by the following code, which turns on an LED, waits for a pushbutton press, and then turns off the LED (saving power) before placing the PIC permanently into sleep mode (effectively shutting it down):

```
        ; turn on LED
        bsf     LED

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

        ; go into standby (low power) mode
        bcf     LED             ; turn off LED
        sleep                   ; enter sleep mode

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

XC8

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 XC8 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 1                                 *
*                                                                       *
*   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 watchdog, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & WDT_OFF & OSC_IntRC);
```

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


/***** MAIN PROGRAM *****/
void main()
{
    //*** Initialisation
    // configure port
    TRIS = ~(1<<nLED);          // configure LED pin (only) as an 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)
        ;
}
```

CCS PCB

Consistent with CCS' stated approach of allowing most tasks to be performed through built-in functions, the PCB compiler provides a function for entering sleep mode: '`sleep()`'.

Unlike the XC8 version, this is a built-in function, not a macro. But it's used the same way.

Complete program

Here is the CCS PCB version of the "sleep after pushbutton press" program:

```
************************************************************************
*                                                                      *
*   Description:    Lesson 4, example 1                                *
*                                                                      *
*   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 <12F509.h>

#define GP0 PIN_B0              // define GP pins
```

```
#define GP1 PIN_B1
#define GP2 PIN_B2
#define GP3 PIN_B3
#define GP4 PIN_B4
#define GP5 PIN_B5


/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC clock
#fuses NOMCLR,NOPROTECT,NOWDT,INTRC

// Pin assignments
#define LED     GP1             // Indicator LED
#define BUTTON  GP3             // Pushbutton (active low)


/***** MAIN PROGRAM *****/
void main()
{
    // turn on LED
    output_high(LED);

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

    // go into standby (low power) mode
    output_low(LED);                // turn off LED
    sleep();                        // enter sleep mode

    while (TRUE)                    // (this loop should never execute)
        ;
}
```

Wake-up on Change

We saw in baseline assembler lesson 7 that, if the $\overline{\text{GPWU}}$ bit in the OPTION register is cleared, the PIC12F509 will come out of sleep ("wake-up"), if any of the GP0, GP1 or GP3 inputs change.

Note that, in the baseline PIC architecture, wake-up on change can only be enabled on certain pins, and that it is either enabled for all of these pins or for none of them; it is not individually selectable.

This feature can be used in low-power applications, where the PIC spends most of the time sleeping (saving power), waking only to respond to external events which lead to an input change. It's also useful when designing devices with a "soft" on/off feature, such as the Gooligum Electronics "Hangman" project, which is based on a baseline PIC (the 16F505).

Note also that it's important to read any input pins configured for wake on change, and to ensure that they are stable (by debouncing any switches) just prior to entering sleep mode, to avoid the PIC immediately waking up.

Baseline PICs restart when they wake from sleep, recommencing execution at the reset vector (0x000), in the same way they do when first powered on (power-on reset) or following an external reset on $\overline{\text{MCLR}}$.

If the reset was due to a wake on change, it may be necessary to debounce whichever input changed, to avoid the program responding to spurious input transitions from the switch bounce. You could perform this debounce "just in case", regardless of why the PIC (re)started.

But in some cases you'll want your program to behave differently if it was restarted by a wake on change, can be done by testing the GPWUF flag in the STATUS register. GPWUF is set to '1' only if a wake on change reset has occurred.

These concepts were demonstrated in baseline lesson 7, through an example similar to that in the sleep mode section, above, and using the same circuit. One of the LEDs is turned on and then, when the pushbutton is pressed, it is turned off and the PIC is put into sleep mode. But in this example, wake on change is enabled, so that when the pushbutton is pressed again (changing the input on GP3), the program restarts and the LED is turned on again. If GPWUF is set, a second LED is lit, to indicate that a wake on change happened.

This was implemented in assembler as:

```
        ; turn on LED
        bsf     LED

        ; test for wake-on-change reset
        btfss   STATUS,GPWUF    ; if wake-up on change has occurred,
        goto    wait_lo
        bsf     WAKE            ;   turn on wake-up indicator
        DbnceHi BUTTON          ;   wait for button to stop bouncing

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

        ; go into standby (low power) mode
        clrf    GPIO            ; turn off LEDs

        DbnceHi BUTTON          ; wait for stable button release

        sleep                   ; enter sleep mode
```

XC8

To enable wake-up on change using XC8, simply ensure that the $\overline{\text{GPWU}}$ bit in the OPTION register is cleared, for example:

```
    OPTION = 0b01000111;        // configure wake-up on change and Timer0:
             //0-------             enable wake-up on change (/GPWU = 0)
             //--0-----             timer mode (T0CS = 0)
             //----0---             prescaler assigned to Timer0 (PSA = 0)
             //-----111             prescale = 256 (PS = 111)
             //                     -> increment every 256 us
```

Testing the GPWUF flag is simple; it is defined as a bit-field in the header files provided with the compiler (as all the special function register bit are), and can be accessed directly:

```
    if (STATUSbits.GPWUF)       // if wake on change has occurred,
    {
        WAKE = 1;               //   turn on wake-up indicator
        DbnceHi(BUTTON);        //   wait for button to stop bouncing
    }
```

The test could instead be written more explicitly as:

```
    if (STATUSbits.GPWUF == 1) { … }    // if wake on change has occurred...
```

But it's considered quite acceptable, and perfectly clear, to leave out the '`== 1`' when testing a flag bit.

Complete program

Here is how the above fragments fit into the program:

```
/************************************************************************
*   Description:    Lesson 4, example 2                                 *
*                                                                       *
*   Demonstrates wake-up on change                                      *
*       plus differentiation from POR reset                             *
*                                                                       *
*   Turn on LED after each reset                                        *
*   Turn on WAKE LED only if reset was due to wake on change            *
*   then wait for button press, turn off LEDs, debounce, then sleep     *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       GP1 = on/off indicator LED                                      *
*       GP2 = wake-on-change indicator LED                              *
*       GP3 = pushbutton switch (active low)                            *
*                                                                       *
************************************************************************/

#include <xc.h>

#include "stdmacros-HTC.h"  // DbnceHi() - debounce switch, wait for high
                            // Requires: TMR0 at 256us/tick

/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & WDT_OFF & OSC_IntRC);

// Pin assignments
#define LED     GPIObits.GP1    // LED to turn on/off
#define nLED    1               //   (port bit 1)
#define WAKE    GPIObits.GP2    // indicates wake on change condition
#define nWAKE   2               //   (port bit 2)
#define BUTTON  GPIObits.GP3    // Pushbutton (active low)


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

    // configure port
    GPIO = 0;                   // start with both LEDs off
    TRIS = ~(1<<nLED|1<<nWAKE); // configure LED pins as outputs

    // configure wake-on-change and timer
    OPTION = 0b01000111;        // configure wake-up on change and Timer0:
             //0-------             enable wake-up on change (/GPWU = 0)
             //--0-----             timer mode (T0CS = 0)
             //----0---             prescaler assigned to Timer0 (PSA = 0)
             //-----111             prescale = 256 (PS = 111)
             //                     -> increment every 256 us


    //***** Main code

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

```
    // test for wake-on-change reset
    if (STATUSbits.GPWUF)       // if wake on change has occurred,
    {
        WAKE = 1;               //   turn on wake-up indicator
        DbnceHi(BUTTON);        //   wait for button to stop bouncing
    }

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

    // go into standby (low power) mode
    GPIO = 0;                   // turn off both LEDs

    DbnceHi(BUTTON);            // wait for stable button release

    SLEEP();                    // enter sleep mode
}
```

CCS PCB

Although the CCS PCB compiler provides built-in functions to perform most tasks, *there are no built-in functions for explicitly enabling wake on change, or for detecting a wake on change reset*.

We saw in lesson 2 that weak pull-ups are enabled implicitly whenever Timer0 is configured, and that the only way to disable weak pull-ups is to use the `setup_counters()` function with an additional '`DISABLE_PULLUPS`' symbol.

Similarly, wake-up on change is enabled implicitly whenever Timer0 is configured, whether you use `setup_timer_0()` or `setup_counters()`.

To setup the timer without enabling wake-up on change, you must use the `setup_counters()` function with the '`DISABLE_WAKEUP_ON_CHANGE`' symbol ORed with the second parameter.

For example:

```
setup_counters(RTCC_INTERNAL,RTCC_DIV_1|DISABLE_WAKEUP_ON_CHANGE);
```

CCS PCB does not provide any built-in function which can be used to detect that a wake-up on change reset has occurred. Although the compiler does provide a '`restart_cause()`' function, which returns a value indicating the cause of the last reset, this does not encompass wake on change resets on the baseline PICs. The "12F509.h" header file does define the symbol '`PIN_CHANGE_FROM_SLEEP`', which is presumably intended to be used in detecting a wake-up on pin change, but it is not a valid return code from the '`restart_cause()`' function.

So to detect a wake on change reset, we need to use the `#bit` directive, introduced in lesson 3, to allow access to the GPWUF flag, as follows:

```
#bit GPWUF = 0x03.7         // GPWUF flag in STATUS register
```

This flag can then be tested directly, in the same way as we did with XC8:

```
    if (GPWUF)                  // if wake on change has occurred,
    {
        output_high(WAKE);      //   turn on wake-up indicator
        DbnceHi(BUTTON);        //   wait for stable button high
    }
```

Complete program

The following listing shows how these fragments fit into the "wake-up on change demo" program:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 4, example 2                                 *
*                                                                       *
*   Demonstrates wake-up on change                                      *
*       plus differentiation from POR reset                             *
*                                                                       *
*   Turn on LED after each reset                                        *
*   Turn on WAKE LED only if reset was due to wake on change            *
*   then wait for button press, turn off LEDs, debounce, then sleep     *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       GP1 = on/off indicator LED                                      *
*       GP2 = wake-on-change indicator LED                              *
*       GP3 = pushbutton switch (active low)                            *
*                                                                       *
************************************************************************/

#include <12F509.h>

#define GP0 PIN_B0              // define GP pins
#define GP1 PIN_B1
#define GP2 PIN_B2
#define GP3 PIN_B3
#define GP4 PIN_B4
#define GP5 PIN_B5

#bit GPWUF = 0x03.7         // GPWUF flag in STATUS register

#include "stdmacros-CCS.h"  // DbnceHi() - debounce switch, wait for high
                            // Requires: TMR0 at 256us/tick

/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC clock
#fuses NOMCLR,NOPROTECT,NOWDT,INTRC

// Pin assignments
#define LED     GP1             // LED to turn on/off
#define WAKE    GP2             // indicates wake on change condition
#define BUTTON  GP3             // Pushbutton (active low)


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

    // configure port
    output_b(0);                // start with both LEDs off

    // configure wake-on-change and timer
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_256);  // enable wake-up on change
                                                // configure Timer0:
                                                //  timer mode, prescale = 256
                                                //  -> increment every 256 us
```

```
    //*** Main code

    // turn on LED
    output_high(LED);           // turn on LED

    // test for wake-on-change reset
    if (GPWUF)                  // if wake on change has occurred,
    {
        output_high(WAKE);      //   turn on wake-up indicator
        DbnceHi(BUTTON);        //   wait for button to stop bouncing
    }

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

    // go into standby (low power) mode
    output_b(0);                // turn off both LEDs

    DbnceHi(BUTTON);            // wait for stable button release

    sleep();                    // enter sleep mode
}
```

Watchdog Timer

As described in baseline assembler lesson 7, the watchdog timer is free-running counter which, if enabled, operates independently of any 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 baseline 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.

The examples in this section illustrate these concepts.

Enabling the watchdog timer and detecting WDT resets

We saw in baseline assembler 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.

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 protect, watchdog, int RC clock
            __CONFIG    _MCLRE_ON & _CP_OFF & _WDT_ON & _IntRC_OSC
        ELSE
                    ; ext reset, no code protect, no watchdog, int RC clock
            __CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _IntRC_OSC
        ENDIF
```

To select the maximum watchdog time-out period of 2.3 seconds, the prescaler was assigned to the watchdog timer (by setting the PSA bit in OPTION), with a prescale ratio of 128:1 (18 ms × 128 = 2.3 s), by:

```
movlw   1<<PSA | b'111'     ; prescaler assigned to WDT (PSA = 1)
                            ; prescale = 128 (PS = 111)
option                      ; -> WDT period = 2.3 s
```

To demonstrate the effect of the watchdog timer, an LED is turned on for 1 second, and then turned off, before the program enters an endless loop. Without the watchdog timer, the LED would remain off, until the power is cycled. But if the watchdog timer is enabled, a WDT reset will occur after 2.3 seconds, restarting the program, lighting the LED again. The LED will be seen to flash – on for 1 s, with a period of 2.3 s.

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 baseline assembler lesson 7 used this approach to turn on an "error" LED, to indicate if a restart was due to a WDT reset:

```
; test for WDT-timeout reset
btfss   STATUS,NOT_TO       ; if WDT timeout has occurred,
bsf     WDT                 ;   turn on "error" LED

; flash LED
bsf     LED                 ; turn on "flash" LED
DelayMS 1000                ; delay 1 sec
bcf     LED                 ; turn off "flash" LED

; wait forever
goto    $
```

XC8

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 '`WDT_ON`' replacing '`WDT_OFF`'.

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

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

#ifdef WATCHDOG
    // ext reset, no code protect, watchdog, int RC clock
    __CONFIG(MCLRE_ON & CP_OFF & WDT_ON & OSC_IntRC);
#else
     // ext reset, no code protect, no watchdog, int RC clock
    __CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & OSC_IntRC);
#endif
```

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

```
    OPTION = PSA | 0b111;       // prescaler assigned to WDT (PSA = 1)
                                // prescale = 128 (PS = 111)
                                // -> WDT period = 2.3 s
```

The symbol '`PSA`' is defined in the header files provided with XC8.

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

```
    if (!STATUSbits.nTO)        // if WDT timeout has occurred,
        WDT = 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 3a                                *
*                                                                       *
*   Demonstrates watchdog timer                                         *
*       plus differentiation from POR reset                             *
*                                                                       *
*   Turn on LED for 1 s, turn off, then enter endless loop              *
*   If enabled, WDT timer restarts after 2.3 s -> LED flashes           *
*   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 protect, watchdog, int RC clock
    __CONFIG(MCLRE_ON & CP_OFF & WDT_ON & OSC_IntRC);
#else
     // ext reset, no code protect, no watchdog, int RC clock
    __CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & OSC_IntRC);
#endif

// Pin assignments
#define LED     GPIObits.GP1    // LED to flash
#define nLED    1               //   (port bit 1)
#define WDT     GPIObits.GP2    // watchdog timer reset indicator
#define nWDT    2               //   (port bit 2)


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

    // configure port
    GPIO = 0;                   // start with all LEDs off
    TRIS = ~(1<<nLED|1<<nWDT);  // configure LED pins as outputs
```

```
    // configure watchdog timer
    OPTION = PSA | 0b111;       // prescaler assigned to WDT (PSA = 1)
                                // prescale = 128 (PS = 111)
                                // -> WDT period = 2.3 s

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

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

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

CCS PCB

To enable the watchdog timer, simply replace the symbol '`NOWDT`' with '`WDT`' in the `#fuses` statement.

Once again, we can use a conditional compilation construct to allow the watchdog to be enabled or disabled when building the code:

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

#ifdef WATCHDOG
    // ext reset, no code protect, watchdog, int RC clock
    #fuses MCLR,NOPROTECT,WDT,INTRC
#else
     // ext reset, no code protect, no watchdog, int RC clock
    #fuses MCLR,NOPROTECT,NOWDT,INTRC
#endif
```

Unlike the situation for enabling and detecting wake-up on change, the CCS PCB compiler provides built-in functions for setting up the watchdog timer and detecting that a WDT reset has occurred.

Although it is possible to use the '`setup_counters()`' function to setup the watchdog timer, CCS has de-emphasised its use, in favour of the more specific '`setup_wdt()`'.

The `setup_wdt()` function takes a single parameter, which on the baseline PICs specifies the watchdog timeout period, from '`WDT_18MS`' (18 ms), '`WDT_36MS`' (36 ms), '`WDT_72MS`' (72 ms), etc., through to '`WDT_2304MS`' (2.3 s).

So in this example we have:

```
    setup_wdt(WDT_2304MS);                  // WDT period = 2.3 s)
```

As mentioned above, one of the available built-in functions is '`restart_cause()`', which returns a value indicating why the PIC was (re)started. Although it doesn't accommodate wake-up on change resets, it does correctly detect WDT resets, in which case it returns the value corresponding to '`WDT_TIMEOUT`' (a symbol defined in the "12F509.h" header file). For example:

```
    if (restart_cause() == WDT_TIMEOUT)     // if WDT timeout has occurred,
        output_high(WDT);                   //   turn on "error" LED
```

There is, however, one complicating factor: `setup_wdt()` has the side effect of resetting the $\overline{\text{TO}}$ flag, which the `restart_cause()` function relies on to determine whether a WDT timeout had occurred.

That is, if `setup_wdt()` is called before `restart_cause()`, the information about why the restart had happened is lost. Therefore, it is important to call `restart_cause()` before `setup_wdt()`, as in the following program.

Complete program

Here is how the code fits together, when using CCS PCB:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 4, example 3a                                *
*                                                                       *
*   Demonstrates watchdog timer                                         *
*       plus differentiation from POR reset                             *
*                                                                       *
*   Turn on LED for 1 s, turn off, then enter endless loop              *
*   If enabled, WDT timer restarts after 2.3 s -> LED flashes           *
*   Turns on WDT LED to indicate WDT reset                              *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       GP1 = flashing LED                                              *
*       GP2 = WDT-reset indicator LED                                   *
*                                                                       *
************************************************************************/

#include <12F509.h>

#define GP0 PIN_B0              // define GP pins
#define GP1 PIN_B1
#define GP2 PIN_B2
#define GP3 PIN_B3
#define GP4 PIN_B4
#define GP5 PIN_B5

#use delay (clock=4000000)      // oscillator frequency for delay_ms()


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

#ifdef WATCHDOG
    // ext reset, no code protect, watchdog, int RC clock
    #fuses MCLR,NOPROTECT,WDT,INTRC
#else
     // ext reset, no code protect, no watchdog, int RC clock
    #fuses MCLR,NOPROTECT,NOWDT,INTRC
#endif

// Pin assignments
#define LED     GP1             // LED to flash
#define WDT     GP2             // watchdog timer reset indicator


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

```
    // configure port
    output_b(0);                            // start with both LEDs off

    // test for WDT-timeout reset
    // (note: must be done before initialising watchdog timer)
    if (restart_cause() == WDT_TIMEOUT)     // if WDT timeout has occurred,
        output_high(WDT);                   //   turn on "error" LED

    // configure watchdog timer
    setup_wdt(WDT_2304MS);                  // WDT period = 2.3 s


    //***** Main code
    // flash LED
    output_high(LED);                       // turn on "flash" LED
    delay_ms(1000);                         // delay 1 sec
    output_low(LED);                        // turn off "flash" LED

    // wait forever
    while (TRUE)
        ;
}
```

Clearing the watchdog timer

The previous example shows what happens when the watchdog timer overflows, but of course most of the time, during "normal" program operation, we want to prevent that from happening; a WDT reset should only occur when something has gone wrong.

As mentioned above, to avoid overflows, 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 baseline assembler lesson 7:

```
;***** Main code
        bsf     LED                 ; turn on LED

        DelayMS 1000                ; delay 1 sec

        bcf     LED                 ; turn off 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

Similar to the '`SLEEP()`' macro we saw earlier, 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 example above can be implemented with XC8 as follows:

```
//*** Main code
LED = 1;                    // turn on LED

__delay_ms(1000);           // delay 1 sec

LED = 0;                    // turn off LED

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

CCS PCB

Instead of a macro, the CCS PCB compiler provides a built-in function for clearing the watchdog timer: `restart_wdt()`.

Here is the CCS PCB code, equivalent to the example above, using the `restart_wdt()` function:

```
//*** Main code
output_high(LED);               // turn on LED

delay_ms(1000);                 // delay 1 sec

output_low(LED);                // turn off LED

while (TRUE)                    // repeatedly clear watchdog timer forever
    restart_wdt();
```

Periodic wake from sleep

As explained in baseline assembler lesson 7, the watchdog timer is also often used to periodically wake the PIC from sleep mode, typically to check or log 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 that lesson replaced the endless loop with a '`sleep`' instruction:

```
;***** Main code
        bsf     LED             ; turn on LED

        DelayMS 1000            ; delay 1 sec

        bcf     LED             ; turn off LED

        sleep                   ; enter sleep mode
```

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

There are no new instructions or concepts needed for this example; the main code is simply:

```
//*** Main code
LED = 1;                    // turn on LED

__delay_ms(1000);           // delay 1 sec

LED = 0;                    // turn off LED

SLEEP();                    // enter sleep mode
```

CCS PCB

Again, there are no new statements needed; the main code is much the same as we have seen before:

```
//*** Main code
output_high(LED);               // turn on LED

delay_ms(1000);                 // delay 1 sec

output_low(LED);                // turn off LED

sleep();                        // enter sleep mode
```

Clock (Oscillator) Options

Baseline lesson 7 also discussed the various clock, or oscillator, configurations available on the PIC12F509.

A number of examples were used to demonstrate the various options. Since the only new features in these examples were the configuration word settings, and no other new concepts were introduced, there would be little point in reproducing C versions of those examples here.

However, for reference, here is a summary of the oscillator configuration options for the XC8 and CCS compilers, with the corresponding MPASM symbols:

FOSC<1:0>	Oscillator configuration	MPASM	XC8	CCS PCB
00	LP oscillator	_LP_OSC	OSC_LP	LP
01	XT oscillator	_XT_OSC	OSC_XT	XT
10	Internal RC oscillator	_IntRC_OSC	OSC_IntRC	INTRC
11	External RC oscillator	_ExtRC_OSC	OSC_ExtRC	RC

For example, to configure the processor for use with a LP crystal using XC8, you could use:

```
// ext reset, no code protect, watchdog, LP crystal
__CONFIG(MCLRE_ON & CP_OFF & WDT_ON & OSC_LP);
```

Or to set the processor configuration for an external RC oscillator using CCS PCB, you could use:

```
// ext reset, no code protect, watchdog, ext RC oscillator
#fuses MCLR,NOPROTECT,WDT,RC
```

Summary

Overall, we have seen that the sleep mode, wake-up on change, and watchdog timer features of the baseline PIC architecture can be accessed effectively in C programs, using both the XC8 and CCS compilers.

However, CCS PCB lacks support for detecting wake-on-change resets, and its watchdog timer setup function has a side effect which meant that we had to rearrange one example. This (non-obvious side effects) is certainly a potential issue when using compilers such as CCS', which hide the details of the PIC architecture behind built-in functions. On the other hand, the CCS source code in all of the examples is concise and clear. You just need to be aware of the potential for side effects from those functions.

The next lesson will focus on driving 7-segment displays (revisiting the material from baseline assembler lesson 8), showing how lookup tables and multiplexing can be implemented using C.

And to do that, we'll introduce the 14-pin PIC16F506.