Introduction to PIC Programming

Baseline Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 7: Sleep Mode, the Watchdog Timer and Clock Options

We've now covered, at least at an introductory level, the major features of the PIC12F508/509 (admittedly, one of the simplest of the "modern" PICs; only the PIC10F200 and PIC10F202 are more limited), including digital input, output, and using the Timer0 module as either a timer or counter.

That's enough to build a surprising number of applications, but these MCUs have a few other features which can be quite useful. These are covered in chapter 7 of the PIC12F508/509/16F505 data sheet, titled "Special Features of the CPU". Although you should refer to the latest data sheet for the full details, this lesson will introduce the following "special" (and very useful) features:

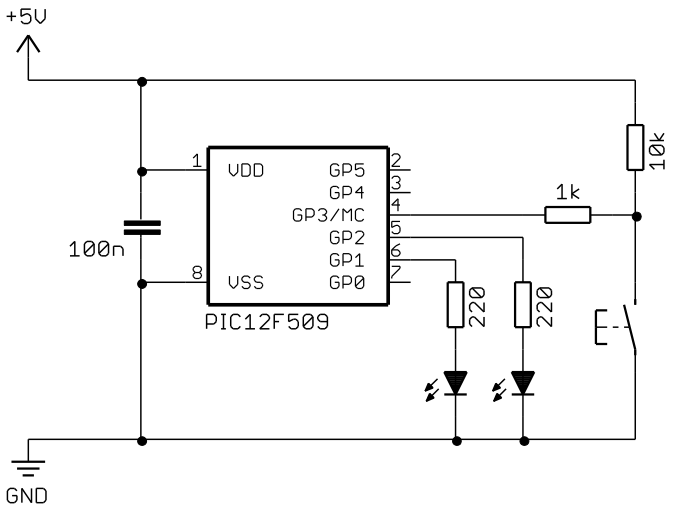
- Sleep mode (power down)
- Wake-up on change (power up)
- The watchdog timer
- Oscillator (clock) configurations

Sleep Mode

The material covered so far in these tutorials should allow you to design a simple project such as the Gooligum Electronics "Toy Traffic Lights" kit: lighting LEDs, responding to and debouncing buttons and switches, and timing. But there's one thing the Toy Traffic Lights project does, that hasn't been covered yet; it turns itself "off" (saving power), and comes back "on" at the touch of a button. There is no on/off switch; the circuit is always powered, and yet the batteries are able to last for years.

That is done by putting the PIC into the power-saving standby, or *sleep* mode.

To demonstrate how it is used, we'll use the circuit from lesson 5, shown on the right.



It consists of a PIC12F508 or 509, 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 lesson 1. But if you want to be able 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 inline 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   b'111101'       ; configure GP1 (only) as an output
        tris    GPIO

        bsf     GPIO,1          ; turn on LED

waitlo  btfsc   GPIO,3          ; 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 5V supply, before pressing the button, with the LED on, was measured to be 10.83 mA. After pressing the button, the measured current dropped to 10.47 mA, a fall of only 0.36 mA.

This happens because, when the PIC goes into standby mode, the PIC stops executing instructions, saving some power (360 µA 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   b'111101'       ; configure GP1 (only) as an output
        tris    GPIO

        bsf     GPIO,1          ; turn on LED

waitlo  btfsc   GPIO,3          ; wait for button press (low)
        goto    waitlo

        bcf     GPIO,1          ; turn off LED

        sleep                   ; enter sleep mode
```

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 will draw unnecessary current.

> *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

Most baseline PICs include a facility for coming out of standby mode when an input changes, called *wake-up on change*. This is used, for example, in the "Toy Traffic Lights" project to power up the device when the button on it is pressed.

Wake-up on change is available on the GP0, GP1 and GP3 pins on the PIC12F508/509 (these are the same pins that internal pull-ups are available for). Note that on the baseline PICs, this is all or nothing; either all of the available pins are enabled for wake-up on change, or none of them are.

On the PIC12F508/509, wake-up on change is controlled by the $\overline{\text{GPWU}}$ bit in the OPTION register:

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

By default (after a power-on or reset), $\overline{\text{GPWU}}$ = 1 and wake-up on change is disabled.

To enable internal wake-up on change, clear $\overline{\text{GPWU}}$.

Assuming no other options are being set (leaving all the other bits at the default value of '1'), wake-up on change is enabled by:

```
movlw   b'01111111'     ; enable wake-up on change
        ; 0-------          (NOT_GPWU = 0)
option
```

If wake-up on change is enabled, the PIC will be reset if, in sleep mode, the value at any of the "wake-up on change" pins becomes different to the last time those pins were read, prior to entering sleep.

> *Note: You should read the input pins configured for wake-up on change just prior to entering sleep mode. Otherwise, if the value at a pin had changed since the last time it was read, a "wake up on change" reset will occur immediately upon entering sleep mode, as the input value would be seen to be different from that last read.*

It is also important that any input which will be used to trigger a wake-up is stable before entering sleep mode. Consider what would happen if wake-up 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 be reset.

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 reset and the LED will come back on! To successfully use the pushbutton to turn the circuit (PIC and LED) "off", it is necessary to wait for the button to go high and remain stable (debounced) before entering sleep mode.

But there's still a problem: when the button is pressed while the PIC is in sleep mode, the PIC will reset, and the LED will light. That's what we want. The problem is that PICs are fast, and human fingers are slow – the button will still be down when the program first checks for "button down", 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", in case the program is starting following a button press.

So the necessary sequence is:

```
turn on LED
wait for stable button high
wait for button low
turn off LED
wait for stable button high
sleep
```

The following code, which makes use of the debounce macro defined in lesson 6, implements this:

```
;***** Initialisation
        movlw   ~(1<<nLED)      ; configure LED pin (only) as an output
        tris    GPIO
        movlw   b'01000111'     ; configure wake-up on change and Timer0:
                ; 0-------          enable wake-up on change (NOT_GPWU = 0)
                ; --0-----          timer mode (T0CS = 0)
                ; ----0---          prescaler assigned to Timer0 (PSA = 0)
                ; -----111          prescale = 256 (PS = 111)
        option                  ;   -> increment every 256 us

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

        DbnceHi BUTTON          ; wait for stable button high
                                ;   (in case restarted after button press)
waitlo  btfsc   BUTTON          ; wait for button press (low)
        goto    waitlo

        bcf     LED             ; turn off LED

        DbnceHi BUTTON          ; wait for stable button release

        sleep                   ; enter sleep mode

        END
```

(the labels '`LED`' and '`BUTTON`' are defined earlier in the program; see the complete listing below)

This code does essentially the same thing as the "toggle a LED" programs developed in lesson 4, except that in this case, when the LED is off, the PIC is drawing negligible power.

> *Note: Wake-up from sleep on pin change on baseline PICs causes a processor reset; instruction execution recommences from the reset vector, as it does following all types of reset, including power-on. Execution does* ***not*** *resume at the instruction following "sleep".*

Since the same start-up instructions are executed, whether the PIC has been powered on for the first time, or was reset by a wake-up from sleep, how is it possible to tell whether a wake-up on change has occurred?

Of course, that's not necessarily important. The program above debounces the pushbutton when it first starts, just in case it had restarted because of a wake-up from sleep. If the PIC had just been powered on, there would be no need to do this debouncing, but it doesn't hurt to do it anyway – if the button is already up, then the debounce routine only introduces a 10ms delay.

But sometimes you would like your program to behave differently, depending on why it was (re)started.

You can do that by testing the GPWUF flag bit in the STATUS register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
STATUS	GPWUF	-	PA0	$\overline{TO}$	$\overline{PD}$	Z	DC	C

GPWUF is set to '1' by a wake-up on change, and is cleared by all other resets. So if GPWUF has been set, it means that a wake-up on change reset has occurred.

Complete program

To demonstrate how the GPWUF flag can be tested, to differentiate between wake-up on change and power-on resets, the following program, based on the code above, lights the LED on GP2 following a wake-up on change reset, but not when the PIC is first powered on. And since the wake-up on change condition is being tested anyway, the initial button debounce is only performed if a wake-up on change has occurred. (Note that the debounce macro is defined in an include file.)

```
;*****************************************************************
;                                                                *
;   Description:    Lesson 7, example 2b                         *
;                                                                *
;   Demonstrates differentiation between wake up on change       *
;   and 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 *
;                                                                *
;*****************************************************************

    list        p=12F509
    #include    <p12F509.inc>
    #include    <stdmacros-base.inc> ; DbcneHi - debounce switch, wait for high

    radix       dec


;***** CONFIGURATION
                ; int reset, no code protect, no watchdog, 4MHz int clock
    __CONFIG    _MCLRE_OFF & _CP_OFF & _WDT_OFF & _IntRC_OSC

; pin assignments
    #define     LED     GPIO,1      ; on/off indicator LED on GP1
    constant    nLED=1              ;   (port bit 1)
    #define     WAKE    GPIO,2      ; wake on change indicator LED on GP2
    constant    nWAKE=2             ;   (port bit 2)
    #define     BUTTON  GPIO,3      ; pushbutton on GP3 (active low)


;*****************************************************************
RESET   CODE    0x000           ; effective reset vector
        movwf   OSCCAL          ; update OSCCAL with factory cal value


;***** MAIN PROGRAM

;***** Initialisation
start
        movlw   ~(1<<nLED|1<<nWAKE) ; configure LED pins as outputs
        tris    GPIO
        clrf    GPIO            ; start with all LEDs off
        movlw   b'01000111'     ; configure wake-up on change and Timer0:
                ; 0-------         enable wake-up on change (NOT_GPWU = 0)
                ; --0-----         timer mode (T0CS = 0)
                ; ----0---         prescaler assigned to Timer0 (PSA = 0)
                ; -----111         prescale = 256 (PS = 111)
        option                  ;   -> increment every 256 us
```

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

        btfss   STATUS,GPWUF    ; if wake-up on change has occurred,
        goto    waitlo
        bsf     WAKE            ;   turn on wake-up indicator
        DbnceHi BUTTON          ;   wait for stable button high

waitlo  btfsc   BUTTON          ; wait for button press (low)
        goto    waitlo

        clrf    GPIO            ; turn off LEDs

        DbnceHi BUTTON          ; wait for stable button release

        sleep                   ; enter sleep mode

        END
```

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 the prescaler.

As we saw in lesson 5, the prescaler 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	$\overline{\text{GPWU}}$	$\overline{\text{GPPU}}$	T0CS	T0SE	PSA	PS2	PS1	PS0

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

> *Note: The baseline PICs include a single prescaler, which can be used with either the Timer0 module or the Watchdog Timer, but not both.*
>
> *If the prescaler is assigned to the Watchdog Timer, it cannot be used with Timer0.*

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

PS<2:0> bit value	WDT prescale ratio
000	1 : 1
001	1 : 2
010	1 : 4
011	1 : 8
100	1 : 16
101	1 : 32
110	1 : 64
111	1 : 128

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 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-	-	-	-	-	-	-	MCLRE	$\overline{CP}$	WDTE	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.

Watchdog Timer example

To demonstrate how the watchdog timer allows the PIC to recover from a crash, we'll use a simple program which turns on a 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 1.0s and off for 1.3s (approximately).

To make it easy to select between 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 protect, watchdog, 4MHz int clock
    __CONFIG    _MCLRE_ON & _CP_OFF & _WDT_ON & _IntRC_OSC
ELSE
                ; ext reset, no code protect, no watchdog, 4MHz int clock
    __CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _IntRC_OSC
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 code to flash the LED once and then enter an endless loop is simple, making use of the '`DelayMS`' macro introduced in lesson 6:

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

        movlw   ~(1<<nLED)      ; configure LED pin (only) as an output
        tris    GPIO

; Main code
        bsf     LED             ; turn on LED
        DelayMS 1000            ; delay 1s
        bcf     LED             ; turn off LED

        goto    $               ; wait forever
```

If you build and run this 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.

As mentioned in the discussion of "wake-up on change", sometimes you'd like your program to behave differently, depending on why it was restarted. In particular, if, in normal operation, a watchdog timer reset should never occur, you may wish to turn on an alarm indicator if a watchdog timer reset has happened, to show that an unexpected problem has occurred.

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

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
STATUS	GPWUF	-	PA0	$\overline{TO}$	$\overline{PD}$	Z	DC	C

The $\overline{TO}$ bit is cleared to '0' by a reset caused by watchdog timer, and is set to '1' by power-on reset, entering sleep mode, or by execution of the '`clrwdt`' instruction.

If $\overline{TO}$ has been cleared, it means that a watchdog timer reset has occurred.

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

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

        movlw   ~(1<<nLED|1<<nWDT)  ; configure LED pins as outputs
        tris    GPIO
        clrf    GPIO                ; start with all LEDs off

; Main code
        btfss   STATUS,NOT_TO   ; if WDT timeout has occurred,
        bsf     WDT             ;   turn on "error" LED

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

        goto    $               ; wait forever
```

Of course, you will normally want to avoid watchdog timer resets.

As discussed earlier, to prevent the watchdog timer timing out, simply place a '`clrwdt`' instruction within the main loop, with the watchdog timer period set to be longer than it should ever take to complete the loop.

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 remains off – regardless of the watchdog timer setting.

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 drain, 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.

Setting up a periodic wake-up is easy: simply configure the watchdog timer for the desired wake-up period, perform the "main code" tasks (testing and responding to inputs), then enter sleep mode. When the watchdog timer period has elapsed, the PIC will wake up, perform the main tasks, and then go to sleep again.

To illustrate this process, we can simply replace the endless loop with a 'sleep' instruction:

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

        movlw   ~(1<<nLED)          ; configure LED pin (only) as an output
        tris    GPIO

; Main code
        bsf     LED                 ; turn on LED
        DelayMS 1000                ; delay 1s
        bcf     LED                 ; turn off LED

        sleep                       ; enter sleep mode
```

You'll find that the LED is on for 1s, and then off for around 2 s. That is because the watchdog timer is cleared automatically when the PIC enters sleep mode.

Clock Options

Every example in these lessons, until now, has used the 4 MHz internal RC oscillator as the PIC's clock source. It's usually a very good option – simple to use, needing no external components, using none of the PIC pins, and reasonably accurate.

However, there are situations where it is more appropriate to use some external clock circuitry.

Reasons to use external clock circuitry include:

- *Greater accuracy and stability.*
  A crystal or ceramic resonator is significantly more accurate than the internal RC oscillator, with less frequency drift due to temperature and voltage variations.
- *Generating a specific frequency.*
  For example, as we saw in lesson 5, the signal from a 32.768 kHz crystal can be readily divided down to 1Hz. Or, to produce accurate timing for RS-232 serial data transfers, a crystal frequency such as 1.843200 MHz can be used, since it is an exact multiple of common baud rates, such as 38400 or 9600 (1843200 = 48 × 38400 = 192 × 9600).
- *Synchronising with other components.*
  Sometimes it simplifies design if a number of microcontrollers (or other chips) are clocked from a common source, so that their outputs change synchronously – although you need to be careful; clock signals which are subject to varying delays in a circuit will not be synchronised in practice (a phenomenon known as *clock skew*), leading to unpredictable results.
- *Lower power consumption.*
  At a given supply voltage, PICs draw less current when they are clocked at a lower speed. For example, the PIC12F508/509 data sheet states (parameter D010) that, with V_{DD} = 2.0 V, supply current is typically 170 µA for a clock speed of 4MHz, but only 15 µA at 32 kHz.
  Power consumption can be minimised by running the PIC at the slowest practical clock speed. And for many applications, very little speed is needed.

PICs support a number of clock, or oscillator, configurations, allowing, through appropriate oscillator selection, any of these goals to be met (but not necessarily all at once – low power consumption and high frequencies don't mix!)

The oscillator configuration is selected by the FOSC bits in the configuration word:

Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-	-	-	-	-	-	-	MCLRE	$\overline{CP}$	WDTE	FOSC1	FOSC0

The PIC12F508 and 509 have two FOSC bits, allowing selection of one of four oscillator configurations, as in the table on the right.

FOSC<1:0>	Oscillator configuration
00	LP oscillator
01	XT oscillator
10	Internal RC oscillator
11	External RC oscillator

The internal RC oscillator is the one we have been using so far, providing a nominal 4MHz internal clock source, and has already been discussed.

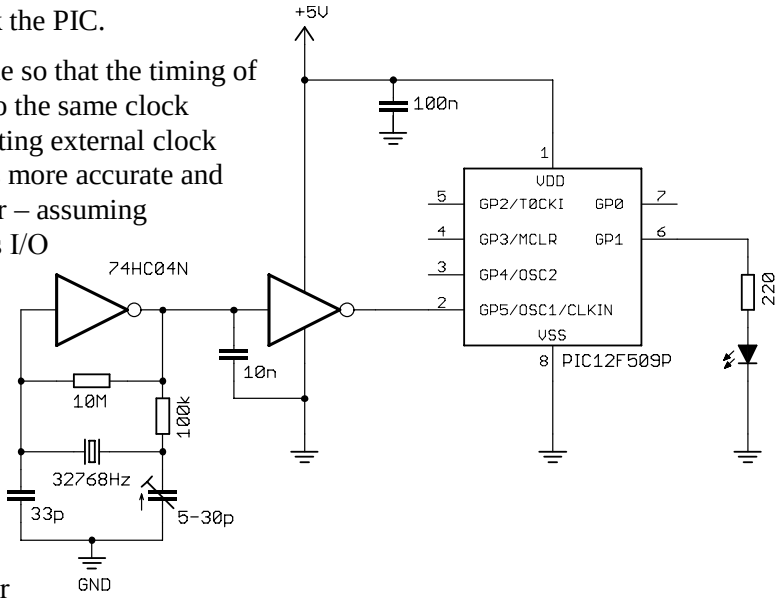
The other oscillator options are described in detail in the PIC12F508/509 data sheet, as well as in a number of application notes available on the Microchip web site, www.microchip.com. As mentioned in lesson 5, oscillator design can be something of a black art!

Instead of needlessly repeating all that material here, the following sections outline some of the most common oscillator configurations.

External clock input

An external oscillator can be used to clock the PIC.

As discussed above, this is sometimes done so that the timing of various parts of a circuit is synchronised to the same clock signal. Or, you may choose to use an existing external clock signal simply because it is available and is more accurate and stable than the PIC's internal RC oscillator – assuming you can afford the loss of two of the PIC's I/O pins.



Lesson 5 included the design for a 32.768 kHz crystal oscillator, as shown in the circuit on the right. We can use it to demonstrate how to use an external clock signal.

To use an external oscillator with the PIC12F508 or 509, the PIC has to be configured in either 'LP' or 'XT' oscillator mode. You should use 'LP' for frequencies below around 200 kHz, and 'XT' for higher frequencies.

The clock signal is connected to the CLKIN input: pin 2 on a PIC12F508/509.

When using an external clock signal in the 'LP' and 'XT' oscillator modes, the OSC2 pin (pin 3 on a PIC12F508/509) is unused; it is left disconnected and the associated I/O port (GP4) is not available for use.

Many PICs, such as the 16F505, offer an 'EC' (external clock) oscillator mode, which leaves the OSC2 pin available for I/O. But that's not an option on the 12F508/509.

To illustrate the operation of this circuit, we can modify the crystal-driven LED flasher program developed in lesson 5. In that program, the external 32.768 kHz signal was used to drive the Timer0 counter.

Now, however, the 32.768 kHz signal is driving the processor clock, giving an instruction clock rate of 8192 Hz. If Timer0 is configured in timer mode with a 1:32 prescale ratio, TMR0<7> will be cycling at exactly 1 Hz (since 8192 = 32 × 256) – as is assumed in the main body of the program from lesson 5.

Therefore, to adapt that program for this circuit, all we need to do is to change the configuration statement from:

```
    __CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _IntRC_OSC
```

to:

```
    __CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _LP_OSC
```

(The _XT_OSC option should be used instead of _LP_OSC for higher clock frequencies)

And also to change the initialisation code from:

```
        movlw   b'11110110'     ; configure Timer0:
                ; --1-----          counter mode (T0CS = 1)
                ; ----0---          prescaler assigned to Timer0 (PSA = 0)
                ; -----110          prescale = 128 (PS = 110)
        option                  ;   -> increment at 256 Hz with 32.768 kHz input
```

to:

```
        movlw   b'11010100'     ; select Timer0:
                ; --0-----          timer mode (T0CS = 0)
                ; ----0---          prescaler assigned to Timer0 (PSA = 0)
                ; -----100          prescale = 32 (PS = 100)
        option                  ;   -> increment at 256 Hz with 32.768 kHz clock
```

With these changes made, the LED on GP1 should flash at almost exactly 1 Hz – to within the accuracy of the crystal oscillator.

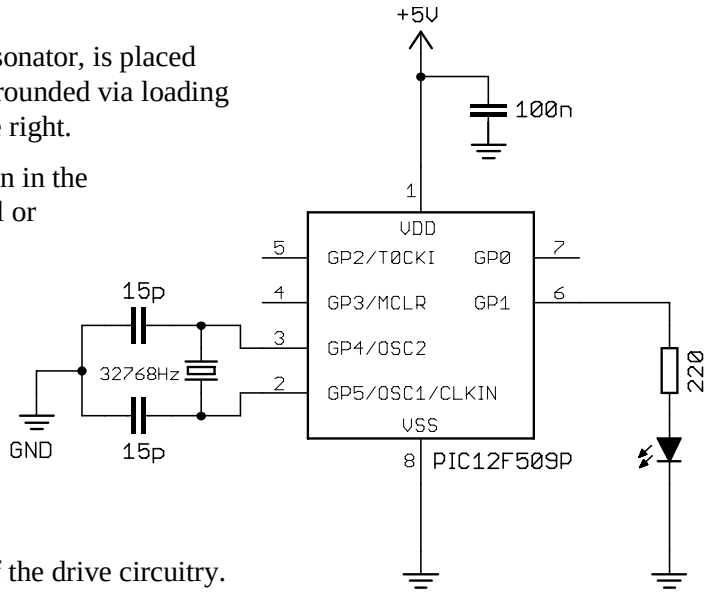
Crystals and ceramic resonators

Generally, there is no need to build your own crystal oscillator; PICs include an oscillator circuit designed to drive crystals directly.

A parallel (not serial) cut crystal, or a ceramic resonator, is placed between the OSC1 and OSC2 pins, which are grounded via loading capacitors, as shown in the circuit diagram on the right.

Typical values for the loading capacitors are given in the PIC datasheets, but you should consult the crystal or resonator manufacturer's data to be sure.

For some crystals it may be necessary to reduce the current drive by placing a resistor between OSC2 and the crystal, but in most cases it is not needed, and the circuit shown here can be used.



The PIC12F508/509 provides two crystal oscillator modes: 'XT' and 'LP'.

They differ in the gain and frequency response of the drive circuitry.

'XT' ("crystal") is the mode used most commonly for crystal or ceramic resonators operating between 200 kHz and 4 MHz.

Lower frequencies generally require lower gain. The 'LP' ("low power") mode uses less power and is designed to drive common 32.786 kHz "watch" crystals, as used in the external clock circuit above, although it can also be used with other low-frequency crystals or resonators.

The circuit as shown here can be used to operate the PIC12F508/509 at 32.768 kHz, giving low power consumption and an 8192 Hz instruction clock rate, which, as in the external clock example, is easily divided to create an accurate 1 Hz signal.

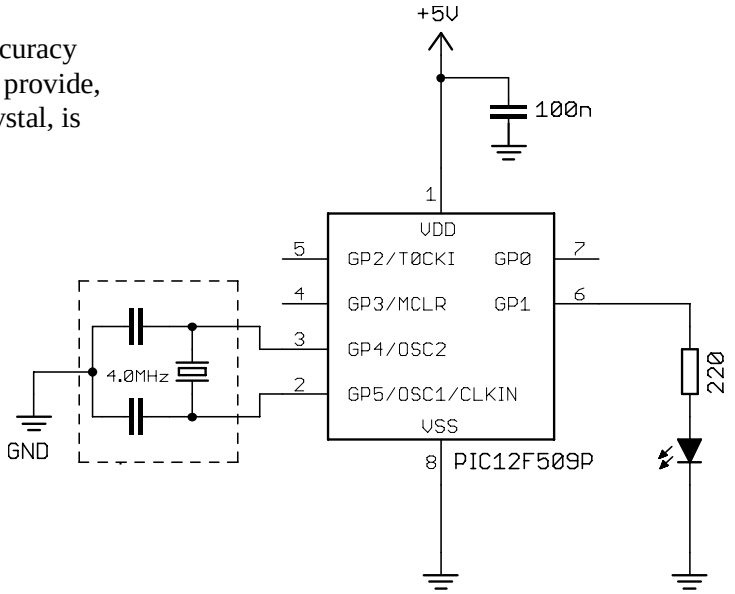
To flash the LED at 1 Hz, the program is exactly the same as for the external clock, except that the configuration statement must include the `_LP_OSC` option:

```
__CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _LP_OSC
```

A convenient option, when you want greater accuracy and stability than the internal RC oscillator can provide, but do not need as much as that offered by a crystal, is to use a ceramic resonator.

These are particularly convenient because they are available in 3-terminal packages which include appropriate loading capacitors, as shown in the circuit diagram on the right. The resonator package incorporates the components within the dashed lines.

Usually the built-in loading capacitors are adequate and no additional components are needed, other than the 3-pin resonator package.



Complete program

To test this circuit, we can use the "flash an LED" program developed in lesson 2; the only necessary change is to replace the `_IntRC_OSC` configuration option with `_XT_OSC`, to select crystal oscillator mode.

However, in lesson 2 we concluded that, since the internal RC oscillator is only accurate to within 1% or so, there was no reason to strive for precise loop timing; a delay of 499.958 ms was considered close enough to the desired 500 ms.

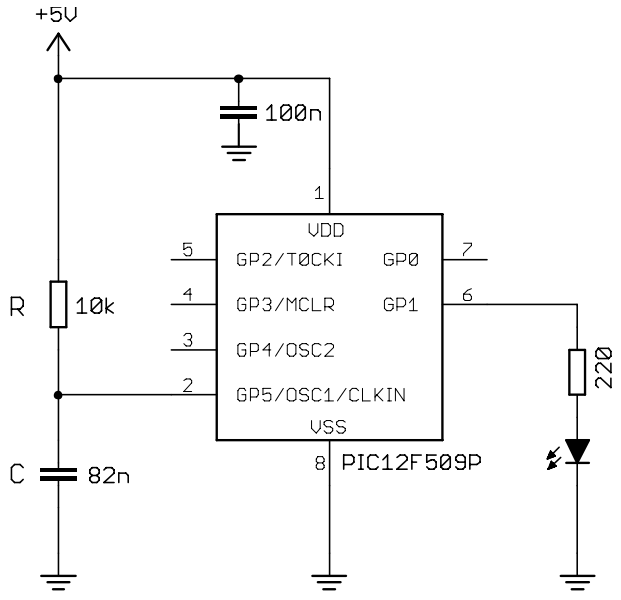
In this case, given that we are using a more accurate oscillator, driven by a crystal or ceramic resonator, it is worth the effort to be more precise. Therefore in the following program an additional short loop and some `nop` instructions have been added to pad out the total loop time to exactly 500,000 instruction cycles, which will be as close to 500 ms as the accuracy of the crystal or resonator allows, as shown:

```
;*****************************************************************************
;                                                                            *
;   Description:    Lesson 7, example 4b                                     *
;                                                                            *
;   Demonstrates use of 4 Mhz external clock (crystal or resonator)          *
;                                                                            *
;   LED on GP1 flashes at 1 Hz (50% duty cycle),                             *
;   with timing derived from 1 MHz instruction clock                         *
;                                                                            *
;*****************************************************************************

    list        p=12F509
    #include    <p12F509.inc>

    radix       dec


;***** CONFIGURATION
                ; ext reset, no code protect, no watchdog, crystal osc
    __CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _XT_OSC

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


;***** VARIABLE DEFINITIONS
VARS    UDATA_SHR
sGPIO   res 1                   ; shadow copy of GPIO
dc1     res 1                   ; delay loop counters
dc2     res 1


;*****************************************************************************
RESET   CODE    0x000           ; effective reset vector
        movwf   OSCCAL          ; update OSCCAL with factory cal value


;***** MAIN PROGRAM

;***** Initialisation
startn
        movlw   ~(1<<nLED)      ; configure LED pin (only) as an output
        tris    GPIO
        clrf    GPIO            ; start with GPIO clear (LED off)
        clrf    sGPIO           ; update shadow register
```

```
;***** Main loop
flash
        ; toggle LED
        movf    sGPIO,w
        xorlw   1<<nLED         ; flip shadow of LED pin bit
        movwf   sGPIO
        movwf   GPIO            ; write to GPIO

        ; delay 500ms
        movlw   .244            ; outer loop: 244 x (1023 + 1023 + 3) + 2
        movwf   dc2             ;   = 499,958 cycles
        clrf    dc1             ; inner loop: 256 x 4 - 1
dly1    nop                     ; inner loop 1 = 1023 cycles
        decfsz  dc1,f
        goto    dly1
dly2    nop                     ; inner loop 2 = 1023 cycles
        decfsz  dc1,f
        goto    dly2
        decfsz  dc2,f
        goto    dly1
        movlw   .11             ; delay another 11 x 3 - 1 + 2 = 34 cycles
        movwf   dc2             ;  -> delay so far = 499,958 + 34
dly3    decfsz  dc2,f           ;  = 499,992 cycles
        goto    dly3
        nop                     ; main loop overhead = 6 cycles, so add 2 nops
        nop                     ;  -> loop time = 499,992 + 6 + 2 = 500,000 cycles

        goto    flash           ; repeat forever


        END
```

External RC oscillator

Finally, a low-cost, low-power option: the baseline PICs can be made to oscillate with timing derived from an external resistor and capacitor, as shown on the right.



External RC oscillators, with appropriate values of R and C, are used when a very low clock rate can be used – drawing significantly less power than when the internal 4 MHz RC oscillator is used.

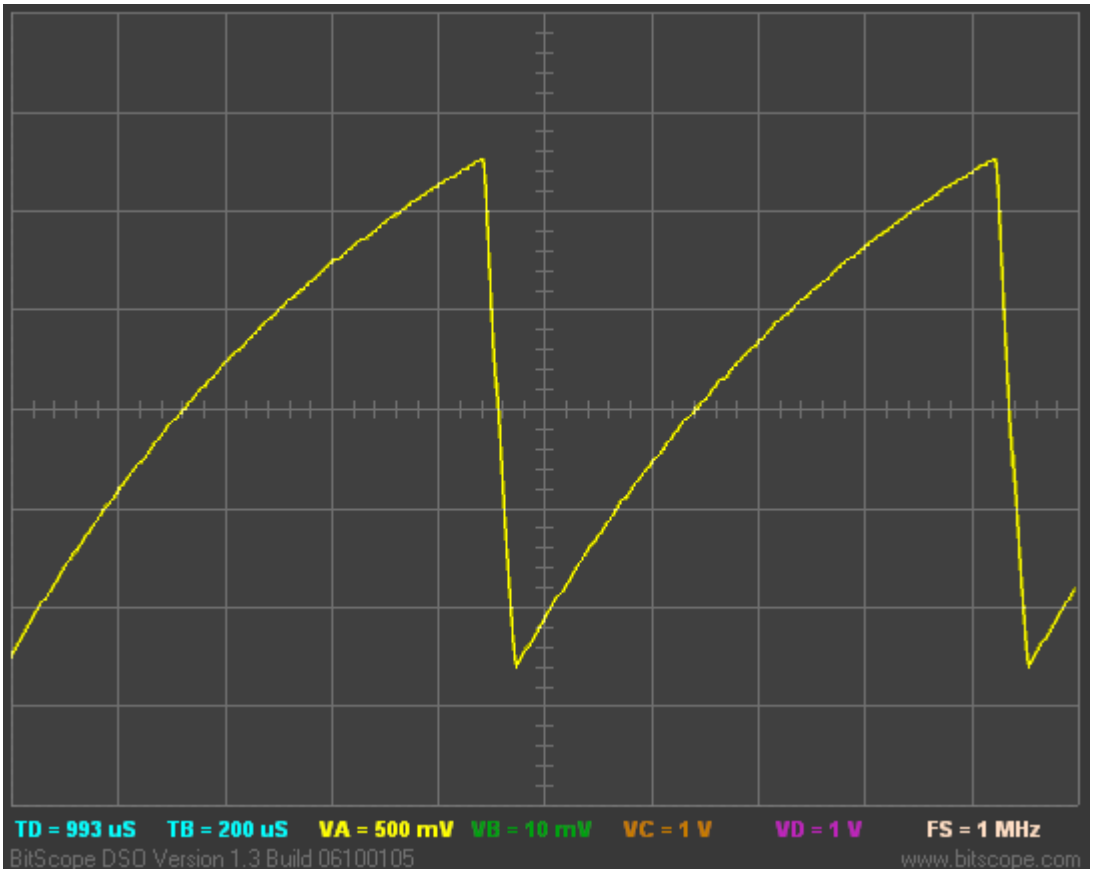
It can also simplify some programming tasks when the PIC is run slowly, needing fewer, shorter delay loops.

The external RC oscillator is a *relaxation* type. The capacitor is charged through the resistor, the voltage *v* at the OSC1 pin rising with time *t* according to the formula:

$$v = V_{DD}\left(1 - e^{-t/RC}\right)$$

The voltage increases until it reaches a threshold, typically 0.75 × V_{DD}. A transistor is then turned on, which quickly discharges the capacitor until the voltage falls to approx. 0.25 × V_{DD}. The capacitor then begins charging through the resistor again, and the cycle repeats.

This is illustrated by the following oscilloscope trace, recorded at the OSC1 pin in the circuit above, with the component values shown:



In theory, assuming upper and lower thresholds of 0.75 × V_{DD} and 0.25 × V_{DD}, the period of oscillation is equal to 1.1 × RC (in seconds, with R in Ohms and C in Farads).

In practice, the capacitor discharge is not instantaneous (and of course it can never be), so the period is a little longer than this. Microchip does not commit to a specific formula for the frequency (or period) of the external RC oscillator, only stating that it is a function of V_{DD}, R, C and temperature, and in some documents providing some reference charts. But for rough design guidance, you can assume the period of oscillation is approximately 1.2 × RC.

Microchip recommends keeping R between 5 kΩ and 100 kΩ, and C above 20 pF.

In the circuit above, R = 10 kΩ and C = 82 nF. Those values will give a period of approximately:

$$1.2 \times 10{\times}10^{3} \times 82{\times}10^{-9}\ \text{s} = 984\ \mu\text{s}$$

Inverting that gives 1016 Hz.

In practice, the measured frequency was 1052 Hz; reasonably close, but the lesson should be clear: don't use an external RC oscillator if you want high accuracy or good stability.

> *Only use an external RC oscillator if the exact clock rate is unimportant.*

So, given a roughly 1 kHz clock, what can we do with it? Flash an LED, of course!

Using a similar approach to before, we can use the instruction clock (approx. 256 Hz) to increment Timer0. In fact, with a prescale ratio of 1:256, TMR0 will increment at approx. 1 Hz.

TMR0<0> would then cycle at 0.5 Hz, TMR0<1> at 0.25 Hz, etc.

Now consider what happens when the prescale ratio is set to 1:64. TMR0 will increment at 4 Hz, TMR0<0> will cycle at 2 Hz, and TMR0<1> will cycle at 1 Hz, etc.

And that suggests a very simple way to make the LED on GP1 flash at 1Hz. If we continually copy TMR0 to GPIO, each bit of GPIO will continually reflect each corresponding bit of TMR0. In particular, GPIO<1> will always be set to the same value as TMR0<1>. Since TMR0<1> is cycling at 1 Hz, GPIO<1> (and hence GP1) will also cycle at 1 Hz.

Complete program

The following program implements the approach described above. Note that the external RC oscillator is selected by using the option `_ExtRC_OSC` in the configuration statement.

The "main loop" is only three instructions long – by far the shortest "flash an LED" program we have done so far, illustrating how a slow clock rate can sometimes simplify some programming problems.

On the other hand, it is also the least accurate of the "flash an LED" programs, being only approximately 1 Hz. But for many applications, the exact speed doesn't matter; it only matters that the LED visibly flashes.

```
;*****************************************************************
;   Description:    Lesson 7, example 4c                        *
;                                                               *
;   Demonstrates use of external RC oscillator (~1 kHz)         *
;                                                               *
;   LED on GP1 flashes at approx 1 Hz (50% duty cycle),         *
;   with timing derived from instruction clock                  *
;                                                               *
;*****************************************************************
;   Pin assignments:                                            *
;       GP1 - flashing LED                                      *
;                                                               *
;*****************************************************************

    list        p=12F509
    #include    <p12F509.inc>

    radix       dec


;***** CONFIGURATION
                ; ext reset, no code protect, no watchdog, external RC osc
    __CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _ExtRC_OSC


;*****************************************************************
RESET   CODE    0x000           ; effective reset vector
        movwf   OSCCAL          ; update OSCCAL with factory cal value


;***** MAIN PROGRAM

;***** Initialisation
start
        movlw   b'111101'       ; configure GP1 (only) as output
        tris    GPIO
        movlw   b'11010101'     ; configure Timer0:
                ; --0-----          timer mode (T0CS = 0)
                ; ----0---          prescaler assigned to Timer0 (PSA = 0)
                ; -----101          prescale = 64 (PS = 101)
        option                  ;   -> increment at 4 Hz with 1 kHz clock
```

```
;***** Main loop
loop    ; TMR0<1> cycles at 1 Hz
        ; so continually copy to GP1
        movf    TMR0,w          ; copy TMR0 to GPIO
        movwf   GPIO

        ; repeat forever
        goto    loop


        END
```

That completes our coverage of the PIC12F508/509.

To introduce further topics, we need a larger device. In the next lesson we'll move onto the 14-pin version of the 12F508/509, the 16F505, and see how to use lookup tables (and those extra pins) to drive 7-segment displays, and how to use multiplexing to drive multiple displays.