

# Introduction to PIC Programming

## Midrange Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

### Lesson 8: Reset, Power and Clock Options

The lessons until now have focussed on programming, but as engineers we also need to consider some of the “hardware” aspects of designing with midrange PIC microcontrollers.

PICs require an oscillator to drive the processor clock, and although we have been using the internal RC oscillator so far, it is not necessarily the most appropriate choice, as we will see in this lesson.

And of course PICs, like all electronic devices, need a reliable power supply. Program execution should not commence until the power supply is stable, and it may be appropriate to hold the device in a reset state if the power supply sags (a *brown-out*), to prevent unreliable operation.

In summary, this lesson covers:

- Oscillator (clock) options
- Power-on reset (POR)
- Power-up timer (PWRT)
- Brown-out detection (BOD)

#### Clock Options

Every example in this tutorial series, until now, has used the PIC12F629’s 4 MHz internal RC oscillator as the processor clock source. It’s often 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 4](#), the signal from a 32.768 kHz crystal can be readily divided down to 1 Hz. 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 ( $1843200 = 48 \times 38400 = 192 \times 9600$ ).
- *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.

Another approach is to make the PIC's clock available externally, so that other components can be synchronised with it.

- *Lower power consumption.*

At a given supply voltage, PICs draw less current when they are clocked at a lower speed. For example, the PIC12F629/675 data sheet states (parameter D015) that, with  $V_{DD} = 2.0\text{ V}$ , supply current is typically  $340\ \mu\text{A}$  when using the internal 4MHz RC oscillator, but only  $9\ \mu\text{A}$  when a 32 kHz crystal oscillator is used.

Power consumption can be minimised by running the PIC at the slowest practical clock speed and power supply voltage. And for many applications, very little speed is needed.

- *Faster operation.*

Most midrange PICs can operate at a clock rate of up to 20 MHz, while the internal RC oscillator generally runs at only 4 or 8 MHz. If you need more speed than the internal oscillator can provide, you need to use a crystal or other external clock source.

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

The PIC12F629 has three FOSC bits, allowing selection of one of eight oscillator configurations, as in the table below:

FOSC<2:0>	Standard MPASM symbol	Oscillator configuration
000	<code>_LP_OSC</code>	LP oscillator
001	<code>_XT_OSC</code>	XT oscillator
010	<code>_HS_OSC</code>	HS oscillator
011	<code>_EC_OSC</code>	EC oscillator
100	<code>_INTRC_OSC_NOCLKOUT</code>	Internal RC oscillator + GP4
101	<code>_INTRC_OSC_CLKOUT</code>	Internal RC oscillator + CLKOUT
110	<code>_EXTRC_OSC_NOCLKOUT</code>	External RC oscillator + GP4
111	<code>_EXTRC_OSC_CLKOUT</code>	External RC oscillator + CLKOUT

### **Internal RC oscillator**

We have become familiar with the 4 MHz internal RC oscillator, but as you can see, it is available in two configurations.

In the first (the configuration we have been using), the internal RC oscillator provides an (approx) 4 MHz processor clock (FOSC), which is used to drive the execution of instructions at (approx) 1 MHz.

In the second configuration, '`_INTRC_OSC_CLKOUT`', this instruction clock (FOSC/4) is output on the CLKOUT pin, to allow external devices to be synchronised with the PIC's instruction clock.

Since, on the 12F629, CLKOUT shares pin 3, GP4 cannot be used for I/O in '`_INTRC_OSC_CLKOUT`' mode.

To see that the instruction clock appears on CLKOUT in ‘\_INTRC\_OSC\_CLKOUT’ mode, you need to either use an external device (such as a 20-bit counter) to divide the 1 MHz signal down to a speed that humans can perceive, or, as we’ll do here, simply use an oscilloscope to look at the signal on CLKOUT.

To relate the signal we see back to the instruction clock rate, it’s useful to toggle a pin as quickly as possible, for comparison with CLKOUT, using a simple program such as:

```

;*****
;
; Description: Lesson 8, example 1
;
; Demonstrates CLKOUT function in Internal RC oscillator mode
;
; Toggles a pin as quickly as possible (0.167 MHz)
; for comparison with 1 MHz CLKOUT signal
;
; Uses inline 500 ms delay routine
;
;*****
;
; Pin assignments:
; GP2 - 0.167 MHz output
;
;*****

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

errorlevel -302          ; no warnings about registers not in bank 0

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

; pin assignments
constant      nOUT=2          ; fast-changing (0.167 MHz) output on GP2

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

;***** Initialisation
                banksel  TRISIO          ; configure all pins (except GP3 and GP4)
                clrf    TRISIO          ; as outputs

;***** Main loop
                movlw   1<<nOUT          ; toggle output pin
                banksel GPIO
loop           xorwf   GPIO,f          ; as fast as possible
                goto    loop

                END

```

The internal RC oscillator with CLKOUT configuration was selected by:

```

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

```

To toggle the GP2 pin as quickly as possible, the main loop was made as tight as possible:

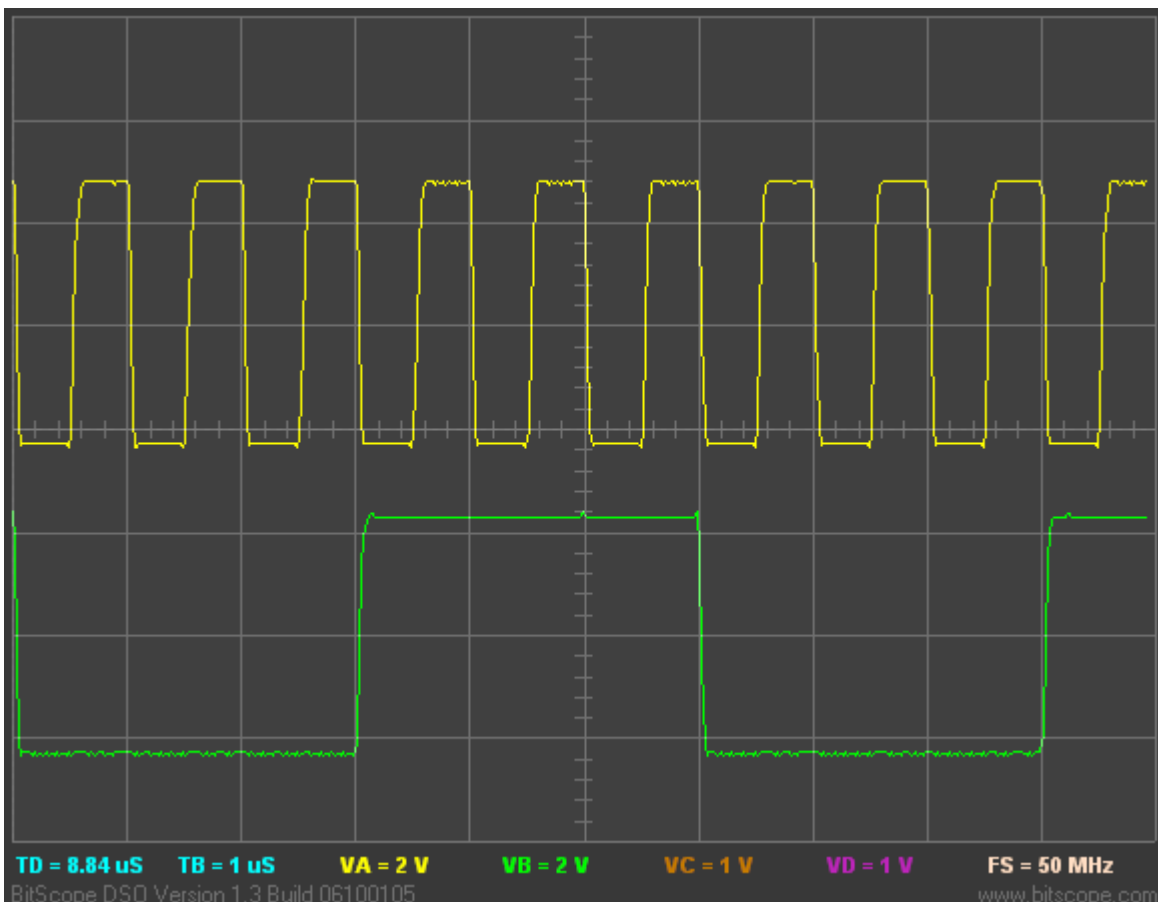
```

loop    xorwf    GPIO,f          ; as fast as possible
        goto    loop

```

With only two instructions in the loop, GP2 is toggled every three cycles (one cycle for the `xorwf` instruction and two for the `goto`), i.e. every 3  $\mu$ s, at a frequency of approx. 166 kHz.

This is apparent in the following oscilloscope plot:



The top trace is the instruction clock signal on CLKOUT, which, as you can see, has a period very close to 1  $\mu$ s, giving a frequency of 1 MHz, as expected.

The bottom trace is the signal on GP2, which changes state every three instruction cycles, as expected. Also note that the transitions on GP2 are aligned with the falling edge of the instruction clock on CLKOUT.

### External clock input

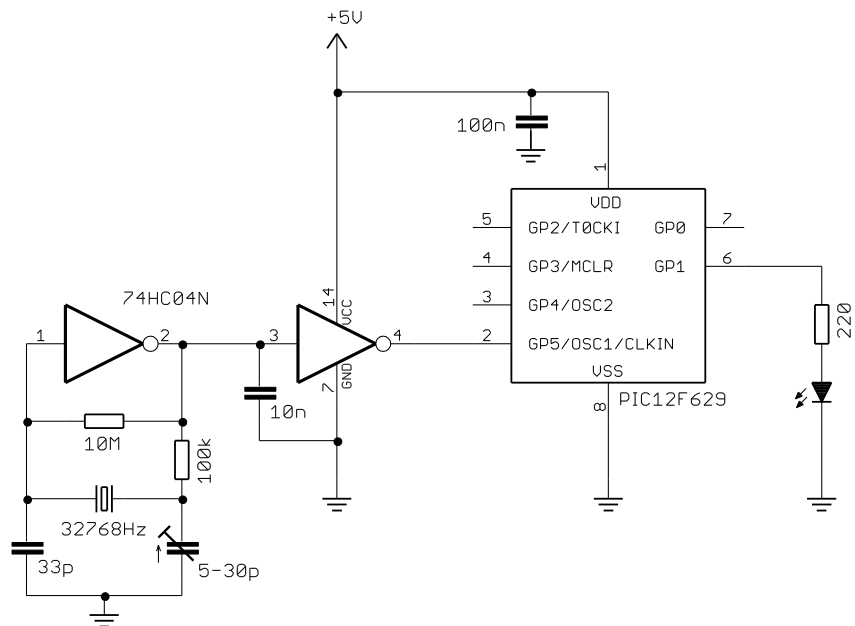
An external oscillator can be used as the PIC's clock source.

This is sometimes done so that various parts of a circuit are 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 one of the PIC's I/O pins.

[Baseline 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 PIC12F629, the 'EC' oscillator mode should be used, with the clock signal (with a frequency of up to 20 MHz) connected to the CLKIN input: pin 2 on a PIC12F629.

Since CLKIN uses the same pin as GP5, GP5 cannot be used for I/O when the PIC is in '\_EC\_OSC' mode.



Note that it is also possible to use an external clock to drive CLKIN in the 'LP', 'XT' and 'HS' oscillator modes, but in those modes the OSC2 pin (pin 3 on a PIC12F629) must be left disconnected and the associated I/O port (GP4) is not available for use – so it is much better to choose the 'EC' mode when you are using an external clock source.

To illustrate the operation of this circuit, we can modify the crystal-driven LED flasher program developed in [lesson 4](#). In that example, 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  $8192 = 32 \times 256$ ) – as is assumed in the example from [lesson 4](#).

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

```

; int reset, no code or data protect, no brownout detect,
; no watchdog, power-up timer, external clock
__CONFIG _MCLRE_OFF & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _EC_OSC

```

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)
banksel OPTION_REG ; -> incr at 256 Hz with 32.768 kHz input
movwf OPTION_REG

```

to:

```

movlw b'11000100' ; configure Timer0:
; --0----- timer mode (T0CS = 0)
; ----0---- prescaler assigned to Timer0 (PSA = 0)
; -----100 prescale = 32 (PSA = 100)
banksel OPTION_REG ; -> incr at 256 Hz with 8192 Hz inst clock
movwf OPTION_REG

```

Since we are no longer using the internal RC oscillator, there is no need to include the usual OSCCAL calibration code at the start of the program; this routine can safely be removed.

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

### Complete program

Here is the program from [lesson 4](#), modified as described above:

```

;*****
;
; Description: Lesson 8, example 2
;
; Demonstrates use of external clock mode
; (using 32.768 kHz clock source)
;
; LED flashes at 1Hz (50% duty cycle),
; with timing derived from 8192 Hz instruction clock
;
;*****
;
; Pin assignments:
; GP1 - flashing LED
;
;*****

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

errorlevel -302 ; no "register not in bank 0" warnings

;***** CONFIGURATION
; int reset, no code or data protect, no brownout detect,
; no watchdog, power-up timer, external clock
__CONFIG _MCLRE_OFF & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _EC_OSC

;***** VARIABLE DEFINITIONS
UDATA_SHR
temp res 1 ; temp register used for rotates

;*****
RESET CODE 0x0000 ; processor reset vector

;***** Initialisation
; setup port
movlw ~(1<<GP1) ; configure GP1 (only) as an output
banksel TRISIO
movwf TRISIO
; setup Timer0
movlw b'11000100' ; configure Timer0:
; --0----- timer mode (T0CS = 0)
; ----0--- prescaler assigned to Timer0 (PSA = 0)
; -----100 prescale = 32 (PSA = 100)
banksel OPTION_REG ; -> incr at 256 Hz with 8192 Hz inst clock
movwf OPTION_REG

;***** Main loop
loop ; TMR0<7> cycles at 1 Hz
; so continually copy to GP1
banksel TMR0

```

```

rlf    TMR0,w           ; copy TMR0<7> to C
clrf   temp
rlf    temp,f          ; rotate C into temp
rlf    temp,w          ; rotate once more into W (-> W<1> = TMR0<7>)
movwf  GPIO            ; update GPIO with result (-> GP1 = TMR0<7>)

; repeat forever
goto  loop

END

```

## 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 series resistor between **OSC2** and the crystal, but in most cases it is not needed, and the circuit shown here can be used.

The PIC12F629 offers three crystal oscillator modes: 'XT', 'LP' and 'HS'. They differ in the gain and frequency response of the drive circuitry.

'XT' ("crystal") is the mode used most commonly for crystals or ceramic resonators operating between 100 kHz and 4 MHz.

'HS' ("high speed") mode provides higher gain and is typically used for crystals or ceramic resonators operating above 4 MHz, up to a maximum frequency of 20 MHz. Because of the higher drive level, a series resistor is more likely to be necessary in 'HS' oscillator mode.

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.

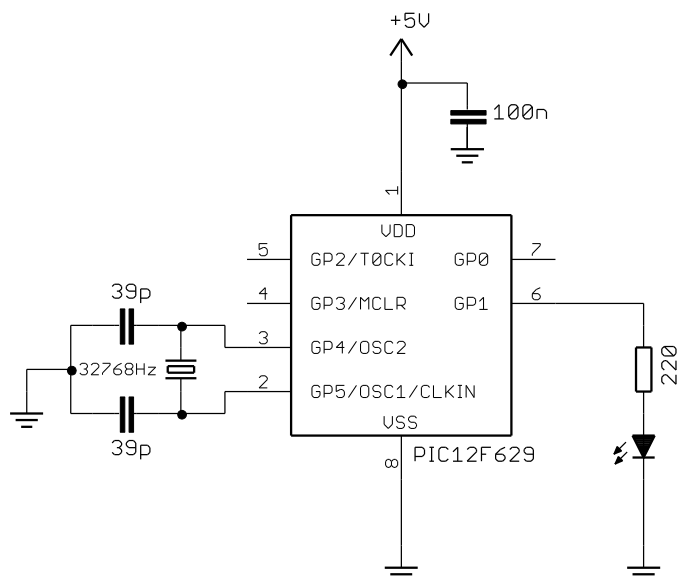
The circuit as shown here can be used to operate the PIC12F629 at 32.768 kHz, giving low power consumption and an 8192 Hz instruction clock rate, which, as in the external clock example, is easily divided to create an accurate 1 Hz signal.

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

```

__CONFIG _MCLRE_OFF & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _LP_OSC

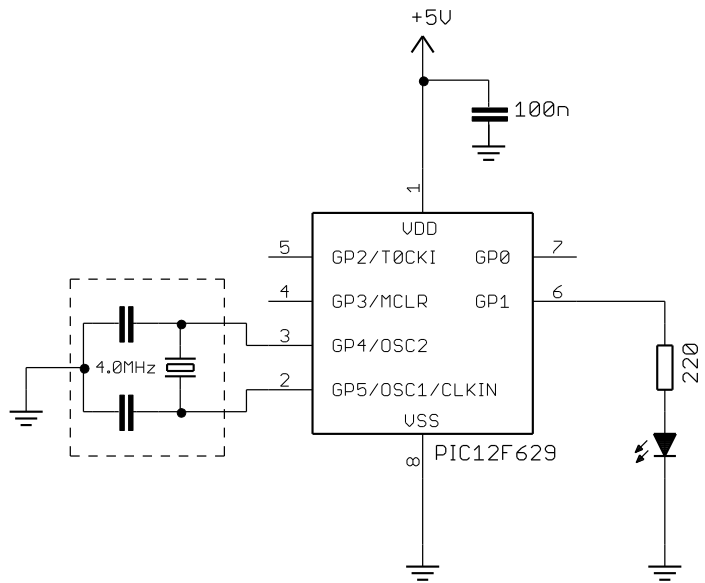
```



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.



If you are using a 4 MHz resonator, to test this circuit, you can change the ‘\_INTRC\_OSC\_NOCLKOUT’ configuration option to ‘\_XT\_OSC’ in the \_\_CONFIG directive in any programs from the examples in any of the earlier lessons, since they all used a 4 MHz clock.

A good choice is the “flash an LED at exactly 1 Hz” program developed in [lesson 6](#), since it will generate an output of exactly 1 Hz, given a processor clock of exactly 4 MHz, and so should benefit from the more accurate clock source.

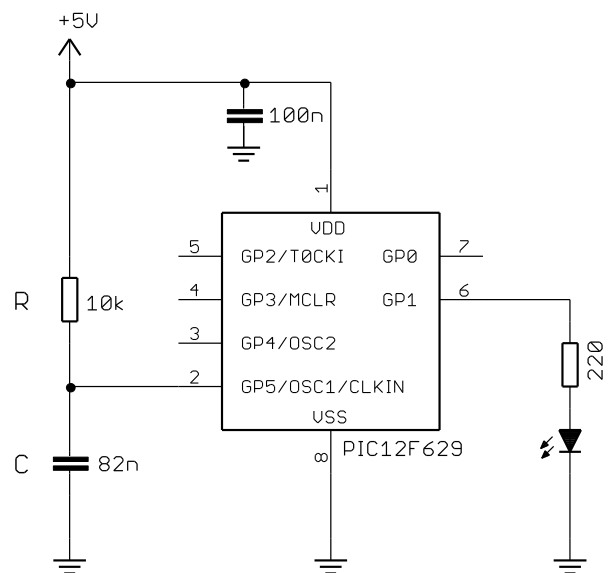
### External RC oscillator

Finally, a low-cost, low-power option: midrange PICs can use an oscillator based on an external resistor and capacitor, as shown on the right.

The availability of internal RC oscillators in modern PICs has meant that external RC oscillators are much less commonly used now, than in the days when they were the only alternative to a crystal or resonator.

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

Running the PIC slowly can also simplify some programming tasks, needing fewer, shorter delays.



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  $0.75 \times V_{DD}$ . A transistor is then turned on, which quickly discharges the capacitor until the voltage falls to approx.  $0.25 \times V_{DD}$ . The capacitor then begins charging through the resistor again, and the cycle repeats.

In theory, assuming upper and lower thresholds of  $0.75 \times V_{DD}$  and  $0.25 \times V_{DD}$ , the period of oscillation is equal to  $1.1 \times RC$  (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  $1.2 \times RC$ .

Microchip recommends keeping R between 5 k $\Omega$  and 100 k $\Omega$ , and C above 20 pF.

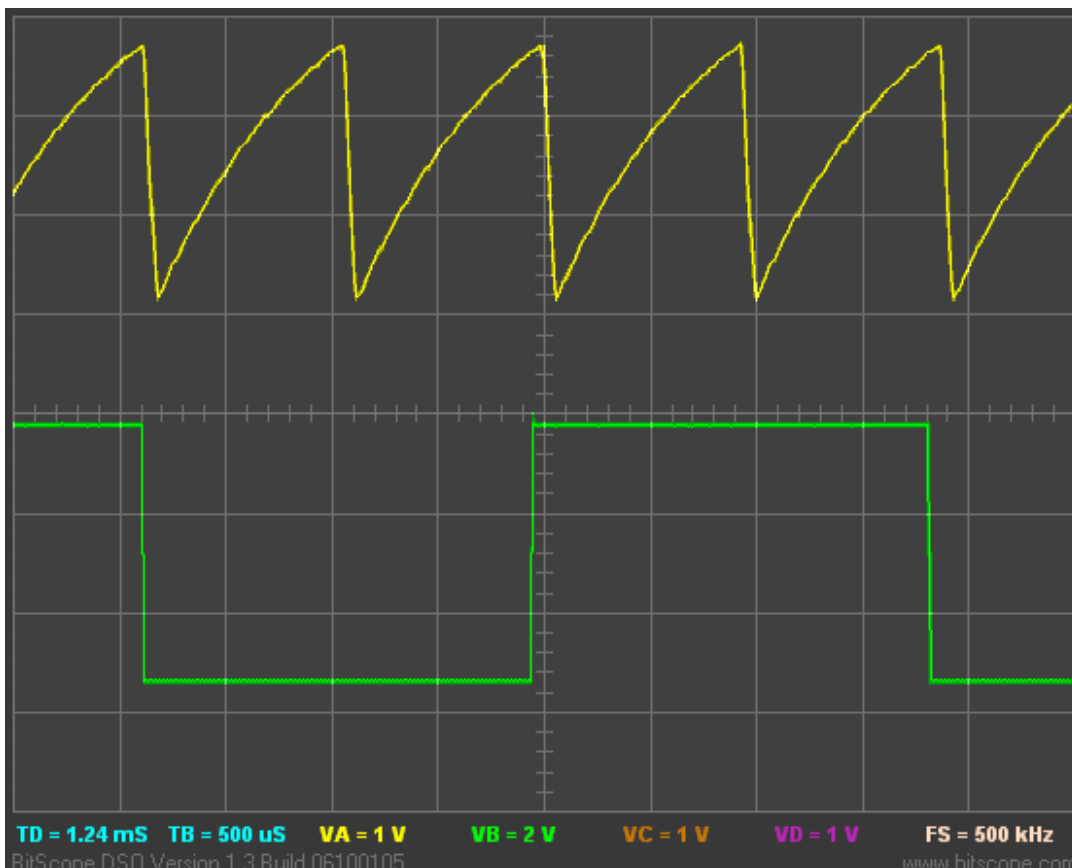
In the circuit above, R = 10 k $\Omega$  and C = 82 nF.

Those values will give a period of approximately:

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

Hence, we can expect to generate a clock frequency of around 1 kHz.

This circuit was tested, using the component values shown, giving the following oscilloscope traces:



The top trace was recorded at the OSC1 pin, and shows the expected RC charge/discharge cycles.

The bottom trace shows the instruction clock output at the CLKOUT pin; as expected, it is one quarter of the frequency of the clock input at OSC1.

In practice, the measured frequency was 1080 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 1 Hz:

If we continually copy TMR0 to GPIO, each bit of GPIO will 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_CLKOUT` in the configuration statement.

```

;*****
; Description: Lesson 8, example 5 *
; *
; 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=12F629
#include <p12F629.inc>

errorlevel -302 ; no "register not in bank 0" warnings

;**** CONFIGURATION
; int reset, no code or data protect, no brownout detect,
; no watchdog, power-up timer, ext RC osc (~ 1kHz) + clkout
__CONFIG _MCLRE_OFF & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _EXTRC_OSC_CLKOUT

;*****
RESET CODE 0x0000 ; processor reset vector

;**** Initialisation
; setup port
movlw ~(1<<GP1) ; configure GP1 (only) as an output
banksel TRISIO
movwf TRISIO

```

```

; setup Timer0
movlw   b'11000101'   ; configure Timer0:
; --0-----         timer mode (T0CS = 0)
; ----0---         prescaler assigned to Timer0 (PSA = 0)
; -----101       prescale = 64 (PS = 101)
banksel OPTION_REG   ; -> increment at 4 Hz with 1 kHz clock
movwf  OPTION_REG

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

; repeat forever
goto  loop

END

```

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, not how fast.

## Power-On Reset

When we apply power to a PIC, we expect it to start executing whatever program is loaded into it, and to do so reliably, every time.

Unfortunately, it may not be that simple. “Apply power” is not the same as “instant on”. Real power supplies have a source impedance, and the circuits attached to them have some capacitance, so it takes some time for the power supply to ramp up to its final voltage. And while doing so, devices in the circuit will begin drawing current unevenly as they come to life, meaning that the power supply may initially be unsteady, taking some time to settle, as the circuit reaches equilibrium.

That can be a problem for devices, such as PICs, which have a minimum operating voltage. For example, the PIC12F629 requires a power supply of at least 2.0 V when running at a clock rate of 4 MHz or less. But at least 4.5 V is required for operation at frequencies above 10 MHz.

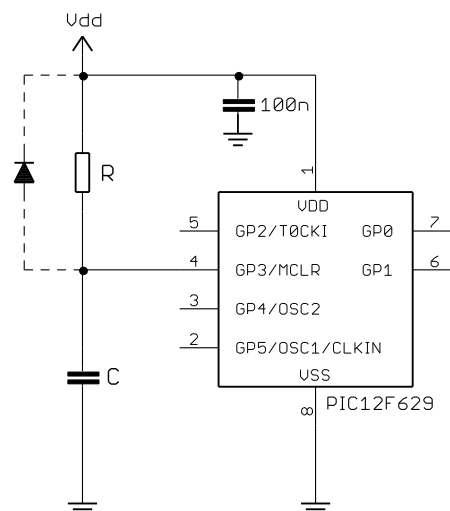
Below these voltages, the device may operate, but not reliably. Some parts of it may operate, while others fail. For example, the analog-to-digital converter (ADC) on the PIC12F675 requires at least 2.5 V, while other parts of the device (which is essentially the same as a 12F629) may operate down to 2.0 V.

And if the power supply is unstable while it ramps up, the PIC may drop in and out of operation while this minimum operating voltage region is crossed.

For reliable start-up, it is necessary to hold the PIC in a reset condition until the power supply has reached a high enough, and stable, voltage.

This was traditionally done by a simple RC circuit attached to the external  $\overline{\text{MCLR}}$  pin, as shown on the right.

The capacitor is initially discharged, so  $\overline{\text{MCLR}}$  is initially low, and remains low as the capacitor charges through the resistor. Values of R and C are chosen so that  $\overline{\text{MCLR}}$  goes high after



enough time has elapsed for the power supply to reach an adequate voltage and settle.

Microchip recommends that R be at least 1 kΩ for adequate ESD (electrostatic discharge) protection, and below 40 kΩ to avoid too large a voltage drop across the resistor.

The diode across the resistor is optional, being used to help discharge the capacitor more quickly when power is removed.

However, modern midrange PICs have less need for these external reset components, because they include a *power-up timer* (PWRT), which, if enabled, holds the device in reset for a nominal 72 ms from the initial *power-on reset* (POR) which occurs when power-on is detected.

To enable the power-up timer, clear the  $\overline{\text{PWRTE}}$  bit in the processor configuration word:

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

Setting  $\overline{\text{PWRTE}}$  to '1' *disables* the power-on timer.

To enable the power-up timer, use the symbol '`_PWRTE_ON`' in the `__CONFIG` directive.

To disable it, use '`_PWRTE_OFF`' instead.

So why would you ever want to disable the power-up timer?

Note that the PWRT begins to operate from the moment when the PIC detects a power-on condition, and for that to happen (for the PIC12F629), VDD has to rise from VSS at a minimum rate of 0.05 V/ms. If these conditions are not met, the power-on condition may not be detected; the power-on reset will not occur, and the PIC will not start properly. In this case, you would have to use an external circuit to hold the PIC in reset when power is applied.

Or, your power supply may take more than 72 ms to settle. And note that this is a nominal value – the actual PWRT delay on a PIC12F629 may be as short as 28 ms. If the power supply has not stabilised in this time, an external reset circuit should be used to hold the device in reset for longer.

Or, you may be using an external supervisory circuit, such as one of Microchip's MCP10X devices.

If, for any of these reasons, you are using an external circuit which holds the PIC in reset during power-up, it may appropriate to disable the internal power-up timer, so that there is only one source of power-up delay.

But most of the time, unless your circuit is operating in difficult power supply conditions, you can enable the power-up timer (as we have done so far) and, if you are using an external reset, use a 10 kΩ resistor between  $\overline{\text{MCLR}}$  and VDD.

If you are using the LP, XT or HS clock mode (which implies that you're probably using a crystal or resonator driven by the PIC's on-board oscillator circuitry), the *oscillator start-up timer* (OST) is invoked to give the crystal or resonator time to settle, after the PWRT delay completes.

The OST counts pulses on the OSC1 pin, and holds the device in reset for 1024 oscillator cycles. Hence, the OST delay depends on the clock speed. With a 4 MHz resonator, the OST delay is only 256 μs, while the delay with a 32.768 kHz crystal is 31 ms. Note that these delay times are nominal; after all, the reason the OST is invoked is that it takes a while for a crystal or resonator to begin stable oscillation, so in practice the delay times will probably be longer than this.

The OST is also used when the PIC wakes from sleep in LP, XT or HS clock mode, for the same reason – the oscillator is disabled while the device is in sleep mode, and takes a while to start and become stable.

Note that the OST is invoked whether or not PWRT is enabled. The only way to avoid the oscillator start-up delay is to use one of the EC, internal RC or external RC oscillator modes.

For fastest processor start-up at power-on, disable the power-up timer and use an external clock, avoiding both the PWRT and OST delays – and hope that you have a very fast-starting and stable power supply! But it's generally best to simply accept that your program won't start running for up to 100 ms after you turn the power on...

## Brown-out Detect

We've seen that the PIC should be held in reset during power-up, to avoid instability while the power supply is ramping up through the device's minimum operating voltage.

Similarly, the PIC's operation can become unreliable if the power supply voltage falls too far during normal operation – a condition known as a *brown-out*. It can happen when the load on the power supply varies; battery-powered systems can be susceptible to this, particularly when the batteries are running down, as well as industrial or automotive settings where large loads are switched in (think of the headlights of your car dimming when you use the starter motor).

In general, it is preferable to stop program execution as long as the brown-out situation persists, instead of risking unreliable operation; it's better to be able to recover cleanly after the brown-out, instead of not knowing what your program might do.

Most mid-range PICs provide a *brown-out detect* (BOD, also called *brown-out reset*, or BOR) facility, which, if enabled, will reset the device if the supply voltage falls below the brown-out detect voltage (between 2.025 V and 2.175 V on the PIC12F629), and hold it in reset until the voltage rises again. If the power-up timer is enabled (recommended if you are using BOD), the device will remain in reset for a further 72 ms after the brown-out condition clears – and if another brown-out occurs during this PWRT delay, it will be detected and the process will repeat.

To enable brown-out detect on the PIC12F629, set the **BODEN** bit in the processor configuration word:

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

Setting **BODEN** to '1' enables brown-out detection.

To enable BOD, use the symbol '`_BODEN_ON`' in the `__CONFIG` directive.

To disable it, use '`_BODEN_OFF`' instead.

### Detecting a brown-out reset

If a brown-out occurs, resetting the PIC and hence restarting your program, you may want your application to react to this, behaving differently to a power-on, watchdog timer, or other reset. In particular, if your program has restarted because of a brown-out, you may want it to try to continue doing whatever it was doing before the brown-out, instead of running through the full initialisation routine. Or you may wish to initialise external devices, which may also have been affected by the brown-out, and perhaps run through some system checks to ensure that everything is now ok. Or you might want to simply log the fact that a brown-out has occurred.

Fortunately, midrange PICs provide flags which allow us to detect and respond differently to both power-on and brown-out resets.

In the 12F629, these flags are contained in the power control register, PCON:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PCON	-	-	-	-	-	-	$\overline{\text{POR}}$	$\overline{\text{BOD}}$

The  $\overline{\text{POR}}$  (power-on reset status) flag is cleared when a power-on reset occurs, and is set if a brown-out reset occurs. It is unaffected by all other resets.

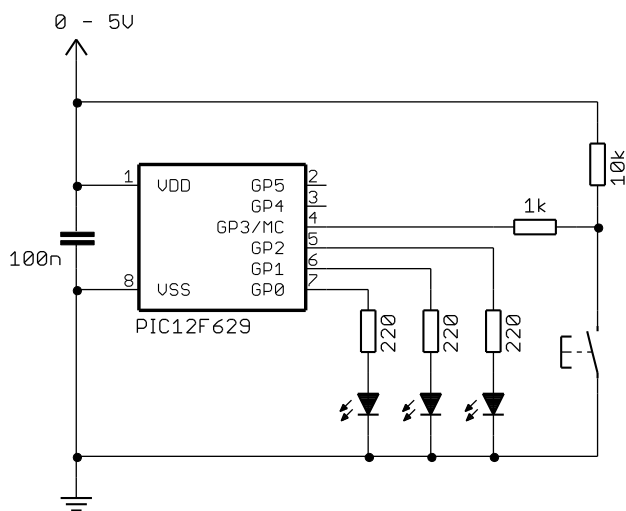
This means that, to use this flag to differentiate power-on from other resets, you must set  $\overline{\text{POR}}$  to '1' whenever a power-on reset occurs. Since all the other types of reset either set this bit or leave it unchanged, it will then only ever be '0' when a power-on reset has occurred.

Similarly, the  $\overline{\text{BOD}}$  (brown-out detect status) flag is cleared when a brown-out reset occurs, and is unaffected by all other resets.

To use this flag to differentiate brown-out from other resets, you must set  $\overline{\text{BOD}}$  to '1' following power-on. Since all the other resets leave this bit unchanged, it will only ever be '0' when a brown-out has occurred.

Since  $\overline{\text{BOD}}$  is unaffected by a power-on reset, its value is unknown when the device is first powered on. Therefore, the first flag you should test is  $\overline{\text{POR}}$ . If it is clear, you can be sure that a power-on reset has occurred, and you can then set both  $\overline{\text{POR}}$  and  $\overline{\text{BOD}}$ , ready for testing after subsequent resets.

An example may help to clarify this.



We'll use the circuit shown on the left, which you can implement using Microchip's Low Pin Count Demo Board, by making connections on the 14-pin header, as explained in [lesson 1](#): 'RA0' to 'RC0', 'RA1' to 'RC1' and 'RA2' to 'RC2'.

The program will simply turn on the LED on GP0, regardless of why the PIC had been reset (or powered on).

In addition, the LED on GP1 will be lit on power-on (and not for any other reset), and the LED on GP2 will indicate that a brown-out has occurred.

To generate a brown-out, you'll need to be able to vary your power supply voltage – or you could simply add a variable resistor in line with a fixed supply.

The pushbutton will be used to generate an external  $\overline{\text{MCLR}}$  reset. When this happens, only the LED on GP0 should light, because the reset is caused by neither power-on nor brown-out.

Brownout detection has to be enabled in the device configuration:

```

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

After the usual initialisation code, the first task is to test the  $\overline{\text{POR}}$  flag to see if a power-on reset has occurred:

```

; check for POR or BOD reset
banksel PCON
btfsc PCON,NOT_POR ; if power-on reset (NOT_POR = 0),
goto chk_bod

```

If this is a power-on reset, we should set the  $\overline{\text{POR}}$  and  $\overline{\text{BOD}}$  flags, to set them up for any subsequent resets (as discussed above), and light the POR LED:

```

bsf PCON,NOT_POR ; set POR and BOD flags for next reset
bsf PCON,NOT_BOD
bsf sGPIO,nP_LED ; turn on POR LED (shadow)

```

Note the use of a shadow copy of GPIO here. Since it is held in shared (unbanked) memory, updating the shadow register avoids the extra banksel directives we'd need if we were instead writing directly to GPIO – and it has the advantage of avoiding potential read-modify-write issues, as discussed [before](#).

Now we can reliably test for a brown-out reset:

```

chk_bod btfsc PCON,NOT_BOD ; if brown-out detect (NOT_BOD = 0)
goto main

```

and, if one has occurred, set the  $\overline{\text{BOD}}$  flag for next time, and light the BOD LED:

```

bsf PCON,NOT_BOD ; set BOD flag for next reset
bsf sGPIO,nB_LED ; turn on BOD LED (shadow)

```

Note that, if a power-on reset had occurred, this brown-out detect code will never be executed, because the earlier code sets the  $\overline{\text{BOD}}$  flag, whenever a power-on reset is detected.

Finally, regardless of the reason for the reset, we light the “on” LED:

```

main bsf sGPIO,nO_LED ; turn on "on" indicator LED (shadow)

movf sGPIO,w ; copy shadow to GPIO
banksel GPIO
movwf GPIO

```

and then do nothing (wait forever), until the next reset:

```

goto $ ; wait forever

```

If the pushbutton is pressed, generating a  $\overline{\text{MCLR}}$  reset, only this “on” LED will be lit.

### Complete program

Here is how these pieces fit together:

```

;*****
;
; Description: Lesson 8, example 6
;
; Demonstrates use of brown-out detect
; and differentiation between POR, BOD and MCLR resets
;

```

```

; Turns on POR LED only if power-on reset is detected          *
; Turns on BOD LED only if brown-out detect reset is detected  *
; Turns on indicator LED in all cases                          *
; (no POR or BOD implies MCLR, as no other reset sources are active) *
;                                                                *
;*****
;
; Pin assignments:
; GP0 - "on" indicator LED (always turned on)                 *
; GP1 - POR LED (indicates power-on reset)                     *
; GP2 - BOD LED (indicates brown-out detected)                 *
;*****

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

errorlevel -302          ; no warnings about registers not in bank 0

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

; pin assignments
constant  nO_LED=0      ; on indicator LED on GP0 (always on)
constant  nP_LED=1      ; POR LED on GP1 to indicate power-on reset
constant  nB_LED=2      ; BOD LED on GP2 to indicate brown-out

;***** VARIABLE DEFINITIONS
        UDATA_SHR
sGPIO   res 1           ; shadow copy of GPIO

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

;***** Initialisation
; configure port
movlw   ~(1<<nO_LED|1<<nP_LED|1<<nB_LED) ; configure LED pins as outputs
banksel TRISIO
movwf   TRISIO
; initialise port
banksel GPIO             ; start with all LEDs off
clrf    GPIO
clrf    sGPIO           ; update shadow

; check for POR or BOD reset
banksel PCON
btfsc   PCON,NOT_POR    ; if power-on reset (NOT_POR = 0),
goto    chk_bod
bsf     PCON,NOT_POR    ; set POR and BOD flags for next reset
bsf     PCON,NOT_BOD
bsf     sGPIO,nP_LED    ; turn on POR LED (shadow)

```

```

chk_bod btfsc   PCON,NOT_BOD   ; if brown-out detect (NOT_BOD = 0)
        goto   main
        bsf    PCON,NOT_BOD   ; set BOD flag for next reset
        bsf    sGPIO,nB_LED   ; turn on BOD LED (shadow)

;***** Main code
main    bsf    sGPIO,nO_LED   ; turn on "on" indicator LED (shadow)

        movf   sGPIO,w        ; copy shadow to GPIO
        banksel GPIO
        movwf  GPIO

        goto  $               ; wait forever

END

```

If you try this program, using a variable power supply, you should find that if you set the supply to say 4 V and apply power, the POR LED should light, along with the “on” LED.

If you then simulate a brown-out, by lowering the voltage until both LEDs turn off (at around 2 V; by this time they will be very dim, since the forward voltage of most normal-brightness LEDs is around 2 V), without taking the voltage all the way to zero, and then raise the voltage again, the BOD LED should light, along with the “on” LED, indicating that the brown-out was detected.

If you then turn off the power supply, and turn it back on again, the POR LED should light again, and not BOD, because this was a normal power-on, not a brown-out.

Finally, if you press the pushbutton, generating a  $\overline{\text{MCLR}}$  reset, while either the POR or BOD LED is lit, all the LEDs will go out while the button is pressed, and then only the “on” LED will come on, indicating that this reset was neither a power-on nor a brown-out.

If you are able to raise the power supply voltage very slowly from zero, you may be able to get the BOD LED to light, instead of POR, if the voltage rise is too slow to trigger a power-on reset.

That’s all for this lesson.

The [next lesson](#) will focus on comparators – the single comparator in the PIC12F629, and then in the [following lesson](#) we will introduce the 14-pin PIC16F684, which includes a dual comparator module.