

Introduction to PIC Programming

Programming Mid-Range PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 5: Reset, Power and Clock Options

[Mid-range lesson 8](#) looked at some of the more “hardware-related” aspects of the mid-range PIC architecture, including clock sources, the power-on reset conditions needed to successfully power-up a mid-range PIC, and brown-out resets and detection. This lesson covers the same topics, re-implementing the examples using Microchip’s XC8 compiler¹ (running in “Free mode”), as usual.

However, there is no to repeat all of the theory here, so you may wish to refer back to [mid-range lesson 8](#) for more detail.

In summary, this lesson covers:

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

Oscillator (Clock) Options

Although it is often appropriate to use the internal RC oscillator as the processor clock source, there are some situations where it is more appropriate to use some external clock circuitry, for reasons such as:

- *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 2](#), 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 bit rates, such as 38400 or 9600 ($1843200 = 48 \times 38400 = 192 \times 9600$).
- *Synchronising with other components.*
Clocking a number of devices from a common source, so that their outputs change synchronously, may simplify your design – although you need to be careful; clock signals which are subject to varying delays in different parts of your circuit will not be properly synchronised (a phenomenon known as *clock skew*), leading to unpredictable results.

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

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. Power consumption can be minimised by running the PIC at the slowest practical clock speed and power supply voltage. And for many applications, a high clock rate is unnecessary.

- *Faster operation.*

Most mid-range 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.

Mid-range 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 following table summarises the oscillator configuration options available for the PIC12F629, and the corresponding MPASM and XC8 symbols:

FOSC<2:0>	MPASM symbol	XC8 symbol	Oscillator configuration
000	_LP_OSC	FOSC_LP	LP oscillator
001	_XT_OSC	FOSC_XT	XT oscillator
010	_HS_OSC	FOSC_HS	HS oscillator
011	_EC_OSC	FOSC_EC	EC oscillator
100	_INTRC_OSC_NOCLKOUT	FOSC_INTRCIO	Internal RC oscillator + GP4
101	_INTRC_OSC_CLKOUT	FOSC_INTRCCLK	Internal RC oscillator + CLKOUT
110	_EXTRC_OSC_NOCLKOUT	FOSC_EXTRCIO	External RC oscillator + GP4
111	_EXTRC_OSC_CLKOUT	FOSC_EXTRCCLK	External RC oscillator + CLKOUT

Internal RC oscillator

Until now we've been using the 'FOSC_INTRCIO' configuration, where the internal RC oscillator provides a (nominally) 4 MHz processor clock (FOSC), driving the execution of instructions at approximately 1 MHz, and every pin is available for I/O.

In the 'FOSC_INTRCCLK' configuration, the instruction clock (FOSC/4) is output on the CLKOUT pin, to allow external devices to be synchronised with the PIC's internal RC clock.

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

You can use an oscilloscope to look at the signal on CLKOUT in 'FOSC_INTRCCLK' mode, but to verify that this signal is indeed the instruction clock, it's useful to toggle another pin as quickly as possible, for comparison with CLKOUT, using a simple program such as:

```

/*****
*   Description:      Lesson 5, example 1
*
*   Demonstrates CLKOUT function in Internal RC oscillator mode
*
*   Toggles a pin as quickly as possible
*   for comparison with 1 MHz CLKOUT signal
*****/

```

```

*****
*
*   Pin assignments:
*   GP2      = fast-changing output
*   CLKOUT   = 1 MHz clock output
*
*****/

#include <xc.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, 4 Mhz int clock with CLKOUT
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCCLK);

// Pin assignments
#define OUT      GP2          // fast-changing output

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

    // configure port
    TRISIO = 0;                // configure all pins (except GP3 and GP4/CLKOUT)
                               // as outputs

    /*** Main loop ***/
    for (;;)
    {
        OUT = ~OUT;           // toggle output pin as fast as possible
    }
}

```

The internal RC oscillator with CLKOUT configuration was selected by:

```

// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, 4 Mhz int clock with CLKOUT
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
         PWRTE_OFF & FOSC_INTRCCLK);

```

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

```

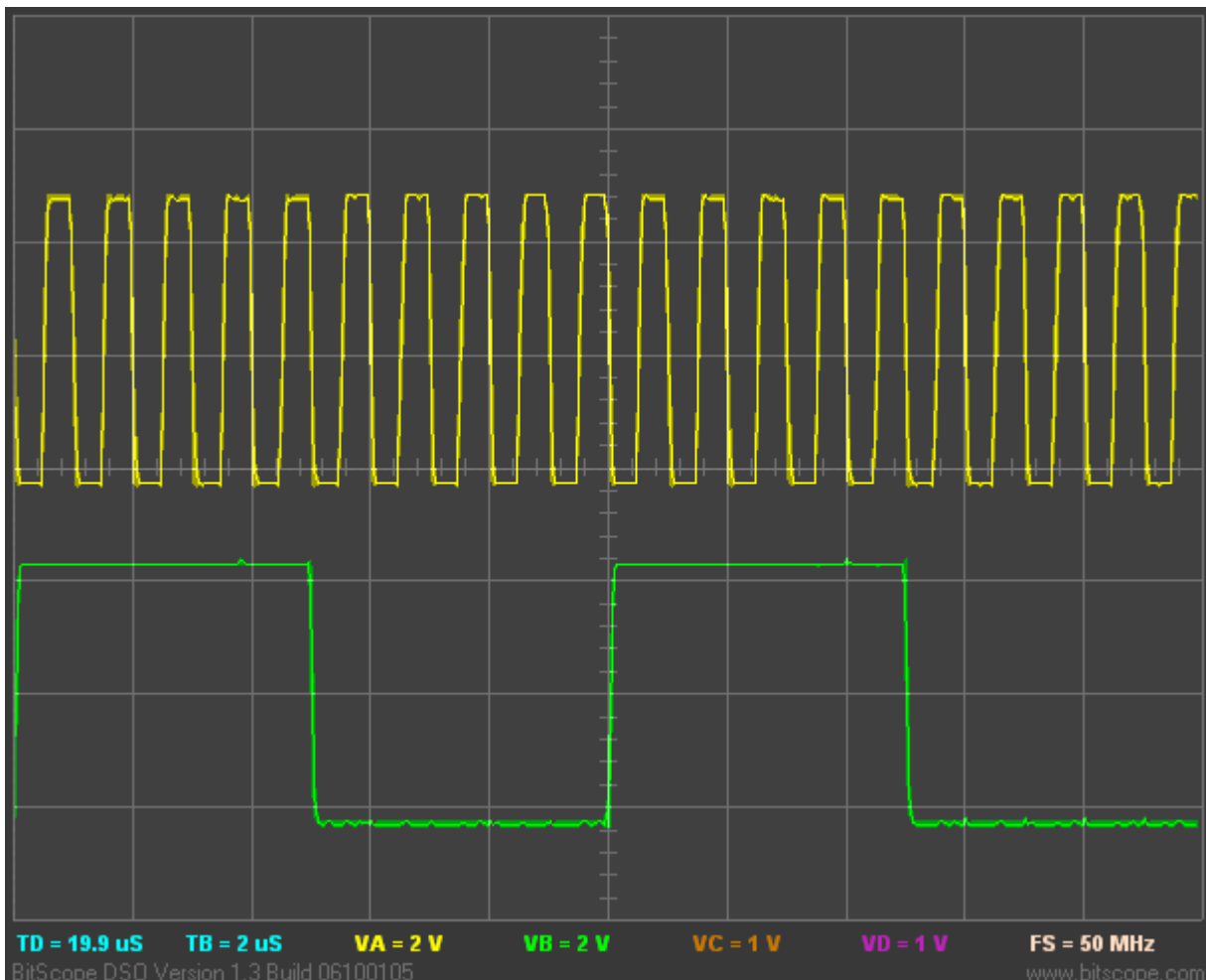
for (;;)
{
    OUT = !OUT;           // toggle output pin as fast as possible
}

```

The XC8 compiler, running in “Free mode”, generates code which toggles GP2 every five cycles, i.e. every 5 μ s, giving an output frequency of 100 kHz.

This is not as fast as we were able to toggle the pin in the example in [mid-range lesson 8](#) – demonstrating that for best results in time-critical code, it may be necessary to use assembler.

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 μ s, giving a frequency of 1 MHz, as expected.

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

These signals are available on pins 3 ('GP/RA/RB4') and 13 ('GP/RA/RB2') of the 16-pin header on the [Gooligum training board](#); the ground reference is pin 16 ('GND').

External clock input

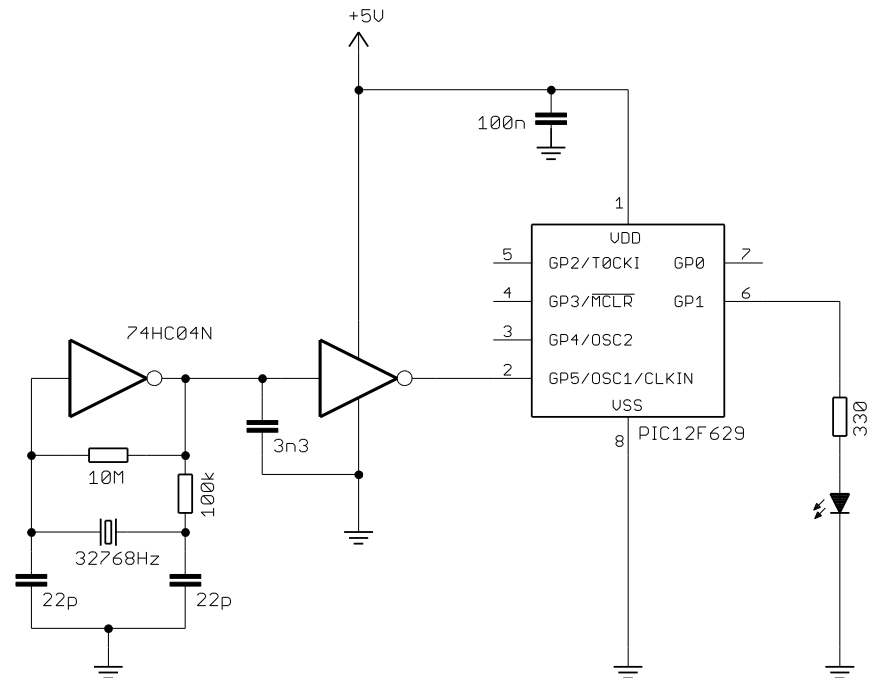
An external oscillator can be used as the PIC's clock source.

This is sometimes done so that the timing of various parts of a circuit is synchronised to the same clock signal. Or, your circuit may have an existing clock signal available, and it may make sense to use it if it is more accurate and/or stable than the PIC's internal RC oscillator – assuming you can afford the loss of one of the PIC's I/O pins.

To demonstrate the use of an external clock signal, we'll use a 32.768 kHz crystal oscillator, such as the one from [baseline assembler lesson 5](#), as shown in the circuit on the right.

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.

To implement this circuit using the [Gooligum training board](#), place a shunt in position 4 ("EC") of jumper block JP20, connecting the 32.768 kHz signal to CLKIN, and in JP3 and JP12 to enable the external $\overline{\text{MCLR}}$ pull-up resistor (not shown here) and the LED on GP1.



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

To illustrate the operation of this circuit, we can modify the crystal-driven LED flasher program developed in [lesson 2](#). 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 cycle at exactly 1 Hz (since $8192 = 32 \times 256$) – as is assumed in the example from [lesson 2](#).

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

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

and change the initialisation code from:

```
// configure Timer0
OPTION_REGbits.T0CS = 1;           // select counter mode
OPTION_REGbits.PSA = 0;           // assign prescaler to Timer0
OPTION_REGbits.PS = 0b110;        // prescale = 128
// -> incr at 256 Hz with 32.768 kHz input
```

to:

```
// configure Timer0
OPTION_REGbits.T0CS = 0;         // select timer mode
OPTION_REGbits.PSA = 0;           // assign prescaler to Timer0
OPTION_REGbits.PS = 0b100;      // prescale = 32
// -> incr at 256 Hz with 8192 Hz inst clock
```

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 2](#), modified as described above:

```

/*****
 *
 * Description: Lesson 5, example 2
 *
 * Demonstrates use of external clock mode
 * (using 32.768 kHz clock source)
 *
 * LED flashes at 1 Hz (50% duty cycle),
 * with timing derived from 8192 Hz instruction clock
 *
 *****/
 *
 * Pin assignments:
 *
 * GP1 = flashing LED
 *
 * CLKIN = 32.768 kHz signal
 *
 *****/

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

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

// Pin assignments
#define sFLASH sGPIO.GP1 // flashing LED (shadow)

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

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

    // configure port
    TRISIO = ~(1<<1); // configure GP1 (only) as an output

    // configure Timer0

```

```

OPTION_REGbits.T0CS = 0;           // select timer mode
OPTION_REGbits.PSA = 0;           // assign prescaler to Timer0
OPTION_REGbits.PS = 0b100;       // prescale = 32
                                   // -> incr at 256 Hz with 8192 Hz inst clock

/** Main loop */
for (;;)
{
    // TMR0<7> cycles at 1 Hz, so continually copy to LED
    sFLASH = (TMR0 & 1<<7) != 0;   // sFLASH = TMR0<7>

    GPIO = sGPIO.port;            // copy shadow to GPIO
}

```

Crystals and ceramic resonators

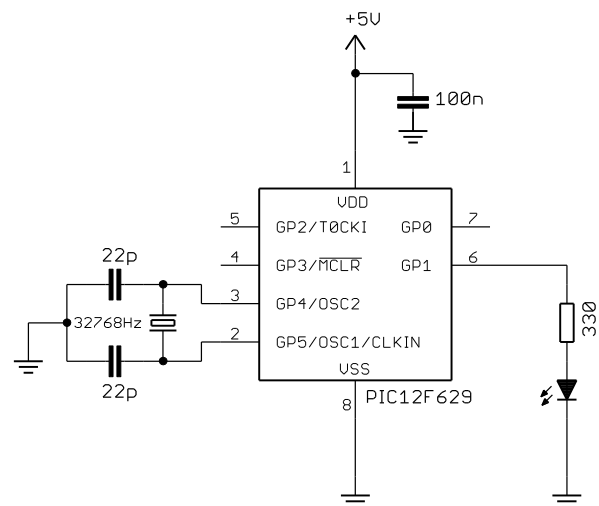
Generally, there is no need to build your own crystal oscillator; PICs include an oscillator circuit designed to drive crystals directly.

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.

You should consult the crystal or resonator manufacturer's data when selecting load crystals; those shown here are appropriate for a crystal designed for a load capacitance of 12.5 pF.

For some crystals it may be necessary to reduce the drive current by placing a resistor between OSC2 and the crystal, but in most cases it is not needed, and the circuit shown here (with the reset switch and pull-up omitted for clarity) can be used.

If you are using the [Gooligum training board](#), place shunts in position 2 ("32kHz") of JP20² and position 2 of JP21 ("32kHz"), connecting the 32.768 kHz crystal between OSC1 and OSC2, and close JP3 and JP12 to enable the external $\overline{\text{MCLR}}$ pull-up resistor (not shown here) and the LED on GP1.



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 most commonly used for crystals or ceramic resonators operating between 100 kHz and 4 MHz.

² You will find, with the Gooligum training board that the LED in this 32.768 kHz crystal example will flash, even with no shunt installed in JP20! This is because, when configured in `_LP_OSC` mode, the OSC1 input is very sensitive, and picks up crosstalk from the external 32.768 kHz signal on the board. If you want to prevent this effect, you can dampen the external 32.768 kHz signal by loading it with a 100 Ω resistor, placed between pin 1 of the 16-pin expansion header and ground, via the breadboard. The external clock example will still work with this resistor in place, and this 32.768 kHz crystal example will only work with shunts in the "32kHz" positions of JP20 and JP21 – as we'd expect.

‘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 (on the 12F629) of 20 MHz. The higher drive level means that 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, although it can also be used with other low-frequency crystals or resonators.

The circuit shown above can be used to operate the PIC12F629 at 32.768 kHz, giving low power consumption and an 8192 Hz instruction clock, which, as we have seen, is easily divided to create an accurate 1 Hz signal.

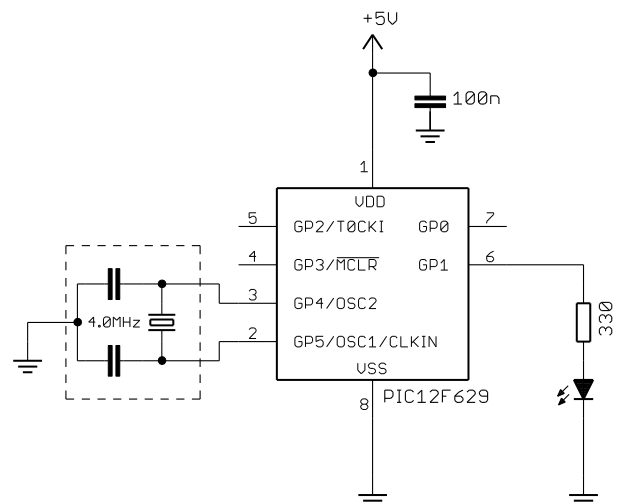
To flash the LED at 1 Hz, the program is exactly the same as for the external clock example above, except that the configuration statement must instead include the `FOSC_LP` option:

```
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, LP oscillator
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
        PWRTE_OFF & FOSC_LP);
```

Another 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 available in convenient 3-terminal packages which include appropriate loading capacitors, as shown in the circuit diagram (with the reset switch and pull-up omitted for clarity) on the right. The resonator package incorporates the components within the dashed lines.

If you have the [Gooligum training board](#), move the shunts to position 3 (“4MHz”) of JP20 and position 1 of JP21 (“4MHz”), connecting the 4.0 MHz resonator between OSC1 and OSC2, and leave JP3 and JP12 closed to enable the external `MCLR` pull-up resistor (not shown here) and the LED on GP1.



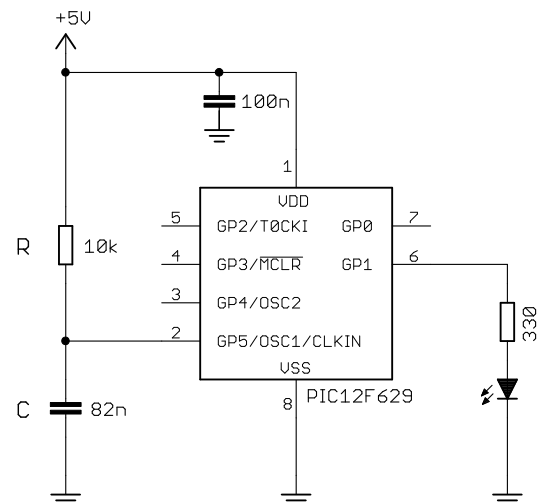
To test this circuit, you can change the ‘`FOSC_INTRCIO`’ configuration option to ‘`FOSC_XT`’ in the `__CONFIG()` macro in any program from the examples in any of the earlier lessons, since they all used a 4 MHz clock.

A good choice is the “flash an LED at exactly 1 Hz” program developed in [lesson 3](#), 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: mid-range PICs can use an oscillator based on an external resistor and capacitor, as shown (with the reset switch and pull-up omitted for clarity) on the right.

To implement this circuit using the [Gooligum training board](#), move the shunt to position 1 (“RC”) of JP20, connecting the 10 kΩ resistor and 82 nF capacitor to OSC1. Remove the shunt from JP21 and leave JP3 and JP12 closed, enabling the external $\overline{\text{MCLR}}$ pull-up resistor (not shown here) and the LED on GP1.



External RC oscillators, with appropriate values of R and C, can 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.

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 VDD, 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 $1.2 \times RC$.

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

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

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

Those values will give a period of approximately $1.2 \times 10 \times 10^3 \times 82 \times 10^{-9} \text{ s} = 984 \mu\text{s}$

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

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 'RCCLK' in the configuration statement.

```

/*****
*
* Description: Lesson 5, 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 ~256 Hz instruction clock
*
*****/
*
* Pin assignments:
* GP1 = flashing LED
* OSC1 = R (10k) / C (82n)
*
*****/

#include <xc.h>

/***** CONFIGURATION *****/
// ext reset, no code protect, no brownout detect, no watchdog,
// power-up timer enabled, ext RC oscillator (~ 1kHz) + clkout
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF &
        PWRTE_OFF & FOSC_EXTRCCLK);

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

    // configure port
    TRISIO = ~(1<<1); // configure GP1 (only) as an output

    // configure Timer0
    OPTION_REGbits.T0CS = 0; // select timer mode
    OPTION_REGbits.PSA = 0; // assign prescaler to Timer0
    OPTION_REGbits.PS = 0b101; // prescale = 64
    // -> incr at 4 Hz with 256 Hz inst clock

    /*** Main loop ***/
    for (;;)
    {
        // TMR0<1> cycles at 1 Hz, so continually copy to LED (GP1)
        GPIO = TMR0; // copy TMR0 to GPIO
    }
}

```

The “main loop” is only a single assignment statement – by far the shortest “flash an LED” program we have done, demonstrating that slowing the clock rate can simplify certain 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

As explained in greater detail in [mid-range lesson 8](#), to reliably start program execution on a mid-range (or any) PIC, it is necessary to hold the device in a reset condition until the power supply has reached a consistently high enough voltage.

This was traditionally done by a simple RC circuit attached to the external $\overline{\text{MCLR}}$ pin. However, there is often no need to use external reset components with modern mid-range PICs, 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.

The power-up timer is controlled by the $\overline{\text{PWRTE}}$ bit in the processor configuration word; setting $\overline{\text{PWRTE}}$ to '1' *disables* the power-on timer.

To enable it using XC8, include the symbol 'PWRTE_ON' in the `__CONFIG()` macro.

To disable it, use 'PWRTE_OFF' instead.

You may need to disable the power-up timer if your power supply takes more than 72 ms to settle. You should then use an external RC reset circuit, or an external supervisory circuit, such as one of Microchip's MCP10X devices, to hold the device in reset for longer. If so, it may be 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 k Ω 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 until it has counted 1024 oscillator cycles.

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

[Mid-range lesson 8](#) also explained that 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*. In general, it is preferable to stop program execution while 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.

Brown-out detection on the PIC12F629 is controlled by the **BODEN** bit in the processor configuration word; setting **BODEN** to '1' enables brown-out detection.

To enable BOD (or BOR) using XC8, use the symbol 'BOREN_ON' in the `__CONFIG()` macro.

To disable it, use 'BOREN_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. For example, 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.

Fortunately, mid-range 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**.

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.

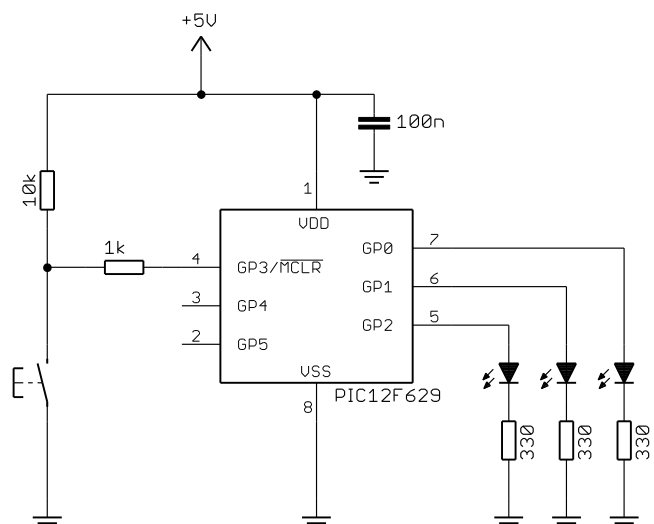
So to use this flag to differentiate brown-out from other resets, you must set $\overline{\text{BOD}}$ to '1' following power-on. Since all the other resets leave this bit unchanged, it will only ever be '0' when a brown-out has occurred.

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 right, which you can implement with the [Gooligum training board](#) by closing jumpers JP3, JP11, JP12 and JP13 to enable the pull-up resistor on GP3 and the LEDs on GP0, GP1 and GP2.

If you are using Microchip's Low Pin Count Demo Board, you can connect LEDs to GP0, GP1 and GP2, by making connections on the 14-pin header: 'RA0' to 'RC0', 'RA1' to 'RC1' and 'RA2' to 'RC2'.



The program will simply light 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.

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.

After enabling brownout detection in the device configuration:

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

After initialising TRISIO and clearing GPIO (so that all LEDs are initially off), the first task is to test the $\overline{\text{POR}}$ flag to see if a power-on reset has occurred. If so, 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:

```
if (!PCONbits.nPOR)           // if power-on reset (/POR = 0),
{
    PCONbits.nPOR = 1;        // set POR and BOD flags for next reset
    PCONbits.nBOD = 1;
    sP_LED = 1;              // enable POR LED (shadow)
}
```

A shadow copy of GPIO is used to avoid potential read-modify-write problems, as we have done [before](#).

Now we can reliably test for a brown-out reset, and, if one has occurred, set the $\overline{\text{BOD}}$ flag for next time, and light the BOD LED:

```
if (!PCONbits.nBOD)          // if brown-out detect (/BOD = 0)
{
    PCONbits.nBOD = 1;        // set BOD flag for next reset
    sB_LED = 1;              // enable 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 and copy the shadow register to the port:

```
// enable "on" indicator LED
sO_LED = 1;                   // (via shadow register)

// light enabled LEDs
GPIO = sGPIO.port;           // copy shadow GPIO to port
```

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

Finally, we simply wait until the next reset:

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

Complete program

Here is how these pieces fit together:

```

/*****
*
* Description: Lesson 5, 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)
*
*****/

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

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

// Pin assignments
#define sO_LED  sGPIO.GP0    // "on" indicator LED - always on (shadow)
#define sP_LED  sGPIO.GP1    // POR LED to indicate power-on reset (shadow)
#define sB_LED  sGPIO.GP2    // BOD LED to indicate brown-out (shadow)

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

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

    // configure port
    GPIO = 0;                    // start with all LEDs off
}

```

```

sGPIO.port = 0;           // update shadow
TRISIO = 0b111000;      // configure GP0, GP1 and GP2 as outputs

// check for POR or BOD reset
if (!PCONbits.nPOR)     // if power-on reset (/POR = 0),
{
    PCONbits.nPOR = 1;   // set POR and BOD flags for next reset
    PCONbits.nBOD = 1;
    sP_LED = 1;         // enable POR LED (shadow)
}
if (!PCONbits.nBOD)    // if brown-out detect (/BOD = 0)
{
    PCONbits.nBOD = 1;   // set BOD flag for next reset
    sB_LED = 1;         // enable BOD LED (shadow)
}

/** Main code */

// enable "on" indicator LED
sO_LED = 1;            // (via shadow register)

// light enabled LEDs
GPIO = sGPIO.port;    // copy shadow GPIO to port

// wait forever
for (;;)
    ;
}

```

To test this program, you will need a variable power supply.

If you have the [Gooligum training board](#), you can connect your power supply to Vdd and ground via pins 15 ('+V') and 16 ('GND') on the 16-pin expansion header.

You should find that if you set the supply to say 4 V and apply power, the POR LED (GP1) should light, along with the "on" LED (GP0)

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 (GP2) should light, indicating that the brown-out was detected. The "on" LED should light, as always, but not POR, because this was a brown-out, not a power-on reset..

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.

Summary

Most of the examples in this lesson did not require any new programming techniques; the first few being minor adaptations of programs from earlier lessons, with different processor configuration options, to select the oscillator mode being demonstrated.

However, the final example demonstrated that power-on and brown-out resets can be detected and responded to effectively, using the XC8 compiler – the detection code being simple and elegant, compared with the assembler version.

The next lesson (supplied with the [Gooligum baseline and mid-range training board](#), or available for purchase from www.gooligum.com.au) focuses on comparators – the single comparator in the PIC12F629, and the dual comparator module in the PIC16F684.