Introduction to PIC Programming

Programming Midrange PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 5: Reset, Power and Clock Options

Midrange lesson 8 looked at some of the more "hardware-related" aspects of the midrange PIC architecture, including clock sources, the power-on reset conditions needed to successfully power-up a midrange PIC, and brown-out resets and detection. This lesson covers the same topics, re-implementing the examples using "free" C compilers from HI-TECH Software: PICC-Lite and HI-TECH C[1] (in "Lite" mode), as usual.

However, there is no to repeat all of the theory here, so you may wish to refer back to midrange 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)

with examples for HI-TECH C and PICC-Lite.

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 baud rates, such as 38400 or 9600 (1843200 = 48 × 38400 = 192 × 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] PICC-Lite was bundled with versions of MPLAB up to 8.10. HI-TECH C (earlier known as "HI-TECH C PRO") was bundled with MPLAB 8.15 and later, although you should download the latest version from www.htsoft.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 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 following table summarises the oscillator configuration options available for the PIC12F629, and the corresponding MPASM and HI-TECH C symbols:

FOSC<2:0>	**MPASM symbol**	**HI-TECH C symbol**	**Oscillator configuration**
000	_LP_OSC	LP	LP oscillator
001	_XT_OSC	XT	XT oscillator
010	_HS_OSC	HS	HS oscillator
011	_EC_OSC	EC	EC oscillator
100	_INTRC_OSC_NOCLKOUT	INTIO	Internal RC oscillator + GP4
101	_INTRC_OSC_CLKOUT	INTCLK	Internal RC oscillator + CLKOUT
110	_EXTRC_OSC_NOCLKOUT	RCIO	External RC oscillator + GP4
111	_EXTRC_OSC_CLKOUT	RCCLK	External RC oscillator + CLKOUT

Internal RC oscillator

Until now we've been using the 'INTIO' 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 'INTCLK' 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 'INTCLK' mode.

You can use an oscilloscope to look at the signal on CLKOUT in 'INTCLK' 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                             *
*                                                                       *
```

```
*   Uses inline 500 ms delay routine                                    *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       GP2 - fast-changing output                                      *
*                                                                       *
************************************************************************/

#include <htc.h>


/***** CONFIGURATION *****/
// Config: ext reset, no code or data protect, brownout detect,
//         no watchdog, power-up timer enabled, 4MHz int clock with CLKOUT
__CONFIG(MCLREN & UNPROTECT & BOREN & WDTDIS & PWRTEN & INTCLK);

// Pin assignments
#define OUT     GPIO2           // fast-changing output on GP2


/***** GLOBAL VARIABLES *****/
unsigned char   sGPIO;          // shadow copy of GPIO


/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation ***/
    // configure port
    TRISIO = 0;             // config all pins (except GP3 and GP4) as outputs

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

The internal RC oscillator with CLKOUT configuration was selected by:

```
// Config: ext reset, no code or data protect, brownout detect,
//         no watchdog, power-up timer enabled, 4MHz int clock with CLKOUT
__CONFIG(MCLREN & UNPROTECT & BOREN & WDTDIS & PWRTEN & INTCLK);
```

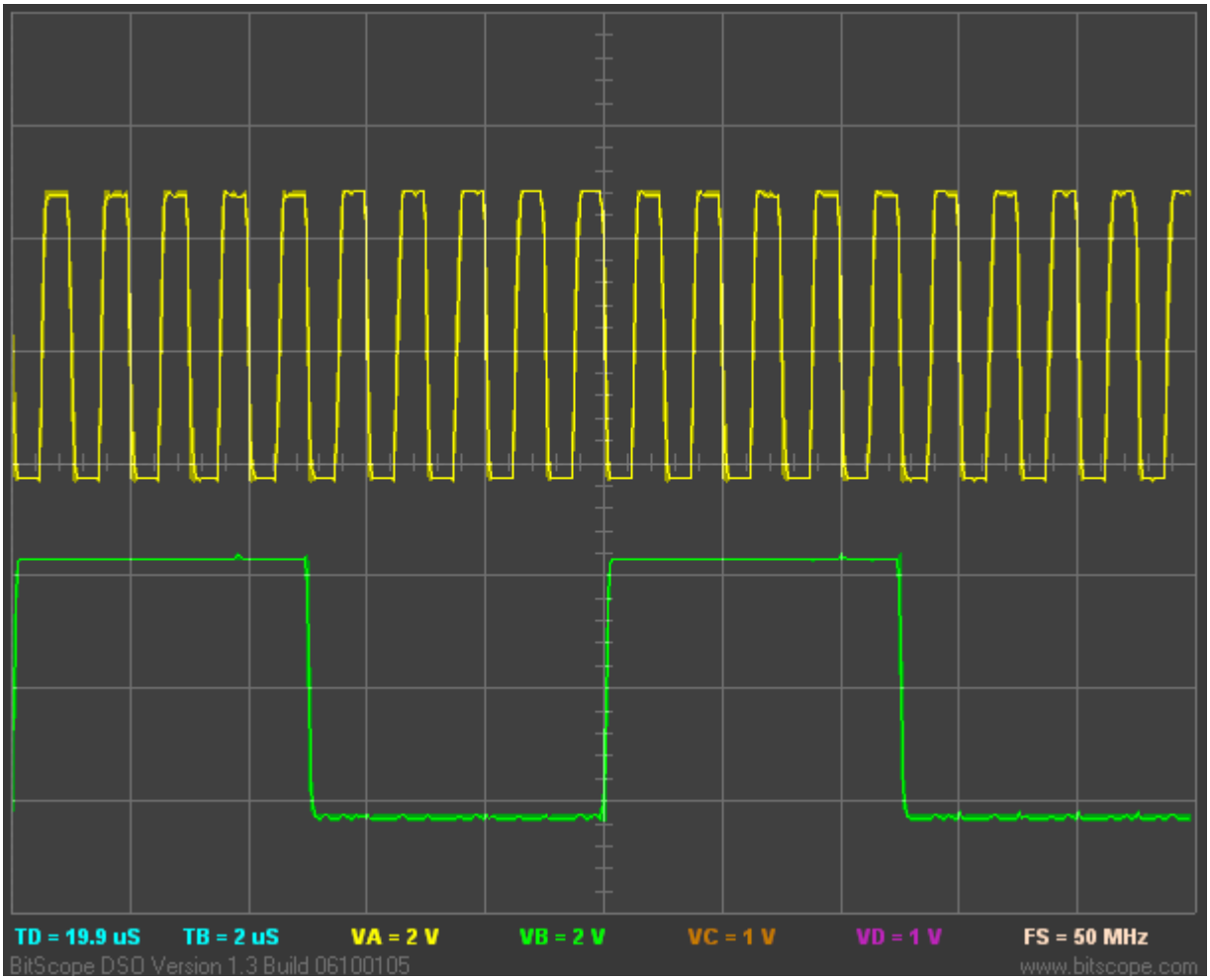
To toggle the GP2 pin as quickly as possible, the main loop was made as tight as possible:

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

Both of the HI-TECH C compilers generate code which toggles GP2 every five cycles, i.e. every 5 µs, giving an output frequency of 100 kHz.

This is not as fast as we were able to toggle the pin in the example in midrange lesson 8 – demonstrating that for best results in time-critical code, it can 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 µs, 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.

Comparisons

The following table compares the source code length and memory usage for this example by both HI-TECH C compilers and the corresponding assembler program from midrange lesson 8:

IntRC+CLKOUT

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	16	12	0
HI-TECH PICC-Lite	8	13	3
HI-TECH C PRO Lite	8	29	2

As usual, the C source code is only half as long as the assembler source, while the code generated by the PICC-Lite compiler, with optimisation enabled, is barely any larger than the hand-written assembly version.

But, as noted above, it was not able to toggle the output as quickly, demonstrating that tight loops written in assembler can be faster.

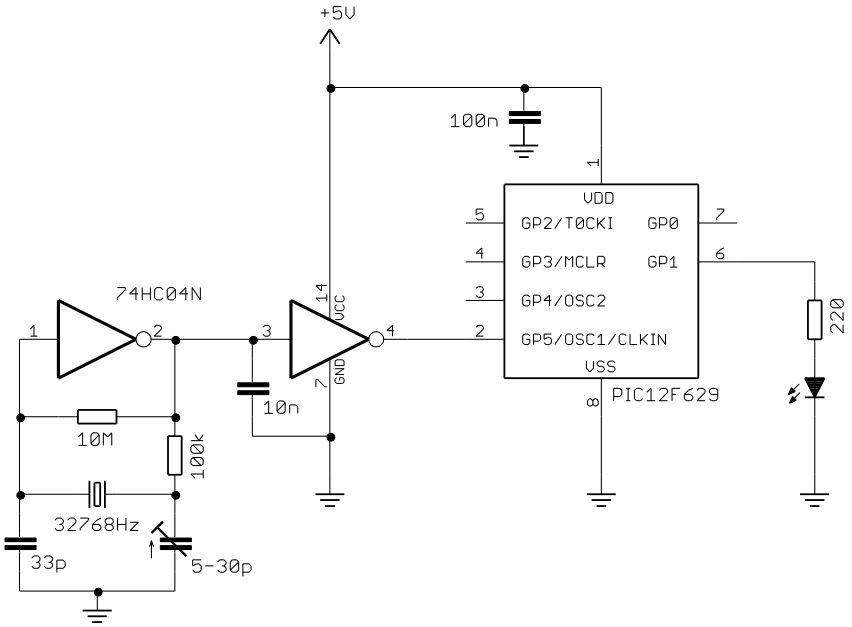
And as expected, HI-TECH C PRO generated much larger code, because it is running in "Lite" mode, which does not perform any optimisation.

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 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.

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

```
// Config: int reset, no code protect, no brownout detect, no watchdog,
//         power-up timer enabled, external clock
__CONFIG(MCLRDIS & UNPROTECT & BORDIS & WDTDIS & PWRTEN & EC);
```

and change the initialisation code from:

```
    OPTION = 0b11110110;        // configure Timer0:
             //--1-----             counter mode (T0CS = 1)
             //----0---             prescaler assigned to Timer0 (PSA = 0)
             //-----110             prescale = 128 (PS = 110)
                                //  -> increment at 256 Hz with 32.768 kHz input
```

to:

```
    OPTION = 0b11010100;   // configure Timer0:
             //--0-----        timer mode (T0CS = 0)
             //----0---        prescaler assigned to Timer0 (PSA = 0)
             //-----100        prescale = 32 (PS = 100)
                              //  -> increment 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 1Hz (50% duty cycle),                                *
*   with timing derived from 8192 Hz instruction clock                  *
*                                                                       *
************************************************************************
*   Pin assignments:                                                    *
*       GP1 - flashing LED                                              *
************************************************************************/

#include <htc.h>

// Config: int reset, no code protect, no brownout detect, no watchdog,
//         power-up timer enabled, external clock
__CONFIG(MCLRDIS & UNPROTECT & BORDIS & WDTDIS & PWRTEN & EC);

// Pin assignments
#define nLED    1                   // flashing LED on GP1

void main()
{
    unsigned char   sGPIO;          // shadow copy of GPIO

    // Initialisation
    TRISIO = ~(1<<nLED);            // configure LED pin (only) as output
    OPTION = 0b11010100;            // configure Timer0:
             //--0-----                 timer mode (T0CS = 0)
             //----0---                 prescaler assigned to Timer0 (PSA = 0)
             //-----100                 prescale = 32 (PS = 100)
                                    //  -> incr at 256 Hz with 8192 Hz inst clock

    // Main loop
    for (;;)
    {
        // TMR0<7> cycles at 1Hz, so continually copy to LED
        sGPIO = 0;                  // assume TMR<7>=0 -> LED off (shadow)
        if (TMR0 & 1<<7)            // if TMR0<7>=1
            sGPIO |= 1<<nLED;       //   turn on LED (shadow)

        GPIO = sGPIO;               // copy shadow to GPIO

    }   // repeat forever
}
```

Comparisons

Here is the resource usage summary for the "external clock" examples:

Flash_LED_32k_EC

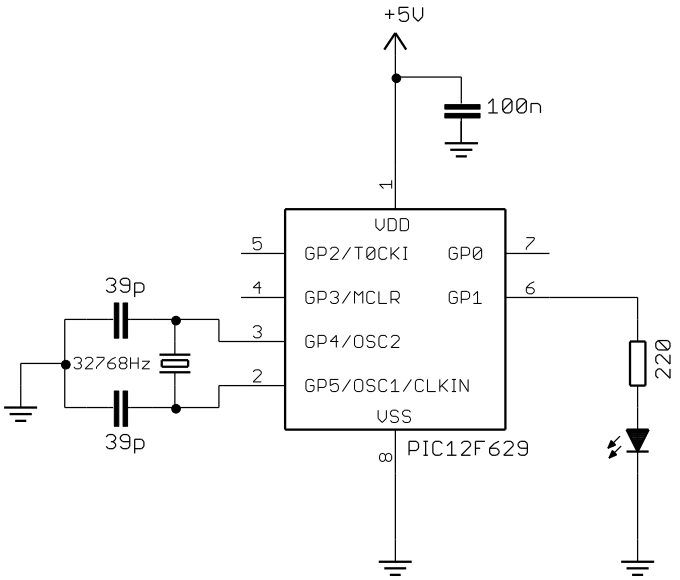
Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	21	16	1
HI-TECH PICC-Lite	12	19	3
HI-TECH C PRO Lite	12	26	3

Once again, the optimised code generated by the PICC-Lite compiler is only a little larger than the hand-written assembler version, while the C source code is around half the length of the assembler source.

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. For some crystals it may be necessary to reduce the drive current 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 most commonly used 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 (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 here 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.

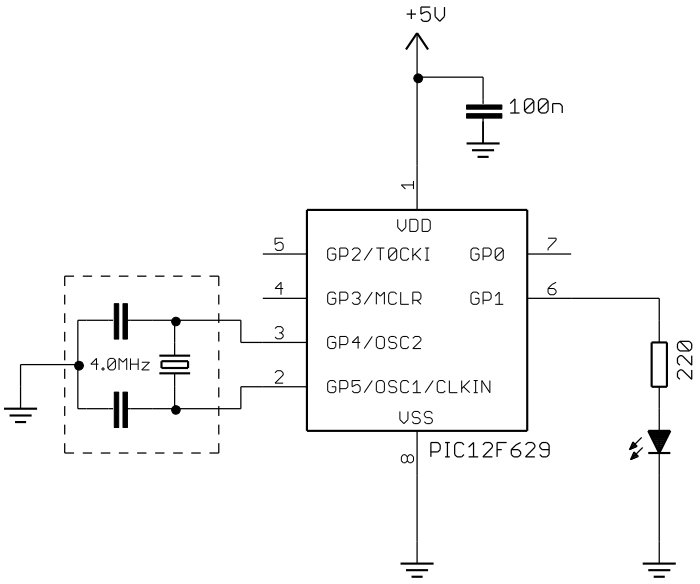
To flash the LED at 1 Hz, the program is exactly the same as for the external clock example above, except that the configuration statement must instead include the `LP` option:

```
// Config: int reset, no code protect, no brownout detect, no watchdog,
//         power-up timer enabled, LP oscillator
__CONFIG(MCLRDIS & UNPROTECT & BORDIS & WDTDIS & PWRTEN & 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 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 '`INTIO`' configuration option to '`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.

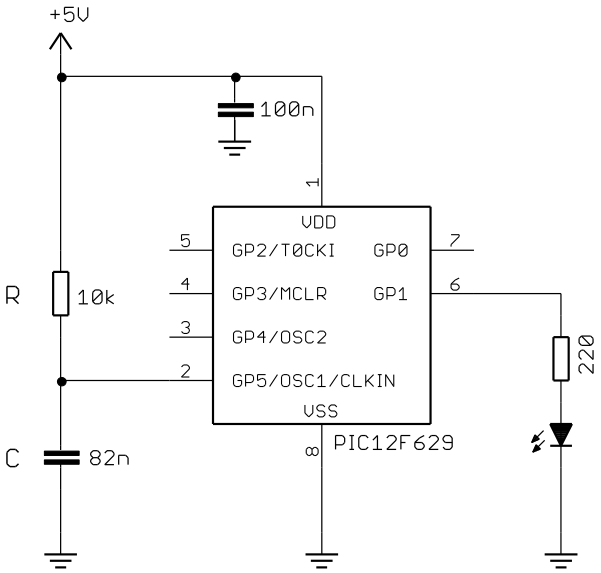
A good choice is the "flash an LED at exactly 1 Hz" program developed in lesson 3, since it will generate an output of exactly 1 Hz, given a processor clock of exactly 4 MHz, and so should benefit from the more accurate clock source.

External RC oscillator

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

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.



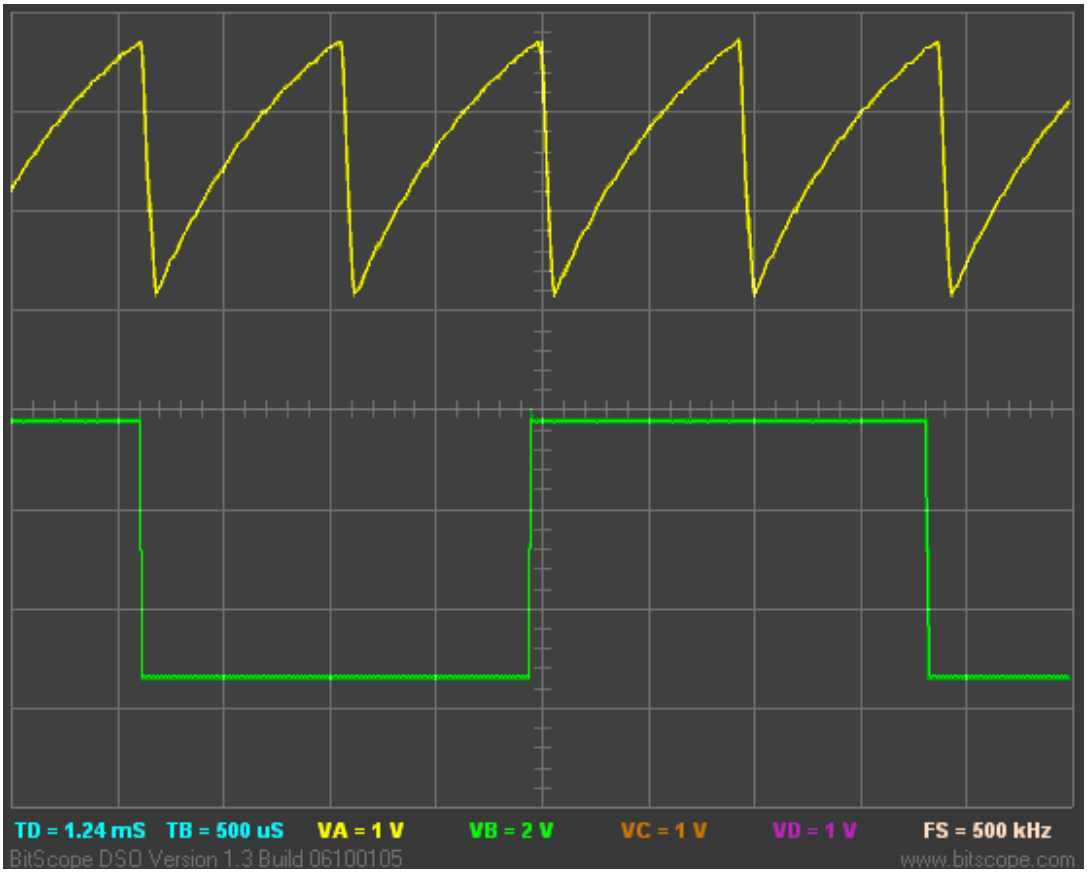
Microchip does not commit to a specific formula for the frequency (or period) of the external RC oscillator, only stating that it is a function of V_{DD}, R, C and temperature, and in some documents providing some reference charts. But for rough design guidance, you can assume the period of oscillation is approximately 1.2 × RC.

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

Those values R = 10 kΩ and C = 82 nF give a period of approximately $1.2 \times 10 \times 10^3 \times 82 \times 10^{-9}$ s = 984 µs.

Hence, we can expect that the clock frequency in the circuit above will be 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; you can see that 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 '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                                              *
*                                                                       *
************************************************************************/

#include <htc.h>

// Config: int reset, no code protect, no brownout detect, no watchdog,
//         power-up timer enabled, ext RC oscillator (~1 kHz) + clkout
__CONFIG(MCLRDIS & UNPROTECT & BORDIS & WDTDIS & PWRTEN & RCCLK);


void main()
{
    // Initialisation
    TRISIO = ~(1<<1);               // configure GP1 (only) as an output
    OPTION = 0b11010101;            // configure Timer0:
             //--0-----                 timer mode (T0CS = 0)
             //----0---                 prescaler assigned to Timer0 (PSA = 0)
             //-----101                 prescale = 64 (PS = 101)
                                    //  -> incr at 4 Hz with 256 Hz inst clock
    // Main loop
    for (;;)
    {
        // TMR0<1> cycles at 1 Hz, so continually copy to GP1
        GPIO = TMR0;                // copy TMR0 to GPIO

    }   // repeat forever
}
```

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 midrange lesson 8, to reliably start program execution on a midrange (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 midrange 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 HI-TECH C, include the symbol '`PWRTEN`' in the `__CONFIG()` macro.

To disable it, use '`PWRTDIS`' 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 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 V_{DD}.

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

Midrange 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 HI-TECH C, use the symbol '`BOREN`' in the `__CONFIG()` macro.

To disable it, use '`BORDIS`' 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, 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.

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.

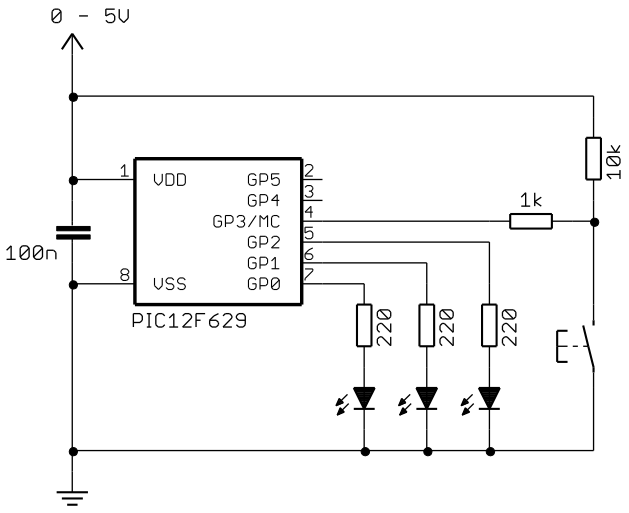
This means that, to use this flag to differentiate power-on from other resets, you must set $\overline{\text{POR}}$ to '1' whenever a power-on reset occurs. Since all the other types of reset either set this bit or leave it unchanged, it will then only ever be '0' when a power-on reset has occurred.

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 left, which you can implement using Microchip's Low Pin Count Demo Board, by making connections on the 14-pin header, as explained in midrange 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.

After enabling brownout detection in the device configuration:

```
// Config: ext reset, no code or data protect, brownout detect,
//         no watchdog, power-up timer enabled, 4MHz int clock
__CONFIG(MCLREN & UNPROTECT & BOREN & WDTDIS & PWRTEN & INTIO);
```

and initialising TRISIO and clearing GPIO (so that all LEDs are initially off), as usual, 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 (!POR)                   // if power-on reset (*POR = 0),
    {
        POR = 1;                //   set POR and BOD flags for next reset
        BOD = 1;
        sGPIO |= 1<<nP_LED;     //   turn on 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 (!BOD)                   // if brown-out detect (*BOD = 0)
    {
        BOD = 1;                //   set BOD flag for next reset
        sGPIO |= 1<<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:

```
    sGPIO |= 1<<nO_LED;         // turn on "on" indicator LED (shadow)

    GPIO = sGPIO;               // copy shadow to GPIO
```

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

Finally, we 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 <htc.h>

/***** CONFIGURATION *****/
// Config: ext reset, no code or data protect, brownout detect,
//         no watchdog, power-up timer enabled, 4MHz int clock
__CONFIG(MCLREN & UNPROTECT & BOREN & WDTDIS & PWRTEN & INTIO);

// Pin assignments
#define nO_LED  0           // "on" indicator LED on GP0 (always on)
#define nP_LED  1           // POR LED on GP1 to indicate power-on reset
#define nB_LED  2           // BOD LED on GP2 to indicate brown-out


/***** GLOBAL VARIABLES *****/
unsigned char   sGPIO;          // shadow copy of GPIO


/***** MAIN PROGRAM *****/
void main()
{
    /*** Initialisation ***/
    // configure port
    TRISIO = ~(1<<nO_LED|1<<nP_LED|1<<nB_LED);  // configure LED pins as outputs

    // initialise port
    GPIO = 0;                       // start with all LEDs off
    sGPIO = 0;                      //   update shadow

    // check for POR or BOD reset
    if (!POR)                   // if power-on reset (*POR = 0),
    {
        POR = 1;                //   set POR and BOD flags for next reset
        BOD = 1;
        sGPIO |= 1<<nP_LED;     //   turn on POR LED (shadow)
    }
    if (!BOD)                   // if brown-out detect (*BOD = 0)
    {
        BOD = 1;                //   set BOD flag for next reset
        sGPIO |= 1<<nB_LED;     //   turn on BOD LED (shadow)
    }

    /*** Main code ***/
    sGPIO |= 1<<nO_LED;         // turn on "on" indicator LED (shadow)

    GPIO = sGPIO;               // copy shadow to GPIO

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

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.

Comparisons

Here is the resource usage summary for the "POR and BOD demo" examples:

POR+BODdemo

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	35	27	1
HI-TECH PICC-Lite	20	31	3
HI-TECH C PRO Lite	20	55	2

Again, the same pattern: the optimised code generated by the PICC-Lite compiler is little larger than the hand-written assembler version, while the C source code is significantly shorter than the assembler source.

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 either of the HI-TECH C compilers – the detection code being simple and elegant, compared with the assembler version.

In fact, all of the examples could be expressed succinctly in C, as illustrated by the code length comparisons:

Source code (lines)

Assembler / Compiler	Example 1	Example 2	Example 3	Example 4	Example 5	Example 6
Microchip MPASM	16	21	21	61	16	35
HI-TECH PICC-Lite	8	12	12	22	7	20
HI-TECH C PRO Lite	8	12	12	22	7	20

As we have come to expect, the C source code is significantly shorter than the corresponding assembler source, being typically half as long. The difference is more extreme in example 4, reflecting the ease with which interrupts can be implemented using HI-TECH C.

And once again, the PICC-Lite compiler was able to generate optimised code almost as small as the hand-written equivalent, in every example:

Program memory (words)

Assembler / Compiler	Example 1	Example 2	Example 3	Example 4	Example 5	Example 6
Microchip MPASM	12	16	16	46	13	27
HI-TECH PICC-Lite	13	19	19	52	15	31
HI-TECH C PRO Lite	29	26	26	92	16	55

The programs generated by the C compilers continue to use more data memory than the assembler versions:

Data memory (bytes)

Assembler / Compiler	Example 1	Example 2	Example 3	Example 4	Example 5	Example 6
Microchip MPASM	0	1	1	5	0	1
HI-TECH PICC-Lite	3	3	3	7	2	3
HI-TECH C PRO Lite	2	3	3	7	2	2

However, this data memory use is still only a small fraction of the 64 bytes available on the 12F629.

The next lesson will revisit material from midrange lessons 9 and 10, focussing on comparators – the single comparator in the PIC12F629, and the dual comparator module in the PIC16F684.