

Introduction to PIC Programming

Mid-Range Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 1: Basic Digital Output

The [Baseline PIC Assembler](#) tutorial series introduced the baseline (12-bit) PIC architecture, using devices including the 8-pin digital-only PIC12F509 and 14-pin analog-capable PIC16F506. The series culminated in the development of a simple light meter with smoothed 2-digit decimal output on 7-segment LED displays. However, it was apparent that the limitations of the baseline architecture, such as the lack of interrupts, a maximum of 16 contiguous bytes of banked data memory, and the availability of only a single 8-bit timer, make it difficult to develop applications significantly more complex than this. The baseline architecture's limitations became especially evident when implementing the same examples in C, in the [Baseline PIC C Programming](#) tutorial series.

The mid-range (14-bit) PIC architecture overcomes many of these limitations, offering more memory, larger contiguous blocks of data memory, with simpler and less restricted memory access, more timers, greater flexibility in many areas, additional assembler instructions, a much greater range of peripherals, and support for interrupts – significant, because interrupts allow a different (better) approach to many programming problems, as we will see in later lessons.

This tutorial series introduces the mid-range architecture. Assembly language is used, as that is the best way to gain a thorough understanding of the PIC core and peripherals (languages like C or BASIC hide many of the implementation details, which can make life much easier for the programmer – but the aim here is to gain a good understanding of the underlying hardware).

These lessons assume some familiarity with the content covered in the Baseline PIC Assembler series. Although there is some repetition of material, wherever a topic has been covered in the baseline tutorials, it is described more briefly here, along with a reference to the baseline lesson where the topic was introduced. This approach is practical because the mid-range architecture builds on the baseline architecture we are already familiar with; most of the concepts, and nearly all the assembler instructions, are the same.

This lesson introduces one of the simplest of the mid-range PICs – the PIC12F629. It then goes on to describe basic digital output by lighting and flashing LEDs, as covered in lessons [1](#) and [2](#) of the baseline assembler tutorial series.

In summary, this lesson covers:

- Introduction to the PIC12F629
- Simple digital output to LEDs
- Using loops to create delays
- Using shadow registers to avoid the 'read-modify-write' problem

Getting Started

These tutorials assume that you are using a Microchip PICkit 2 or PICkit 3 programmer and either the [Gooligum Baseline and Mid-range PIC Training and Development Board](#) or Microchip's Low Pin Count Demo Board, with Microchip's MPLAB 8 or MPLAB X integrated development environment. But it is of course possible to adapt these instructions to a different programmers and/or development boards.

See [lesson 0](#) and [baseline lesson 1](#) for more details

As mentioned, we're going to start with one of the simplest of the mid-range PICs – the 8-pin PIC12F629. It is roughly equivalent to the PIC12F509, introduced in [baseline lesson 3](#), but in addition to simple digital I/O, it also includes an analog comparator, a 16-bit timer, and a 128-byte EEPROM. However, it does not include an analog-to-digital converter, nor does it include any advanced peripherals or interfaces. That makes it a good chip to start with; we'll look at the additional features of more advanced mid-range PICs in later lessons.

In summary, for this lesson you should ideally have:

- A PC running Windows (XP, Vista or 7), with a spare USB port
- Microchip's MPLAB 8 IDE software
- A Microchip PICkit 2 or PICkit 3 PIC programmer
- The Gooligum mid-range training board
- A PIC12F629-I/P microcontroller (supplied with the Gooligum training board)

Introducing the PIC12F629

When working with any microcontroller, you should always have on hand the latest version of the manufacturer's data sheet, which, for the 12F629, can be downloaded from www.microchip.com.

The data sheet for the 12F629 also covers the 12F675, which is essentially the same device, with the addition of an analog-to-digital converter (ADC).

The features of various 8-pin PICs are summarised in the following table:

Device	Memory (words or bytes)			Timers		Analog		Clock rate (max MHz)
	Program	Data	EEPROM	8-bit	16-bit	Comp-arators	ADC inputs	
12F508	512	25	0	1	0	0	0	4
12F509	1024	41	0	1	0	0	0	4
12F510	1024	38	0	1	0	1	3	8
12F519	1024	41	64	1	0	0	0	8
12F609	1024	64	0	1	1	1	0	20
12F615	1024	64	0	2	1	1	4	20
12F629	1024	64	128	1	1	1	0	20
12F675	1024	64	128	1	1	1	4	20
12F683	2048	128	256	2	1	1	4	20
12F1501	1024	64	0	2	1	1	4	20
12F1822	2048	128	256	2	1	1	4	32
12F1840	4096	256	256	2	1	1	4	32

The 12F629 has only a little more data memory than the 12F509, but it is arranged differently, as shown in the following register map:

PIC12F629 Registers

Address	Bank 0	Address	Bank 1
00h	INDF	80h	INDF
01h	TMR0	81h	OPTION_REG
02h	PCL	82h	PCL
03h	STATUS	83h	STATUS
04h	FSR	84h	FSR
05h	GPIO	85h	TRISIO
06h		86h	
09h		89h	
0Ah	PCLATH	8Ah	PCLATH
0Bh	INTCON	8Bh	INTCON
0Ch	PIR1	8Ch	PIE1
0Dh		8Dh	
0Eh	TMR1L	8Eh	PCON
0Fh	TMR1H	8Fh	
10h	T1CON	90h	OSCCAL
11h		91h	
		94h	
18h		95h	WPU
19h	CMCON	96h	IOC
1Ah		97h	
		98h	
		99h	VRCON
		9Ah	EEDATA
		9Bh	EEADR
		9Ch	EECON1
		9Dh	EECON2
		9Eh	
1Fh		9Fh	
20h	General Purpose Registers	A0h	Map to Bank 0 20h – 5Fh
5Fh			
60h			
7Fh			
		DFh	
		E0h	
		FFh	

The 12F509’s register map, and the concept of banked register access, was described in [baseline lesson 3](#).

A few differences are immediately apparent:

In the mid-range PICs, each bank consists of 128 registers, compared with only 32 registers in the baseline architecture.

The first 32 addresses in each register bank are used for special function registers (SFRs); the remaining 96 addresses in each bank are available for general-purpose registers (GPRs), allowing much larger contiguous blocks of data memory to be created.

This means that, although the 12F629 has less data memory than the 16F506 (64 bytes compared with 72 bytes), the larger address space of the mid-range architecture means that the 12F629’s 64 bytes are mapped into a single bank, not spread across four banks, as they would be in the baseline architecture.

Note that the GPRs are mapped into both banks, meaning that all data memory in the 12F629 is shared, not banked.

Another significant difference from the baseline architecture is that most SFRs appear in only in one bank or the other. This means that, *when accessing SFRs on mid-range PICs, it is very important to ensure that the correct bank is selected.*

As described in [baseline lesson 3](#), the `banksel` assembler directive will reliably set the bank selection bits for the specified register address. Some SFRs are grouped, so that once the correct bank is selected for one of them, you can be sure that the bank selection will not need to be changed before accessing other registers in the group. But if you are ever in doubt, use `banksel`. And remember that just because two registers happen to be in the same bank in the 12F629, it may not be guaranteed to be true in other mid-range PICs.

Bank selection is controlled by the RP0 bit in the STATUS register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
STATUS	IRP	RP1	RP0	\overline{TO}	\overline{PD}	Z	DC	C

If $RP0 = 0$, bank 0 is selected; if $RP0 = 1$, bank 1 is selected.

The RP1 and IRP bits are unused on the 12F629; they are used in mid-range devices, such as the 16F690, which have four register banks. In devices with four banks, RP0 and RP1 are used in combination to select the bank for direct register access, while IRP is used select the bank for indirect register access (see [lesson 14](#)) – necessary because FSR, being 8-bits wide, can only point to one of 256 registers, but a four-bank device has 512 register addresses (128 addresses in each bank).

This is much more convenient than the bank selection scheme used in the baseline architecture, where bits in the FSR register were used, which meant that indirect register access could not be done separately from direct register access – a limitation which makes it very difficult for C compilers to implement banked array access on baseline devices, as we saw in [baseline C lesson 7](#). The mid-range architecture has no such limitation.

The remaining bits in the STATUS register, \overline{TO} , \overline{PD} , Z, DC and C, are equivalent to their counterparts in the baseline architecture.

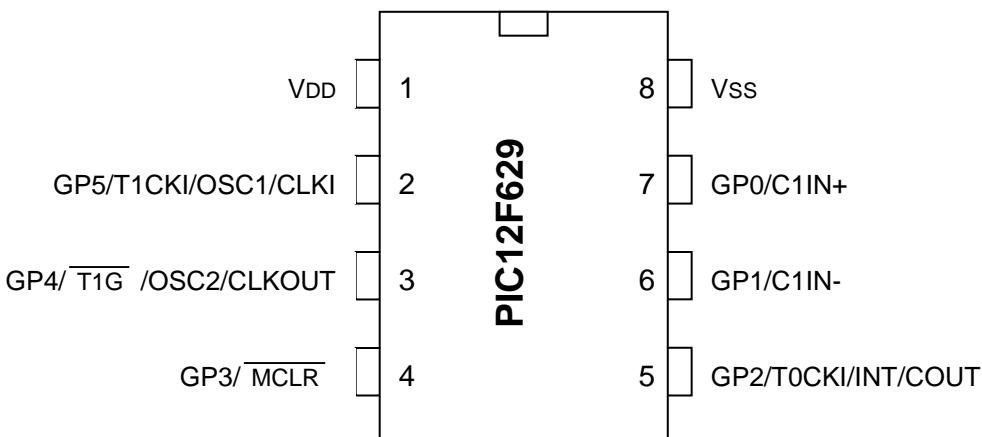
The TRIS (called TRISIO on the 12F629) and OPTION registers are no longer accessed through special instructions, but appear in the register map and are directly accessible, in the same way as any other register. Importantly, this means that these registers are now readable, as well as writable, making it possible to update individual bits.

Note that the OPTION register is called OPTION_REG on mid-range PICs, because “option” is a reserved word in MPASM.

The working register, ‘W’ (equivalent to the ‘accumulator’ in some other microprocessors), is not mapped into memory, and so does not appear in the register map.

PIC12F629 Input/Output

Like the 12F509, the 12F629 provides six I/O pins in an eight-pin package:



VDD is the positive power supply.

VSS is the “negative” supply, or ground. All of the input and output levels are measured relative to VSS.

In most circuits, there is only a single ground reference, at 0 V, and VSS will be connected to ground.

The power supply voltage (VDD, relative to VSS) can range from 2.0 V to 5.5 V, although at least 3.0 V is needed if the clock rate is greater than 4 MHz, and at least 4.5 V is needed to run the PIC at more than 10 MHz.

A *bypass capacitor*, typically 100 nF and preferably ceramic, should be placed between VDD and VSS, as close to the chip as practical, to provide transient power as the current drawn by the PIC changes, and to limit the effect of noise on the power rails. You may find that you can “get away” without using a bypass capacitor, particularly in a small battery-powered circuit. But figuring out why your PIC keeps randomly resetting itself is hard, while 100 nF capacitors are cheap, so include them in your designs!

The remaining pins, GP0 to GP5, are the I/O pins. They are used for digital input and output, except for GP3, which can only be an input. The other pins – GP0, GP1, GP2, GP4 and GP5 – can be individually set to be inputs or outputs.

Note however that each I/O pin has one or more functions that can be assigned to it, such as a comparator output, or a counter input. As we will see later, in some cases these alternate functions need to be disabled before a pin can be used for digital I/O.

Taken together, the six I/O pins comprise the general-purpose I/O *port*, or GPIO port.

If a pin is configured as an output, the output level is set by the corresponding bit in the GPIO register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
GPIO			GP5	GP4	GP3	GP2	GP1	GP0

Setting a bit to ‘1’ outputs a ‘high’ on the corresponding pin; setting it to ‘0’ outputs a ‘low’.

If a pin is configured as an input, the input level is represented by the corresponding bit in the GPIO register. If the input on a pin is high, the corresponding bit reads as ‘1’; if the input pin is low, the corresponding bit reads as ‘0’.

The TRISIO register controls whether a pin is set as an input or output:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TRISIO			TRISIO5	TRISIO4		TRISIO2	TRISIO1	TRISIO0

To configure a pin as an input, set the corresponding bit in the TRISIO register to ‘1’. In the input state, the PIC’s output drivers are effectively disconnected from the pin.

To configure a pin as an output, clear the corresponding TRISIO bit to ‘0’.

By default, each pin is an ‘input’; the TRISIO register is set to all ‘1’s when the PIC is powered on.

Note that TRISIO<3> is greyed-out. Clearing this bit will have no effect because, as mentioned above, the GP3 pin is always an input.

When configured as an output, each I/O pin on the 12F629 can source or sink up to 25 mA – enough to directly drive an LED, without needing an external transistor.

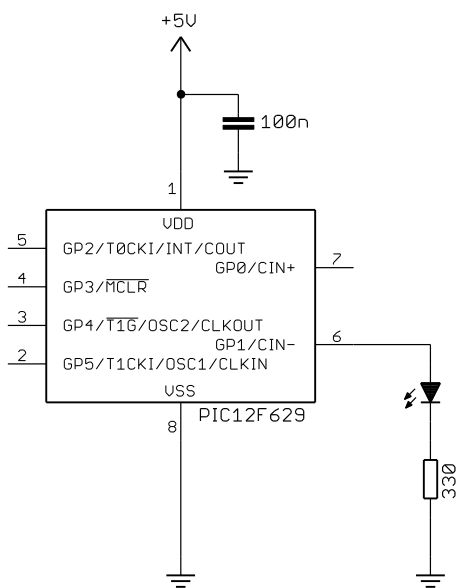
In total, the GPIO port can source or sink up to 125 mA.

Example 1: Turning on an LED

We'll start the same way that we did in [baseline lesson 1](#), by simply lighting a single LED, connected to one of the 12F629's digital I/O pins.

This might appear to be a trivial task, but if you can do something as simple as lighting an LED, you will have proven that you have a functioning circuit, that your PIC code is correct, that you have properly set up an appropriate development environment, and that you can use it effectively to assemble your code and load it into the PIC. When you have achieved all that, you have a firm base to build on.

The complete circuit looks like this:



As you can see, besides the PIC, there isn't very much needed at all.

The power supply should be at least 3 V to properly light the LED. A 5 V supply is assumed in these lessons, reflecting the default voltage provided by the PICKit 2 and PICKit 3 programmers.

A 330 Ω resistor, in series with the LED, is shown here, because that is the value used on the Gooligum training board. But you can choose any value for this resistor, as long as it limits the LED current to no more than 25 mA – the maximum rated current for each pin.

If you are using the [Gooligum training board](#), plug your PIC12F629 into the top section of the 14-pin IC socket – the section marked '12F'¹. Connect a shunt across the jumper (JP12) on the LED labelled 'GP1', and ensure that every other jumper is disconnected.

Plug your PICKit 2 or PICKit 3 programmer into the ICSP connector on the training board, with the arrow on the board aligned with the arrow on the PICKit, and plug the PICKit into a USB port on your PC. The PICKit 2 or PICKit 3 can supply enough power for this circuit, so there is no need to connect an external power supply.

If you have the board, refer back to [lesson 1](#) to see how to build this circuit, either by soldering a resistor and to the demo board, or by.

As we saw in [baseline lesson 1](#), the Microchip Low Pin Count Demo Board provides four LEDs, but they cannot be used directly with 8-pin PICs, such as the 12F629. You should refer back to that lesson to see how to add an LED to the prototyping area or make a connection to GP1 via the 14-pin header on the Microchip board.

With this simple circuit in place, and connected to your PC via a PICKit 2 or PICKit 3 programmer (assuming you are using the recommended development environment), it's time to move on to programming!

¹ Note that, although the PIC12F629 comes in an 8-pin package, **it will not work** in the 8-pin '10F' socket. You must install it in the '12F' section of the 14-pin socket.

The [baseline tutorial series](#) explained how to use the MPLAB 8 or MPLAB X environment to create a new assembler project. If you are not familiar with either version of MPLAB, you should follow the instructions in [baseline lesson 1](#), but selecting the 12F629 in the project wizard, instead of the 12F509.

If you choose to use a Microchip-supplied code template, you should choose ‘12F629TMPO.ASM’ in the ‘...\MPASM Suite\Template\Object’ (for MPLAB 8) or ‘...\mpasmx\templates\Object’ (for MPLAB X) directory. But since this template provides a framework for a number of features, including interrupts, which are not covered in this lesson, it is probably best not to include a copy of the template code, but to instead start with an empty file.

If you are using MPLAB 8, after finishing the project wizard, you can create a new (empty) file and add it to your project by selecting the “Project → Add New File to Project...” menu item (also available under the “File” menu, or by right-clicking in the project window), browsing to the project directory, typing a name (ending in ‘.asm’) for the new file, and then clicking “Save”.

Or, if you are using MPLAB X, there are a number of ways to create a new source file and add it to your project, but a simple way is to right-click “Source Files” in the project tree, and select “New → ASM File...”. Enter a name for your new file, select ‘.asm’ as the extension, then click on “Finish”.

To begin writing your program, double-click the assembler source file in the project window. A text editor window will open; it will either be blank, or showing the Microchip-supplied template code (if you created your file from a copy of it), in which case you will need to edit the template code, deleting some parts and changing others, to make it similar to the code presented below.

The MPLAB text editor is aware of PIC assembler (MPASM) syntax and will colour-code text, depending on whether it’s a comment, assembler directive, PIC instruction, program label, etc.

As we did in the [baseline tutorial series](#), we’ll begin each program with a block of comments, giving the name of the program, modification date and version, who wrote it, and a general description of what it does. The template code includes a “Files required” section. This is useful in larger projects, where your code may rely on other modules; you can list any dependencies here. We’ll also document what processor the code is written for, and how each pin is used – and anything else which will help anyone working on this code who needs to understand what the program does, and how.

MPASM comments begin with a ‘;’. They can start anywhere on a line. Anything after a ‘;’ is ignored by the assembler.

For example:

```
;*****
;
;   Filename:      MA_L1-Turn_on_LED.asm
;   Date:         1/5/12
;   File Version:  1.2
;
;   Author:       David Meiklejohn
;   Company:     Gooligum Electronics
;
;*****
;
;   Architecture:  Mid-range PIC
;   Processor:    12F629
;
;*****
;
;   Files required: none
;
;*****
```

```

;
; Description: Lesson 1, example 1
;
; Turns on LED. LED remains on until power is removed.
;
;*****
;
; Pin assignments:
; GP1 = indicator LED
;
;*****

```

Next we need to tell MPLAB what processor we're using:

```

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

```

The first line tells the assembler which processor to assemble for. It's not strictly necessary, as it is set in MPLAB (configured when you selected the device in the project wizard). MPLAB displays the processor it's configured for at the bottom of the IDE window; see the screen shot above. Nevertheless, you should always use the `list` directive at the start of your assembler source file, in case you have accidentally selected the wrong processor in MPLAB. If there is a mismatch between the `list` directive and MPLAB's setting, MPASM will warn you and you can correct the problem.

The next line uses the `#include` directive which causes an *include file* (p12F629.inc, located in the '...\MPASM Suite' directory) to be read by the assembler. This file sets up aliases, or *labels*, for all the features of the 12F629, so that we can refer to registers etc. by name (e.g. 'GPIO') instead of numbers, as was explained in [baseline lesson 6](#).

So to correctly specify which processor (such as 12F629) is to be used, you need to select that processor when you set up the project in MPLAB and include appropriate `list` and `include` directives in the assembler source.

Next the processor configuration is set:

```

__CONFIG    _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
             _PWRTE_ON & _INTRC_OSC_NOCLKOUT

```

[this directive must be written as a single line in the assembler source code]

Mid-range PICs have one or more "configuration words" (sometimes referred to as *fuses*), mapped outside the user program memory space, which define a number of aspects of the processor's configuration. Like the baseline PICs, the 12F629 has a single configuration word.

The `__CONFIG` directive is used to define the value(s) to be loaded into the configuration word(s). It is usually used with labels (defined in the processor's include file) representing values which are intended to be ANDed together to set or clear the configuration bits corresponding to the options being selected. We'll examine these in greater detail in later lessons, but briefly the options being selected here are:

- `_MCLRE_ON`
Enables the external reset, or "master clear" ($\overline{\text{MCLR}}$) on pin 4.

If external reset is enabled, pulling this pin low will reset the processor. If disabled, the pin can be used as an input: GP3. That's why, in the circuit diagram, pin 4 is labelled "GP3/MCLR"; it can be either an input pin or an external reset, depending on the setting of this configuration bit.

The Gooligum training board includes a pushbutton which will pull pin 4 low when pressed, resetting the PIC if external reset is enabled. The PICKit 2 and PICKit 3 are also able to pull the

reset line low, allowing MPLAB to control $\overline{\text{MCLR}}$ (if enabled) – useful for starting and stopping your program.

So unless you need to use every pin for I/O, it's a good idea to enable external reset by including `'_MCLRE_ON'` in the `__CONFIG` directive.

- `_CP_OFF`
Turns off program memory code protection.

When your code is in production and you're selling PIC-based products, you may want to prevent others (such as competitors) from accessing your code. If you specify `_CP_ON`, the program memory will be protected, meaning that if someone tries to use a PIC programmer to read it, all they will see are zeros.

- `_CPD_OFF`
Turns off data memory code protection.

The 12F629 includes "EEPROM" (more correctly, "flash") data memory, which is separate to the register file address space, and is accessed indirectly through special function registers. This memory is non-volatile; it retains its contents when the PIC is powered off. EEPROM data may be considered to be an integral part of the program, and worthy of protection. If you specify `_CPD_ON`, the EEPROM memory will be protected; its contents cannot be accessed by an external PIC programmer. Or the EEPROM may be used to hold data, such as system configuration or logged data, which the user should be able to access, even if the program code is protected. To provide this flexibility, program and data (EEPROM) memory are protected independently.

- `_BODEN_OFF`
Disables brown-out detection.

The PIC's operation can become unreliable if the supply voltage drops too low, which can happen during a *brown-out*, when the supply voltage sags, but does not fall quickly to zero. The 12F629 has brown-out detect circuitry, which will reset the PIC in a brown-out situation, if `_BODEN_ON` is selected. But if your power supply is not likely to suffer from brown-outs, you can leave this feature disabled.

- `_WDT_OFF`
Disables the watchdog timer.

As we saw in [baseline lesson 7](#), the watchdog timer provides a means of automatically restarting a crashed program, or to regularly wake the device from sleep. Although the watchdog timer is very useful in a production environment, it can be a nuisance when prototyping, so it is best left disabled to begin with.

- `_PWRTE_ON`
Enables the power-up timer.

When a power supply is first turned on, it can take a while for the supply voltage to stabilise, during which time the PIC's operation may be unreliable. If the power-up timer is enabled, the PIC is held in reset (it does not begin running the user program) for some time, nominally 72 ms, after the supply voltage reaches a minimum level.

For reliable operation, you should leave this option enabled, unless you are using an external supervisor circuit, which monitors system voltages and controls the PIC's external reset.

- `_INTRC_OSC_NOCLKOUT`
Selects the internal RC oscillator as the clock source, with no clock output.

PICs can be clocked in a number of ways, as we saw for the 12F509 in [baseline lesson 7](#) and the 16F506 in [baseline lesson 8](#). The 12F629 supports the same clock options as the 16F506, although without the ability to select the frequency of the internal 'RC' oscillator, which on the

12F629 always runs at a nominal 4 MHz. It is not as accurate or stable as an external crystal, but has the advantage of not needing any external components and leaves all of the PIC's pins free for I/O, unless the instruction clock (one quarter of the processor clock rate, i.e. 1 MHz, or 1 µs per instruction, given a 4 MHz processor clock) is output on CLKOUT.

To turn on an LED, we don't need accurate timing. And there is no need to make the clock signal available externally, so the `_INTRC_OSC_NOCLKOUT` option is appropriate for this application.

If you have based your project on the Microchip-supplied template code, you will see that the next sections in the template relate to defining variables and initialising the EEPROM with data. Since we do not need to use variables or the EEPROM in this example, you can safely delete these sections.

The next section of the template code refers to the oscillator calibration value:

```

;-----
; OSCILLATOR CALIBRATION VALUE
;-----
OSC          CODE      0x03FF

```

The `CODE` directive is used to introduce a *section* of program code.

The `0x03FF` after `CODE` is an address in hexadecimal (signified in MPASM by the '0x' prefix). Program memory on the 12F629 extends from `0000h` to `03FFh`. This `CODE` directive is telling the linker to place the section of code that follows it at `0x3FF` – the very top of the 12F629's program memory.

However, in this case, there *is* no code following this first `CODE` directive. Instead, this is simply a marker to remind us that the oscillator calibration value is held, as an instruction, at the top of program memory.

Like the 12F509, the speed of the internal RC oscillator in the 12F629 can be varied over a small range by changing the value of the `OSCCAL` register, to compensate for variability in the manufacturing process. Microchip tests every 12F629 in the factory, and calculates the value which, if loaded into `OSCCAL`, will make the oscillator run as close as possible to 4 MHz. This calibration value is inserted into the instruction placed at the top of the program memory (`0x3FF`), which is:

```
retlw k
```

where 'k' is the calibration value inserted in the factory.

A value like this, which is embedded in an instruction, is referred to as a *literal*.

As explained in [baseline lesson 3](#), the `retlw` instruction is used to exit a subroutine, returning a value in the `W` register to the code which called the subroutine – “return with literal in **W**”.

This is different from the scheme used in the baseline architecture, where the instruction at the top of program memory, which loads the calibration value into `W`, is the first instruction executed when the PIC is reset, and program execution “wraps around” at the start of memory.

Instead, in the mid-range architecture, the *reset vector*, where program execution begins, is always at the start of memory: address `0000h`.

On a PIC12F629, the user program, beginning at `0000h`, can choose to *call* the calibration “subroutine” (consisting of a single `retlw` instruction, as above) at the end of program memory, to “look up” the correct oscillator calibration for this device. Or, if the internal RC oscillator is not being used, or if exact timing is not important, the calibration instruction can simply be ignored.

In the baseline examples, we used the `res` directive in a construct like this:

```

RESET      CODE      0x3FF          ; processor reset vector
           res        1            ; holds internal RC cal value, as a movlw k

```

to reserve the program memory used by the calibration instruction, ensuring that it could not be overwritten by the user program. This is not necessary when using the default, Microchip-supplied linker script for the 12F629, because that script (unlike the ones that Microchip supply for the baseline PICs) declares the memory used by the calibration instruction to be “protected”, so that it will not be overwritten.

Therefore, there is no need to include a `CODE` directive, like either of those above, in our program. It is only useful for documentation, but that is not really necessary, since we can adequately comment the code which loads `OSCCAL` – see below. But of course, how you choose to comment your code is very much a matter of personal style.

The next sections in the Microchip-supplied template consist of code used to implement an *interrupt service routine (ISR)* (to be introduced in [lesson 6](#)) and some code to jump around the ISR. Since we are not using interrupts in this example, these sections can be deleted.

Since we are using the internal RC oscillator, we should start the program by calibrating it:

```
RESET    CODE    0x0000          ; processor reset vector
          ; calibrate internal RC oscillator
          call    0x03FF          ; retrieve factory calibration value
          banksel OSCCAL          ; (stored at 0x3FF as a retlw k)
          movwf   OSCCAL          ; then update OSCCAL
```

Because program execution begins at address `0x0000`, this address is specified in the `CODE` directive, placing this code section at the start of program memory, so that it will be executed whenever the PIC is powered on or reset. Another way to say this is that the *program counter*, which points to the next instruction to be executed, is initialised to `0x0000` when the PIC is reset.

This code section is labelled ‘`RESET`’ here, but you can use any label you want, as long as it’s not a reserved word and is not the name of any other code section in your program.

Next the oscillator calibration value is retrieved, by using the ‘`call`’ instruction (“**call** subroutine”) to call the calibration instruction at the end of program memory, which returns with the factory calibration value in `W`, as described above. Note again that this scheme is different from that used in the baseline devices.

The calibration value can then be written to the `OSCCAL` register, but before doing so, the bank selection bits must be configured to allow it to be accessed. As mentioned above, this is an important difference between the baseline and mid-range architectures. On mid-range devices, such as the 12F629, you must ensure that the correct bank is selected when accessing special function registers. The best way to ensure this, avoiding errors and making your code more portable, is to use the ‘`banksel`’ directive, as shown in the code above, and as explained in [baseline lesson 3](#).

Finally, the ‘`movwf`’ instruction – “**move W to file register**” – is used to copy (“move”, in Microchip-speak) the factory calibration value, held in `W`, into the `OSCCAL` register.

At this point, all the preliminaries are out of the way. The processor has been specified, the configuration set, and the oscillator calibration value updated.

Next it is usual to initialise special function registers, to configure the PIC’s ports and peripherals appropriately.

In this case, we need to configure the GP1 pin as an output:

```

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

```

Recall that, to configure a pin as an output, the corresponding bit in the TRISIO register must be cleared; by default the TRIS bits are set to '1', meaning that all pins are configured as inputs at power-up.

The first instruction, 'movlw' – “**move literal to W**” – loads a value into W.

We could have written this instruction as:

```

movlw  b'111101'      ; configure GP1 (only) as an output

```

Note that to specify a binary number in MPASM, the syntax b'*binary digits*' is used, as shown.

This binary value, when loaded into TRISIO, will configure GP1 as an output, leaving the remaining pins configured as inputs.

However, it is often clearer to make use of expressions containing symbols defined in the processor include file, such as 'GP1', instead of writing binary constants. For example, the expression '~ (1<<GP1)' is equivalent to the binary constant b'1111101' (only six bits of this value need be specified in the instruction above, because the top two bits of TRISIO are unused). Another advantage of using symbols is that mistyping a symbol is likely to be picked up by the assembler, while mistyping a binary constant is likely to be missed, making the use of symbols less error-prone.

Having loaded the correct value into W, the 'movwf' instruction is used to write it to TRISIO. And, of course, banksel is used to select the bank containing TRISIO, before it is accessed.

*Note: The tris instruction is **not** used to write to the TRIS registers on mid-range devices. The TRIS registers are accessed using general instructions, such as movwf.*

To make GP1 output a 'high', we have to set bit 1 of GPIO to '1'.

This could be done by:

```

banksel GPIO
movlw  1<<GP1      ; set GP1 high
movwf  GPIO

```

using the 'movlw' and 'movwf' instructions we have already seen.

The remaining bits in GPIO are cleared, but since the other pins are all inputs, it doesn't matter, in this example, what their corresponding GPIO bits are set to.

However, in many cases you will want to set or clear a single bit, while leaving the other bits in a register unchanged. This can be done with the bit set and clear instructions:

'bsf f,b' sets bit 'b' in register 'f' to '1' – “**bit set file register**”.

'bcf f,b' clears bit 'b' in register 'f' to '0' – “**bit clear file register**”.

These instructions, and any like them, which operate by reading a register, modifying its contents, and then writing the changed value back to the register, can create problems when used with port registers, such as GPIO. This is referred to as the *read-modify-write* problem, and is explained in more detail in [baseline lesson 2](#). It can happen because, in the mid-range and baseline architectures, whenever an instruction reads a port register, the external pins are read, not the internal “output latch” which had been

written to. This means that, if an output is slow to change because of a capacitive load, or is being held low or high by an excessive external load, the value read may not match the value written to it. And that can lead to unexpected results, when using instructions such as ‘bsf’ and ‘bcf’.

However, in this simple example, it is very unlikely that there will be any problem with simply turning on a single output, since we are not making any fast changes (and hence capacitive loading is not an issue), we are not changing multiple pins in the same port using sequential instructions (not giving a pin time to change, before being read by the next instruction) and there is no significant load on the pin. So it is safe to use:

```
banksel GPIO
bsf      GPIO,GP1      ; set GP1 high
```

If we leave it there, when the program gets to the end of this code, it will continue executing whatever instructions happen to be in the rest of the program memory; not what we want! So we need to get the PIC to just sit doing nothing, with the LED still turned on, until it is powered off.

What we need is an “infinite loop”, where the program does nothing but loop back on itself, indefinitely. Such a loop could be written as:

```
here     goto     here
```

‘here’ is a label representing the address of the `goto` instruction.

‘goto’ is an unconditional branch instruction. It tells the PIC to **go to** a specified program address.

This code will simply go back to itself, always. It’s an infinite, do-nothing, loop.

A shorthand way of writing the same thing, that doesn’t need a unique label, is:

```
goto     $          ; loop forever
```

‘\$’ is an assembler symbol meaning the current program address.

So this line will always loop back on itself.

Finally, at the end of your program source, you must include an ‘END’ directive.

If you put together all the pieces of code presented above, and assemble it, the assembler will give you a couple of messages like:

```
Message[302] C:\...\MA_L1-TURN_ON_LED.ASM 49 : Register in operand not in bank 0. Ensure that bank bits are correct.
```

These messages are generated whenever your code references a register which is not in bank 0, to remind you that you should be taking care to set the bank selection bits correctly. Since we have been taking care to ensure that the bank selection bits are correct, it can be annoying to see these messages – particularly in a larger program, where there will be many more of them. And worse, having a large number of unnecessary messages can make it easy to miss more important messages and warnings.

Luckily, messages and warnings can be disabled, using the ‘errorlevel’ directive:

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

This should be placed toward the beginning of your program.

Complete program

Putting together all the above, here's our complete assembler source for turning on an LED:

```

;*****
;
; Description: Lesson 1, example 1
;
; Turns on LED. LED remains on until power is removed.
;
;*****
;
; Pin assignments:
;   GP1 = indicator LED
;
;*****

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, 4 Mhz int clock
__CONFIG   _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

;***** RESET VECTOR *****
RESET CODE 0x0000          ; processor reset vector
; calibrate internal RC oscillator
call 0x03FF              ; retrieve factory calibration value
banksel OSCCAL           ; (stored at 0x3FF as a retlw k)
movwf OSCCAL             ; then update OSCCAL

;***** MAIN PROGRAM *****

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


;***** Main code
; turn on LED
banksel GPIO
bsf GPIO,GP1           ; set GP1 high

; loop forever
goto $

END

```


Now that you have the complete assembler source, you can build the application, which involves assembling the source files to create object files, and then linking the object files to build the executable

To allow the program to run, click on the  icon, or select the “Programmer → Release from Reset” menu item.

The LED should now light up!

If you are using MPLAB X, you must first ensure that your PICkit 2 or PICkit 3 is selected as the hardware (programmer) tool in the project properties window, which you can open by right-clicking your project in the Projects window and selecting “Properties”, or simply click on the “Project Properties” button on the left side of the Project Dashboard.

To program the PIC and run your program (in a single operation):


- Right-click your project in the Projects window, and select “Run”, or
- Select the “Run → Run Main Project” menu item, or
- Press ‘F6’, or
- Click on the “Make and Program Device” button in the toolbar: 


Whichever of these you choose, you should see output messages ending in:

Running target...

The LED on GP1 should now light.

Being able to build, program and run in a single step, by simply pressing ‘F6’ or clicking on the “Make and Program Device” button is very useful, but what if you don’t want to automatically run your code, immediately after programming?

If you want to avoid running your code, click on the “Hold in Reset” toolbar button  () before programming. You can now program your PIC as above.

Your code won’t run until you click the reset toolbar button again, which now looks like  and is now tagged as “Release from Reset”.

Example 2: Flashing an LED (50% duty cycle)

Having lit a single LED, the next step is to make it flash.

Although it is often preferable to make use of timer-driven interrupt routines (as we will see in [lesson 6](#)) to do something like flashing an LED in the “background”, the simplest approach is to simply light the LED, wait for some time by using a fixed delay, toggle the LED, wait again, and then repeat.

Or, if the LED is going to be on half the time (on for the same period that it is off, for a 50% *duty cycle*), we can simply continue to repeatedly toggle the LED, following a single fixed delay, as expressed in the following pseudo-code:

```
start with LED off
repeat
    delay 500 ms
    toggle LED
done
```

Note that the 500 ms delay gives a total flash period of 1 s, meaning that the LED is flashing at 1 Hz.

But first, you’ll need to create a new project. It makes sense to base it on the project and code you created in example 1; one method for doing this is given in [baseline lesson 2](#).

The configuration sections of the code (specifying the device and its configuration) remain the same, but of course you should update the comments to reflect this new project.

If you want really accurate timing, you'd use a crystal or external clock source, but the internal RC oscillator is good enough for simple LED flashing. Nevertheless, to make the LED flash timing as accurate as possible, it's important to include the oscillator calibration code at the start of your program.

To generate the delay, we need to make the PIC "do nothing" for some amount of time, and, as explained in more detail in [baseline lesson 2](#), this means implementing *delay loops*.

A loop needs a loop counter: a variable which is incremented or decremented on every pass through the loop.

Variables are defined by reserving data memory (or general purpose registers), using the 'UDATA' and 'res' directives. For example:

```
;***** VARIABLE DEFINITIONS
        UDATA
dc1     res 1           ; delay loop counters
dc2     res 1
```

However, if you include these directives in your program for the PIC12F629, you will find that, although the code compiles ok, the build fails in the link phase, with an error like:

```
Error - section '.udata' can not fit the section.
```

What's going on?

[Baseline lesson 3](#) explained that, on many baseline PICs, some registers are *banked*, being mapped into only one of the PIC's banks of data memory, while another set of registers (usually much smaller) are *shared*, or unbanked, being mapped into every bank. This is also true for mid-range PICs.

The 'UDATA' directive declares a section of banked data memory.

If you do not specify a label for a UDATA section, MPASM will name it '.udata'.

Recall that the 12F629 does not have any banked data memory; it is all shared. So this error message is telling us that the linker cannot find space for our UDATA section, because there is no banked memory to put it into.

Therefore, on the 12F629 (or any mid-range PIC without banked GPRs), all variables must be defined using 'UDATA_SHR', which declares a section of shared data memory, instead of 'UDATA'.

For example:

```
;***** VARIABLE DEFINITIONS
        UDATA_SHR
dc1     res 1           ; delay loop counters
dc2     res 1
```

This will declare two single-byte variables, 'dc1' and 'dc2', in shared memory.

And note that, since the variables are held in shared memory, there is no need to use `banksel` before accessing them.

Here's an example of a simple "do nothing" delay loop:

```
        movlw    .N
        movwf    dc1           ; dc1 = 10 = number of loop iterations
dly1    nop
        decfsz   dc1, f
        goto     dly1
```

The first two instructions initialise the loop counter variable ‘dcl’ to the decimal value “N”. Since the mid-range PICs are 8-bit devices, “N” has to be between 0 and 255.

Note that numbers in MPASM are specified as being decimal constants by prefixing them with a ‘.’, or using the syntax d‘*decimal digits*’. If you don’t do this, the assembler will use the default *radix* (hexadecimal), and you may not be using the number you think you are! Although it’s possible to set the default radix to decimal, you’ll run into problems if you rely on a particular default radix being set, and then later copy and paste your code into another project, with a different default radix, giving different results. It’s much safer to simply prefix all decimal numbers with ‘.’.

The ‘decfsz’ instruction performs the work of implementing the loop – “**d**ecre**ment** file register, skip if zero”. First, it decrements the contents of the specified register, and either writes the result back to the file register (if ‘,f’ is specified as the destination) or to W, (if ‘,w’ is specified as the destination). If the result is not yet zero, the next instruction is executed, which will normally be a ‘goto’ which jumps back to the start of the loop. But if the result is zero, the next instruction is skipped, exiting the loop.

Mid-range PICs also have an ‘incfsz’ instruction, equivalent to ‘decfsz’, except that it increments a file register instead of decrementing it. It’s used in loops where you want to count up from an initial value, instead of down.

For a ‘decfsz’ loop, the number of loop iterations is equal to the initial value of the loop counter (“N” in the example above), assuming it is greater than zero.

The ‘nop’ instruction – “**n**o **o**peration” – was included to pad out the example delay loop, to make the delay longer. It does nothing but take some time to execute.

How much time depends on the clock rate. Instructions are executed at one quarter the rate of the processor clock. In this case, the PIC is using the internal RC clock, running at a nominal 4 MHz. The instructions are clocked at ¼ of this rate: 1 MHz. So in this example, each instruction cycle is 1 µs.

Most mid-range PIC instructions, including ‘nop’, execute in a single cycle. The exceptions are those which jump to another location, such as ‘goto’, which take two cycles to execute.

This means that another useful “do nothing” instruction is ‘goto \$+1’. Since ‘\$’ stands for the current address, ‘\$+1’ is the address of the next instruction. Hence, ‘goto \$+1’ jumps to the following instruction – apparently useless behaviour. But like all ‘goto’ instructions, it executes in two cycles. So ‘goto \$+1’ provides a two cycle delay in a single instruction – equivalent to two ‘nop’s, but using less program memory.

The ‘decfsz’ instruction normally executes in a single cycle. But if the result is zero, and the next instruction is skipped, an extra cycle is added, making it a two-cycle instruction.

To calculate the total time taken by the loop, add the execution time of each instruction in the loop:

nop		1
decfsz	dcl,f	1 (except when result is zero)
goto	dly1	2

That’s a total of 4 cycles, except the last time through the loop, when the decfsz takes an extra cycle and the goto is not executed (saving 2 cycles), meaning the last loop iteration is 1 cycle shorter. And there are two instructions before the loop starts, adding 2 cycles.

Therefore the total delay time = $(N \times 4 - 1 + 2)$ cycles = $(N \times 4 + 1)$ µs

If there was no ‘nop’, the delay would be $(N \times 3 + 1)$ µs.

It may seem that, because 255 is the highest 8-bit number, the maximum number of iterations (N) should be 255. But not quite. If the loop counter is initially 0, then the first time through the loop, the ‘decfsz’

instruction will decrement it to 255, which is non-zero, and the loop continues – another 255 times. Therefore the maximum number of iterations is in fact 256, with the loop counter initially 0.

So for the longest possible single loop delay, we can do something like:

```

                clr    dc1                ; loop 256 times
dly1           nop
                decfsz dc1, f
                goto   dly1

```

The two “move” instructions have been replaced with a single ‘`clr`’ instruction, which clears (to 0) the specified register – “clear file register”.

This uses 1 cycle less, so the total time taken is $256 \times 4 = 1024 \mu\text{s} \approx 1 \text{ ms}$.

That’s still well short of the 0.5 s needed, so we need to wrap (or *nest*) this loop inside another, using separate counters for the inner and outer loops, as shown:

```

                movlw  .N                ; loop (outer) N times
                movwf  dc2
                clr    dc1                ; loop (inner) 256 times
dly1           nop                      ; inner loop = 256 x 4 - 1 = 1023 cycles
                decfsz dc1, f
                goto   dly1
                decfsz dc2, f
                goto   dly1

```

The loop counter ‘`dc2`’ is being used to control how many times the inner loop is executed.

Note that there is no need to clear the inner loop counter (`dc1`) on each iteration of the outer loop, because every time the inner loop completes, `dc1 = 0`.

The total time taken for each iteration of the outer loop is 1023 cycles for the inner loop, plus 1 cycle for the ‘`decfsz dc2, f`’ and 2 cycles for the ‘`goto`’ at the end, except for the final iteration, which, as we’ve seen, takes 1 cycle less. The three setup instructions at the start add 3 cycles, so the total delay (assuming $N > 0$) is:

$$\text{delay time} = (N \times (1023 + 3) - 1 + 3) \text{ cycles} = (N \times 1026 + 2) \mu\text{s}.$$

The maximum delay would be with 256 outer loop iterations, giving 262,658 μs . We need a bit less than double that. We could duplicate all the delay code, but it takes fewer lines of code if we only duplicate the inner loop, as shown:

```

                ; delay 500 ms
                movlw  .244              ; outer loop: 244 x (1023 + 1023 + 3) + 2
                movwf  dc2                ; = 499,958 cycles
                clr    dc1                ; inner loop: 256 x 4 - 1
dly1           nop                      ; inner loop 1 = 1023 cycles
                decfsz dc1, f
                goto   dly1
dly2           nop                      ; inner loop 2 = 1023 cycles
                decfsz dc1, f
                goto   dly2
                decfsz dc2, f
                goto   dly1

```

The two inner loops of 1023 cycles each, plus the 3 cycles for the outer loop control instructions (`decfsz` and `goto`) make a total of 2049 μs . Dividing this into the required 500,000 gives 244.02. This is very close to a whole number, so an outer loop count of 244 will give a good result.

The total execution time for this delay code is 499.958 ms – within 0.01% of the desired result!

Since the internal RC oscillator has a precision of only around $\pm 2\%$, there is no point trying to make this delay any more accurate. But in some cases, to generate a given delay, you will need to add or remove ‘nop’ or ‘goto \$+1’ instructions while adjusting the number of loop iterations. With a little experimentation, it is generally possible to get quite close to the delay you need.

For delays longer than about 0.5 s, you’ll need to add more levels of nesting – with enough levels you generate delays which last for years!

Next we need to be able to toggle, or flip the GP1 output from low to high and back again.

As we saw in [baseline lesson 2](#), to flip a single bit, you can exclusive-or it with 1.

For example, to toggle GP1, we could write:

```
movlw    1<<GP1           ; bit mask to flip only GP1
xorw    GPIO,f           ; flip bits in GPIO
```

The ‘xorwf’ instruction exclusive-ors the W register with the specified register – “exclusive-or W with file register”, and writes the result either to the specified file register (GPIO in this case) or to W, depending on whether ‘,f’ or ‘,w’ is given as the instruction destination.

However, as mentioned earlier, there is a danger in using instructions, such as ‘xorwf’, which read from a register, modify the contents and then write the new value back to the register, to operate directly on port registers, because the value read from a port pin will not always be the same as that written to it.

To avoid these potential read-modify-write problems, it is better to use a *shadow register*, which holds a copy of the value the port register is supposed to have, operating on that shadow copy and then copying the updated value to the port register in a single operation.

For example, if we define a variable to use as a shadow register:

```
          UDATA_SHR
sGPIO    res 1           ; shadow copy of GPIO
```

we can use it in a loop to flash the LED, as follows:

```
        clrf    sGPIO           ; start with shadow GPIO zeroed

flash   ; toggle LED
        movf    sGPIO,w         ; get shadow copy of GPIO
        xorlw   1<<GP1         ; flip bit corresponding to GP1
        movwf   sGPIO           ;   in shadow register
        banksel GPIO           ; and write to GPIO
        movwf   GPIO

        ; delay 500 ms (delay code goes here)

        goto   flash           ; repeat forever
```

The ‘movf’ instruction – “move file register to destination” – is used to read a register.

With ‘,w’ as the destination, ‘movf’ copies the contents of the specified register to W.

With ‘,f’ as the destination, ‘movf’ copies the contents of the specified register to itself. That would seem to be pointless; why copy a register back to itself? The answer is that the ‘movf’ instruction affects the Z (zero) status flag, so copying a register to itself is a way to test whether the value in the register is zero.

The ‘xorlw’ instruction exclusive-ors the given literal (constant) value with the W register, placing the result in W – “exclusive-or literal to W”.

Complete program

Putting together all the above pieces, here's the complete program for flashing an LED:

```

;*****
;
; Description: Lesson 1, example 2
;
; Flashes an LED at approx 1 Hz.
; LED continues to flash until power is removed.
;
; Uses inline 500 ms delay routine
;
;*****
;
; Pin assignments:
; GP1 = indicator LED
;
;*****

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, 4 Mhz int clock
__CONFIG      _MCLRE_ON & _CP_OFF & _CPD_OFF & _BODEN_OFF & _WDT_OFF &
_PWRTE_ON & _INTRC_OSC_NOCLKOUT

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

;***** RESET VECTOR *****
RESET      CODE      0x0000      ; processor reset vector
                ; calibrate internal RC oscillator
                call      0x03FF      ; retrieve factory calibration value
                banksel  OSCCAL      ; (stored at 0x3FF as a retlw k)
                movwf   OSCCAL      ; then update OSCCAL

;***** MAIN PROGRAM *****

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

                clrf   sGPIO          ; start with shadow GPIO zeroed

;***** Main loop
main_loop
                ; toggle LED

```

```

        movf    sGPIO,w           ; get shadow copy of GPIO
        xorlw  1<<GP1           ; toggle bit corresponding to GP1
        movwf  sGPIO            ; in shadow register
        banksel GPIO            ; and write to GPIO
        movwf  GPIO

        ; delay 500 ms
        movlw  .244             ; outer loop: 244 x (1023 + 1023 + 3) + 2
        movwf  dc2              ; = 499,958 cycles
        clrf   dc1              ; inner loop: 256 x 4 - 1
dly1    nop                    ; inner loop 1 = 1023 cycles
        decfsz dc1,f
        goto   dly1
dly2    nop                    ; inner loop 2 = 1023 cycles
        decfsz dc1,f
        goto   dly2
        decfsz dc2,f
        goto   dly1

        ; repeat forever
        goto   main_loop

END

```

If you follow the programming procedure described earlier, you should now have an LED flashing at something very close to 1 Hz.

Conclusion

There has been a lot of theory in this lesson, but we now have a solid base to build on.

By flashing an LED, you have shown that you have a working development environment and that you can create projects, modify your code, load (program) your code into your PIC, and make it run.

We've seen how to toggle a pin, and how to use shadow registers can be used to avoid potentially problematic "read-modify-write" operations on a port.

We also saw how to use decrement instructions with conditional tests to implement loops, and how to use loops to create delays of any length.

In the [next lesson](#) we'll see how to make the code more modular, so that useful code such as the 500 ms delay developed here can be easily re-used within a program, or in other programs.