Introduction to PIC Programming

Baseline Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 2: Flash an LED

In lesson 1 we lit a single LED connected to pin GP1 of a PIC12F509 (or 508). Now we'll make it flash.

To do this, we need to learn about:

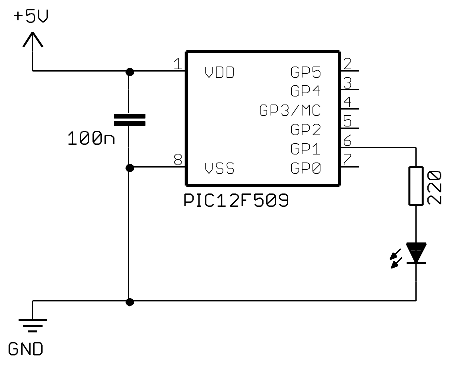
- Using loops to create delays
- Variables
- Using exclusive-or (xor) to flip bits
- The 'read-modify-write' problem

The Circuit

The development environment and circuit for this is identical to that in lesson 1.

Again, it is assumed that you are using Microchip's PICkit 2 programmer, Low Pin Count Demo Board, and MPLAB integrated development environment.

Here's the circuit again from lesson 1:



Refer back to lesson 1 to see how to build it by either soldering a resistor, LED (and optional isolating jumper) to the demo board, or by making connections on the demo board's 14-pin header.

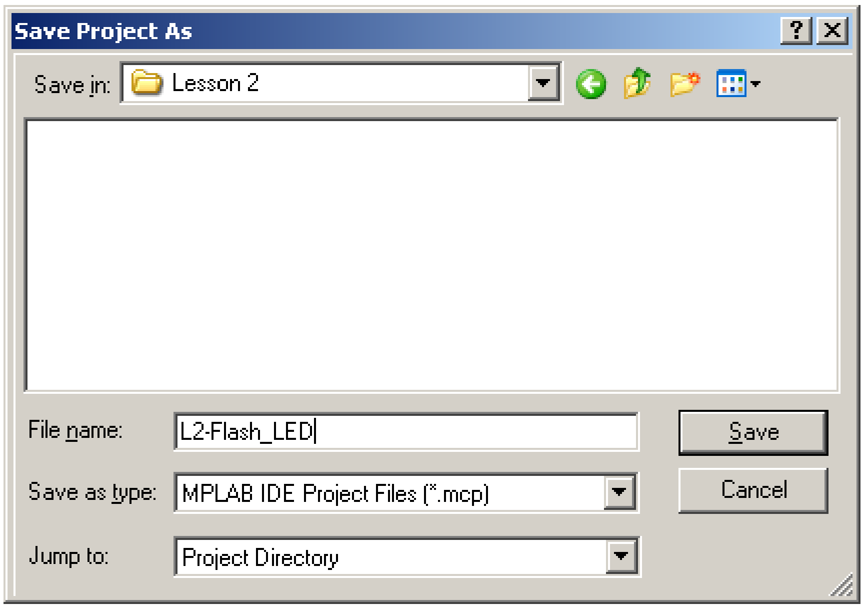
Creating a new project

It is a good idea, where practical, to base a new software project on work you've done before. In this case, it makes sense to build on the program from lesson 1 – we just have to add extra instructions to flash the LED.

There are a couple of ways to create a new project based on an existing one, but the following method works well.

First, open the project from lesson 1 in MPLAB. You can do this easily by double-clicking on the '*.mcp' project file in the project directory you created in lesson 1.

Now close the editor window then use the "Project → Save Project As…" menu item to save the project in a new directory, with a new name. For example:


Save Project As

Save in: Lesson 2

File name: L2-Flash_LED

Save as type: MPLAB IDE Project Files (*.mcp)

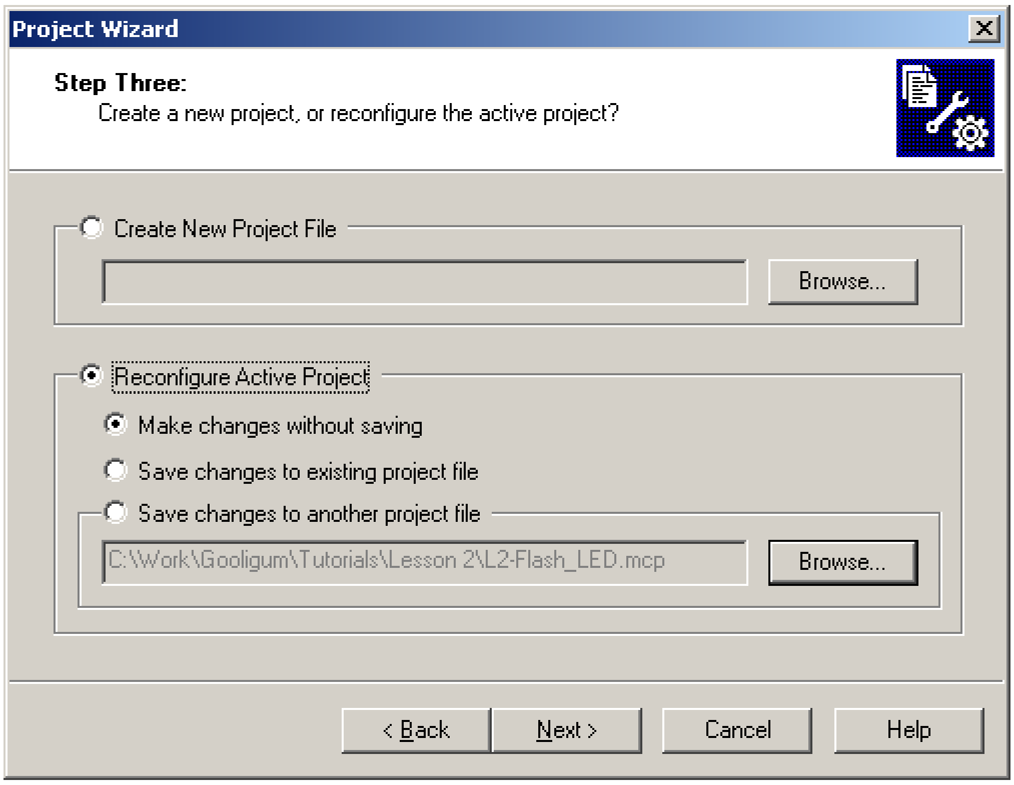
Jump to: Project Directory

When a project is saved to a new location, all the files belonging to that project ("User" files, with relative paths) are copied to that location. You will find that in this case the '*.asm' source file from lesson 1 has been copied into the new directory.

The next step is to run the project wizard to reconfigure the project, to give the source file a new name.

Use the "Project → Project Wizard…" menu item to run the project wizard.

The correct device (PIC12F509) will already be selected, as will the toolsuite (MPASM), so simply click "Next" until you get to Step Three, and select "Reconfigure Active Project" and "Make changes without saving", as shown:


Project Wizard

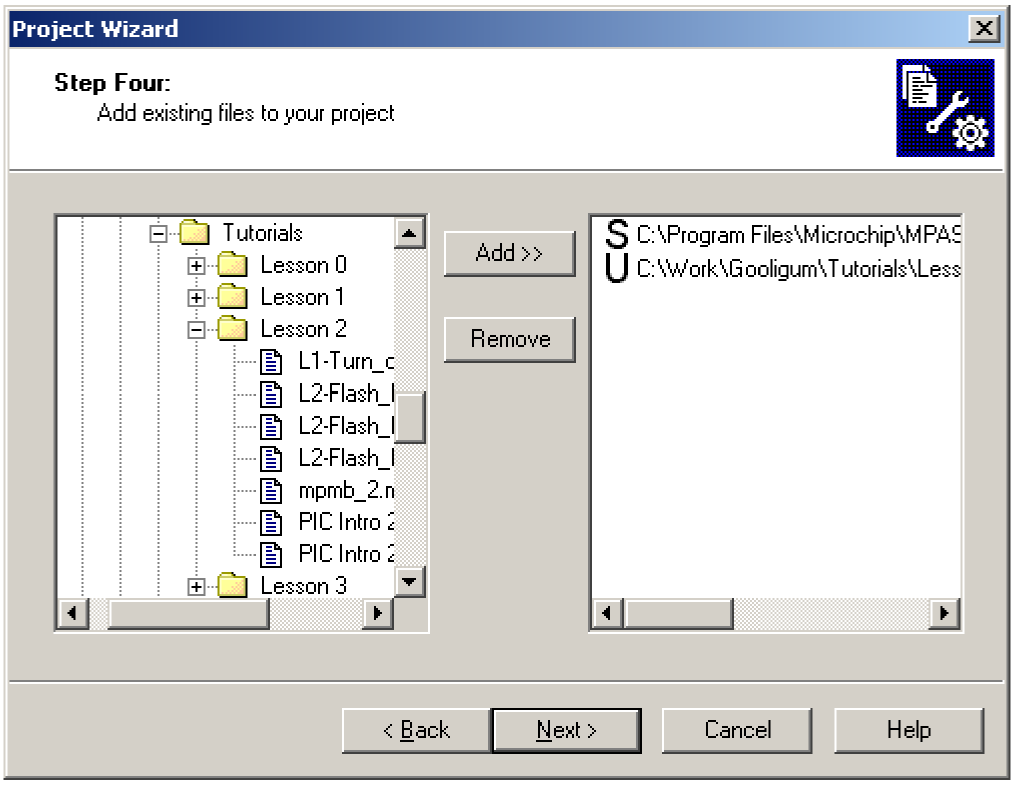
Step Three:
Create a new project, or reconfigure the active project?

Create New Project File [✓ Reconfigure Active Project]

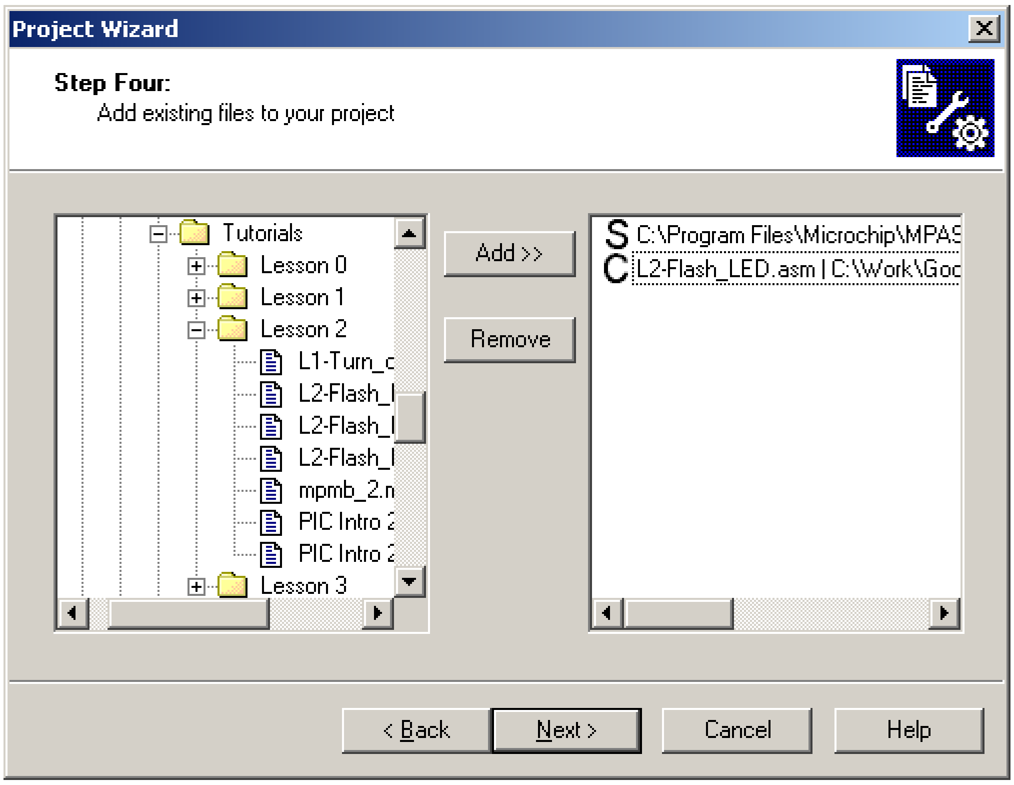
[✓ Make changes without saving] Save changes to existing project file Save changes to another project file

C:\Work\Gooligum\Tutorials\Lesson 2\L2-Flash_LED.mcp

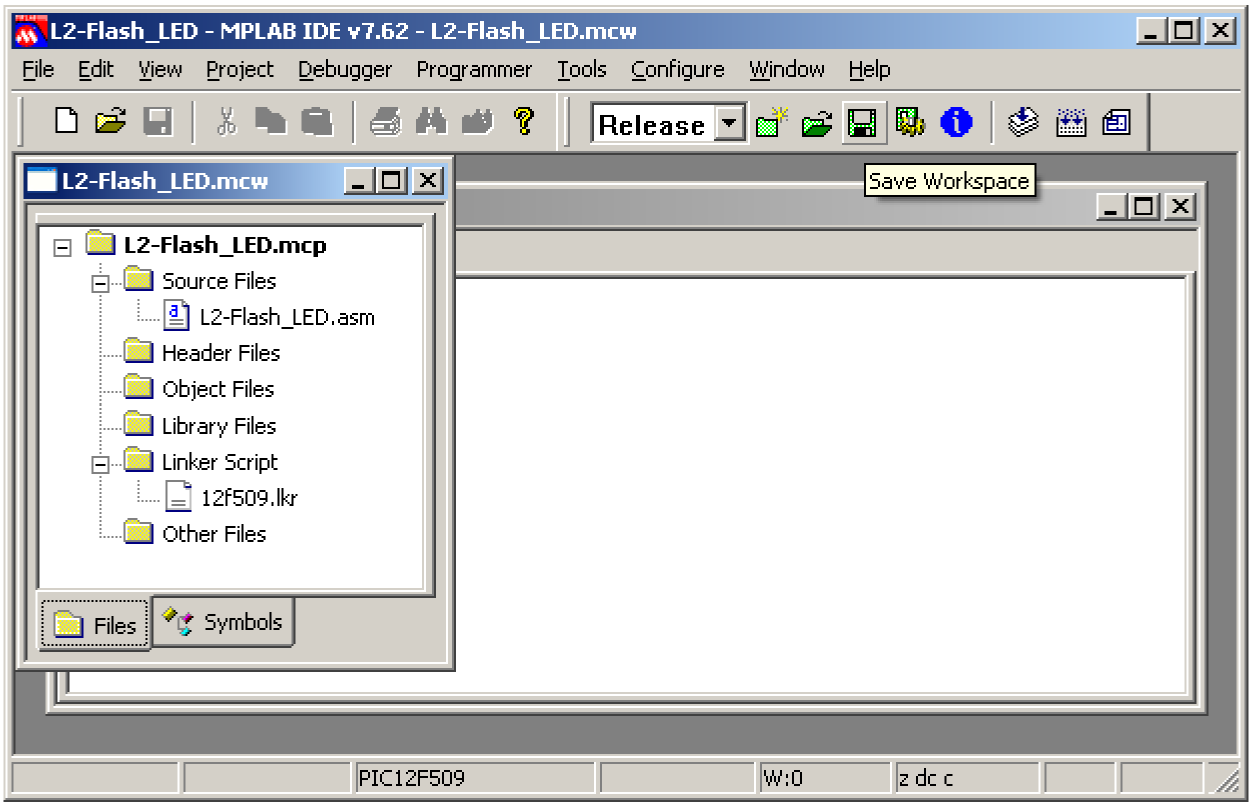
You are now presented with the following window, showing the linker file with an “S” to indicate a system file (and therefore still in it’s original location, under the MPASM install directory) and the assembler source file with a “U” to indicate a user file, in the new project directory, but with the same name as before:



Click on the “U” until it changes to a “C”. You can now click on the file name and rename it to something like ‘L2-Flash_LED.asm’:

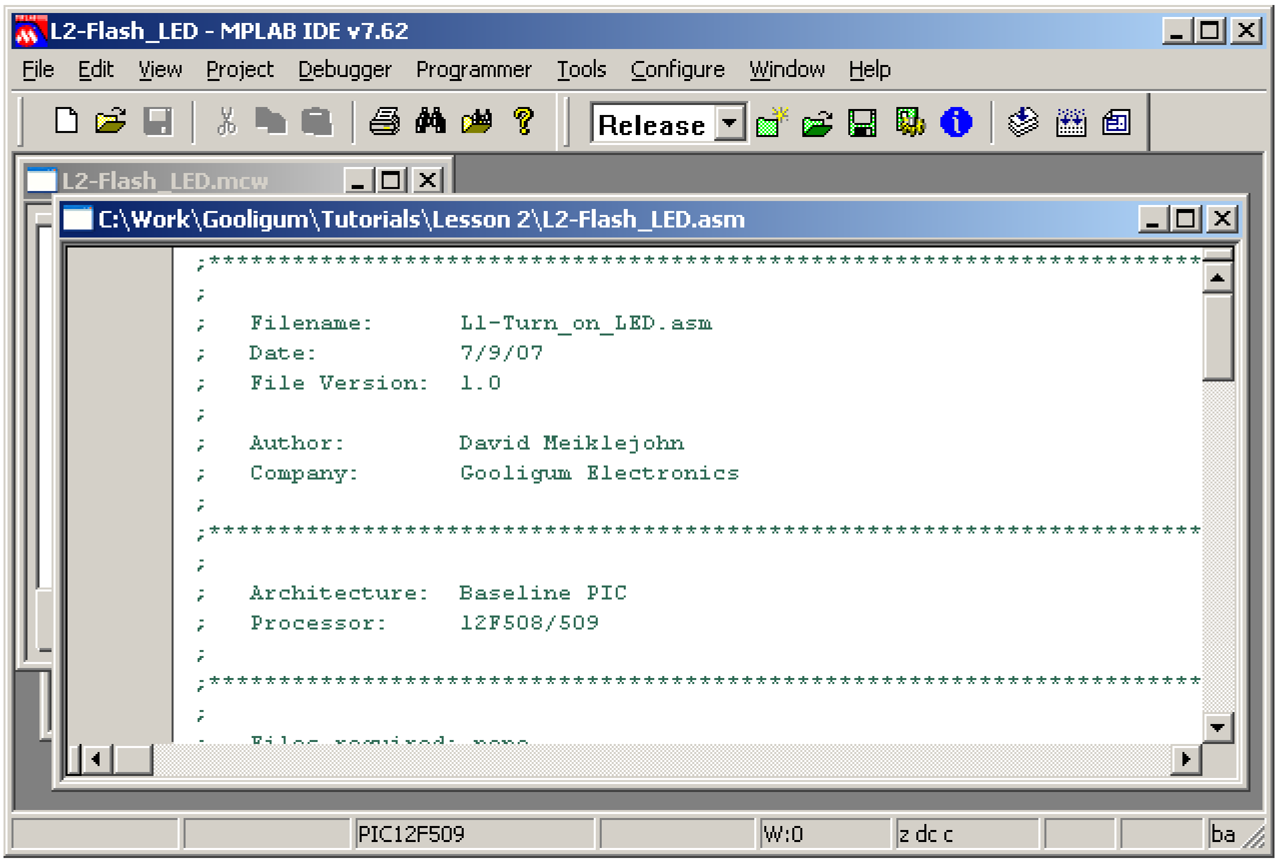


Finally click “Next” then “Finish” and the project is reconfigured, with the renamed source file:



It’s a good idea at this point to save your new project, using the “Save Workspace” icon, or the “Project → Save Project” menu item.

If you double-click on the source file (‘L2-Flash_LED.asm’), you’ll see a copy of your code from lesson 1:


L2-Flash_LED - MPLAB IDE v7.62
File Edit View Project Debugger Programmer Tools Configure Window Help
Release
L2-Flash_LED.mcw
C:\Work\Gooligum\Tutorials\Lesson 2\L2-Flash_LED.asm
PIC12F509
W:0
z dc c
ba

```
;************************************************************************
;
;   Filename:       L1-Turn_on_LED.asm
;   Date:           7/9/07
;   File Version:   1.0
;
;   Author:         David Meiklejohn
;   Company:        Gooligum Electronics
;
;************************************************************************
;
;   Architecture:   Baseline PIC
;   Processor:      12F508/509
;
;************************************************************************
;
;   Files required: none
```

Flashing the LED

First, of course, the comments should be updated to reflect the new project. For example:

```
;************************************************************************
;                                                                       *
;   Filename:      BA_L2-Flash_LED.asm                                  *
;   Date:          8/9/07                                               *
;   File Version:  1.0                                                  *
;                                                                       *
;   Author:        David Meiklejohn                                     *
;   Company:       Gooligum Electronics                                 *
;                                                                       *
;************************************************************************
;                                                                       *
;   Architecture:  Baseline PIC                                         *
;   Processor:     12F508/509                                           *
;                                                                       *
;************************************************************************
;                                                                       *
;   Files required: none                                                *
;                                                                       *
;************************************************************************
;                                                                       *
;   Description:    Lesson 2, example 1                                 *
;                                                                       *
;   Flashes an LED at approx 1Hz.                                       *
;   LED continues to flash until power is removed.                      *
;                                                                       *
;************************************************************************
;                                                                       *
;   Pin assignments:                                                    *
;       GP1 - flashing LED                                              *
;                                                                       *
;************************************************************************
```

As we did in lesson 1, we need to set up the PIC so that only GP1 is configured as an output, so we can leave this code intact:

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

                ; ext reset, no code protect, no watchdog, 4Mhz int clock
    __CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _IntRC_OSC


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


;***** MAIN PROGRAM

;***** Initialisation
start
        movlw   b'111101'       ; configure GP1 (only) as an output
        tris    GPIO
```

In lesson 1, we made GP1 high, and left it that way. To make it flash, we need to set it high, then low, and then repeat. You may think that you could achieve this with something like:

```
flash
        movlw   b'000010'       ; set GP1 high
        movwf   GPIO
        movlw   b'000000'       ; set GP1 low
        movwf   GPIO
        goto    flash           ; repeat forever
```

If you try this code, you'll find that the LED appears to remain on continuously. In fact, it's flashing too fast for the eye to see. With the 12F509 configured with the internal RC oscillator, it is clocked at close to 4 MHz, and each instruction executes in 1 µs (except instructions which branch to another location, such as 'goto', which execute in 2 µs). This loop takes a total of 6 µs, so the LED flashes at 1/(6 µs) = 166.7 kHz. That's much to fast to see!

To slow it down to a more sedate (and visible!) 1 Hz, we have to add a delay. But before looking at delays, we can make a small improvement to the code.

To flip a single bit – to change it from 0 to 1 or from 1 to 0, you can exclusive-or it with 1. That is:

0 XOR 1 = 1

1 XOR 1 = 0

So to flip GP1, we can read the current state of GPIO, exclusive-or the bit corresponding to GP1, then write it back to GPIO, as follows:

```
        movlw   b'000010'       ; bit mask to flip GP1 only
flash
        xorwf   GPIO,f          ; flip GP1 using mask in W
        goto    flash           ; repeat forever
```

The '`xorwf`' instruction exclusive-ors the W register with the specified register – "e**x**clusive-**or W** with **f**ile register", and writes the result either to the specified file register (GPIO in this case) or to W.

Note that there is no need to set GP1 to an initial state; whether it's high or low to start with, it will be successively flipped.

Many of the PIC instructions, like `xorwf`, are able to place the result of an operation (e.g. add, subtract, or in this case XOR) into either a file register or W. This is referred to as the instruction destination. A '`,f`' at the end indicates that the result should be written back to the file register; to place the result in W, use '`,w`' instead.

This single instruction – '`xorwf GPIO,f`' – is doing a lot of work. It reads GPIO, performs the XOR operation, and then writes the result back to GPIO.

The read-modify-write problem

And therein lays a potential problem. You'll find it referred to as the *read-modify-write* problem. When an instruction reads a port register, such as GPIO, the value that is read back is not necessarily the value that you wrote to the register. When the PIC reads a port register, it doesn't read the value in the "output latch" (i.e. the value you wrote to it). Instead, it reads the pins themselves – the voltages present in the circuit.

Normally, that doesn't matter. When you write a '1', the corresponding pin (if configured as an output) will go to a high voltage level, and when you then read that pin, it's still at a high voltage, so it reads back as a '1'. But if there's excessive load on that pin, the PIC may not be able to drive it high, and it will read as a '0'. Or capacitance loading the output line may mean a delay between the PIC's attempt to raise the voltage and the voltage actually swinging high enough to register as a '1'. Or noise in the circuit may mean that a line that normally reads as a '1', sometimes (randomly) reads as a '0'.

In this simple case, particularly when we slow the flashing down to 1 Hz, you'll find that this isn't an issue. The above code will usually work correctly. But it's good to get into good habits early. For the reasons given above, it is considered "bad practice" to assume a value you have previously written is still present on an I/O port register. So what's "good practice"? Keep a copy of what the port value is supposed to be, and operate on that, then copy it to the port register. This is referred to as using a *shadow register*.

We could use W as a shadow register, as follows:

```
        movlw   b'000000'       ; start with W zeroed
flash
        xorlw   b'000010'       ; flip W bit corresponding to GP1 (bit 1)
        movwf   GPIO            ; and write to GPIO
        goto    flash           ; repeat forever
```

Each time around the loop, the contents of W are updated and then written to the I/O port.

The '`xorlw`' instruction exclusive-ors a literal value with the W register, placing the result in W – "e**x**clusive-**or l**iteral to **W**".

Normally, instead of '`movlw   b'000000'`' (or simply '`movlw   0`') you'd use the '`clrw`' instruction – "**clear W**".

'`clrw`' has the same effect as '`movlw   0`', except that '`clrw`' sets the 'Z' (zero) status flag, while the '`movlw`' instruction doesn't affect any of the status flags, including Z.

Status flags are bits in the STATUS register:

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

Certain arithmetic or logical operations will set or clear the Z, DC or C status bits, and other instructions can test these bits, and take different actions depending on their value. We'll see examples of testing these flags in later lessons.

We're not using Z here, so we can use `clrw` to make the code more readable:

```
        clrw                    ; use W to shadow GPIO - initially zeroed
flash
        xorlw   b'000010'       ; flip W bit corresponding to GP1 (bit 1)
        movwf   GPIO            ; and write to GPIO
        goto    flash           ; repeat forever
```

It would be very unusual to be able to use W as a shadow register, because it is used in so many PIC instructions. When we add delay code, it will certainly need to be able to change the contents of W, so we'll have to use a file register to hold the shadow copy of GPIO.

In lesson 1, we saw how to allocate data memory for variables (or any data, including shadow copies), using the `UDATA` and `RES` directives. In this case, we need something like:

```
;***** VARIABLE DEFINITIONS
        UDATA
sGPIO   res 1                    ; shadow copy of GPIO
```

The flashing code now becomes:

```
        clrf    sGPIO           ; clear shadow register
flash
        movf    sGPIO,w         ; get shadow copy of GPIO
        xorlw   b'000010'       ; flip bit corresponding to GP1 (bit 1)
        movwf   GPIO            ; write to GPIO
        movwf   sGPIO           ; and update shadow copy
        goto    flash           ; repeat forever
```

That's nearly twice as much code as the first version, that operated on GPIO directly, but this version should be much more robust.

There are two new instructions here.

'`clrf`' clears (sets to 0) the specified register – "**cl**ear **f**ile register".

'`movf`', with '`,w`' as the destination, copies the contents of the specified register to W – "**mov**e **f**ile register to destination". This is the instruction used to read a register.

'`movf`', with '`,f`' as the destination, 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 status flag, so copying a register to itself is a way to test whether the value in the register is zero.

Delay Loops

To make the flashing visible, we need to slow it down, and that means getting the PIC to "do nothing" between LED changes.

The baseline PICs do have a "do nothing" instruction: '`nop`' – "**no op**eration". All it does is to 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 (see lesson 1). The instructions are clocked at ¼ of this rate: 1 MHz. Each instruction cycle is then 1 µs.

Most baseline PIC instructions, including '`nop`', execute in a single cycle. The exceptions are those which jump to another location (such as '`goto`') or if an instruction is conditionally skipped (we'll see an example of this soon). So '`nop`' provides a 1µs delay – not very long!

Another "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 as for all '`goto`' instructions, it executes in two cycles. So '`goto $+1`' provides a 2 µs delay in a single instruction – equivalent to two '`nop`'s, but using less program memory.

To flash at 1 Hz, the PIC should light the LED, wait for 0.5 s, turn off the LED, wait for another 0.5 s, and then repeat.

Our code changes the state of the LED once each time around the loop, so we need to add a single delay of 0.5 s within the loop. That's 500,000 µs, or 500,000 instruction cycles. Clearly we can't do that with '`nop`'s or '`goto`'s alone!

The answer, of course, is to use loops to execute instructions enough times to build up a useful delay. But we can't just use a '`goto`', or else it would loop forever and the delay would never finish. So we have to loop some finite number of times, and for that we need to be able to count the number of times through the loop (incrementing or decrementing a loop counter variable) and test when the loop is complete.

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

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

The first two instructions write the decimal value "10" to a loop counter variable called '`dc1`'.

Note that to specify a decimal value in MPASM, you prefix it with a '.'. If you don't include the '.', the assembler will use the default *radix* (hexadecimal), and you won't 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, and then later copy and paste your code into another project, with a different default radix, giving different results. It's much safer, and clearer, to simply prefix all hexadecimal numbers with '0x' and all decimal numbers with '.'.

The '`decfsz`' instruction performs the work of implementing the loop – "**dec**rement **f**ile register, **s**kip if **z**ero". First, it decrements the contents of the specified register, writes the result back to the register (as specified by the '`,f`' destination), then tests whether the result was zero. If it's 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 of the decrement is zero, the next instruction is skipped; since this is typically a '`goto`', skipping it means exiting the loop.

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.

There is also an '`incfsz`' instruction, which is equivalent to '`decfsz`', except that it increments instead of decrementing. It's used if you want to count up instead of down. For a loop with a fixed number of iterations, counting down is more intuitive than counting up, so '`decfsz`' is more commonly used for this.

In the code above, the loop counter, '`dc1`', starts at 10. At the end of the first loop, it is decremented to 9, which is non-zero, so the '`goto`' instruction is not skipped, and the loop repeats from the '`dly1`' label. This process continues – 8,7,6,5,4,3,2 and on the 10th iteration through the loop, `dc1` = 1. This time, `dc1` is decremented to zero, and the "skip if zero" comes into play. The '`goto`' is skipped, and execution continues after the loop.

You can see that the number of loop iterations is equal to the initial value of the loop counter (10 in this example). Call that initial number N. The loop executes N times.

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

`nop`		1
`decfsz`	`dc1,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 × 4 – 1 + 2) cycles = (N × 4 + 1) µs

If there was no '`nop`', the delay would be (N × 3 + 1) µs; if two '`nop`'s, then it would be (N × 5 +1) µs, etc.

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, and if an 8-bit counter is decremented from 0, the result is 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 write something like:

```
        clrf    dc1             ; loop 256 times
dly1    nop
        decfsz  dc1,f
        goto    dly1
```

The two "move" instructions have been replaced with a single '`clrf`', using 1 cycle less, so the total time taken is 256 × 4 = 1024 µs ≈ 1 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   .200            ; loop (outer) 200 times
        movwf   dc2
        clrf    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 if the number of outer loop iterations is N:

Total delay time = (N × (1023 + 3) – 1 + 3) cycles = (N × 1026 + 2) µs.

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

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 needed 500,000 gives 244.02 – pretty close to a whole number, so an outer loop count of 244 will be very close to what's needed.

The calculations are shown in the comments above. The total time for this delay code is 499,958 cycles. In theory, that's 499.958 ms – within 0.01% of the desired result! Given that that's much more accurate than the 4 MHz internal RC oscillator, there is no point trying for more accuracy than this.

But suppose the calculation above had come out as needing some fractional number of outer loop iterations, say 243.5 – what would you do? Generally you'd fine-tune the timing by adding or removing '`nop`'s. E.g. suppose that both inner loops had 2 '`nop`'s instead of 1. Then they would execute in 256 × 5 – 1 = 1279 cycles, and the calculation for the outer loop counter would be 500,000 ÷ (1279 + 1279 + 3) = 195.24. That's not as good a result as the one above, because ideally we want a whole number of loops. 244.02 is much closer to being a whole number than 195.24.

For even finer control, you can add '`nop`'s to the outer loop, immediately before the '`decfsz  dc2,f`' instruction. One extra '`nop`' would give the outer loop a total of 1023 + 1023 + 4 = 2050 cycles, instead of 2049. The loop counter calculation becomes 500,000 ÷ 2050 = 243.90. That's not bad, but 244.02 is better, so we'll leave the code above unchanged.

With a bit of fiddling, once you get some nested loops close to the delay you need, adding or removing '`nop`' or '`goto  $+1`' instructions can generally get you quite close to the delay you need. And remember that it is pointless to aim for high precision (< 1%) when using the internal RC oscillator. When using a crystal, it makes more sense to count every last cycle accurately, as we'll see in lesson 7.

For delays longer than about 0.5 s, you'll need to add more levels of nesting to your delay loops – with enough levels you can count for years!

Complete program

Putting together all the above, here's the complete assembler source needed for flashing an LED:

```
;************************************************************************
;                                                                       *
;   Description:    Lesson 2, example 1                                 *
;                                                                       *
;   Flashes an LED at approx 1Hz.                                       *
;   LED continues to flash until power is removed.                      *
;                                                                       *
;************************************************************************
;                                                                       *
;   Pin assignments:                                                    *
;       GP1 - flashing LED                                              *
;                                                                       *
;************************************************************************

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

                ; ext reset, no code protect, no watchdog, 4Mhz int clock
    __CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _IntRC_OSC


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


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


;***** MAIN PROGRAM
```

```
;***** Initialisation
start
        movlw   b'111101'       ; configure GP1 (only) as an output
        tris    GPIO
        clrf    sGPIO           ; start with shadow GPIO zeroed

;***** Main loop
flash
        movf    sGPIO,w         ; get shadow copy of GPIO
        xorlw   b'000010'       ; flip bit corresponding to GP1 (bit 1)
        movwf   GPIO            ; write to GPIO
        movwf   sGPIO           ; and update shadow copy

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

        goto    flash           ; repeat forever


        END
```

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

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.