Introduction to PIC Programming

Baseline Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 1: Light an LED

This initial exercise is the "Hello World!" of PIC programming.

The apparently straightforward task of simply making an LED connected to one of the output pins of a PIC light up – never mind flashing or anything else – relies on:

- Having a functioning circuit in a workable prototyping environment
- Being able to use a development environment; to go from text to assembled PIC code
- Being able to correctly use a PIC programmer to load the code into the PIC chip
- Correctly setting the PIC's operating mode
- Writing code that will set the correct pin to output a high or low (depending on the circuit)

If you can get an LED to light up, then you know that you have a development, programming and prototyping environment that works, and enough understanding of the PIC architecture and instructions to get started. It's a firm base to build on.

Sound complicated? It's not…

Getting Started

For some background on PICs in general and details of the recommended development environment, see lesson 0. Briefly, these tutorials assume that you are using Microchip's PICkit 2 programmer and Low Pin Count (LPC) Demo Board, with Microchip's MPLAB integrated development environment. But it is of course possible to adapt these instructions to a different programmer and/or development board.

As mentioned in lesson 0, we're going to start with one of the simplest of PICs – the PIC12F509, an 8-pin "baseline" device. This PIC is supported by the LPC demo board, and is only capable of simple digital I/O. It does not support analog input, 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 PICs later.

In summary, for this lesson you should have:

- A PC running Windows XP or Vista (32-bit only), with a spare USB port
- Microchip's MPLAB IDE software
- Microchip's PICkit 2 PIC programmer and Low Pin Count Demo Board
- A PIC12F509-I/P microcontroller
- Optionally, a standard 5 mm LED (preferably not blue or white), a 220 Ω ¼W resistor, some hook-up wire, solder and a soldering iron
- Or, if you prefer not to add components to your demo board, a little solid core hook-up wire.

Introducing the PIC12F509 (and PIC12F508)

When working with any microcontroller, you should always have on hand the latest version of the manufacturer's data sheet. You should download the download the current data sheet for the 12F509 from www.microchip.com.

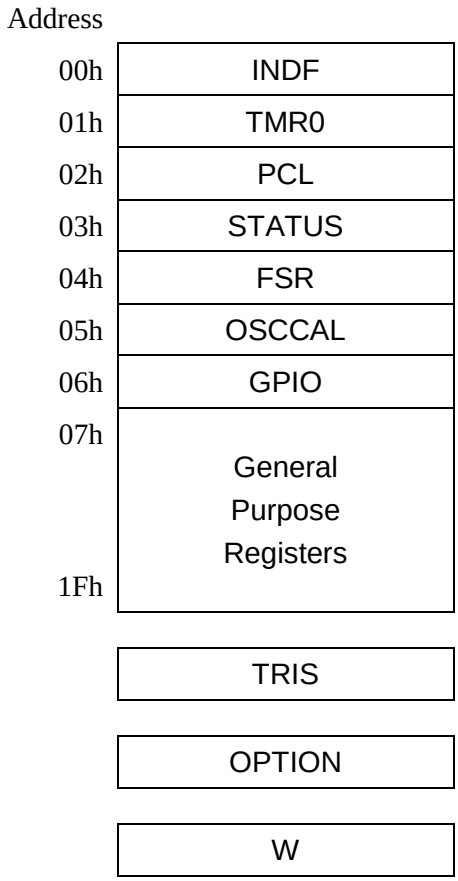
You'll find that the data sheet for the 12F509 also covers the 12F508 and 16F505. These are essentially variants of the same chip. The differences are as follows:

Device	Program Memory (words)	Data Memory (bytes)	Package	I/O pins	Clock rate (maximum)
12F508	512	25	8-pin	6	4 MHz
12F509	1024	41	8-pin	6	4 MHz
16F505	1024	72	14-pin	12	20 MHz

The 12F509 has more memory than the 12F508, but is otherwise identical. The 16F505 adds extra I/O pins, some more data memory, and can run at a higher speed (if driven by an external clock or crystal).

To keep this introduction as simple as possible, we will treat the 12F509 as if it were a 12F508 and ignore the extra memory (which needs special techniques to access that will be explored in a later lesson). So why not simply start with a real 12F508? You could, and everything in the rest of this tutorial would be the same, but then you'd be left with a more limited chip than if you'd bought the (only slightly more expensive) 12F509.

PIC12F508 Registers



As mentioned in lesson 0, PIC microcontrollers use a so-called Harvard architecture, where program and data memory is entirely separate.

In the 12F508, program memory extends from 000h to 1FFh (hexadecimal). Each of these 512 addresses can hold a separate 12-bit program instruction. User code starts by executing the instruction at 000h, and then proceeds sequentially from there – unless of course your program includes loops, branches or subroutines, which any real program will!

Microchip refers to the data memory as a "register file". If you're used to bigger microprocessors, you'll be familiar with the idea of a set of registers held on chip, used for intermediate values, counters or indexes, with data being accessed directly from off-chip memory. If so, you have some unlearning to do! The baseline PICs are quite different from mainstream microprocessors. The only memory available is the on-chip "register file", consisting of a number of registers, each 8 bits wide. Some of these are used as general-purpose registers for data storage, while others, referred to as special-function registers, are used to control or access chip features, such as the I/O pins.

The register map for the 12F508 is shown at left. The first seven registers are special-function, each with a specific address. They are followed by twenty-five general purpose resisters, which you can use to store program variables such as counters.

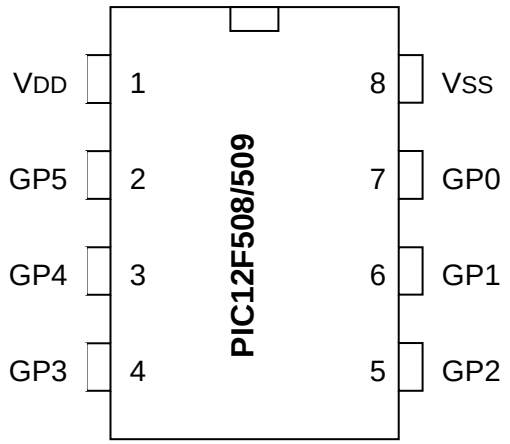
The next two registers, TRIS and OPTION, are special-function registers which cannot be addressed in the usual way; they are accessed through special instructions.

The final register, W, is the working register. It's the equivalent of the 'accumulator' in some other microprocessors. It's central to the PIC's operation. For example, to copy data from one general purpose register to another, you have to copy it into W first, then copy from W to the destination. Or, to add two numbers, one of them has to be in W. W is used a lot!

It's traditional at this point to discuss what each register does. But that would be repeating the data sheet, which describes every bit of every register in detail. The intention of these tutorials is to only explain what's needed to perform a given function, and build on that. We'll start with the I/O pins.

PIC12F508/509 Input/Output

The 12F508 and 12F509 provide six I/O pins in an eight-pin package; see the pin diagram:



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, considered to be at 0V (zero volts), 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.

This wide range means that the PIC's power supply can be very simple. Depending on the circuit, you may need no more than a pair of 1.5 V batteries (3 V total; less as they discharge).

Normally you'd place a capacitor, typically 100 nF, between VDD and VSS, close to the chip, to smooth transient changes to the power supply voltage caused by changing loads (e.g. motors, or something as simple as an LED turning on) or noise in the circuit. I always place these "bypass capacitors" in any circuit beyond the simplest prototype stage, although you'll find that, particularly in a small battery-powered circuit, the PIC will often operate correctly without them. 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.

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. 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':

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

The TRIS 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
TRIS			GP5	GP4		GP2	GP1	GP0

To configure a pin as an input, set the corresponding bit in the TRIS register to '1'. To make it an output, clear the corresponding TRIS bit to '0'.

Why is it called 'TRIS'? Each pin (except GP3) can be configured as one of three states: high-impedance input, output high, or output low. In the input state, the PIC's output drivers are effectively disconnected from the pin. Another name for an output than can be disconnected is '*tri-state*' – hence, TRIS.

Note that bit 3 of TRIS is greyed-out. Clearing this bit will have no effect, as GP3 is always an input.

The default state for each pin is 'input'; TRIS is set to all '1's when the PIC is powered on or reset.

When configured as an output, each I/O pin on the 12F508/509 can source or sink (i.e. current in either into or out of the pin) up to 25 mA – enough to directly drive an LED.

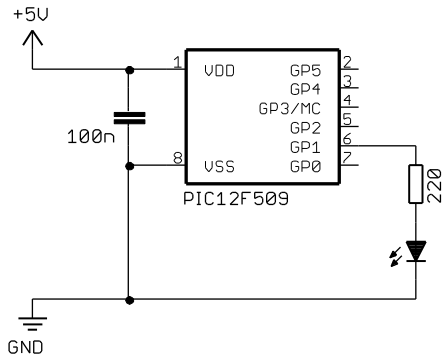
In total, the I/O port can source or sink up to 75 mA.

The Circuit

We now have enough background information to design the circuit.

We can choose any I/O pin other than GP3 (which is input-only) to drive the LED. Since, on the LPC Demo Board, GP0 is connected to a potentiometer, we shouldn't use GP0. So let's use GP1.

The complete circuit looks like this:



That's all there is too it! Modern microcontrollers really do have minimal requirements; as discussed above, in practice, for a quick and dirty prototype, you could even do without the capacitor.

The power supply should be at least 3 V to properly light the LED.

The resistor in series with the LED is necessary to limit the current, to protect both the LED and the PIC. To calculate the appropriate resistance, you first decide on a maximum LED current. Most common 5 mm LEDs can handle the PIC's maximum 25 mA, but most will light very strongly with 15 mA or less. And at 15 mA, even if we had an LED on every output pin, the total current drain would be 75 mA (5 LEDs × 15 mA / LED), the maximum rated current for the port. So we'll design for a maximum current of 15 mA.

Next, consider voltage. The data sheet does not characterise the 'high' output voltage for output currents > 5 mA. But assuming a 5 V power supply, the maximum possible output voltage will also be 5 V. In practice, it will be less – probably below 4 V at 15 mA – but it is good practice to design to the maximum value. A red or green LED will typically drop 2.1 V across it when lit. So the resistor will drop 5.0 V – 2.1 V = 2.9 V. According to Ohm's law, R = V / I = 2.9 V ÷ 15 mA = 193 Ω. It's best to err on the safe side, so use the next highest standard value: 220 Ω.

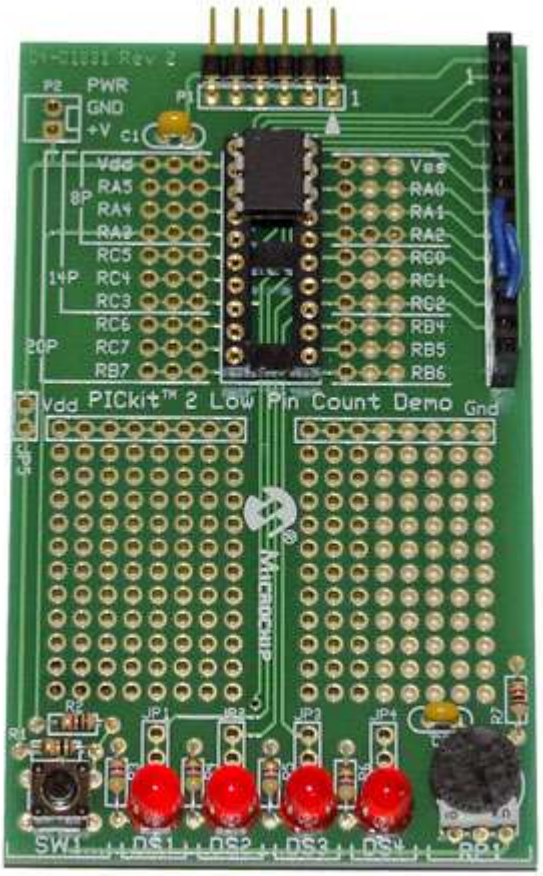
Although the LPC Demo Board provides four LEDs, they cannot be used directly with a 12F509 (or any 8-pin PIC), because they are connected to DIP socket pins which are only used with 14-pin and 20-pin devices.

However, the circuit can be readily built by adding an LED, a 220 Ω resistor and a piece of wire to the LPC Demo Board, as illustrated on the right. The board already includes a 100 nF

capacitor (C1) across the power pins. Power (5 V) is supplied from the PICkit 2.

In the pictured board, a green LED is wired to GP1 (labelled 'RA1') and a red LED to GP2 (labelled 'RA2'); we'll use both LEDs in later lessons. Jumper blocks have been added so that these LEDs can be easily disconnected from the PIC, to facilitate prototyping other circuits. These jumpers are wired in series with each LED.

Note that on the LPC Demo Board, the pins are labelled 'RA1', 'RA2', etc., because that is the nomenclature used on the larger 20-pin PICs, such as the 16F690. They correspond to the 'GP' pins on the 12F508/509 – simply another name for the same thing.



If you prefer not to solder components onto your demo board, you can use the LEDs on the board, labelled 'DS1' to 'DS4', by making connections on the 14-pin header on the right of the demo board, as illustrated on the left.

This header makes available all the 12F508/509's pins, GP0 – GP5 (labelled 'RA0' to 'RA5'), as well as power (+5 V) and ground. It also brings out the additional pins, labelled 'RC0' to 'RC5', available on the 14-pin devices.

The LEDs are connected to the pins labelled 'RC0' to 'RC3' on the IC socket, via 470 Ω resistors (and jumpers, if you choose to install them). 'DS1' connects to pin 'RC0', 'DS2' to 'RC1', and so on.

So, to connect LED 'DS2' to pin GP1, simply connect the pin labelled 'RA1' to the pin labelled 'RC1', which can be done by plugging a short piece of solid-core hook-up wire into pins 8 and 11 on the 14-pin header.

Similarly, to connect LED 'DS3' to pin GP2, simply connect header pins 9 and 12.

That's certainly much easier than soldering, so why bother adding LEDs to the demo board? The only real advantage is that, when using 14-pin and 20-pin PICs later, you may find it useful to have LEDs available on RA1 and RA2, while leaving RC0 – RC3 available to use, independently. In any case, it is useful to leave the 14-pin header free for use as an expansion connector: see, for example, lesson 8.

Time to move on to programming!

Programming Environment

As discussed in lesson 0, you'll need Microchip's MPLAB Integrated Development Environment (MPLAB IDE), which you can download from www.microchip.com.

When installing, you should select as a minimum:

- ✓ Microchip Device Support
  - ✓ 8 bit MCUs (all 8-bit PICs, including 10F, 12F, 16F and 18F series)
- ✓ Third Party Applications
  - ✓ CCS PCB Full Install (C compiler for baseline PICs)
  - ✓ HI-TECH C (C compiler for baseline and midrange PICs)
- ✓ Microchip Applications
  - ✓ MPASM Suite (the assembler)
  - ✓ MPLAB IDE (the development environment)
    - ✓ MPLAB SIM (software simulator – extremely useful!)
    - ✓ PICkit 2

It's worth selecting the two free C compilers, since they will be properly integrated with MPLAB this way; we'll see how to use them in future lessons. But for now, we'll start with assembly, as that is the best way to understand the PIC architecture and features – from the ground up.

When programming in PIC assembler, you have to choose whether to create absolute or relocatable code. Originally, only absolute mode was supported, so most of the older PIC programming resources will only refer to it. In absolute mode, you specify fixed addresses for your code in program memory, and fixed addresses for your variables in data memory. That's ok for small programs, and seems simple to start with (which is another reason why many guides for beginners only mention absolute mode). But as you start to build larger applications, perhaps making use of reusable modules of previously-written code, and you start to move code from one PIC chip to another, you'll find that absolute mode is very limiting.

Relocatable code relies on a *linker* to assign object code (the assembled program instructions) and variables to appropriate addresses, under the control of a script specific to the PIC you're building the program for. Microchip supplies linker scripts for every PIC; unless you're doing something advanced, you don't need to touch them – so we won't look at them. Writing relocatable code isn't difficult, and it will grow with you, so that's what we'll use.

Creating a New Project

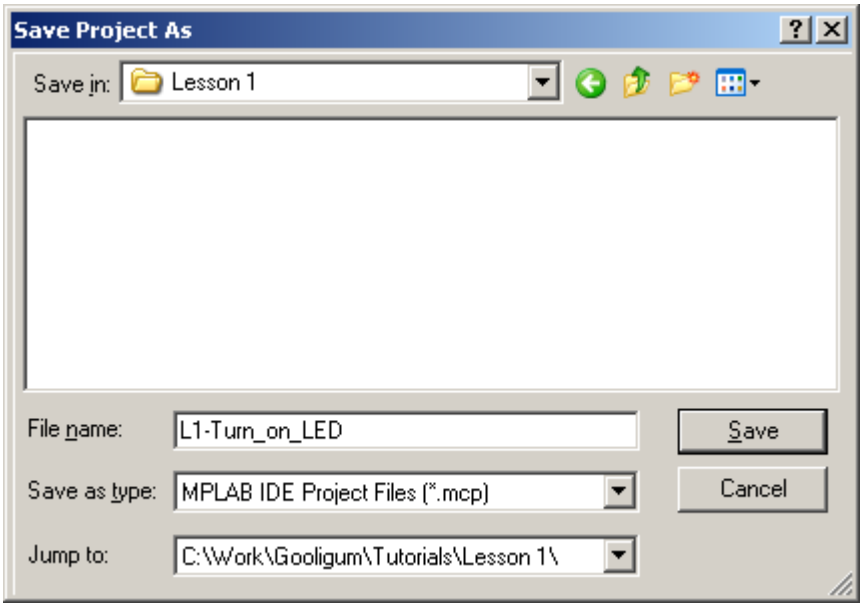
To start a new project, you should run the project wizard (Project → Project Wizard…).

First, you'll be asked to select a device. Choose the PIC you're using: in our case, the 12F509.

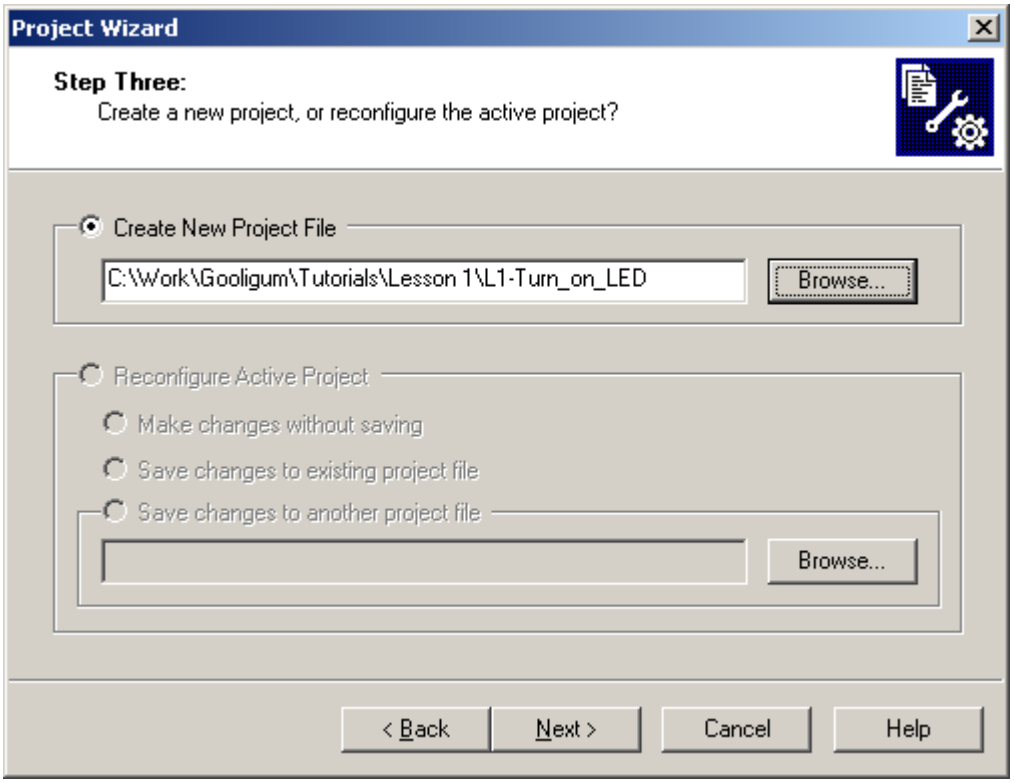
Then ensure that MPASM is selected as the active toolsuite. This tells MPLAB that this is an assembler project. Don't worry about the toolsuite contents and location; if you're working from a clean install of MPLAB, they'll be correct.

Next, select "Create New Project File" then browse to where to you want to keep your project files, creating a new directory if appropriate. It's easiest to keep your files straight if you create a new directory

for the project, but you may already have a project directory if, for example, you have created circuit diagrams or other documentation for the project. I normally keep everything related to a project in a single directory. Then enter a descriptive file name and click on "Save". For example:



You should end up back in the Project Wizard window:



Microchip supplies templates you can use as the basis for new code. It's a good idea to use these until you develop your own. Step 4 of the project wizard allows you to copy the appropriate template into your project, and select a linker script to use.

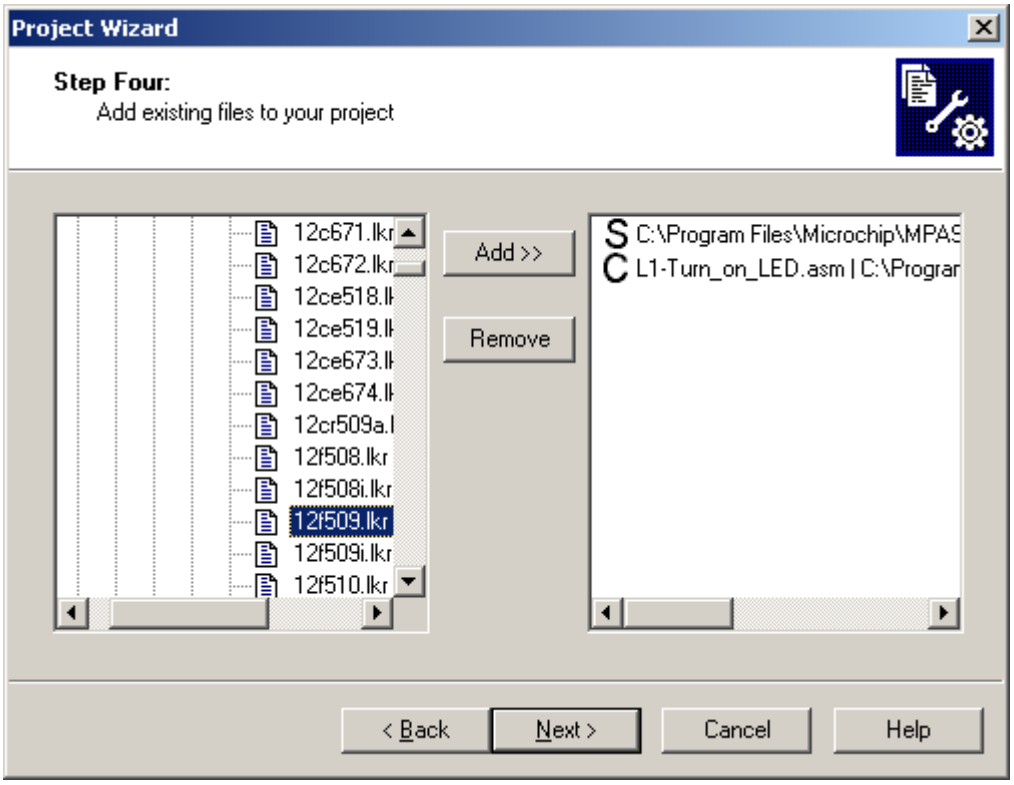
In the left hand pane, navigate to 'C:\Program Files\Microchip\MPASM Suite\Template\Object' and select '12F509TMPO.ASM', then click on the "Add>>" button. The file should appear in the right hand pane, with an "A" to the left of it. The "A" stands for "Auto". In auto mode, MPLAB will guess whether the

file you have added should be referenced via a *relative* or *absolute* path. Relative files are those that should move with the project (if it is ever moved or copied). Absolute files should always remain in a fixed location; they don't belong specifically to your project and are more likely to be shared with others. Clicking the "A" will change it to "U", indicating a "User" file which should be referenced through a relative path, or "S", indicating a "System" file which should have an absolute reference.

We don't want either; we need to copy the template file into the project directory, so click on the "A" until it changes to "C", for "Copy". When you do so, it will show the destination file name next to the "C". Of course, you're not going to want to call your copy '12F509TMPO.ASM', so click the file name and rename it to something like 'L1-Turn_on_LED.asm'.

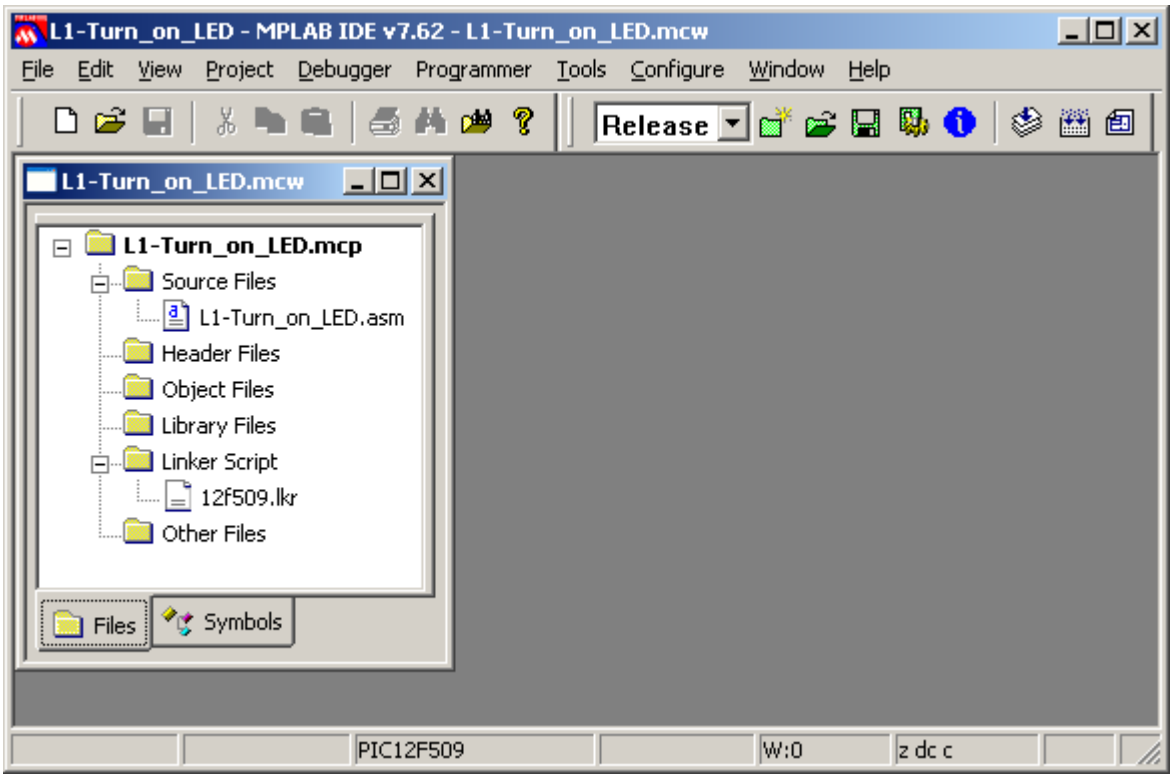
Next you need to tell MPLAB which linker script to use. In the left hand pane, navigate to 'C:\Program Files\Microchip\MPASM Suite\LKR' and select '12f509.lkr', then click the "Add>>" button again. We won't be changing the linker file, only referring to it. It's ok to leave it in the MPASM directory and refer to it there. So to tell MPASM that this is a system file, click on the "A" next to the file until it changes to an "S". Or you could leave it in auto mode; MPLAB will usually guess correctly. But it's better to select "S" if you know it is a system file.

The window should now look similar to that shown below, with two project files selected in the right hand pane: one linker file remaining in the Microchip directory and one assembler file renamed and copied to your project directory:

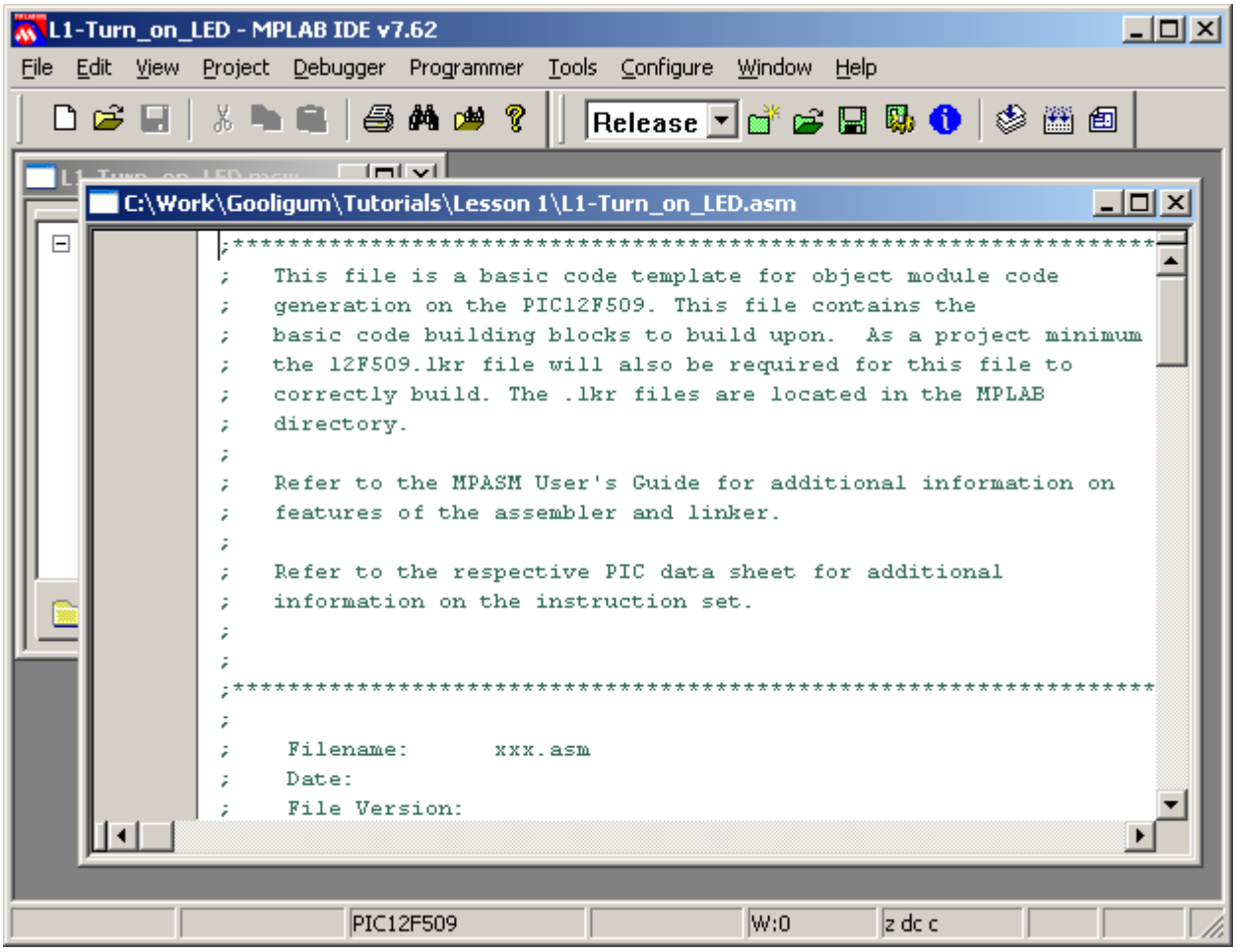


After you click "Next" and then "Finish" on the final Project Wizard window, you should be presented with an empty MPLAB workspace. You may need to select View → Project to see the project window, which shows your project and the files it comprises as a tree structure. For a small, simple project like this, the project window will show only two files: an assembler source file (.asm) and a linker script (.lkr).

Your MPLAB IDE window should be similar to that illustrated at the top of the next page:



To get started writing your program, double-click "L1-Turn_on_LED.asm". You'll see a text editor window open; finally you can see some code!



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. If you right-click in the editor window, you can set editor properties, such as auto-indent and line numbering, but you'll find that the defaults are quite usable to start with.

We'll take a look at what's in the template, and then add the instructions needed to turn on the LED.

Template Code

The first section of the templates is a series of blocks of comments.

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

The template begins with some general instructions telling us that "this is a template" and "refer to the data sheet". We already know all that, so the first block of comments can be deleted.

The following comment blocks illustrate the sort of information you should include at the start of each source file: what it's called, modification date and version, who wrote it, and a general description of what it does. There's also a "Files required" section. This is useful in larger projects, where your code may rely on other modules; you can list any dependencies here. It is also a good idea to include information on what processor this code is written for; useful if you move it to a different PIC later. You should also document what each pin is used for. It's common, when working on a project, to change the pin assignments – often to simplify the circuit layout. Clearly documenting the pin assignments helps to avoid making mistakes when they are changed!

For example:

```
;*****************************************************************************
;                                                                            *
;   Filename:       BA_L1-Turn_on_LED.asm                                    *
;   Date:           7/9/07                                                   *
;   File Version:   0.1                                                      *
;                                                                            *
;   Author:         David Meiklejohn                                         *
;   Company:        Gooligum Electronics                                     *
;                                                                            *
;*****************************************************************************
;                                                                            *
;   Architecture:   Baseline PIC                                             *
;   Processor:      12F508/509                                               *
;                                                                            *
;*****************************************************************************
;                                                                            *
;   Files required: none                                                     *
;                                                                            *
;*****************************************************************************
;                                                                            *
;   Description:    Lesson 1, example 1                                      *
;                                                                            *
;   Turns on LED.  LED remains on until power is removed.                    *
;                                                                            *
;*****************************************************************************
;                                                                            *
;   Pin assignments:                                                         *
;       GP1 - indicator LED                                                  *
;                                                                            *
;*****************************************************************************
```

Note that the file version is '0.1'. I don't call anything 'version 1.0' until it works; when I first start development I use '0.1'. You can use whatever scheme makes sense to you, as long as you're consistent.

Next in the template, we find:

```
list      p=12F509            ; list directive to define processor
#include <p12F509.inc>        ; processor specific variable definitions
```

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 shots above.

Nevertheless, you should always use the `list` directive at the start of your assembler source file. If you rely only on the setting in MPLAB, mistakes can easily happen, and you'll end up with unusable code, assembled for the wrong processor. If there is a mismatch between the `list` directive and MPLAB's setting, MPLAB will warn you when you go to assemble, and you can catch and correct the problem.

The next line uses the `#include` directive which causes an *include file* (p12F509.inc, located in 'C:\Program Files\Microchip\MPASM Suite') to be read by the assembler. This file sets up aliases for all the features of the 12F509, so that we can refer to registers etc. by name (e.g. 'GPIO') instead of numbers. Lesson 6 explains how this is done; for now we'll simply used these pre-defined names, or *labels*.

Note that in the comments, the processor is given as "12F508/509", suggesting that this code will work on either a 12F508 or 12F509, unchanged. But if you do use a 12F508 instead of a 12F509, you must change these two lines to reflect that. And also change the linker script to 12f508.lkr, instead of 12f509.lkr. These three things – the linker script, the `list` directive and the include file – are specific to the processor. If you remember that, it's easy to move code to other PICs later.

Next we have:

```
__CONFIG   _MCLRE_OFF & _CP_OFF & _WDT_OFF & _IntRC_OSC
```

This sets the processor configuration. The 12F509 has a number of options that are set by setting various bits in a "configuration word" that sits outside the normal address space. The `__CONFIG` directive is used to set these bits as needed. We'll examine these in greater detail in later lessons, but briefly the options being set here are:

- `_MCLRE_OFF`
  Disables the external reset, or "master clear" ($\overline{\text{MCLR}}$). If enabled, the processor will be reset if pin 4 is pulled low. If disabled, pin 4 can be used as an input: GP3.
  That's why, on the circuit diagram, pin 4 is labelled "GP3/MC"; it can be either an input pin or an external reset, depending on the setting of this configuration bit.
  The LPC Demo Board includes a pushbutton which will pull pin 4 low when pressed, resetting the PIC if external reset is enabled. The PICkit 2 is 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 code protection. When your code is in production and you're selling PIC-based products, you may not want others (e.g. competitors) stealing your code. If you use `_CP_ON` instead, your code will be protected, meaning that if someone tries to use a PIC programmer to read it, all they will see are zeros.
- `_WDT_OFF`
  Disables the watchdog timer. This is a way of automatically restarting a crashed program; if the program is running properly, it continually resets the watchdog timer. If the timer is allowed to expire, the program isn't doing what it should, so the chip is reset and the crashed program restarted. Very useful in production, but a nuisance when prototyping, so we leave it disabled.

- _IntRC_OSC
  Selects the internal RC oscillator as the clock source. Every processor needs a clock – a regular source of cycles, used to trigger processor operations such as fetching the next program instruction. PICs support a number of clock options, as we'll see in lesson 7. Most modern PICs, including the 12F509, include an internal 'RC' oscillator, which can be used as the simplest possible clock source, since it's all on the chip! It's built from passive components – resistors and capacitors (hence the name RC), and is not as accurate as an external crystal. But to turn on an LED, we don't need accurate timing.
  The internal RC oscillator runs at approximately 4MHz. Program instructions are processed at one quarter this speed: 1MHz, or 1µs per instruction.

The comments after the __CONFIG directive can be deleted; we know what it is for.

The next piece of template code demonstrates how to define variables:

```
;***** VARIABLE DEFINITIONS
TEMP_VAR    UDATA
temp        RES     1             ;example variable definition
```

The UDATA directive tells the linker that this is the start of a *section* of uninitialised data. This is data memory space that is simply set aside for use later. The linker will decide where to place it in data memory. The label, such as 'TEMP_VAR' here, is only needed if there is more than one UDATA section.

The RES directive is used to reserve a number of memory locations. Each location in data memory is 8 bits, or 1 byte, wide, so in this case, 1 byte is being reserved for a *variable* called 'temp'. The address of the variable is assigned when the code is linked (after assembly), and the program can refer to the variable by name (i.e. temp), without having to know what its address in data memory is.

We'll use variables in later tutorials, but since we don't need to store any data to simply turn on an LED, this section can be deleted.

So far, we haven't seen a single PIC instruction. It's only been assembler or linker directives. The next piece of template code introduces our first instruction:

```
;*******************************************************************
RESET_VECTOR      CODE   0x3FF      ; processor reset vector

; Internal RC calibration value is placed at location 0x3FF by Microchip
; as a movlw k, where the k is a literal value.
```

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

The 0x3FF after CODE is an address in hexadecimal (signified in MPASM by the '0x' prefix). Program memory on the 12F509 extends from 000h to 3FFh. This CODE directive is telling the linker to place the section of code that follows it at 0x3FF – the very top of the 12F509's program memory.

But there *is* no code following this first CODE directive, so what's going on? Remember that the internal RC oscillator is not as accurate as a crystal. To compensate for that inherent inaccuracy, Microchip uses a calibration scheme. The speed of the internal RC oscillator can be varied over a small range by changing the value of the OSCCAL register (refer back to the register map). Microchip tests every 12F509 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 an instruction placed at the top of the program memory (0x3FF). The instruction placed there is:

```
movlw k
```

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

'movlw' is our first PIC assembler instruction. It loads the W register with an 8-bit value (between 0 and 255), which may represent a number, character, or something else.

Microchip calls a value like this, one that is embedded in an instruction, a *literal*. It refers to a load or store operation as a 'move' (even though nothing is moved; the source never changes).

So, 'movlw' means "**mov**e **l**iteral to **W**".

When the 12F509 is powered on or reset, the first instruction it executes is this movlw instruction at 0x3FF. After executing this instruction, the W register will hold the factory-set calibration value. And after executing an instruction at 0x3FF, there is no more program memory, so the *program counter*, which points to the next instruction to be executed "wraps around" to point at the start of memory – 0x000.

0x3FF is the 12F509's true "reset vector" (where code starts running after a reset), but 0x000 is the effective reset vector, where you place your own code.

Confused?

What it really boils down to is that, when the 12F509 starts, it picks up a calibration value stored at the top of program memory, and then starts executing code from the start of program memory. Since you should never overwrite the calibration value at the top of memory, the start of your code will always be placed at 0x000, and when your code starts, the W register will hold the oscillator calibration value.

Since the "RESET_VECTOR" code section, above, is really only there for documentation (telling you what you should already know from reading the data sheet), to avoid confusion we'll simply leave it out.

You can choose to use the oscillator calibration value, or simply ignore it. But if you're using the internal RC oscillator, you should immediately copy this value to the OSCCAL register, to calibrate the oscillator with the factory setting, and that's what the next piece of code from the template does:

```
MAIN  CODE    0x000
      movwf   OSCCAL            ; update register with factory cal value
```

This CODE directive tells the linker to place the following section of code at 0x000 – the effective reset vector.

The 'movwf' instruction copies (Microchip would say "moves") the contents of the W register into the specified register – "**mov**e **W** to **f**ile register".

In this case, W holds the factory calibration value, so this instruction writes that calibration value into the OSCCAL register.

At this point, all the preliminaries are out of the way. The processor has been specified, the configuration set, the oscillator calibration value updated, and program counter pointing at the right location to run user code. The final bit of template code is simply an example showing where and how to place your code:

```
start
      nop                       ; example code
      movlw   0xFF              ; example code
      movwf   temp              ; example code

; remaining code goes here


      END                       ; directive 'end of program'
```

'start' is an example of a program label, used in loops, branches and subroutines. It's not necessary to label the start of your code 'start'. Nor is it necessary to label the main code section 'MAIN'. But it does make it easier to follow the code.

'nop' is a "**no** **op**eration" instruction; it does nothing other than waste an instruction cycle – something you might want to do as part of a delay loop (we'll look at examples in the next lesson).

'movlw' and 'movwf' we've seen before.

'END' is an assembler directive, marking the end of the program source. The assembler will ignore any text after the 'END' directive – so it really should go right at the end!

Of course, we need to replace these example instructions with our own. This is where we place the code to turn on the LED!

Turning on the LED

To turn on the LED on GP1, we need to do two things:

- Configure GP1 as an output
- Set GP1 to output a high voltage

We could leave the other pins configured as inputs, or set them to output a low. Since, in this circuit, they are not connected to anything, it doesn't really matter. But for the sake of this exercise, We'll configure them as inputs.

When the 12F509 is powered on, all pins are configured by default as inputs, and the contents of the port register, GPIO, is undefined.

To configure GP1 as an output, we have to write a '0' to bit 1 of the TRIS register. This is done by:

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

The 'tris' instruction stores the contents of W into a **TRIS** register.

Although there is only one TRIS register on the 12F509, it is still necessary to specify 'GPIO' (or equivalently the number 6, but that would be harder to follow) as the operand.

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

To set the GP1 output to 'high', we have to set bit 1 of GPIO to '1'. This can be done by:

```
movlw   b'000010'       ; set GP1 high
movwf   GPIO
```

As the other pins are all inputs, it doesn't matter what they are set to.

Note again that, to place a value into a register, you first have to load it into W. You'll find that this sort of load/store process is common in PIC programming.

Finally, if we leave it there, when the program gets to the end of this code, it will restart. So we need to get the PIC to just sit doing nothing, indefinitely, 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.

Complete program

Putting together all the above, here's the complete assembler source needed 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=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
        movlw   b'000010'       ; set GP1 high
        movwf   GPIO

        goto    $               ; loop forever


        END
```

That's it! Not a lot of code, really…

Building the Application

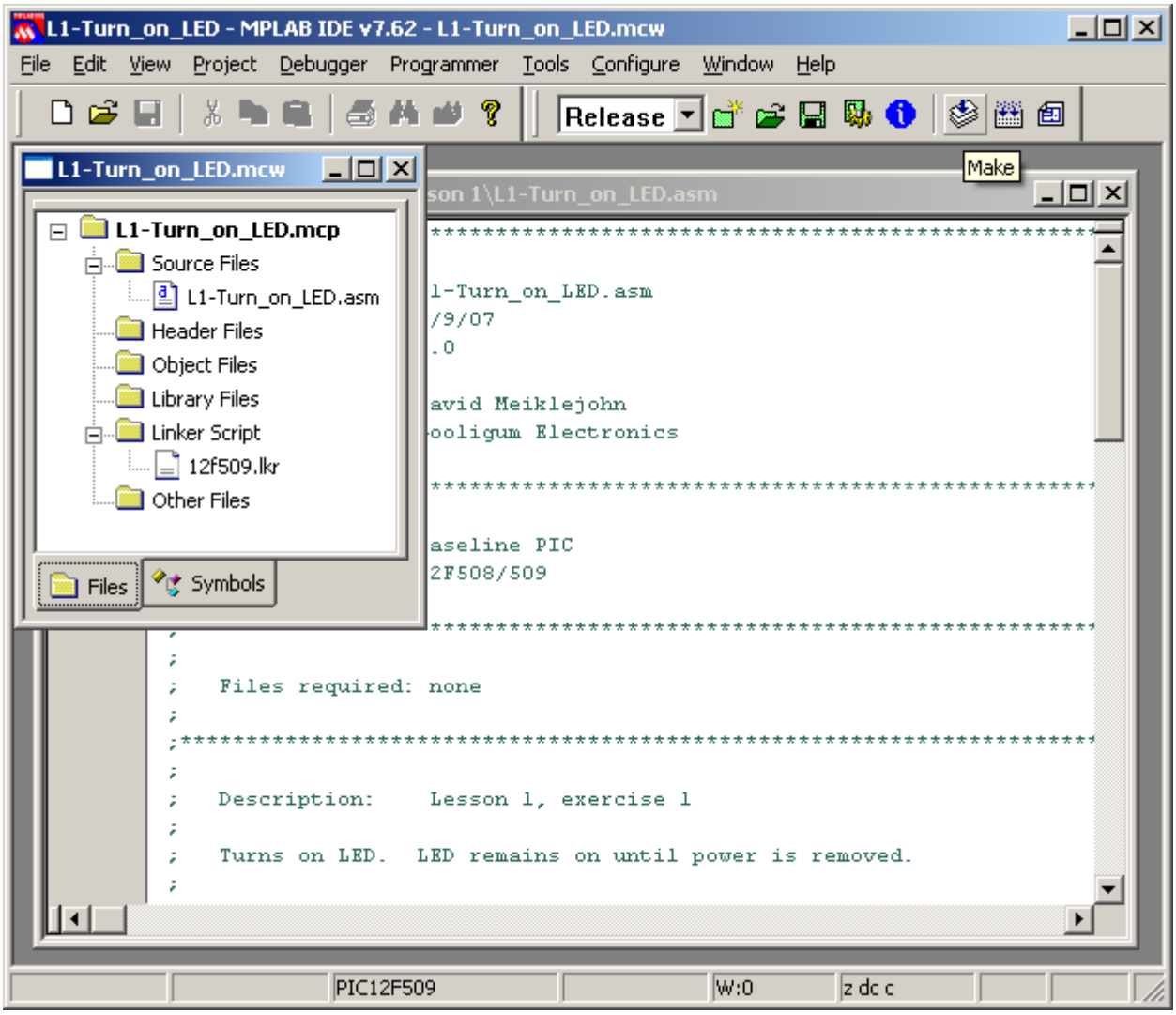
Now that we have the complete assembler source, we can build the final application code to be loaded into the PIC.

This is done in two steps:

- Assemble the source files to create object files
- Link the object files to build the executable code

Normally this is transparent; MPLAB does both steps for you in a single operation. The fact that, behind the scenes, there are multiple steps only becomes important when you start working with projects that consist of multiple source files or libraries of pre-assembled routines.

To build the project, select the "Project → Make" menu item, press F10, or click on the "Make" toolbar button:

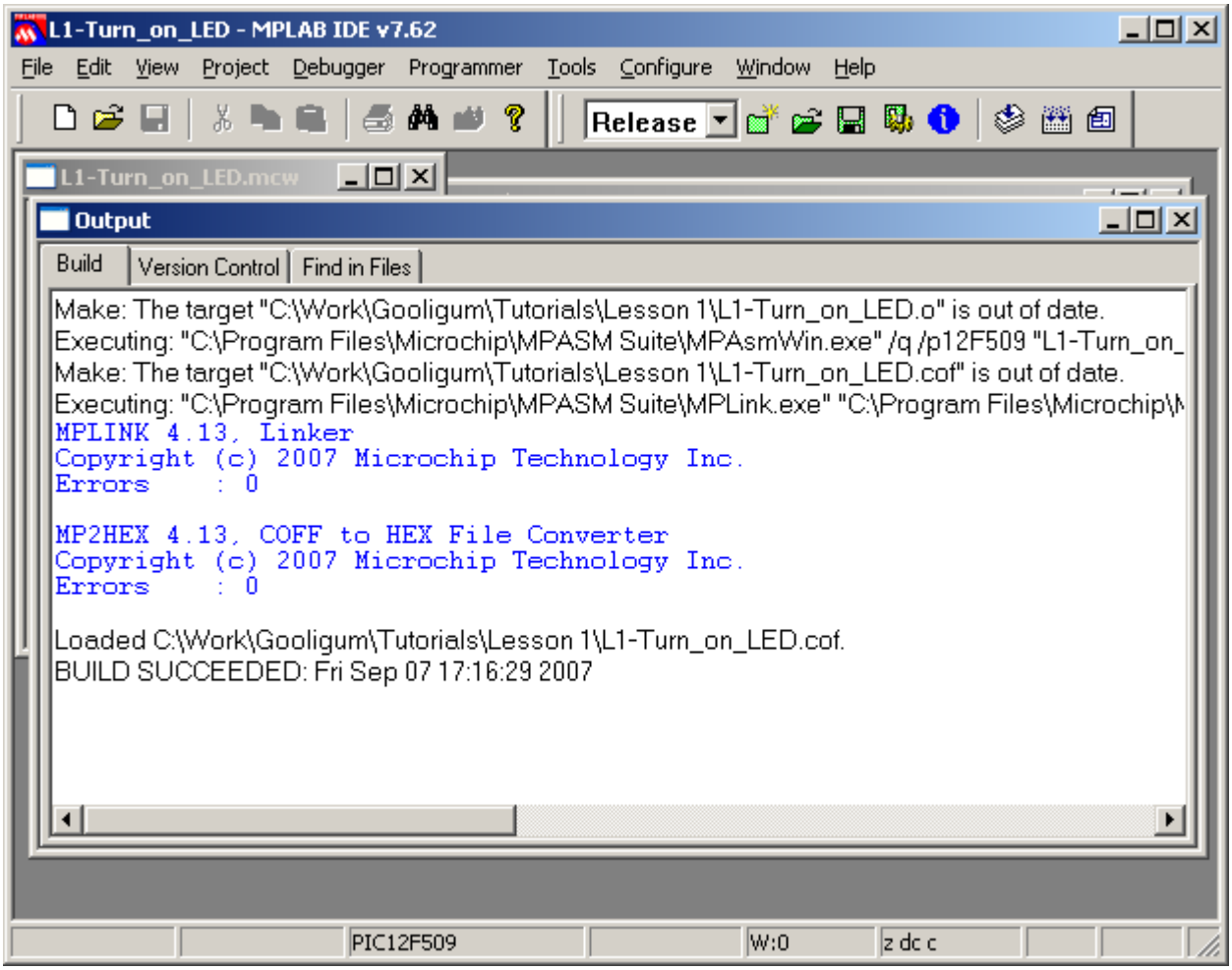


"Make" will assemble any source files which need assembling (i.e. ones which have changed since the last time the project was built), then link them together.

The other option is "Project → Build All" (Ctrl+F10), which assembles all the source files, regardless of whether they have changed (are "out of date") or not.

For a small, single-file project like this, "Make" and "Build All" have the same effect; you can use either. In fact, the only reason to use "Make" is that, in a large project, it saves time to not have to re-assemble everything each time a single change is made.

When you build the project (run "Make"), you'll see text in the Output window similar to this:



The first two lines show the code being assembled. If you see any errors or warnings, you probably have a syntax error somewhere, so check your code against the listing above (page 15).

The next two lines show the code being linked. The linker, "MPLINK", creates a "COFF" object module file, which is converted into a "*.hex" hex file, which contains the actual machine codes to be loaded into the PIC.

In addition to the hex file, other outputs of the build process include a "*.lst" list file which allows you to see how MPASM assembled the code and the values assigned to symbols, and a "*.map" map file, which shows how MPLINK laid out the data and code segments in the PIC's memory.

Programming the PIC using the PICkit 2

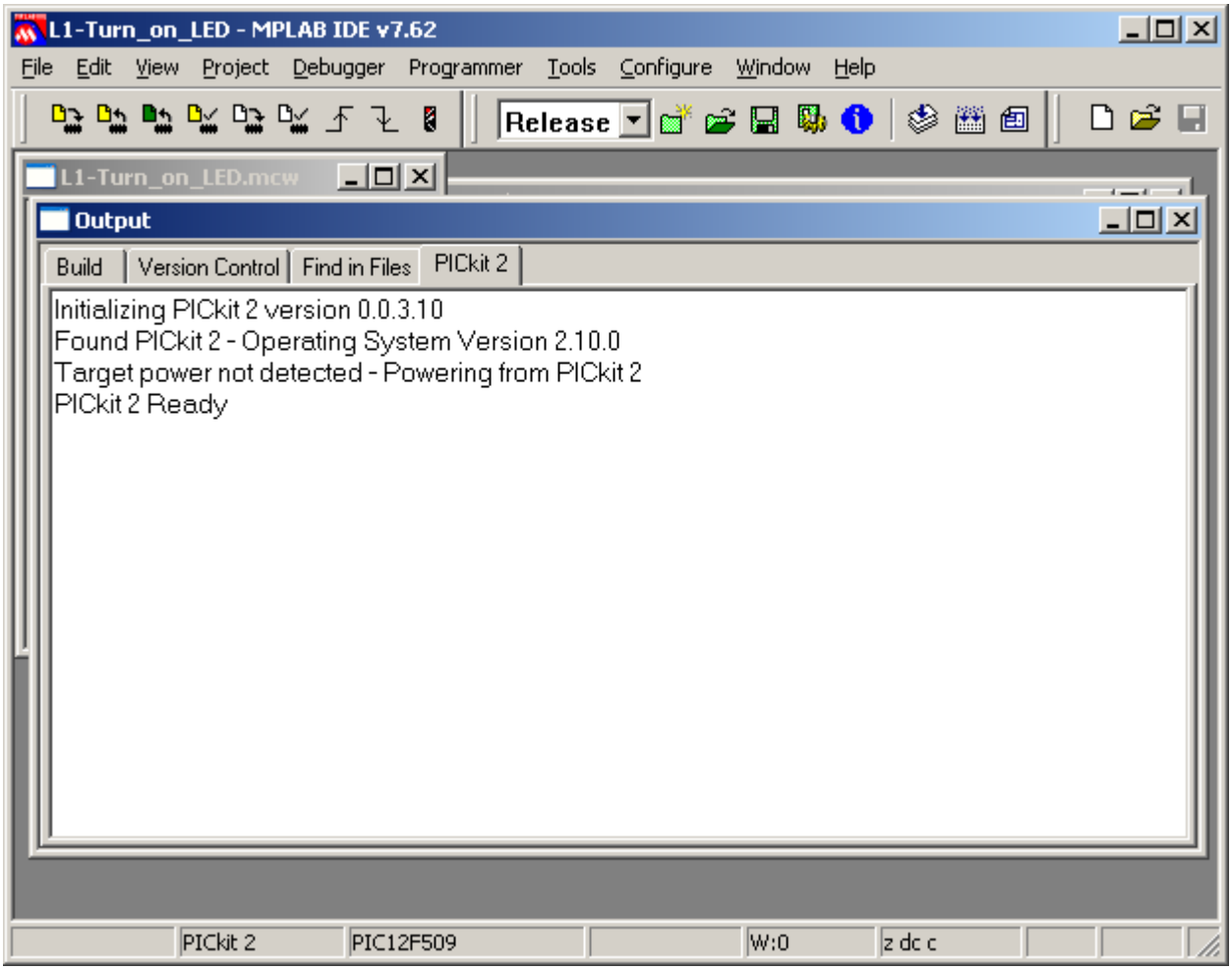
The final step is to load the final assembled and linked code into the PIC.

If you are using the PICkit 2 programmer, the PIC12F509 can be conveniently programmed from within MPLAB, assuming you are using MPLAB 8.0 or later.

First, ensure that the PICkit 2 is connected to a USB port on your PC and that the Low Pin Count Demo Board is plugged into the PICkit 2, with the PIC12F509 correctly installed in the IC socket.

Then select the PICkit 2 option from the "Programmer → Select Programmer" menu item.

This will initialise the PICkit 2, and you should see messages in the Output window similar to that shown at the top of the next page:



If the operating system in the PICkit 2 is out of date, you will see some additional messages while it is updated.

If you don't get the "Found PICkit 2" and "PICkit 2 Ready" messages, you have a problem somewhere and should check that everything is plugged in correctly.

Note that an additional toolbar appeared: 

The first icon (on the left) is used to initiate programming. When you click on it, you should see messages similar to these:

```
Programming Target (7/09/2007  6:02:22 PM)
Erasing Target
Programming Program Memory (0x0 - 0x7)
Verifying Program Memory (0x0 - 0x7)
Programming Configuration Memory
Verifying Configuration Memory
PICkit 2 Ready
```

Your PIC is now programmed!

But the LED is not yet lit.

That is because, by default, the PICkit 2 holds the $\overline{\text{MCLR}}$ line low after programming. Since we have used the `_MCLRE_ON` option, enabling external reset, the PIC is held in reset and the program will not run.

If we had not configured external resets, the LED would have lit as soon as the PIC was programmed.

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

The LED should now light!

Congratulations! You've taken your first step in PIC development!

That first step is the hardest. Now we have a solid base to build on.

In the next lesson, we'll make the LED flash…