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 configuring the PIC

- 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.

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 a Microchip PICkit 2or PICkit 3 programmer and either the Gooligum Baseline and Mid-range PIC Training and Development Board or Microchip's Low Pin Count (LPC) 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.

We're going to start with the simplest PIC of all – the PIC10F200, a 6-pin "baseline" device[1]. It is only capable of simple digital I/O and does not 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 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 baseline training board

- A PIC10F200-I/P microcontroller (supplied with the Gooligum training board)

---

[1] If you are using Microchip's LPC Demo Board, you will have to substitute a PIC12F508, because the LPC board does not support the 10F PIC family.

You could use Microchip's new MPLAB X IDE software, which runs under Linux and Mac OS X, as well as Windows, instead of MPLAB 8.

Note however that Microchip's Low Pin Count Demo Board does not support the 6-pin 10F PICs. If you are using the LPC demo board, you can substitute a 12F508 (the simplest 8-pin baseline PIC) instead of the 10F200, with only a minor change in the program code, which we'll highlight later.

The four LEDs on the LPC demo board don't work (directly) with 8-pin PICs, such as the 12F508. So, to complete this lesson, using an LPC demo board, you need to either add an additional LED and resistor to the prototyping area on your board, or use some solid core hook-up wire to patch one of the LEDs to the appropriate PIC pin, as described later.

This is one reason the Gooligum training board was developed to accompany these tutorials – if you have the Gooligum board, you can simply plug in your 10F or 12F PIC, and go.

Introducing the PIC10F200

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 10F200 from www.microchip.com.

You'll find that the data sheet for the 10F200 also covers the 10F202, 10F204 and 10F206. These are essentially variants of the same chip. The differences are as follows:

Device	Program Memory (words)	Data Memory (bytes)	I/O pins	Comparators	Clock rate
10F200	256	16	4	0	4 MHz
10F202	512	24	4	0	4 MHz
10F204	256	16	4	1	4 MHz
10F206	512	24	4	1	4 MHz

The 10F202 and 10F206 have more memory, and the 10F204 and 10F206 include a comparator (used to compare analog signals – see lesson 9), but they are otherwise the same.

The 10F family are all 6-pin PICs, with only four pins available for I/O (input and output). They are typically used to implement simple functions, such as replacing digital logic, in space-constrained situations, so are generally used as tiny surface-mount devices.

However, they are also available in 8-pin DIP packages, as shown here.
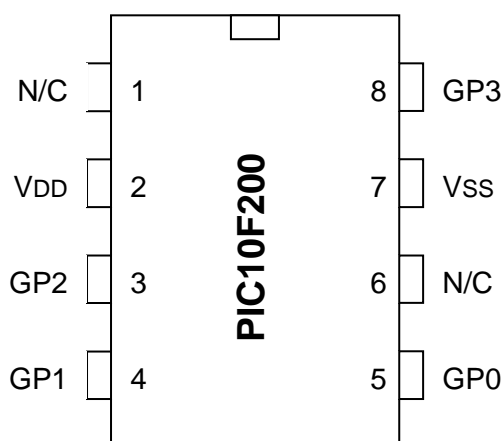
These 8-pin DIP packages make prototyping easy, so we will use them in these lessons.

Note however, that although this is an 8-pin package, the PIC10F200 is still a 6-pin device. Only six of these pins are usable: the pins marked 'N/C' are not connected, and cannot be used.

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. It is a good practice to 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 GP3, 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 and GP2 – can be individually set to be inputs or outputs.

PIC10F200 Registers

Address

00h	INDF
01h	TMR0
02h	PCL
03h	STATUS
04h	FSR
05h	OSCCAL
06h	GPIO

10h	General
	Purpose
	Registers
1Fh	

TRIS

OPTION

W

8-bit PICs use a so-called Harvard architecture, where program and data memory is entirely separate.

In the 10F200, program memory extends from 000h to 0FFh (hexadecimal). Each of these 256 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 10F200 is shown at left. The first seven registers are special-function, each with a specific address. They are followed by sixteen general purpose resisters, which you can use to store program variables such as counters. Note that there are no registers from 07h to 0Fh on the 10F200.

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.

PIC10F200 Input and Output

As mentioned above, the 10F200 has four I/O pins: GP0, GP1 and GP2, which can be used for digital input and output, plus GP3, which is input-only.

Taken together, the four 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 voltage[2] on the corresponding pin; setting it to '0' outputs a low voltage[3].

If a pin is configured as an input, the input level is represented by the corresponding bit in the GPIO register. If the voltage on an input pin is high[4], the corresponding bit reads as '1'; if the input voltage is low[5], the corresponding bit reads as '0':

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
GPIO					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						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 10F200 can source or sink (i.e. current 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.

So, if you were driving four LEDs, and it is possible for all to be on at once, you should limit the current in each LED to 18 mA, so that the total for the port will never be more than 75 mA, even though each pin can supply up to 25 mA on its own.

PICs are tough devices, and you may get away with exceeding these limits – but if you ignore the absolute maximum ratings specified in the data sheet, you're on your own. Maybe your circuit will work, maybe not. Or maybe it will work for a short time, before failing. It's better to follow the data sheet…

---

[2] a 'high' output will be close to the supply voltage (V_{DD}), for small pin currents

[3] a 'low' output is less than 0.6 V, for small pin currents

[4] the threshold level depends on the power supply, but a 'high' input is any voltage above 2.0 V, given a 5 V supply

[5] a 'low' input is anything below 0.8 V, given a 5 V supply – see the data sheet for details of each of these levels

Introducing the PIC12F508

If you're using the Microchip Low Pin Count Demo Board, you can't use a PIC10F200, because that board doesn't support the 10F family.

The simplest baseline PIC that you can use with the LPC demo board is the 8-pin 12F508.

As with the 10F200, the data sheet for the 12F508 also covers some related variants: in this case the 12F509 and 16F505. 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).

The 12F508 is essentially a 10F200 with more pins and memory.

It has six I/O pins (instead of four), labelled GP0 to GP5, in an 8-pin package. Like the 10F200, each pin can be configured as a digital input or output, except GP3, which is input-only.

The 12F508 has twice as much program memory as the 10F200 (512 words instead of 256), extending from 000h to 1FFh.

The register map, shown on the right, is nearly identical to that of the 10F200. The only difference is that the 12F508 has 25 general purpose registers, instead of 16, with no 'gaps' in the memory map.

PIC12F508 Registers

Address

00h	INDF
01h	TMR0
02h	PCL
03h	STATUS
04h	FSR
05h	OSCCAL
06h	GPIO
07h 1Fh	General Purpose Registers

TRIS

OPTION

W

PIC12F508

VDD	1	8	Vss
GP5	2	7	GP0
GP4	3	6	GP1
GP3	4	5	GP2

Like the 10F200, the 12F508's six pins are mapped into 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

And the pin directions (input or output) are controlled by corresponding bits in the TRIS register:

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

Note that bit 3 is greyed out. That's because, as with the 10F200, GP3 is always an input.

Most other differences between the 12F508 and 10F200 concern the processor oscillator (or clock, governing how fast the device runs) configurations, which we'll look at in lesson 7.

Example Circuit

We now have enough background information to design a circuit to light an LED.

We'll need a regulated power supply, let's assume 5 V, connected to VDD and VSS. And remember that we should add a bypass capacitor, preferably a 100 nF (or larger) ceramic, across it.

We'll also need an LED of course, and a resistor to limit the current.

Although the PIC10F200 or PIC12F508 can supply up to 25 mA from a single pin, 10 mA is more than enough to adequately light most LEDs. With a 5 V supply and assuming a red or green LED with a forward voltage of around 2 V, the voltage drop across the resistor will be around 3 V.

Applying Ohm's law, $R = V / I = 3\ V \div 10\ mA = 300\ \Omega$. Since precision isn't needed here (we only need "about" 10 mA), it's ok to choose the next highest "standard" E12 resistor value, which is 330 Ω. It means that the LED will draw less than 10 mA, but that's a good thing, because, if we're going to use a PICkit 2 or PICkit 3 to power the circuit, we need to limit overall current consumption to 25 mA, because that is the maximum current those programmers can supply.

Finally, we need to connect the LED to one of the PIC's pins.

We can't choose GP3, because it's input-only.

If you're using the Gooligum training board, you could choose any of the other pins, but if you use the Microchip LPC Demo Board to implement the circuit, it's not a good idea to use GP0, because it's connected to a trimpot on the LPC demo board, which would divert current from the LED.

So, we'll use GP1, giving the circuits (10F200 and 12F508 versions) shown on the right.

Simple, aren't they? Modern microcontrollers really do have minimal requirements.

Of course, some connections are also needed for the ICSP (programmer) signals. These will be provided by your development board, unless you are building the circuit yourself. But the circuit as shown here is all that is needed for the PIC to run, and light the LED.
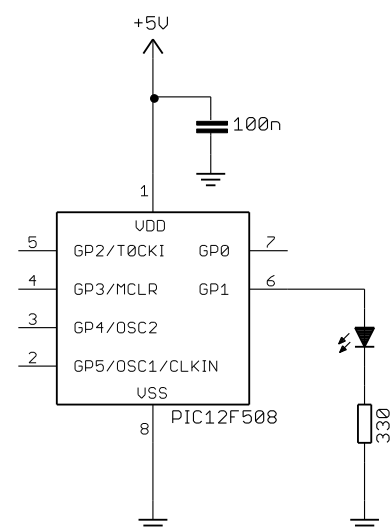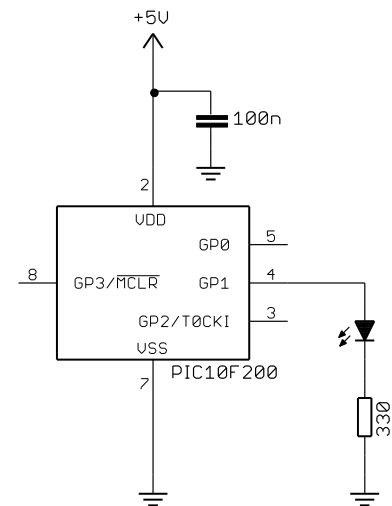
Gooligum training and development board instructions

If you have the Gooligum training board, you should use it to implement the first (10F200) circuit.

Plug the PIC10F200 into the 8-pin IC socket marked '10F'.[6]

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.

Microchip Low Pin Count Demo Board instructions

If you are using Microchip's LPC Demo Board, you'll need to take some additional steps.

Although the board provides four LEDs, they cannot be used directly with a 12F508 (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 330 Ω resistor and a piece of wire to the LPC Demo Board, as illustrated on the right.

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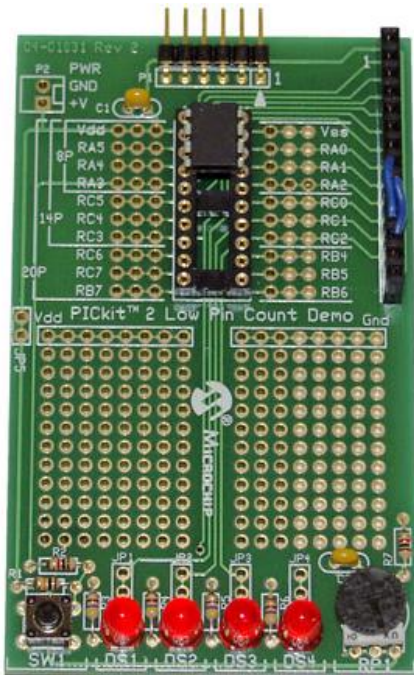
Note that on the LPC Demo Board, the pins are labelled 'RA1', 'RA2', etc., because that is the nomenclature used on the larger 20-pin PICs, such as the 16F690. They correspond to the 'GP' pins on the 12F508 – simply another name for the same thing.

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 shown on the left. This header makes available all the 12F508'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.

---

[6] Ensure that no device is installed in the 12F/16F socket – you can only use one PIC at a time in the training board.

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, to allow you to build more complex circuits, such as those found in the later tutorial lessons: see, for example, lesson 8.

Time to move on to programming!

Development Environment

You'll need Microchip's MPLAB Integrated Development Environment (MPLAB IDE), which you can download from www.microchip.com.

As discussed in lesson 0, MPLAB comes in two varieties: the older, established, Windows-only MPLAB 8, and the new multi-platform MPLAB X. If you are running Windows, it is better to use MPLAB 8 as long as Microchip continues to support it, because it is more stable and in some ways easier to use. It also allows us to use all the free compilers referenced in the C tutorial series. However, it is clear that Microchip's future development focus will be on MPLAB X, and that it will become the only viable option. So in these tutorials we will look at how to install and use both MPLAB IDEs.

MPLAB 8.xx

When installing, if you choose the 'Custom' setup type, 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)
- ✓ Microchip Applications
    - ✓ MPASM Suite   (the assembler)
    - ✓ MPLAB IDE   (the development environment)
        - ✓ MPLAB SIM   (software simulator – extremely useful!)
        - ✓ PICkit 2
        - or PICkit 3 Programmer/Debugger

It's worth selecting the CCS PCB C compiler, since it will then be properly integrated with MPLAB; we'll see how to use it later, in the C tutorial series.

You may need to restart your PC.

You can then run MPLAB IDE.

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: in our case, the PIC10F200 or PIC12F508.

Then select MPASM as the active toolsuite, which 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, they'll be correct.

Next, select "Create New Project File" and browse to where to you want to keep your project files, creating a new folder if appropriate. Then enter a descriptive file name and click on "Save", as shown:
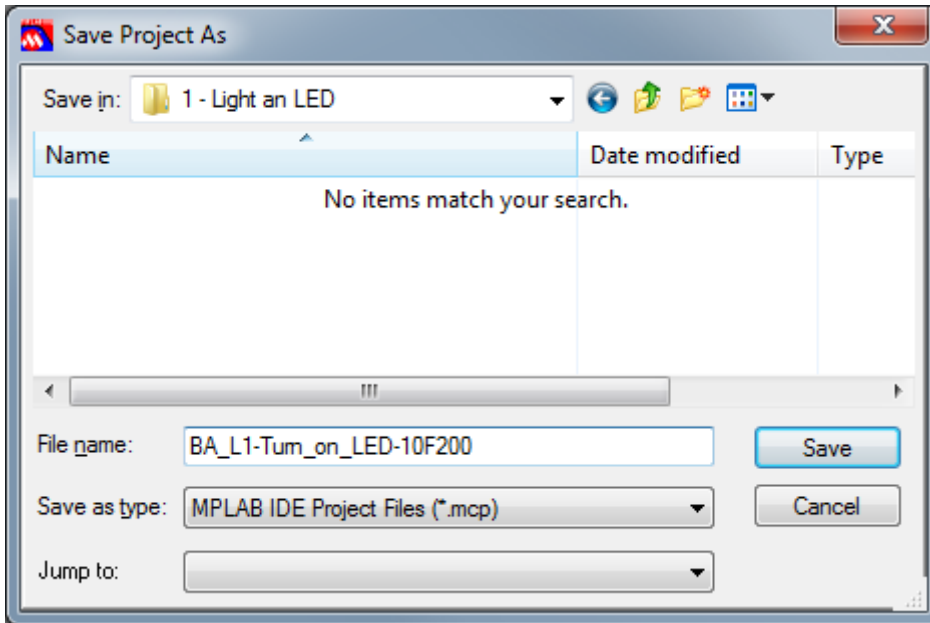
You should end up back in the Project Wizard window, shown below.
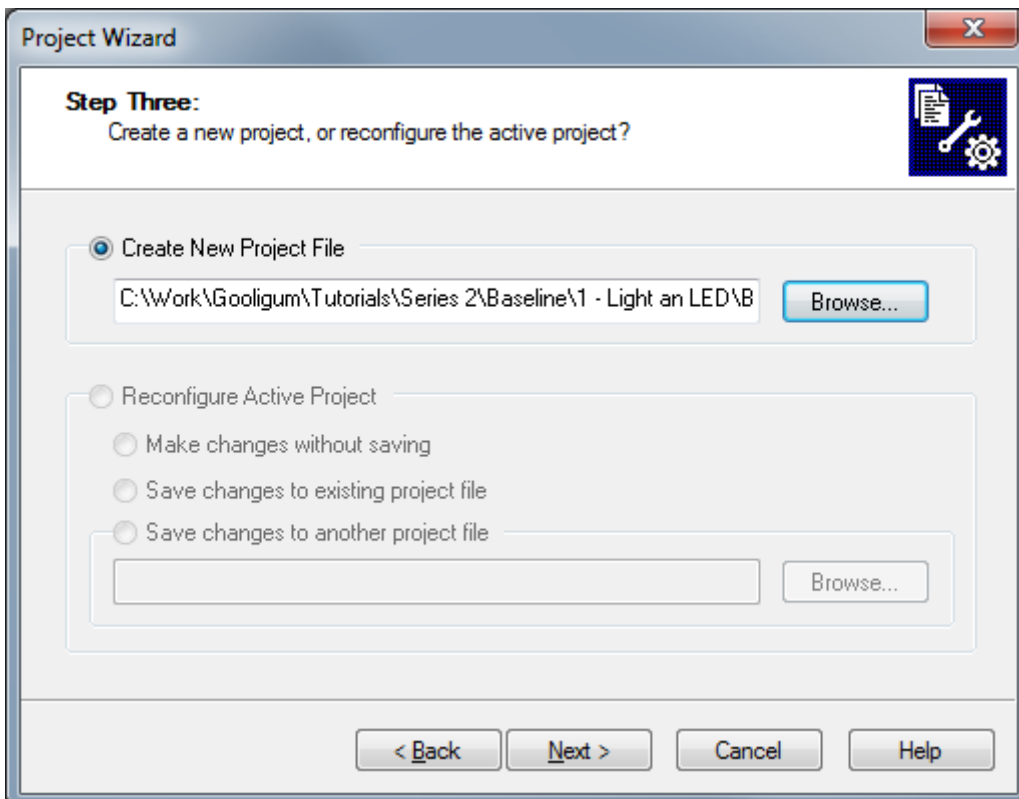
Microchip supplies templates you can use as the basis for new code. It's a good idea to use these until you develop your own. Step 4 of the project wizard allows you to copy the appropriate template into your project.

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.

The templates are located under the 'Template' directory, within the MPASM installation folder (usually 'C:\Program Files\Microchip\MPASM Suite'[7]). The templates for absolute code are found in the 'Code' directory, while those for relocatable code are found in the 'Object' directory. Since we'll be doing relocatable code development, we'll find the templates we need in '<MPASM Suite>\Template\Object'.

In the left hand pane, navigate to the '<MPASM Suite>\Template\Object' directory and select the template for whichever PIC you are using: '10F200TMPO.ASM' or '12F208TMPO.ASM'.


---
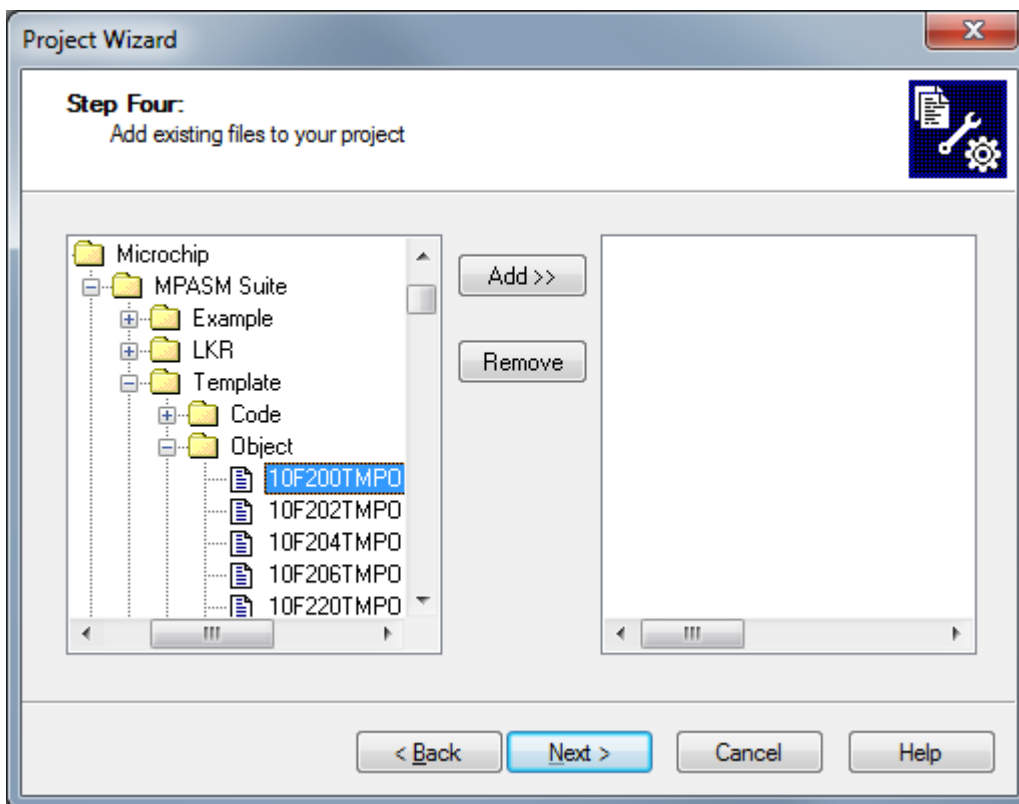
[7] On a 64-bit version of Windows, the MPASM folder will be under 'Program Files (x86)'

When you 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 '10F200TMPO.ASM', so click the file name and rename it to something more meaningful, like 'BA_L1-Turn_on_LED-10F200.asm'.

The window should now look similar to that shown below, with your (renamed) copy of the template file selected in the right hand pane:
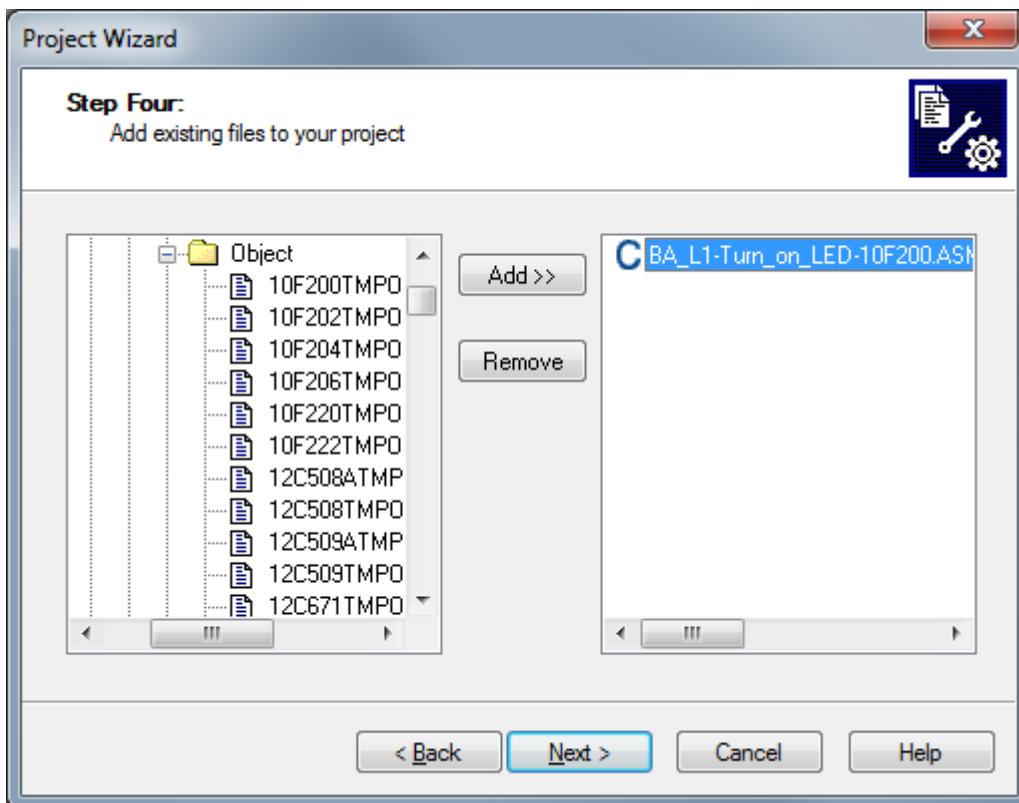

After you click "Next" and then "Finish" on the final Project Wizard window, you will be presented with an empty MPLAB workspace. You may need to select View $\rightarrow$ Project to see the project window, which shows your project and the files it comprises as a tree structure.

For a small, simple project like this, the project window will show only the assembler source file (.asm).

Your MPLAB IDE window should be similar to that illustrated at the top of the next page.
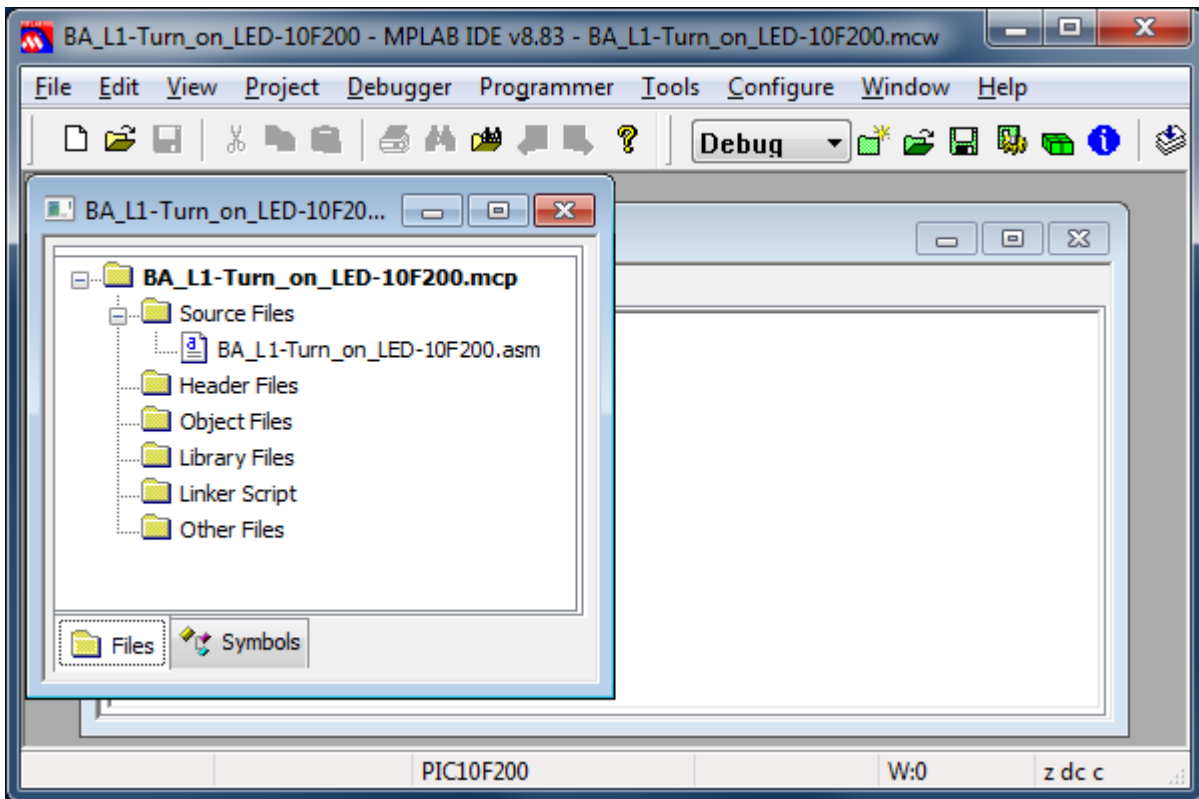
To get started writing your program, double-click your .asm source file. You'll see a text editor window open; finally you can see some code!
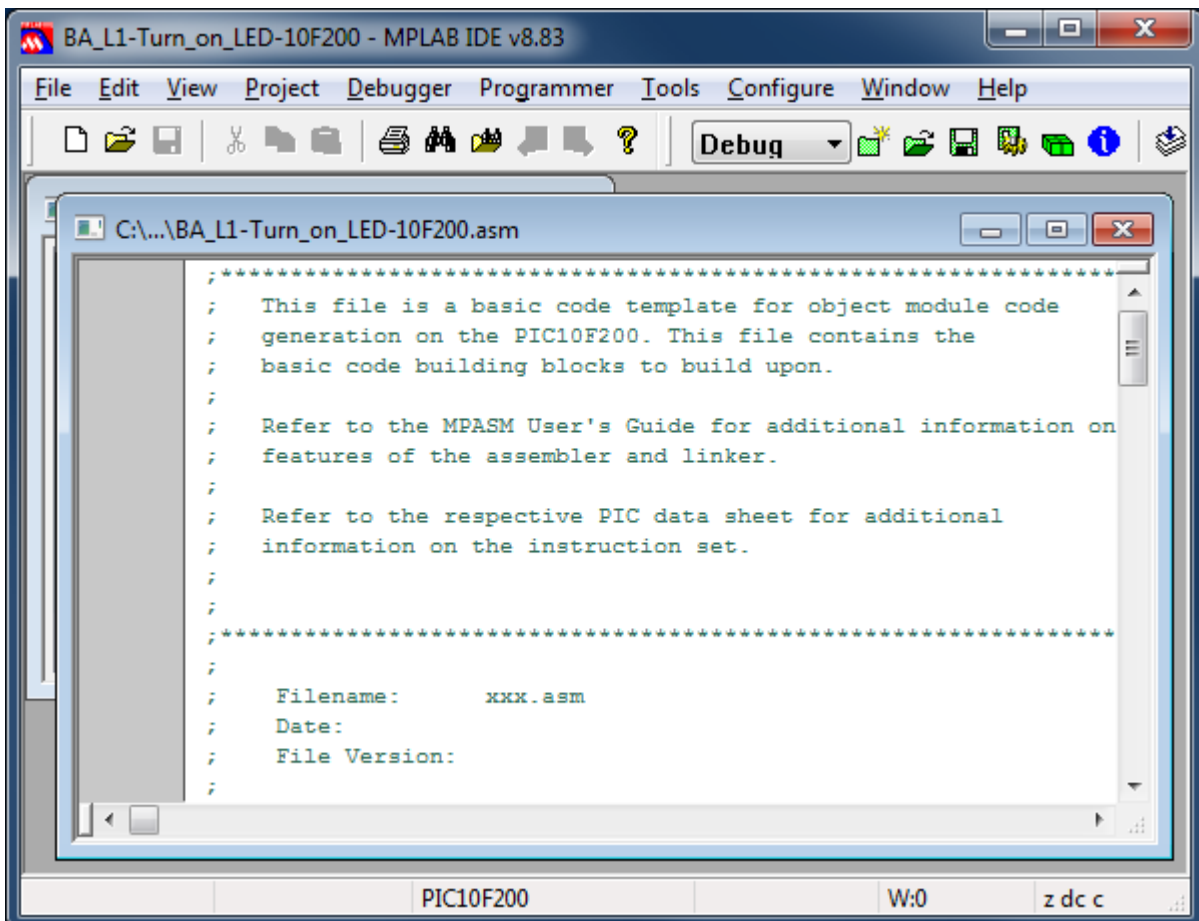
The MPLAB text editor is aware of PIC assembler (MPASM) syntax and will colour-code text, depending on whether it's a comment, assembler directive, PIC instruction, program label, etc. If you right-click in the editor window, you can set editor properties, such as auto-indent and line numbering, but 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.

But first it's a good idea to save your new project (Project → Save Project, or click on , or simply exit MPLAB and click 'Yes', when asked if you wish to save the workspace).

When you re-open your project in MPLAB, which you can easily do by double-clicking the '.mcp' project file in your project directory, you may be asked if you are developing absolute or relocatable code.

You should choose 'Relocatable'.

Later, after building your project (see below), you won't be asked this question. But if ever see this prompt, just choose 'Relocatable'.
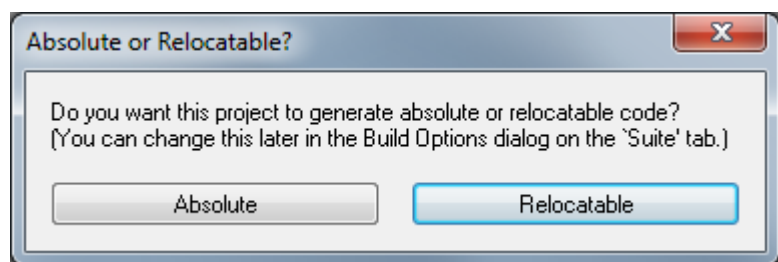
MPLAB X

You should download the MPLAB X IDE installer for your platform (Windows, Linux or Mac) from the MPLAB X download page at www.microchip.com, and then run it.

Unlike MPLAB 8, there are no installation options (other than being able to choose the installation directory). It's an "all or nothing" installer, including support for all of Microchip's PIC MCUs and development tools.

You can then run MPLAB X IDE.

Creating a New Project

When you first run MPLAB X, you will see the "Learn & Discover" tab, on the Start Page.

To start a new project, you should run the New Project wizard, by clicking on 'Create New Project'.

In the first step, you need to specify the project category. Choose 'Standalone Project'.

Next, select the PIC family and device.

In our case, we need 'Baseline 8-bit MCUs' as the family, and either the PIC10F200 or PIC12F508 as the device.

The third step allows you to optionally select a debug header. This is a device used to facilitate hardware debugging (see explanation in lesson 0), especially for PICs (such as the baseline devices we are using) which do not include internal hardware to support debugging. If you are just starting out, you are unlikely to have one of these debug headers, and you don't need one for these tutorials. So, you should not select a header. Just click 'Next'.

The next step is to select the tool you will use to program your PIC.

First, you should plug in the programmer (e.g. PICkit 2 or PICkit 3) you intend to use. If it is properly connected to your PC, with a functioning device driver[8], it will appear in the list of hardware tools, and you should select it, as shown:
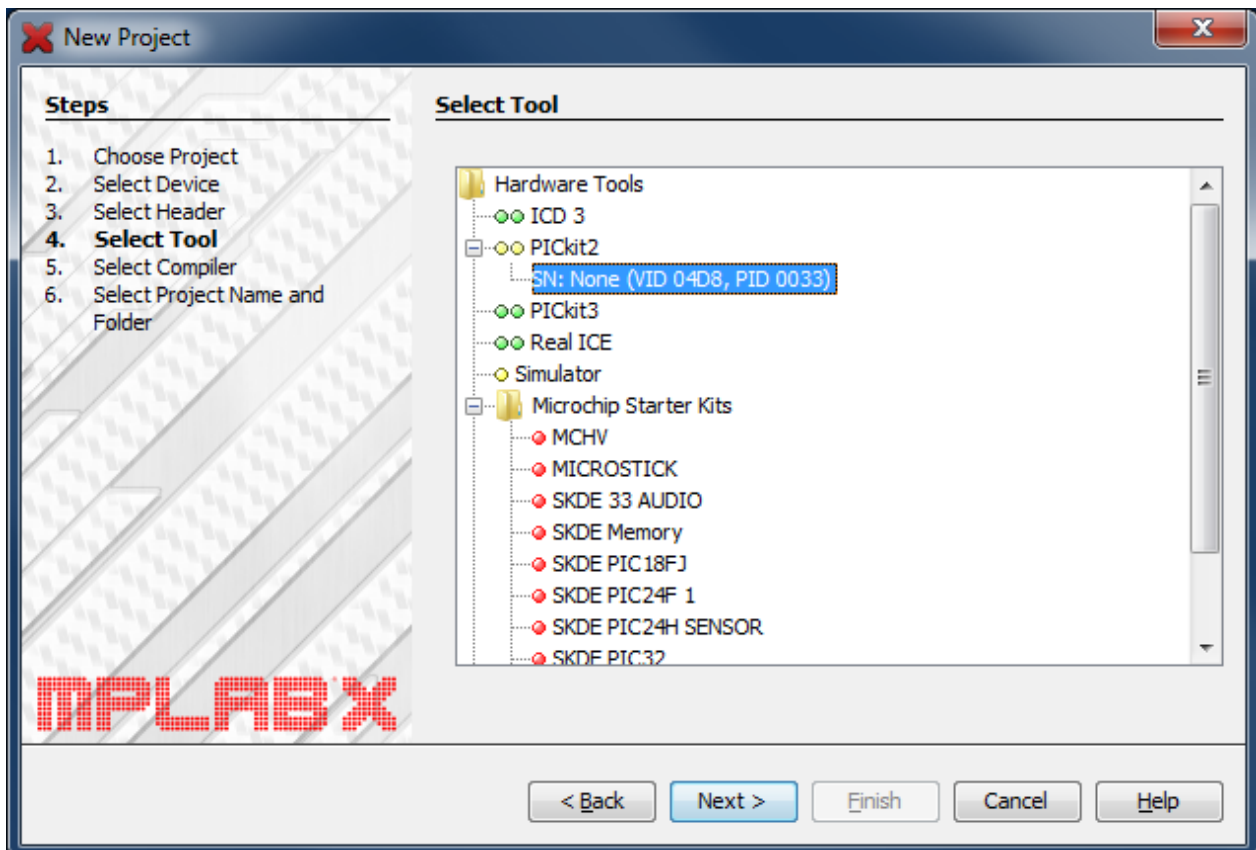
In this case, a PICkit 2 is connected to the PC.

If you have more than one programmer plugged in (including more than one of the same type, such as two PICkit 3s), they will all appear in this list, and you should select the specific one you intend to use for this project – you may need to check the serial number. Of course, you probably only have one programmer, so your selection will be easy.

---

[8] There is no need to install a special device driver for the PICkit 2 or PICkit 3; they work "out of the box".

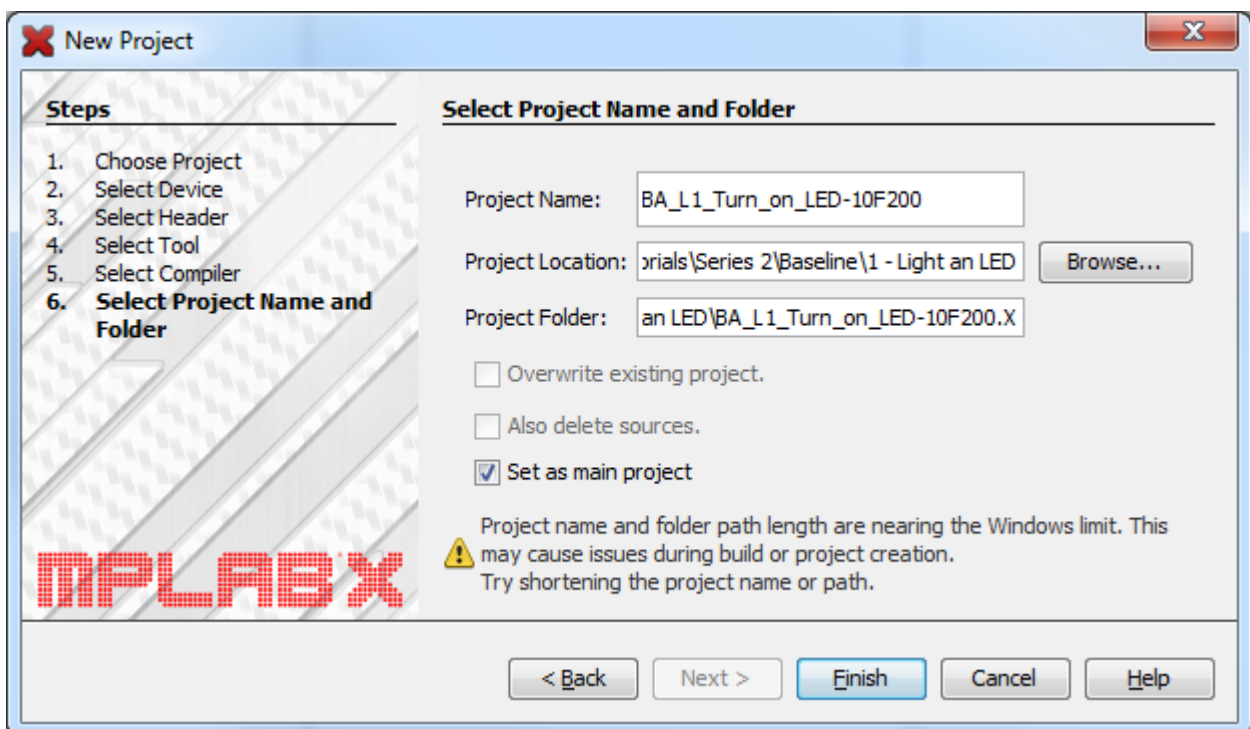After selecting the hardware tool, you select the compiler (or, in our case, assembler) you will be using:
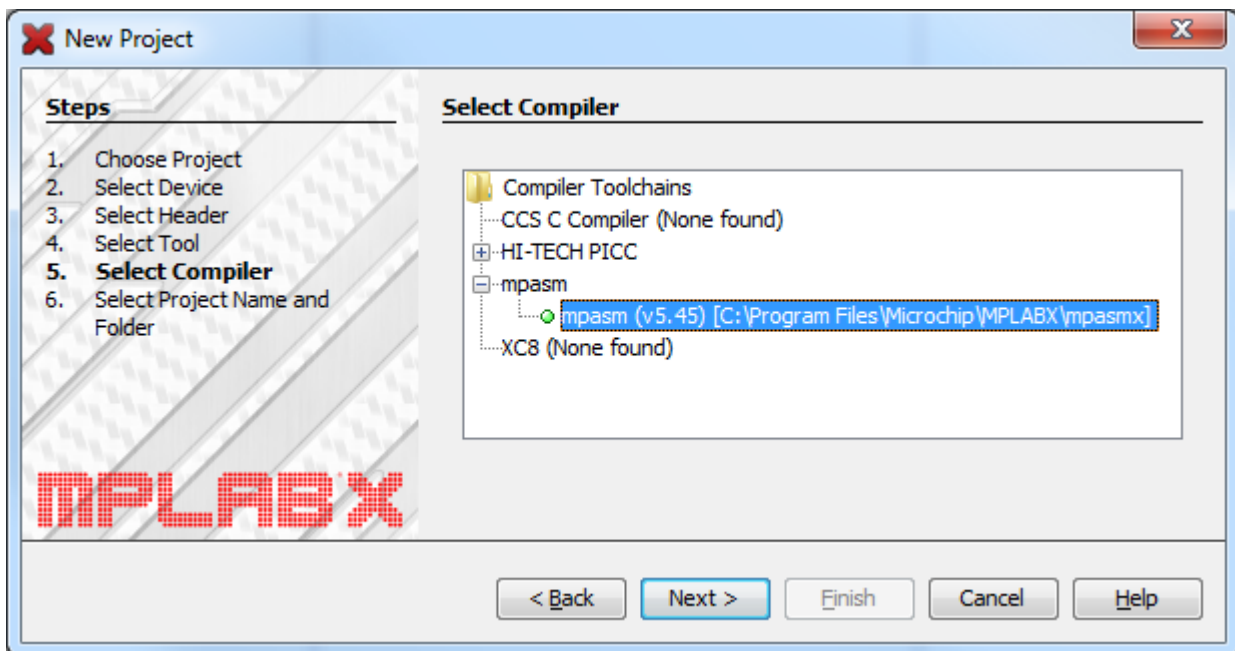
To specify that we will be programming in assembler, select the 'mpasm' option.

Finally, you need to specify your project's location, and give it a name:

MPLAB X creates a folder for its files, under the main project folder.

For example, in the environment used to develop these tutorials, all the files related to this lesson, including schematics and documentation, are placed in a folder named '1 - Light an LED', which is the "Project Location" given above. MPLAB X then creates a separate folder for the PIC source code and other files related to this project, in a subfolder that, by default, has the same name as the project, with a

'.X' on the end. If you wish, you can remove the '.X' extension from the project folder, before you click on 'Finish'.

Note the warning about project name and folder path length. To avoid possible problems, it's best to use shorter names and paths, when using Windows, although in this case it's actually ok.

Since this is the only project we're working on, it doesn't make much difference whether you select 'Set as main project'; this is something that is more useful when you are working with multiple projects.

After you click "Finish", your project will appear in the project window, and your workspace should look something like this:
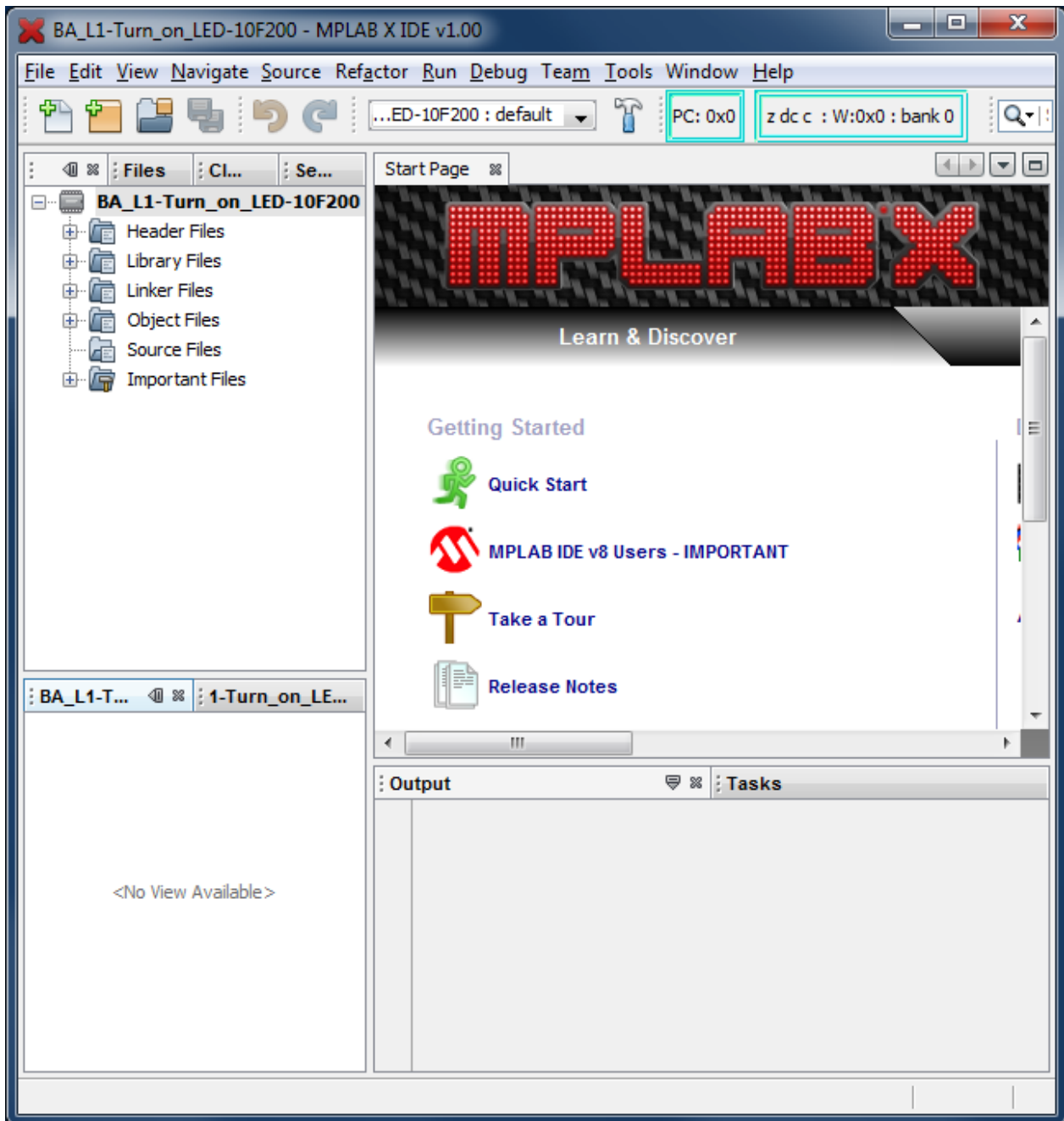
It is usually best to base your new program on an existing template (which could be a similar program that you developed earlier).

As explained in the MPLAB 8 instructions above, a set of templates for either absolute or relocatable code development is provided with MPASM. These are located under the 'templates' directory, within the MPASM installation directory, which, if you are using a 32-bit version of Windows, will normally be 'C:\Program Files\Microchip\MPLABX\mpasmx'[9]. As before, the templates for absolute code are found in the 'Code' directory, while those for relocatable code are found in the 'Object' directory.

We will be developing relocatable code, so you need to copy the appropriate template, such as '10F100TMPO.ASM' or '12F508TMPO.ASM', from the 'mpasmx\templates\Object' directory into the project folder created above, and give it a more meaningful name, such as 'BA_L1-Turn_on_LED-10F200.asm'.

For example, with the names and paths given in the illustration for step 6 of the New Project wizard above, you would copy:

    C:\Program Files\Microchip\MPLABX\mpasm\templates\Object\10F200TMPO.ASM

to

    C:\...\Baseline\1 – Light an LED\BA_L1-Turn_on_LED-10F200\ BA_L1-Turn_on_LED-10F200.asm

Of course, you'll put your project somewhere of your own choosing, and your MPASM installation may be somewhere else (especially if you are using Linux or a Mac), and if you're using a PIC12F508 you'd substitute '12F508' instead of '10F200', but this should give you an idea of what's needed.

Note that, unlike MPLAB 8, this copy step cannot be done from within the MPLAB X IDE. You need to use your operating system (Windows, Linux or Mac) to copy and rename the template file.

Now we need to tell MPASM X that this file is part of the project.

One way to do this is to right-click 'Source Files' in the project tree within the Projects window, and select 'Add Existing Item…':

---
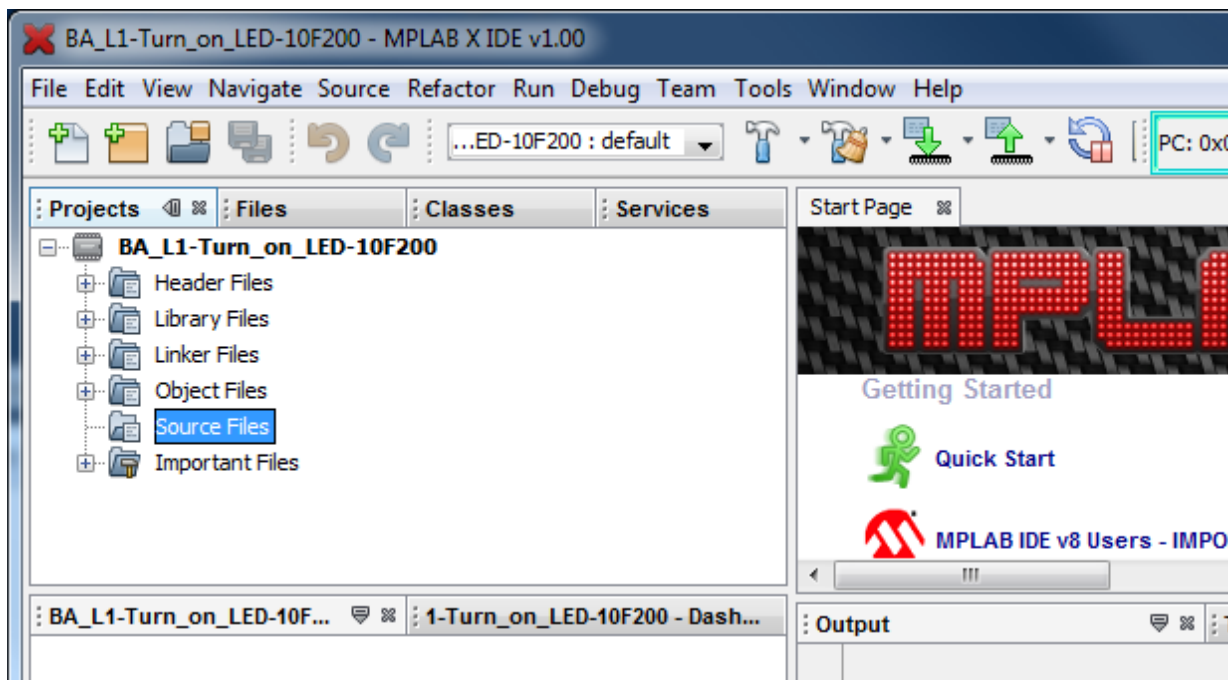
[9] On a 64-bit version of Windows, the MPASM folder will be under 'Program Files (x86)'

You can then select the template file that you copied (and renamed) into the project directory:
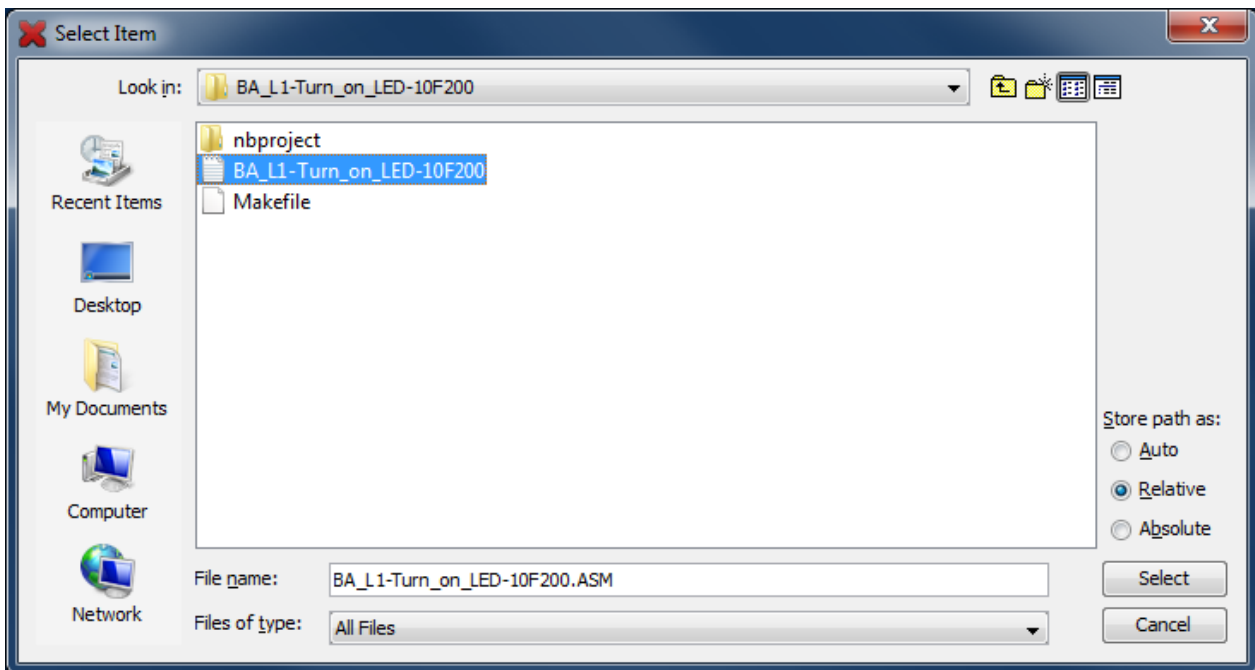
As with MPLAB 8, you need to tell MPLAB X whether the files you add are *relative* (they move with the project, if it is ever moved or copied) or *absolute* (remain in a fixed location). If you choose 'Auto', MPLAB X will guess. But since we know that this file is specific to your project, and should always move with it, you should select 'Relative' here.

Your .asm source file will now appear under 'Source Files' in the project tree.

If you double-click it, a text editor window will open, and you can finally start working on your code!
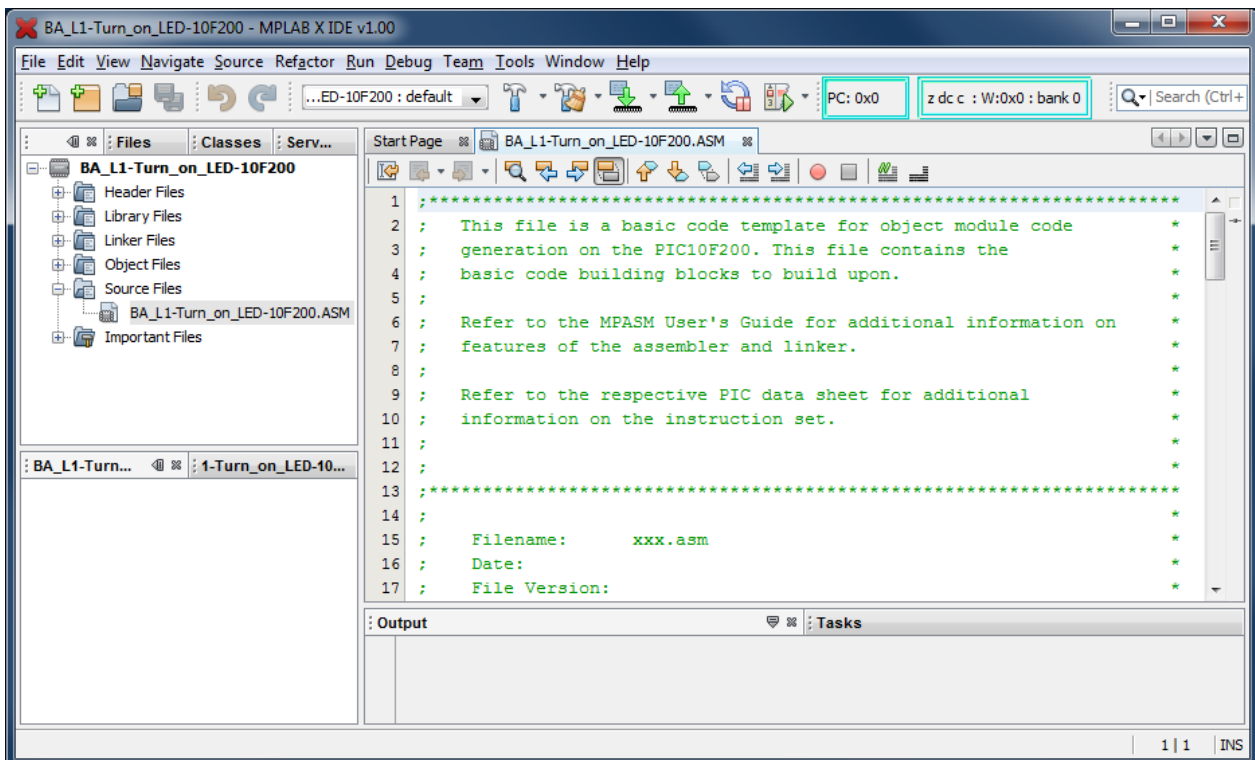
As in MPLAB 8, the text editor is aware of PIC assembler (MPASM) syntax and will colour-code the text. If you select 'Options' under the 'Tools' menu, you can set editor properties, such as tab size, but you'll find that the defaults are quite usable to start with.

Template Code

The first section of the template 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-10F200.asm                        *
;   Date:          2/1/12                                              *
;   File Version:  0.1                                                 *
;                                                                      *
;   Author:        David Meiklejohn                                    *
;   Company:       Gooligum Electronics                                *
;                                                                      *
;************************************************************************
;                                                                      *
;   Architecture:  Baseline PIC                                        *
;   Processor:     10F200                                              *
;                                                                      *
;************************************************************************
;                                                                      *
;   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=10F200            ; list directive to define processor
#include <p10F200.inc>         ; processor specific variable definitions
```

or, if you are using the PIC12F508, you would have:

```
list       p=12F508            ; list directive to define processor
#include <p12F508.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 8 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* ('p10F200.inc' or 'p12F508.inc', located in the MPASM install directory) to be read by the assembler. This file sets up aliases for all the features of the processor, 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*.

These two things – 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, in the 10F200 template:

```
__CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF
```

This sets the processor configuration. The 10F200 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_ON`
  Enables the external reset, or "master clear" ($\overline{\text{MCLR}}$) signal.

  If enabled, the processor will be reset if pin 8 is pulled low. If disabled, pin 8 can be used as an input: GP3. That's why, on the circuit diagram, pin 8 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 8 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 code protection.

  When your code is in production and you're selling PIC-based products, you may not want 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.

  Since we're not designing anything for sale, we'll make our lives easier by leaving code protection turned off.

- `_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 – see [lesson 7](#).

  The watchdog timer is very useful in production systems, but a nuisance when prototyping, so we'll leave it disabled.

The 12F508 template has a very similar `__CONFIG` directive:

```
     __CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _IntRC_OSC
```

The first three options are the same as before, but note that $\overline{\text{MCLR}}$ is on pin 4 on the 12F508, instead of pin 8. As on the 10F200, $\overline{\text{MCLR}}$ is shared with GP3. The LPC Demo Board also includes a pushbutton which will pull GP3/$\overline{\text{MCLR}}$ (pin 4) low when pressed. Unless we want to use the pushbutton on the LPC Demo Board as an input, it's best to leave it as a rest button, and enable external reset by specifying '`_MCLRE_ON`'.

The 12F508 `__CONFIG` directive also includes:

- `_IntRC_OSC`
  This 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.

  Most modern PICs, including the 10F200 and 12F508, 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.

  The internal RC oscillator on the 10F200 and 12F508 runs at approximately 4 MHz. Program instructions are processed at one quarter this speed: 1 MHz, or 1 μs per instruction.

  Most PICs, including the 12F508, support a number of clock options, including more accurate crystal oscillators, as we'll see in [lesson 7](#), but the 10F200 does not; it only has the internal RC oscillator, which is why this wasn't part of the 10F200's `__CONFIG` directive.

  To turn on an LED, we don't need accurate timing, so we'll stick with the internal RC oscillator, and include '`_IntRC_OSC`' in the 12F508's `__CONFIG` directive.

The comments following the `__CONFIG` directive in the template can be deleted, but it is a good idea to use comments to document the configuration settings.

For example:

```
;***** CONFIGURATION
                ; ext reset, no code protect, no watchdog, int RC clock
    __CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF & _IntRC_OSC
```

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 the 10F200 template code introduces our first instruction:

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

; Internal RC calibration value is placed at location 0xFF 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 0xFF after CODE is an address in hexadecimal (signified in MPASM by the '0x' prefix). Program memory on the 10F200 extends from 000h to 0FFh. This CODE directive is telling the linker to place the section of code that follows it at 0x0FF – the very top of the 10F200'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 10F200 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 (0x0FF). 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, 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 10F200 is powered on or reset, the first instruction it executes is this movlw instruction at address 0x0FF. After executing this instruction, the W register will hold the factory-set calibration value. And after executing an instruction at 0x0FF, there is no more program memory[10], so the *program counter*, which points to the next instruction to be executed, "wraps around" to point at the start of memory: 0x000.

0x0FF is the 10F200's true "reset vector" (where code starts running after a reset), but 0x000 is the effective reset vector, where you place the start of your own code.

Confused?

---

[10] The 10F200 has only 256 words of program memory, from address 000h to 0FFh.

What it really boils down to is that, when the 10F200 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.

This "RESET_VECTOR" code section, as presented in the template, is not really very useful (other than comments telling you what you should already know from reading the data sheet), because it doesn't actually stop your program overwriting the calibration value. Sure, it's unlikely that your program would completely fill available memory, but to be absolutely sure, we can use the RES directive to reserve the address at the top of program memory:

```
;***** RC CALIBRATION
RCCAL   CODE   0x0FF        ; processor reset vector
        res 1               ; holds internal RC cal value, as a movlw k
```

Now, even if our program grows to fill all available program memory, the linker won't allow us to overwrite the calibration value, because it's been reserved.

The corresponding code section for the 12F508 version is much the same:

```
;***** RC CALIBRATION
RCCAL   CODE   0x1FF        ; processor reset vector
        res 1               ; holds internal RC cal value, as a movlw k
```

The only difference is that, because the 12F508 has 512 words of program memory, extending from address 000h to 1FFh, its calibration instruction is located at 0x1FF.

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.

Changing the labels and comments a little (for consistency with later lessons), we have:

```
;***** RESET VECTOR **************************************************
RESET   CODE   0x000        ; effective reset vector
        movwf   OSCCAL       ; apply internal RC factory calibration
```

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.

If this seems complicated (and unfortunately, it is!), at least the worst is over. We can finally start the main part of the program, and focus on actually implementing the application.

The final piece 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'. But it does make it easier to follow the code.

'nop' is a "**n**o **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 a baseline PIC is powered on, all pins are configured by default as inputs, and the content 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 10F200 or 12F508, 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.

This little program, although small, has a structure common to most PIC programs: an initialisation section, where the I/O pins and other facilities are configured and initialised, followed by a "main loop", which repeats forever. Although we'll add to it in future lessons, we'll always keep this basic structure of initialisation code followed by a main loop.

Complete program

Putting together all the above, here's the complete assembler source needed for turning on an LED, for the PIC10F200:

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

    list        p=10F200
    #include    <p10F200.inc>


;***** CONFIGURATION
                ; ext reset, no code protect, no watchdog
    __CONFIG    _MCLRE_ON & _CP_OFF & _WDT_OFF
```

```
;***** RC CALIBRATION
RCCAL   CODE    0x0FF           ; processor reset vector
        res 1                   ; holds internal RC cal value, as a movlw k


;***** RESET VECTOR *************************************************
RESET   CODE    0x000           ; effective reset vector
        movwf   OSCCAL          ; apply internal RC factory calibration


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

;***** Initialisation
start
        movlw   b'111101'       ; configure GP1 (only) as an output
        tris    GPIO
        movlw   b'000010'       ; set GP1 high
        movwf   GPIO

;***** Main loop
        goto    $               ; loop forever


        END
```

The 12F508 version is very similar, with changes to the `list`, `#include`, `__CONFIG` and `RCCAL CODE` directives, as noted earlier.

That's it! Not a lot of code, really…

Building the Application and Programming the PIC

Now that we have the complete assembler source, we can build the final application code and program it into the PIC.

This is done in two steps:

- Build the project
- Use a programmer to load the program code into the PIC

The first step, building the project, involves assembling the source files[11] to create object files, and linking these object files, to build the executable code. Normally this is transparent; MPLAB does all of this 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.

A PIC programmer, such as the PICkit 2 or PICkit 3, is then used to upload the executable code into the PIC. Although a separate application is sometimes used for this "programming" process, it's convenient when developing code to do the programming step from within MPLAB, which is what we'll look at here.

Although the concepts are the same, the details of building your project and programming the PIC depend on the IDE you are using, so we'll look at how it's done in both MPLAB 8 and MPLAB X.

---

[11] Although there is only one source file in this simple example, larger projects often consist of multiple files; we'll see an example in lesson 3.

### MPLAB 8.xx

Building the project

When you build a project using MPLAB 8, it needs to know whether you will be using a hardware debugger to debug your code.

We won't be debugging, so select "Release" under the "Project → Build Configuration" menu item, or, more conveniently, select "Release" from the drop-down menu in the toolbar.

To build the project, select the "Project → Make" menu item, press F10, or click on the "Make" button in the toolbar:
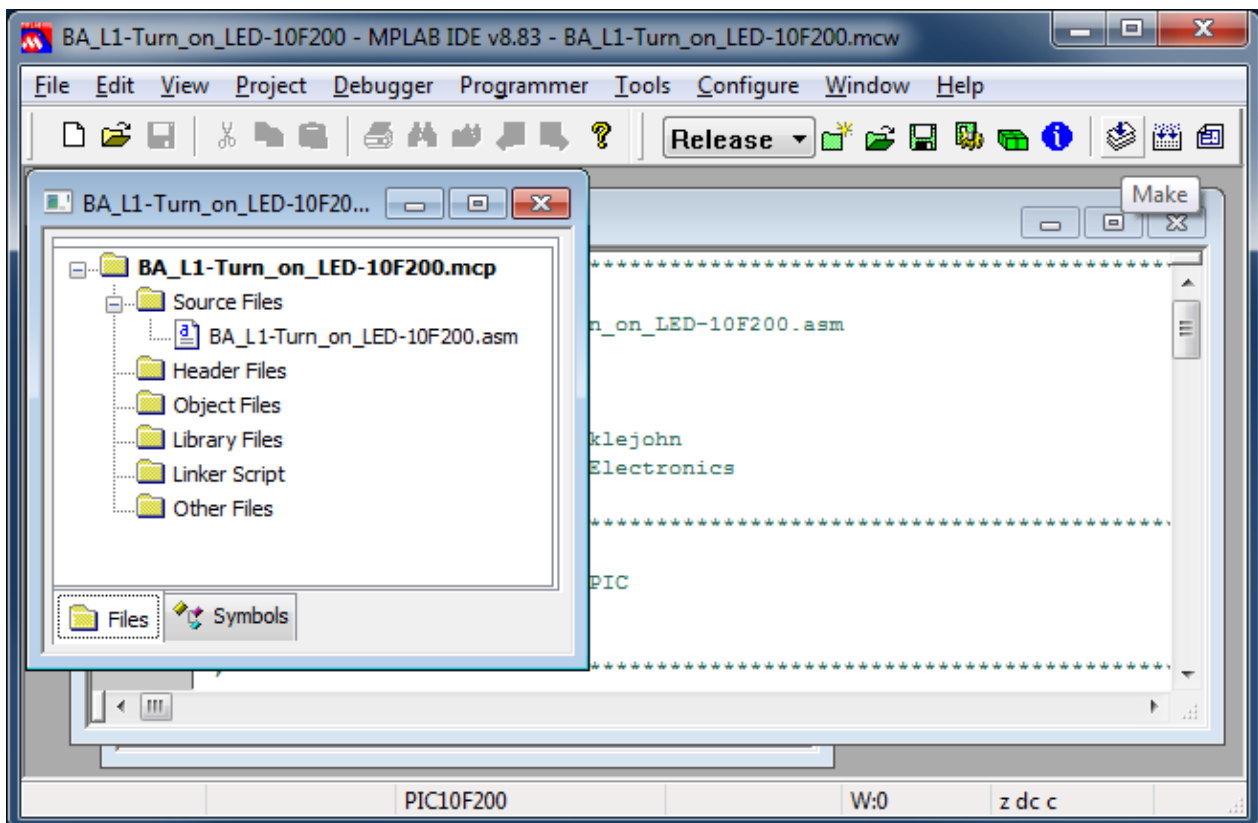
"Make" will assemble any source files which need assembling (ones which have changed since the last time the project was built), then link them together.

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 showing the code being assembled, similar to this:

```
Make: The target "C:\Work\Gooligum\Tutorials\Series 2\Baseline\1 - Light an LED\BA_L1-Turn_on_LED-
10F200.o" is out of date.
Executing: "C:\Program Files\Microchip\MPASM Suite\MPASMWIN.exe" /q /p10F200 "BA_L1-Turn_on_LED-
10F200.asm" /l"BA_L1-Turn_on_LED-10F200.lst" /e"BA_L1-Turn_on_LED-10F200.err" /o"BA_L1-Turn_on_LED-
10F200.o"
```

If you see any errors or warnings, you probably have a syntax error somewhere, so check your code against the listing above.

The next several lines show the code being linked:

```
Make: The target "C:\Work\Gooligum\Tutorials\Series 2\Baseline\1 - Light an LED\BA_L1-Turn_on_LED-
10F200.cof" is out of date.
Executing: "C:\Program Files\Microchip\MPASM Suite\mplink.exe" /p10F200 "BA_L1-Turn_on_LED-10F200.o"
/z__MPLAB_BUILD=1 /o"BA_L1-Turn_on_LED-10F200.cof" /M"BA_L1-Turn_on_LED-10F200.map" /W
MPLINK 4.41, Linker
Device Database Version 1.5
Copyright (c) 1998-2011 Microchip Technology Inc.
Errors    : 0

MP2HEX 4.41, COFF to HEX File Converter
Copyright (c) 1998-2011 Microchip Technology Inc.
Errors    : 0

Loaded C:\Work\Gooligum\Tutorials\Series 2\Baseline\1 - Light an LED\BA_L1-Turn_on_LED-10F200.cof.
```

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

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

First, ensure that you have connected you PICkit 2 or PICkit 3 programmer to your Gooligum training board or Microchip LPC Demo Board, with the PIC correctly installed in the appropriate IC socket[12], and that the programmer is plugged into your PC.

You can now select your programmer from the "Programmer → Select Programmer" menu item.

If you are using a PICkit 2, you should see messages in a "PICkit 2" tab in the Output window, similar to these:

```
Initializing PICkit 2 version 0.0.3.63
Found PICkit 2 - Operating System Version 2.32.0
Target power not detected - Powering from PICkit 2 ( 5.00V)
PICkit 2 Ready
```

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.

---

[12] Or, in general, that the PIC you wish to program is connected to whichever programmer or debugger you are using, whether it's in a demo/development/training board, a production board, or a standalone programmer.

If you are using a PICkit 3, you may see a message telling you new firmware must be downloaded.

This is because, unlike the PICkit 2, the PICkit 3 uses different firmware to support each PIC device family, such as baseline or mid-range.

You'll only see this prompt when you change to a new device family.

If you do see it, just click 'OK'.
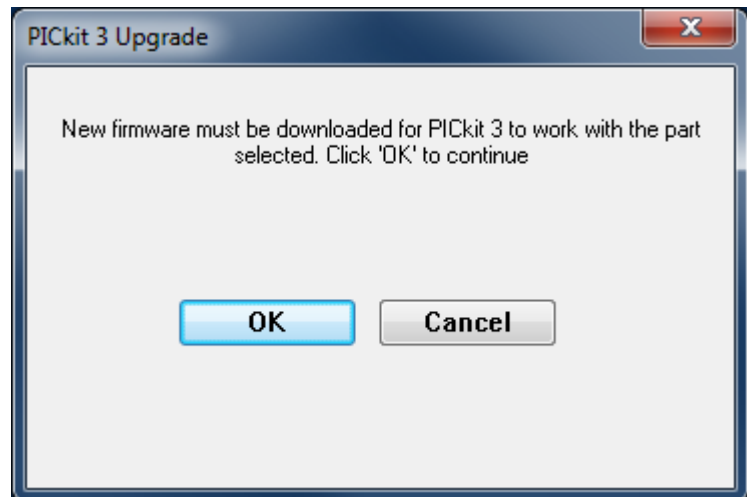
You may also see a voltage caution warning, as shown on the right.

This is because many newer PIC devices, including those with 'LF' in their part number, do not support 5 V operation, and can be damaged if connected to a 5 V supply.

The PICkit 3 will supply 5 V if you select a 5 V device, such as the 10F200 or 12F508.

We are using a 5 V device, so you can click 'OK'.
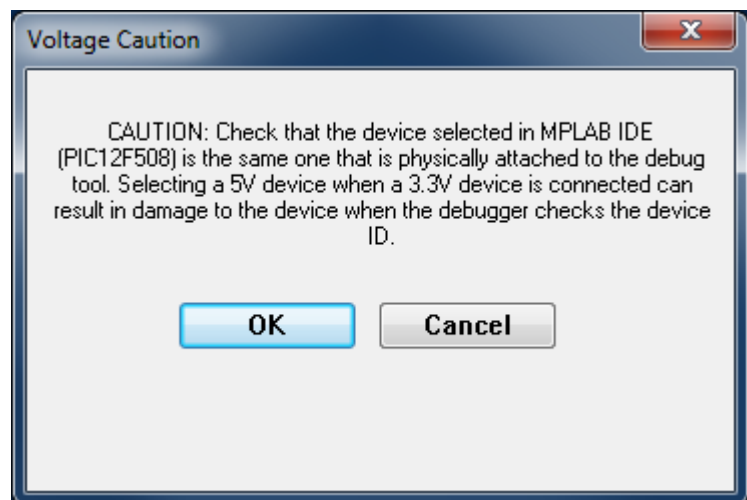
You may now see an error message in the output window, such as:

    PK3Err0045: You must connect to a target device to use PICkit 3.

This is because, despite the previous warning about power, the PICkit 3 will not actually supply power to our circuit (necessary to program the PIC, unless you have separate power supply), until you tell it to do so.

Select the "Programmer → Settings" menu item. This will open the PICkit 3 Settings window. In the "Power" tab, select "Power target circuit from PICkit 3", as shown:
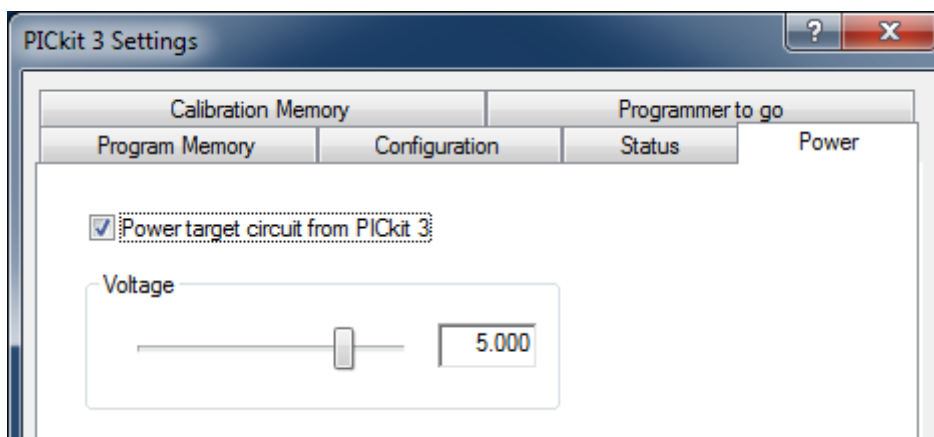
You can leave the voltage set to 5.0 V, and then click 'OK'.

You'll get the voltage caution again, so click 'OK' again.

If you now choose the "Programmer → Reconnect" menu item, you should see messages in a "PICkit 3" tab in the Output window (after yet another voltage caution), similar to these:

```
PICkit 3 detected
Connecting to PICkit 3...
Firmware Suite Version...... 01.26.92
Firmware type.....................Baseline
PICkit 3 Connected.
```

It does seem that the PICkit 3 is a little fiddlier to work with, than the PICkit 2…

After you select your programmer, an additional toolbar will appear.

For the PICkit 2, it looks like:

For the PICkit 3, we have:

As you can see, they are very similar.

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

```
Programming Target (5/01/2012  12:41:49 PM)
Erasing Target
Programming Program Memory (0x0 - 0x7)
Verifying Program Memory (0x0 - 0x7)
Programming Configuration Memory
Verifying Configuration Memory
PICkit 2 Ready
```

Or, if you are using a PICkit 3, simply:

```
Programming...
Programming/Verify complete
```

Your PIC is now programmed!

If you are using a PICkit 3, the LED on GP1 should immediately light up.

If you have a PICkit 2, you won't see anything yet. That is because, by default, the PICkit 2 holds the $\overline{\text{MCLR}}$ line low after programming. Since we have used the _MCLRE_ON configuration 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!

MPLAB X

Building the project

Before you build your project using MPLAB X, you should first ensure that it is the "main" project. It should be highlighted in bold in the Projects window.

To set the project you want to work on (and build) as the main project, you should right-click it and select "Set as Main Project". If you happen to have more than one project in your project window, you can by removing any project you are not actively working on (to reduce the chance of confusion) from the Projects window, by right-clicking it and selecting "Close".

To build the project, right-click it in the Projects window and select "Build", or select the "Run → Build Main Project" menu item, or simply click on the "Build Main Project" button (looks like a hammer) in the toolbar:
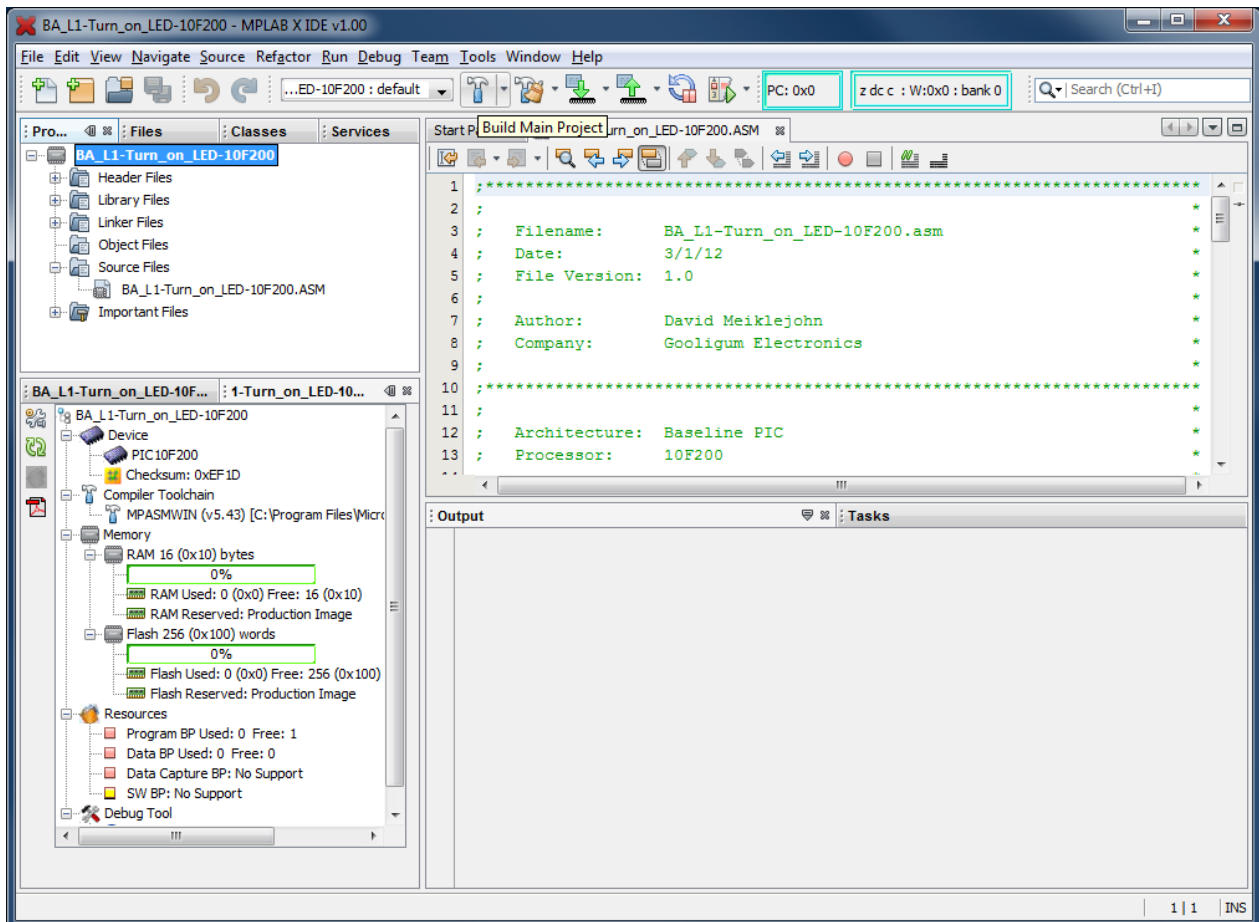
This will assemble any source files which have changed since the project was last built, and link them.

An alternative is "Clean and Build", which removes any assembled (object) files and then re-assembles all files, regardless of whether they have been changed. This action is available by right-clicking in the Projects window, or under the "Run" menu, or by clicking on the "Clean and Build Main Project" button (looks like a hammer with a brush) in the toolbar.

When you build the project, you'll see messages in the Output window, showing your source files being assembled and linked. At the end, you should see:

    BUILD SUCCESSFUL (total time: 2s)

(of course, your total time will probably be different…)

If, instead, you see an error message, you'll need to check your code and your project configuration.

Programming the PIC

The final step is to upload the executable code into the PIC.

First, ensure that you have connected you PICkit 2 or PICkit 3 programmer to your Gooligum training board or Microchip LPC Demo Board, with the PIC correctly installed in the appropriate IC socket, and that the programmer is plugged into your PC.

If you have been following this lesson, you will have specified the programmer when you created your project (in step 4 of the wizard).

If you want to check that the correct programmer is selected, or if you want to change your tool selection, you can right-click your project in the Projects window and select "Properties", or simply click on the "Project Properties" button on the left side of the Project Dashboard:
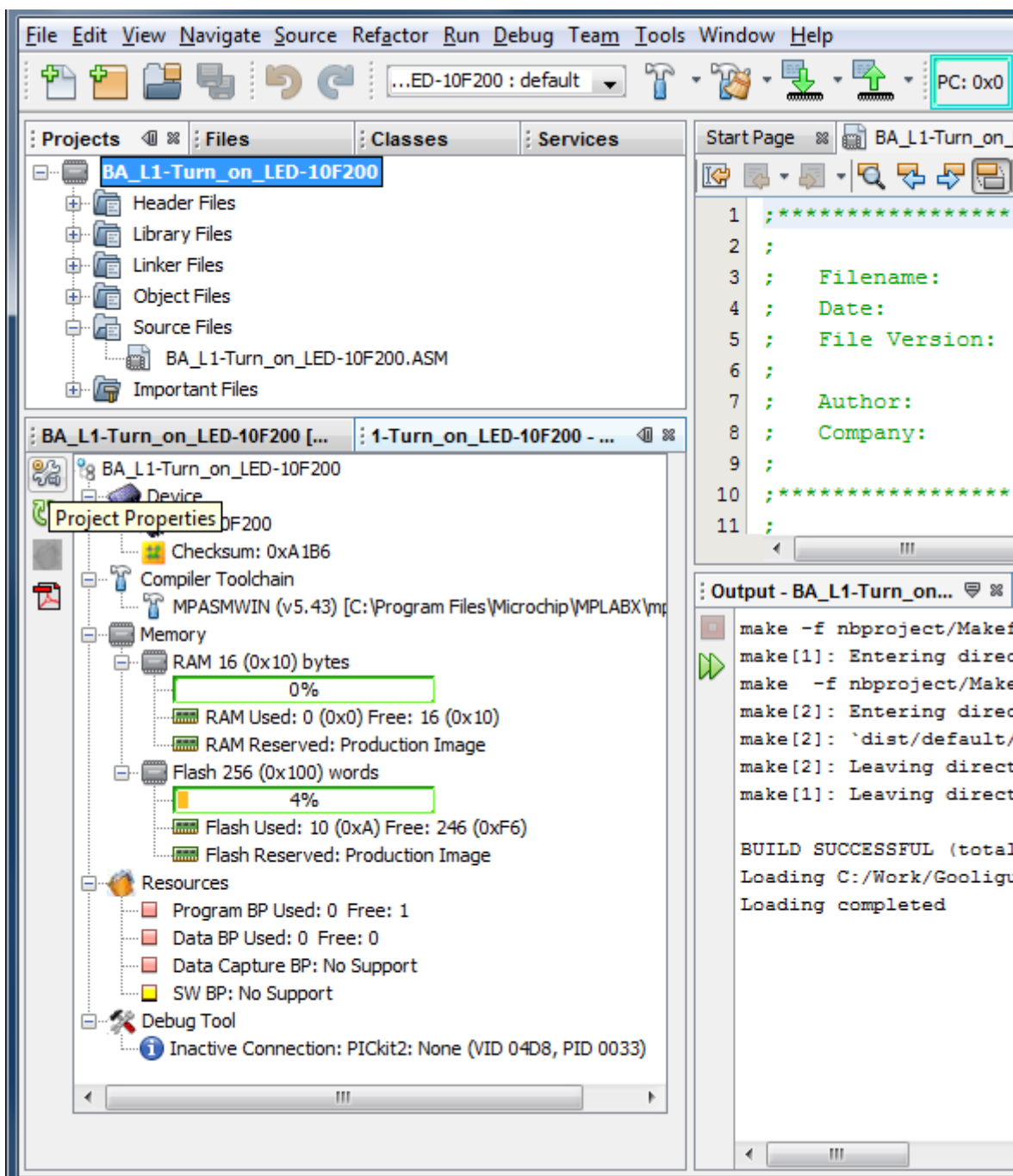
This will open the project properties window, where you can verify or change your hardware tool (programmer) selection:
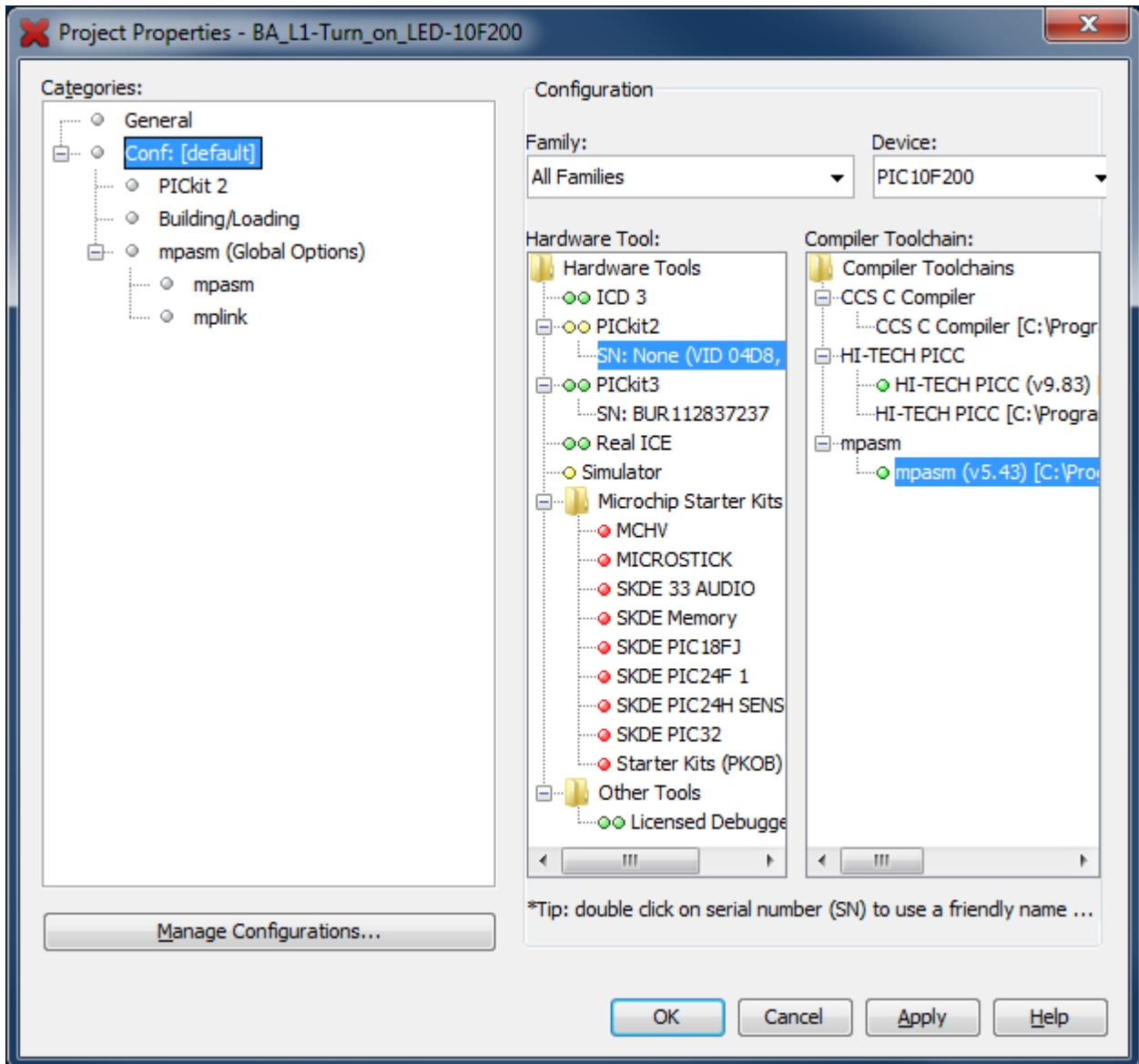
After closing the project properties window, you can now program the PIC.

You can do this by right-clicking your project in the Projects window, and select "Make and Program Device". This will repeat the project build, which we did earlier, but because nothing has changed (we have not edited the code), the "make" command will see that there is nothing to do, and the assembler will not run.

Instead, you should see output like:

```
BUILD SUCCESSFUL (total time: 266ms)
Loading C:/Work/Gooligum/Tutorials/Series 2/Baseline/1 - Light an LED/BA_L1-Turn_on_LED-
10F200/dist/default/production/BA_L1-Turn_on_LED-10F200.production.hex...
Loading completed
Connecting to programmer...
Programming target...
```

(the total time is much smaller than before, because no assembly had to be done).

If you are using a PICkit 3, MPLAB X will download new firmware into it, and you will see messages in the PICkit 3 output window like:

    Downloading Firmware...
    Downloading AP...
    AP download complete
    Firmware Suite Version.....01.27.20
    Firmware type..............Baseline

You may also see a voltage caution warning, as shown below:
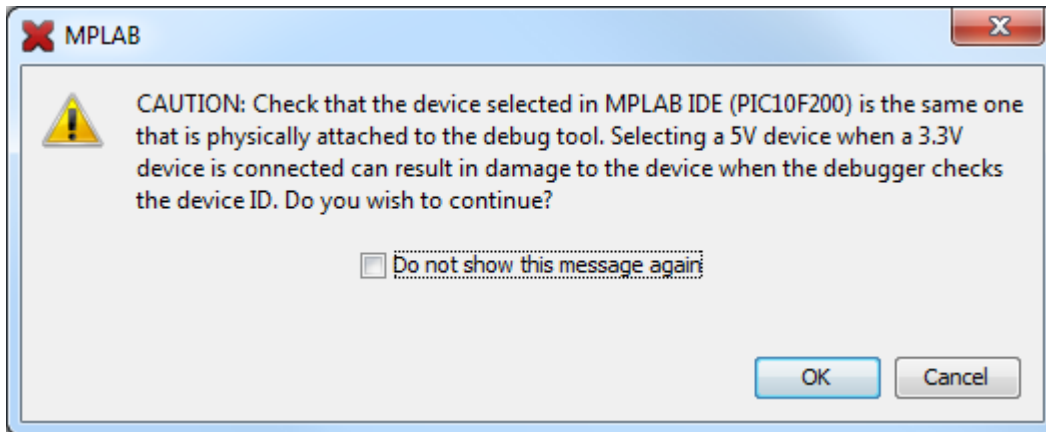
Since we are using a 5 V device, you can click 'OK'. And feel free to click "Do not show this message again", to avoid seeing this caution every time you program your PIC.

You may now see an error message in the PICkit 3 output window, stating:

    Target device was not found. You must connect to a target device to use PICkit 3.

This happens if the PIC is unpowered, so we need to tell the PICkit 3 to supply power.

Open the project properties window (as on the previous page), select 'PICkit 3' in the categories tree, and choose the 'Power' option in the drop-down categories list:
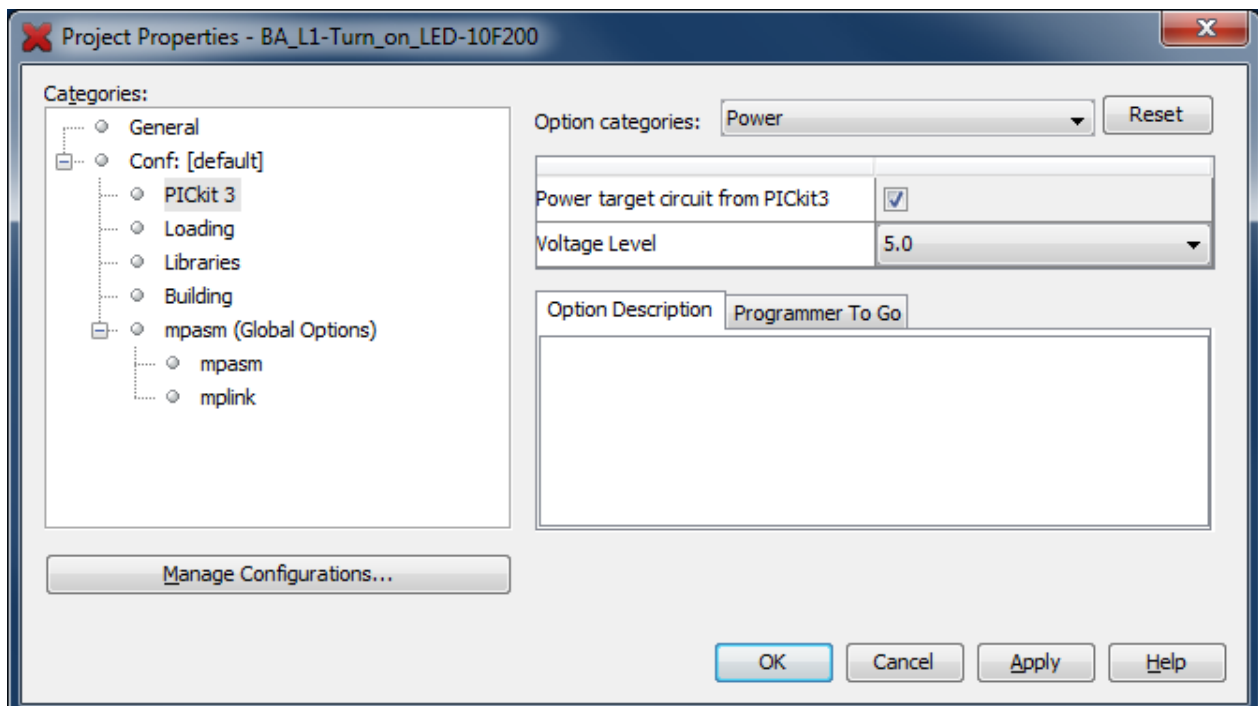
Select "Power target circuit from PICkit3", as shown. You can leave the voltage set to 5.0 V, and then click 'OK'.

If you now perform "Make and Program Device" again, the programming should be successful and you should see, in the build output window, messages ending in:

    Programming completed

Note that this action combines making (or building) the project, with programming the PIC. What were two steps in MPLAB 8 are combined into one step here. In fact, with MPLAB X, there is no straightforward way to simply program the PIC, without building your project as well.

This makes sense, because you will almost always want to program your PIC with the latest code. If you make a change in the editor, you want to program that change into the PIC. With MPLAB X, you can be sure that whatever code you see in your editor window is what will be programmed into the PIC.

But most times, you'll want to go a step further, and run your program, after uploading it into the PIC, to see if it works. For that reason, MPLAB X makes it very easy to build your code, program it into your PIC, and then run it, all in a single operation.

There are a few ways to do this:

- 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".

Summary

The sections above, on building your project and programming the PIC, have made using MPLAB X seem much more complicated than it really is.

Certainly, there are a lot of options and ways of doing things, but in practice it's very simple.

Most of the time, you will be working with a single project, and only one hardware tool, such as a programmer or debugger, which you will have selected when you first ran the New Project wizard.

In that case (and most times, it will be), just press 'F6' or click on to build, program and run your code – all in a single, easy step.

That's all there is to it. Use the New Project wizard to create your project, add a template file to base your code on, use the editor to edit your code, and then press 'F6'.

Conclusion

For such a simple task as lighting an LED, this has been a very long lesson!

In summary, we:

- Introduced two baseline PICs:
    o 10F200
    o 12F508
- Showed how to configure and use the PIC's output pins
- Implemented an example circuit using two development boards:
    o Gooligum training and development board
    o Microchip Low Pin Count Demo Board
- Looked at Microchip's assembly template code and saw:
    o some PIC assembler directives
    o some PIC configuration options
    o our first few PIC instructions
- Modified it to create our (very simple!) PIC program
- Introduced two development environments:
    o MPLAB 8.xx
    o MPLAB X
- Showed how to use these development environments to:
    o Create a new project
    o Include existing template code
    o Modify that template code
    o Build the program
    o Program the PIC, using:
        ▪ PICkit 2
        ▪ PICkit 3
    o Run the program

That is a lot, to accomplish so little – although you can of course ignore the sections that aren't relevant to your environment. You should use MPLAB 8 if it's still available, supported, and works with your PC, in which case you can ignore the MPLAB X sections for now, and come back to them if you upgrade later.

If you have the Gooligum training board, you can ignore sections about the Microchip LPC Demo board. And if you're lucky enough to have a PICkit 2, you can ignore the sections about the PICkit 3.

Nevertheless, after all this, you have a solid base to build on. You have a working development environment. You can create projects, modify your code, load (program) your code into your PIC, and make it run.

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

That first step is the hardest. From this point, we build on what's come before.

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