Knowledge of baseline

1A - Light Up an LED - Assembly

What we want to achieve

The purpose of the tutorial is to turn on an LED through the microcontroller.

It may, at first glance, seem completely trivial or senseless to practice on this topic, given that to turn on the LED it is enough to connect it between Vdd and Vss.
But this is not the case, because:

Even just to achieve this simple result, you need to act on the microcontroller with a real program.

For those who want to have a confirmation, try to make the diagram proposed in the following pages and insert a new, unprogrammed PIC on the socket: by giving voltage, the LED will NOT light up.

This is easily explained by the fact that the chip's pins are, by default, configured as inputs or with functions other than digital output. To change this setting, you need to use a program: since no sequence of instructions is written in the memory of the component to change the defaults, the pin cannot change function and supply current. Only with a program, no matter how simple, will it be possible to turn on the LED.
And, for those who are starting out, what is important is precisely the learning of what is necessary for this action of "programming" and the manoeuvres that it requires.
For the beginner, they can be more difficult than the drafting of the program itself, **since in this simple operation are involved all the elements that are essential to obtain a microcontroller circuit working according to your program, namely:**

- **the writing of the source, with the related Assembly rules**
- **the use of the MPLAB development environment and its MPASM Assembler**
- **Chip programming**
- **the necessary hardware connections**

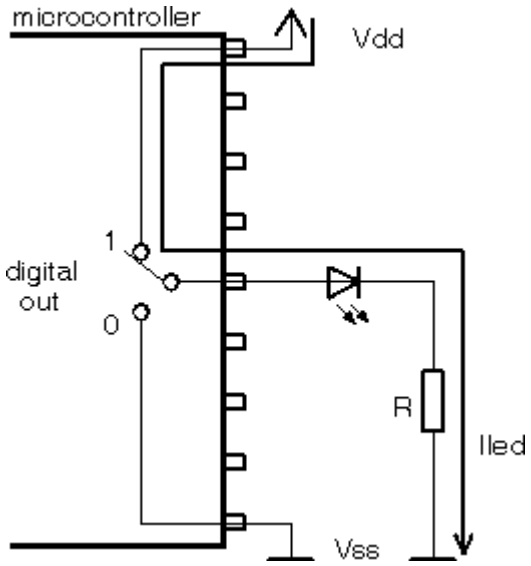
This first exercise, therefore, aims to begin to understand:

- **How to Write the Source of an Assembly Program**
- **how to compile it in the MPLAB environment**
- **How to program the chip with the compile result**

These are the minimum steps to get any response from the microcontroller and they will repeat,
This is an extremely complicated or absolutely simple program like this.

How to turn on the LED

To turn on the LED, we take advantage **of the digital output function** that can be attributed to many pins of the microcontroller. When we set a pin in this way, we enable an internal buffer through the program that can deliver or absorb a certain current on the pin.



From an electrical point of view, the buffer of each pin is the electrical equivalent of a diverter, as schematized in the image on the side:

- By bringing the pin control bit to level 1, i.e. to a voltage close to Vdd, the digital output, through the buffer, which works as a switch, will feed the load consisting of the LED connected between the pin and the Vss (GND or ground, negative pole of the power supply).
- By turning the control bit to 0, i.e. to a voltage close to Vss, the LED will be off.

The current in the LED is limited by the required resistance in the R-series.

In reality, these are circuits made in CMOS technology, where MOSFET transistors act as switches, controlled by the state of the bits of the control registers.

More information on the internal pin structure of microcontrollers can be found here.
We have a "direct" logic:

Logic Level	Voltage at pin	LED
0	Vss	off
1	Vdd	on

However, we can also connect the LED between the pin and the Vdd: in this case, to turn on the LED it will be necessary to send the cathode to a low level, programming the bit relative to the pin to 0.

The choice of one connection rather than the other will depend on the circuit needs and is considered completely indifferent for both the microcontroller and the program.

In our case we connect the LED to the Vss as shown in the diagram, so the LED lighting is activated by the following actions:

- set the pin that controls the LED as the digital output
- Raise the pin level to 1

For more details, it is necessary to know that, inside the microcontroller, one register defines the "direction" attributed to the pin, which can be input or output, while another register manages the logic level applied to the pin.

Each pin, therefore, has as its corresponding:

- **a direction control bit; We can set the PIN as input or output**
- **a level control bit as output; we can program the pin at high level (about Vdd) or low level (about Vss).**

The first control bit is located in a so-called "direction" register, which Microchip calls, albeit incorrectly, the three-state register (**TRIS**), since in input mode the pin assumes a high-impedance level, typical of three-state situations.

The second bit will be part of another register, called **PORT** or **GPIO** (**General** *Purpose* **I/O**).

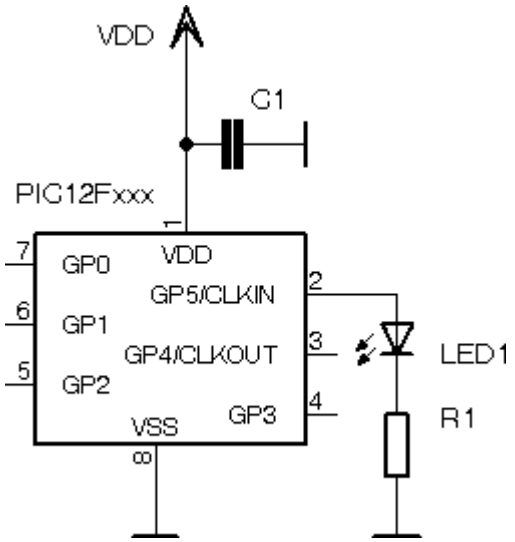
These registers collect in an orderly manner a certain number of bits/pins, generally in modules of 8, since this is the width of the data bus of the processors we are considering (PICs are **RISCs**in **Harvard architecture**, where the data and address buses are different and separate).
These registers must be modified by the program through the appropriate instructions that we will see immediately.

This is possible in a practically identical way with **any PIC**, but, to begin with, we use components of the **Baseline family**: this is because these are the 8-bit processors with the greatest possible structural simplicity and therefore with the minimum difficulties of approach, which allows you to begin to understand the philosophy on which the PICs are based, This is essential for tackling more complex microcontrollers later on, without encountering excessive difficulties.

We use a cheap component, in an 8-pin DIP package, like the **PIC12F519,** but any PIC will do, with the appropriate adaptations to the source - here the variations for 12F508/509/510 and 10F200/202 will be proposed.

If you need it, here you will find a short guide that describes the salient features of the **12F519** and that we recommend reading beforehand to get an overview of the component. Obviously, the data sheet and the documentation indicated in the links complete the description of what is needed to use the chip correctly.

Find the same for 12F508/509, 12F510, 10F200/2/4/6, 10F220/2.



The wiring diagram on the side shows the hardware needed for a generic PIC12Fxxx: **all of these, in 8-pin DIP packages, have the same pinout, so the scheme is fine for any of them.**

C1 is a 100nF (ceramic multilayer, 50V) capacitor located between pins 1 and 8, as close to the chip as possible. Its purpose is to decouple the power supply. This capacitor is not a useless "extra", but it is an indispensable element for the stability and guarantee of operation of the microcontroller and should not be omitted. If anything, it can take on higher values or be associated with an electrolyte element in case the power source is far from the chip.

The LED is connected between the **GP5 pin** of the microcontroller (pin 2 of the 8-pin package) and the ground (Vss). The LED turn-on current will be provided by the pin (*current source*), with a maximum of **25 mA**. A series resistor to the LED is required to limit this current to about 10 mA or so.
Low current LEDs (1-2mA) can be used with a series resistor of 1K or more; with less sensitive diodes (5-20mA) values from 330 to 820 ohms will be used.
As for **the calculation of resistance**, you can find some information here.

A few words about LEDs

A few words for those who don't use the development board, but build the circuit on breadboards are not an "extra", as we tend to gloss over superficially topics that seem trivial.

It should be noted that, even if each LED has its own typical current, indicated on its data sheet, this does not oblige you to circulate that current: it will probably be possible, for example in an LED with a nominal 20 mA, to pass only 5 or 10mA to obtain more than enough brightness. The less current we circulate in the chip's pins, the better its working condition will be.
So, in the absence of other indications, we can use, with a Vdd of 5V, typically any resistance between 470 ohms and 1k.
Obviously, it will NOT be possible to omit the resistor: if we did, we would damage both the LED and the PIC due to the excessive current.


In general, it is advisable to use as little power as possible, both for energy savings, which allows battery-powered devices to operate for a longer time, and to stress the components as little as possible and ensure their longer life.

For now, however, the value of the resistance is not fundamental, which only affects the current in the LED and certainly not the operation of the program we are going to write. The only important thing is not to exceed the maximum current that can be delivered by the pin.

The Clock

The internal operations of the microcontroller are cadenced by a square wave clock signal, with a frequency that can reach several tens of MHz, depending on the type of chip.
Typically, in 8-bit PICs, each operation (instruction) requires 4 clock strokes, so we can distinguish two basic frequencies:

- the frequency of the ***Fosc oscillator***
- the frequency of the instructions, which is ***Fosc/4***

This means that, for example, with a clock of 4MHz, the instructions have a frequency of 1/4, i.e. 1MHz and therefore their duration (period) is 1us. Thus, if the clock rises to 20MHz, the instruction period, i.e. the time taken to execute it, is 200ns.

In the Baselines we are going to use, most of the instructions take only one cycle to execute, and only a few take two cycles; So we can say that the microcontroller has the ability to carry out a million instructions in one second.

Although it is common to observe the presence of external oscillator systems, typically made with a quartz and a pair of capacitors, **the microcontrollers we use do not require any external element to generate the clock**, since this is produced by an internal oscillator.

The typical frequency value of this oscillator is 4MHz, but depending on the chip, you may have different options.

Properly calibrated, it generates a frequency that is sufficiently stable with temperature and accurate as a value (typ. 1%) for most applications and, at the same time, allows the use of all pins in the small package.

Hardware

No external objects are required outside of a power source, which can be made with a power supply.

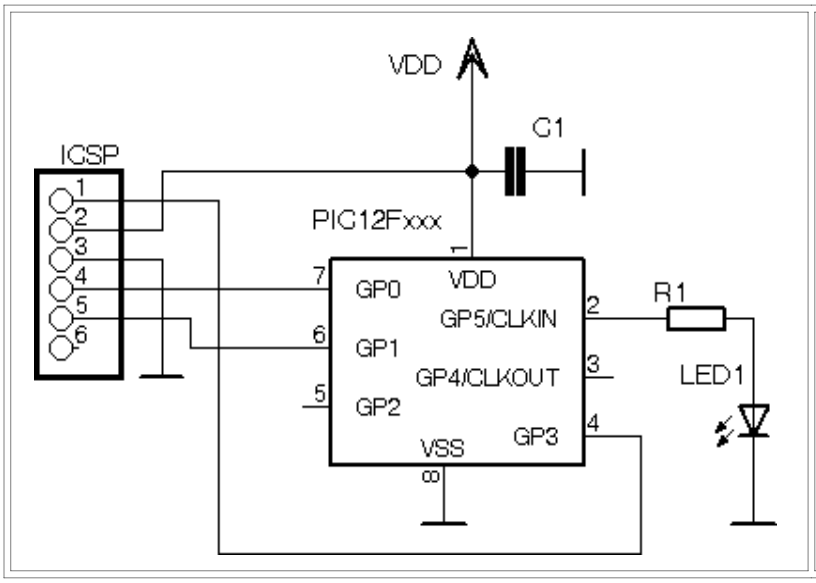
 In practice, however, using a PICkit and an ICSP connection you won't even need any power supply system, since we will derive the current needed for the circuit from the PICkit itself, which in turn takes it from the USB port with which it is connected to the PC.
We will see later how to set the Pickit to power the connected circuits.

As a result, an external power supply is not strictly necessary.

In fact, we also have to consider the need to program the PIC: for this **we do NOT need** any programming device, since it is possible to carry out the operation without removing the chip from the circuit through **an ICSP connection**.

It is a common 2.54 pitch plug connector, 6 poles, into which the PICkit that we will use for programming can be inserted directly.
In the picture we can see the position of pin 1 clearly indicated.

The wiring diagram is completed as follows:



ICSP	Signal	Pin	Label pin
1	**Vpp**	4	GP3
2	**Vdd**	1	VDD
3	**Vss**	8	VSS
4	**ICSPDAT**	7	GP0
5	**ICSPCLK**	6	GP1
6	n.c.	-	-

With this connection we can not only **program the chip on-board**, through the serial lines that are called **ICSPDAT/ICSPCLK** (or also **PGD/PGC**), in addition to the Vpp programming voltage, which is applied to the MCLR pin.

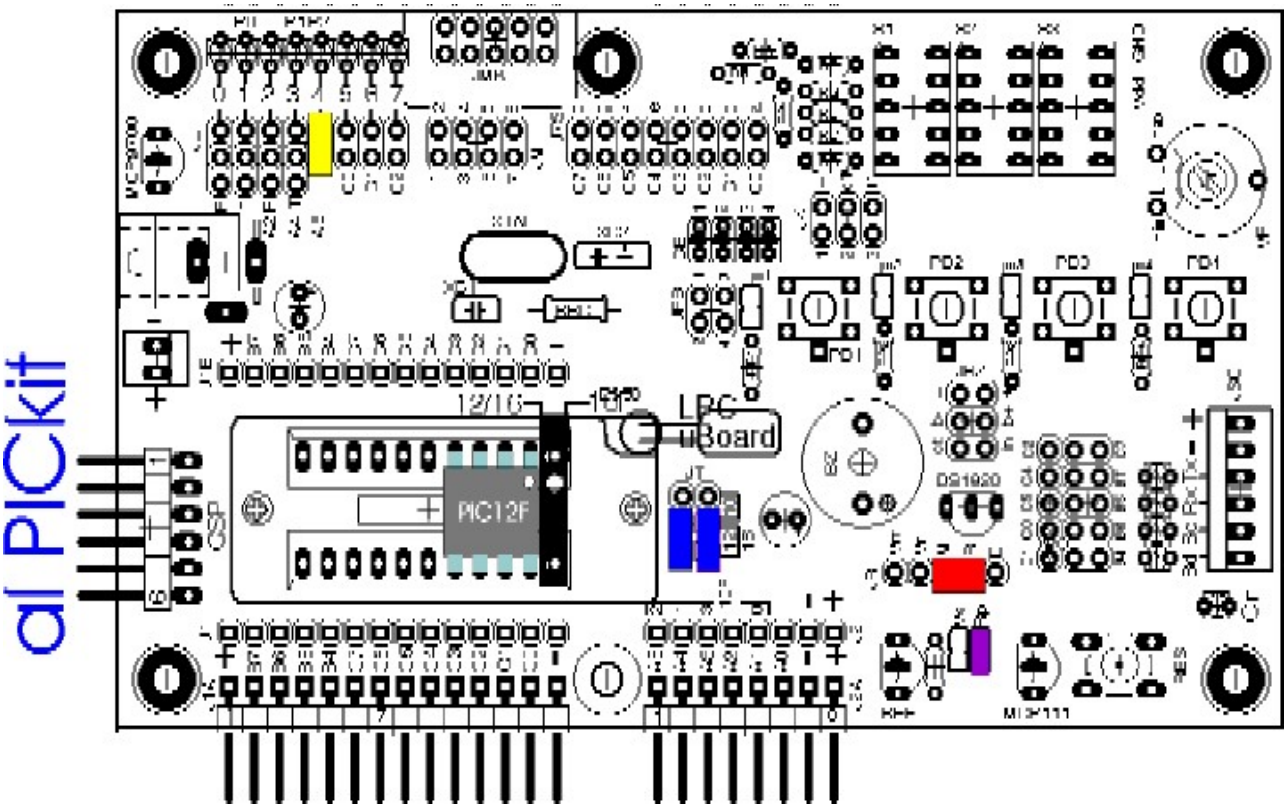
Note that the programming tools and the circuit have the ground **Vss** in common, but that the **ICSP connection** also contains the **Vdd**, in order to be able to power the connected chip during programming. This is possible with the **Pickit 3** and **2**.

By taking advantage of this option we can also **power the entire test board** , within the limit of about ten mA, which is enough for most of the simple circuits used in the exercises.

It should also be noted that, using a chip with a built-in **ICD** engine or a suitable header, we can also perform **on-circuit debugging** directly on the hardware.

An ICSP/ICD socket is usually already present on demo boards, while if we use a breadboard the necessary connections must be made once and for all, since this basic scheme will be used in most of the other first exercises without too many modifications.

If we use our **LPCmicroBoard** , we don't need to do anything special, as the ICSP socket is integrated. Here you will find the simple connections to be made, which are reduced to just four jumpers.



- JL "yellow" **jumper** inserted on C0 connection to connect **`GP5`** with LED **`4`**
- "**Blue**" JT **jumpers** inserted downwards to select a non-10F MPU
- "red" **MR jumper** to connect the `RES line`
- "**purple**" RP **jumper** to connect the pull-up to the **`RES line`**

Note that the RES function of the MCLR pin is not used in this first tutorial.

You don't need anyone else. Obviously, the color of the jumpers is of no other importance than to identify at first glance the groups divided by function; That is, elements of any color you have available can be used, although it is preferable to maintain a certain "standard" to facilitate the work.

The last two connections in the above list are not functional to the tutorial, but only to the PICkit if used as an **ICD** debugger, an important function that we will see in the Midrange tutorials, since Baselines on have an internal debugging engine. For your information, it is still possible to emulate ICD debugging for this chip as well, using the header AC162050.

A view of the real board, where the orange arrows indicate the jumpers to be inserted:

An important detail is the positioning of pin 1 of the chip with respect to the socket: pins 1 and 22 of the socket must be left free, since they are reserved for the **PIC10Fxxx**.
In this sense, the correct **ICSP** connections are already provided by the board and all you have to do is select the type of microcontroller used with the **JT** jumpers ("blue").

You need to have the **PICkit2** or **3** and the **MPLAB** development environment, which will need to be installed correctly. The **PICkit** must be connected to a USB socket on the computer capable of delivering at least the nominal 150mA required by the minimum standard.

Once the MPLAB **window is open**, you can **enable the PICkit to supply power to the circuit** and verify its operation by lighting up the **On LED** on the board.

The circuit can work with a wide range of voltages, from 3 to 5.5V. The difference in the applied voltage is only reflected in the brightness of the LEDs, whose current depends on the ratio between the voltage itself and the limiting resistance placed in series. So it is possible to power the circuit with two or three AA batteries or a small wall plug power supply, without exceeding 5V in any case.

In particular, low-current (typ. 1-5mA) and high-brightness LEDs are used in the **LPCmicroBoard**. It will still be possible to use any LED up to a current of 25mA (which is the maximum that can be delivered by the pins of the microcontroller). With voltages between 2.5 and 3.3V the circuit still works, but the brightness of the LED will be reduced.

Once you have made this selection, you will observe the On LED on the **LPCmicroBoard** light up. The LED is located at the bottom right, near the **JSC** connector.
This confirms that the PICkit is powering the circuit.

As far as the microcontroller is concerned, it must be said that:

With PIC12F519 you need to use PICkit3 (which is also good for all the others).

Those who have the PICkit2 for 8-pin chips will have to use a 12F508/509.

The 12F519 is an improved version of **the 12F509**, with the addition of an EEPROM area and an internal clock that can be set to 4 or 8 MHz. For the rest, the resources are similar and it is possible to move the source from one to the other with minimal modifications. A program like this that will be described can be run on ALL PICs, from 8 to 32 bits. However, it is necessary to vary the source according to the chosen chip and this requires some knowledge that you may not have now.

To begin to see how easily you can adjust a source from one PIC to another, sources are provided for 12F508/509/510/519 and also for 10F20x, which are significant enough for the Baseline family.

What we need to do

A first step in the implementation of any program is to **be clear about what you want to achieve** , namely:

- **turn on the LED connected to the GP5 as soon as the voltage is applied and keep it lit as long as the voltage is present**

and how to:

- **programming the GP5 pin as a digital output and bringing it to a high level**

For those who have doubts, it is necessary to remember that the pins of the microcontroller can be configured as:

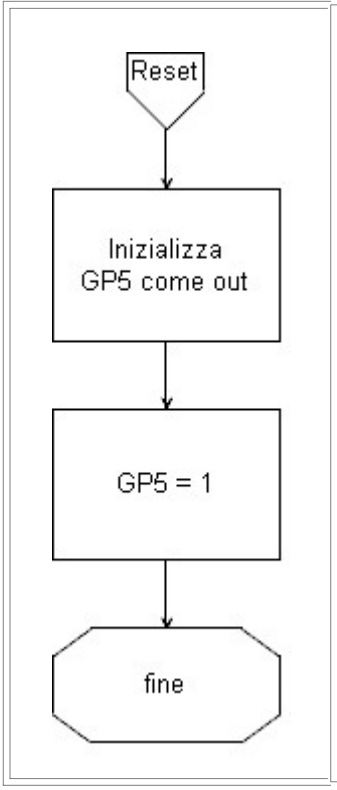
- **Digital Inputs**
- **Digital Outputs**
- **Other functions dependent on the microcontroller model**

In order to deliver (or absorb) current to the pin, it must be set as a **digital output** and the desired voltage level must be applied, as indicated at the beginning.

Therefore, the sequence of typical operations at the beginning of the program will be:

- Take control of the chip's resources and adapt them to the needs of the application (initial configuration)
- Establish the functions you want to apply to the pins used by the application
- Determine the direction of the pins that you use as digital I/O
- Set the logic levels at these pins to achieve the desired outputs.

Once this has been established, it will be possible to draw a flow *chart* that shows in graphic form what will be performed by the program.



In this case, given the simplicity of the operation, the flow chart will also be very simple.

One might think that it is not important, but if it is now a small thing, it can become the key that allows the programmer to resolve situations that would otherwise be too complex for a purely mental analysis action.

In fact, through the flow diagram, we graphically represent the logical organization of the program and the phases of its development. This allows you to both order ideas and verify the correctness of the applied logic and, during the creation of the flow of instructions, greatly facilitates the work.

Furthermore, during the debugging phase, by following the logical path graph you can easily find any defects and errors.
So, don't forget to draw a flowchart, even an approximate one: it could be what you need to formalize the idea and transform it into instructions.

Now we can start writing the source of the program, that is the series of lines that the compiler, the Assembler, will transform into a set of bytes that will be written to the processor's memory.

The source

First of all, let's say that:

the source is a simple text file, written with any editor, ideal if equipped with the text coloring function (Textpad, Notepad++, etc.). There is also such an Editor inside MPLAB.

So it's a .txt-type file, but not **.doc** or ***.htm*** files or files that are formatted and contain characters other than simple printable ASCII. So Notepad is fine, but not Word.

In practice, we can also use the editor of the MPLAB environment for the drafting of the source, but in practice it is better to separate the two, as the development environment is not necessary for this phase. The internal editor will come in handy for corrections during development.

It should be noted that, compared to a simple writing program such as Notepad, specific editors for programming have the characteristic of highlighting the different logical parts of the source with different colors. This feature, far from being just a gadget, is, on the contrary, a powerful aid for the programmer as it allows at first glance to distinguish the various components of

what he is writing.
On the basis of this consideration, the source examples given in these pages are colored in a similar way.

The template

The source text must contain the codes, commands and anything else needed for compilation.
We can divide it into a few parts:

- Header Description
- Choice of processor
- configuration
- definition of memory usage, possible labels, etc.
- Instruction flow from the reset vector
- Closing the build

The best way to learn how to proceed in an orderly manner is to do it through examples. And it is very convenient to start with the support of a **specific template** for the processor used.
This template is a "grid" that serves as a guide for us to draw up an orderly and effective program.

 It must be understood that giving a form to the source is not just an aesthetic question, but, much more, a way to make what we are writing readable. In this sense, everyone will then be able to develop their own "modus programmandi", but, initially, following a path is the best way to then be able to progress independently.

Here we use a very common basic form, adding a series of information that can significantly reduce the need to consult the component data sheet.

In our first example, the source is made available already fully written; This is to be an example of how to proceed independently. Despite this, it is necessary that we do not accept it as it is, but we make a detailed analysis of it, in order to be clear about all its components and the reason for the various choices.

Writing the source

You can check out **some pages here**.

We provide the ***1A_519.asm*** source file that is in the compressed package for the first exercise, for the PIC12F519. The source is complemented by the presence of the entire MPLAB project, which makes the work easier.
If you have PICkit2, which doesn't support 12F519, you need to use ***1A_5089.asm***, for PIC12F508 or **12F509**.
There is also the ***1A_20x.asm version*** for the **PIC10F200** and **10F202** and the version for the 14-pin chips **16F505** and **16F526** ***1A_526.asm***.

The advice is to print the source text and keep an eye on it while we analyze its composition.

The Header - Comment Lines

The initial part is, typically, a "hat" that contains general information about the theme of the program. Since these are comments, they will be printed in the ***.lst file***, but they do not become part of the compilation of the hexadecimal codes to be inserted into the chip's memory (***.hex file***).

The proposed template also includes a brief summary of the connections of the chip used, which facilitates the verification of the hardware regardless of the wiring diagram of the application, all in the form of comment lines.

```
;-----------------------------------------------------------------
; Impiego pin :
; ----------------
;       12F519 @ 8 pin
;
;           |--\/--|
;      Vdd -|1    8|- Vss
;      GP5 -|2    7|- GP0
;      GP4 -|3    6|- GP1
; GP3/MCLR -|4    5|- GP2
;           |______|
;
;            Vdd   1: ++
; GP5/OSC1/CLKIN   2: Out    LED towards the Vss
; GP4/OSC2         3:
; GP3/!MCLR/VPP    4:
; GP2/T0CKI        5:
; GP1/ICSPCLK      6:
; GP0/ICSPDAT      7:
; Vss              8: --
;
;*****************************************************************
```

Even the "graphics" can only be derived from the minimum set of printable ASCII characters, as no other special characters or elements other than text characters can be present in the source text for any reason.

Of course, this doesn't HAVE to be the only way to write your source, but it's an example of how to write it in a functional and orderly way. The advice is to initially follow this form, in order to be able to derive your own, when the needs of a good source will be clear, even if only from a graphic and organizational point of view.

Please note:

 the PIC12Fxxx all have the same pinout, i.e. the same location of the functions with respect to the pins, including the power supply. So, on the same base, you can use any one.

In this example we can also use 12F508/12F509/12F510, etc.
The PICs in 14-pin and 20-pin packages are stackable for the first 8 pins on the DIP8, both in terms of functions and power supply. When using the PIC10F2xx it should be noted that their DIP8-pin package version has a different pin placement than the PIC12F.

Processor definition - LIST, #include and .inc files

If the descriptive comment part of the introduction is "optional", but important for the understanding of what you are writing, let's now enter the functional part of the source that begins with the description of the type of processor we intend to use.
These lines are not part of the actual Assembly language, but they are essential to communicate to the compiler (the Assembler) which resources the compiler's automatisms will have to refer to in order to assign the right addresses in the formation of the object.
The following two lines are **the typical beginning of any Assembly source for PICs**.

```
     LIST p=12F519                ; processor definition
    #include <p12F519.inc>
```

The first line enables the creation of a list (***.lst*** file) derived from the compilation with reference to the indicated processor. The second line "includes" the ***.inc file*** for the processor used in the compilation.

LIST

The Assembler directive enables the creation of a list (***.lst file***) derived from the compilation with reference to the indicated processor.

Although it stands out in all the examples, it is a command that is not at all known.

First of all, it is not an obligation, in the sense that the compilation of the object and the executable takes place anyway. It is commonly inserted because the list file is an important documentation, even if only for finding any problems.
Its syntax is simple:

sp	**LIST**	sp	**[opzione,… opzione]**

sp indicates one or more spaces or tabs, since the directive cannot start in the first column. Other spaces or tabs separate the various elements of the row.

That is, **LIST** has a number of options that mainly affect the formatting of the text of the ***.lst***. file, but are also substitutes for the **radix**, **processor**, **errorlevel**, **expand/noexpand directives**. They are selected a priori by default and must be modified where required. The possible options are as follows:

Opzione	Default	Funzione	Direttiva sostituita
b = nnn	8	Spaces used in tabulation	
c = nnn	132	Number of columns	
f = format	INHX8M	Formato file .hex INHX32, INHX8M, INHX8S	
mm = [on|off]	on	Memory map sort	
n = nnn	60	Number of lines per page	
p = nomeprocessore	Nessuno	Tipo di PIC	**processor**
r = radix	Hex	Z	**radix**
st = [on|off]	on	Sort of the symbol tb	
t = [on|off]	off	Truncation of lines	
w = [0|1|2]	0	Limit error messages	**errorlevel**
x = [on|off]	on	Macro expansion	**expand/noexpand**

You don't commonly see these parameters because the default ones for the page format are the most suitable; the most commonly used hex file output format is INHX8M and it is advisable not to limit the information messages of possible errors, while the expansion of the implied code to macros is useful in debugging.

For example,:

```
list p=12F519
```

it simply indicates that you want to generate the .lst file, that you are using the 12F519 processor and, by default:

- One tab equals 8 spaces
- One line is 132 characters (maximum possible for old needle lines) and 60 lines per page, with no truncated lines

- the ***.hex file*** will be in INHX8M format
- In the ***.lst file***, the alphabetical sorting will be carried out for the labels of the memory map and the table of symbols
- The default numbering will be HEX
- No limitation on error messages
- macro expansion, i.e. replacing the label with the implied instructions In any

case, you can change the defaults. Thus:

```
list p=12F519, r=dec, x=off
```

It's the same thing as:

```
list
processor
12F519 radix
dec noexpand
```

Note that the labels that identify the Assembler commands (directives or statements) are reserved, i.e. they cannot be used for any other purpose and are not case-sensitive. A note further on makes this clearer.

If we don't want a part of the text to appear, we'll insert the **NOLIST** directive, which blocks list generation until the compiler encounters a new **LIST directive**.

Note that the ***.lst list file*** and the object of the compilation are two very different things:

- The .lst ***list file*** is an ASCII text that reports, in the form of an ordered and formatted list, the result of the compilation. It includes a list of instructions, both as symbols and hexadecimal values, comments, as well as a list of the symbols used and their absolute value. It is useful as documentation, but mostly for reference during debugging. This is the formal description of what the object file contains.
- The .hex ***object file*** obtained from the compilation contains the twin codes that will be written to the chip's memory.

List files and object files can be obtained from compilation independently of each other. The deletion of sections of text in the list with the **NOLIST** option does not have any effect on the object: the generation of the list is suspended, but not the compilation of the object.

#include processorname.inc

This file, which is essential for compilation, is part of the MPASM suite: with a default installation of MPLAB 8.xx, the **.inc** files are located in the ***C:\Program Files\Microchip\MPASM Suite*** folder.

They are ready-made files, which do not need to be modified: all you need to do is include them in the source and the compiler will use it correctly.

These are simple ASCII texts that contain the references of the main elements of the registers and control elements of a specific chip, relating absolute values (memory addresses, bit position, etc.) with labels, in such a way as to be able to write sources that can be relocated or, in any case, in a symbolic way, regardless of the presence of absolute values.

 It is strongly recommended to edit the .inc file used and at least take a look at it, to understand its contents. More info here.

This ***.inc*** file is "included" in the source with the Assembler command **#include**. This is an MPASM *statement* or **directive**, i.e. an internal compiler command that can be invoked in the source.

#include

 **The #include directive causes the compiler to act as if we had written the entire file instead of the command line.**
#include allows the file to reside somewhere else, such as being part of a library, and not require it to be rewritten each time.

Its syntax is simple:

sp	#include	sp	Full path of the file to be included	sp	[; comment]

The file to be included must be indicated with its full path; with the exception of MPASM's own files, such as **.inc** or ***.lkr files*** for which the Macro Assembler already has the "coordinates".

The directive is very important because, in practice, it greatly simplifies the programmer's work by allowing the use of predefined blocks of text as elements of a modular construction.

If we omit this line, the result will be a massive series of error messages from the compiler not finding the symbolic references related to the microcontroller being used.
If these are missing (`compiler error [113]`) the .hex ***file will not be generated*** necessary for the programming of the chip.

Each chip has its own *.inc* file, so you'll need to include the one for the component you're using.

If we use **12F509**, the change will result in the processor being defined differently:

```
      LIST p=12F509            ; processor definition
     #include <p12F509.inc>
```

Similarly, if we use **16F506**:

```
     LIST p=16F506            ; processor definition
     #include <p16F506.inc>
```

and if we use **10F202**:

```
     LIST p=10F202            ; processor definition
     include <p10F202.inc>
```

And so it is with every other processor. These lines will have to be modified in relation to the chip to be used, of which they include the appropriate compilation elements in the source.

As we will see later, it is also possible to write a source suitable for multiple processors.

Incidentally, we observe that each line includes only one directive; you can't do two on the same line.

Find out more about [MPASM guidelines here]

Build Environment Definitions - radix

Following the source listing, we still find a couple of definitions that better determine the working environment. These are two optional directives, which are not essential, but they are useful. The first:

```
          radix     dec
```

RADIX

This directive declares that the compilation will default to numbers based on a certain root; In our example, this is decimal numbering, so all numbers expressed without a specific indication will be considered as decimal.

 Possible alternatives:

- **dec** decimal
- Hexadecimal HEX (default)
- **Octal OCT**

The syntax is simple:

sp	Radix	sp	Object **dec/hex/oct**	sp	[; comment]

Like most directives, it cannot start in the first column and requires one of the following objects.

As we have seen, it is replaceable by an option of the LIST, but we leave it highlighted separately, since the writing:

```
list p=12F519, r=dec
```

it is not commonly used in the examples accessible from the WEB.

 **radix is not a mandatory directive, i.e. it can easily be omitted.**
Just know that, by default, MPASM considers hexadecimal (hex) numbering, that is, it considers all numbers that are not expressed with a precise indication as hexadecimal. Where we want numbers with different roots, it will be enough to indicate them each time, according to the conventions indicated here.

However, it can be useful to have defined a specific root other than hexadecimal, and decimal is the common choice, since we tend to consider numbers in this base.
If you need more information about the numbering systems used in this **area ,you can find it here.**

It is also important to keep in mind that:

 **The compiler cannot correct a missing or incorrect definition of the radix of a number; it can only execute what you indicated in the source.**

So if you indicate that you want to use a hexadecimal root, the compiler will do your setting, but if you then use the numbers as if they were in decimal root, without specifying it, the compiler will certainly not be able to make corrections.
Therefore, particular attention should be paid to those who, coming from environments such as BASIC, have a tendency to use decimal numbers in all circumstances, without particular attention.
In the microcontroller environment, the thoughtful use of hexadecimal, binary, and decimal numbers makes the programmer's job easier, but a minimum of care is required. Therefore:

 regarding numerical values in the source, we strongly recommend always using the indicated signs to clearly state the way in which that given number is expressed, even if a general root has been established

This avoids distracting mistakes.

Case Sensitive


The MPASM Assembler is case sensitive, i.e. it is case sensitive. In these conditions, for example, the names imposed by the source (label) will be considered different depending on the characters used; so label is different from LABEL or LAbel. This is the default setting, but you can change it to make it case insensitive.

It is recommended to use a case sensitive environment, as it allows you to have a much greater number of possibilities for names (labels).

On the other hand, the names reserved by the Assembler, i.e. those of the directives, opcodes, pseudo opcodes, internal labels, etc. They can be written in both uppercase and lowercase letters. So, for example, #INCLUDE will be equivalent to #include and LIST can be written list and so on. The choice of a mode is a matter of personal taste, although it may make sense to use upper and lower case to identify particular elements of the source.

Comments, i.e. the parts of text between ; and end of line, can be written as you wish as they do not become part of the compilation.

The Configuration - Directive __Config

Having exhausted these premises, which concern the definition of the compilation environment, it is still necessary to consider a **fundamental fact**: the microcontroller has a set of instructions, a certain amount of memory, etc., **but it also has a basic hardware structure that must be prepared to be able to do what we want**.

This basic structure must be "configured" according to the needs of the project, BEFORE starting the execution of the instructions.
A comparison could be given, for example, to the need to choose the type of tires of the

depending on whether it is winter or summer and this, of course, must be done before starting the journey. In our case:

 **the initial configuration establishes how the microcontroller hardware support is set up.** This is necessary because the internal resources of the microcontroller can be allocated to a virtually unlimited number of applications and the manufacturer ensures that a certain component is as flexible as possible to adapt to all circumstances.

Based on this, the following options will be determined, among other options:

- the clock system used (internal or external, in the various possible modes)
- Activating the Watchdog
- the use of the MCLR pin

It is clear that if we do not prepare the chip to use a certain oscillator, provided in the hardware connections, it will not work; In the same way, if we activate the watchdog without then providing for its management in the program, it will work incorrectly and randomly.
It is therefore necessary to prepare the internal environment of the microcontroller from the start, defining what is necessary for what we want it to do. This adaptation operation is included in the source of the program, but does not make the instruction wall; it is called **CONFIGURATION**, sometimes reported (quite incorrectly) with the name of "***fuses***".
The implementation of what we define in this configuration consists in having varied the level of some bits in one or more special registers, called ***Configuration Word***

```
;*****************************************************************
;                              CONFIGURATION
;
 __config _CP_OFF & _MCLRE_OFF & _IntRC_OSC & _IOFSCS_4MHZ & _MCPU_ON &
_WDT_OFF
```

__CONFIG

This is done with the command__config of the Assembler, which causes the options indicated below to be filled in with the correct binary values and placed in the memory of the ***Configuration Word*** while writing the chip.

The syntax is always simple:

sp	__config	sp	**AND Chain of Configuration Labels**

We note that:

- The directive cannot start in the first column
- **Start with a double ___**
- **All options should be written in the same line, without a "line break",** even if its length can become considerable.

The label chain is expressed as an **AND** (**&**) function of the chosen options. These options (**_CP_OFF**, **_IntRC_OSC**, etc.) are listed in the aforementioned ***processorname.inc*** file.
In our case, it's the p12f519.inc file, which we still invite you to
View with any editor (view, but don't edit!). Each corresponds to a hexadecimal value.
The result of the ANDs is in turn a hexadecimal value, which the compiler will place with the appropriate references in the ***.hex*** file, so that the programming device can write it in the right position.

This configuration line shape is specific to the Baseline and Midrange. For the PIC18F, microchips have provided a different shape.

For further clarification, **the configuration is not part of the program's instructions and is not made up of parts that will be executed: its purpose is to fix the desired conditions for that application in the processor hardware.**

Since it is written in flash memory (program memory area), it can only be deleted and modified by reprogramming the chip.

You can find **more pages** about the initial setup here.

For the sake of completeness, we report the "***Configuration Bits***" section of the ***p12f519.inc file***, where we see what is described above:

```
;==========================================================================
;       Configuration Bits
;
;   NAME            Address
;   CONFIG             FFFh
;==========================================================================
; The following is an assignment of address values for all of the
; configuration registers for the purpose of table reads
_CONFIG               EQU H'FFF'

;.....CONFIG Options ..................................................................................
_FOSC_LP              EQU  H'0FFC'     ; LP Osc With 18 ms DRT
_LP_OSC               EQU  H'0FFC'     ; LP Osc With 18 ms DRT
_FOSC_XT              EQU  H'0FFD'     ; XT Osc With 18 ms DRT
_XT_OSC               EQU  H'0FFD'     ; XT Osc With 18 ms DRT
_FOSC_INTRC           EQU  H'0FFE'     ; INTRC With 1 ms DRT
_IntRC_OSC            EQU  H'0FFE'     ; INTRC With 1 ms DRT
_FOSC_EXTRC           EQU  H'0FFF'     ; EXTRC With 1 ms DRT
_ExtRC_OSC            EQU  H'0FFF'     ; EXTRC With 1 ms DRT

_WDTE_OFF             EQU  H'0FFB'     ; Disabled
_WDTE_ON              EQU  H'0FFF'     ; Enabled

_CP_ON                EQU  H'0FF7'     ; Code protection on
_CP_OFF               EQU  H'0FFF'     ; Code protection off

_MCLRE_OFF            EQU  H'0FEF'     ; RB3/MCLR Functions as RB3
_MCLRE_ON             EQU  H'0FFF'     ; RB3/MCLR Functions as MCLR

_IOSCFS_4MHz          EQU  H'0FDF'     ; 4 MHz INTOSC Speed
_IOSCFS_8MHz          EQU  H'0FFF'     ; 8 MHz INTOSC Speed

_CPDF_ON              EQU  H'0FBF'     ; Code protection on
_CPDF_OFF             EQU  H'0FFF'     ; Code protection off
```

As mentioned, it is an ASCII text that lists a series of labels to which are associated the hexadecimal values that will have to be inserted in the ***configuration word***.
Matches are created with the `EQU` (equal) command, so the compiler will replace the label with the specified value.
The text clarifies the function of the various options with appropriate comments, which is described in more detail in the component datasheet.

Labels are identified as "somewhat peculiar labels" by the initial _ sign. They are not reserved labels of the Assembler (starting with), but, since they have been defined by the inclusion of the ***processorname.inc*** file, they cannot be defined again in the source, otherwise the duplicate label error will be reported.

ATTENTION!

If we omit the configuration line, the default parameters will hardly be adequate for our application.

The configuration can take up a few pages of the datasheet, but our template collects a summary of it in its final part. This summary is a text, in the form of a commentary and therefore does not form part of the compilation. It is deliberately placed at the end of the source, so as not to disturb during its reading, while being available for consultation. Placed at the end of the source, after the directive to close the compilation, it does not even become part of the **.lst file**.

To clear up some doubts, let's say right away that some commonly exemplified ways to deal with configuration are to be avoided entirely:

- Enter the configuration as a hexadecimal number.
  For example,:

```
__CONFIG 0x3FF1
```

  This is accepted by the compiler because the expression accompanying the directive is considered a valid value and inserted into the ***.hex file***.
  It is, however, an emeritus nonsense, which is all too common, because: who can say, without consulting the data sheet, what is the configuration that corresponds to that hexadecimal?
  The *hexadecimal configuration word* is not at all clear and is not even easily manageable since it is an absolute value. No more time would have been wasted writing a comprehensible line using the labels that the processors' reference files provide:

```
__CONFIG _CP_OFF & _HS_OSC & _WDT_OFF & _PWRTE_ON
```

  from which anyone who reads the source can understand how the chip is configured.

- Program only a part of the bits of the configuration word: this does not ensure that the rest are at the level needed for the application. Of course, the situation of some options may not affect work in progress, such as code proofreading protections, but by programming all the options, you have two advantages:
  - I know for sure what situation I want
  - I know all the possible options.
  And, when in doubt, the`_____CONFIG` It immediately tells me how I set up the processor and I can check if there is an error and fix it in an instant.

- Even in cases where the only solution for the configuration is the one provided by the development environment, which does not accept other ways, it is good practice to indicate in the source how this configuration was defined, in the form of a comment line; In this way the source is complete documentation, whereas if you don't do this, you would have to have the whole project with its environment to have all the elements necessary to program the chip.

More details on the configuration **can be found here**

Configuration Options

Let's see in detail the configuration chosen in the case of the processor we are using:

Label	Function
_CP_OFF	This indicates that we don't want to turn on chip reread protection. This protection, which is used to prevent copying of the program by others, is usually activated in realizations directed to sale (**_CP_ON**), while in the case of development, debugging, instruction, it is advisable not to activate it.
_MCLRE_OFF	This command tells you that we want to use the **MCLR/GP3 pin as the** GP3 **digital input**. In case we want to use the pin as an input for an external reset we would have to declare **_MCLRE_ON** .
_IntRC_OSC	This option sets the type of oscillator we want to use. In our case, we use the internal oscillator. Otherwise, it will be possible to choose: • **_LP_OSC** external low power oscillator (max. 200kHz) • **_XT_OSC** external oscillator (max. 4MHz) • **_ExtRC_OSC** external RC oscillator (max. 4MHz) Obviously, if you choose a mode with an external oscillator, you will need to have the necessary hardware.
_IOFSCS_4MHZ	This command tells you that we want to use the internal 4MHz oscillator. In case we want to use the 8 MHz frequency we would have to declare **_IOFSCS_8MHZ** (this option does not exist for 12F508/509).
_WDT_OFF	This command disables the internal Watchdog, which, for now, we don't want to use. The alternative, which activates WDT, is **_WDT_ON**
_MCPU_ON	This command enables weak pull up on the **GP3 pin** and makes the use of an external pull up unnecessary for this application. If we don't want that, we'll set **_MCPU_OFF**

The configuration is different from PIC to PIC depending on the integrated resources. Therefore it will be necessary to adapt the command line to the microcontroller used.

However, we can consider that it contains elements common to the members of a given family (this becomes evident by comparing the sources for the different PCIs) and is helpful for a correct configuration.

Le Label - Symbols instead of absolutes - EQU and #define

As we can see, for now we have focused a lot of "stuff", but we have not yet arrived at writing actual instructions; we have limited ourselves to preparing the build environment and the hardware resources of the processor, using some directives of the Assembler. We can, however, still "define" something useful.
We must keep in mind that:

A fundamental principle of the Assembly is to use symbolic indications as far as possible and not absolute values.

And we have seen various examples of this in the previous pages. This applies to every aspect of the microcontroller, including the pins and their functions.

It should be more or less known that digital function pins are logically grouped into groups, generally called **PORTs**. In our case, the small chip has only 8 pins, of which 6 are usable as I/O and which are part of a **PORT** called **GPIO** (***General Purpose IO).***

In our case, the **GPIO** , which collects the six available pins, looks like this:

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

Let's look at the very simple relationship between the pins of the chip, the labels related to the digital function and the bits of the **`GPIO.`**
For example, bit 4 of the **`GPIO`** will be called **`GP4`** and in this type of package it will correspond to pin 3 of the chip; and so on.

Since there are only 6 pins that can be used as I/O, bits 6 and 7 of the register have no function: reading them will read a 0 and writing them will have no effect.

On reset, the value of these bits is random.

From the point of view of the Assembly, we will indicate a bit, for example bit 5, with its **`GP5`** label and, to be exact, with the form:

```
GPIO, GP5
```

Which means: " the **`GP5`** bit of the **`GPIO`** **`register`** ".

In Assembly syntax, the comma has a special value, that is, it indicates separation and concatenation between different elements of the same operation.

In this case, the **object** consists of the register address (file) [**6**], **separated by a comma from the bit** [**5**] to which the instruction is directed.

From the point of view of binary codes that indicate addresses in the chip's memory map, the **GPIO** register occupies location 6 of the area containing the RAM and control registers.
We also saw how pin 3 of the DIP-8 package corresponds to bit 5 of the **GPIO register.**

Therefore, if we want to express in numerical (absolute) values, as addresses in the RAM memory area, the situation of the pin that controls the LED we should write:

```
06, 05
```

thus indicating bit 5 of register 6. Now, unless you look at the component data sheet, who can tell at first glance which register corresponds to **6** ? And what is the function of bit **5** ?
But if we associate each absolute value with a label (= label, name), it becomes much more comprehensible.

In the **p12F519.inc** file that we included at the beginning (of which we gave the path and which we invite you to read, with any editor, since it is an integral part of the source), we find the line:

```
GP5 EQU H'0005'
```

EQU

EQU (equivalent, i.e. =) is an assembler directive that indicates the equivalence between a label and a constant. Its syntax is:

The initial label starts in the first column and is therefore considered defined by the

Label	sp	EQU	sp	object	sp	[; comment]

As usual, the comment is optional. I our case:

```
GP5 EQU H'0005'
```

the compiler, every time it encounters the **GP5** label, replaces it with the value 5. Also for the **GPIO** register there is a label with an equate:

```
GPIO EQU H'0006'
```

which means the address of this register in the memory map is 6.

Writing H'0006' or 06h or 0x06 or .6 or simply 6 is completely equivalent.
In fact, the digit 6 has the same value both in decimal and hexadecimal or octal base (in binary base we should write, on 8 bits, 00001010).

Consequently the writing:

```
GPIO,GP5
```

causes the compiler to replace it with:

```
H'0006', H'0005'
```

But it is evident that indicating **`GPIO, GP5`** is much more practical and understandable.

La RAM

At this point in the source you might be wondering how to define the RAM memory you need, but in this simple program you don't use it and we'll leave this section out for now.

With this we have thus exhausted the initial "formalities" and we can consider the instructions true and Own.

Instructions

An assembly source is based on the processor's own instructions, the so-called opcodes.

Until now, we haven't written any of the instructions that the processor will execute; Everything we've seen is simply related to the configuration or build environment.

The abundant amount of lines written for now is composed only of comments or Assembler commands. Of course, this introduction can be lightened, but for the time being, until you are quite sure, it is advisable to follow the example of this long, but obligatory, preamble, since it allows us to arrive in an orderly manner at the actual program, that is, the sequence of instructions that must be executed.

The Syntax of Instructions

The syntax of a line that contains an instruction is:

[label]	sp	Opcode	sp	object	sp	[; comment]

The square brackets [ ] indicate that the element is optional and therefore it is not mandatory to insert it in the line.

Therefore:

- **It is possible to set a unique opcode for each line**, i.e. the Assembly is a line-based language
- **The opcode cannot start in the row in the first column**.
- **It must be separated from its object by another space**
- **The object is required for some opcodes and is not present in others**
- The line can be completed with a comment, separated by a space and a *semicolon* (;)
- The line can be started with a label, in the first column

What is placed in square brackets is indicative of an optional field, i.e. not mandatory and which may not be entered at the discretion of the programmer.

Note that "space" refers to one or more spacings in the text, which can also be obtained with the TAB key.

The initial space/tab is used to prevent the opcode from being in the first column, a position reserved by the Assembler for labels, comments and some directives, while the following spaces/tabs separate the various components of the line.

We can add a further detail: the line containing an opcode can be associated with a label. In this case **, the label is self-defined, starting in the first column**. This label identifies the line to which it belongs, and in the compilation it takes the value of the program memory location where the following opcode is placed.

Assigning a label to an instruction line allows the compiler to uniquely identify it and make it available to the programmer to retrieve that line through the symbol.

Since Assembly programming is based on the codes in the instruction set, it is necessary to study them fairly carefully. Here you will find some pages related to this topic.

As for the set, Baseline is very small, only 33 opcodes, which is easy to learn in no time.

We will see for each opcode used to draw a brief description.

Before the first instruction : Positioning in program memory

The actual program typically begins with the assignment to the reset vector of the first instruction to be executed. This operation is necessary as far as the **POR** (***Power On Reset***, i.e. when the power supply arrives) as well as after a Reset controlled by the **MCLR pin**, the ***Program Counter*** is positioned on the 00 address and from there the scan of the stored opcodes begins.
We command the compiler to set the program memory address to this value:

```
RESVEC     ORG   0x00
```

The line assigns the **RESVEC label** to the address (**ORG** - source) in program memory **00h**. From here on, the instructions will follow one another in the memory, until they are placed again

ORG

The function of the **ORG** directive is to place the compile address in program **memory** at the specified value.

The syntax is as follows:

[label]	sp	ORG	sp	object	sp	[; comment]

The initial label is optional, and the directive cannot start in the first column. The subject is the address from which the subsequent instructions will be completed.
For example, if 00h is specified as object, the first instruction will be located at address 0 of the program memory. If the subject is 0x200 the following statement will be filled in at this address.

Remember that the 0x00 address is the reset vector, i.e. the point where the program instructions must start as it is the location that is pointed first by the Program Counter of the PICs. Notice that **RESVEC** starts in the first column: in this way the word is self-declared as a label; it is given the value of the object of **ORG,** i.e. **00h**.
In this case, associating a label with this address has no practical function: the label will not be called at any point in the program. You could, therefore, write the line simply as:

```
          ORG   0x00
```

However, it is not bad to assign labels to the salient points of the source, as a guide and indication of what you are doing.

Incidentally, please indicate:

0 of **00** of **00h** is **0000h** of **H'0000' 0x00**

It's the same thing. You can use one or the other diction as an indicator: for example, in the PIC12F519, the program memory is 1k words wide, so it goes from 0 to 3FFh; You can then write 000h to indicate the width of the addresses.

Likewise, **03FFH** or **3FFh** or **003FFh** or **H'003FF'** or **0x3FF** are the same thing.

If you're still struggling with the number systems, **check out these pages**.

Calibrating the Oscillator

Since we use the internal oscillator, it is advisable to calibrate it, to obtain the expected clock frequency, acting on the **OSCCAL register**.
This register is one of the **SFRs** (***Special File Registers***) and has the purpose of adjusting the frequency generated by the internal oscillator within certain limits.
By simply copying **W into the** OSCCAL **SFR**, we ensure that the frequency of the oscillator is as precise as possible: this is possible since at reset the calibration value set by the manufacturer is automatically loaded in **W**. So it's time to use it to adjust the internal oscillator.

```
; VALUE CALIBRATION , First Intrinsic Instruction

RESVEC    ORG   0x00

; Internal Oscillator Calibration
        movwf      OSCCAL
```

MOWF

The opcode we use is **movwf** (***Move W to File***), which copies the contents of the W register (also called WREG) into the file in question. The structure of the row is:

[label]	sp	movwf	sp	object	sp	[; comment]

The object is the "file" to which the copy is directed.

In the line we wrote there is no initial label (which would be useless) and not even a comment, which is also optional. However, the comment on the action was expressed in the previous line.

A few words about calibration.

Before the chip leaves the production line, the test systems also check the frequency of the clock generated by the internal oscillator and write an optimal calibration value to Flash memory to have the most accurate possible frequency compared to the nominal value.
This calibration value in the Baselines is not "automatic", but must be applied through the program instructions.
The calibration operation is not mandatory, but there is no reason not to do it as we have seen, for two reasons:

- The first instruction is the moment when the calibration value is present in W. If **W** is used for other purposes, this value, written to program memory, is no longer recoverable during operation, as the Baselines have no instructions for accessing this area intended as data.

- The accuracy of the oscillator may not be important to the application, but it is generally common for a program to contain time elements for which a

reasonable accuracy. Leaving the oscillator uncalibrated can lead to even significant differences from the nominal frequency, which would alter all the parts of the program that evaluate timing.

So, even if it's not specifically important to your program, as is the case in this tutorial, it's always a good idea to put the simple calibration instruction at the top of the source instruction list. At 4MHz clock, it takes only 1 us of time to run.

A few notes on the W register.

We have entered the actual program, that is, the list of instructions that will be executed, and we will see how the W register occupies a fundamental part.

The **W** register (***Work register***) is essential in the structure of 8-bit PICs as it is the medium with which to process data, numbers, results of operations.
It has a width of 8 bits and acts on the RAM memory, both on the data area and on the peripheral control registers (SFR).

For example, to load a certain value into a RAM or SFR register, I need to:

- **load the value in W** `movlw` (***Move L**iteral to **W***)
- **copy W to the destination register** `movwf` (***move W** To **file***)

file and literal

Please note that for Microchip:

1. ***file*** indicates a location in the RAM area, i.e. a data register or a control register of internal functions (**SFR** - ***Special F**unction **Register***)

2. ***literal*** indicates a fixed number

I/O Initialization

Once the oscillator has been calibrated, we can control the I/O pin to which the LED is connected in order to cause it to light up.

To get to this, it is necessary to consider a point already mentioned at the beginning: pins can take on numerous functions, among which that of simple digital I/O may not be the prevailing one. This

it depends on the philosophy on which Microchip bases the construction of its chips, which is to favor functions that have the lowest power consumption.
That is, when the voltage arrives, an internal mechanism assigns a default to the various pins, which, in principle, favors the analog functions and the digital inputs, since these are the ones that determine the minimum energy consumption.

In our case, from the description of the component, available here, we can detect an important table, called Order of precedence for pin function from which the **GP5** pin results, in addition to the generic I/O function, also has the management function of the external oscillator. This situation is determined in the initial config: since we have selected the internal oscillator, the pin is readily available as digital I/O.

We would like to remind you that:

A microcontroller pin can

1. Perform as a digital input or output or as an input or output for one or more of the function modules integrated in the chip
2. Can only perform one function at a time
3. On reset, defaults are set that privilege the conditions of minimum power consumption

By default, the **GP5** pin is configured as digital I/O and, to be exact, as a digital input (lowest power consumption situation).

But we need it as an output, to control the LED and, in order to use it in this way, we need to act on the direction register of the digital I/O (**TRISGPIO**). Through this register, whose bits correspond to the I/O pins, it is possible to determine the direction of the signal, whether incoming or outgoing.

For Microchip, the equivalence is typical:

bit	direction
1	entrance
0	exit

This is not accidental, but due to the fact that, in general, the unprogrammed cells are at level 1 and that the condition with the lowest consumption is the input condition.
We can consider that:

- **On reset, pins that can take on a digital function are by default arranged as inputs**. If we want to use them as outputs, we need to program the corresponding control bit in the TRIS register to 0
- **Not all pins can be used as digital outputs**; e.g. in PIC 12F508/9/10/19 the GP3 pin can only be used as an input. The others (GP0/1/2/4/5) can operate as inputs and outputs.

- **Not all pins are immediately available as digital**, as the chip integrates other functions to which, by default, the pins are dedicated. In the component datasheet, there is a section dedicated to digital I/O that specifies these points.

We have said that the pins in digital function are logically collected in groups, called PORTs.
In our case, the PORT GPIO looks like this:

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

Similar correspondence in the TRIS register:

TRIS	Bit	7	6	5	4	3	2	1	0
	Label	-	-	TRISGP5	TRISGP4	TRSIGP3	TRISGP2	TRISGP1	TRISGP0
	pin	-	-	2	3	4	5	6	7

As before, since there are only 6 valid pins, bits 6 and 7 of the register have no function: reading them will read a 0 and writing them will have no effect.

At reset all bits are forced to 1, i.e. the pins are intended as digital inputs. If I want to use the pins as outputs, I have to set the corresponding bits to 0.

For example, in the case of bit 5, I'll have to send bit TRISGP5 to 0, which makes pin 2 a digital output. So let's add more lines to the source:

```
MAIN:
; inizializzazioni al reset
     clrf    GPIO          ; preset GPIO latch a 0

; TRISGPIO     --0----- GP5 out
     movlw   b'11011111'  ; PORT Direction Mask
    ;movlw   0<<GP5       ; Alternative Writing
     tris     GPIO        ; To the Management
```

MAIN is in the first column, so it's the label declaration.
The **MAIN** label, as in the previous case, is purely indicative and has no practical function in this program, since it is not invoked, but it is useful to indicate the starting point of the main instructions.
The **:** is an additional operator, which is not part of the label definition; this is a custom derived from the rules of some assemblers where it becomes necessary for the label declaration **if this is the only line entry**. In MPASM it is not mandatory and can be omitted

CLRF, MOVLW, TRIS

The initialization sequence of pin 2 consists of three instructions:

`clrf GPIO`	(**CLRF** - *Clear File*) is an instruction that zeroes the contents of a registry. Here it resets the port latch and sets the outputs to a low level. Used for safety if the connected loads are active at a high level
`movlw b'11011111'`	(**movlw** - ***move literal to W***) loads in W the value decided for the direction register, with the bit 5 = 0, corresponding to GP5, which indicates the digital output function to write it to the direction register
`trip-of-a-kind GPIO`	**tic-tac-toe is a special instruction to be used only with Baselines** that aims to copy W to the direction register (tristate or tris) **TRISGPIO,** in order to change the input-output direction of the pins as desired.

This "quirk" is necessary as the **TRIS** register is not in the memory map and is only accessible in write, with this articular method.

All the instructions have the syntax already seen; In particular, it must be noted that they cannot start in the first column.

As far as the function of the **clrf GPIO line** is concerned, it is present at this point of the source because it is common practice to predefine the state that the output pins will have through an action on **the GPIO** register; this is due to the fact that the circuit upstream of the pin is a set of latches whose output is transferred to the pin only if the pin is configured as an output in **TRISGPIO**.
So I can write the contents of the latches even if the pins are configured as inputs: the value saved in the latch will be passed to the pin when and if it is configured as an output. The pre-definition is not an obligation, but a good custom, as it allows you to determine in advance the value assigned to the pin; So I first write the value to be assigned, which is stored in the latch register, then determine the direction of the PIN.

If you determined the direction first, as soon as the pin was configured as the output, its value would be random, as the contents of the latches are not determined at the reset. Of course, once the direction has been established, the value will be given to the pin, but this means that you will find yourself with a certain time (minimum 2 cycles) with the pin that may have a different level from what you initially want and this may be unwelcome to particular peripherals.

So, since it only costs 1 cycle (or at most 2) to determine the status of the output pins, it is appropriate to enter the instruction. Here we bring all latches with a **clrf** statement to 0. Since we need to bring **GP5** to a high level, we could also write:

```
; inizializzazioni I/O al reset
; GP2 out
  Bsf     GPIO,GP5        ; GP5 latch preset =1
  movlw   b'11011111'     ; PORT Direction Mask
  Tris    GPIO            ; To the Management
                          Register
```

ahead of its time. However, it is common to consider the active pins at a high level and then start from their "disabling" at level 0 and activate them only with a subsequent instruction.
If what is connected has different needs, for example it is active at level 0 or needs to be activated as soon as possible after the reset, you will act accordingly.

It should be noted that the

```
movlw b'11011111'
```

it could be replaced by `movlw 0xDF` or `movlw H'DF'` or `movlw .223` depending on the numerical base used, since it is the same value, expressed in different roots. If you have declared a decimal root at the beginning, you could also write

```
movlw 223
```

since the compiler would consider an unspecified number as a decimal base, but it is not recommended to write, both because not defining the base of each number risks introducing inattention errors.

Moreover, by using the number in binary form you will immediately be able to see "graphically" the expected situation for the various bits.

bit	7	6	5	4	3	2	1	0
binary	1	1	0	1	1	1	1	1

An alternative to the number expressed above is the use of the **<<** sign, a left-shift command.

```
; TRISGPIO     --0----- GP5 out
    movlw     0<<GP5          ; shift 0 to GP5 position
    tris      GPIO            ; To the Management
```

The `movlw 0<<xxx` line instructs the compiler to use a binary number, 11111111 by default, as the object of the instruction, in which the value 0 is inserted at the position corresponding to the xxx label.
This is advantageous as it does not require us to know where the xxx position is: the label, defined elsewhere, informs the Assembler of where the 0 bit should be placed. In our case it is the file ***pic12F519.inc*** contains the desired information:

```
GP5 EQU H'0005'
```

so the 0 bit will be inserted at position 5, generating the number 11011111.
This is an example of the possibility of the Assembler to help the programmer significantly, as long as he follows the fundamental rule explained above: as far as possible, do not use absolutes, but only and exclusively labels.

Turn on the LED

Now the **GP5 pin** is defined as the digital output. We have created the situation to be able to turn on the LED:

```
        bsf   GPIO, GP5      ; Lights up LEDs
```

BSF

The bsf instruction (*bit set on file*) brings **set = 1** the bit of the indicated register. The structure of the row is:

[label]	sp	movwf	sp	object	sp	[; comment]

The object is the indication of which bit of which register is to be set.

In the line we wrote there is no initial label (which would be useless) and not even a comment, which is also optional. However, the comment on the action was expressed in the previous line.

In this case, the **object** consists of the GPIO register (file) address [**6**], separated by a comma from the **GP5** bit [**5**] to which the instruction is directed.

The following equivalences apply to Microchips

Set	**1**	**High Level**	**H**	**true**
Clear	**0**	**Low level**	**L**	**false**

Since we decided at the beginning to have linked:

```
LED => GPIO,GP5
```

and given that

```
GPIO,GP5 = 6.5
```

The compiler will turn the line into instruction binary code that will take Bit 5 of register 6 to level 1. The LED is lit !

Instructions - Closing the Execution

One last instruction: **since our program is not made up of a loop**, i.e. it ends when the LED lights up, it is necessary to make sure that the ***Program Counter*** does not advance further.

In fact, it should be understood that, once an instruction has been executed, the program counter automatically advances to the next and so on until there is a clock available (and supply voltage...).

Since once the LED is lit, the program has exhausted its work and it is not necessary to execute other instructions (which, by the way, do not exist), it is necessary to block the execution. Otherwise, execution will continue on the rest of the program memory, with the ALU trying to decode the random contents of the cells as opcodes.

The easiest way is to close the list with a loop on itself:

```
stop     goto stop       ; blocco
```

Stop is a label that indicates the address at which the line is located once it is placed in program memory. Starting in the first column, it defines itself at that point.

Goto

The instruction is simple as an action: it blows up the program at the location it has as its object. The syntax is the same as the previous one for the other statements:

The structure of the row is:

[label]	sp	Goto	sp	destination	sp	[; comment]

GoTo requires an object that is the address of the destination to which you are making the jump and that can be a label or an absolute value. Obviously, the address does not need to be expressed with a number, since it can be assigned a label.

In reality, **goto** is a complex instruction, which takes 2 cycles to execute (2us @ 4MHz clock); it acts on the ***Program Counter***, replacing the address contained therein, which points to the next line, with the one indicated as the destination.
It is therefore a so-called unconditional jump, i.e. it is carried out immediately.

Why stop the program?

In the case we are studying, the instruction makes a jump of the Program Counter to the same address, a loop that will be executed indefinitely as long as the supply voltage is present and

the clock. In this case, the result is to block the program from executing this instruction only. Then the line:

```
loop     goto loop      ; block
```

It tells the program to keep jumping on the same line, tending the execution stuck there indefinitely. For those who know, C corresponds to:

```
for (;;)              while (1)
 {                     {
      ;        or           ;        and equivalent.
 }                     }
```

This is necessary because, as we have pointed out, as long as the clock is available, the internal mechanisms of the chip advance the ***Program Counter*** **(PC)** and therefore the execution of what they find in the program memory cells.

If our program ends up at that point, we must prevent the ***Program Counter*** from proceeding further with the scanning of the program memory.
If we didn't block execution in this indeterminate loop, it would go on trying to use the contents of the next program memory as valid opcodes; but this part of the memory has not been written and may also contain random values. You would have the ALU trying to decode these values as instructions: at best, in the unprogrammed part of memory the content is FFh, which corresponds to the **`movlw 0xFF instruction`**, which is executable.

The instruction has no effect on the outside, it just loads the FFh value into the W register, and we can't realize what is happening: the PC continues to scan the program memory, executing this instruction up to the last location, where it finds the **`valore_di_calibrazione movlw`** and then resumes from the reset at 00h, in a never-ending cycle.
This, defined by English speakers as ***wrap around*** , is easily understandable if we imagine that the program memory is wound as if on a cylinder: once you reach the bottom, you start from the beginning.

There would be a constant restart of the program, which is not what we want.
And in the worst-case scenario, if the memory contained values other than FFh, instructions would be executed with possible undesirable effects outside the micro, with the possibility of crashing the internal logic.

Therefore, it is necessary to put a stop to the progress of the Program Counter and hence the need to close the program with a closed loop on itself.

To be even clearer, once the LED is lit, the execution of the instructions does not stop, since it cannot stop, but continues, as long as there is the supply voltage, in the execution of the **`goto`** on itself. Since the instruction is internal and has no effect on the pins, we don't notice it, but that's the way it is.

You can avoid the definition of the label with the use of the **`$`** symbol, which we talked about earlier. Please note that it **indicates the current address of the *Program Counter***:

```
        goto    $           ; Execution Block
```

The use of $ is not too desirable, but in cases such as this, where the implied indication of the symbol is obvious, it is pointless to attach a label, since the meaning of the statement is clear.

Another way to stop such a program is to bring the microcontroller into
Sleep:

```
          sleep             ; Execution Block
```

In this case, the stop is achieved by locking the oscillator, which stops the unfolding of the instructions and reduces power consumption. This is a procedure with more complex aspects and which has the advantage of bringing the micro into a condition of reduced power consumption (since the clock is blocked), while a restart (***wake up***) is possible, which has different implications, as we will see later.

Closing the Build - END

One last thing is needed: **if the program is finished, you have to tell the compiler that its work is also finished and** this is achieved with the `END command`

```
;*****************************************************************
;                           THE END
  END
```

END

The directive terminates the compilation: **what is written beyond the line containing the `END`, therefore, will not be considered at all by the compiler.**

In our case, all the lines in the "Documentation" section will not affect the result of the assembly and are only there for quick reference to avoid having to resort to the data sheet.
The directive, as usual, does not have to start in the first column. The directive can have comments, but it does not need any object.

Program Close vs. Build Close

There is a big difference between closing the program and closing compiling:

- **the stop imposed on the PC, forcing it to loop on itself (`goto $`), serves to prevent the program from continuing to scan the contents of the memory**

program following the last instruction, which would send the processor to execute random codes contained in the part of the unprogrammed memory, with the result, in the best case, of continuously restarting from the reset.

- **the `END directive`, on the other hand, tells the compiler that this is where the part to be considered ends and that whatever is written after that does not become part of the compilation.**

Obviously, a programme needs both actions.

One last important point.

If we "purged" the source we have in our hands of all the non-essential comments and directives and we don't use labels (except for opcodes), we would end up with only 11 lines of text.

```
LIST p=12F519
#include <p12F519.inc>
__config H'0FCA'
ORG 0x00
    movwf 5
    CLRF  6
    movlw b'11011111'
    Tris  6
    Bsf   6,5
    Goto  $
END
```

These **11 lines** are enough to get the compiler to generate the object to be programmed in the chip's memory, exactly like the commented and detailed source we have just seen.

For those who want to try, just pass them to the compiler: you will get the same **.hex** as you have with the source list detailed so far.

But **the difference between the two cases should be obvious**:

- **In this "reduced" version there is no way to understand at first glance what the program does, nor how it does it, also thanks to the use of absolute values and not explanatory labels.**

- **In the commented source, the comments, although overly abundant, carefully define what and how you are doing and the use of symbols clarifies things.**

Certainly, for such a simple program it can be objected that the drafting of the source with a strict order and form, given perhaps, as in this case, by the use of a template, is superfluous.

But it is precisely the adherence to an order and a form that makes the action of the programmer effective, facilitating the verification of the operation and the correction of possible errors, both logical and syntactic (debugging).

This becomes absolutely essential when dealing with more complex programs than the one we have just seen: you can well imagine the difficulties that would be encountered if the lines were 1100 and not 11 and a good method of writing the source had not been used.

So, **if you really want to learn how to code sensibly:**

- **Maintain a neat shape**
- **Abolish the use of absolute values and use labels as much as possible**
- **Comment wherever there is an operation that is not immediately comprehensible and, even more so, where a logical, even simple, structure is implemented.**

When you have enough independence, you can create your own way of writing the source, even a very short one, but at that point you will have it clear that a form and an order are indispensable.

The effort of writing the source is finally over (... *just to turn on an LED !* , but we will see how most of the lines that do not directly involve the instructions are a "constant" that, once defined and understood, will then be repeated almost by copy and paste in subsequent programs). Now we need two more steps:

1. have the Assembler compile the source list
2. Write the hexadecimal result (executable) to the chip

The MPLAB.

In the case of this first example, the entire working environment (project) for MPLAB 8.83 is provided.

MPLAB is an integrated development environment, i.e. it provides a series of functions to develop a program in an organic way. The work with MPLAB is based on the creation of a "**project**", i.e. a set of information and elements that organize the environment in the optimal way for that particular job.

There are two ways to go:

- create the project directly from scratch, overwriting what is present
- Use the existing project

In the first case, just follow the instructions here. This choice allows you to practice the project creation operation.
If you still don't feel confident, you'll opt to open the ready-made project.

It is necessary to

- download and install MPLAB
- start it and from the main menu line click ***Project->Open*** and in the **Browse** click *C:\PIC\A&C\baseline\CourseA\Baseline\1A_519.mcp.*

This opens the project for this tutorial. Initially, the project

includes:

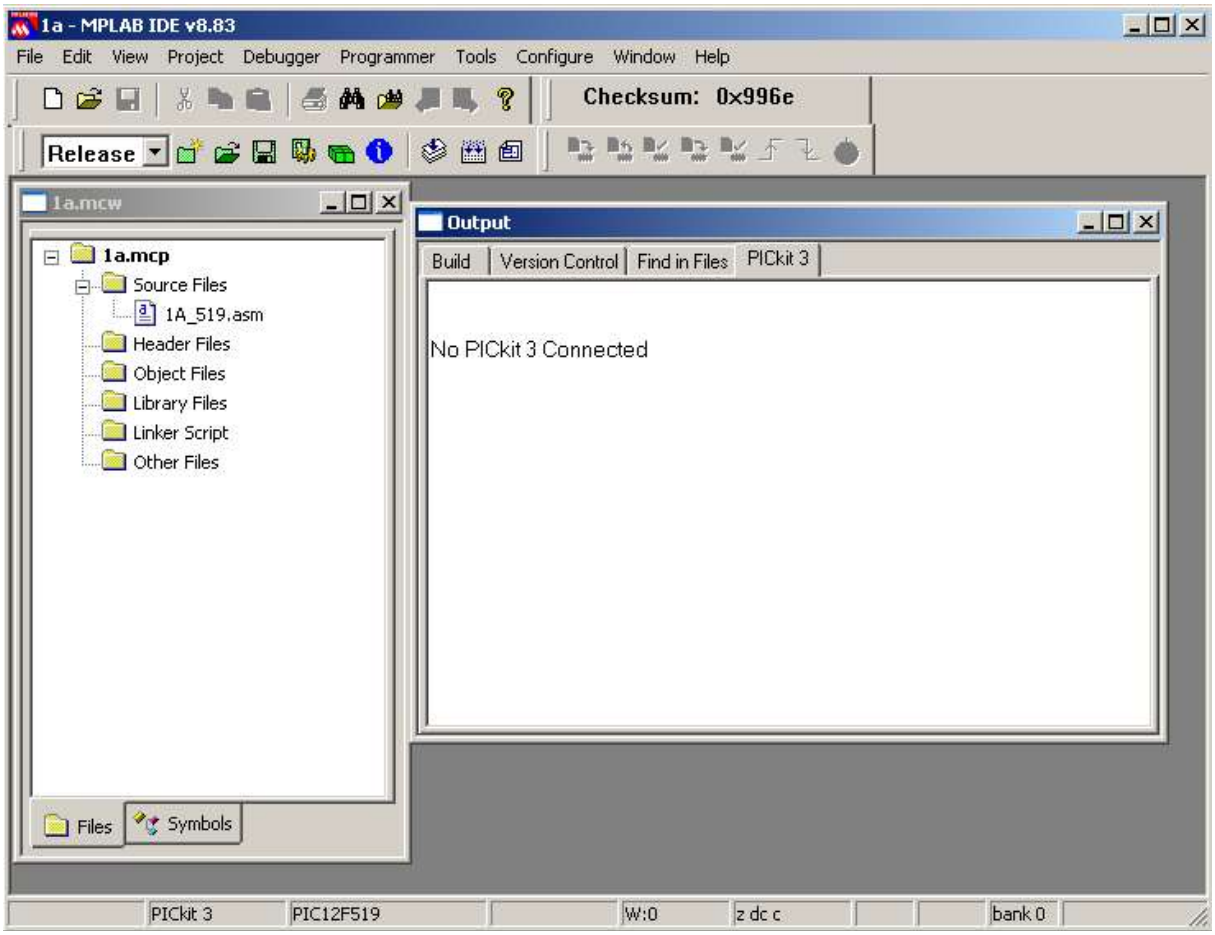
- the use of PIC12F519
- the **PICkit3**

Blueprints for 12F508/508 and 10F200/202 are also available, which we describe below.

Opening the ready-made project is simple: just point to the main menu and select ***Project/Open Project*** and locate the existing project that you want to open. MPLAB will automatically load the necessary items.

If we wanted to compile the existing project for a different processor, we would have to adjust things starting from the main menu command ***Configure->Select Device*** and selecting the desired chip from the list, whose references `p =` and `.inc` should be the same as those contained in the source.

Once the project is open, we find ourselves with this initial window



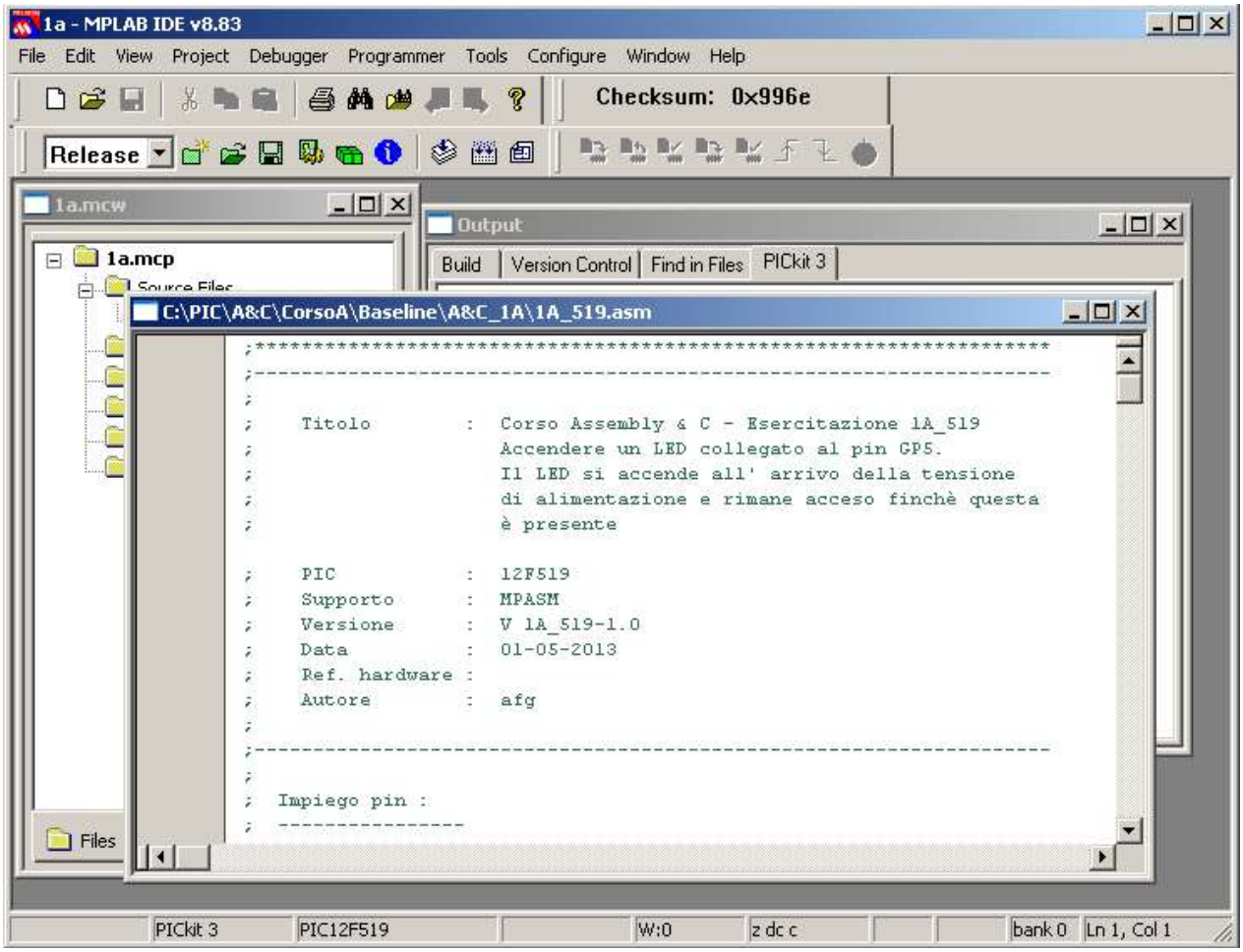
Notice that MPLAB has a virtual directory in the project files window that divides them by genre and function. This directory tree, however, does not physically exist on the disk, but is a creation of the development environment in order to facilitate the work, grouping in a functional way the files that contribute to the project.

The Output window is crucial because it tells us:

- the Build Situation
- Version Information (***Version Control***)
- ***a Find in*** Files tool
- the situation of the tool used (in this case a Pickit3, which is not yet connected to the computer)

These and other available windows are all scalable with typical Windows mechanisms.

We can double-click on the file name ***1A_519.asm*** in the ***1a.mcw window*** to open it with the built-in editor

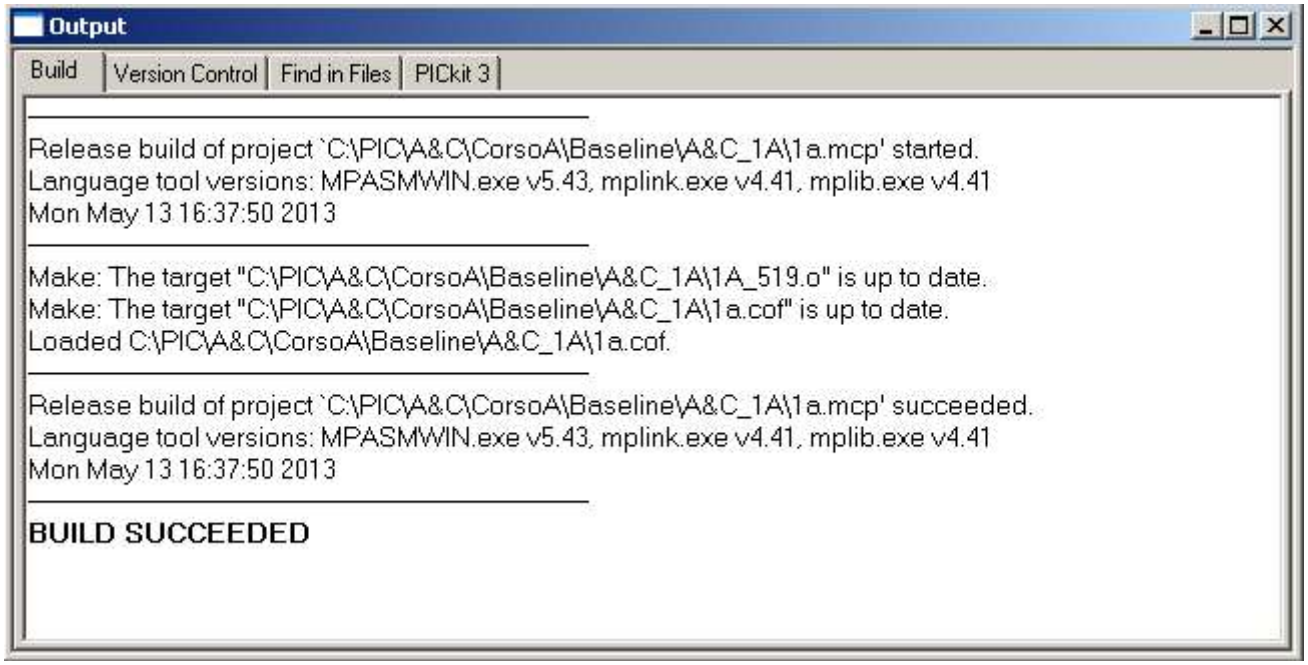


The editor allows us to view and edit the source file at will.

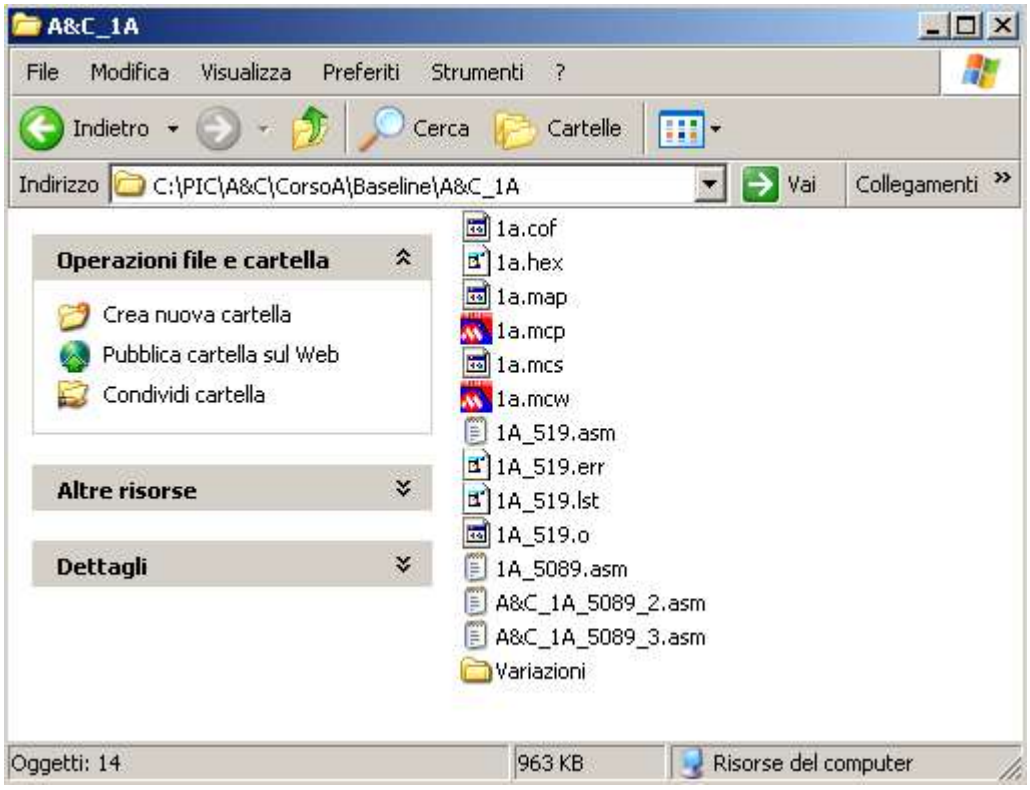
In the use of the editor, we observe the coloring of the various fields which helps significantly in the reading of the text.

The editor is quite simple and straightforward and every time the text is changed, it automatically saves it. It should be noted that it is possible to use any external editor to modify the source file even when the MPLAB window is active: in this case, the saving of the file, by the user, is detected by the development environment and reported appropriately.

Let us now have it filled in by the Assembler. From the icon menu, press  ***Make***, which starts compiling the ***1A_519.asm file***.



The output window informs us of the progress and the result of the operation. Since there are no compilation errors, **BUILD SUCCEEDED** certifies the success.
If we look at the folder where the project resides, for example *C:\PIC\A&C\baseline\CorsoA\Baseline\A&C_1A* we see that some files have appeared, created by the compilation:



These are:

- **.cof** Build symbol and object code information files
- **.err** list of errors generated in the Assembler build

- **.Hex** Machine code in HEX format
- **.Inc** A file to include in the Assembler build
- **.lkr** a file used with the Linker
- **.Lst** List file generated by the compilation
- **.Map** file generated by the linker
- **.or** Object

files in addition to the

above files:

- **.asm** Assembly source file
- **.Mcp** MPLAB project information file
- **.mcw** MPLAB project workspace information file

We can use an editor to open the **1A_519.lst** file which is the list of the assembly and observe its contents. The list contains the source formatted and associated with a line number, to which the generated object code and the location where it will be programmed are paired. At the end of the file are listed the labels that contributed to the compilation.
The **1A_519.err** file is empty, as the compilation did not give any errors.
The **1A_519.hex file** is the hexadecimal object, formatted so that it can be processed by the programming device.

These files, even where they are editable, obviously should not be modified.

With these simple operations, we have transformed the source into a compiled object. We can now move on to the chip programming

---

The programming of the chip.

"Programming the chip" means transferring the contents of the .hex file to the microcontroller's flash memory. To do this, you need a programming device; Its purpose is to constitute an interface between a PC port (serial, parallel, USB) on which the development is carried out and the **ICSP connection** of the microcontroller.

We said in the introduction, that you do NOT need any particular programmer, since the PICkit, among other functions, does this job very well, both under the control of the MPLAB development environment, and stand alone, with an appropriate software package, through the ICSP connection.

We can also program the PIC using a PICkit and its stand-alone programmer software. In the case of the PICkit3, just download it from the Microchip website. The same applies to the PICkit2 and attach the tool to a socket like **this**.

But if you're in the MPLAB environment, you don't need any additional software.
We are talking about Pickit because, due to its low cost, it is one of the most popular Microchip tools, but, of course, you can also use the others, such as ICD or RealIce. However, despite being inferior in performance, Pickit offers an advantage, namely the possibility of powering the circuit under test, which, for small applications, is really very convenient.

Here we find some pages that describe the chip programming operation in the MPLAB environment.

Now the chip contains the program: every time we apply voltage, the LED will be lit.

It must be said that all operations, complicated at first, after a very short time of operation become almost automatic routines and require only a minimum part of the time required to carry out the working program.

Version for PIC12F508/509

We can have different PICs perform the same operation and notice that they are always the same actions.
In the case of PIC12F508/509, the 1A_5089.asm source is provided .

The treatment of the source is more concise than the one written for the PIC12F519 as well as what was written there. This is mainly about the differences.


Let us remember what we said at the beginning: if it is not indispensable, what is explained in one chapter is not repeated in others, so as not to make the discussion more difficult. You should therefore take good note of it and take it into account in the following steps

First of all, let's start from the consideration that **PIC12F508** and **PIC12F509** are completely identical, distinguishing themselves only by the size of the available memory. In fact, they have a single datasheet, in common with the 16F505, which is the 14-pin version of it.

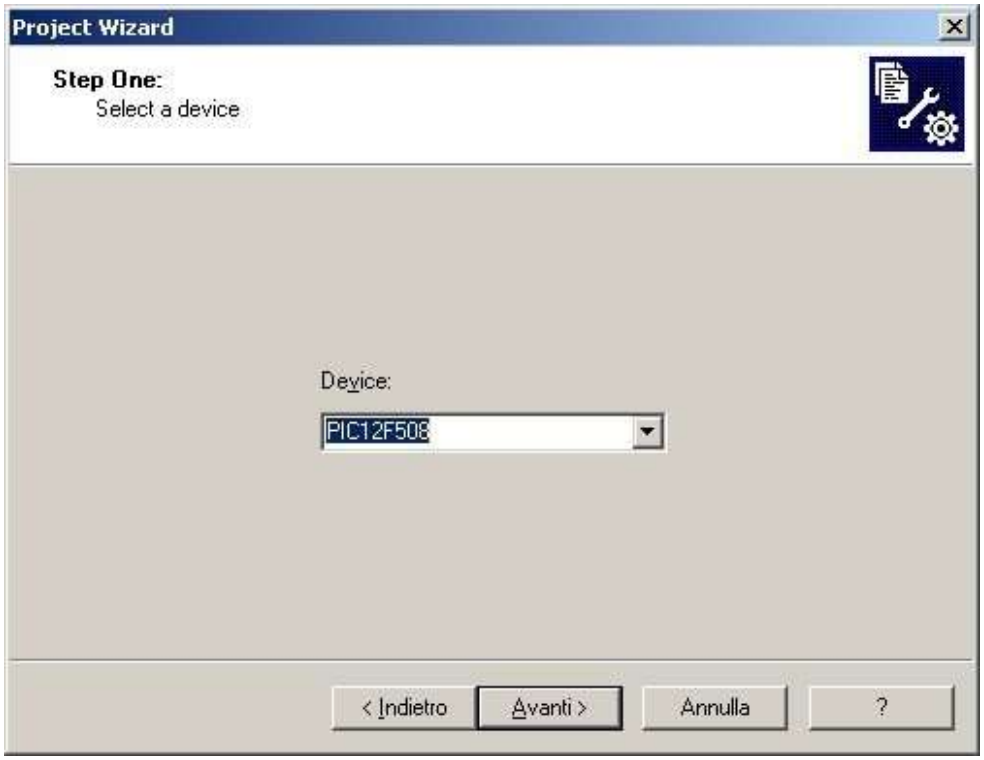
As a result, a program written for one can also be run on the other without substantial modification (as long as the memory availability is not exceeded).

For this reason we can use only one source for both chips, selecting one or the other, depending on availability:

```
#ifdef _12F509
    LIST p=12F509
    #include <p12F509.inc>
#endif
#ifdef __12F508
    LIST p=12F508
    #include <p12F508.inc>
#endif
```

We introduce an automatic choice based on the processor used, with the statement **#ifdef** that is part of the set made available by the Assembler.
This statement, in the structure **#ifdef/#else/#endif** (if defined/otherwise/end of if) simply tells the compiler which file **.Inc** it must be considered.

The **__12F508** or **_12F509** derives from the MPLAB environment: in fact, as the first action of the creation of a new project, the MPLAB wizard asks you to indicate which processor you intend to use:



This choice turns into a parameter **__processorname**, in the case of example **__12F508**, which becomes part of the compilation, both as a symbol and as a control reference.
Then, the **__12F508** will take a value of 1 (i.e. "true"): we can find confirmation of this by scrolling through the file ***.Lst*** generated by the compiler, in the final section that lists the defined symbols, where we will find this equivalence.

If invoked by a conditional **#ifdef** in the source, this condition will make the implied choices active, to the exclusion of the others.

Note that the label of this parameter has a "special" look, starting with two underscores, which indicates that the label rank defined by the development environment (the user cannot define labels that begin with__).

Attention should also be paid to the fact that, while **#define** should be placed in the first column, staements **#if**, as #include, should not be placed in the first column; an error in this regard is detected and reported by the compiler.

the use of uppercase or lowercase in MPASM directives is equivalent, so, for example, **#ifdef** and **#IFDEF** are equivalent.
The use of the **# symbol** is also irrelevant, so that, for example, **#if** and **IF** are equivalent. The choice of one way or the other is personal.

Here the tendency is to use lowercase letters and insert the **#** to make it clear that this is not C language.

It is also useful to maintain a "form" and an alignment of the conditional statements, in order to facilitate their control, since the absence of a closure **#endif, for** example, will generate errors in the compilation.

This practice is very present to C programmers and gives the listings of this language a distinctive look.

```
#if       --> ----|
                  |
#else     --> ----|
  #if     -->---| |
                | |
 #else    -->---| |
  #if     -->-| | |
              | | |
              | | |
  #endif  --<-| | |
                | |
 #endif   --<---| |
#endif    --< ----|
```

So it's a good idea to start each set of statements in the **#if** selection in the same column, as is common for C.

Since each **#if** must be closed by a **#endif** , it is useful to start each conditional chain nested in a previous one by moving the start column backwards, so that the graphics imposed at the source are a powerful means of verifying the congruence of starts and ends.

Going back to the example above, if the PIC12F509 has been chosen in the project , this makes the relative **#ifdef** that is verified valid and introduces the next two lines into the source. So this trait boils down to:

```
#ifdef _PIC12F509
    LIST p=12F509           ;

    #include <p12F509.inc>
#endif
#ifdef _PIC12F508
    LIST p=12F508           ;

    #include <p12F508.inc>
#endif
```

The lines in light gray are as if they don't exist.
If we had defined instead **in the project PIC12F508** , the active selection would have been the second :

```
#ifdef _PIC12F509
    LIST p=12F509           ;

    #include <p12F509.inc>
#endif
#ifdef _PIC12F508
    LIST p=12F508           ;

    #include <p12F508.inc>
#endif
```

To switch from one to the other without opening a new project, just select the desired processor from the MPLAB ***Configure/Select Device*** command line, entering the name of the chip you want to compile from the list.

Obviously, the processor for which you compile must be the same as the one to which you will then transfer the executable program obtained.

ATTENTION!

This point is important to consider, in order to avoid errors of judgement in the drafting of the source. We repeat, therefore, that:

 **`#if` and `#while` are conditional on the operations performed by the Assembler compiler and ARE NOT ASSEMBLY LANGUAGE FUNCTIONS OR STATEMENTS.**

That is, the related loops or conditionings act only during compilation, to format the object in a deliberate way and have nothing to do with the execution of the program. This is to clarify a point that could mislead those who use higher-level languages, where **if** and **while** are parts of the logical structures of the language itself.
Unlike C, in Assembly these logical structures do not exist (and, if necessary, they must be constructed).

The fact that Assembler uses many terms common to other languages derives from the consideration that the implied functions are analogous, even if applied in a different context. In addition to the fact that Assembly is the "oldest" of languages.

The configuration.

As far as the configuration is concerned , it is similar to what we have seen before, with the difference that in PIC12F508/509 there is no possibility to vary the frequency of the internal oscillator, which is fixed at 4MHz, so the relative option will be missing.

There's also no option to have a built-in pull-up on the MCLR, so you'll miss that option as well.

```
;#####################################################################
;                              CONFIGURATION
;
 __config _CP_OFF & _MCLRE_ON & _IntRC_OSC & _WDT_OFF
```

Otherwise, since both processors are completely similar, the configuration will be the same.

As far as the I/O is concerned, if we look at the Order of precedence for pin function table of these PICs, **`GP5`** is also free if an external oscillator is not configured, so the program runs as seen for the 12F519.

Then, having obtained only the digital I/O function on the pin, set it as the output, similar to what we saw for the 12F519.

Now the **GP5** pin is defined as the digital output. The pin is turned to high and the LED lights up.

Now we still need to take the last precaution of closing the execution of the program with a loop

Basically, notice how the source differs from that of the PIC12F519 only in the definition of the processor.

Once the source is verified, we create an MPLAB project and act as seen before.

From the study of these "variations" for different processors we should begin to understand that:

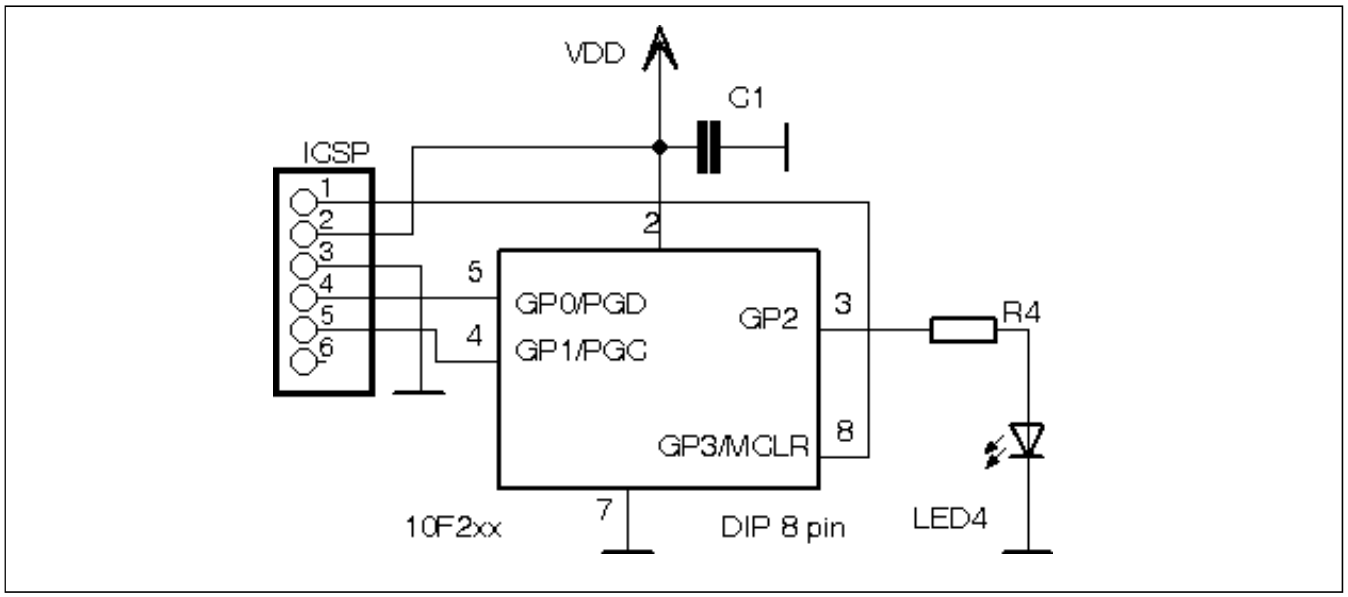
a certain set of actions is similar for different chips, and as far as fundamental operations are concerned, this is true for all PCIs, which are built according to the same philosophy.
The differences between one chip and another of the same family, in this case the Baselines, essentially concern the different availability of integrated peripherals, pins, etc.

Therefore, by setting up an orderly job, there are no particular problems in transferring a source from one chip to another, generally with slight changes of the source, which can be reduced to the definition of the processor and little else.

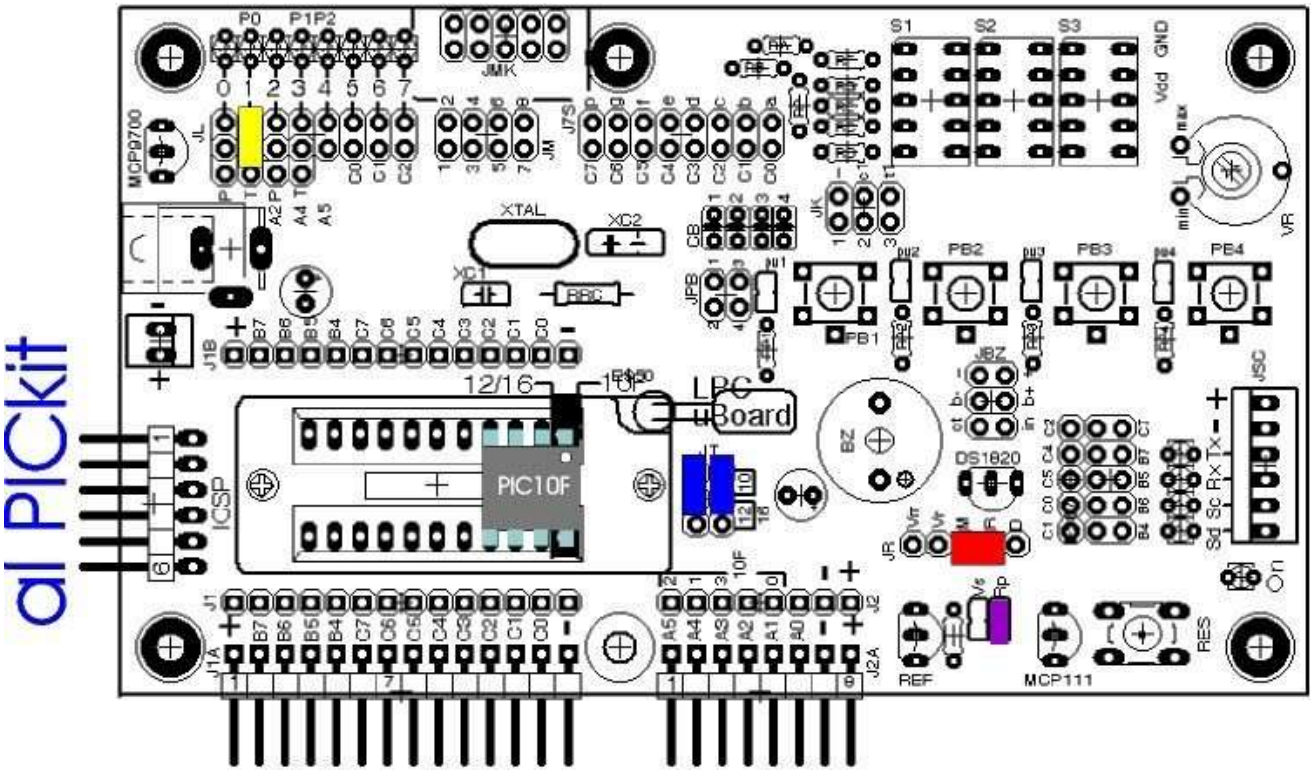
Version for 10F200/202

A version for the 10F200/202 **chips is also offered**.
Basically, these are small 6-pin SMD components (SOT-23-6), but luckily Microchip also produces them in the 8-pin DIP version, which is easy to handle.

The wiring diagram:



And this is the arrangement of the jumpers on the **LPCmicroBoard** for the **10F20x**, in the DIP-8 version



Note1: Since there is no GP5 in these chips, which have only 4 I/O, GP0 is used to control the LED.
So the "yellow" jumper must be positioned differently

Note2: The pinout of the 10Fxxx, even if the package is a DIP 8, is DIFFERENT from that of the 12F. The chip should be placed at the end of the socket and the "blue" jumpers should be switched.

The program

The structure of this chip is quite similar to what we have seen for the 12F508/9/10/19 already seen; The differences are essentially that there is less I/O resources available. There are other differences, such as the amount of available memory, RAM, integrated modules and their SFRs, which, for now, we do not consider, because, where necessary, they require specific management.

The treatment of the source is more concise than the one written for the PIC12F519 since we refer to what is written there. This is mainly about the differences.

 Let us remember what we said at the beginning: if it is not indispensable, what is explained in one chapter is not repeated in others, so as not to make the discussion more difficult. You should therefore take good note of it and take it into account in the following steps

Since PIC10F200 and 10F202 are similar, differing only in the amount of memory available, a program written for one can also run on the other without modification (as long as the memory availability is not exceeded). For this reason we can use only one source for both chips, selecting one or the other, depending on availability, with the switch system already seen for 12F508/509:

```
#ifdef__10F200
    LIST p=10F200           ;
    #include <p10F200.inc>
#endif

#ifdef__10F202
    LIST p=10F202           ;
    #include <p10F202.inc>
#endif
```

To switch from one to the other without opening a new project, just select the desired processor from the command line of MPLAB ***Configure/Select Device***, as seen above.
Obviously, the processor for which you compile must be the same as the one to which you will then transfer the executable program obtained.

As far as the configuration is concerned , it is similar to what we have seen so far, with the difference of greater simplicity due to the low number of integrated resources.

```
;*****************************************************************
;                          CONFIGURAZIONE
; No WDT, no CP, pin4=GP3
 __config  _CP_OFF & _MCLRE_OFF  & _WDT_OFF
```

As far as the I/O is concerned, what we intend to use is **GP2** is: this pin is free to become digital I/O only after excluding its alternative function, which is the T0CKI input function of the Timer0.

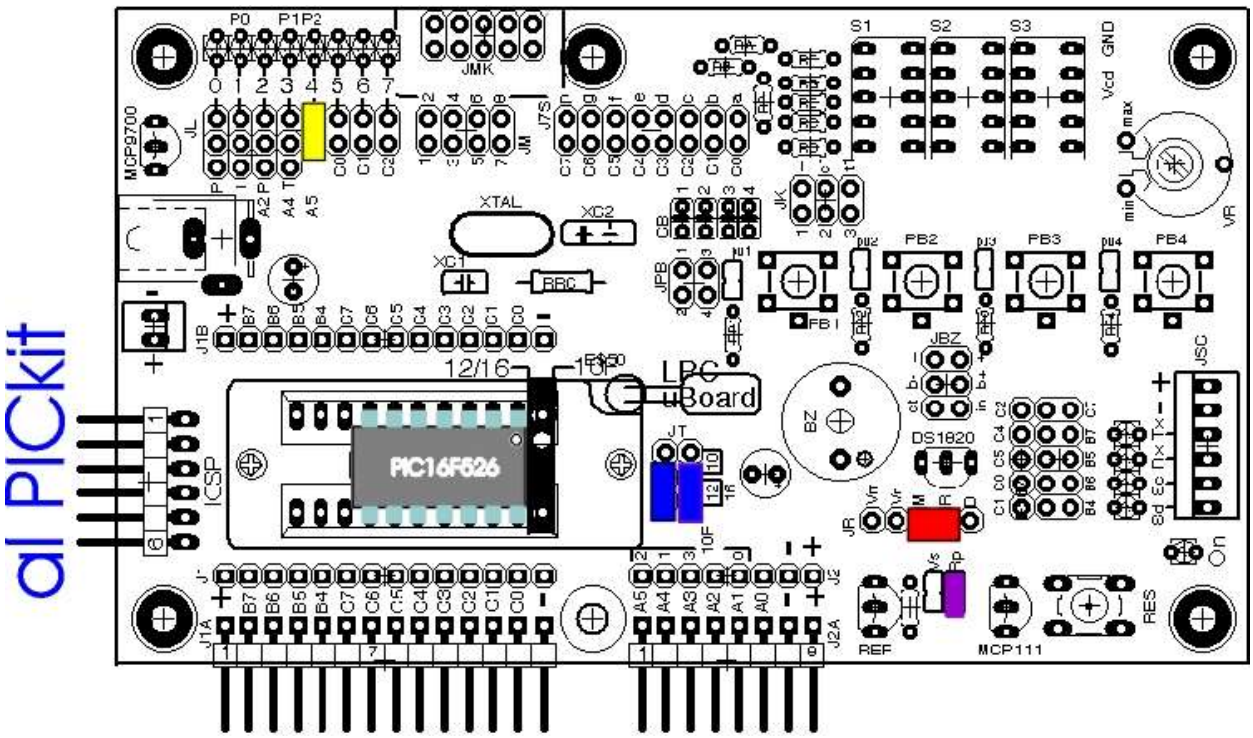
This function is active by default at the POR and must be disabled by acting on the T0CS bit of the OPTION register:

```
; Reset Initializations
      clrf  GPIO          ; preset port latch a 0

; disable T0CKI
 movlw b'11011111'
      OPTION

; GP2 come out
      movlw b'00011111'
      tris  GPIO          ; To the Management Register
```

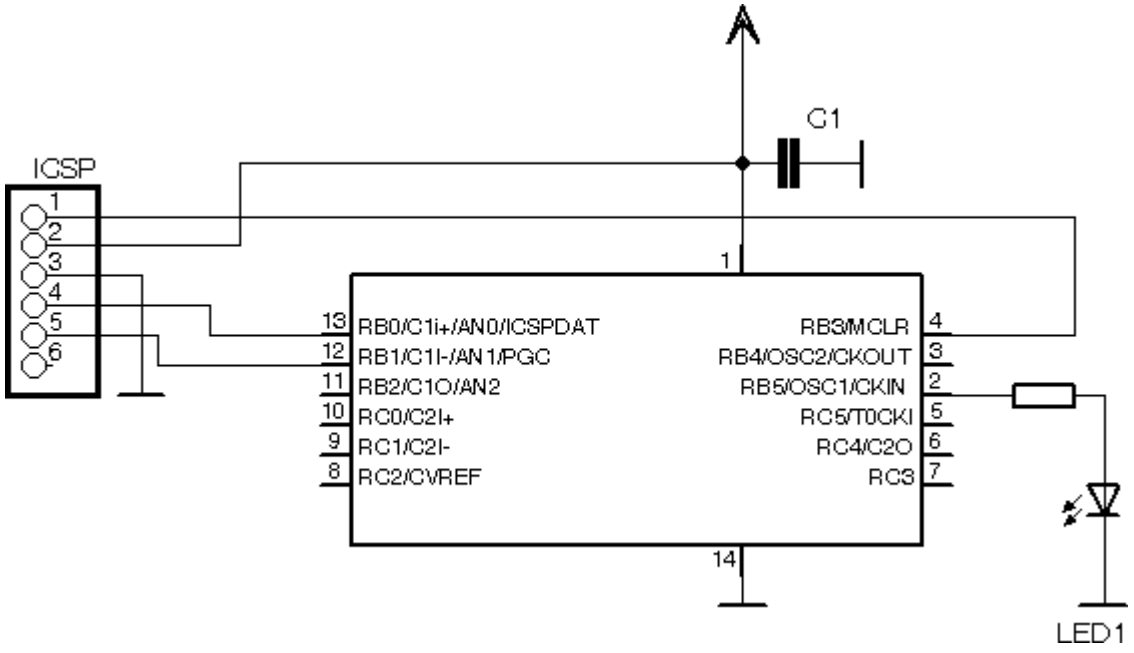
The rest of the source is similar to those already seen. Basically, observe how you deviate only in the few points of difference between the various chips.

Version for 16F526/16F505

In the Baseline family we find not only 6-pin and 8-pin processors, but also with a higher number of I/O, with 14, 18, 20 and even 40 pins.
We can therefore use other Baseline PICs, such as the **16F526** (or even 16F505). The layout on the development board looks like this:



The related wiring diagram:



Connect the LED to the **RB5** pin (but you can also connect it to others, just change the reference in the program).

The same tutorial will be available for Midrange and PIC18F and therefore it will be possible to use other chips, starting from the obsolete 16F84A (which, however, is not a Baseline and that we will see in the chapter dedicated to Midrange).

You will always notice how the source can switch from one processor to another with minimal variations, essentially related to the specific characteristics of the chip, leaving the logic of the program unchanged. In this section, we only deal with Baseline components; two other sections are dedicated to the Midrange and the PIC18F.

If we have seen how the 6-pin 10F chips have a small number of I/Os useful as digital outputs (only 3), we can see instead a 14-pin chip, the **PIC16F526**, of which we find a fairly detailed description here.

It integrates two 6-bit ports (**PORTB** and **PORTC**).

PORTB	**Bit**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	Label	-	-	**RB5**	**RB4**	**RB3**	**RB2**	**RB1**	**RB0**
	pin	-	-						
PORTC	**Bit**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	Label	-	-	**RC5**	**RC4**	**RC3**	**RC2**	**RC1**	**RC0**
	pin	-	-						

Since there are only 6 valid pins in each register and since the registers are 8 bits (which is the width of the data bus), bits 6 and 7 of the registers have no function: reading them, you will get 0 and writing them will have no effect. This allows us to easily control 11 output loads and therefore the 8 LEDs that it makes available.
We said 11 and not 12, since **RB3** has the usual **MCLR** function and can only be used digitally as an input.


Note that Microchip, for processors with more than 8 pins, identifies the I/O blocks not as **GPIO,** but as **PORTx**. So, for example, we won't refer to **GP0**, but to **Px0** or **Rx0.**

The structure of this chip is very similar to what we have seen for the 12F508/9/10/19 already seen; the differences are essentially that a greater amount of resources is available and consequently a greater amount of internal registers will be required. We will then have, for example, a **TRISx** register and a **PORTx** register for each port group.
There are other differences, such as the amount of available memory, RAM, integrated modules and related SFRs, which, for now, we do not consider, because, where necessary, they require specific management.

The treatment of the source is more concise than the one written for the PIC12F519 since we refer to what is written there. This is mainly about the differences.

Since PIC16F526 and PIC16F505 , although not identical, are very similar, both belonging to the Baseline family, a program written for one can also be run on the other without modification (as long as the memory availability is not exceeded). For this reason we can use only one source for both chips, selecting one or the other, depending on availability, with the switch system already seen for 12F508/509:

```
#ifdef___16F526
    LIST p=16F526           ;
    #include <p16F526.inc>
#endif

#ifdef___16F505
    LIST p=16F505           ;
    #include <p16F505.inc>
#endif
```

To switch from one to the other without opening a new project, just select the desired processor from the MPLAB ***Configure/Select Device command line***.
Obviously, the processor for which you compile must be the same as the one to which you will then transfer the executable program obtained.

As for the configuration , it is similar to what we have seen before, with the difference that in PIC16F526 there is the possibility to vary the frequency of the internal oscillator to 4MHz or 8MHz, while in 16F505 there is only the 4MHz frequency.
In addition, 16F526 has EEPROM, which we do not want to protect (**_CPDF_OFF**). On the other hand, the 16F505 has the ability to output the internal clock (Fosc/4) from the RB4/Clockout pin, a function that we do not want to activate (**_IntRC_OSC_RB4EN**).
Since the configurations are different, we will have to resort again to the choice with **#ifdef**.

```
;*****************************************************************
;                           CONFIGURTION
  #ifdef___16F526
;4MHz, no WDT, no CP, no MCLR
  __config _IntRC_OSC & _IOSCFS_4MHz & _WDTE_OFF & _CP_OFF & _CPDF_OFF &
_MCLRE_OFF
 #endif

 #ifdef___16F505
 4MHz, no WDT, no CP, no MCLR
  __config _IntRC_OSC_RB4EN & _WDTE_OFF & _CP_OFF & _MCLRE_OFF
 #endif
```

As far as I/O is concerned, **RB5** is free if an external oscillator is not configured. Although the 16F526 has an ADC module and comparators, they are connected to other pins, so you don't need any special manoeuvres to use this **RB5** (which was chosen for this reason...).

However, it must be considered that we are dealing with **PORTB** and not with **GPIO** :

```
; Reset Initializations
      clrf   PORTB          ; preset port latch a 0
; RB5 come out
      movlw b'00011111'
      tris   PORTB          ; To the Management

mainloop:
      bsf    LED            ; LED ON
```

Basically, observe how the source differs from the previous ones only in the few points of difference between the various chips.

The differences to be applied in the source between one chip and another of the same family essentially concern:

- **the different availability of integrated resources (peripherals and modules)**
- **The number of pins and their organization**
- Any differences in memory equipment

The rest of the instructions, i.e. the logical part of the program, are very likely to remain absolutely identical, since all the PICs of the same family are made according to the same philosophy.

By setting up an orderly job, there are no particular problems in transferring a source from one chip to another, generally with slight changes in the source, which can be reduced to just the definition of the processor and little else.

Extensions

We can propose some variations on the theme:

1. turn on an LED connected to GP2
2. turn on an LED connected to GP3
3. turn on 4 LEDs connected to GP0, GP1, GP2, GP4

If you can't solve the "cases", here are the solutions.

1. Consulting **the 12F519 documentation**, it can be seen that **GP2** assumes by default the **T0CKI input function** of the Timer0.

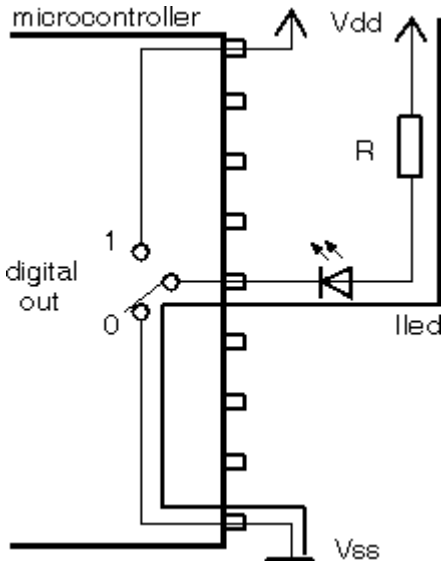
   Therefore, in order to use it, you need to act on the **T0CS** bit of the **OPTION register**.

```
; I/O initializations on reset
; OPTION default '11111111'
          movlw b'11011111'
   ;              1-------  GPWU Disabled
   ;              -1------  GPPU disabled
   ;              --0-----  internal clock, free GP2
   ;              ---1----  Edge
   ;              ----1---  Prescaler to Timer
   ;              -----111  1:256
          OPTION            ; to the OPTION register
```

   Note the use of the special OPTION instruction to transfer the contents of **W** to the **OPTION_REG**, which has similar characteristics to the tic-tac-toe register (not in memory map, read-only, and accessed with a special opcode). The same is true for **the 12F508/509**.

2. **Lighting an LED with GP3 is not possible**, since this pin, if it is not defined as **MCLR**, can take on **the function of digital input, but not output.** As an additional practice, you can then try other pin and LED combinations.

3. Simply set the bits of the TRIS relative to the pins you want as digital outputs to 0 and then set the corresponding GPIO bits to 1



It is also possible to turn on LEDs connected between the pin and the Vdd.

In this case, the LED will light up bringing the pin to level 0 (**current sink**) and will be off when the pin is level 1.

Then the LED power command will be **`bcf LED`**, where the **`bcf`** (*bit clear on file*) instruction brings the corresponding bit to level 0.

Also in this case the maximum current managed by the pin is 25mA.

We have an "inverse" logic:

Logic Level	Voltage at pin	LED
0	Vss	on
1	Vdd	off

To turn on the LED you will need to connect its cathode to the Vss, programming the bit relative to the pin at level 0.

Important Notes

1. Before moving on to the next tutorials, it's a good idea to have a clear idea of what you've done so far.
   If there are any points that are still unclear, it is appropriate to try to clarify them as much as possible before continuing.

2. What is said in one exercise is also reflected in the following ones, being of increasing complexity.
   Since it does not make sense to repeat every time what is detailed in one exercise, this will NOT be repeated in the following ones, since they will have new topics as their theme, on which the explanation will mainly focus.
   So keep this in mind.

3. As far as possible, all sources have been verified and are operational. If your design doesn't work, check your hardware first.

4. It should begin to be clear that, since ICPs are derived from a single design philosophy, the transition from one type to another, especially if within the same family, involves minimal changes to the source.
   Portability is enhanced by the use of symbols instead of absolute values.

12F519 – 1A_519.asm

```
;*****************************************************************
;-----------------------------------------------------------------
;
;    Title          :  Assembly & C Course - Tutorial 1A_519
;                      Turn on an LED connected to the GP5 pin.
;                      The LED lights up when the power is turned on
;                      power supply and remains on as long as this
;                      is present
;    PIC            :  12F519
;    Support        :  MPASM
;    Version        :  1.0
;    Date           :  01-05-2013
;    Hardware ref.  :
;    Author         :  Afg
;
;-----------------------------------------------------------------
;
; Pin use :
;   ---------------------------
;       12F519 @ 8 pin
;
;                    |__\/__|
;              Vdd -|1    8|- Vss
;              GP5 -|2    7|- GP0
;              GP4 -|3    6|- GP1
;         GP3/MCLR -|4    5|- GP2
;                    |______|
;
;    Vdd                 1: ++
;    GP5/OSC1/CLKIN      2: Out LED at Vss (R in series)
;    GP4/OSC2            3:
;    GP3/! MCLR/VPP      4:
;    GP2/T0CKI           5:
;    GP1/ICSPCLK         6:
;    GP0/ICSPDAT         7:
;    Vss                 8: --
;
;*****************************************************************
;=================================================================
;            DEFINITION OF PORT USE
;=================================================================
;
; GPIO map
;| 5 | 4 | 3 | 2 | 1 | 0 |
;|---|---|---|---|---|---|
;| LED |   |   |   |   |   |
;
;#define    GPIO,GP0    ;
;#define    GPIO,GP1    ;
;#define    GPIO,GP2    ;
;#define    GPIO,GP3    ;
;#define    GPIO,GP4    ;
;#define    GPIO,GP5    ; LED between pin
                        and Vss
;
; ################################################################

       LIST      p=12F519         ; Processor definition
       #include <p12F519.inc>
```

```
        Radix     DEC               ; Decimal Root

; ######################################################################
;                           CONFIGURATION
;
; Internal oscillator, no WDT, no CP, pin4=GP3
;
 __config _IntRC_OSC & _IOSCFS_4MHz & _WDTE_OFF & _CP_OFF & _CPDF_OFF &
_MCLRE_OFF

; ######################################################################
;                          RESET ENTRY
;
; movlw valore_calibrazione   First Intrinsic Instruction

; Reset Vector
RESET_VECTOR   ORG     0x00

 ; MOWF Internal Oscillator
        Calibration   OSCCAL

; ######################################################################
;                         MAIN PROGRAM
MAIN:
; I/O initializations on reset
        CLRF    GPIO            ; GPIO preset latch to 0

; Assign GP5 Digital Output Function
; TRISGPIO        xx011111     GP5 out
        movlw   b'11011111'
        Tris    GPIO            ; To the Management Register

; lights up LEDs bringing the GP5 bsf pin to
        level 1 GPIO,GP5

; Execution Lock - Goto Closed
        Loop    $

;**********************************************************************
;                             THE END
; End of source
        END
```

```
PAGE

; Commentary text for documentation
;*****************************************************************
; ================================================================
; =                          DOCUMENTATION                       =
;================================================================
;*****************************************************************
; CONFIG Options
;*****************************************************************
;
; Oscillator
;  ----------------------------------------------------------------
;_FOSC_LP          ; Osc. External LP with 18ms DRT
;_LP_OSC           ; alias
;_FOSC_XT          ; Osc. External XP with 18ms DRT
;_XT_OSC           ; alias
;_FOSC_INTRC       ; Osc. internal with 1,125ms DRT
;_IntRC_OSC        ; alias
;_FOSC_EXTRC       ; Osc. External RC with 1.125ms DRT
;_ExtRC_OSC        ; alias

; Internal Oscillator Frequency
;  ----------------------------------------------------------------
;_IOSCFS_4MHz      ; 4 MHz INTOSC
;_IOSCFS_8MHz      ; 8 MHz INTOSC

; Watchdog
;  ----------------------------------------------------------------
;_WDTE_ON          ; qualified
;_WDTE_OFF         ; disabled

; Program Memory Protection
;  ----------------------------------------------------------------
;_CP_ON            ; Security Enabled
;_CP_OFF           ; Protection Disabled


; Pin MCLR
;  ----------------------------------------------------------------
;_MCLRE_ON         ; GP3/MCLR as MCLR
;_MCLRE_OFF        ; GP3/MCLR as GP3

; Securing the EEPROM
;  ----------------------------------------------------------------
;_CPDF_ON          ; Security Enabled
;_CPDF_OFF         ; Protection Disabled

;*****************************************************************
;                            RAM                                 *
;*****************************************************************
; On two benches
; 07-0Fh (27-2Fh) 9 bytes accessible from both banks
; 10-1Fh          16 bytes accessible only from Bank0
; 30-3Fh          16 bytes accessible only from Bank1

; Bank Switch: STATUS,PA0
```

```
;*****************************************************************
;                        SFR PRINCIPALS                          *
;*****************************************************************
; On two benches
; STATUS, INDF, PCL, FSR accessible from both counters
; GPIO, OSCCAL, TMR0     accessible only from Bank0
; EECON, EEDATA, EEADR   only accessible from Bank1

;+++++++++++++++++++++++++++++++++++++++++++++++++++
; Registers in Bank 0
;
;OPTION_REG - Options & TIMER0 - writable with optino
;-----------------------------------------------------
 ; to the ROP
            '11111111'
 OPTIONVAL   EQU B'11010111'
         ;          1-------     GPWU Disabled
         ;          -1------     GPPU disabled
         ;          --0-----     internal clock - GP2 available
         ;          ---1----     Foul
         ;          ----0---     Prescaler to Timer
         ;          -----111     1:256
         ;
         ; NOT_GPWU B7 : enable wake-up pin on change
         ;                1= disabled
         ;                0= enabled
         ; NOT_GPPU b6 : Enable Weak Pull-ups bit
         ;                1= disabled
         ;                0= enabled
         ; TOCS      b5 : 1= external TMR0 clock from GP2
         ;                0= internal clock
         ; TOSE      b4 : 1= GP2 descending front (b5=1)
         ;                0= GP2 uphill front       (b5=1)
         ; PSA       b3 : 1= Watchdog predivisor
         ;                0= predictor to TMR0
         ; PS2/PS0   B2/B0 Value of the Predictor
         ;                  Watchdog   Timer0
         ;          000 = 1/1         1/2 mm
         ;          001 = 1/2         1/4 mm
         ;          010 = 1/4         1/8
         ;          011 = 1/8         1/16
         ;          100 = 1/16        1/32
         ;          101 = 1/32        1/64
         ;          110 = 1/64        1/128
         ;          111 = 1/128       1/256

; OSCCAL - Internal Oscillator Calibration - bank 0
;  ----------------------------------------------------------------------------
OSCCALDEF EQU     B'000000000' ; Intermediate value
        ; CAL6:0 b7:1 01111 = Maximum frequency
        ;              00000 = Center frequency
        ;              10000 = Minimum frequency
        ; Res.    b0           reserved


; Order of precedence of PORT functions
; # | GP0 | GP1 | GP2 | GP3 | GP4 | GP5 |
;-----------------|------|------|-------------|
; 1 | TRIS | TRIS | T0CKI|I/MCLR| TRIS | TRIS |
; 3 |   - |    - | TRIS | -     |   - |    - |
```

```
; GPIO - GPIO Register
;  ..........................................
; to the ROP       '------ xxxx'

; TRISGPIO - GPIO Direction
;  ..............................................
; to the ROP       '--111111'
TRISValue EQU    B'--011111'
          ;          ||||||||____TRISGP0 in
          ;          |||||||_____TRISGP1 in
          ;          ||||||______TRISGP2 in
          ;          |||||_______TRISGP3 only in
          ;          ||||________TRISGP4 in
          ;          |||_________TRISGP5 out
          ;          ||__________Not used

;++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; Registers in Bank 1
;

; EECON - EEPROM Control - bank 1
;  ..........................................................
        ; reserved B7:5-------: reserved
        ; UNCONSTRAINED     B4 --------: Force Erase
        ; WRERR     B3 --------: Reset Error
        ; WREN      B2 --------: Enable Writing
        ; WR        B1 --------: Writing
        ; RD        B0 --------: Reading

; EEDATA - EEPROM Data Registry - bank 1
;  ..................................................................

; EEADR - EEPROM Address Registry - bank 1
;  ..........................................................................
```

12F508/509 - 1A_5089.asm

```
;*****************************************************************
;-----------------------------------------------------------------
;
;    Title          :  Assembly & C - Tutorial 1A_5089
;                      Turn on an LED connected to the GP5 pin.
;                      The LED lights up when the power is turned on
;                      power supply and remains on as long as this
;                      is present
;
;    PIC            :  Code 12F508/509
;    Support        :  MPASM
;    Version        :  1.0
;    Date           :  01-05-2013
;    Hardware ref.  :
;    Author         :  Afg
;
;-----------------------------------------------------------------
;
; Pin use :
;   ----------------------------
;       12F508/509 @ 8 pin
;
;                    |‾‾\/‾‾|
;              Vdd -|1    8|- Vss
;              GP5 -|2    7|- GP0
;              GP4 -|3    6|- GP1
;         GP3/MCLR -|4    5|- GP2
;                    |______|
;
;    Vdd                 1: ++
;    GP5/OSC1/CLKIN      2: Out LED at Vss (R in series)
;    GP4/OSC2            3:
;    GP3/! MCLR/VPP      4:
;    GP2/T0CKI           5:
;    GP1/ICSPCLK         6:
;    GP0/ICSPDAT         7:
;    Vss                 8: --
;
;*****************************************************************
;=================================================================
;             DEFINITION OF PORT USE
;=================================================================
;
; GPIO map
;| 5 | 4 | 3 | 2 | 1 | 0 |
;|---|---|---|---|---|---|
;| LED |   |   |   |   |   |
;
;#define    GPIO,GP0    ;
;#define    GPIO,GP1    ;
;#define    GPIO,GP2    ;
;#define    GPIO,GP3    ;
;#define    GPIO,GP4    ;
;#define    GPIO,GP5    ; LED between pin 2
                        and Vss
;
; ###############################################################
 Choice of #ifdef
 processor      12F509
        LIST      p=12F509         ; Processor Definition
        #include <p12F509.inc>
```

```
#endif
#ifdef   12F508

        LIST      p=12F508          ; Processor definition
        #include <p12F508.inc>
#endif

        Radix     DEC               ; Decimal Root

; ##########################################################################
;                           CONFIGURATION
;
; Internal oscillator, no WDT, no CP, pin4=GP3
 __config _IntRC_OSC & _WDT_OFF & _CP_OFF & _MCLRE_ON

; ##########################################################################


; ##########################################################################
;                          RESET ENTRY
; movlw valore_calibrazione   First Intrinsic Instruction

; Reset Vector
        ORG     0x00

 ; MOWF Internal Oscillator
        Calibration   OSCCAL


; ##########################################################################
;                         MAIN PROGRAM
;
 MAIN:
; Reset Initializations
        CLRF    GPIO            ; GPIO preset latch to 0


; Assign GP5 Digital Output Function
; TRISGPIO        xx011111     GP5 out
        movlw   b'11011111'
        Tris    GPIO            ; To the Management Register


; lights up LEDs bringing the GP5 bsf pin to
        level 1 GPIO,GP5

; Execution Lock - Goto Closed
        Loop    $


;**************************************************************************
;                             THE END

; End of source
        END
```

```
 PAGE
; Commentary text for documentation
;*****************************************************************
; ===============================================================
; =                          DOCUMENTATION                      =
;===============================================================
;*****************************************************************
; CONFIG Options
;*****************************************************************
;
; Oscillator
;  ------------------------------------------------------------
;_OSC_LP           ; Osc. External LP with 18ms DRT
;_LP_OSC           ; alias
;_OSC_XT           ; Osc. External XP with 18ms DRT
;_XT_OSC           ; alias
;_OSC_IntRC        ; Osc. 4MHz internal with 1.125ms DRT
;_IntRC_OSC        ; alias
;_OSC_ExtRC        ; Osc. External RC with 1.125ms DRT
;_ExtRC_OSC        ; alias

; Watchdog
;  ------------------------------------------------------------
;_WDT_ON                  qualified
;_WDT_OFF                 disabled

; Securing the EEPROM
;  ------------------------------------------------------------
;_CPD_ON                  Security Enabled
;_CPD_OFF                 Protection Disabled

; Pin MCLR
;  ------------------------------------------------------------
;_MCLRE_ON                GP3/MCLR as MCLR
;_MCLRE_OFF               GP3/MCLR as GP3

;*****************************************************************
;                           RAM                                  *
;*****************************************************************
; 12F508 single bench
; 07-1Fh   25 bytes
;
; 12F509 Two Benches
; 07-0Fh (27-2Fh) 9 bytes accessible from both banks
; 10-1Fh          16 bytes accessible only from Bank0
; 30-3Fh          16 bytes accessible only from Bank1
; Bank Switch: STATUS,PA0

;*****************************************************************
;                         SFR PRINCIPALS                         *
;*****************************************************************
; SFR on a single bench
;
;OPTION_REG - Options & TIMER0 - writable with optino
;-----------------------------------------------------
 ; to the          '11111111'
 ROP          EQU B'11010111'
 OPTIONVAL
         ;          1-------     GPWU Disabled
         ;          -1------     GPPU disabled
         ;          --0-----     Internal Clock
```

```
          ;           ---1----     done
          ;           ----0---     Prescaler to Timer
          ;           -----111     1:256
          ;
          ; NOT_GPWU B7 : enable wake-up pin on ch'ange
          ;                1= disabled
          ;                0= enabled
          ; NOT_GPPU b6 : Enable Weak Pull-ups bit
          ;                1= disabled
          ;                0= enabled
          ; TOCS      b5 : 1= external TMR0 clock from GP2
          ;                0= internal clock
          ; TOSE      b4 : 1= GP2 descending front (b5=1)
          ;                0= GP2 uphill front       (b5=1)
          ; PSA       b3 : 1= Watchdog predivisor
          ;                0= predictor to TMR0
          ; PS2/PS0   B2/B0 Value of the Predictor
          ;                   Watchdog   Timer0
          ;           000 = 1/1          1/2 mm
          ;           001 = 1/2          1/4 mm
          ;           010 = 1/4          1/8
          ;           011 = 1/8          1/16
          ;           100 = 1/16         1/32
          ;           101 = 1/32         1/64
          ;           110 = 1/64         1/128
          ;           111 = 1/128        1/256


; OSCCAL - Internal Oscillator Calibration
;  ..........................................................................
OSCCALDEF EQU     B'000000000' ; Intermediate value
        ; CAL6:0    b7:1 : 01111 = Maximum frequency
        ;                   00000 = Center frequency
        ;                   10000 = Minimum frequency
        ; Reserved B0


; Order of precedence of PORT functions
; # | GP0 | GP1 | GP2 | GP3 | GP4 | GP5 |
;-----------------|------|------|-------------|
; 1 | TRIS | TRIS | T0CKI|I/MCLR| TRIS | TRIS |
; 3 |   - |    - | TRIS | -     |   - |    - |


; GPIO - GPIO Register
;  ..........................................
; to the ROP       '----- xxxx'

; TRISGPIO - GPIO Direction
;  ..................................................
; to the ROP       '--111111'
TRISValue EQU    B'--011111'
          ;         |||||||____TRISGP0 in
          ;         ||||||_____TRISGP1 in
          ;         |||||______TRISGP2 in
          ;         ||||_______TRISGP3 only in
          ;         |||________TRISGP4 in
          ;         ||_________TRISGP5 out
          ;         |__________Not used
```

10F200/202 - 1A_20x.asm

```
;*****************************************************************
;-----------------------------------------------------------------
;
;    Title          : Assembly & C Course - Tutorial 1A_20x
;                     Turn on an LED connected to the GP2 pin.
;                     The LED lights up when the power is turned on
;                     power supply and remains on as long as this
;                     is present
;    PIC            : 10F200/2
;    Support        : MPASM
;    Version        : 1.0
;    Date           : 01-05-2013
;    Hardware ref.  :
;    Author         :Afg
;
;
; Pin use :
;   ________________
;       10F200/202 @ 8 pin DIP       10F200/202 @ 6-pin SOT-23
;
;
;                  |__\/__|                  *______|
;              NC -|1    8|- GP3         GP0 -|1    6|- GP3
;             Vdd -|2    7|- Vss         Vss -|2    5|- Vdd
;             GP2 -|3    6|- NC          GP1 -|3    4|- GP2
;             GP1 -|4    5|- GP0               |______|
;                  |______|
;
;                       DIP  SOT
;    NC                  1:     Nc
;    Vdd                 2:  5: ++
;    GP2/T0CKI/FOSC4     3:  4:
;    GP1/ICSPCLK         4:  3:
;    GP0/ICSPDAT         5:  1: Out LED to Vss
;    NC                  6:     Nc
;    Vss                 7:  2: --
;    GP3/MCLR/VPP        8:  6:
;
;*****************************************************************
;=================================================================
;            DEFINITION OF PORT USE
;=================================================================
; GPIO map
; | 3 | 2 | 1 | 0 |
; |---|---|---|---|
; | in  | LED |     |     |
;
;#define    GPIO,GP0    ;
;#define    GPIO,GP1    ;
;#define    GPIO,GP2    ; LED between pin
                        and Vss
;#define    GPIO,GP3    ; Input only
```

```
;
; ######################################################################
; Choice of processor

 #ifdef__10F200
        LIST      p=10F200     ; Processor Definition
        #include <p10F200.inc>
 #endif

 #ifdef__10F202
        LIST      p=10F202     ; Processor Definition
       #include <p10F202.inc>
 #endif

        Radix     DEC          ; Decimal Numbers Defaults

; ######################################################################
;                           CONFIGURATION
;
; No WDT, no CP, pin4=GP3
  __config _CP_OFF & _MCLRE_OFF & _WDT_OFF


; ######################################################################
;                          RESET ENTRY
; Reset Vector
RESET_VECTOR   ORG     0x00

 ; MOWF Internal Oscillator
        Calibration OSCCAL

; ######################################################################
;                         MAIN PROGRAM
;
MAIN:
; Reset Initializations
        CLRF    GPIO            ; GPIO preset latch to 0

; disable T0CKI function from GP2
        movlw   b'11011111'
        OPTION                  ; to the OPTION_REG register

; TRISGPIO        --111011     GP2 out
        movlw   b'11111011'
        TRIS    GPIO            ; To the      direction
                                Registry

; Lights up LEDs
        Bsf     GPIO,GP2

; Execution Lock - Goto Closed
        Loop    $

;*********************************************************************
;                             THE END
        END
```

```
 PAGE
;*****************************************************************
; ===============================================================
; =                          DOCUMENTATION                      =
;===============================================================
;*****************************************************************
; CONFIG Options
;
; Program Protection
;  ____________________________
;_CP_ON                  Security Enabled
;_CP_OFF                 Protection Disabled

; Pin MCLR
;  __________
;_MCLRE_ON               GP3/MCLR as MCLR;
;_MCLRE_OFF              GP3/MCLR as GP3

; Watchdog
;  _________
;_WDT_ON                 qualified
;_WDT_OFF                disabled
;_WDTE_ON                alias
;_WDTE_OFF

; Internal Oscillator
;  ____________________
;_OSC_IntRC              Single option - no need to declare
;_IntRC_OSC              alias

;*****************************************************************
;                         SFR PRINCIPALS
;*****************************************************************
; SFR without benches
;
; Example of an initialization mask and
; Brief description of the registers
;
;OPTION_REG - Wake pull-up & TIMER0
;----------------------------------
; to the ROP
            '11111111'
OPTIONVAL   EQU B'11010111'
        ;           1-------     GPWU Disabled
        ;           -1------     GPPU disabled
        ;           --0-----     Internal Clock
        ;           ---1----     done

        ;           ----0---     Prescaler to Timer
        ;           -----111     1:256
;
        ; NOT_GPWU B7 : enable wake-up pin on change
        ;                1= disabled
        ;                0= enabled
        ; NOT_GPPU b6 : Enable Weak Pull-ups bit
        ;                1= disabled
        ;                0= enabled
```

```
; TOCS      b5 : 1= external TMR0 clock from GP2
```

```
        ;                   0= internal clock
        ; TOSE      b4 : 1= GP2 descending front (b5=1)
        ;                   0= GP2 uphill front       (b5=1)
        ; PSA       b3 : 1= Watchdog predivisor
        ;                   0= predictor to TMR0

        ; PS2/PS0   B2/B0 Value of the Predictor
        ;                      Watchdog   Timer0
        ;           000 =  1/1         1/2
        ;                                 mm
        ;           001 =  1/2         1/4
        ;                                 mm
        ;           010 =  1/4         1/8
        ;           011 =  1/8         1/16
        ;           100 =  1/16        1/32
        ;           101 =  1/32        1/64
        ;           110 =  1/64        1/128
        ;           111 =  1/128       1/256


; OSCCAL - Internal Oscillator Calibration
;  ____________________________________________

; to the           '11111110'
ROP        EQU    B'000000000' ; Intermediate value
OSCCALDEF

        ; CAL6:0    B7-1  : 0111111 = Maximum frequency
        ;                    0000000 = Center frequency
        ;                    1000000 = Minimum frequency
        ; FOSC4     b0    : 1= INTOSC/4 on GP2
        ;                    0= GP2/T0CKI/COUT


; Order of precedence of PORT functions
; # | GP0 | GP1 | GP2 | GP3 |
;---|------|------|------|-------|
; 1 | TRIS | TRIS | T0CKI|I/MCLR |
; 3 |   - |    - | TRIS | -      |

; GPIO - PORT Registry
;----------------------
; to the ROP      '------xxxx'

; TRISGPIO - GPIO Direction
;  ____________________________
; to the ROP      '------1111'
DIRPORT    EQU   B'------1110'
          ;         | |||||_____TRISGP0 in
          ;         | ||||______TRISGP1 in
          ;         | |||_______TRISGP2 in
          ;         | ||________TRISGP3 only in
          ;         |__|________-
```

16F526/505 - 1A_526.asm

```
;*****************************************************************
;-----------------------------------------------------------------
;
;    Title          : Assembly & C Course - Tutorial 1A_526
;                     Turn on an LED connected to the GP5 pin.
;                     The LED lights up when the power is turned on
;                     power supply and remains on as long as this
;                     is present
;
;    PIC            : 16F526
;    Support        : MPASM
;    Version        : 1.0
;    Date           : 01-05-2013
;    Hardware ref.  :
;    Author         :Afg
;
;-----------------------------------------------------------------
;
;  Pin use :
;  ________________
;       16F505 - 16F526 @ 14 pin
;
;                   |‾‾\/‾‾|
;              Vdd -|1   14|- Vss
;              RB5 -|2   13|- RB0
;              RB4 -|3   12|- RB1
;         RB3/MCLR -|4   11|- RB22
;              RC5 -|5   10|- RC0
;              RC4 -|6    9|- RC1
;              RC3 -|7    8|- RC2
;                   |______|
;
;    Vdd                       1: ++
;    RB5/OSC1/CLKIN            2: Out LED to Vss
;    RB4/OSC2/CLKOUT           3:
;    RB3/! MCLR/VPP            4: MCLR
;    RC5/T0CKI                 5:
;    RC4/[C2OUT]               6:
;    RC3                       7:
;    RC2/[Cvref]               8:
;    RC1/[C2IN-]               9:
;    RC0/[C2IN+]              10:
;    RB2/[C1OUT/AN2]          11:
;    RB1/[C1IN-/AN1/]ICSPC    12:
;    RB0/[C1IN+/AN0/]ICSPD    13:
;    Vss                      14: --
;
;    [ ] 16F526
; ###############################################################
;                          PROCESSOR SELECTION
 #ifdef___16F526
```

```
      LIST p=16F526           ;
      #include <p16F526.inc>
  #endif

  #ifdef__16F505
      LIST p=16F505           ;
      #include <p16F505.inc>
 #endif

; ##########################################################################
;                           CONFIGURATION
;
 #ifdef__16F526
; Internal Oscillator, 4MHz, No WDT, No CP, No MCLR
 __config _IntRC_OSC & _IOSCFS_4MHz & _WDTE_OFF & _CP_OFF & _CPDF_OFF
& MCLRE_OFF
 #endif

 #ifdef__16F505
; Internal Oscillator, 4MHz, No WDT, No CP, No MCLR
 __config _IntRC_OSC_RB4EN & _WDT_OFF & _CP_OFF & _MCLRE_OFF
 #endif


 PAGE

; ##########################################################################
;                          RESET ENTRY
;
; Reset Vector
        ORG     0x00

 ; MOWF Internal Oscillator
        Calibration OSCCAL

; ##########################################################################
;                         MAIN PROGRAM
Main:
; Reset Initializations
        CLRF    PORTB            ; Preset Port Latch to 0

; RB5 come out
        movlw   b'00011111'
        Tris    PORTB           ; To the Management Register

Mainloop:
        Bsf     PORTB,RB5       ; LED On

; Execution Lock - Goto Closed
        Loop    $

;**************************************************************************
;                            THE END
        END
```

```
 PAGE
;*****************************************************************
; ===============================================================
; =                          DOCUMENTATION                      =
;===============================================================
; PIC16F526

;*****************************************************************
; CONFIG Options
;*****************************************************************
;
; Oscillator
;  ____________________________________________________
;_FOSC_LP            ; Osc. External LP with 18ms DRT
;_LP_OSC             ; alias
;_FOSC_XT            ; Osc. External XP with 18ms DRT
;_XT_OSC             ; alias
;_FOSC_HS            ; HS oscillator and 18 ms DRT
;_HS_OSC             ; alias
;_FOSC_EC            ; EC oscillator with RB4 and 1 ms DRT
;_EC_OSC             ; alias
;_FOSC_INTRC_RB4     ; INTRC with RB4 function and 1 ms DRT
;_IntRC_OSC_RB4      ; alias
;_FOSC_INTRC_CLKOUT ; INTRC with CLKOUT function and 1 ms DRT
;_IntRC_OSC_CLKOUT ; alias
;_FOSC_ExtRC_RB4     ; EXTRC with RB4 function and 1 ms DRT
;_ExtRC_OSC_RB4      ; alias
;_FOSC_ExtRC_CLKOUT ; EXTRC with CLKOUT function and 1 ms DRT
;_ExtRC_OSC_CLKOUT ; alias

; Internal Oscillator Frequency
;  ____________________________________________________
;_IOSCFS_4MHz      ; 4 MHz INTOSC
;_IOSCFS_8MHz      ; 8 MHz INTOSC

; Watchdog
;  ____________________________________________________
;_WDTE_ON          ; qualified
;_WDTE_OFF         ; disabled

; Program Memory Protection
;  ____________________________________________________
;_CP_ON            ; Security Enabled
;_CP_OFF           ; Protection Disabled

; Pin MCLR
;  ____________________________________________________
;_MCLRE_ON         ; GP3/MCLR as MCLR
;_MCLRE_OFF        ; GP3/MCLR as GP3

; Securing the EEPROM
;  ____________________________________________________
;_CPDF_ON          ; Security Enabled
;_CPDF_OFF         ; Protection Disabled
```

```
;*****************************************************************
;                          SFR PRINCIPALS                        *
;*****************************************************************
; SFR on two benches
; STATUS, INDF, PCL, FSR accessible from both counters
; GPIO, OSCCAL, TMR0     accessible only from Bank0
; EECON, EEDATA, EEADR   only accessible from Bank1

;+++++++++++++++++++++++++++++++++++++++++++++++++++++
; Registers in Bank 0
;
;OPTION_REG - Options & TIMER0 - writable with optino
;-----------------------------------------------------
; to the ROP
            '11111111'
OPTIONVAL   EQU B'11010111'
          ;          1-------     GPWU Disabled
          ;          -1------     GPPU disabled
          ;          --0-----     Internal Clock
          ;          ---1----     Foul
          ;          ----0---     Prescaler to Timer
          ;          -----111     1:256
          ;
          ; NOT_GPWU B7 : enable wake-up pin on ch'ange
          ;                1= disabled
          ;                0= enabled
          ; NOT_GPPU b6 : Enable Weak Pull-ups bit
          ;                1= disabled
          ;                0= enabled
          ; TOCS      b5 : 1= external TMR0 clock from GP2
          ;                0= internal clock
          ; TOSE      b4 : 1= GP2 descending front (b5=1)
          ;                0= GP2 uphill front       (b5=1)
          ; PSA       b3 : 1= Watchdog predivisor
          ;                0= predictor to TMR0
          ; PS2/PS0   B2/B0 Value of the Predictor
          ;                   Watchdog   Timer0
          ;           000 =  1/1        1/2
                                        mm
          ;           001 =  1/2        1/4
                                        mm
          ;           010 =  1/4        1/8
          ;           011 =  1/8        1/16
          ;           100 =  1/16       1/32
          ;           101 =  1/32       1/64
          ;           110 =  1/64       1/128
          ;           111 =  1/128      1/256


; OSCCAL - Internal Oscillator Calibration - bank 0
;  ____________________________________________________
OSCCALDEF EQU     B'000000000' ; Intermediate value
        ; CAL6:0    b7:1 : 01111 = Maximum frequency
        ;                   00000 = Center frequency
        ;                   10000 = Minimum frequency
        ; Reserved B0
```

```
; Order of precedence of PORT functions
```

```
; PORTB
; # | RB5 | RB4 | RB3 | RB2 | RB1 | RB0 |
;-----------------|------|------|-------------|
; 1 | TRIS | TRIS | TRIS | C1OUT| C1IN-| C1IN+|
; 2 |      |      |      | AN2  | AN1  | AN0  |
; 3 |      |      |      | TRIS | TRIS | TRIS |
;   | OSC | OSC | MCLR |       |      |      | by config



;P ORTC
; # | RC5 | RC4 | RC3 | RC2 | RC1 | RC0 |
;-----------------|------|------|-------------|
; 1 | T0CKI| C2OUT| TRIS | CVref| C2IN-| C2IN+|
; 2 | TRIS | TRIS |      | TRIS | TRIS | TRIS |



;++++++++++++++++++++++++++++++++++++++++++++++++++++++++
; Registers in Bank 1
;
; EECON - EEPROM Control - bank 1
;---------------------------------
        ; reserved B7:5-------: reserved
        ; UNCONSTRAINED  B4 --------: Force Erase
        ; WRERR    B3 --------: Reset Error
        ; WREN     B2 --------: Enable Writing
        ; WR       B1 --------: Writing
        ; RD       B0 --------: Reading


; EEDATA - EEPROM Data Registry - bank 1
;  ____________________________________________

; EEADR - EEPROM Address Registry - bank 1
;  ____________________________________________
```

```
PAGE
;*****************************************************************
; ===============================================================
; =                        DOCUMENTATION                        =
;===============================================================
; PIC16F505

;*****************************************************************
; CONFIG Options
;*****************************************************************
;
; Oscillator
;  ____________________________________________
;_OSC_LP               ; Osc. External LP
;_LP_OSC               ; alias
;_OSC_XT               ; Osc. External XP
;_XT_OSC               ; alias
;_HS_OSC               ; Osc. External HS
;_OSC_HS               ; alias
;_OSC_EC               ; EC oscillator/RB4 function
;_EC_RB4EN             ; alias
;_OSC_IntRC_RB4EN      ; alias
;_IntRC_OSC_RB4EN      ; alias
;_OSC_IntRC_CLKOUTEN ; Internal RC oscillator/CLKOUT
;_IntRC_OSC_CLKOUTEN ; alias
;_OSC_ExtRC_RB4EN      ; External RC, RB4 function
;_ExtRC_OSC_RB4EN      ; alias
;_OSC_ExtRC_CLKOUTEN ; External RC oscillator/CLKOUT
;_ExtRC_OSC_CLKOUTEN ; alias

; Watchdog
;  ____________________________________________
;_WDT_ON          ; qualified
;_WDT_OFF         ; disabled

; Program Memory Protection
;  ____________________________________________
;_CP_ON            ; Security Enabled
;_CP_OFF           ; Protection Disabled

; Pin MCLR
;  ____________________________________________
;_MCLRE_ON         ; GP3/MCLR as MCLR
;_MCLRE_OFF        ; GP3/MCLR as GP3


;*****************************************************************
;                          SFR PRINCIPALS                        *
;*****************************************************************
; RAM on 4 banks
 SFRs accessible from any bench
```

```
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++
;
;OPTION_REG - Options & TIMER0 - writable with option
;-------------------------------------------------------
; to the ROP
            '11111111'
OPTIONVAL   EQU B'11010111'
         ;          1-------     GPWU Disabled
         ;          -1------     GPPU disabled
         ;          --0-----     Internal Clock
         ;          ---1----     Foul
         ;          ----0---     Prescaler to Timer
         ;          -----111     1:256
         ;
         ; NOT_GPWU B7 : enable wake-up pin on ch'ange
         ;                1= disabled
         ;                0= enabled
         ; NOT_GPPU b6 : Enable Weak Pull-ups bit
         ;                1= disabled
         ;                0= enabled
         ; TOCS      b5 : 1= external TMR0 clock from GP2
         ;                0= internal clock
         ; TOSE      b4 : 1= GP2 descending front (b5=1)
         ;                0= GP2 uphill front       (b5=1)
         ; PSA       b3 : 1= Watchdog predivisor
         ;                0= predictor to TMR0
         ; PS2/PS0   B2/B0 Value of the Predictor
         ;                   Watchdog   Timer0
         ;           000 =  1/1        1/2
         ;           001 =  1/2        1/4
         ;           010 =  1/4        1/8
         ;           011 =  1/8        1/16
         ;           100 =  1/16       1/32
         ;           101 =  1/32       1/64
         ;           110 =  1/64       1/128
         ;           111 =  1/128      1/256


; OSCCAL - Internal Oscillator Calibration - bank 0
;  ____________________________________________________
OSCCALDEF EQU     B'000000000' ; Intermediate value
        ; CAL6:0    b7:1 : 01111 = Maximum frequency
        ;                   00000 = Center frequency
        ;                   10000 = Minimum frequency
        ; Reserved B0


; Order of precedence of PORT functions
; PORTB
; # | RB5 | RB4 | RB3 | RB2 | RB1 | RB0 |
;-----------------|------|------|-------------|
; 1 | TRIS | TRIS | TRIS | TRIS | TRIS | TRIS |
;   | OSC | OSC | MCLR | -      |   - |    - | by config
```

```
;P ORTC
; # | RC5 | RC4 | RC3 | RC2 | RC1 | RC0 |
;-----------------|------|------|-------------|
; 1 |T0CKI | TRIS | TRIS | TRIS | TRIS | TRIS |
; 2 | TRIS |   - |       | -    |   - |    - |
```