Knowledge of baseline

Assembly - Exploiting the possibilities of I/O

We have seen in the previous tutorials how it is possible to control a pin of the microcontroller as a digital output and with this turn an LED on and off.

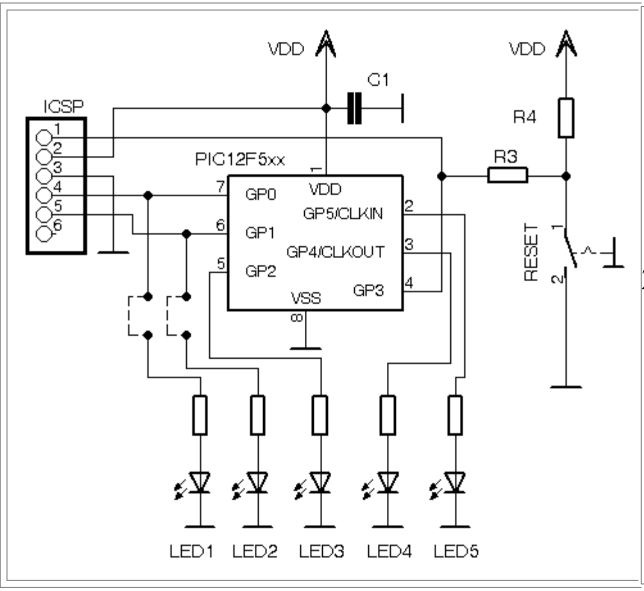
The possibilities offered to the user should be obvious; here we see one: that of controlling several LEDs in sequence.

It is the basis of a large number of practical applications. Instead of LEDs, using a **suitable buffer**, it will be possible to connect relays and other actuators that allow loads to be switched on and off in timed sequences (sequencers).

What we want to achieve

The purpose of the exercise is to control some LEDs sequentially through the microcontroller.

We have fun for a moment with some light plays, using the usual LEDs and verifying how it is possible to easily obtain interesting results.
We always use the **12F519** (but any PIC is fine, with the appropriate adaptations to the source) In addition, variations for 12F508/509 and 16F505/526 will be proposed.
For this tutorial we do not consider the 10F2xx micro PIC which, having only 3 output pins, allow few variations (but they can be sufficient for particular uses and, as in the previous exercises, the source can be adapted without effort).



The wiring diagram is the same as in exercise 1, to which we add:

1. an LED with relative resistance on all pins configurable as digital outputs (`GP0/1/2/4/5`)
2. the reset button with resistors R4 (10k) and R3 (1k).

The button will be used to highlight the function of the MCLR pin.

R4 is the pull-up and R3 is used to isolate the button from the ICSP connection.

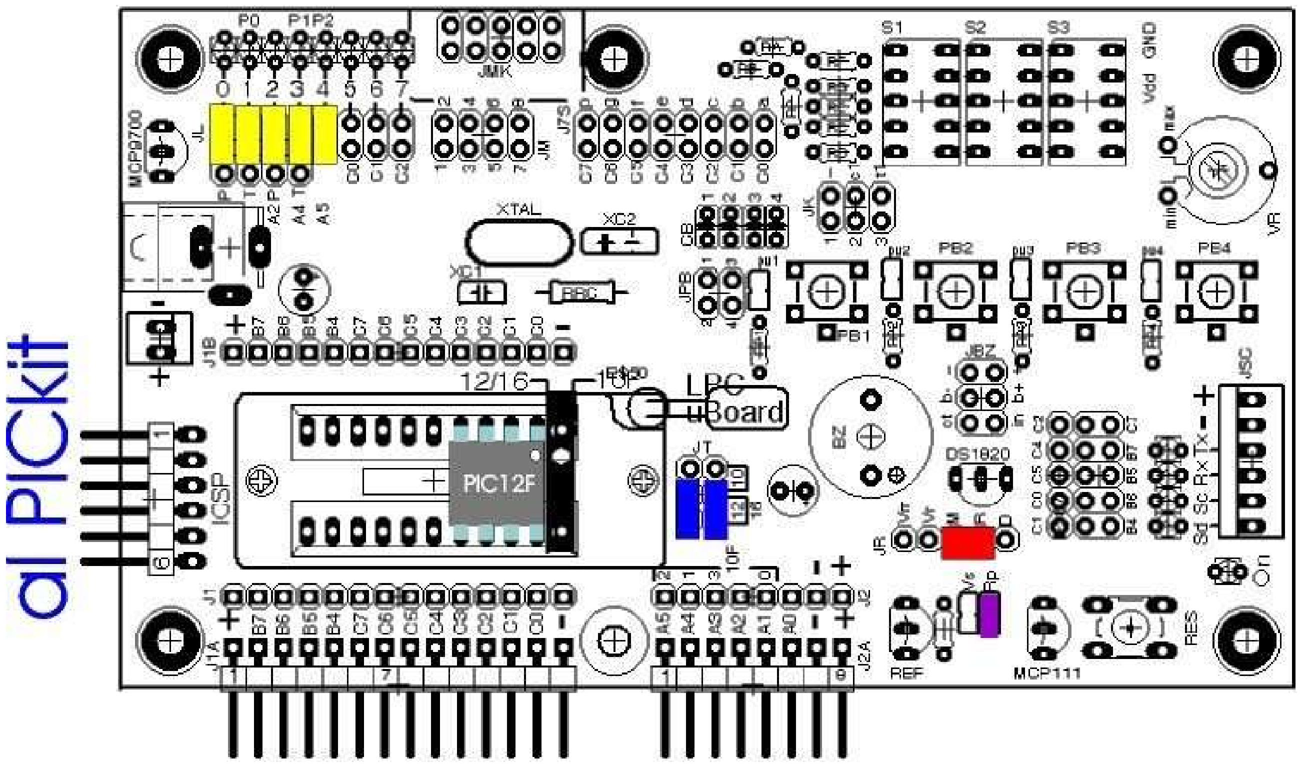
Note that LEDs 1 and 2, which depend on `GP0`/`GP1`, are jumper isolable. This is necessary since the two pins are used by the **ICSP connection** for the clock and data during the

Writing Program Memory.
The signals of these two lines must be basically free of interference elements.

If, as in this case, they are also used to control a load, it must be negligible in terms of absorption and added capacity, so as not to deform the signals of the **ICSP** connection. So it makes sense to provide jumpers to disconnect this load when programming the chip. In the case of the **LPCuB**, the LEDs are of the very low current type (about 2mA), which allows them to be kept on even during programming; at this stage you will be able to observe a rapid flashing. If you use different circuits, with LEDs that draw higher currents, as a precaution it is better to insert the jumpers as shown in the diagram above.

These links are already present on the demo board, while if we use a breadboard they must be done once and for all, since this basic scheme will be used in most of the other tutorials.

For the **LPCuB**:



As mentioned at the beginning, the PIC10F can also be used without problems for this experience, but, since they only have 3 pins useful as digital outputs, the result would be less interesting. However, this does not prevent these chips from being used for applications where a maximum of three outputs are required.

On the other hand, a 14-pin Baseline version will be proposed, like PIC16F526, capable of controlling 8 LED.

The source

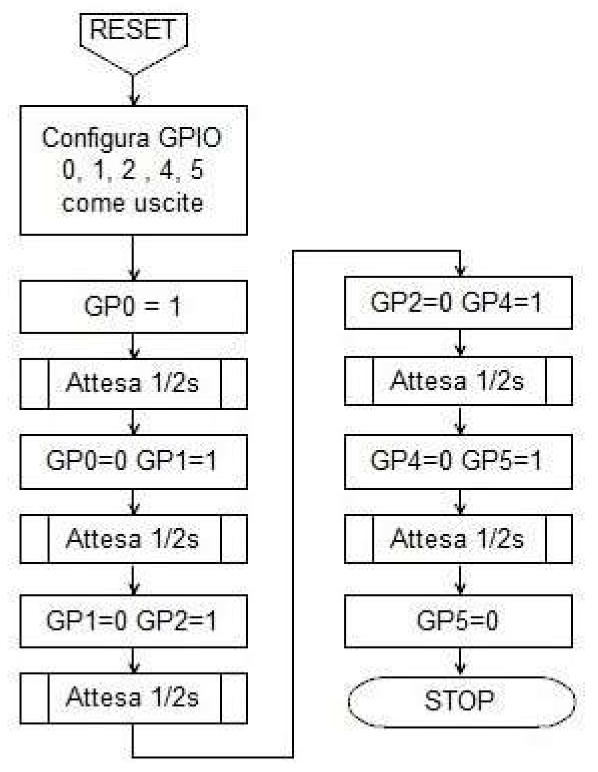
As always, the first thing is to be clear about what you want to achieve:

- **turn on 1 LED in succession along the line of the 5 LEDs connected to the outputs of the microcontroller**

And how:

- **programming the GP0/1/2/4/5 pins as digital outputs and bringing them one after the other to high level at a distance of 1/2 second, turning off the previous one.**

Then we draw the flowchart that shows in graphical form how much the program will have to perform.



On reset, `GP0/1/2/4/5` are programmed as digital outputs.
`GP3` is excluded because, remember, the `GP3`/`MCLR` pin can only be used as a digital input.

`GP0` is set to level 1 (LED1 on); after 1/2 second it is turned off and the LED connected to `GP1 is` lit and so on for the others.

If the LEDs are placed next to each other in a row, you will get the visual effect of a light dot moving along the line.

At the end of the sequence, the program stops after turning off the LED connected to the `GP5`.

The tempo routine is the same as the one used in the previous tutorials, `Delay10msW.asm.`

Let's see how, through the block digraph, we can have a clear vision of what we want to achieve and this makes it easy to transform it into instructions.
Also in this example the source is made available already completely written, in the **4A_519.asm** and **4A_5089.asm** for PIC12F508/509.
It is always 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, even if we continue to use the structures already seen, both for time and for the management of the shadow of the I/O port.
The source, therefore, does not present "novelty" in its general parts, except for the different sequence of instructions necessary to carry out the desired work.

Sequence

Various methods can be used to sequence switching on and off. Here we see how it is possible by introducing an instruction that has not yet been used, the rotation.
In the Baseline instruction set, there are two rotation instructions that slide the bits of a memory location to the right (`rrf`) or left (`rlf`).

ROTATE

Rotation instructions are also called shift statements because they operate by moving the contents of a file to the right or left one position.
The syntax of rotation statements is:

[label]	sp	**RRF/RLF**	**, f/w**	sp	**object**	sp	**[; comment]**

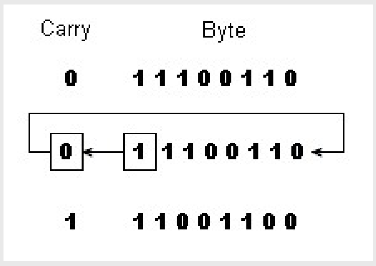
This is one of those opcodes that need a "tail". Remember that the queue indicates where the result of the operation is stored.

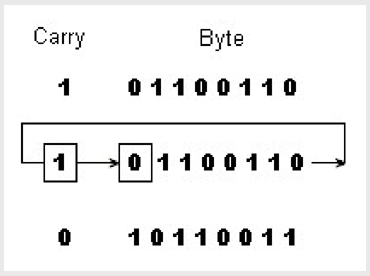
- By indicating **,f** you will have the result of the rotation in the same file that has been rotated.
- By entering **,w** you will have the result in W, without modifying the file in question. This makes education very powerful.

As already mentioned, if we omit the "queue", **MPASM** will report the problem by intervening with the default
,f , without interrupting the compilation and producing the ***.hex*** file. Obviously, if we meant this choice, there is no problem, but if the logic of the program required the other, the executable file obtained will not work as we expect. So, we repeat, it is highly recommended:

- always complete the "queue", in any case
- take a look at the ***Output*** window at the end of the build and check if there are any messages or warnings, even if the **`BUILD SUCCEED line has appeared`**

This seemingly strange instruction is actually indispensable in mathematical operations. From a practical point of view, the rotation instructions work like this:

RLF	 Carry
	Byte
	0
	11100110
	0
	1 1100110
	1
	11001100
		Rotate theeft file through Carry, rotate the ***f*** file to the left through the ***Carry***. The result can go in f or W. Change the C flag of the STATUS.
		
		Example:
		`movlw b'11100110' ; target= 11100110`
		`movwf Target ;`
		`Bcf STATUS, C ; C flag = 0`
		`rlf Target ; target = b'11001100'`
		`C = 0`

RRF	 Carry
	Byte
	1
	01100110
	1
	0 1100110
	0
	10110011
		Rotate right file through Carry, rotate file ***f*** to the right through Carry. The result can go in f or W. Change the C flag of the STATUS.
		
		Example:
		`movlw b'01100110' ; target= 01100110`
		`movwf Target ;`
		`Bsf STATUS, C ; C flag = 1`
		`RRF target, f ; target = b'10110011"`
		`C = 0`

These operations use a special bit of the SFR register called **STATUS :** the **C** bit, i.e. the ***Carry*** (usually corresponding to the 0 bit of the register).

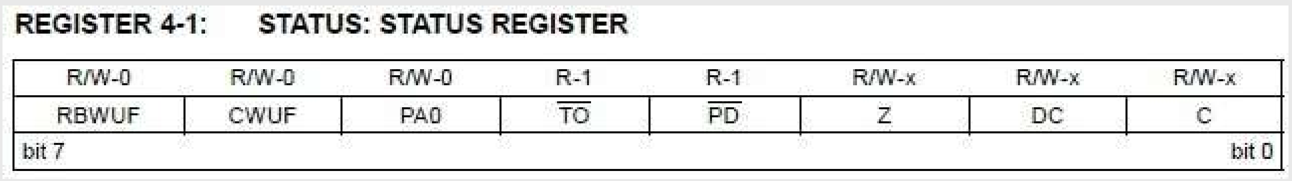
This bit, as the name indicates, has functions of "transport" the result of an operation: its value changes depending on the result of many instructions. In particular, for rotation, it has the function of "receiving" the bit ejected from the rotation itself and passing its value to the bit that has been emptied in the register that is being rotated.

In particular, we notice that the rotation, of one bit at a time, is carried out as if the register were "rolled" including the ***Carry bit*** of the STATUS.

Essentially, each time the RLF instruction is executed , all bits move one position to the left, including the contents of the Carry in bit 0 and passing bit 7 to the Carry.
On the other hand, there is the **RRF** , which rotates to the right. Essentially, it's as if the register to be rotated and the Carry are wrapped on a cylinder.

Status

Among the various **SFRs**, the **`STATUS`** register is always present in all microcontrollers and performs an important function: its bits are essentially **signals of the result of instructions**.
Here, for example, is the map for the 12F519:



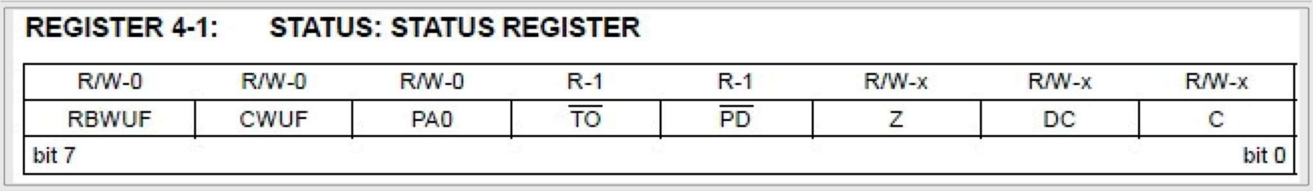
REGISTER 4-1: STATUS: STATUS REGISTER

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
RBWUF	CWUF	PA0	$\overline{TO}$	$\overline{PD}$	Z	DC	C
bit 7							bit 0

In particular, in the **`STATUS`** of the Baselines there are bits with various functions:

- **C** (Carry), **DC** (Digit Carry) and **Z** (Zero) are signal bits (flags) and are relative to the result of instructions.
- **T0 and PD** are signal bits (flags), relative to the ways in which a reset occurs.
- **GPWUF** is a signal bit and indicates a reset due to the change in pin level, the function of which we will see later
- **PA0** is a control bit and serves as a switch for switching between memory pages.

In the example above (12F508-509), bit 6 is not used, but in other chips it can take on a function. For example, in 16F526 the situation is as follows:



REGISTER 4-1: STATUS: STATUS REGISTER

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
RBWUF	CWUF	PA0	$\overline{TO}$	$\overline{PD}$	Z	DC	C
bit 7							bit 0

Here bits 7 and 6 are signals, respectively of reset due to the change of state of a pin of the **`PORTB`** **`(RBWUF)`** and comparator switching (**`CWUF)`** **.**
The rest are identical, as the manufacturer's tendency is to maintain the same structure between the various chips as much as possible, to facilitate the user. However, despite this tendency, it is advisable to check the real situation of the STATUS bits for the chip you are using on the component datasheet.

The log bits are writable and readable, but the reporting bits **are automatically updated** as a result of the numerous operations where it is essential that a specific condition of the result is reported:

- **C - Carry** is important to indicate a carryover in addition or subtraction (where it is called ***Borrow***).

- **Z - Zero** indicates that the result of an operation is zero
- **DC - Digit Carry** indicates whether a carry has occurred in the low 4 bits and is used for conversion operations

At the POR, these bits, since no instruction has yet been made, have a random value.

- **TO** and **PD** are relative to the conditions on which the Reset depends. At the ROP, they both have a value of 1. Their function is to make it possible to distinguish the cause that produced the reset.

`PA0` at the POR is zeroed, making page 0 of memory operational).
`GPWUF` at the POR is at zero, since the level change function on the pins is not activated.

For now, we are not interested in knowing anything else about **STATUS** because we will see in the course of the following exercises to deepen the subject, where necessary.

And, in fact, it is not even what position the C bit occupies, since one of the fundamental elements of programming, as we have repeatedly said, is the substitution of absolutes with labels.
Also in this case, the file ***processorname.inc*** which contains the definitions of the processor used and which we insert in the source with a `#include`, also contains the symbolic elements for the management of the STATUS and its bits.

So, **to reset the Carry bit, we'll never use an absolute address indication** like this:

```
    bcf 3, 0          ; reset Carry
```

This writing is functional, but completely contrary to the rules of good Assembly programming, because:

- First of all, it will be usable only and exclusively with a processor that has the STATUS at memory address 3 and the Carry at bit 0, which can be valid for one chip, but not for others and therefore is not portable.
- It also does not make it clear in the slightest what the instruction does, unless there is a comment. It is, therefore, a way of writing Assembly lines that is completely condemnable and that must be avoided at all costs.

Proper writing will be:

```
    bcf STATUS, C
```

which is in itself clear and does not need comment, as well as adapting to any processor, since with this writing we leave to the compiler automatisms the task of assigning the right numerical values to the `STATUS` and `C` `labels`, taking them from the file ***processorname.inc*** .
And, if we don't like repeating this line, we can replace it using the pseudo opcode `clrc` offered by MPASM, or, where not supported, creating a Macro:

```
; Macro per clear Carry
clrc    MACRO
             bcf     STATUS,C
        ENDM
```

Let's apply bit rotation

Given the function of the rotation instruction and the Carry bit, we apply them to our case, where we use the **RLF** with the aim of moving the bit that keeps the LED on along the **GPIO** register of the I/O, which we have configured as outputs.
However, we have the problem of the non-continuity of GPs, since **GP3** cannot be used as an output, but only as an input (and, moreover, in this case it has the function of **MCLR**). It is therefore excluded from the rotation, which, at that point, must change from **GP2** to **GP4**.
In the case of the **PIC12F5xx** the **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**

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

If we bring the 0 bit (**GP0) to 1**, we find ourselves in this situation:

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

The LED connected to **GP0** will be on, the others will be off. Now let's rotate (shift) bit 0 to the left, injecting a ***Carry=0 instead***:

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

Now the lit LED will be the one connected to bit 1 (**GP1**). By repeating the operation at a cadence that allows you to see the LEDs on and off, we will obtain the desired effect.
Once we get to bit 3, as mentioned, we will have to skip it, moving on to bit 4. With bit 5, we've run out of

I/O pins and we can close the loop.
We render this in instructions:

```
Mainloop:
        Bsf     GPIO,GP0        ; LED1 on
        movf    GPIO,w          ; Copy I/O State to Shadow
        movwf   sGPIO
        movlw   .50             ; Hold 1/2 s
        Call    Delay10msW
; LED2
        CLRC
        rlf     sGPIO, f        ; Shadow Wheel
        movf    sGPIO, w
        movwf   GPIO            ; LED On
        movlw   .50             ; Hold 1/2 s
        Call    Delay10msW
; LED3
        CLRC
        rlf     sGPIO, f        ; Shadow Wheel
        movf    sGPIO, w
        movwf   GPIO            ; LED On
        movlw   .50             ; Hold 1/2 s
        Call    Delay10msW
; LED4 on GP4 - skip GP3
        CLRC
        rlf     sGPIO, f        ; shadow wheel to jump GP3 rlf
                 sGPIO, f        ; Shadow Wheel
        movf    sGPIO, w
        movwf   GPIO            ; LED On
        movlw   .50             ; Hold 1/2 s
        Call    Delay10msW
; LED5
        CLRC
        rlf     sGPIO, f        ; Shadow Wheel
        movf    sGPIO, w
        movwf   GPIO            ; LED On
        movlw   .50             ; Hold 1/2 s
        Call    Delay10msW
; End of Sequence
        CLRF    sGPIO           ; All LEDs Off
        movf    sGPIO, w
        movlw   .100            ; wait for 1s
        Call    Delay10msW
        Goto    $               ; stop
```

Let's compress...

We can see that the sequence of rotation instructions repeats identically for each LED. You might want to turn it into a subroutine:

```
;                         SUBROUTINES
; ruota shadow e I/O
Rotate  clrc
```

```
        rlf     sGPIO, f            ; Shadow Wheel
        movf    sGPIO, w            ; transfer to I/O
        movwf   GPIO                ; LED On
        movlw   .50                 ; Hold 1/2 s
        Call    Delay10msW
        retlw   0
```

So the main becomes:

```
mainloop:
        bsf     GPIO,GP0            ; LED0 on
        movf    GPIO,w              ; Copy I/O State to Shadow
        movwf   sGPIO
        movlw   .50                 ; Hold 1/2 s
        call    Delay10msW
; LED1
        call    Rotate              ; shadow rotate and
; LED2
        call    Rotate              ; shadow rotate and I/O
; LED3 su GP4 - salta GP3
         clrc
         rlf     sGPIO, f            ; shadow rotate to jump GP3
        call    Rotate              ; shadow rotate and I/O

; LED4
        call    Rotate              ; ruota shadow e I/O
; fine sequenza
        clrf    sGPIO               ; All LEDs Off
        movf    sGPIO, w
        movlw   .100                ; wait per 1s
        call    Delay10msW
        goto    $                   ; stop
```

Definitely tighter.

The solution has the following advantages:

- The use of program memory is reduced. This is essential for large programs, given the small size of the chip's flash
- makes the source more readable

On the other hand, it should be remembered that Baselines:

- they have serious limitations both in terms of the placement of subroutines (within the first 256 of each page)
- They have a stack with only two levels that do not lend themselves to the intensive use of subroutines

It should also be noted that the execution time is heavier than the call and the return from the subroutine, but this is usually not of great importance, unless you are dealing with a very critical application in terms of execution times.

One important thing must be kept in mind:

STOP In the **Rotate subroutine**, an additional subroutine, the Delay10msW delay subroutine, is invoked. Due to the above-mentioned limitation of the stack to only two levels, it is not possible for the **DelayW10ms** to invoke another subroutine in turn, otherwise the stack will overflow and the program will fail.

Incidentally, we can observe that the stack situation when executing the Rotate call will be to have loaded the indent address to the next instruction to **the Rotate call** first and just above the indentation address from the **DelayW10ms call**. With this, the two layers of the minimal Baseline stack are fully engaged.
An additional call within ("nested") the **Rotate call is not possible**.

... and a few tricks...

However, we can limit the use of the stack by using a trick to make an indentation from the chain of subroutines. If we look at the source, we notice that the sequence:

```
        call     Delay10msW
        retlw    0
```

This means that:

- the stack already contains the indent address from the **Rotate procedure call**
- the Delay10msW **subroutine is called**
- The address of the next line is also loaded onto the stack. Both levels are occupied
- when re-entering from the subroutine, the stack is unloaded one location and now contains the indent address for the **Rotate**
- The **retlw line** is a subroutine indent that picks up the address from the stack

You can then use a form that in English is called ***dirty return*** , i.e. call the **Delay10msW** not with **call**, but with **goto** !
goto transfers the Progam Counter to the specified address, but does not touch the stack, which contains the return coordinates of the instruction following the **Rotate call**. The sequence is as follows:

- the stack contains the indent address from the **Rotate procedure call**
- a jump is ordered to the label address **Delay10msW**
- At the end of the execution of the tempo routine, the program encounters a **retlw** whose function is to retrieve the indent address from the stack
- but the stack contains only the indentation address from the Rotate procedure call

When the execution reaches **retlw 0** of the **Delay10msW**, the Program Counter fetches a return address from the stack and this is the one for the Rotate call .
You save a statement at the end of the subroutine, a stack level, and a microsecond in execution.

```
;                    SUBROUTINES
;Rotate
clrc
rlf      sGPIO, f        ; Shadow Wheel
movf     sGPIO, w        ; transfer to I/O
movwf    GPIO            ; LED On
movlw    .50             ; Hold 1/2 s
Goto     Delay10msW
```

This technique is one of the many tricks that can be put in place to improve the efficiency of a program. It must be said, however, that it, like the others we will see, should only be used when it is clear what its purpose and implications are.

... in addition to an optimization

A second variation to the source can be made considering that the flashing time may be unsuitable for the application; changing it would mean substituting the value passed from W to the wait sub. It is much better to declare this value a priori in symbolic form:

```
;                          CONSTANTS
LEDtime  Equ  .50        ; LED switch-on time 50 x 10ms = 500ms
```

And, accordingly, replacing the value with the symbol:

```
 movlw    LEDtime             ; tempo accensione LED
goto     Delay10msW
```

This allows you to easily change the tempo by acting at only one point in the source.

Pay attention to the route

A final note concerns the placement of the subroutine that is included in the source:

```
; 10ms x W delay subroutine
 #include  C:\PIC\Library\Baseline\Delay10msW.asm
```

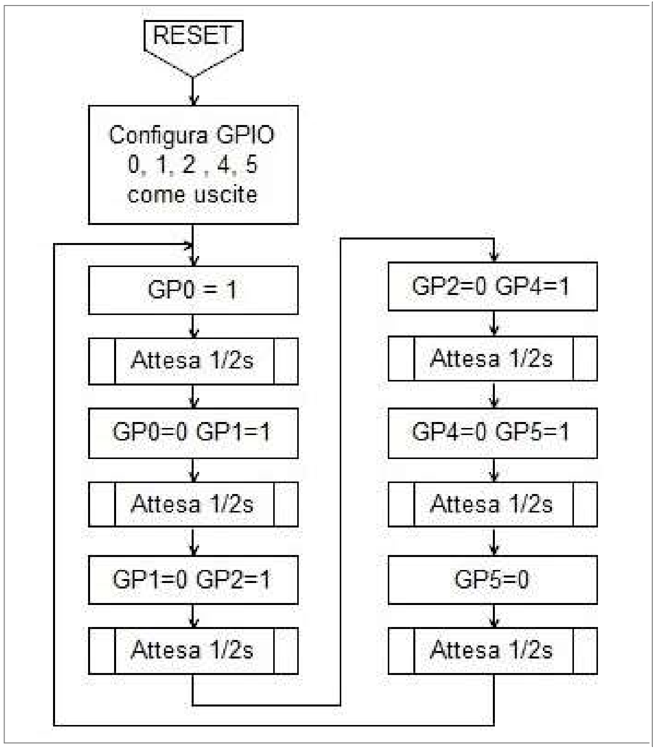
This means that the sub must be in the indica folder, i.e.
`C:\PIC\Library\Baseline\`.
If this is not the case, you must modify the complete path (= path, track, trace) so that it points to the location where the file is located, otherwise the compiler will indicate an error related to the impossibility of inclusion, as well as a large number of other errors every time it encounters an element defined in the missing file.

A simple variation.

As written, the program scrolls through the LEDs and then stops. To restart the sequence we need to act on the RESET, which brings the Program Counter back to the beginning. If, however, we replace the `goto $ statement` with a `goto mainloop`, we determine a continuous loop

```
; fine conteggio
countend     goto     $              ; End of Count and Block
```

we create a loop that repeats the sequence of the moving LED indefinitely.



The drafting of the flow chart allows you to immediately grasp the difference: we do not close the program with a loop on itself, but we bring the loop to the beginning of the sequence, so that it repeats itself indefinitely.

A pause is introduced before the sequence restarts to improve the effect.

Obviously, for a different speed of execution, it will be enough to vary the tempo routine.

Here, too, we will be able to check the function of the reset pin that brings the execution back to the beginning of the program.

We can also insert, at the end of a sequence, a longer wait, for example 1 second. In the same way, we can use different times for each LED.

More LEDs - PIC16F526

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.

If we didn't consider the 6-pin chips in this tutorial, since the small number of I/Os useful as digital outputs (only 3) would have made it uninteresting, we can see instead a 14-pin chip, the **PIC16F526**, of which we find a fairly detailed description here.

This chip integrates two 6-bit ports (PORTB and PORTC).

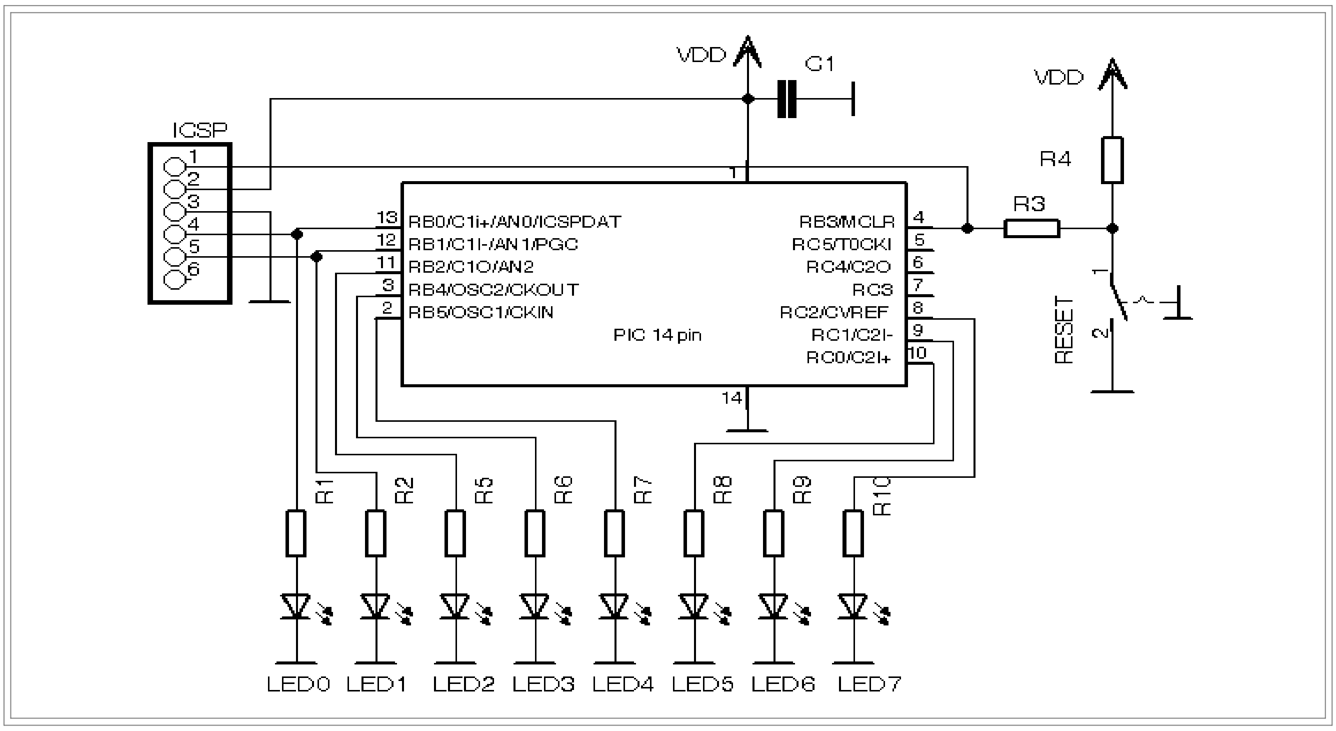
PORTB	**Bit**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	Label	-	-	**RB5**	**RB4**	**RB3**	**RB2**	**RB1**	**RB0**
	pin	-	-						
PORTC	**Bit**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	Label	-	-	**RC5**	**RC4**	**RC3**	**RC2**	**RC1**	**RC0**
	pin	-	-						

Since there are only 6 valid pins, bits 6 and 7 of the registers have no function: reading them will give you 0 and writing them will have no effect.

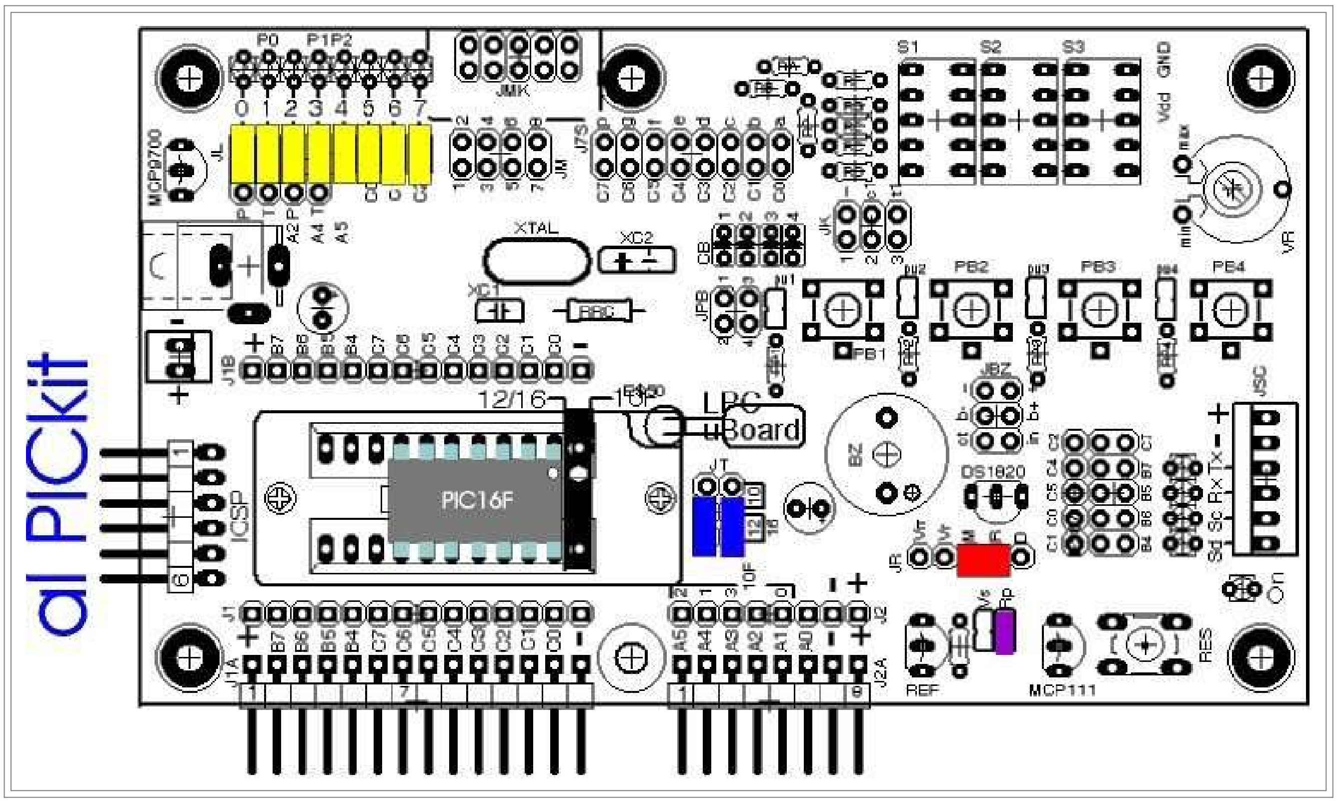
This allows us to easily control 11 output loads and therefore the 8 LEDs that it makes available. We said 11 and not 12, since **`RB3`** has the usual **`MCLR`** function and can only be used digitally as an input.

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, **`TRISB`** and **`TRISC`** registers and **`PORTB`** and **`PORTC`** registers for the two I/O groups.
There are other differences, such as the amount of memory available, RAM, integrated function modules and related SFRs, which, for now, we do not consider in the specific operation, because, where necessary, they require their own management, but only from the point of view of the use of digital I/O.
The wiring diagram of the application is as follows:



The board is easily configurable with the usual jumpers:



Here we can connect all 8 LEDs, dependent on the JL "yellow" jumpers.

Again, what is connected on the **ICSP** data and clock lines (**`RB0`** and **`RB1`**) must be isolated with jumpers during programming if they constitute an excessive load for the programmer. In the case of the board we use, this problem does not arise, given the minimal consumption of the LED.

It is always worth noting the position of the "blue" jumpers, for the 16F chips and the insertion of the chip on the socket, which skips the first position, reserved for the 10F.

The source for the PIC16F526

Consulting the data sheet of the component, we observe that, in addition to the two ports, it has some **Analog Integrated Modules:**

- **Dual Comparator**
- **ADC**

For chips that have analog function modules integrated, special caution is required on the part of the programmer:

1. These modules have shared inputs and outputs on some pins, along with digital I/O functions.
2. It is not possible to enable multiple functions on the same pin at the same time.
3. Based on Microchip's aforementioned philosophy, minimum power configurations, or, if available, analog configurations, are active by default.

As a result, you are required to disable them if you want to access the use of pins as simple digital I/O.

This can be achieved by acting on the **ADCON0**, **CM1CON0** and **CM2CON0** registers, which respectively control the operation of the ADC module and the comparators.

```
; Disable Analog Inputs
clrf   ADCON0

; Disable comparators to free up the digital function
bcf    CM1CON0, C1ON
       bcf     CM2CON0, C2ON
```

As usual for Baselines, the **T0CKI** function, if not used, must be disabled. It wouldn't be necessary here, since we don't use the **RC5** pin, but it's useful to keep this peculiarity in mind.

```
; disable T0CKI from RB5
; OPTION       11111111
       movlw b'11011111'
       OPTION
```

Among the resources, the chip has ***Wake Up for Pin Change*** and related pull-ups, which are however disabled bringing bits 6 and 7 of the Option_Reg to 1.
There are also other modules, such as ***CVref***, the reference voltage for comparators, but it is disabled by default and has no influence on I/O.
The chip has SFR and RAM on **4 banks**, but, given the minimum power consumption required by the program, we do not take this into account, since we do not exceed the availability of what is accessible from bank 0. The chip

it also has **EEPROMs**, but we don't use this either, so you don't need to take it into account.

On the other hand, it is necessary to take into account a detail that a superficial investigation of the data sheet could miss; This is the fact that:

- **the internal oscillator can be set to two frequencies**, i.e. 4 or 8MHz
- and that the **RB4** pin can be used as the output of the F0sc/4 clock.

These options must be laid out in the initial config:

```
; Internal oscillator, 4MHz, no WDT, no CP, RB3=MCLR
 __config _IntRC_OSC_RB4 & _IOSCFS_4MHz & _WDTE_OFF & _CP_OFF & _CPDF_OFF
& _MCLRE_ON
```

We observe that the selection of the internal oscillator, with no output on RB4, is done with the command **_IntRC_OSC_RB4** , while that of 4MHz with **_IOSCFS_4MHz**.
If we wanted an 8MHz clock, we would have to set **_IOSCFS_8MHz** . These options will be discussed in depth in a later tutorial on PIC clock systems.

As in the previous examples, we take care to exclude the Watchdog (_WDTE_OFF) in the configuration, which, otherwise, is active by default and should be managed appropriately.

We also take care to disable the protections against rereading the memory of the chip, to avoid problems in experimental use; This is not essential, but highly recommended. Protections should only be activated when you do not want the firmware to be copied in the final production version.
Also, since we use the **RB3** pin as **the MCRL,** we activate the RB3 function.

As always, the configuration line, with this write mode, takes on a disproportionate length, but it is useful that all the options are configured. By default it is possible that some options that we are interested in are already in the right value (such as protections), but it is always advisable to make **a complete configuration of all the available parameters** of the chip, in order to have a clear understanding of the basic situation of the hardware.
Leaving this aside, it is very easy to run into errors that will prevent the program from working properly and are then very difficult to detect.

As always, this configuration information can be found in the ***16F526.inc file***.

More **detail pages on the 16F526** can be found here.

Subroutines for the 16F526

If we want to use 8 LEDs we have to use the **PORTB bits** and part of the **PORTC** bits, since each port is composed of only 6 bits.
The sequence of LEDs, based on the hardware connections made, is as follows, skipping **RB3** :

1.	**RB0**	**LPCuB LED0**
2.	**RB1**	**LPCuB LED1**
3.	**RB2**	**LPCuB LED2**
4.	**RB4**	**LPCuB LED3**
5.	**RB5**	**LPCuB LED4**
6.	**RC0**	**LPCuB LED5**
7.	**RC1**	**LPCuB LED6**
8.	**RC2**	**LPCuB LED7**

Reading the source we can see that two rotation subroutines have been realized, one involving **PORTB** and one involving **PORTC**. Among the many possibilities, they could take this form:

```
; Rotation and Wait Subroutines
RotateB:
    CLRC                    ; Reset Carry
    rlf     sPORTB, f       ; Shadow Wheel
    movf    sPORTB, w
    movwf   PORTB           ; LED On
    movlw   .50             ; 50*10ms = 500ms
    Call    Delay10msW      ; time
    Retlw   0

RotateC:
    CLRC                    ; Reset Carry
    RLF     sPORTC, f       ; Shadow Wheel
    MOVF    sPORTC, w
    movwf   PORTC           ; LED On
    movlw   .50             ; 50*10ms = 500ms
    Call    Delay10msW      ; time
    Retlw   0
```

The two sequences are entirely analogous; the difference lies in the PORT on which they act. The sequence of instructions is quite simple:

- The carry is reset to inject a 0 into the log rotation
- The shadow of the port is rotated, resulting in the rotation stored in the same file
- the shadow is copied to **WREG** and then to the port
- The desired delay time is added

More tricks

Here, too, we can introduce dirty *return*, saving space and stacks. Then the source becomes:

```
; Rotation and Wait Subroutines
RotateB:
    CLRC                 ; Reset Carry
    rlf    sPORTB, f     ; Shadow Wheel
    movf   sPORTB, w
    movwf  PORTB         ; LED On
    movlw  LEDtime       ; 50*10ms = 500ms
    Goto   DelayW10ms    ; dirty return
RotateC:
    CLRC                 ; Reset Carry
    rlf    sPORTC, f     ; Shadow Wheel
    movf   sPORTC, w
    movwf  PORTC         ; LED On
    movlw  LEDtime       ; 50*10ms = 500ms
    Goto   DelayW10ms    ; dirty return
```

More tricks !

The similarity between the two subroutines means that there are points in common that allow a reduction in the number of lines written in the source:

```
; Rotation and Wait Subroutines
RotateB:
    clrc                 ; reset Carry
    rlf    sPORTB, f     ; rotate shadow
    movf   sPORTB, w
    movwf  PORTB         ; LED on
    goto   rcomm         ; Common Output

RotateC:
    clrc                 ; reset Carry
    rlf    sPORTC, f     ; rotate
    movf   sPORTC, w
    movwf  PORTC         ; LED on
rcomm
    movlw  LEDtime       ; 50*10ms = 500ms
    goto   DelayW10ms    ; dirty return
```

By jumping we end the first subroutine in the "tail" of the second.
This technique saves space in the program memory because instead of the two lines

```
    movlw LEDtime
    goto DelayW10ms
```

Only one line is written

```
    Goto rcomm
```

with a saving of two bytes of program memory. This way of operating is not mandatory, but it can be indispensable when dealing with programs of a certain size and limited program memory availability, such as in Baselines.
Be careful because this has a negative side in the execution, as the two cycles necessary for the execution of the `goto are added`. Therefore, even in this case, it is necessary to evaluate the advantages and disadvantages of the "trick". In the example in the present example, there are no problems of space in the program memory and not even problems of criticality in the execution; Therefore, the application of the solution described is entirely optional.

```
mainloop:bsf  PORTB,RB0       ; LED 0 on
     movf  PORTB,sPORTB     ; copy I/O      in shadow
                                   status

; Hold 1/2 s and rotate movlw
     LEDtime
     call  Delay10msW

; Start of Rotation
     call  RotateB          ; LED0 on
     call  RotateB          ; LED1 on

; Skip PB3
     bcf   STATUS,C
     rlf   sPORTB, f        ; rotate shadow

; Continuous rotation
     call  RotateB          ; LED2 on
     call  RotateB          ; LED3 on
     call  RotateB          ; LED4 on
```

and the transition from **PORTB** at **PORTC**:

```
; off LED di PORTBbcf
           PORTB

; on LED su PORTCbsf
           PORTC,RC0        ; LED5 on
     movf  PORTC,sPORTC     ; Copy I/O State to Shadow

; Hold 1/2 s and rotate
     movlw LEDtime
     call  DelayW10ms

; First Rotation
     call  RotateC          ; LED6 on
     call  RotateC          ; LED7 on

; off LED di PORTC
bcf  PORTC
```

```
    goto $                     ; Loop Stuck
```

The **final goto** closes the program's indefinite shutdown loop. Pressing the Reset button restarts the cycle.

Where we have previously stated:

```
;#################################################################
;COSTANTS
;
LEDtime   EQU   .50            ; LED switch-on time 50 x 10ms = 500ms
ENDtime   equ   (LEDtime * 2)  ; Sequence end time - 1s
```

Note that the parameter for the time of 1s (100 x 10ms) is computed by the Assembler with the expression **(LEDtime * 2).**

You can also consult the pages linked here.

Alternate ending!

As indicated before, if we want a continuous loop, we just have to aim the final jump not on itself, but on the beginning of the sequence of rotations, so that it repeats indefinitely.

```
; off LED di PORTC
bcf   PORTC

; Hold 1s and new cycle
     movlw    ENDtime
     call     Delay10msW
     ; Goto    $             ; stop
     Goto     Mainloop       ; Alternate Ending: Repeat Sequence
```

In the sources provided, it will be enough to commit(start with ;) o decommute (eliminate the initial semicolon) the trailing lines to switch from one version to another. Obviously, for each change you will have to recompile. In addition, we have declared two constants with labels, so that we can only control them at one point in the source:

```
;#################################################################
;COSTANTs
LEDtime   EQU   .50            ; LED switch-on time 50 x 10ms = 500ms
ENDtime   equ   (LEDtime * 2)  ; Sequence end time - 1s
```

Note that the parameter for the time of 1s (100 x 10ms) is computed by the Assembler with the expression **(LEDtime * 2).**

You can also consult the pages linked here.

Other changes

It should be clear how many possibilities open up with simple outbound I/O management. Let's look at some simple variations.

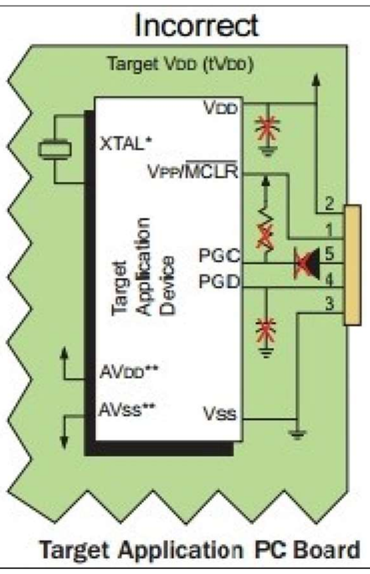
You can experiment independently with continuous or non-continuous sequences, inventing any other variation both on the number of LEDs involved and with the variation of intervention times.

These structures are the basis of circuits called sequencers (*seuqencer*), which have considerable importance in many sectors, from automation to music, from home automation to civil applications.

Programming Note

From the wiring diagram you can see that the **`GP0`**/**`GP1`** (**`RB0`**/**`RB1`**) pins are used to control the **0/1 LED** of the **LPCuB board**.

These pins are also the ones used for programming the chip, i.e. **PGC** and **PGD**, signals that come from the **Pickit**.
The programming of the chip, as already mentioned, takes place with a synchronous serial communication that is established between the chip and the programming tools. This communication takes place at a fairly high frequency and requires the signals to maintain well-defined switching times and durations. Applying a load to these pins could damage this communication, making it impossible to program the component.
Microchip's documentation is very strict in this regard:



- Do not use mulitplexing on PGC/PGD – they are dedicated for communications to PICkit 3.
- Do not use pull-ups on PGC/PGD – they will divide the voltage levels since these lines have 4.7 kΩ pull-down resistors in PICkit 3.
- Do not use capacitors on PGC/PGD – they will prevent fast transitions on data and clock lines during programming and debug communications.
- Do not use capacitors on MCLR – they will prevent fast transitions of VPP.
- Do not use diodes on PGC/PGD – they will prevent bidirectional communication between PICkit 3 and the target PIC® MCU.

When **we insert the** LED0 **and** LED1 **jumpers on the LPCuB** board, we connect these LEDs to the communication lines. But this is not a problem as these are loads with negligible capacity and therefore do not deform the switching edges and do not introduce errors in the timings. In addition, the current drawn by these LEDs is minimal, less than 2mA, which does not disturb the communication, which is driven by the pins of the microcontrollers (the one being programmed and the one inside the programming tool) that can provide 25mA.

As a result, **during programming, you will see the two LEDs flash rapidly,** and this will not create any problems for the operation.

If you are not using the development board and are using higher current LEDs, there may be communication problems between the tool and the chip. In this case, insert, as recommended at the beginning, two jumpers to isolate the LEDs during programming.

12F519 - 4A_519.ASM

```
;*********************************************************************
;---------------------------------------------------------------------
;
;    Title          : Assembly & C Course - Tutorial 4A_519
;                       LED scrolling with rotation instructions
;                       Cadence of 1/2 second. Loop.
;
;    PIC            : 12F519
;    Support        : MPASM
;    Version        : V.519-1.0
;    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 to Vss
;    GP4/OSC2             3: Out LED to Vss
;    GP3/! MCLR/VPP       4: MCLR
;    GP2/T0CKI            5: Out LED at Vss
;    GP1/ICSPCLK          6: Out LED to Vss
;    GP0/ICSPDAT          7: Out LED to Vss
;    Vss                  8: --
;
;*********************************************************************
;=====================================================================
;            DEFINITION OF PORT USE
;
; GPIO map
;| 5 | 4 | 3 | 2 | 1 | 0 |
;|-----|-----|-----|-----|-----|-----|
;|LED5 |LED4 | MCLR|LED2 |LED1 |LED0 |
;
#define LED0 GPIO,GP0 ; LED between pin and Vss
#define LED1 GPIO,GP1 ; LED between pin and Vss
#define LED2 GPIO,GP2 ; LED between pin and Vss
;#define GPIO,GP3 ; MCLR
#define LED3 GPIO,GP4 ; LED between pin and Vss
#define LED4 GPIO,GP5 ; LED between pin and Vss
;
;*********************************************************************
;
```

```
;    Notes:
;
; ######################################################################

        LIST      p=12F519         ; Processor Definition
        #include  <p12F519.inc>
        Radix     DEC

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


; ######################################################################
;=====================================================================
;                        RAM
;
; general purpose RAM
        CBLOCK 0x07          ; start of RAM
      sGPIO area             ; GPIO's shadow
      d1,d2,d3               ; ENDC Delay Counters


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

 ; MOWF Internal Oscillator
        Calibration OSCCAL


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

; disable T0CKI to have GP2 as digital I/O
         ;      b'11010111'
         ;        1-------     GPWU Disabled
         ;        -1------     GPPU disabled
         ;        --0-----     Internal Clock
         ;        ---1----     Falling
         ;        ----1---     prescaler at WDT
         ;        -----111     1:256
        movlw   b'11011111'
        OPTION
```

```
; all GPs come out
        movlw   0
        Tris    GPIO

Mainloop:
         Bsf     GPIO,GP0         ; LED0 on
         movf    GPIO,w
         movwf   sGPIO            ; Copy I/O State to Shadow

         movlw   .50              ; Hold 1/2 s
         call    Delay10msW

; LED1
         Call    Rotate           ; shadow wheel and I/O

; LED2
         Call    Rotate           ; shadow wheel and I/O

; LED3 on GP4 - skip GP3
         CLRC
         rlf     sGPIO, f         ; shadow wheel to jump GP3
         call    Rotate           ; shadow wheel and I/O

; LED4
         Call    Rotate           ; shadow wheel and I/O

; End of Sequence
         CLRF    sGPIO            ; All LEDs off movf
                 sGPIO, w

         Goto    $                ; stop
            ;    Mainloop         ; Alternate Ending: Repeat Sequence
         Goto

; ######################################################################
;                        SUBROUTINES
; Rotate
rotation CLRC
Rot1     rlf     sGPIO, f         ; Shadow Wheel
         MOVF    sGPIO, w
         movwf   GPIO             ; LED On
         movlw   .50              ; Hold 1/2 s
         goto    Delay10msW

; 10ms x W delay subroutine
  #include C:\PIC\Library\Baseline\Delay10msW.asm

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

12F508/509 - 4A_5089.ASM

```
;********************************************************************
;--------------------------------------------------------------------
;
;    Title          : Assembly & C Course - Tutorial 4A_5089
;                      LED scrolling with rotation instructions
;                      Cadence of 0.5s
;
;    PIC            : 12F508/509
;    Support        : MPASM
;    Version        : V.519-1.0
;    Date           : 01-05-2013
;    Hardware ref.  :
;    Author         :Afg
;
;--------------------------------------------------------------------
;
; Pin use :
;   _______________
;       12F508/9 @ 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 LED4 at Vss
;    GP4/OSC2            3: Out LED3 at Vss
;    GP3/! MCLR/VPP      4: MCLR
;    GP2/T0CKI           5: Out LED2 at Vss
;    GP1/ICSPCLK         6: Out LED1 at Vss
;    GP0/ICSPDAT         7: Out LED0 at Vss
;    Vss                 8: --
;
;********************************************************************
;             DEFINITION OF PORT USE
;
; GPIO map
;| 5 | 4 | 3 | 2 | 1 | 0 |
;|-----|-----|-----|-----|-----|-----|
;|LED4 |LED3 | MCLR|LED2 |LED1 |LED0 |
;
;#define     LED0 GPIO,GP0     ; LEDs    pin   and
;#define     LED1 GPIO,GP1     between   Vss   pin
;#define     LED2 GPIO,GP2     ; LEDs    and   Vss
;#define     GPIO,GP3          between   pin   and
                               ; LEDs    Vss
                               between
                               ; MCLR
;#define     LED3 GPIO,GP4       ; LEDs  pin and
                                between    Vss
;#define     LED4 GPIO,GP5       ; LEDs  pin and
```

```
                                  between    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

; #####################################################################
;                         CONFIGURATION
;
; Internal oscillator, no WDT, no CP, MCLR
 __config _IntRC_OSC & _WDT_OFF & _CP_OFF & _MCLRE_ON


; #####################################################################
;                         RAM
;
; general purpose RAM
        CBLOCK 0x07
   sGPIO                    ; GPIO's shadow
   D1, D2, D3               ; ENDC Delay Counters

; #####################################################################
;                         RAM
;
; general purpose RAM
        CBLOCK 0x07
   sGPIO                    ; shadow of GPIO
        ENDC

; #####################################################################
;                          CONSTANTS
;
LEDtime EQU    .50   ; LED switch-on time 50 x 10ms = 500ms

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

; all GPs come out
        movlw   0
        Tris    GPIO

mainloop:
        bsf     GPIO,GP0        ; LED0 acceso
        movf    GPIO,w          ; copia stato I/O in shadow
        movwf   sGPIO
        movlw   LEDtime          ; Hold 1/2 s
        call    Delay10msW

; LED1
        Call    Rotate           ; shadow wheel and I/O

; LED2
        Call    Rotate           ; shadow wheel and I/O

; LED3 on GP4 - skip GP3
        CLRC
        rlf     sGPIO, f         ; shadow wheel to jump GP3
        call    Rotate           ; shadow wheel and I/OO

; LED4
        Call    Rotate           ; shadow wheel and I/O

; End of Sequence
        CLRF    sGPIO            ; All LEDs off movf
                sGPIO, w

         Goto    $                ; stop
            ;    Mainloop         ; Alternate Ending: Repeat Sequence
         Goto

; ######################################################################
;                        SUBROUTINES
; Shadow Wheel and
I/O Rotate    CLRC
         rlf     sGPIO, f         ; Shadow Wheel
         movf    sGPIO, w
         movwf   GPIO             ; LED On
         movlw   LEDtime          ; Hold 1/2 s
         Call    Delay10msW
         retlw   0

; 10ms x W delay subroutine
 #include C:\PIC\Library\Baseline\Delay10msW.asm

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

     END
```

16F526/505 - 4A_526.asm

```
;**********************************************************************
;----------------------------------------------------------------------
;
;    Title          : Assembly & C Course - Tutorial 4A_526
;                        LED scrolling with rotation instructions
;                        Cadence of 1/2 second.
;
;    PIC            : 16F526
;    Support        : MPASM
;    Version        : V.519-1.0
;    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 LED4 at Vss
;    RB4/OSC2/CLKOUT              3: Out LED3 at Vss
;    RB3/! MCLR/VPP               4: MCLR
;    RC5/T0CKI                    5:
;    RC4/[C2OUT]                  6:
;    RC3                          7:
;    RC2/[Cvref]                  8: Out LED7 at Vss
;    RC1/[C2IN-]                  9: Out LED6 at Vss
;    RC0/[C2IN+]                 10: Out LED5 at Vss
;    RB2/[C1OUT/AN2]             11: Out LED2 at Vss
;    RB1/[C1IN-/AN1/]ICSPC 12: Out LED1 at Vss
;    RB0/[C1IN+/AN0/]ICSPD 13: Out LED0 at Vss
;    Vss                         14: --
;
;    [ ] only 16F526
;
; #####################################################################

        LIST      p=16F526         ; Processor Definition
        #include <p16F526.inc>
        Radix     DEC

; #####################################################################
;                           CONFIGURATION
;
; Internal Oscillator, 4MHz, No WDT, No CP, RB3=MCLR
 __config _IntRC_OSC_RB4 & _IOSCFS_4MHz & _WDTE_OFF & _CP_OFF &
_CPDF_OFF & _MCLRE_ON


; ######################################################################
;                            RAM
; general purpose RAM
        CBLOCK 0x10          ; start of RAM
      area sPORTB            ; shadow by PORTB
      sPORTC                 ; shadow by PORTC
      d1,d2,d3               ; ENDC Delay Counters

 ; ######################################################################
 ;                          CONSTANTS
 ;
LEDtime EQU .50              ; LED switch-on time 50 x 10ms = 500ms
ENDtime equ (LEDtime * 2) ; Sequence end time - 1s

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

 ; MOWF Internal Oscillator
        Calibration OSCCAL

; ######################################################################
;                         MAIN PROGRAM
Main:
; Reset Initializations
        CLRF    PORTB             ; GPIO preset latch to
        0 clrf  PORTC

; Disable CLRF Analog Inputs
                ADCON0

; Disable comparators to free the BCF digital function
                CM1CON0, C1ON
        Bcf     CM2CON0, C2ON

; disable T0CKI from RB5
         ;      b'11010111'
         ;        1-------     GPWU Disabled
         ;        -1------     GPPU disabled
         ;        --0-----     Internal Clock
         ;        ---1----     Falling
         ;        ----1---     prescaler at WDT
         ;        -----111     1:256
        movlw   b'11011111'
        OPTION

; All useful ports come out
```

```
        movlw   0
        Tris    PORTB             ; To the Management
                                    Register
        Tris    PORTC

Mainloop:
        Bsf     PORTB,RB0         ; LED0 on
        movf    PORTB,w
        movwf   sPORTB            ; I/O Status Copy in Shadow

; Hold 1/2 s and rotate
        movlw   LEDtime
        Call    Delay10msW

; LED1
        Call    RotateB
; LED2
        Call    RotateB

; LED3 to PB4 - jump PB3
        CLRC
        rlf     sPORTB, f         ; Shadow Call
        Wheel   RotateB
; LED4
        Call    RotateB

; LED turn off by PORTB
        CLRF    PORTB

; LED5 on RC0
        Bsf     PORTC,RC0
        movf    PORTC,w
        movwf   sPORTC            ; Copy I/O State to Shadow

; Hold 1/2 s and rotate
        movlw   LEDtime
        Call    Delay10msW

; LED6
        Call    RotateC
; LED7
        Call    RotateC

; LED turn off by PORTC
        CLRF    PORTC

; Hold 1s and new cycle
        movlw   ENDtime
        Call    Delay10msW

         Goto    $                ; stop
            ;    Mainloop         ; Alternate Ending: Repeat   sequence
         Goto

; ######################################################################
;                         SUBROUTINES
;
; RotateB Rotation and Wait
```

Subroutine:

```
        CLRC                        ; Carry = 0
        rlf     sPORTB, f           ; wheel Shadow
        movf    sPORTB, w           ;  copy in W
        movwf   PORTB               ;  copy On the
                                           port
        Goto    rcom                ; Common indentation

RotateC:
        CLRC                        ; Carry = 0
        rlf     sPORTC, f           ; wheel Shadow
        movf    sPORTC, w           ; copy  in W
        movwf   PORTC               ; copy  On the port
rcom    movlw   LEDtime             ; Hold 1/2 s and rotate
        Goto    Delay10msW          ; dirty return

; 10ms x W delay subroutine
 #include C:\PIC\Library\Baseline\Delay10msW.asm

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