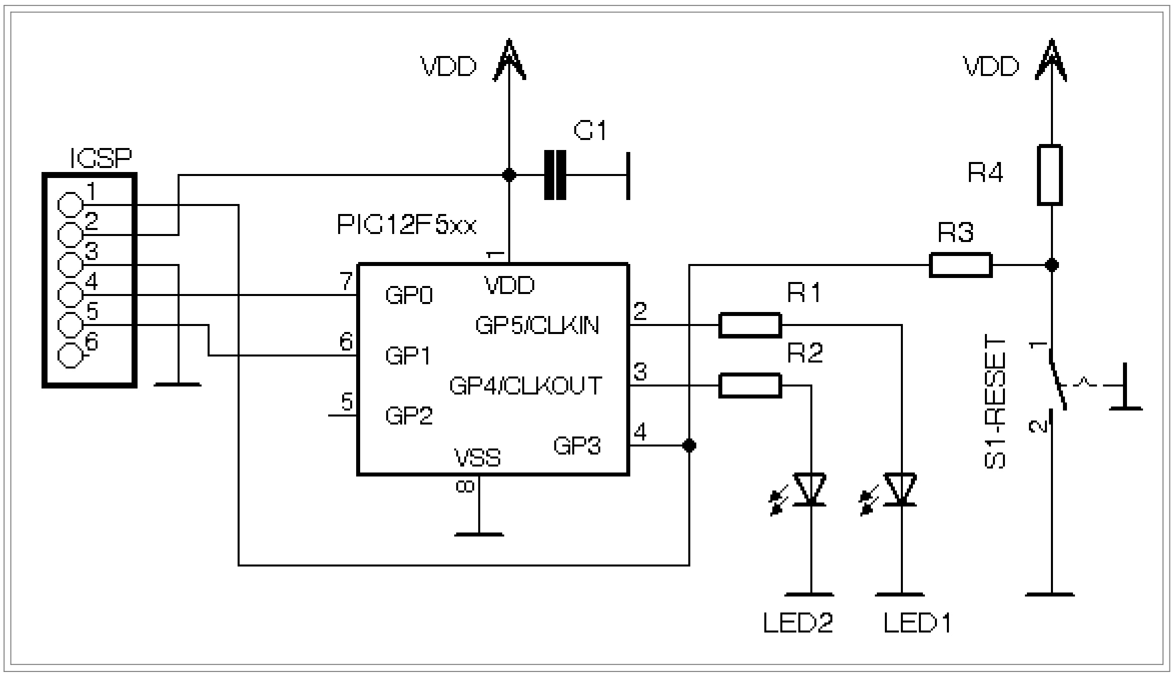
Knowledge of baseline

Assembly - Two flashing LEDs : R-M-W

What we want to achieve

We want to flash two LEDs alternately and check the operation of the MCLR pin.

We keep the main hardware connections, including the ICSP connection, and add as needed.

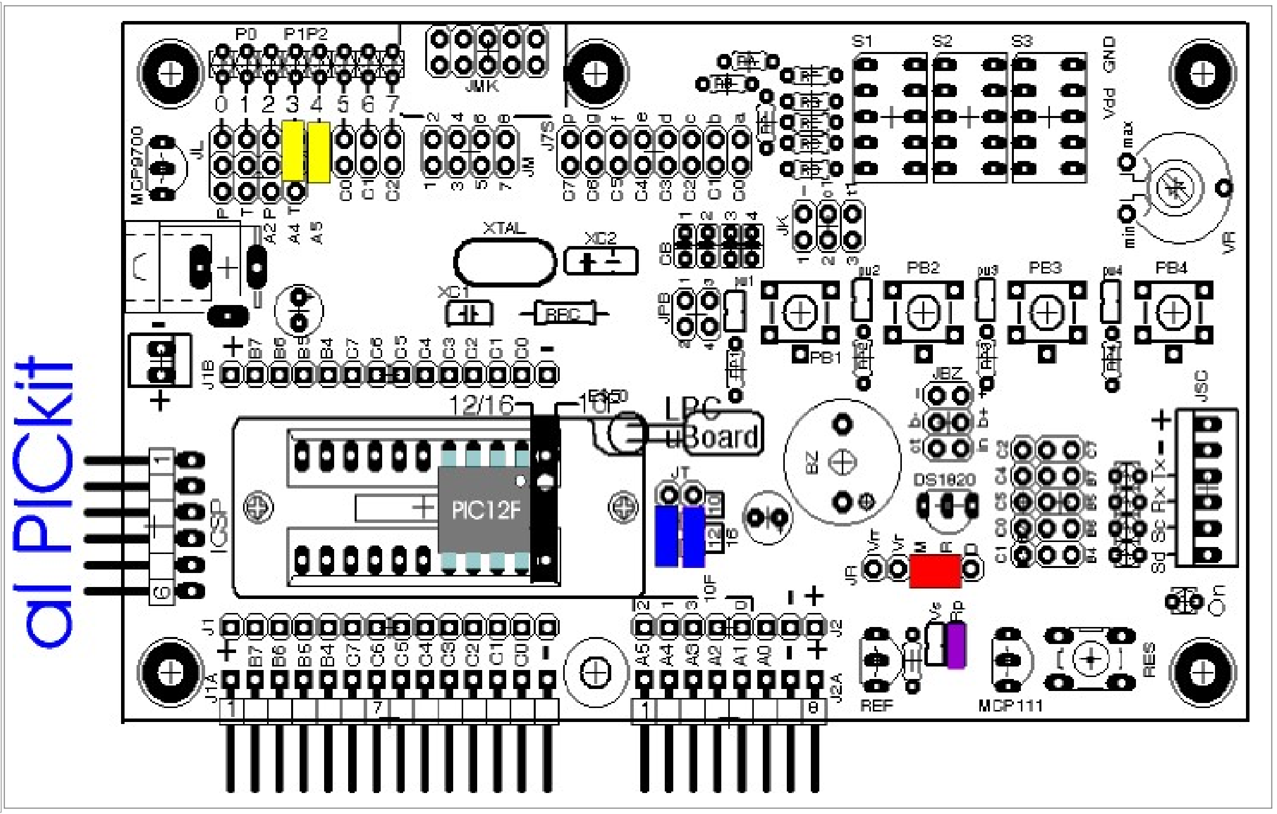


The two LEDs will be connected to **`GP4`** and **`GP5`**. Each pin controls an LED in the same way that we have already described in the previous tutorials. The LEDs are connected between pin and ground (Vss), with the relative limiting resistor in series and will therefore be lit when the pin, configured as a digital output, is set to a high level.

We also connect a reset button, with its resistors, to the **`MCLR/GP3`** pin. This button, when pressed, connects the pin at low level through the R3, pin which is normally at high level through the R4 pull-up.
In this way, by pressing the button, we activate a reset condition in the microcontroller and we can see the consequences.

The R3 is necessary because MCLR is also the pin where the Vpp programming voltage is applied; the function of the resistor is to separate the Vpp from the Vdd of the circuit. A small Schottky diode can also be used for this purpose, but in general the indicated resistance is adequate.

On the [LPCuB development board](#)all these elements are already wired and you just need to insert jumpers: we will have this situation for the **PIC12Fxxx**:

12F	

The connection with the Reset button and the corresponding resistors is simply completed with the "red" jumper.

Always observe the placement of the chip on the 22-pin socket, excluding the first position, reserved for the 10F2xx, along with the "blue" jumpers

The source

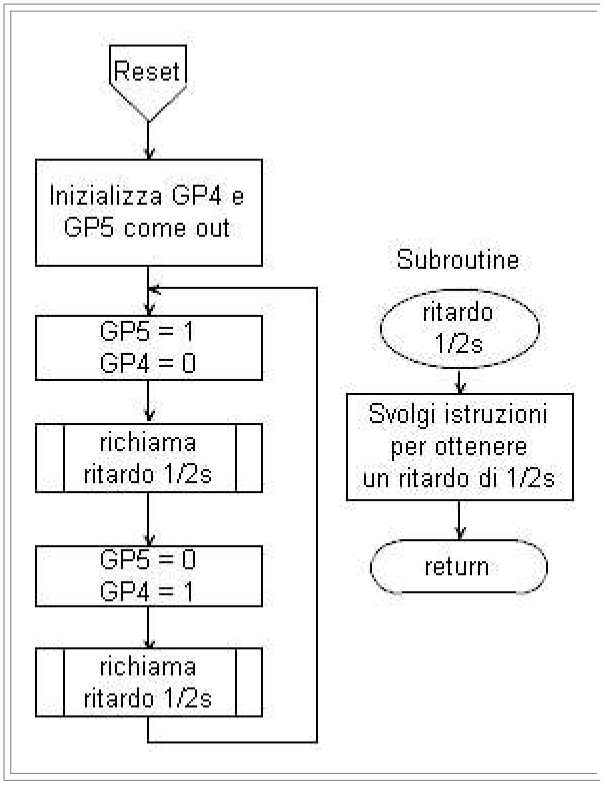
As always, the first thing is to be clear about what you want to achieve:

- **flash alternately two LEDs connected to GP5 and GP4 as soon as the voltage is given, at a frequency of about 1/2 Hz until the voltage is present**

And how:

- **programming the GP0 pin and GP1 pin as digital outputs and alternately turning them to high level for 1/2 second (LED on), then low level (LED off) for another half second, and so on, indefinitely (endless loop).**

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



On reset, **GP0** and **GP1** are programmed as digital outputs. **GP1** is set to level 1 (LED1 on) and **GP0** to level 0 (LED2 off).

The program makes a delay of 1/2 second, then turns off LED1 and turns LED2 on.

This is followed by another 1/2 second wait, then the program loops by repeating the LED1 on and LED2 off, and then the next steps indefinitely.

With this image, it becomes very simple to write the necessary instructions. That's why we use the **template** we've already seen.

Compared to what we saw in the second exercise, there is the addition of an additional LED to be controlled, while the overall structure remains identical, including the tempo subroutine.

Also in this example the source is made available already completely written, in the **3A_519.asm** and its MPLAB project.
Despite this, it is always necessary that we do not accept it as it is, but we make a detailed analysis of it, in order to make clear all its components and the reason for the various choices.

Of course, there will also be versions for 10F200/202 and 16F505/526.

R-M-W

Here we must introduce a further knowledge: that of the **RMW problem**.
This is quite a complex topic as it requires detailed knowledge of how I/O works.

In order not to weigh down this text, we refer to the link above, where there is a detailed explanation. A copy of this, however, can be found in the attachment for completeness.

To simplify, just know that Baselines and Midrange have a particular port access structure where the bits that control digital I/O are collected. Because of this structure, **when we use a pin as an output**, an unpleasant situation can occur: the **R-M-W** (***Read-Modify-Write) problem***. It arises from the way in which the internal logic of the PICs modifies one or more bits in the port. When we want to change the logic level to a pin, for example with:

```
 Bsf    port, pin   ; Raise the pin level to 1
```

This instruction operates in three steps:

- **R** - the **`PORTx register`** (or **`GPIO`**) is read, i.e. the real electrical state (logic level) of the pin is passed to the data bus
- **M** – the logic unit modifies the desired bit, performing the operation in an internal support register that is not otherwise accessible
- **W** – the contents of the support log are copied to the **`PORTx`** or **`GPIO`** t, i.e. written to the relevant Data Latch

This procedure takes place in a single cycle, using a period equal to 1/4 of the primary oscillator; Thus, with a clock of 4MHz, the instruction is completed in 1 microsecond. If the pin:

- A high inductance or capacitive load is connected.
- it is connected to an excessive load (e.g. an LED without current-limiting resistors).
- drives a transistor with no basic resistance or the gate of a MOSFET (which has a high capacitance).

A certain amount of time passes from the moment you apply a level change to the moment the pin actually reaches the desired voltage. Therefore, it is possible that if you make two successive changes to the value of the bits of the same port, the operation will not be performed correctly. All instructions that you want to change bits in the port later act in the same way.

Although not frequent, as two concurrent elements are required: a successive writing of two or more bits of the same port, for example:

```
   bsf GPIO, GP5
    bsf GPIO, GP4
```

of which the first connected a high load
The **R-M-W** problem really exists for Baselines and Midrange and **must be taken into account, otherwise there is a risk of creating automatisms that do not do what is expected, with possible damage to what is connected to the microcontroller.**

It should be noted that Enhanced Midrange and PIC18F implement a different set-up of the I/O control registers and are not affected by the problem described.

The phenomenon becomes noticeable as the instruction clock increases: at 20MHz clock, a cycle takes place in 200ns!! Under these conditions, even small capacities, such as that of the gate of a directly driven MOSFET, can create the problem we are seeing now.

In practice, there are two possible methods:

- introduce a wait between one bit modification and the next, for a time to allow the load to settle. For example,:

  ```
  bsf GPIO, GP5
  NOP
  NOP
  bsf GPIO, GP4
  ```

- rewrite the entire port in **one operation** (e.g., with the `movwf GPIO` instruction).

The first method, although simple, **is certainly not the most practical**, since it could be unacceptable to delay the sequence of bit changes and in any case it would be necessary to be sure that the imposed delay is sufficient, which is difficult to say and in any case not generalizable.

The second solution, on the other hand, is based on a simple consideration: if I have to modify for example 3 bits of `GPIO`, I do not execute a series of single instructions, with the risk of finding myself in the situation just described, but I modify the bits in an auxiliary register obtained from a RAM location, then I copy the RAM to the GPIO.
This modifies all the desired port bits in a single action. Despite the need to use a RAM location for each port, the method is the only one that can be practically generalised, i.e. it is good for any situation, although it involves minimal additional time.
The RAM location that supports this operation is called shadow.

Caution: The **R-M-W** problem affects I/O command register modification operations (`PORT` or `GPIO`), as irregular external loads can be connected to them. Therefore, the precautions described only apply to this case.
In the case of data RAM registers or SFRs, this is not a problem, as they do not have direct external connections and cannot be subject to it. Therefore:

```
    bsf GPIO, GP5
    bsf GPIO, GP4
```

is at risk and should be avoided, but:

```
    bsf RAM1,5
    bsf RAM1,4
```

It is not and has no problems. In the same way, a subsequent action on bits of different registers is not problematic.

SHADOW?

A **shadow** is a memory area, in this case 1 byte, that is a copy of the corresponding port. As we see below, writing to the port is done "indirectly" through the shadow.

Let's define the shadow

Although the RAM in the Baselines is very limited in extension, the use of a location for each port is usually not a problem. Then, in the RAM memory definition, we add the shadow for the GPIO.

```
####################################################################
;                          RAM
;
; general purpose RAM
   CBLOCK 0x07      ; start of RAM
     area sGPIO;     shadow of GPIO
     d1, d2, d3     ; Delay counters
     ENDC
```

The directive tells the compiler that there is this label-to-address relationship in the RAM data map:

Label	Address in RAM
sGPIO	07h
D1	08h
D2	09h
D3	0Ah

First we can update the source. If you want to use pin 4 as the **MCLR**, the configuration must be adapted to the request:

```
;                           CONFIGURATION
;
 __config _CP_OFF & _MCLRE_ON & _IOFSCS_4MHZ & _MCPU_ON & _WDT_OFF
```

We can complete the pin definition with a graphical view of the port situation. Unlike the previous tutorials, where we operated on the **GPIO** register, here we replace it with its shadow:

```
;*********************************************************************
;                DEFINITION OF PORT USE
;
;sGPIO map
;|  5  |  4  |  3  |  2  |  1  |  0  |
;|-----|-----|-----|-----|-----|-----|
;|     |     |     |     | LED | LED |
;D Shadow Finishes to Avoid RMW
;#define sGPIO,GP0 ;
;#define sGPIO,GP1 ;
;#define sGPIO,GP2 ;
;#define sGPIO,GP3 ; MCLR
#define LED2 sGPIO,GP4 ; LED tra pin e Vss
#define LED1 sGPIO,GP5 ; LED tra pin e Vss
```

We refer to the LED labels not as parts of the **GPIO**, but as part of the shadow, for ease of later use; they now indicate the bits related to the RAM location defined as **the sGPIO:**

Symbolic	Absolute
LED2 = sGPIO, GP4	= 7, 4
LED1 = sGPIO, GP5	= 7, 5

And these values are the ones that the compiler will use whenever it encounters this label.
As far as relative macros are concerned, they become:

```
;####################################################################
;                       LOCAL MACROS
;
; Command by and LED
; turns LED1 on and LED2 off
LED1     MACRO
       bsf    LED1              ; LED1 on
       bcf    LED2              ; LED2 off
       movf sGPIO, W            ; copia shadow in W
       movwf GPIO               ; copy W on port
         ENDM
  ; turns LED2 on and LED1 off
  LED2     MACRO
         bsf    LED2              ; LED1 off
         bcf    LED1              ; LED2 on
         movf   sGPIO, W          ; copia shadow in W
         movwf GPIO               ; copia W sul port
           ENDM
```

The macros are the same, with the only difference being the exchange of state of the LEDs.

We observe that the macros perform the modification operation through the shadow and only when this is modified, it is copied to the port in a single operation.

Of course, macros can be dispensed with, but they often prevent us from rewriting repeated sequences, which are defined only once in the source and, if in common use, can be collected in libraries to be included with the `#include directive`.

Remember that

1. **a macro MUST be defined BEFORE its label is used**

2. Conversely, the label of a subroutine can be invoked even if the subroutine itself will be defined later in the source. This is due to the fact that the Assembler is just a stone's throw away and keeps track of the labels and their function**.**

MOVF

MOVF is an acronym for ***Move File*** is intended to move files. We remind you that for Microchip file it indicates a location in the RAM area (data RAM or FSR).

Its syntax is the usual one of Microchip's opcodes:

[label]	sp	movf	, f/w	sp	object	sp	[; comment]

The initial label can be omitted, if it is not needed; The opcode, as usual, can't start in the first column anyway.
As already seen for **decfsz** , we observe that the opcode must have a "tail" that defines the destination of the file that is being moved:

- **movf,w** tells the compiler that the file in question should be copied to W
- **movf,f** tells the compiler that the file in question should be copied to itself

While the first option is easy to understand, given that the W register is the fundamental element for data movement, in this case the second option is less so.
What's the point of copying the data on itself? To understand its purpose, you should know that the **STATUS** registry is modified by this operation. Let's see something about it right after that. In general, it should be kept in mind that the transfer of the contents of files in the PICs takes place through the W register.
So, if we want to copy a numeric value to a file, we need the sequence:

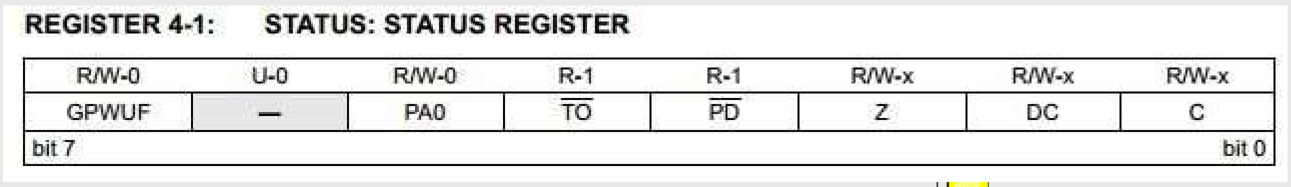
```
; Copy number in the register
      movlw number          ; Charge a number (literal) in W
      movwf destination ; copy W to the destination file
```

If we want to copy the contents of one file to another:

```
; Copy number in the register
      movf   sorgente,w     ; copy the source file to W
      movwf  destinazione   ; copy W to the destination file
```

Status

The **`STATUS`** register is one of the fundamental registers for the operation of the processor. This is his typical map in Baselines:



REGISTER 4-1: STATUS: STATUS REGISTER

R/W-0	U-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
GPWUF	—	PA0	$\overline{\text{TO}}$	$\overline{\text{PD}}$	Z	DC	C
bit 7							bit 0

In this family of chips, the **`STATUS`** contains both flags and control bits.

A ***flag*** is a bit that is used to signal an event.
The following are flags in the **`STATUS register`**:

- **`C`**
- **`DC`**
- **`Z`**
- **`PD`**
- **`TO`**

While **`PA0`** and **`GPWUF`** are control bits. We will see more details during the tutorials: for now we just need to know that the **`Z`** (Zero) bit is a flag that automatically changes its value depending on the result of an instruction.

Specifically, this occurs:

Risultato operazione	bit Z
= 0	1
≠ 0	0

That is, the **`Z`** bit goes to level 1 if the result of an operation is zero; otherwise it takes the value of 0. This is useful for testing the condition where the contents of a file are 0, without making comparisons.

```
        movf    Register,F        ; Copy Log on Itself BTFSC
        STATUS,Z                  ; Z bit clear ?
         Goto zero                ; No - register = 0
        Goto    nonzero           ; Yes - Register ≠
        0
```

In our example, the sequence:

```
        movf    sGPIO, W           ; Shadow copy to W
        movwf   GPIO               ; copy W on port
```

It is necessary, as we mentioned above, a pass through the W register to copy the contents of one register to another. So, you copy the contents of the register, in this case **sGPIO,** to the W and then copy W to the destination register (**GPIO)** .

The main loop is very similar to the one seen in the previous tutorial. We use the structure with the subroutine included through #include:

```
mainloop:                      ;<<--<<<--|
; turns LED1 on, LED2 turns              |
  off LED1                     ;         |
                               ;         |
; attesa 500ms 50 x 10ms                 |
  movlw   .50                  ;         |
  call    Delay10msW           ;         |
                               ;         |
; turns off LED1, turns on             loop
  LED2 LED2                    ;         |
                               ;         |
; attesa 500ms 50 x 10ms                 |
  movlw   .50                  ;         |
  call    Delay10msW           ;         |
                               ;         |
; loop                         ;         |
  goto    mainloop             ;>>-->>---|
```

The use of macros and subroutines, predefined blocks, makes the writing of the source very simple. The tempo routine is the one we've seen before, modulated by the content of W.

Once the source has been successfully compiled, simply transfer the .hex to the chip's memory to immediately have the connected LEDs flashing.

The MCLR Reset

All PICs have the possibility of acting on the reset with an external signal that arrives through the **MCLR pin** (**Master CL**eaR).

Within this tutorial we can verify its action.

MCLR is an externally generated reset, bringing the MCLR pin (which in these processors is identified with the GP3 pin) to a low level.

Its purpose is to put the processor in a RESET condition at any time, through a hardware command.

It should be noted that in most PICs the **MCLR** pin can also be used as a digital input, which is done by disabling the **MCLR** function in the initial config. It is necessary to be clear that this is only a digital input, since the pin does not have an internal buffer to control an output.

In addition, it is useful to know that the pin is also the input of the Vpp programming voltage: by applying this voltage (>10V) the internal logic sets the chip in programming mode.

Let's also try to clarify an important aspect, that of "resets".
And we talk about Reset in the plural as the PICs offer multiple possibilities of "reset", making the cause come not only from the MCLR pin.

We can list the main ones present in the Baselines:

- **POR** – ***Power On Reset***, which is done by applying the power supply voltage to the chip
- **MCLR** – ***Master Clear***, which is done by sending the corresponding pin to the low level
- Reset caused by an internal mode (***pin level change***)
- Reset caused by ***Watchdog Overflow*** (**WDT**)

Other processor families have a larger number of reset sources, which we'll look at in the relevant tutorials, including a specific instruction (`reset`), stack overflow/underflow, ***Power On Timer*** - **PWRT**, ***Brown Out Detector*** - **BOR,** etc.

It should be kept in mind that, although the most obvious aspect of the reset action is to return the execution to the vector at address 0000h, different reset sources have different effects on register defaults.
We will see during the next exercises to highlight these aspects that are generally overlooked, but that can become important if exploited appropriately.

For now, let us limit ourselves to clarifying a very little-known difference between the action of the ROP and that of the
MCLR.

MCLR vs. POR

There is a fundamental difference, not at all clear to most, between the automatic reset that is generated when the power supply voltage arrives (POR - Power On Reset) and the one produced by MCLR:

- **POR is generated by an internal module.**
  This module, which cannot be deactivated in any way, has the function of monitoring the arrival of the Vdd voltage and keeping the hardware locked until it has exceeded a certain level (defined on the data sheet, around 1-1.8V). This allows the integrated components to start with a minimum of voltage that ensures safe operation of every part of the complex structure, from the oscillator to the program counter mechanism.

- **MCLR is activated when the relative pin is set to a low level.**
  So it's an action that has to be generated outside the microcontroller and that can happen at any time. Here MCLR does not have internal pull-ups and should not be left disconnected.

The **RESET,** in any case, aims to:

1. "reset" the internal situation of the processor and bring it to a defined point, preparing the internal logic to the conditions necessary for the start
2. pre-set various bits of the internal registers (the so-called defaults **to the POR)**
3. Load the program counter with the value of the reset vector (**0000h**)

Once the cause of the RESET has ceased, the processor will start executing the program contained in its program memory.

It is important to be clear that:

1. **the external MCLR reset pin has nothing to do with the reset due to the arrival of the power supply, since this event is managed exclusively by the internal circuitry of the microcontroller**

2. **The reset pin does not have to be connected to a button. This is in case a manual reset action is required. MCLR is activated by going to low level, so the pin is controllable by logic, a transistor, a reset controller or anything other than necessarily a mechanical contact.**

3. **The reset pin does not normally have any function on a device made with a microcontroller and is not a requirement in any type of circuit**

4. **For this reason, in many PICs, the MCLR pin can be usefully disabled and configured as a digital input to add an extra I/O capability.**

Why claim that the MCLR reset pin has no function in a device? Simply because the microcontroller **MUST** work properly and therefore does not need to be "reset" to recover from error situations!

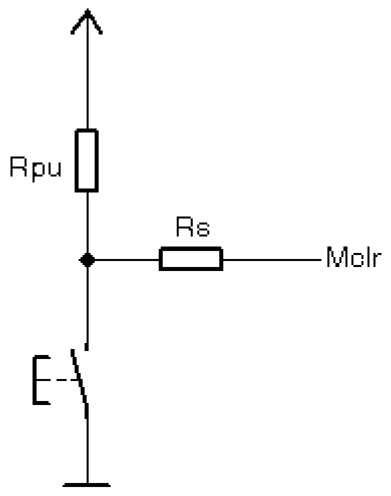
Microcontrollers govern automatic devices that **MUST** work and therefore the presence of the Reset button is meaningless.

In case you can think of the possibility of problems during the execution of the program (extreme complexity of the firmware, not properly verified, or possibility of external disturbances) other ways are implemented to prevent the program from getting out of control, the simplest of which is the activation of the Watchdog Timer present in all microcontrollers or the **BOR** module(Brown Out Reset) or other more complex ones, such as the stack overflow control, present in the PIC18F.

And, in any case, it will be necessary to implement **a management of the recognition of the causes of reset**, which is a situation of high anomaly, and of which a prompt correction is needed.

On the other hand, **MCLR** is useful in debugging operations, when the program is being developed, and also fundamental in the HV programming of the chip, being the pin on which the Vpp write voltage must be applied. It should be noted that this function is activated automatically and independently of how MCLR was configured, at the time of application of the Vpp.

If we need to perform a manual reset controlled by a button, the typical connection of an external contact is this:



Pressing the button brings the MCLR level to 0, connecting it to the Vss.

The Rpu pull up resistance has a typical value of 10k, but can very well be increased to 100k in low-power applications.

The Rs resistor, typically 1K, is not optional, but separates the button from the MCLR pin for two reasons:

- protect against ESD effects, and
- isolate the button from the Vpp applied to the MCLR pin through the ICSP connection.

There is no capacitor in parallel to the button that is present on almost all schemes, simply because it serves practically no purpose. Or rather, its purpose is to, together with Rpu, form an RC network that delays the achievement of the high level at the MCLR pin, prolonging the reset time. Where this is not needed (i.e. practically always), the capacitor could be understood as an "absorber" of the bounces of the button, but even here it is not an indispensable element. In this sense, nothing prevents you from putting the usual 10nF-100nF ceramic multilayer.
The correct solution, in case you need controlled resets in terms of time or voltage level, is to add externally components that perform these functions (reset controller, reset manager) in a well-defined way and can be integrated into the system project. Here you can find more pages on the use of the reset pin in practice. Most development boards have a similar connection, and our **LPCuB** is not exempt either.

The red RES button, at the bottom right, is connected to MCLR as we just saw. Next to it, a three-pin socket allows you to insert a simple reset controller (MCP111 or similar), which is essential in cases where the PIC does not integrate the Brown-out module, such as for Baselines. It is an integrated circuit that keeps its output at a low level until the supply voltage has reached a very precise value.

This is essential in cases where the PIC does not integrate the Brown-out module, such as for Baselines, cancelling the problems that could occur due to voltage drops such as not to fall below the minimum voltage of the POR, but which could still put the internal logic of the microcontroller in difficulty, generating random operations and instability.

If you want to evaluate the effect of the capacitor in parallel with the button, it can be inserted into the socket of the reset controller, using the two outermost contacts.

If you are working on breadboards, you need to add the button and the two resistors to the previous circuit.

Once the program has started and the LED is flashing as required, if we press the reset button, and as long as we hold it down, the two LEDs will turn off.
This is due to the fact that the low level applied to the MCLR blocks the operation of the processor, resets the I/O and various registers to default, and prevents the Program Counter from operating. On release, i.e. when MCLR reaches logical level 1, the program restarts from the reset vector instruction.

It is very important, therefore, to understand well that, **when we release the button, the microcontroller resumes its operation from the reset vector, as if we had removed and reapplied the power supply.**

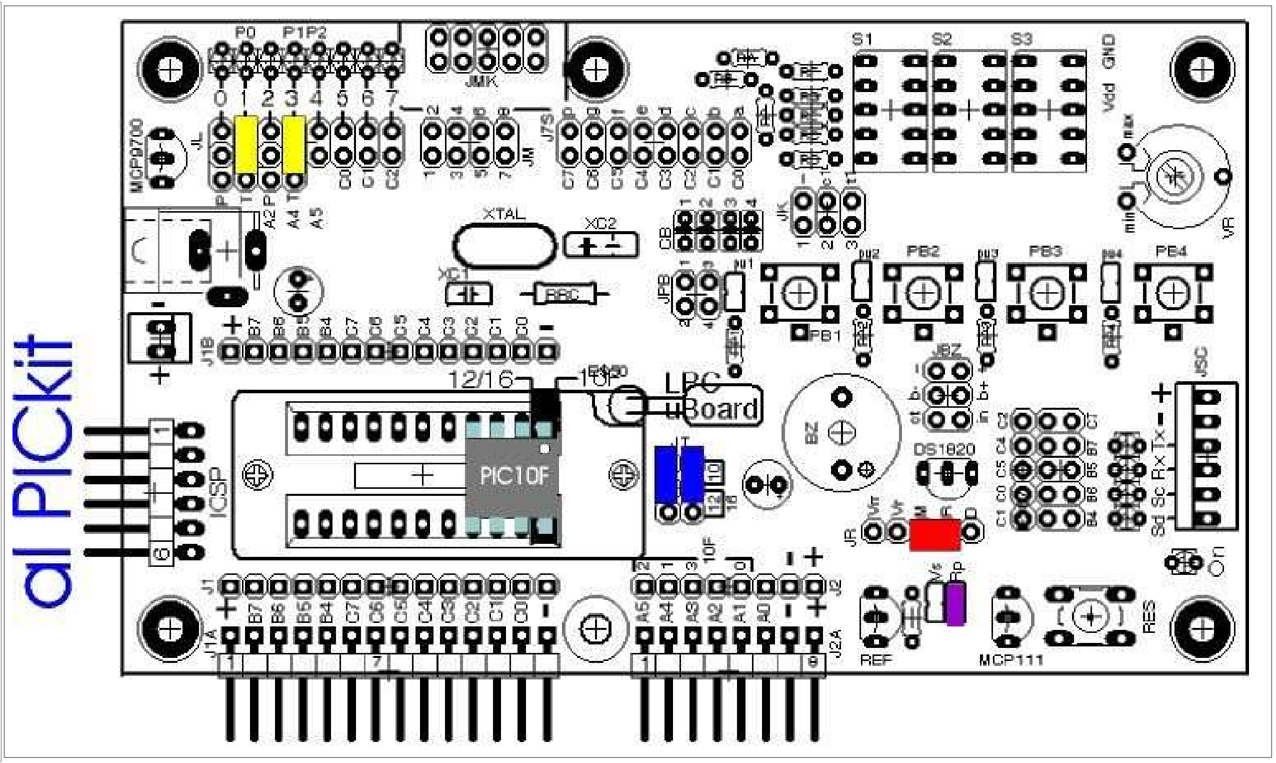
ATTENTION!

Care must be taken not to press the reset button while programming the chip. Forcing the MCLR pin to a low level during programming prevents it from completing correctly and risks damaging the PICkit, as it would short-circuit the applied Vpp to ground.

The resistor is inserted in series between the reset button and the MCLR pin in precisely for the purpose of avoiding damage to the tool or the board.

Other versions – 10F200/202

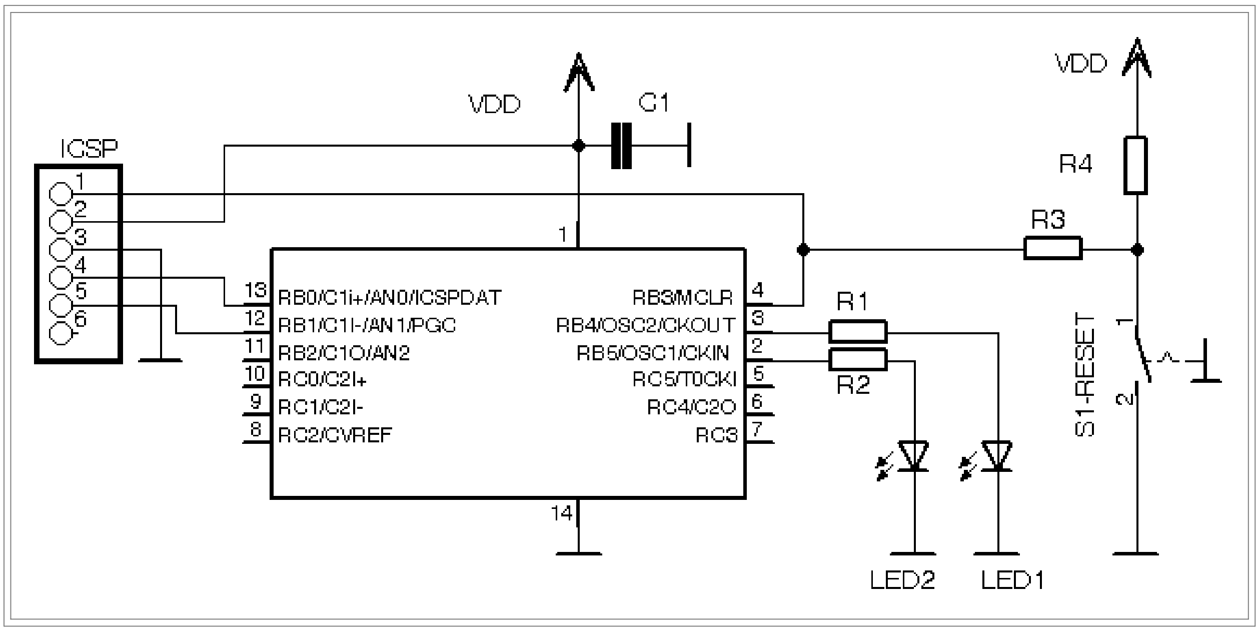
For the 10F2xx **series chips** we need to vary the placement of the chips on the socket and the relative position of the "blue" jumpers.

10F	 al PICkit PIC10F ICSP LPC uBoard XTAL DS1820 MCP9700 MCP111 REF RES S1 S2 S3 PB1 PB2 PB3 PB4 JMK J1 J2 12/16 10F

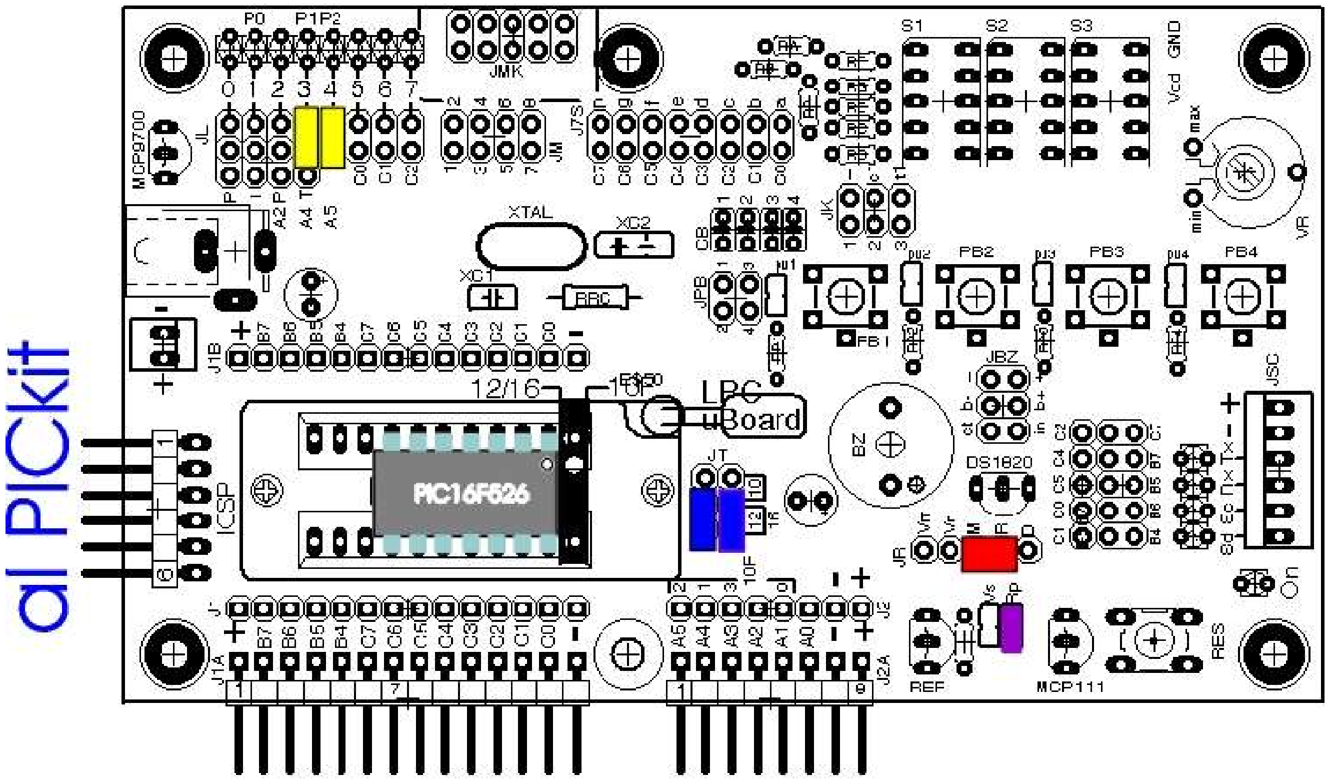
The different arrangement of the "yellow" jumpers is related to the use of `GP0` and `GP1`. The source is modified accordingly to suit the use of these pins.

Code 16F505/526

For 16F505/526 the wiring diagram is as follows:



We use the **RB4** and **RB5** **pins** to control the two LEDs.
This is the arrangement of the components on the development board:



Here, too, the variations on the source concern the changes already seen relating to the presence of PORT instead of GPIO and the use of the two pins indicated for the control of the LEDs.

12F519 - 3A_519.asm

```
;*********************************************************************
; 3A_519.asm
;---------------------------------------------------------------------
;
;    Title          : Assembly & C Course - Tutorial 3A_519
;                     Flashes alternately two LEDs with the
;                     cadence of 1s.
;    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
;    GP4/OSC2             3: Out   LED at Vss
;    GP3/! MCLR/VPP       4:
;    GP2/T0CKI            5:
;    GP1/ICSPCLK          6:
;    GP0/ICSPDAT          7:
;    Vss                  8: --
;
;*********************************************************************
;            DEFINITION OF PORT USE
;
; Shadow definitions
; sGPIO map
;| 5 | 4 | 3 | 2 | 1 | 0 |
;|-----|-----|-----|-----|-----|-----|
;| LED | LED | MCLR|     |     |     |
;
;#define    sGPIO,GP0     ;
;#define    sGPIO,GP1     ;
;#define    sGPIO,GP2     ;
;#define    sGPIO,GP3          ; MCLR
#define     LED1 sGPIO,GP4     ; LED between pin
                               and Vss
#define     LED2 sGPIO,GP5     ; LED between pin
;                              and Vss
; ####################################################################
```

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

        Radix     DEC

; ####################################################################
;                            CONFIGURATION
;
; Internal oscillator, no WDT, no CP, 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            ; shadow I/O
      d1,d2,d3              ; ENDC Delay Counters

; ####################################################################
;                        LOCAL MACROS
;
; Controls for LED1
LED       MACRO
        Bcf     LED2
        Bsf     LED1
        movf    sGPIO,w
        movwf   GPIO
          ENDM
LED2      MACRO
        Bcf     LED1
        Bsf     LED2
        movf    sGPIO,w
        movwf   GPIO
          ENDM

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

; MOWF Internal Oscillator
        Calibration OSCCAL

; ####################################################################
;                         MAIN PROGRAM
;
Main
     CLRF    GPIO         ; GPIO preset latch to 0

; TRIS         --001111 RB5/4 out
     movlw   b'11001111'
     Tris    GPIO         ; To the Management Register
```

```
;==================================================================


Mainloop:                ; <<--<<<--|
; Lights up LEDs         ;          |
     LED1                ;          |
                         ;          |
; Wait 100ms                        |
     movlw   .10         ; 10 x 10ms = 100ms
     call    Delay10msW ;           |
                         ;          |
; turns off LEDs                  Loop
     LED2                ;          |
                         ;          |
; Wait 400ms                        |
     movlw   .40         ; 40 x 10ms = 400ms
     call    Delay10msW ;           |
                         ;          |
; Loop                   ;          |
    Goto    Mainloop    ; >>-->>---|

;*******************************************************************
;                            SUBROUTINE

 #include C:\PIC\LIBRARY\Delay\Baseline\Delay10msW.asm

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

Code 12F508/509 - 3A_509.asm

```
;*******************************************************************
; 3A_509.asm
;-------------------------------------------------------------------
;
;    Title          : Assembly & C Course - Tutorial 3A_519
;                     Flashes alternately two LEDs with the
;                     cadence of 1s.
;    PIC            : 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
;    GP4/OSC2            3: Out   LED at Vss
;    GP3/! MCLR/VPP      4:
;    GP2/T0CKI           5:
;    GP1/ICSPCLK         6:
;    GP0/ICSPDAT         7:
;    Vss                 8: --
;
;*******************************************************************
;            DEFINITION OF PORT USE
;
; Shadow definitions
; sGPIO map
;| 5 | 4 | 3 | 2 | 1 | 0 |
;|-----|-----|-----|-----|-----|-----|
;| LED | LED | MCLR|     |     |     |
;
;#define    sGPIO,GP0    ;
;#define    sGPIO,GP1    ;
;#define    sGPIO,GP2    ;
;#define    sGPIO,GP3         ; MCLR
#define     LED1 sGPIO,GP4   ; LED between pin and
                                                 Vss
#define     LED2 sGPIO,GP5   ; LED between pin and
;                                                Vss
; ####################################################################

; Choice of processor
```

```
 #ifdef____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_ON & _CP_OFF & _MCLRE_ON

; ######################################################################
;                         RAM
; general purpose RAM
        CBLOCK 0x07         ; start of RAM
      sGPIO area            ; shadow I/O
      d1,d2,d3              ; ENDC Delay Counters

; ######################################################################
;                       LOCAL MACROS
;
; Controls for LED1
LED       MACRO
        Bcf     LED2
        Bsf     LED1
        movf    sGPIO,w
        movwf   GPIO
          ENDM
LED2      MACRO
        Bcf     LED1
        Bsf     LED2
        movf    sGPIO,w
        movwf   GPIO
          ENDM

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

; MOWF Internal Oscillator
        Calibration OSCCAL

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

```
; GPIO         --001111 GP5/4 out
     movlw   b'11001111'
     Tris    GPIO        ; To the Management Register

;===================================================================


Mainloop:                ; <<--<<<--|
; Lights up LEDs         ;          |
     LED1                ;          |
                         ;          |
; Wait 100ms                        |
     movlw   .10         ; 10 x 10ms = 100ms
     call    Delay10msW ;           |
                         ;          |
; turns off LEDs                  Loop
     LED2                ;          |
                         ;          |
; Wait 400ms                        |
     movlw   .40         ; 40 x 10ms = 400ms
     call    Delay10msW ;           |
                         ;          |
; Loop                   ;          |
    Goto    Mainloop    ; >>-->>---|

;*******************************************************************
;                          SUBROUTINE

 #include C:\PIC\LIBRARY\Delay\Baseline\Delay10msW.asm

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

Code 10F200/202 - 3A_20X.asm

```
;**********************************************************************
; 3A_202.asm
;----------------------------------------------------------------------
;
;    Title          : Assembly & C Course - Tutorial 3A
;                     Flashes alternately two LEDs with the
;                     cadence of 1s.
;    PIC            : 10F200/202
;    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: Out LED at Vss
;    GP0/ICSPDAT           5:   1: Out LED at Vss
;    NC                    6:      nc
;    Vss                   7:   2: --
;    GP3/MCLR/VPP          8:   6:
;
;**********************************************************************
;             DEFINITION OF PORT USE
;
; Shadow definitions
; sGPIO map
;| 3 | 2 | 1 | 0 |
;|-----|-----|-----|-----|
;| MCLR|     | LED | LED |
;
;#define    sGPIO,GP3         ; MCLR
;#define    sGPIO,GP2    ;
#define     LED1 sGPIO,GP1  ; LED between pin and
                                               Vss
#define     LED2 sGPIO,GP0  ; LED between pin and
;                                              Vss
; ######################################################################

; 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

; ###################################################################
;                         CONFIGURATION
;
; No WDT, no CP, MCLR
  __config _CP_OFF & _MCLRE_ON & _WDT_OFF

; ###################################################################
;                       RAM
; general purpose RAM
        CBLOCK 0x07         ; start of RAM
      sGPIO area            ; shadow I/O
      d1,d2,d3              ; ENDC Delay Counters

; ###################################################################
;                     LOCAL MACROS
;
; Controls for LED1
LED        MACRO
        Bcf     LED2
        Bsf     LED1
        movf    sGPIO,w
        movwf   GPIO
          ENDM
LED2       MACRO
        Bcf     LED1
        Bsf     LED2
        movf    sGPIO,w
        movwf   GPIO
          ENDM

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

; MOWF Internal Oscillator
        Calibration OSCCAL

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

```
; TRIS          --111100   GP0/1 out
     movlw   b'11111100'
     Tris    GPIO        ; To the Management Register

;====================================================================


Mainloop:                ; <<--<<<--|
; Lights up LEDs         ;         |
     LED1                ;         |
                         ;         |
; Wait 100ms                       |
     movlw   .10         ; 10 x 10ms = 100ms
     call    Delay10msW ;          |
                         ;         |
; turns off LEDs                  Loop
     LED2                ;         |
                         ;         |
; Wait 400ms                       |
     movlw   .40         ; 40 x 10ms = 400ms
     call    Delay10msW ;          |
                         ;         |
; Loop                   ;         |
    Goto    Mainloop    ; >>-->>---|

;********************************************************************
;                           SUBROUTINE

 #include C:\PIC\LIBRARY\Delay\Baseline\Delay10msW.asm

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

16F526/505 - 3A_526.asm

```
;*******************************************************************
; 3A_526.asm
;-------------------------------------------------------------------
;
;    Title          : Assembly & C Course - Tutorial 3A
;                      Flashes alternately two LEDs with the
;                      cadence of 1s.
;    PIC            : 16F526-16F505
;    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 at Vss
;    RB4/OSC2/CLKOUT            3: Out   LED at Vss
;    RB3/! MCLR/VPP             4:
;    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: --
;
;    [ ] only 16F526
;
;***************************************************************
;===============================================================
;            DEFINITION OF PORT USE
;
;P ORTC not used
;
```

```
; PRTB map
; | 5 | 4 | 3 | 2 | 1 | 0 |
; |-----|-----|-----|-----|-----|-----|
; | LED | LED | MCLR|     |     |     |
;
#define   LED2  PORTB,RB5    ; LED between pin
#define   LED1  PORTB,RB4    and Vss
;#define        PORTB,RB2    ; LED between pin
                             and Vss
                             ; MCLR
;#define        PORTB,RB2
;#define        PORTB,RB1
;#define        PORTB,RB0

; ######################################################################
;                          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, MCLR
 __config _IntRC_OSC_RB4 & _IOSCFS_4MHz & _WDTE_OFF & _CP_OFF &
_CPDF_OFF & _MCLRE_ON
 #endif

 #ifdef___16F505
; Internal Oscillator, 4MHz, No WDT, No CP, MCLR
 __config _IntRC_OSC & _WDT_OFF & _CP_OFF &
 _MCLRE_ON #endif

; ######################################################################
;                        RAM
;
; general purpose RAM
        CBLOCK 0x10          ; start of RAM
      sGPIO area             ; shadow I/O
      d1,d2,d3               ; ENDC Delay Counters

; ######################################################################
;                      LOCAL MACROS
;
; Controls for LED1
LED        MACRO
        Bcf     LED2
        Bsf     LED1
        movf    sGPIO,w
        movwf   PORTB
          ENDM
LED2       MACRO
```

```
        Bcf     LED1
        Bsf     LED2
        movf    sGPIO,w
        movwf   PORTB
          ENDM

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

; MOWF Internal Oscillator
        Calibration OSCCAL

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

; TRIS         --001111   RB5/4 out
     movlw   b'11001111'
     Tris    PORTB        ; To the Management
                          Register

;===================================================================
Mainloop:                ; <<--<<<--|
; Lights up LEDs         ;          |
     LED1                ;          |
                         ;          |
; Wait 100ms                        |
     movlw   .10         ; 10 x 10ms = 100ms
     call    Delay10msW ;           |
                         ;          |
; turns off LEDs                  Loop
     LED2                ;          |
                         ;          |
; Wait 400ms                        |
     movlw   .40         ; 40 x 10ms = 400ms
     call    Delay10msW ;           |
                         ;          |
; Loop                   ;          |
    Goto    Mainloop    ; >>-->>---|

;*******************************************************************
;                         SUBROUTINE

 #include C:\PIC\LIBRARY\Baseline\Delay10msW.asm

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

Appendix A – The R-M-W Problem in Baseline and Midrange

Users of **PICs** should be familiar with the problem of unwanted latching of the state of a digital I/O, called ***the R-M-W Problem***.
It occurs when you try to modify two or more bits of the same port, acting with successive instructions, for example:

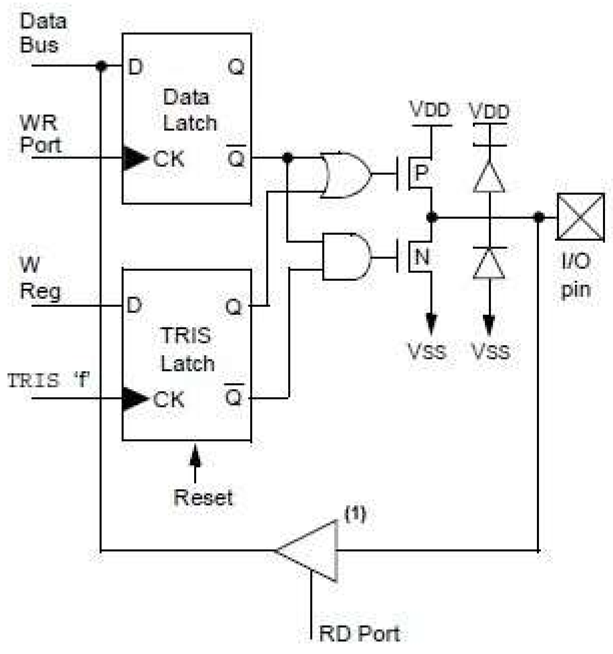
```
    bsf     PORTB, 1
    bsf     PORTB, 2
```

This results in the possibility that bit 0 is set, but bit 2 remains at zero.
Let's try to understand why this can happen.

In order to have a clear idea, however, you need to know how an I/O works.

How an I/O works

The diagram below shows the simplified structure of what lies behind a digital I/O pin in Baseline and **Midrange** (**PIC10/12/16**).



The ***Data Bus*** is connected to:

- a Data ***Latch***
- a direction latch (***TRIS Latch***)
- one read gate (***RD Port***)

The ***Data Latch*** is connected to the pin through a buffer, consisting of P- and N-channel MOSFETs. The buffer is enabled by the 0 layer coming out of the ***TRIS Latch*** and is disabled when it is 1; so this register determines whether or not the logical layer written to the Data Latch is transferred to the pin.

The value contained in the TRS Latch and Data Latch is taken from the data bus when a write is directed to the port (or gpio) and the TRIS (WR Port and WR Reg signals), respectively.

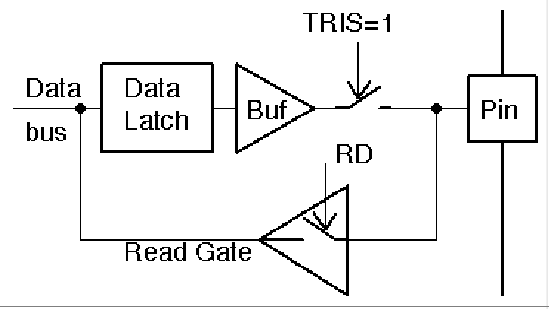
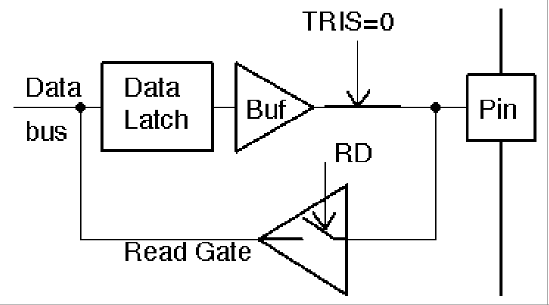
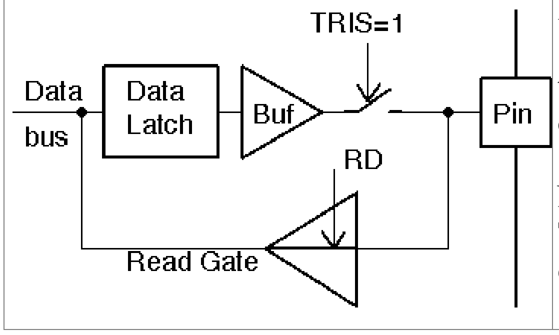
The read gate (***RD Port***) connects the pin directly with the data bus, transferring the voltage on the pin to it, which must obviously be at logical level 0 or 1 to be treated as data. It should be considered that latches are **RAM** memories, i.e. they retain their contents as long as the supply voltage is present or they are rewritten.

We also observe the fundamental fact that the content of the ***Data Latch*** and the ***TRIS Latch*** can be written, but not reread, as there is no gate that allows this. It should be noted, therefore, that:

- a write to the PORT register **writes the Data Latch**,
- a PORT reading **reads the true state of the pin**, **not the value stored in the latch**.

While this may seem like a minor difference, it is this situation that creates the problem, along with the mechanism that PICs implement to modify an I/O register.

For a better understanding, let's further outline the functions:

 TRIS=1 Data bus Data Latch Buf RD Pin Read Gate	By writing the PORT or GPIO, e.g. with `movwf GPIO` the data switches from the data bus to the Data Latch. If the TRIS is at 1, the buffer is disconnected from the pin and the logical level of the Data Latch cannot go outside (but remains stored in the latch): the set logic level is "present", but has no effect outside.
 TRIS=0 Data bus Data Latch Buf RD Pin Read Gate	If we bring the TRIS to 0, `movlw 0` `TRIS GPIO` the buffer is connected to the pin and the logic layer stored in the Data Latch appears on the outside
 TRIS=1 Data bus Data Latch Buf RD Pin Read Gate	If I request a reading, e.g. with `movwf GPIO,W` the RD signal activates the input gate and the pin is connected to the data bus. Note that **the PORT can be read with any value of the TRIS**, since the pin status is still read: TRIS simply enables and disables the output buffer.

Hence it follows that the I/O setting as an output refers to whether or not the buffer is connected to the pin and whether or not the contents of the Data Latch are reflected to the outside.

It follows that we can have different situations depending on the state of the **Tic-Tac-Toe bits**:

TRIS	Function	PORT	Note s

1	Digital Input	writing	When writing data to **PORTx** or **GPIO**, the data remains stored in the Data Latch until it is rewritten or power failure. There is no effect on the pin because **Tris=1** disables the output buffer
		reading	By reading **PORTx** or **GPIO**, the logic value applied to the pin is passed directly to the data bus
0	Digital Output	writing	**Tic-tac-toe=0** enables the output buffer. When writing data to **PORTx** or **GPIO**, the data is stored in the Data Latch until it is rewritten or power failure and passed to the pin buffer.
		reading	By reading **PORTx** or **GPIO**, the logic value applied to the pin is passed directly to the data bus

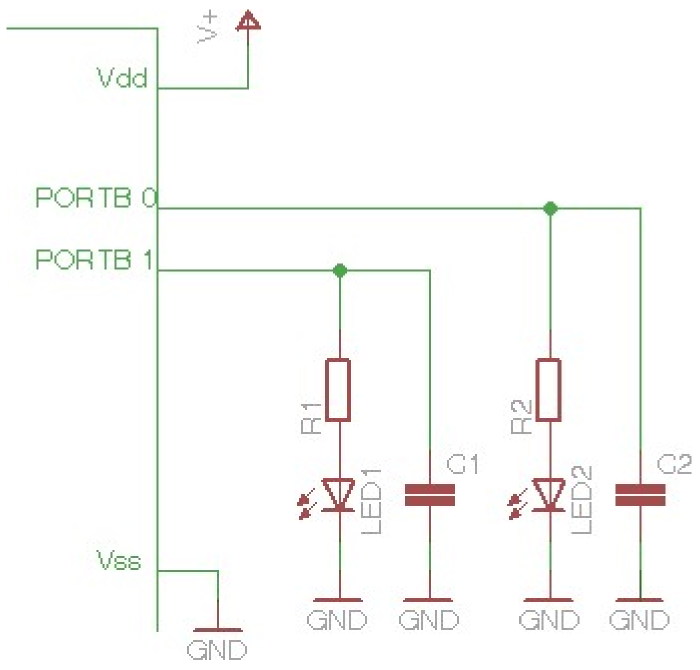
Let's recap the consequences:

- I can write any value to the **PORTx** or **GPIO** register, but this value is only reflected on the corresponding pin if the output buffer is enabled by **TRIS=0**.
  With **TRIS=1** it is still possible to write the Data Latch, which only has the effect of loading the data into the latch; if I load a data into the **GPIO register**, for example with **clrf GPIO**, this value is not lost, but remains in the latch until the power supply voltage drops or a new write. When you set **TRIS=0** this value would be reflected on the physical pin.
- On the other hand, it is very important to understand that the reading of **PORTx** or **GPIO** does not read the state of the Data Latch, but **the actual value that the pin assumes at the precise moment of reading** and which, as we will see, can be different from that of the Data Latch.
  The reading can be carried out independently of the value set in the **TRIS**, since, as we have just seen, the operation consists of enabling the reading gate and transferring the data on the bus.

How the R-M-W problem can arise

Let's look at the mechanism by which the R-M-W problem is formed.

From the principle diagram above, we see that a MOSFET totem pole buffer is interposed between the data latch and the pin. This buffer is capable of providing a current of 25 mA, and the semiconductors have an internal conduction resistance, albeit minimal.
25 mA is not an immense current, but it is more than enough for the purposes for which the chip was designed. However, it is quite possible to apply **less than ideal loads** to the pin, such as something capacitive.



We know that a discharged capacitor is, at the time of voltage application, the equivalent of a short circuit and that **it takes a time to complete its charge,** depending on the impedance of the voltage source and the capacitances.

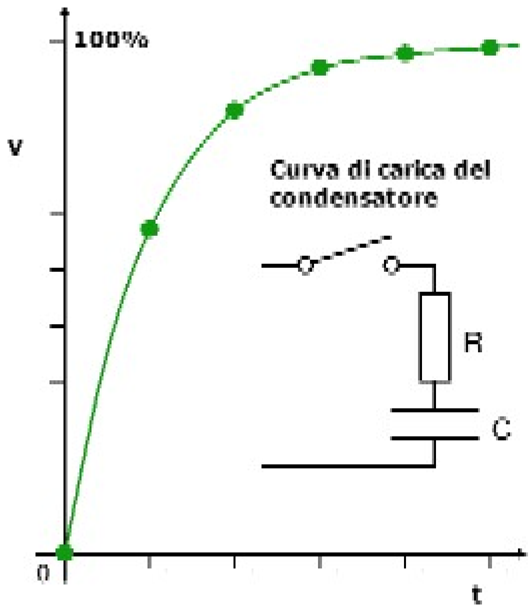
Then, by bringing one of the pins in the figure on the right to level 1, the LED will light up with a delay proportional to the capacitance of the capacitor, as this, in its charge, will hunt most of the current at the initial moment.

This means that, by measuring the voltage on the pin with an instrument, you will find yourself with a voltage-time curve typical of the charge of a capacitor.

From the time voltage is applied to the time the capacitor is charged to 100%, **a time elapses that is** determined by the impedance of the source (in this case the pin buffer) and the capacitance of the capacitor.

The **higher the capacitance** of the capacitor, the longer it takes for the voltage to stabilize. The resistance **R** is the conduction resistance of the buffer MOSFETs.

It may happen that the capacitive load of the pin is such that it takes longer for the voltage to take longer than the **instruction cycle** (which, remember, at 20 MHz is only 200 ns, while the charge of a capacitance could take micro or milliseconds) to get to level 1.



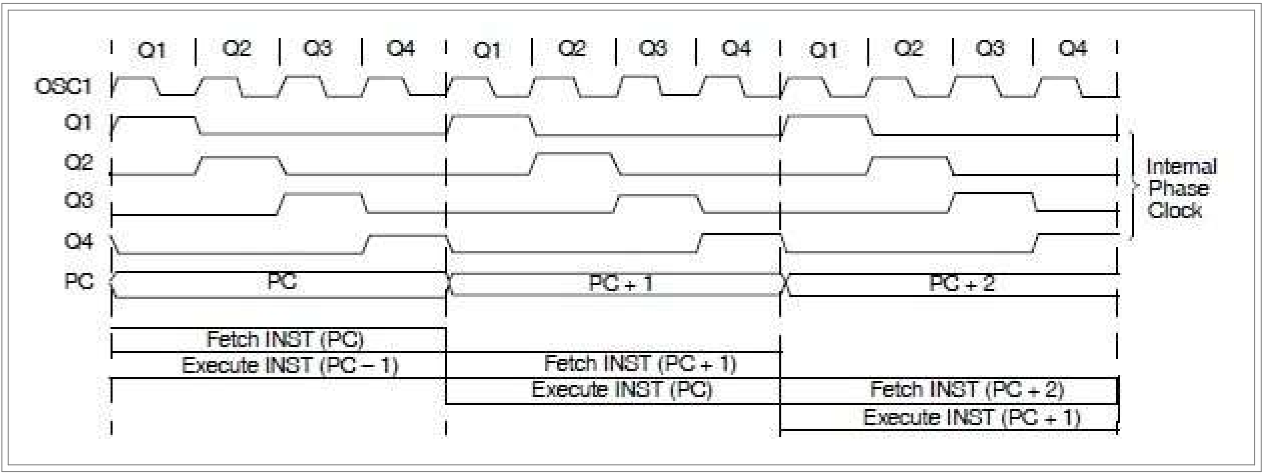
And if the instructions follow one another at a rate higher than the rate of change of the pin voltage, there is **a discrepancy** between the value set in the Data Latch and the real value on the pin itself.

There are two more points to be made.

The first concerns this fact: executing a read or write instruction directed to a specific bit of a port, such as `bsf PORTB,0` or `btfsc PORTB,0` **does not access only the indicated bit, but ALL the bits that are part of the port.**
So, for example, `btfsc PORTB, 0` does not read and verify only bit 0, but reads the entire port to which the bit belongs, which will then be the only one to be considered. This is because it is not possible, given the structure of the chips, to read or write a single bit, but it is necessary to pass through the data bus, which has a width of 8 bits, and which accesses all the bits of the port at the same time.
In this sense, the schemes of principle seen at the beginning can be misleading; You have to think of the single bit as part of a port. This constructive and logical structure is what makes the I/O, consequently, collected in logical groups (PORT or GPIO) with a maximum amplitude of 8 elements.

Thus, if the microcontroller has less than 8 I/O, for example 6 as in the PIC12Fxxx, the GPIO register will still consist of 8 bits, of which the two highest are not implemented. Conversely, if the chip has more than 8 I/O, they will be grouped into multiple PORTs, each with a maximum of 8 elements. Thus, in 16F84, where we have 13 I/Os available, we find them collected in two PORTs, one of 8 and the other of 5 (which, by the way, is an 8-bit data byte of which the three highest are not implemented).

The second point concerns **the mechanism for accessing and writing PORTs**. It is based on the internal clock which is equal to 1/4 of the frequency of the main oscillator.
The operation of modifying a bit (or a byte) takes place, like all one-cycle instructions, using 4 pulses of the primary clock. Thus, if the oscillator's quartz is 4 MHz, it will take 4 pulses to complete an instruction, whose effective cycle will be 1/4 of the clock, i.e. 1 MHz (***Fosc/4***).



The main clock (RC or external internal clock) is divided by four to generate four non-overlapping pulses, i.e., Q1, Q2, Q3, and Q4. The program counter is increased with each Q1. An instruction loop consists of four loops: the of instruction is retrieved and executed in a pipeline that treats one instruction while decoding and executing the previous one. An instruction that commits execution to a single loop is completed in the four phases, to which the fetch of the next instruction is superimposed in the pilpeline.

Exceptions are statements that require PC modification, such as `goto` and `return`, which require two loops because the pipeline, which had already stored the address of the next instruction, must be reloaded with the correct value imposed by the instruction.

Specifically, data memory is read during Q2 and written during Q4. So, behind the simple `bsf PORTB,0` we find, unseen, a complex action:

1. the read gate is enabled and all the pin status of the group to which the pin belongs (in this case **PORTB**) is passed to the data bus
2. this byte is **saved in an internal register**, which is not accessible except by the CPU mechanisms
3. The indicated bit (in this case, bit0) **is changed in the inner register**
4. the register is copied to the Data Latch

Hence the wording **R-M-W**:

1. **R** - read the port and copy to an internal temporary register
2. **M** - change of the read value with the desired one, always in the provisional register
3. **W** - copy of the modified provisional log in the Data Latch

Let's take a practical example: we want to bring bits 4 and 5 of GPIO to 1. External loads, such as two LEDs, are connected to the pins, but **a certain capacitance is also connected to the GP5**. The most obvious thing is:

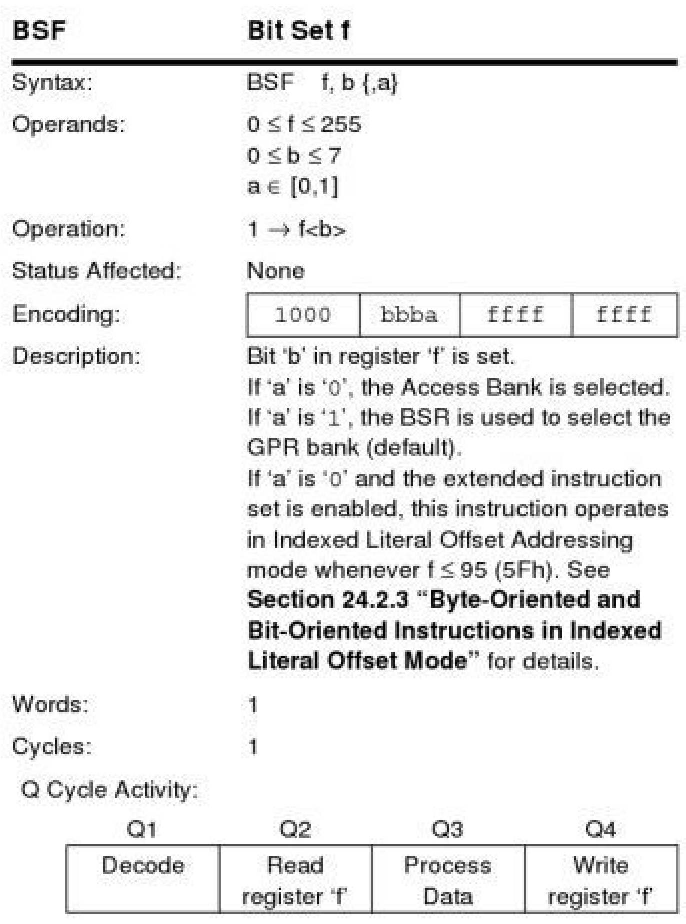
```
; IOpin set
    Bsf   GPIO, GP5
    Bsf   GPIO, GP4
```

But this may not be a good solution. With that in mind, let's see why.

Let's imagine for simplicity that **GPIO** is initially all at 0. The first line:

```
    bsf GPIO, GP5
```

does these things



BSF | **Bit Set f**

Syntax:	BSF f, b {,a}
Operands:	$0 \le f \le 255$ $0 \le b \le 7$ $a \in [0,1]$
Operation:	$1 \rightarrow f\langle b\rangle$
Status Affected:	None
Encoding:	1000 | bbba | ffff | ffff
Description:	Bit 'b' in register 'f' is set. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write register 'f'

- **Q1**: The bsf `instruction` is decoded
- **Q2**: The current status of the `GPIO = 00000000` is read and this value is saved in a temporary log
- **Q3**: Bit 5 is changed to 1; the temporary register will contain `00100000`
- **Q4**: Is the Tentative Log Copied to the Data Latch

This is completed in one cycle, which at 4 MHz takes 1 microsecond.

The **external capacitor** connected to the pin begins to charge at the maximum current that can be delivered by the pin and takes a certain number of microseconds, such as 3 us.

In the meantime, the processor has not stopped waiting for the capacitor to charge, but is executing **the next `bsf` instruction**:

```
Bsf    GPIO, GP4
```

- **Q1: The** bsf `instruction is decoded`
- **Q2**: **The current state** of the `GPIO` is read which is **00000000** , since bit 5 has not yet reached level 1 and this value is saved in the temporary register which will be 00000000
- **Q3**: Bit 4 is changed to 1; the interim register will contain **00010000**
- **Q4**: Is the Tentative Log Copied to the Data Latch

The LED connected to `GP4` will light up, while the LED connected to `GP5` will remain off !! What happened?

- **Q2** read **the actual status of GPIO** a few nano seconds after the end of the previous instruction.
- The capacitor **is still charging** and its level has not yet reached 1 (see curve in the diagram above).
- It is not the value saved in the Data Latch that is read, but **the logical level on the pin** at that precise moment and this level is not yet at 1 !!
- As a result, the change in Q3 does not operate on `001000000`, as you would expect, but on **00000000** (since the 0 bit state is not yet 1). So change results in **00010000** and not **00110000** and that's the value that's written to the Data Latch

The GP5 `pin` made an attempt to go to level 1, but the next `bsf` deleted it: the 5 bit setting is gone!!

This happens even if we change only one bit, for example:

```
; pulse on bit 5 bsf
    GPIO,   GP5  bcf
    GPIO, GP5
```

Given that the following happens:

```
   bsf GPIO, GP5
```

1. the read gate is enabled and all the state of the **GPIO** pins is passed to the data bus
   Since the logic level at pin 5 is at 0, the read makes **00000000**
2. This byte is saved in the internal register, which now contains **0000000**
3. bit 5 is changed in the inner register, which is now **00100000**
4. the register is copied to the **GPIO**, i.e. written to the data latch which is now changed to the value **00100000**

If bit 5 has no sensitive capacitances attached, the level at the pin rises to 1 in a very short time (ns). However, if a certain capacitance is attached to the pin that results in a time of more than 1us, the second instruction produces this:

```
    bcf GPIO, GP5
```

1. the read gate is enabled and all the state of the **GPIO** pins is passed to the data bus
   Since the logic level at pin 5 has not yet reached 1, the read yields **00000000**
2. This byte is saved in the internal register, which now contains **0000000**
3. bit 5 is reset to zero in the internal register, which is now worth **00000000**
4. the register is copied to the **GPIO**, i.e. written to the data latch which is now changed to the value **00000000**

The logic level of the pin does not change!!
If you have used this sequence to give a short pulse on the pin, for example to a strobe of a device and the input capacitance of this and the links generates an error, you will not have any output pulse.

Therefore, in Baselines and Midranges, consecutive instructions to change the state of the I/O of the same port must be avoided and a different approach is needed.

A Few Considerations

The R-M-W problem occurs only in cases where the capacitive load is too high compared to the change rate imposed on the port, or, more generally, when the load applied to the pin is excessive.
Possible cases are:

- capacitive load, e.g. RC network to eliminate the variable component from a signal
- Direct gate drive of large MOSFETs
- Directly connected inductive load
- Excessive current, e.g. LEDs without limiting resistance
- Long and messy outgoing wiring

The problem is proportional to the clock frequency. At 20MHz an instruction is executed in 200ns (200 billionths of a second!) and therefore the effect of even small capacitances is felt.

That is, the two key factors are the load capacity and the switching frequency.

Thus, in the classic command of logic gates, transistor bases for small signals, Logic Level MOSFETs, op-amps or buffers of the ULN2003 type or similar, used to drive displays or relays or other heavy loads that exceed 25mA of the direct capacitance of the pin, or even by controlling LEDs directly connected to the pin with the relative limiting resistance, Within the current limits indicated, the problem usually does not occur and no changes to the program need to be applied.
It is equally difficult to encounter problems when controlling small reed relays, LCD displays and similar low-capacity loads even when connected with short neat wiring.
For driving MOSFETs, however, it is recommended to use specific buffers (MCP1404 and similar) which completely eliminates any possibility of the problem even with non-Logic Gate components.
Note that controlling MOSFETs in PWM with the output pin of the CCP/ECCP module does not cause the problem, since the pin is driven individually by the module. It is not certain, however, that a simultaneous action of other bits on the same port does not create unexpected situations, even if there are no reports to this effect.

It can be concluded that, in general, in order to avoid problems, it is particularly appropriate, first of all, to connect only "regular" loads to the pins, using buffers as much as possible, which eliminate the problem. When this is not possible for any reason, and, in any case, in any case in which simultaneous outputs of bits are commanded on the same port, it is **always** the case to sufficiently verify the behavior of **your specific circuit** and act accordingly.
When in doubt, it is better to implement a software solution anyway than to risk getting into trouble, during operation, for not controlling a pin.

And this, not only for realizations intended for use outside the laboratory, but also in experiments, in order to get used to considering every aspect of the problems related to the management of loads on the outputs of the microcontroller, since, apart from absolutely extraordinary cases, the addition of the instructions necessary to correct the R-M-W problem does not involve significant slowdowns in the program, occupying very few cycles of education.

So let's see how to act.

Secure I/O Mission

Solution 1:

Keep the load capacity low enough not to cause the delay effect.

The D090 "`output-high voltage`" datasheet ***parameter*** declares a minimum output voltage of Vdd-0.7V for a current of 3 mA, and the `D080` "***output-low voltage***" parameter gives a voltage of 0.6V to draw a current of 8.5 mA. These voltage levels ensure that the pin input logics work properly, regardless of whether they are TTL or trigger.
In addition, the `D101` parameter limits the load capacity to 50pF in order to meet the correct timings. It should be noted that these parameters are temperature-dependent, which means that they should not be considered as general safety values, but as limits from which to take a safe distance.

If the load is adequate, the sequence of instructions:

```
; IOpin set
    Bsf       PORTB, 0
    Bsf       PORTB, 1
```

It will definitely work, without any problems, since the delay time of bit 0 will be much less than the instruction cycle. Essentially, you just need to connect non-capacitive loads (such as MOSFET gates, long cables, etc.) and current-limited loads to the pins. Where it is required to control these loads, simply **insert a buffer**.

Solution 2:

Use the **shadow-register technique**.
For each output register, a RAM register is reserved in memory to serve as a copy, and this shadow register is used for editing.

For example,:

```
; IOpin set
   Bsf    PB_Shadow,RB0 ; Set Bit 0 in Copy Register
   movf PB_Shadow,W      ; Copy the Shadow Log movwf
   PORTB ; in the PORTB
   Bsf    PB_Shadow,RB1 ; Set Bit 1 in the Copy
   Register movf PB_Shadow,W; Copy the Shadow Log
   movwf PORTB           ; in the PORTB
```

This works in any case and is the generally used solution, but it has, on the other hand, the increase in instructions and the need to use more memory.

Solution 3:

Use a software delay.
This means inserting a delay with instructions so that the cadence of actions on it

port is such as to allow the charging of any connected capacities.
For example,:

```
; IOpin set
   Bsf   PORTB, 0         ; set bit 0
   Call wait_10US         ; Wait 10 microseconds
   Bsf   PORTB, 1         ; Set Bit 1
   Call wait_10US         ; Wait 10 microseconds
```

This works, but it has, on the other hand, many elements:

- The increase in instructions
- slowing down the implementation
- **It is necessary to verify that the wait introduced is really able to avoid the problem, which depends on the attached load**
- The delay depends on the frequency of the oscillator and the routine may need to be modified

Solution 4:

Polling the status of the output.
After you make a change to the port, wait for the change to take effect before starting a subsequent change. For example,:

```
; IOpin set
   Bsf   PORTB, 0         ; set bit 0
chk_0
   BTFSS PORTB, 0         ; Wait until the pin is Goto
   chk_0                  ; Gone to level 1
   Bsf   PORTB, 1         ; Set Bit 1
chk_1
   BTFSS PORTB, 1         ; Wait until the pin is Goto
   chk_1                  ; Gone to level 1
```

This solution is similar to the previous one, but the time spent is optimized. Obviously, by commanding buffers with low-capacity input, software correction can generally be avoided.

Solution 5:

Switch between Baseline and Midrange to **Enhanced Midrange** and **PIC18F**. For these, in addition to presenting numerous other advantages, **the R-M-W problem does not exist, given the different structure of the port**, as we see in the following pages.

Final Notes

1. **The R-M-W problem is with the I/O command registers and not with the chip's internal registers**, which cannot have irregular loads. Then, the sequence:
   ```
   ; Set Bit
   ```

```
        bsf  GPIO,  GP5
        bsf  GPIO,  GP4
        bsf GPIO, GP2
```

2. is at risk, since success depends on the load attached to the pins, while this:

```
      ; Set bits in RAM
       data BsfRAM1, 5
        Bsf   RAM1, 4
        Bsf   RAM1, 2
```

3. It is successful, since it is an internal registry.

4. It should be understood that this **has nothing to do with the problem of glitch formation as a result of unwanted switching of the output pins** or with **the problem of debounce for input signals**.
   The former depend on how the output is handled. The second depends on the characteristics of the source of the input signal.