cccc

Knowledge of baseline

Indirect addressing, EEPROM

Let's get to know what indirect addressing is all about.
Also, since some Baselines integrate an EEPROM area, let's try to understand how it works.

Direct Addressing

In the RAM area there are two elements: the **actual Data RAM** (*General Purpose Registers*) and the **SFR** (*Special File Registers*) registers that control the various functions.

take a chip with a simple memory map, e.g. 12F508, we find that
tire area is contained in a single block, called the "*Bank*", as shown
diagram opposite, with addresses from 00h to 1Fh.

aselines, the bank has a width of 32 bytes. In

ıse, there are two elements:

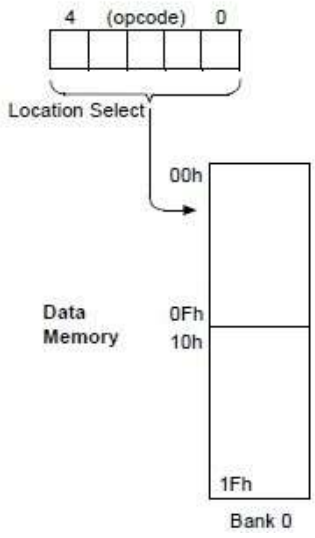
the first 7, from 00h to 06h are SFR
and the remaining 25 are data RAM.

To access the addresses of this memory area, we use the way that is called "direct", that is, simply indicating as the object of the instruction the location on which we want to operate:

```
 CBLOCK 0x10
  temp              ; Temporary RAM location
 ENDC
; copy the contents of GPIO to RAM
    movf   GPIO,w
    movwf  temp
```

zz

Direct Addressing



Recall that the Baseline opcode has a width of 12 bits.

In data management instructions (**bcf**, **bsf**, **movf**, **mowf**, **subwf**, **addwf**, **btfsc**, **btfss**, **decfsz**, **incf**, **decf**, etc.), the opocode contains both the code of the operation to be performed (in the 7 heaviest bits, bit11:6), while the 5 heavier bits (4:0 bits) have the purpose of to contain the address of the RAM location to be reached.

Since the RAM area is only 32 bytes, 5 bits can cover all addresses, from 00h to 1Fh.

So there is no problem in handling the data bytes and SFR, without special care or mechanisms to be respected.

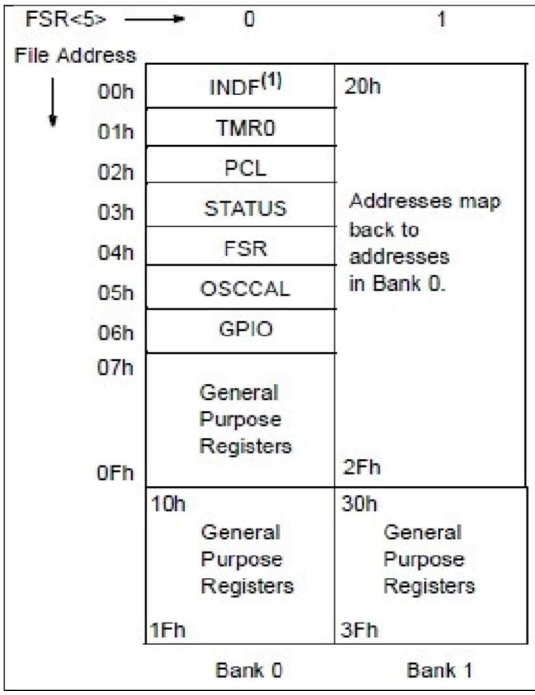
From an operational point of view, the task of creating the appropriate opcodes from source is delegated to the compiler. So, for example, if the source contains the line:

```
movwf     OSCCAL
```

The Assembler, taking the numeric value of the **`OSCCAL label`** from the file ***processorname.inc***, produces ***the value*** 025h in the **.hex** file. This stems from:

11	10	9	8	7	6	5	4	3	2	1	0
movwf instruction code : **02x**							OSCCAL Address : **05h**				
0	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**1**

If this opcode form is the same for all baselines, it should be noted that not all of them have the same amount of RAM.



Se consideriamo una altro chip, come 12F509, ci troviamo con una diversa mappa di memoria:

- gli SFR sono sempre 7, ma
- la RAM dati si estende a 41 bytes

Poiché la dimensione del banco è 32 bytes, occorro **DUE** banchi, chiamati **Bank0** e **Bank1**.

Qui, la RAM è divisa in 5 parti:

1. da 00h a 06h : 7 SFR
2. da 07h a 0Fh : 9 bytes di RAM dati
3. da 10h a 1Fh : 16 bytes di RAM dati
4. da 20h a 2Fh : una copia di1 e 2

5. da 30h a 3Fh : 16 bytes di RAM dati

The area from 00h to 1Fh, 32 bytes, is bank 0 and the area from 20h to 3Fh is bank 1. You may ask: since the opcode contains only 5 bits of address, how is it possible to access the contents of the second bank?
To clarify, suppose we want to reach the RAM location that has address 30h: this number, to be expressed in binary form, requires 6 bits:

10h = 1 0000
30h = 11 0000

An extra bit is needed to complete the address, and this sixth bit, external to the opcode, is provided by the **SFR** register, to be precise by its 5 bit. So, if I write:

```
movwf    0x10
```

the WREG content will be copied to the 10h location of bank0. To copy it in the 20h location, in bank 1, I will have to write:

```
bsf      FSR,5    ; Access Bank 1
movwf    0x30
```

However, it is necessary to understand that, if I wrote:

```
bsf      FSR,5
movwf    0x10
```

The effect would be the same: the address contained in the opcode is still 5 bits. So, writing 10h (10000) or 30h (110000) is the same thing, since bits beyond the fifth have no place in the opcode anyway. The additional bit depends on the value of **FSR,5**; the lowest 5 bits are the same for both addresses:

10h = 1 0000
30h = 11 0000

That is, the bank switching bit is at 0, all operations, however addressed, are directed to the Ram in bank 0; If I set the bit to 1, I address all trades to bank 1.

In this regard, it should be remembered that **at the ROP the FSR bit 5 is set to 0, making bank 0 accessible by default**; where more RAM is needed, moving to the next bank requires bit manipulation.

Therefore, if the program does not use more RAM than that contained in bank0, the problem of switching banks does not exist.
On the other hand, we have to deal with banks when the need for data RAM exceeds what is available in bank0 or when part of the SFRs are located in banks other than bank 0, as we see later in relation to chips with a large number of integrated resources, which need control registers and which do not fit in the 32 bytes of a single bank.

Since our rule is not to use absolutes, the correct procedure is to assign labels to the addresses and use the banksel pseudo-opcode provided by MPASM which acts on the bit according to the position in the banks of the label in question:

```
mem equ 0x30
  Banksel  mem          ; Switch to the counter that contains memes
                                                          (bank1)
  movf     mem,w
  Banksel  GPIO            ; switch to the bank that contains GPIO
                                                          (bank0)
  movwf    GPIO
```

It should be noted that, in order to make the programmer's work less difficult, the SFRs are made accessible in the same way from both desks; and this is also valid for a small portion of data RAM, from 07h to 0Fh. These are 9 bytes called Shared RAM (SHR).
This solution is adopted to reduce the need to continuously switch from one bank to another when using, for example, the memory in bank 1: if the SFRs were not accessible there as well, there could be the need for continuous switching between banks, which would penalize the speed of execution of the program, as well as occupying memory with additional instructions. In general, therefore, the technique is to use as much as possible the RAM resources given in bank 0 and move to the next banks only if necessary, but, once in a bank, try to stay there as much as possible in order to minimize the need to handle the switching bit. The shared memory acts as a "bridge", since it is accessible in the same way from both banks. So it works properly

```
Banksel  1             ; Bench 1
Movf     mem,w         ; Copy of the address 0x30
movwf    GPIO          ; in GPIO
```

since GPIO is identically accessible in both banks. And also:

```
  Banksel  1           ; Bench 1
  movf     TMR0,w      ; copy TMR0 content to W
  movwf    0x07        ; Copy with address 07h
  movwf    0x10        ; Copy with address 30h
  movf     0x08,w      ; copy contents of address 08h in W
  movwf    GPIO        ; in GPIO
```

GPIO, **TMR0** and shared RAM (07h, 08h) can be accessed from any bank, while directing 10h to bank 1 actually acts on 30h.

In any case, it is advisable, as a guideline in the drafting of the source, when it is necessary to use the switches of the banks, to keep the situation under strict control.
One way is to organize the use of RAM in such a way as to reduce the need to switch from one bank to another, to maintain as much as possible the operations on bank0 or with the help of shared RAM and, by moving to another bank, then return to bank0 in such a way as to have a constant reference.

A note for small PICs with only one bank, as in the case of the 12F508: if we insert a line like `banksel 0` in the source we get a message from the compiler that warns us of its uselessness:

Message[312] C:\PIC\PROVA.asm 39 : Page or Bank selection not needed for this device. No code generated.

This is correct, as the RAM in the 12F508 has no banks and no switching instructions are required.

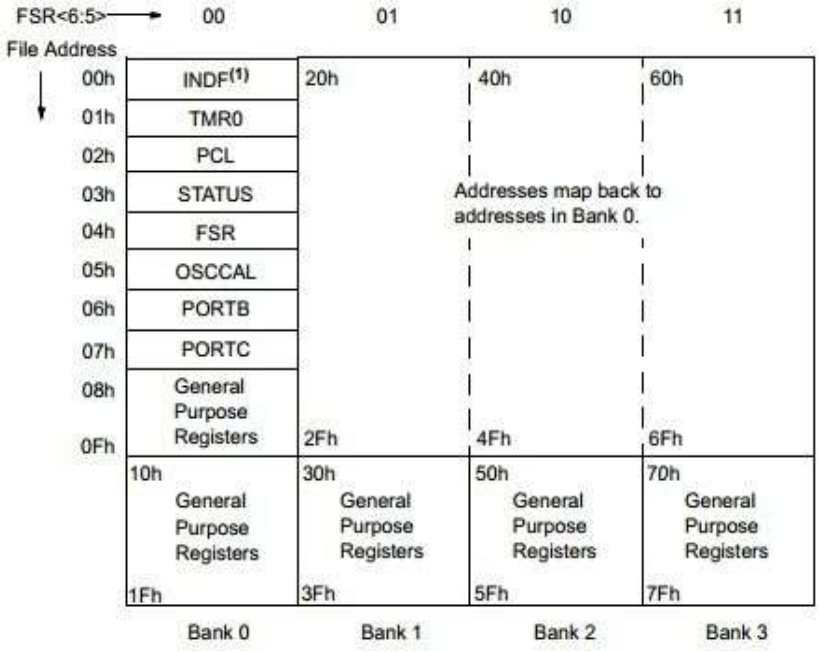
Since it is a "Message", the compilation is successful and the ***.hex*** file is generated: the purpose of the Message is to alert the programmer of an event following which the Assembler has made an automatic decision.

It must also be considered that there are other chips with more complex memory maps.

For example, 16F505 has 72 bytes of data RAM that requires 4 banks of 32 bytes each to contain it along with the 8 SFRs. Here, the additional bits to pass between banks are 2 and they are always bits of the FSR register, to be exact bits 5 and 6.

To clarify, let's say you want to reach the first location of the General Purpose Registers of each bank: these are the addresses 10h, 30h, 50h and 70h:

10h = 1 0000
30h = 11 0000
50h = 101 0000
70h = 111 0000



The banksel **pseudo-opcode** supported by MPASM autonomously manages the bits for changing the bank in relation to the chip you are working on:

```
temp0 = 0x10        ; in bank 0
temp1 = 0x30        ; in bank 1
temp2 = 0x50        ; in bank 2
temp3 = 0x70        ; in bank 3
; Copy the contents of temp0 to other locations
   movf  temp0, w
   banksel temp1
   movwf   temp1
   banksel temp2
   movwf   temp2
   banksel temp3
```

```
    movwf    temp3
```

where the pseudo-instructions correspond to:

Opcode	action
`banksel temp0`	`bcf FSR,5` `bcf FSR,6`
`Banksel Temp1`	`bsf FSR,5` `bcf FSR,6`
`Banksel Temp2`	`bcf FSR,5` `bsf FSR,6`
`Banksel Temp3`	`bsf FSR,5` `bsf FSR,6`

We remind you that by writing:

```
temp0 = 0x10        ; in bank 0
temp1 = 0x30        ; in bank 1
temp2 = 0x50        ; in bank 2
temp3 = 0x70        ; in bank 3

; Copy the contents of temp0 to other locations
     movf   temp0, w
    banksel temp1
    movwf    temp0
    banksel temp2
    movwf    temp0
    banksel temp3
    movwf   temp0
```

You get the same result, since the address of **`temp0`** and subsequent is the same for the low 5 bits.

Therefore, we can conclude that:

- Direct addressing is what we use when writing opcodes
- We have to manually "adjust" the bits that switch the banks in order to access the resources of the different banks

---

Indirect Addressing

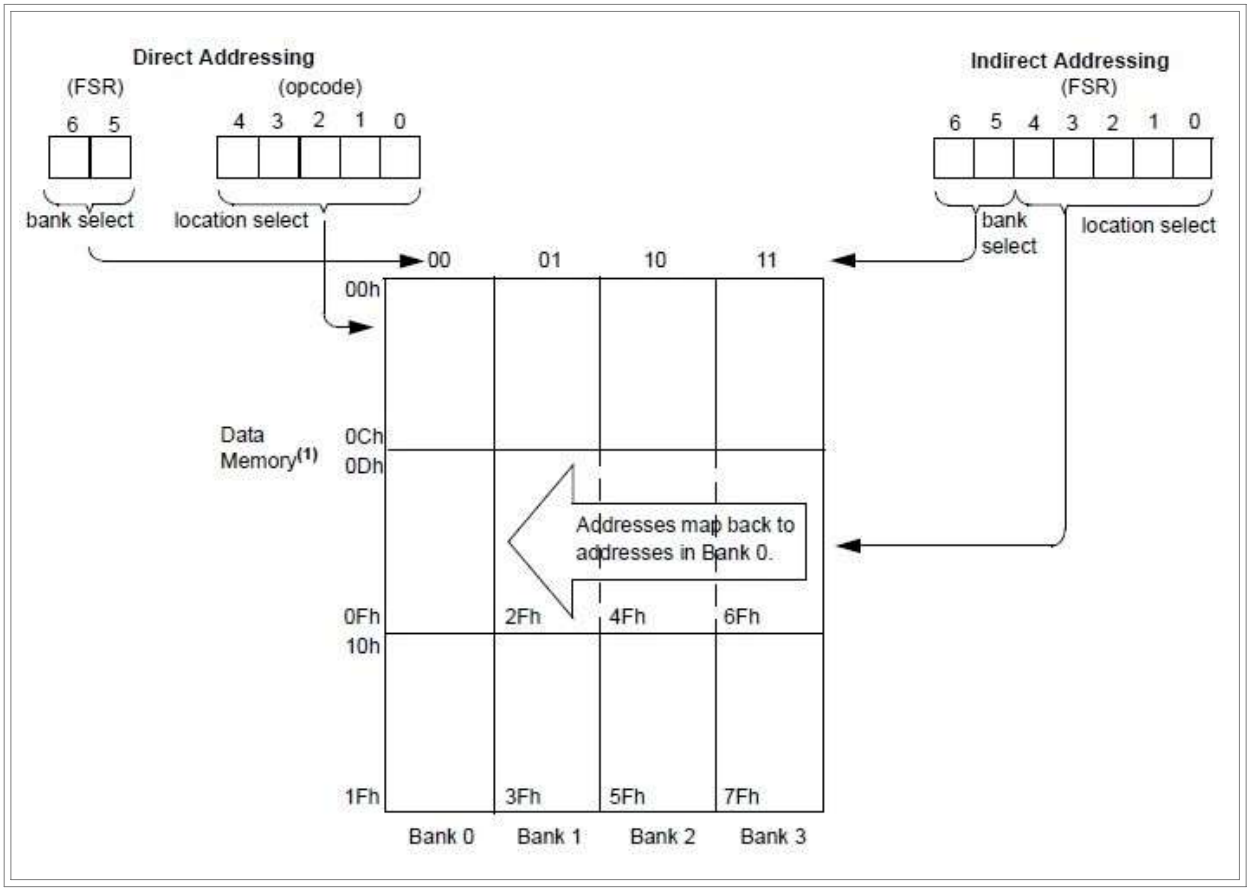
However, the heavy use of switches between banks exacerbates the number of instructions that the processor has to execute, as well as easily creating errors in register selection.

This is one of the reasons why PICs have an address called "indirect" that allows access to the contents of the banks without acting directly on the FSR bits.

This system uses the contents of the **FSR** (*File Select Register*) register as a pointer: since FSR is 8bit wide, addresses in RAM area from 00 to 255 (FFh) can be represented, which far exceeds what is physically available.
In fact, the FSR register of the 16F505 has 7 bits implemented, which allows it to reach all four banks; that of 12F509 has 6, which allows you to reach the contents of the two banks. The SFR of the 12F508 has only 5 bits implemented, since you don't need more than that, as it has no banks.

In the indirect way, the INDF (*Indirect File*) register is used: if we enter the address to be reached in the FSR register, the content of the addressed location will be automatically accessible through the INDF register both in reading and writing.

Let's see how the mechanism works: in the diagram below we have a graphic view of direct and indirect addressing.
In the first, given to the opcode the 5 least significant bits of the address to be reached, we complete it with the FSR bits. The destination will be reached by the opcode.
In the second case, we use the entire FSR as the destination address, which will be reached through the "indirect" INDF register.



It should be noted that the content of FSR is retained until the system is shut down and can only be changed by a rewrite; therefore, once a bank is selected through FSR, it will remain active until bits 5 and 6 are modified.

Therefore:

```
temp0 = 0x10        ; in bank 0
temp2 = 0x30        ; in bank 2
temp3 = 0x70        ; in bank 3

; Clear the contents of the logs
   movlw    temp3, w    ; in bank 3
   movwf    FSR
   clrf     INDF
   movlw    temp0, w    ; in bank 0
   movwf    FSR
   clrf     INDF
   movlw    temp2, w    ; in bank 2
   movwf    FSR
   clrf     INDF
```

This allows us to access any RAM location, without explicit bank switching instructions, which can be a significant advantage in many cases.
In addition, it is not necessary to know the bank where a specific register is located, since it is sufficient to indicate the relative label.

It should be noted **that the modification of FSR acts in a more general sense on the selection of banks; in the example above, after the last statement we find ourselves in Bank2.**

Another important feature that the indirect addressing mechanism enables is the ability **to operate on RAM blocks, tables, and arrays in RAM, using FSR as a pointer**. For example,:

```
; reset bank RAM 0 Clrbank0

        movlw   0x10        ; RAM start address of Bank0
        movwf   FSR
clrloop clrf    INDF
        incf    FSR,f
        btfss   FSR,7       ; Address > 1Fh ?
         goto   clrloop     ; No - other location
        ...                 ; end loop
```

FSR is incremented with each loop, so that **INDF** can be accessed to subsequent locations. The check for the end of the loop is performed on bit 5 which becomes 1 when the increment changes the FSR content from 1Fh to 20h. If you want to limit the area to be zeroed to different addresses, you will use a comparison with an **xorwf** or a **subwf**.

We will see later an application of what has been said.

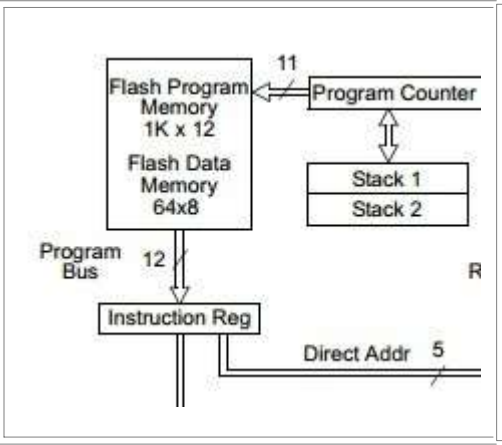
Now let's look at something about EEPROM.

EEPROM

The Baseline Flash is written during the programming of the chip and is only accessible as program memory. This is not accessible as the object of the instructions, which are directed, instead, to the data memory that is in RAM. Moreover, if the Program Counter mechanisms allow you to read the contents of the program memory on the address bus, which is intended as a list of instructions, to write this memory you need a voltage Vpp, higher than Vdd, and which is around 13V nominal. Therefore, for Baselines, during the execution of the program there is no way to alter the content of the Flash (also because the Flash contains the program, which must not be damaged by any possible action of the program itself).

However, the content of the RAM memory is lost when the power supply is lost, while the content of the program memory remains unchanged until a subsequent programming.
In various applications, it would be useful to have non-volatile memory to use for data, such as in data logger applications. For this reason, Microchip has made it possible to insert a small non-volatile memory area that can be read and written by the program as if it were data memory.
The advantage of this option is that it is possible to enter data during the execution of the program and that will not be lost when the power is lost. This data will be readily available upon reboot and can be updated, modified, etc. In addition, it is possible to pre-load, during the programming phase of the chip, the contents of this memory in such a way as to assign a series of "default" data to the chip.

Although the common name for this memory is EEPROM (Electrically ***Erasable PROM***), this is an inaccurate designation, left over from the period when a reprogrammable memory without extracting it from the circuit was made of a different technology; it was developed on the basis of pre-existing EPROM technology. In current PICs, on the other hand, it is a portion of Flash memory, the same used to store program memory, more accurately referred to as **Flash Data Memory**.



We can see it if we look at the block diagram of a PIC containing EEPROM: we see that it is located in the FLASH area.
This means that **it is accessible through the instruction bus (program bus) and not the data bus** (we know that in the Harvard architecture PICs, the two buses are separate).

It should also be noted that, in the Baselines, the "byte" of the Program Bus is 12 bits wide, but the Flash Data Memory is only 8 bits wide, in order to be processed by the data bus,

The mechanism implemented by Microchip to access the Flash Data Memory from the program involves an "indirect way", necessary to overcome the two different aspects that become part of this operation.

The first is to make accessible to the data bus an area that is the responsibility of the address bus.
The second is the need for a voltage higher than the Vdd to write the Flash.
The solution of these two problems involves a relative complexity of access to the EEPROM area, especially in writing.

As far as the voltage is concerned, it is not difficult to generate it internally on the chip with charge-pump circuits, but the operation still requires much more time than it takes to write a RAM memory cell, which is accessible at the rhythm of the processor's clock. Moreover, it is independent of this: whatever the clock frequency, the time it takes to write the Flash Data Memory is the same, since it depends on the characteristics of the Flash. To put it in perspective, while the data RAM is accessible, clocked at 20MHZ, in an instruction cycle, i.e. 200ns, the Flash requires the same reading time, but 3-5ms to be written.

As far as the transition from the address bus to the data bus is concerned, it must be considered that the EEPROM area is not directly accessible from the instructions; The mechanism adopted involves the use of some intermediate auxiliary registers, which have some analogy with indirect addressing: here too one register is used as a pointer and another as a means for the passage of data between the two buses.

This construction complexity (in addition to the cost) is not well suited to Baselines and its implementation is limited to only a few models:

PIC	EEPROM bit
12F519	64
16F526	64
16F527	64
16F570	132

In these chips, the EEPROM is to be considered as an additional function module and has three registers available for its management. These are:

- **EECON** Audit Log
- **EEDATA** Data Exchange Log
- **EEADR** Addressing Register

Basically, we can access the EEPROM only indirectly, through a data register (**EEDATA)** , an address register (**EEADR**) and a control register (**EECON**) that allows you to manage the operation you want to perform.
It is as if, not being able to make a commission directly, we are forced to use an intermediary.
It is clear that we need to be aware of the use of these dedicated registers. And here arises a not insignificant problem.

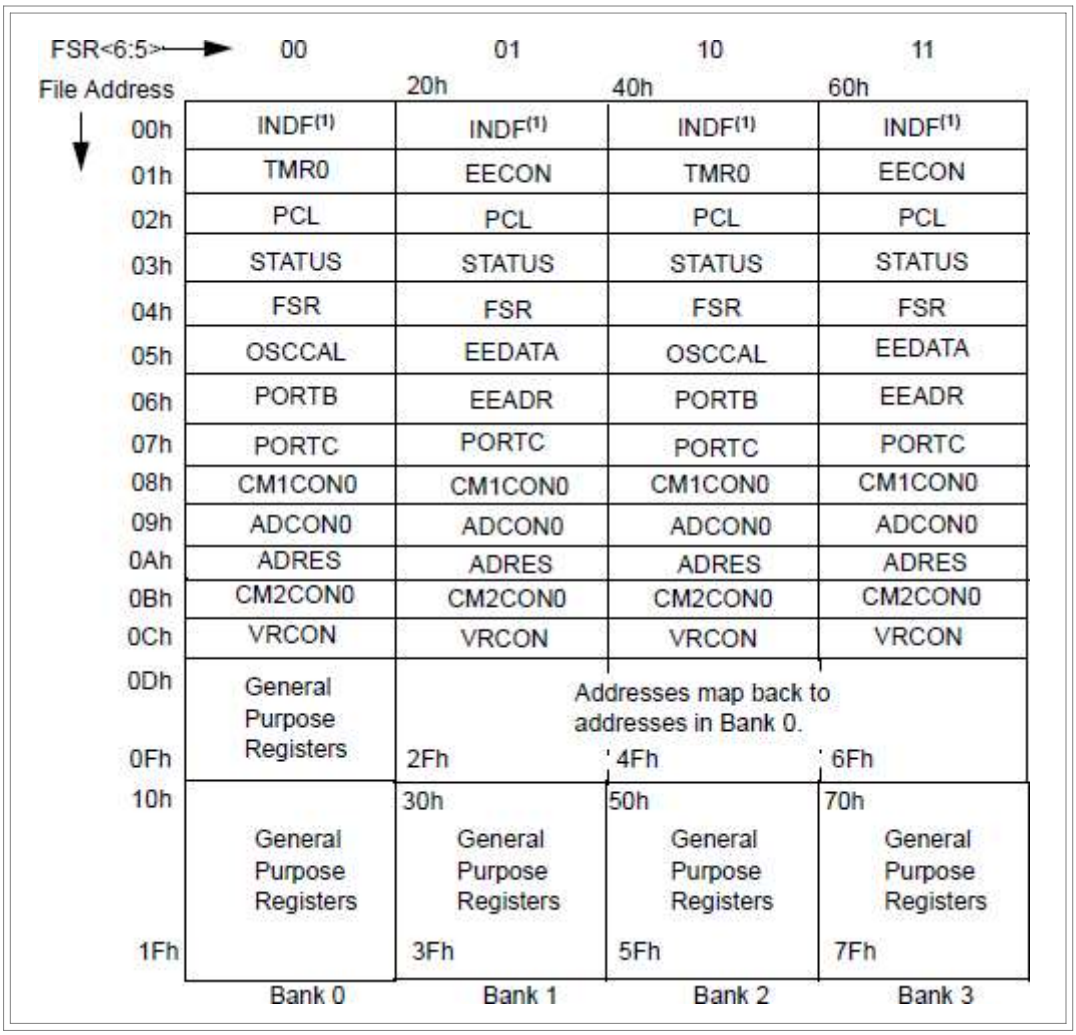
Data RAM & Banks

Let's try to further reiterate the issue of addressing on the benches, which is always a thorny topic for those approaching PICs and, together with the pagination of the program memory, the most unwelcome aspect of these microcontrollers.

We have seen, speaking of indirect addressing, that the RAM area, which contains the SFRs and the data RAM, depending on the chip, can be divided into several blocks, called banks, and access to which is regulated by the use of additional bits due to the impossibility of an opcode to contain addresses larger than the width of a bank.

If we look at the RAM map of 16F526, which has EEPROM, we see how the control SFRs have passed into the second bank, divided like the first into two sections: one part dedicated to these registers and the other to the data RAM locations.

In addition, the contents of the SFRs of banks 0 and 1 are reflected identically in banks 2 and 3 (while the data RAM area of each bank is independent).



We can also observe how some fundamental SFRs, such as **INDF**, **PCL, STATUS** and **FSR** are reproduced identically on multiple banks, with **EECON, EEDATA**, **EEADR** being altered to **TMR0**, **OSCCAL** and **PORTB respectively**.

This means that we need to pay attention to the situation of the benches, which we have not taken into account until now.
To be clearer:

- to the ROP, the default makes it possible to access what is contained in Bank0

So, with the `clrf PORTB` instruction we get the contents of this register to zero. If we are on deck 0 or 2, but if we are on deck 1 or 3 the action will be `clrf EEADR`
The designers tried to make things simpler and, for example, 16F526 banks 1 and 3 are copies of 0 and 2 as far as SFRs are concerned, and there is an area of General Pur pose Registers common to the banks (albeit only 3 bytes).
However, each chip has different characteristics and the situation may be different from this one in this example. But you don't need to know by heart the structure of the PIC you want to use to know which bits to activate in the bank change and the position of the various SFRs, since this is solved simply by using two elements already seen:

- Labels instead of absolutes
- the pseudo opcodes of the MPASM Assembler

The first action makes it useless to know where an element is mapped, since the compiler will replace the labels with the appropriate values.
The second action **makes it useless to know the bank switching mechanism and also to write macros for this function**, since `banksel` exists. So, simply:

```
; access to the EEPROM data register
   banksel  EEDATA
```

Obviously, it should be remembered that, while being in a desk, if it is still possible to access the "general use" registers, for others access is only possible from one's own desk.
Thus:

```
; access to the EEPROM
banksel EEDATA data register
   movf     EEDATA,w    ; copy given in w
   movwf    PORTB       ; this does not copy EEDATA to PORTB, but EEADR
                        to
```

To act on `PORTB` I have to go back to its exclusive desk:

```
; copy EEDATA in PORTB
   Banksel  EEDATA
   movf     EEDATA,w    ; copy given in w
   Banksel  PORTB
   movwf    PORTB
```

Also, as we saw at the beginning, we can use indirect addressing:

```
; copy EEDATA in PORTB
   movlw    EEDATA      ; indirect access to EEDATA
```

```
movwf     FSR
movf      INDF,w      ; copy given in w
banksel   PORTB
movwf     PORTB
```

Having analyzed the essence of the issue of desks, let's see what is still needed to write and read the EEPROM.

Reading and Writing the EEPROM

The **EEDATA register** is the one that is used to exchange data between the data bus and the EEPROM; in short, it is the one in which we have to put the byte to be written and from which we retrieve the one to be read (analogous to INDF).

The purpose of the **EEADR** register is to tell the EEPROM which address we want to access, i.e. where we want to write or read (similar to FSR).

In summary:

- **By writing an address to EEADR, the contents of the corresponding EEPROM location become accessible in the EEDATA register**, in write and read

From the above, it is also evident that only one byte of EEPROM can be written or read at a time.

The third register, **EECON,** is used to control operations on the EEPROM.

U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	—	FREE	WRERR	WREN	WR	RD
bit 7							bit 0

Only 4:0 bits are implemented.

```
bit 4   UNCONSTRAINED   1   Enable Clearing a Row
                        0   Enable Writing

bit 3   WRERR           1   Write or erase not performed (due to a
                            reset)
                        0   Writing Done Correctly

bit 2   WREN            1   Enable Writing
                        0   Disable Write

bit 1   WR              1   Start Writing
                        0   Write cycle completed

bit 0   RD              1   Enable Reading
```

```
0  Disable Read
```

The procedure for **reading**:

1. the address register of the location you want to reach is placed in the EEADR address register
2. you set in the EECON control log the operation to be done
3. and retrieve the value contained in the requested location in the EEDATA data log

In the instructions:

```
; read given in EEPROM - address in W
      banksel  EEDATA       ; select bank 1
      movwf    EEADR        ; Put Address in the Registry
      Bsf      EECON,RD     ; Enable Reading
      movf     EEDATA,w     ; returns with data in W
```

The reading, therefore, is "immediate" and only requires the indirect mechanism.
Writing, on the other hand, is more complex for two reasons: the first concerns the writing time of the Flash cells, which is slower than the instruction cycle and which is independent of it. The second concerns, as mentioned, the need to protect the data saved in EEPROM from unwanted access.

First, you need to delete the cells you want to write in. It is, however, possible to delete the EEPROM, but not in a single byte, but one whole line at a time:

```
; delete data in EEPROM - EE_ADR address banksel
      EEDATA                ; select EEPROM register bank
      movwf    EEADR        ; Address in the line to be deleted
      bsf      EECON,FREE ; Select Erase Mode
      bsf      EECON,WREN   ; Skill Writing
      bsf      EECON,WR     ; Start Erase
```

Bits 5:3 of the address passed to **EEADR** identify the line to be deleted.

It is important to note that, if the **FREE** bit can be set by any suitable opcode, **bsf** is the only one that allows the deletion to start.
This opcode is the "access key" to write the EEPROM.
In addition:
1. If the **WREN** bit is not set immediately after **FREE,** it is automatically deleted
2. if **WR** is not set immediately after **WREN,** it is automatically deleted.

Thus**, a deletion is only possible if the program executes the indicated sequence**, which cannot be changed from the example provided.
Since the deletion takes place in rows of 8 bytes each, if all the bytes of the line contain useful data and only a part of it needs to be modified, it is necessary to save the contents of the line, temporarily copying it to data RAM; You will edit the necessary bytes there and rewrite the entire line:

```
; copy EEPROM line indicated in W and clear contents
      banksel  EEDATA       ; select EEPROM register bank
```

```
      movwf     EEADR         ; Put Address in the Registry
RDLP BSF        EECON,RD      ; Enable Reading
; copy to RAM in bank 1
   movf      EEADATA,w    ;
      movwf     INDF          ;
      incf      FSR,f         ;
      incf      EEADR,f       ;
      btfss  FSR,3
   goto  rdlp
; cancella riga in EEPROM
      decf      EEADR,f       ; Enter the address in the line to be deleted
      Bsf       EECON,FREE    ; Select Erase Mode
      Bsf       EECON,WREN    ; Skill Writing
      Bsf       EECON,WR      ; Start Erase
```

In the deleted location it is possible **to write** a new data in this way:

1. the address register of the location you want to reach is placed in the address register
2. The value to be written is placed in the data register
3. You set the operation to be done in the audit log
4. you use a "key" to enable writing, a key that defends the contents of the EEPROM from deletion or unwanted writes
5. and we are waiting for the completion of the operation, which takes a certain amount of time compared to that typical of an education cycle.

In Instructions:

```
; write data in EEPROM - data in W, address EE_ADR
      banksel   EEADR         ; select bank 1
      movlw     EE_ADR        ; Upload Address
      movwf     EEADR
      movlw     EE_DATA       ; Charge given
      movwf     EEDATA
      Bsf       EECON,WREN    ; Skill Writing
      Bsf       EECON,WR      ; Start Writing
```

The "key" to writing security, in Baselines it is still made up of the opcode **Bsf**: Only with this can write be activated. Also, as above:
- If **WR** is not set immediately after **WREN, WREN** is automatically deleted.

This **means that writing is only possible if the program executes the specified sequence**. These "security systems" are implemented precisely in order to avoid false writes of the content of the EEPROM in case the program is out of control due to a bug or a disturbance.

If you need to rewrite the entire line, you can use a suitable routine:

```
bufstart = 0x70             ; buffer in bank 3
; copy 8 bytes from the buffer in RAM to one line EEPROM
movlw bufstart ; Buffer Start Address
```

```
        movwf     FSR           ; in the indirect pointer
        movlw     EE_ADR        ; Address at the beginning of the
                                line
        movwf     EEADR
wrdlp   movf      INDF,w        ; retrieve data from buffer in
                                RAM
       movwf      EEDATA         ; write it in the EEPROM
                                 registry
        Bsf       EECON,WREN    ; Skill Writing
        Bsf       EECON,WR      ; Start Writing
        incf      FSR,f         ; Increment RAM Pointer
        incf      EEADR,f       ; Increment EEPROM Address
        BTFSS     FSR,3         ; 8 steps taken?
         Goto     wrdlp         ; No - another loop
; fine scrittura riga
        banksel   0
```

The RAM buffer has been placed in bank 3, i.e. in the area from 70h to 7FH. EEPROM control registers are available in both bank 1 and bank 3, so there is no need to move the bank switches, as FSR writing led to bank selection3. At the end of the operation, bank 0 is reactivated.

The EEPROM area has the same data retention guaranteed for the flash program and can be written and deleted, but it is advisable to check its content after writing:

```
; verification of data in EEPROM - data in W, address
        movf    EEDATA, W
        bsf     EECON, RD     ; Enable Reading
        xorwf   EEDATA, W     ; Compare Data
        skpz                  ; if equal, skip
         goto   WREE_ERR      ; if not, handle the error
```

As mentioned before:

 **The average write or erase time is 3.5ms**, with a maximum of 5ms (param. 43 and 44), **regardless of the chip clock**.
Once the write (or delete) starts, the Program Counter does not advance to the next instruction until this time is up.

For reading, on the other hand, there is no waiting time.

If the write or erase operation is interrupted by a reset (due to MCLR or WDT), the status of the **EECON WRERR** bit goes to level 1, indicating that the operation was unsuccessful. If the reset is caused by the power failure and subsequent restart (POR), the flag is invalid. If there is a possibility of these events, it is necessary to manage the causes of reset at startup, as seen in Exercise 14, which also takes into account the EEPROM.

Lines of the EEPROM?

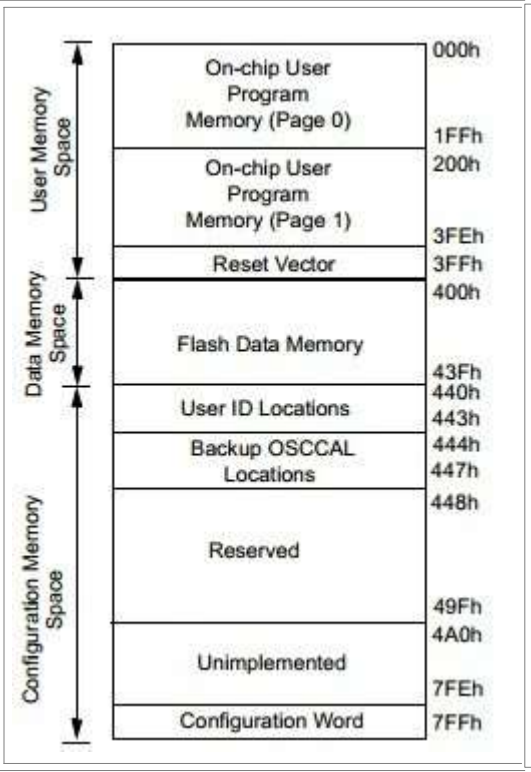
We've casually talked about deleting the lined EEPROM. Let's take a closer look at what this means.

The EEPROM is organized into 8 rows of 8 bytes each (8 x 8 = 64 bytes total).
In the address, bits2:0 identify the position of the cell in the row, while bit5:3 indicate the line. 7:6 bits are not implemented. For example,:

EEADR bits	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	
Function	***not impl.***		***000 to 111***			***00 to 07***			
Example	-	-	0	0	0	0	0	0	First location of row 0
	-	-	0	0	1	0	0	0	First location of row 1
	-	-	0	1	1	0	0	1	Second location of row 3
	-	-	1	1	1	x	x	x	row 7
	-	-	1	1	1	1	1	1	Last location of row 7

So, if you want to write the 64th location, the address to be passed to EEADR will be 3F.
If you want to write the first location, you will have to pass 00.
If you want to delete the line that contains location 60, you will need to delete the entire line 7. If you want to delete the first location of row 1, you will delete all eight locations in the row.

Note that these are the **relative addresses** of the EEPROM area. The absolute position in memory is identifiable in the chip's memory map, which can be found from the datasheet.



For example, for 16F526, the EEPROM (*Flash Data Memory*) is between 400h and 43Fh.

This, however, does not interest us during the use of it, since we cannot access it directly, but only through the mechanism of registers seen above.

So we have to consider the EEPROM area as a block of memory ranging from 00 to 3Fh, for a total of 64 elements, organized as we have just outlined.

We have a symbolic relationship for the absolute addresses of the EEPROM area in the ***processorname.lkr*** file. For example, for 16F526, we find:

```
CODEPAGE NAME=flashdata START=0x400
         END=0x43F
```

Knowing where the EEPROM is placed as an absolute address can only be useful if we want to preload it at the time of programming, using the DE directive.

DE

The **De** (*Declare Eeprom data bytes*) directive allows data to be loaded into EEPROM when the chip is programmed:

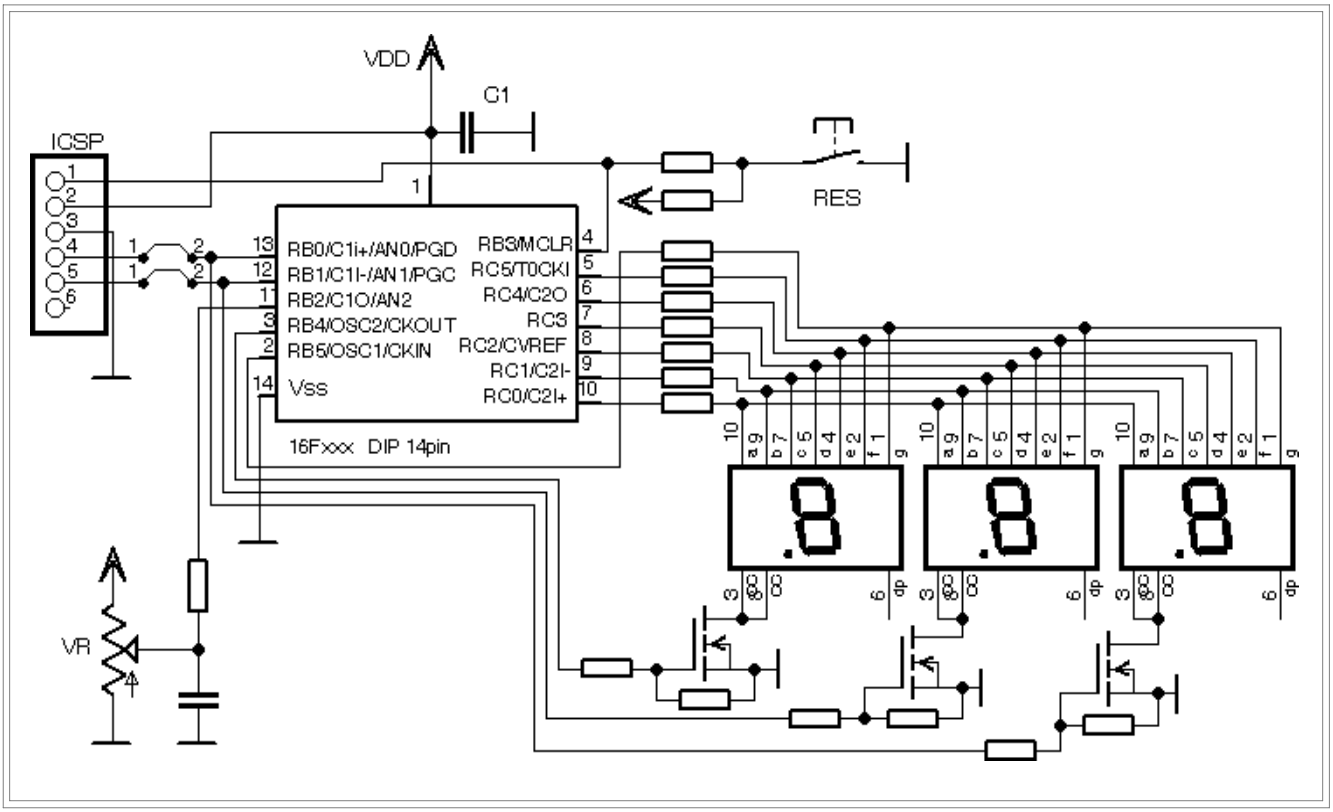
[label]	sp	de	sp	expr, expr,...

The directive reserves 8 bits for each expression in question. Each expression must yield an 8-bit value.
It must be preceded by the memory position indicator:

```
ORG  0x400
de   "www.microcontroller.it",0xAA,0x55,0
```

One application.

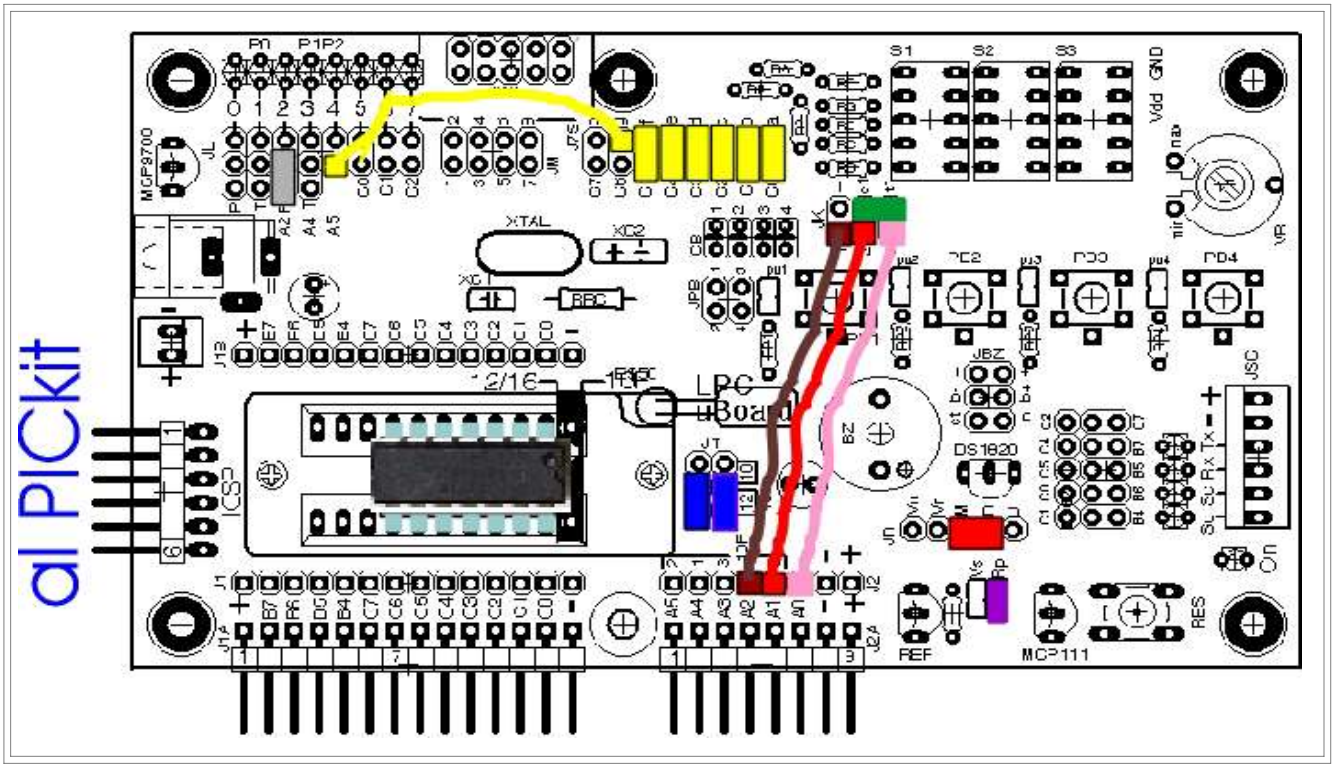
We use indirect addressing and EEPROM to store data that is entered by the user.



When first switched on, three displays show 3 digits 0; this value was entered during the compilation of the source and written in the EEPROM when programming the chip. Pressing the button activates the AD converter which reads the voltage on the potentiometer slider. The value resulting from the conversion, in decimal, is shown on the display and

saved in EEPROM. The next time you switch it on, this value will be presented on the displays. By turning the potentiometer to a different position before pressing the button, we will vary the converted value between 000 8min.) and 255 (max.).

On the **LPCuB** the circuit is made with the usual jumpers:



"Yellow" jumpers connect segments
The "pink", "red" and "brown" flying jumpers connect the gates of the MOSFETs. The "gray" jumper connects the potentiometer slider.

During ICSP programming, the "red" and "pink" jumpers must be disconnected, as the MOSFET load can make programming difficult.

The program.

Let's use much of what we've already written for previous tutorials.

First of all, it will be necessary to set the time generator for the cadence of the multiplex, which we make as already seen, using Timer0.

At startup, the program reads the contents of the EEPROM and copies it to RAM:

```
; copy EEPROM, line 0, to the buffer in
          RAM pagesel RdEepRow
          call    RdEepRow
```

The entire first line of 8 bytes is read, although only the first location will be changed. The buffer is set in the RAM in bank3; this allows you to perform operations on the EEPROM registers and on the buffer without changing banks: if we look at the RAM memory map of the 16F526, we notice that the command registers of the Flash data are accessible in banks 1 and 3 and in them 16bytes of data RAM are available:

```
; copy riga 0 of the EEPROM to RAM
RdEepRow
     movlw rambuf    ; Indirect pointer to start
     movwf FSR       ; the buffer in RAM and the bank for the EEPROM
     movlw eeprow0 ; select first byte of line 0
     movwf EEADR     ;
rdlp bsf   EECON,RD ; Enable Reading
; copia in RAM
     movf  EEDATA,w ; read EEPROM
     movwf INDF      ; indirect copy to buffer
     incf  FSR,f     ; increment pointer of buffer
     incf  EEADR,f   ; Increment EEPROM Address
     btfss FSR,3     ; 8 bytes?
      goto rdlp      ; No - yet another 0
     retlw           ; yes - end
```

We use indirect addressing to point the buffer to RAM; since this is located in bank3, at the same time you also have the right access to the EEPROM registers.
The data is copied from **EEDATA** to the buffer, incrementing both the indirect pointer (**FSR**) and the EEPROM address (in **EEADR**) at each loop. By checking the status of bit 3 we stop the operation after the transfer of 8 bytes.

The data in location 0 of the EEPROM is converted from 3-byte hexadecimal where the MSB is at 0 and the LSB corresponds to the decimal value; it is an uncompressed BCD. This routine has already been covered in the math exercise.
For example, if the byte is **AAh**, which is 170 decimal, converting to BCD produces this result:

bin = unit = 00 bcdL= tens = 07 bcdH = hundreds = 01

These values are passed one at a time through the usual lookup table to command the display segments:

```
; segment data table - display catodo comune
segtbl ;andlw 0x0F  ; solo nibble basso
       addwf PCL,f        ; punta PC
       retlw b'00111111' ; "0" -|-|F|E|D|C|B|A
       retlw b'00000110' ; "1" -|-|-|-|-|C|B|-
       retlw b'01011011' ; "2" -|G|-|E|D|-|B|A
       retlw b'01001111' ; "3" -|G|-|-|D|C|B|A
```

```
        retlw b'01100110' ; "4" -|G|F|-|-|C|B|-
        retlw b'01101101' ; "5" -|G|F|-|D|C|-|A
        retlw b'01111101' ; "6" -|G|F|E|D|C|-|A
        retlw b'00000111' ; "7" -|-|-|-|-|C|B|A
        retlw b'01111111' ; "8" -|G|F|E|D|C|B|A
        retlw b'01101111' ; "9" -|G|F|-|D|C|B|A
        ;retlw b'01110111' ; "A" -|G|F|E|-|C|B|A
        ;retlw b'01111100' ; "b" -|G|F|E|D|C|-|-
        ;retlw b'00111001' ; "C" -|-|F|E|D|-|-|A
        ;retlw b'01011110' ; "d" -|G|-|E|D|C|B|-
        ;retlw b'01111001' ; "E" -|G|F|E|D|-|-|A
        ;retlw b'01110001' ; "F" -|G|F|E|-|-|-|A
```

Notice that the table is reduced to only values between 0 and 9, since we only deal with BCD digits; you don't even need the initial and since the values obtained from the HEX->BCD conversion already have the 4 bits high zeroed.

At the end of the three-digit scan, we enter the button verification:

```
        btfss btn     ; button open?
         goto btnchk ; No - Closed
; Yes - Open Button - Check Previous Status
btfss btnflg ; prima era aperto?
       bsf   btnflg ; no -refresh flag
; prima era chiuso
       goto  displp ; done

; Button Closed - Check Previous Status
btnchk btfss btnflg ; Was it open before?
        goto displp ; no - done
; prima era aperto
       bcf   btnflg ; Update Flag
```

When the button is pressed, the previous situation is compared and the next step is only moved on if there has been a transition from open to closed. Otherwise, it returns to the display loop. The multiplex scan time is used as the debounce time for the button.

If this is pressed, the AD conversion starts and the resulting data in ADRES is written to EEPROM, using the indirect pointer, which does not require an explicit bank switch:

```
; button pressed - start AD conversion
     pagesel ADConv
     call    ADConv
     movlw   rambuf      ; Save Result
     movwf   FSR
     movf    ADRES,w
     movwf   INDF
```

Immediately afterwards, line 0 of the EEPROM is deleted and the buffer is rewritten from RAM.
The loop closes on the conversion of the hexadecimal data for the display.

Then, by pressing the button, the voltage value on the potentiometer cursor will become digital data and will be saved in EEPROM, as well as immediately displayed.

When the circuit is switched on for the first time, the display will present the pre-programmed value in the Data Flash through the statement **de** , i.e. 0:

```
EEPROMdata CODE 0x400
 de 0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07
```

After pressing the button, the saved value will be the one resulting from the AD conversion; By turning the circuit off and on, this will be the new value represented on the displays.

The use of **banksels** is limited by indirect addressing, by the choice of the RAM buffer on the same page as the EEPROM access registers and by the declaration of some variables in the area of RAM shared between the banks:

```
SHR1 UDATA_SHR ; RAM shared - 3 bytes
; Required for Conversion bin
res 1          ; Rail & Unit
bcdL res 1     ; Dozens
bcdH res 1     ; hundreds
```

Unfortunately, it's only 3 bytes; if more RAM is needed, it is declared as generic and overlaps these first bytes, but, since it is used at different times in the program, there is no conflict.
We insert **banksel** only where necessary, for example:

```
; converts in BCD
convert movlw   rambuf
        movwf   FSR
        movf    INDF,w
        pagesel Hex1Bcd3
        call    Hex1Bcd3
        pagesel $
        banksel 0
        clrf    TMR0       ; Initialize Timer
```

since the bank is set at 3 by the **FSR** and the **TMR0** register is only available on bank 0 or 2.

One of these can be easily modified to collect more data, within the limits of the capacity of the Flash data, and thus realize a data logger function. If more data needs to be stored, an external EEPROM will have to be used.

Compilation messages

Going to check the message of the Compiler Output window, we find listed numerous messages like this:

```
Message[302]: Register in operand not in bank 0. Ensure that bank bits are correct.
```

These messages are related to the fact that we are using SFR logs that are not accessible on page 0: the Assembler generates these messages to warn the user to be careful.
We find them in ***the .lst*** file in relation to the lines where these logs appear:

```
Message[302]: Register in operand not in bank 0.  Ensure that bank bits are correct.
002C   0026           00224       movwf     EEADR       ;
Message[302]: Register in operand not in bank 0.  Ensure that bank bits are correct.
002D   0501           00225 rdlp  bsf       EECON,RD    ; attiva lettura
                      00226 ; copia in RAM
Message[302]: Register in operand not in bank 0.  Ensure that bank bits are correct.
002E   0205           00227       movf      EEDATA,w    ; read EEPROM
002F   0020           00228       movwf     INDF        ; Indirect Copy to Buffer
0030   02A4           00229       incf      FSR,f       ; Increment Buffer Pointer
Message[302]: Register in operand not in bank 0.  Ensure that bank bits are correct.
0031   02A6           00230       incf      EEADR,f     ; incrementa indirizzo EEPROM
```

The **`Message`**, as we've said in other tutorials, don't prevent the build from finishing and generating the file ***.Hex***but **They serve to draw the programmer's attention to points that could render the executable file unusable.**

As mentioned, in the first compilation, it is definitely not advisable to abolish this report. Only and exclusively after checking the correctness of what is written and being sure of the correct management of the desks, we can, if we want, add the infamous line at the top of the source:

```
   errorlevel -302
```

This deletes the [302] **messages** from the list.

Avoid inserting this line until you fully understand the mechanism of the banks, and in any case never during the first compilations of the source.

Only after eliminating any other errors and verifying the position of registers and banks can the report be abolished, for the sole practical purpose of having a "clean" listed file.

In the source of the example, the line is already there, but as a comment. To activate it, just uncomment it. We recommend that you do the following:

- Fill in the source with the commented line and verify both the message in the ***Out***, both the ***.lst file***
- uncomment the line and repeat the compilation, checking the ***Out window*** and the listed file as before

A comparison of the two readings will make clear the action of the line **errorlevel -**

15A_526.asm

```
;*****************************************************************
; 15A_526.asm
;-----------------------------------------------------------------
;
;    Title          : Assembly & C Course - Tutorial 15A
;                     Three-digit hex ADC result and
;                     saving to EEPROM
;    PIC            : 16F526
;    Support        : MPASM
;    Version        : V.519-1.0
;    Date           : 01-05-2013
;    Hardware ref.  :
;    Author         :Afg
;
;-----------------------------------------------------------------
;
; Pin use :
;   ________________
;       16F506/526 @ 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 segm g
;    RB4/OSC2/CLKOUT          3: Out gate hundreds
;    RB3/! MCLR/VPP           4: In   button
;    RC5/T0CKI                5: Out segm f
;    RC4/C2OUT                6: Out segm and
;    RC3                      7: Out segm d
;    RC2/CVref                8: Out segm c
;    RC1/C2IN-                9: Out segm b
;    RC0/C2IN+               10: Out segm a
;    RB2/C1OUT/AN2           11: AN2 potentiometer
;    RB1/C1IN-/AN1/ICSPC     12: Out gate tens
;    RB0/C1IN+/AN0/ICSPD     13: Out gate unit
;    Vss                     14: --
;
; ###############################################################
; Choice of processor
        LIST      p=16F526
        #include <p16F526.inc>

       radix     DEC

       ; errorlevel -302
```

```
; ######################################################################
;                           CONFIGURATION
; Internal Oscillator, 4MHz, No WDT, No CP, MCLR
 __config _IntRC_OSC_RB4 & _IOSCFS_4MHz & _WDTE_OFF & _CP_OFF &
_CPDF_OFF & _MCLRE_OFF

; ######################################################################
;                            RAM
; general purpose RAM
SHR1 UDATA_SHR ; Shared RAM - 3 bytes
; Required for Conversion
Bin    RES 1                ; unit
bcdL   RES 1                ; Dozens
bcdH   RES 1                ; hundreds

GRAM      UDATA
Temp   RES 1                ; temporary
flags res 1                 ; Flags

#define BTNFLG Flags,7      ; Button Flag Pressed

; pre load line0 in EEPROM
EEPROMdata    CODE 0x400
 de 0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07

eeprow0 equ 0               ; first EEPROM lease

 ;*********************************************************************
;=====================================================================
;            DEFINITION OF PORT USE
;
;P ORTC map
;| 5 | 4 | 3 | 2 | 1 | 0 |
;|-----|-----|-----|-----|-----|-----|
;|segmf|segme|segmd|segmc|segmb|segma|
;
#define     Segma  PORTC,0    ; F:A Segments
#define     segmb  PORTC,1    ;
#define     SEGMC  PORTC,2    ;
#define     SEGMD  PORTC,3    ;
#define     Segme  PORTC,4    ;
#define     SEGMF  PORTC,5    ;

;P ORTB
map
;| 5 | 4 | 3 | 2 | 1 | 0 |
;|-----|-----|-----|-----|-----|-----|
;|segmg|gate3| byn | AN2 |gate1|gate2|
;
#define     gate2  PORTB,0    ; Gate MOSFET Digit unit
#define     gate1  PORTB,1    ; Gate MOSFET Digit Dozens
;#define           PORTB,2    ; AN2
#define     Btn    PORTB,3    ; RES button
#define     gate3  PORTB,4    ; Gate MOSFET Digit hundreds
#define     segmg  PORTB,5    ; Segment G
;
; ######################################################################
;                          CONSTANTS
```

```
;
#define bit05k TMR0,1 ; TMR0 bits for 512us
#define bit1k    TMR0,2 ;             1024us
#define bit2k    TMR0,3 ;             2048us
#define bit4k    TMR0.4 ;             4096us
#define bit8k    TMR0.5 ;             8192US

#define Bittmr    bit4k ; 4096us

rambuf = 0x70           ; RAM Buffer in Bank 3

; ######################################################################
;                          LOCAL MACROS
; UNITon MOSFET gate
command  MACRO
        bsf gate2
          ENDM
UNIToff MACRO
        bcf gate2
          ENDM
DECon   MACRO
        bsf gate1
          ENDM
DECoff MACRO
        bcf gate1
          ENDM
CENTon   MACRO
        bsf gate3
          ENDM
CENToff MACRO
        bcf gate3
          ENDM

; Includes basic macro set
 #include C:\PIC\Library\Baseline\basemacro.asm

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

Main:
; MOWF Internal Oscillator
        Calibration OSCCAL
        pagesel init
        goto    Init

; ######################################################################
;             TABLES AND SUBROUTINES ON PAGE 0

; Segment Data Table - Display Common Cathode
SEGTBL ANDLW 0x0F ; Low nibble only
        addwf PCL,f ; PC tip
        retlw  b'00111111'  ;  "0"  -|-|F|E|D|C|B|A
        retlw  b'00000110'  ;  "1"  -|-|-|-|-|C|B|-
        retlw b'01011011' ; "2" -|G|-|E|D|-|B|At
```

```
        retlw  b'01001111'  ;  "3"  -|G|-|-|D|C|B|A
        retlw  b'01100110'  ;  "4"  -|G|F|-|-|C|B|-
        retlw  b'01101101'  ;  "5"  -|G|F|-|D|C|-|A
        retlw  b'01111101'  ;  "6"  -|G|F|E|D|C|-|A
        retlw  b'00000111'  ;  "7"  -|-|-|-|-|C|B|A
        retlw  b'01111111'  ;  "8"  -|G|F|E|D|C|B|A
        retlw b'01101111' ; "9" -|G|F|-|D|C|B|At
        ; retlw b'01110111'   ; "A" -|G|F|E|-|C|B|At
        ; retlw b'01111100'   ; "b" -|G|F|E|D|C|-|-
        ; retlw b'00111001'   ; "C" -|-|F|E|D|-|-|At
        ; retlw b'01011110'   ; "d" -|G|-|E|D|C|B|-
        ; retlw b'01111001'   ; "E" -|G|F|E|D|-|-|At
        ; retlw b'01110001'   ; "F" -|G|F|E|-|-|-|At

;--------------
; ADConv Voltage
Measurement:
; waiting
        DLY10US
; Start Conversion
        Bsf     ADCON0,GO
ADLP    BTFSC   ADCON0,NOT_DONE
         goto   ADLP
        retlw   0

;--------------
; Hex -> BCD unpacked conversion
; Hex number input in W
; Output on bcdH (hundreds), bcdL (tens), and bins (units)
; with MSB at
0 Hex1Bcd3
        movwf   Bin
        CLRF    bcdL
        CLRF    bcdH
HCNT    movlw   .100
        subwf   bin,W
        skpc
         Goto   HDEC
        incf    bcdH,F
        movwf   Bin
        Goto    HCNT
HDEC    movlw   .10
        subwf   bin,W
        skpc
         retlw  0           ; return
        movwf   Bin
        incf    bcdL,F
        Goto    HDEC

;--------------
; copy line0 from EEPROM to
RAM RdEepRow
      movlw    Rambuf      ; Indirect Pointer for MovWF
      Start    FSR         ; of the buffer in RAM
      ; banksel EEDATA       ; select register bank EEPROM movlw
               eeprow0     ; Select First Byte of Line 0 MovWF
               EEADR       ;
```

```
RDLP BSF       EECON,RD    ; Enable Reading
; Copy to RAM
      movf     EEDATA,w    ; read EEPROM
      movwf    INDF        ; Indirect Copy to Buffer
      incf     FSR,f       ; Increment Buffer Pointer
      incf     EEADR,f     ; Increment EEPROM Address
      BTFSS    FSR,3       ; 8 bytes?
       Goto    RDLP        ; No - yet another one
      retlw    0           ; Yes - End

;--------------
; delete row0 in EEPROM
ClEepRow0
      Banksel  EEDATA
      movlw    eeprow0     ; Select Row 0
      movwf    EEADR
      Bsf      EECON,FREE  ; Cancellation mode
      Bsf      EECON,WREN  ; Skill Writing
      Bsf      EECON,WR    ; Start Erase
      retlw    0           ; end

;--------------
; copy 8 bytes from the buffer in RAM to one line
EEPROM WrEeprow
      movlw    Rambuf      ; movwf buffer start
      address  FSR         ; in the movlw indirect
      pointer  eeprow0     ; Movwf Line Start
      Address  EEADR
wrdlp movf     INDF,w      ; retrieve data from buffer in
      movwf RAM            EEDATA   ; write it in the
      EEPROM bsf registry  EECON,WREN ; Skill Writing
      Bsf      EECON,WR    ; Start Writing
      incf     FSR,f       ; Increment incf RAM pointer
               EEADR,f     ; increment btfss EEPROM
      address  FSR,3       ; 8 steps taken?
       Goto    wrdlp       ; No - another retlw
      loop     0           ; End of Line Writing



; ###################################################################
;                        MAIN PROGRAM
Init:
; Disable comparators to free the BCF digital function
               CM1CON0, C1ON
        Bcf    CM2CON0, C2ON

; disable T0CKI from RB5, prescaler 1:256 to Timer0
         ;     b'11111111'
         ;       1-------    GPWU Enabled
         ;       -1------    GPPU Enabled
         ;       --0-----    Internal Clock
         ;       ---1----    Falling
         ;       ----0---    prescaler to Timer0
         ;       -----111    1:256
        movlw  b'11010111'
        OPTION
```

```
        CLRF    PORTB      ; Clear Port Latch
        CLRF    PORTC
        Bsf     BTNFLG     ; Preset Flags

; All useful ports come out
        movlw   0
        Tris    PORTB
        Tris    PORTC

; configure ADC for AN2, INTOSC/4 and
        enable movlw     b'01111011'
        movwf    ADCON0

;-----------------------------------------------------------------
; copy EEPROM, line 0, to the buffer in RAM
        pagesel RdEepRow
        Call     RdEepRow

; converts to BCD
convert movlw    Rambuf
        movwf    FSR
        movf     INDF,w
        Pagesel  Hex1Bcd3
        Call     Hex1Bcd3
        Pagesel  $
        Banksel  0
        CLRF     TMR0       ; Initialize Timer

;-----------------------------------------------------------------
; Display Cycle
; Digit Unit
displp btfss    bittmr      ; 4ms ?
         Goto   displp      ; No - Waiting
        movf    bin,w       ; Yes - Load Call
        Unit    segtbl      ; Lookup Table
        Banksel Temp
        movwf   Temp        ; Save to Temporary Movwf
                PORTC       ; Write on the port segm
        f:a btfsc           temp,6    ; Command G-
        Segment
         Bsf    segmg
        BTFSS   temp,6
         Bcf    segmg
        UNITon              ; Unit digit lit
dslp_1 BTFSC    bittmr      ; 4ms ?
         Goto   dslp_1      ; no - waiting
dslp_2 UNIToff              ; Yes - Turn off
units

; digit tens
        movf    bcdL,w      ; Load dozens
        of calls            segtbl
        movwf   Temp
        movwf   PORTC
        BTFSC   temp,6
         Bsf     segmg
        BTFSS   temp,6
```

Bcf segmg

```
        DECon               ; Turn on dozens
dslp_3  BTFSS   bittmr      ; 4ms ?
         Goto   dslp_3      ; No - Waiting
        DECoff              ; Yes - Turn off
                              tens

; digit hundreds
        movf    bcdH,w
        Call    segtbl
        movwf   Temp
        movwf   PORTC
        BTFSC   temp,6
         Bsf     segmg
        BTFSS   temp,6
         Bcf     segmg
        CENTon              ; Turn on hundreds
dslp_4 BTFSC    bittmr      ; 4ms ?
         Goto   dslp_4      ; No - Waiting
        CENToff             ; Yes - Turn off hundreds

        BTFSS   Btn         ; button open?
         Goto   btnchk      ; No - Closed
; Yes - Open Button - Check Previous Status BTFSS
                BTNFLG      ; Was it open before?
         Bsf    BTNFLG      ; No - Update Flag
; It used to be closed
        Goto    displp      ; done

; Button Closed - Check Previous Status BTNCHK
BTFSS           BTNFLG      ; Was it open
before?
         Goto   displp      ; no - done
; It used to be open
        Bcf     BTNFLG      ; Update Flag

; button pressed - start AD pageel
        conversion ADConv
        Call    ADConv
        movlw   Rambuf      ; Save Result
        movwf   FSR
        movf    ADRES,w
        movwf   INDF

; delete line 0 in EEPROM
        pagesel ClEepRow0
        call    ClEepRow0
; write ram buffer in EEPROM
        pagesel WrEeprow call
                WrEeprow

        Goto    Convert     ; Loop

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