Knowledge of baseline

A - Mathematics with Baselines

The purpose of the tutorial is to learn something about the mathematical possibilities of PIC microcontrollers.

In fact, it often happens that it is necessary to make calculations on data that must be processed not only logically, but also mathematically, both for the decisions to be made, and for presentation on the display or sending to peripherals or host computers.

Over time, Microchip has produced various application notes in this regard:

- *AN526 "PIC16C5x/PIC16Cxxx Math Utility Routines"*
- *AN617 "Fixed Point routines"*

and which can be downloaded from the **Microchip** website. Referring to these documents for further information, let's try to have some ideas on what is possible.

Arithmetic : 8-bit opcodes

Baselines, as we have seen, are the most basic family of PICs and this is also reflected in the instruction set that includes only two arithmetic operations:

- the **sum** of the W register and a : `addwf file`
- the **subtraction** between the W register and a file : `subwf`

Since these are processors with an 8-bit data bus, these operations also take place on this amplitude, where the maximum value is 255 (FFh). There are no multiplication or division operators; To perform calculations on larger numbers or other mathematical operations, appropriate algorithms must be implemented. It's all about using the instructions from the set and a sufficient number of RAM registers.

For this purpose, the STATUS register intervenes, which with its flags allows you to overcome the 8-bit limitation.
In the case of sum, the Carry flag and the Zero flag of the STATUS are modified like this:

- if the result of the sum does not exceed 255, the Carry remains at 0
- if the result exceeds 255, the Carry goes to 1, thus containing the carryover

and

- if the sum result is not 0, the Zero flag is 0
- if the sum result is zero, the flag goes to 1

Essentially:

- Carry indicates that the sum has a carryover
- Zero indicates that the result in the byte is 0 Let's

see in detail how an 8-bit sum works:

```
; Somma (AARG + BARG) -> BARG a 8 bit
; The Carry the possible carryover
   Movf AARG,w      ; A in W
   addwf   BARG,f    ; sum A+B in B  (B=B+A)
; if there is an carryover, the Carry going to   1
```

Thus, adding 33h and 22h, the result is 55h, which is less than FFh and therefore entirely containable in the byte.
If, on the other hand, we add 33h and CEh, the result is 101h, which cannot be represented in 8 bits, but requires 9 bits.
As a result, the least significant 8 bits (01h) will be contained in the result register, while the Carry, going to 1, will indicate that the sum has exceeded FFh. Thus, the result will be presented like this:

Carr y	register
1	01

The Carry, indicating the carryover of the sum, becomes a key element for carrying out operations that use more than 8 bits.

For those who still have doubts about the result, let's say that the Carry bit is absolutely adequate to contain the carryover, since the sum of two 8-bit numbers at most results in a 9-bit number. In fact:

FFh +FFh = 1FEh

Then, summing up:

```
; 8-bit sum (AARG + BARG).
; The result is saved in BARG
; The Carry carries any carryover
    movlw   .255  ; W = FFh
   movf    AARG      ; A = W = FFh
   movf    BARG,w    ; B = W = FFh
   movf    AARG,w    ; W = A  = FFh
   addwf   BARG,f    ; somma a B in B -> (B=B+A = FFh+FFh = 1FEh)
; the Carry goes to 1 for the carry
```

and as a result:

Carr y	register
1	FE

The same can be said of subtraction.

Be careful, however, that the **subwf opcode** performs the operation **(file-W)** and not **(W-file)** as you might think:

```
; 8-bit subtraction (BARG - AARG).
; The result is saved in BARG
; The Carry carries any carryover
   movf    AARG,w    ; A in W
   subwf   BARG,f    ; subtracts da B in B -> B=B-A
; se c'è un resto, il Carry/Borrow va a 1
```

In this case, the Carry becomes the Borrow that carries back the rest of the subtraction. As a result, if the subtraction has a remainder or the result is negative, the C and Z flags of the STATUS will indicate the situation:

Result	Carry	Zero
Positive	1	0
Zero	1	1
Negative	0	0

The use of flags allows you to extend the operation to multiple bits, as well as to handle signed numbers.

Incidentally, Baselines do not have the addition and subtraction opcodes with literal values, **addlw** and **sublw** , which are present in the other instruction sets. This requires you to use instruction sets, e.g. with a macro:

```
; ; Subtracts W from the lit value
(sublw)
; A temporary location is used
    movlw   lit      ; W= lit
    movwf   temp     ; temp = W = lit
    movlw   value    ; W = value
    subwf   temp,w   ; W = lit - W
        ENDM
```

For the sake of completeness, it should be added that the addition and subtraction operations also act on **the DC** (***Digit Carry***) bit of the **STATUS,** but this flag is used almost exclusively in 4-bit arithmetic, with numbers encoded in BCD or in binary-BCD conversions, which we will see later.

A further important note concerns the operations of addition and subtraction of the Baselines: for these PICs these are actions that do not take into account the Carry, in the sense that this flag is modified by the result of the operation, but does not become part of the operation itself. So, technically, these are Carry-free trades.
It is necessary to specify this because other families of PICs perform addition and subtraction taking into account the present Carry.
So, for Baselines:

addwf -> (**file+W**) with the Carry as the ninth bit of the

result For Midrange and above, the same opcode renders:

`addwf` -> (`file+W+C`) with the Carry as the ninth bit of the result

And so it is with subtraction, where the Carry becomes Borrow.
This feature, in Baselines, allows you to avoid having to adjust the Carry, zeroing it, when it is an 8-bit sum, but, on the other hand, it requires that it be adjusted correctly when we perform operations on multiple bytes.

16-bit arithmetic

To achieve adequate accuracy, many operations must process at least 16-bit numbers (maximum content 65535 decimal or FFFFh). Since the capacity of the data bus is 8 bits, it is not possible to perform these operations in a single action, but it is necessary to take advantage of the available instructions, creating appropriate algorithms. And, since, in any case, the registers are 8-bit, it will take more than one to preserve operands and result.

For example, if we want to deal on 16-bit numbers, we will need to use registers in pairs:

```
   UDATA     ; variables for 16-bit calculations
AL   res 1    ; 16-bit A register
AH   res 1
BL   res 1       ; 16-bit B register
BH   res 1
```

where **xL** is the least significant byte and **xH** is the largest byte (*little endian arrangement*).

In defining these elements, a problem arises, which English speakers call *endianness*, namely that of the sequence of values in bytes. In practice, we have two options:

variable	variable+1	way
High Byte	Low Byte	*Big Endian*
Low Byte	High Byte	*Little Endian*

So, if we have to store the number 1234h :

variable	variable+1	way
12	34	*Big Endian*
34	12	*Little Endian*

The *big endian* mode has the advantage that it can be read more easily in listings and during debugging, while the *little endian* mode allows better dynamics for numbers of varying sizes. To explain, if we use the *big endian mode* and the variable requires more bytes, it also requires the bytes to change position.

For example,:

variable	variable+1	variable+2 hundreds

Dozens	unit	--
12	34	--
hundreds	**Dozens**	**unit**
12	34	56

On the other hand, an increase in the *little endian* mode allows the bytes to always maintain the same relationship with their position in memory, making it easier to operate:

variable	variable+1	variable+2
unit	**Dozens**	**--**
34	12	--
unit	**Dozens**	**hundreds**
56	34	12

While both modes are valid, little *endian*, for the most dynamic possibilities, is often the preferred one, and Microchip also uses this approach.

Sum at 16 bits and above

To get more definition, it is often necessary to resort to numbers larger than those that can be contained in 8 bits. Numbers on 16 bits are then used, a form referred to as "double precision".

Based on what we have said, the algorithm for realizing a 16-bit sum is a simple consequence:

```
; Add 16-bit A and B.
; The result is saved in B
; The Carry carries any carryover
   movf    AL,w       ; LSB di a in W
   addwf   BL,f       ; sum to the LSB of b in b
; if there is a carry, the Carry goes to 1
   skpnc              ; skip next opcode if C=0
   incf   BH,f        ; if C=1 increases MSB by b
   movf    AH,w       ; MSB of a in W
   addwf   BH,f       ; sum to the MSB of b in b
; the Carry contains any carryover
```

The need to adjust the MSB according to the Carry depends on what we said above, i.e. for Baselines the Carry does not become part of the sum, but only of the result.
Suppose we add a = 0033h and b = 00FFh; We:

a = 33h a+1 = 00h
b = FFh b+1 = 00h

in *Little Endian* format. The algorithm runs like this:

```
   movf    AL,w      ; W = 33h
   addwf   BL,f      ; BL = FFh + 33h = 32h with carryover 1 to Carry
   skpnc             ; does not skip next opcode, since C=1
    incf   BH,f      ; BH = 00h + 1 = 01h
   movf    AH,w      ; W = 00
   addwf   BH,f      ; BH = 01h + 00 = 01h
; quindi:
; 0033h +00FFh = 0132h
; result in BL=32h and BH=01h (little endian) - there is no carry
```

However, when creating a mathematical algorithm, it is necessary to exhaustively verify its correctness. Let's add ***FFFFh*** and ***FFFFh***:

```
   movf    AL,w       ; W = FFh
   addwf   BL,f       ; BL = FFh + FFh = FEh with carryover 1 to Carry
   skpnc              ; it doesn't skip next opcode, since C=1
    incf   BH,f       ; BH = FFh + 1 = 00h with the carryover in C=1
; but the Carry does not become part of the following
   sum... movf      AH,w    ; W = FFh
   addwf BH,f         ; BH = 00h + FFh = FFh without carryover
; therefore:
; FFFFh + FFFh = 1FFFEh
; result in BL = FEh and BH =FFh (little endian)
; but without the carryover 1 in the Carry
```

This flawed algorithm is frequently found on the net, even in the Microchip forum itself. A different approach is therefore needed that manually adjusts the carryover:

```
    movf     AL,w   ; W = FFh
   addwf    BL,f   ; BL = FFh + FFh = FEh with carry 1 to Curry
   movf     AH,w   : Preset W=AH=FFh
   skpnc           ; Does not skip next opcode, since C=1
    incfsz AH,w    ; If C=1 update AH to W-> W=FFh+1=00h with the Carry
che
                      ; remains at 1 since incfsz does not modify the
     addwf BH,f       STATUS ; this opcode is skipped and therefore
; per cui:
; FFFFh + FFFFh = 1FFFEh
;result in BL=FEh e BH=FFh (little endian)
; e il result 1 nel Carry
```

which is correct.

The presence of the Carry allows you to extend the operation to more bytes. You can deal with numbers that are extended to 3 bytes (24 bits) or four bytes (32 bits) or even more. Obviously, the limitation is given by the increase in the complexity of the algorithms and the corresponding need for time for execution and memory space both for instructions and for RAM to host the variables. Remember that Baselines do not have too many memory resources, even if the instruction cycle can be brought to only 200ns with a clock of 20MHz (GS mode with external oscillator), which allows not to penalize too much the execution speed even for complex algorithms.

For example, if we want a 32-bit sum, we just need to define 32-bit registers:

```
   UDATA
```

```
AARG0 res 1   ; registri per 32 bit
AARG1 res 1
AARG2 res 1
AARG3 res 1
BARG0 res 1
BARG1 res 1
BARG2 res 1
BARG3 res 1

...
; somma a 32 bit
   movf     AARG0,w
   addwf    BARG0,f
   movf     AARG1,w
   skpnc
    incfsz  AARG1,W
     addwf  BARG1,f
   movf     AARG2,w
   skpnc
    incfsz  AARG2,w
     addwf  BARG2,f
   movf     AARG3,W
   skpnc
    incfsz  AARG3,W
     addwf  BARG3,f
```

The algorithm is simple and fast: it requires 14 us (@ 4MHz) to execute and occupies only 14 locations of program memory.

Subtraction at 16-bit and above

Similarly, you can proceed with 16-bit subtraction:

```
; 16-bit subtraction
; B - A with result in B
   movf     AL,w
   subwf    BL,f
   movf     AH,w
   skpnc
    incfsz  AH,W
     subwf  BH,f
```

Here the Carry will be the Borrow of the operation, with the rules mentioned above. We can immediately extend to 32-bit:

```
; 32-bit subtraction
; BARG - AARG with result in BARG
   movf     AARG0,w
   subwf    BARG0,f
   movf     AARG1,w
   skpnc
    incfsz  AARG1,W
    subwf   BARG1,f
  movf      AARG2,w
  skpnc
   incfsz   AARG2,w
    subwf   BARG2,f
  movf      AARG3,W
  skpnc
   incfsz   AARG3,W
    subwf   BARG3,f
```

However, there are other ways to achieve the same result: subtraction can be done with a sum !
In fact:

B -A = B +(-A).

The negation of the number A is obtained with the **complement of two**, equal to:

2N-A

For example, the two-complement of 20 decimal places is 28-20 = 256 - 20 = 236.
We can verify this by:

20 +(- 20) = 20 + 236 = 256

which, **over 8 bits, yields 0** (with an overflow of 1).

In general, this way is simpler from the point of view of implementation in opcodes: adding to a number the complement of two of another number is the same as subtracting the number. The two's complement is just a matter of reversing the bits (1's complement) and adding 1.

The two's complement is used for negative numbers in integer arithmetic operations, in a range from 0 to ***2n-1***, or 0 to 255 for 8-bit numbers. In *signed arithmetic*, the range is -***2n-1*** to ***2n-1-1***, or -128 to +127 for 8-bit numbers.

More theoretical details on the 2-complement can be found

here. For the complement to 1 there is a specific opcode:

comf.

COMF

E' acronimo di ***Complent File*** e calcola il complemento a 1 (~) del contenuto del registro in oggetto.

The syntax is the usual:

sp	comf	sp	file	,f/w	sp	[; comment]

As already seen for other opcodes, the result of the operation can be saved in the file or in W.
The 1's complement is the inversion of the value of the bits:

```
; file = b'11000101' (.197 o C5h)
  comf     file, w    ; W=b'00111010' (3Ah o .58)
```

From a symbolic point of view, we can define the operation like this:

~file = file ^ 0xFF

The 1's complement operation is necessary to perform 2's complement, which is the negation of the number.

One procedure for denying an 8-bit register is this:

```
; negation (2's complement) of the register
; register = -register
; destination can be 0 or 1 (w or f)
NEGF MACRO register, destination
   comf     register,f
  incf     register,f destination
       ENDM
```

of which there is a pseudo opcode of MPASM:

```
         NEGF  f,d    ; deny file f with destination W or f
```

And for 16-bit:

```
; negation (2's complement) of the 16-bit register
; register = -register
; destination can be 0 o 1 (w o f)
NEGF16  MACRO register, destination
   comf     register,f
  incf     register,f
  skpnz
    decf     register+1,f
   comf     register,fdestination
          ENDM
```

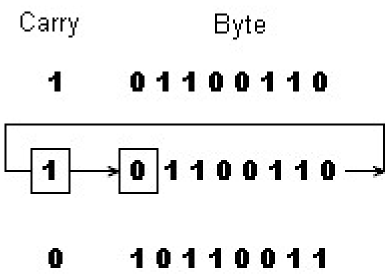
Multiplication and division

While the PIC18Fs have a hardware multiplier that can perform an 8x8 multiplication in a single cycle, the Baselines require a suitable algorithm, consisting of several instructions.

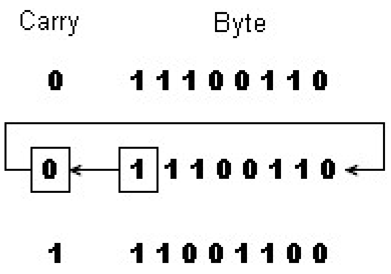
However, we can remember that, since we are dealing with binary logic, multiplication (and division) by 2 or powers of two are performed simply with shift (to the left for multiplication and to the right for division).
For example,:

32 x 2 = 64

In binary, 32 is 00100000. If we rotate the digits to the left one position, we get 01000000 which is precisely 64 decimal. The shift, or rotation, is performed with the opcodes already seen **rrf** and **rlf** . These opcodes integrate the Carry into the rotation:



For example, in the case of a rotation to the right, all bits move one position to the right.
The least significant bit (bit 0) is passed to the Carry.
The previous value of the Carry moves to bit 7.



On the opposite side is the leftward rotation, where all the bits move one position in this direction.
The most significant bit (bit 7) is passed to the Carry.
The previous value of the Carry moves to bit 0.

The use of carry once again allows us to implement operations on multiple bytes. For example, to multiply a 16-bit number by two:

```
; Multiply a 16-bit register by 2
; register = register * 2
   CLRC                       ; Reset Carry
  rlf      AL,f              ; AL=AL*2 with carryover
   rlf      AH,f              ; AH=AH*2 incorporating the Carry
```

The same goes for the division:

```
; divides a 16-bit register by 2
; register = register / 2
   clrc                       ; reset Carry
  rrf      AH,f              ; AH=AH/2 with Curry addition
```

```
   rrf       AL,f          ; AL=AL*2 incorporating the Carry
```

If we have to multiply (or divide) by powers of 2, the above operation will be repeated n times. If, however, we find ourselves having to multiply two registers together, a less simple algorithm is needed. The AN526 details the details and we report the source as such, in the form of a subroutine:

```
;*******************************************************************
;                8x8 Software Multiplier
;*******************************************************************
; Multiply two 8-bit registers.
; The 16-bit result is saved in two other registers.
; Version optimized for execution speed.

; Program Memory : 15 locations
; # of cycles    : 71
;
;*******************************************************************
;
; mulcnd ; multiplying to 8 bits
; mulplr ; 8-bit multiplier
; H_byte ; High byte of 16-bit result
; L_byte ; Low Result Byte
; count   ; Loop Counter
;
mpy_S CLRF   H_byte      ; Reset Result Logs
      CLRF   L_byte
      movlw  8           ; Counter for 8 bits
      movwf  count
      movf   mulcnd,W    ; W=multiplying
      Bcf    STATUS,C    ; clear Carry
Loop  RRF    mulplr,f    ; shift to the right of the multiplier
      BTFSC  STATUS,C    ; Check Bit Value in Carry
       addwf H_byte,f    ; if c=1 add the multiplying to the result
      RRF    H_byte,f    ; Shift to the right of the result (16 bits)
      RRF    L_byte,f
      decfsz count, f    ; counter-1=0 ?
       Goto  Loop        ; No - Other Loop
      retlw  0           ; Yes - End
```

The procedure is very simple: each bit of the multiplier is evaluated and if equal to 1, the multiplier is added to the result; in essence, multiplication is reduced to a series of simple sums: A * B is equivalent to adding A a number of times equal to B.
Despite repeating a loop, the routine runs in up to 71 cycles (equal to 71us @ 4MHz).

The **AN526** mentioned above has several mathematical operations:

- **8x8 unsigned multiplication**
- **Double Precision Multiplication 16x16**
- **Fixed-point division**

- **Double Precision Sum 16x16**
- **Double Precision Subtraction 16x16**
- **BCD (Binary Coded Decimal) conversion routines**
- **Binary to BCD conversion routines**
- **BCD sum**
- **BCD subtraction**
- **square root**

written in Assemblies and usable from Baselines. To be precise, they refer to the PIC16C54, an old OTP (One Time Programmable) chip, predecessor of the Baseline in Flash technology, but similar in terms of instruction set. The algorithms are explained in detail, with the corresponding flowcharts (in English, of course...).
If you're interested in learning more about this, this application note is a good start.

Although the reference of this tutorial is related to Baselines, the instructions used are present in the sets of all families and, with a few limitations, can also be used for Midrange and above.

BCD

A particular area of operations that can be carried out with the microcontroller concerns the presentation of data on displays (7 segments, LCD, CRT, etc.) or their sending to host computers in standard formats. It must be considered that the microcontroller operates on binary-hexadecimal quantities, while we use numbers in decimal form. If we write:

100 +23 = 123

This is much more comprehensible to us than using the hexadecimal form:

64 + 17 = 7B

or, even worse, the binary form:

01100100 + 0010001 = 01111011

But digital logic is binary logic, and you have to resort to some artifice to make the numerical data that the processor has processed immediately comprehensible.

NOTE note: if you have difficulty with the different bases used (hexadecimal, octal, binary), remember that Windows offers a Calculator that allows you to deal with these formats, carry out conversions and operations.

A commonly used standard comes to mind: the BCD. **BCD** stands for ***binary-coded decimal*** and is a commonly used way to represent decimal places in binary code.

In this form, each digit is represented by four bits, for values ranging from 0 (0000) to 9 (1001). The remaining six combinations can be used to represent symbols. Thus, the number 23 in BCD is 0010 0011, which requires two groups of 4 bits, i.e., one byte. A 4-digit number will require 4 groups of 4 bits, i.e. 2 bytes, and so on.
Thus, a byte can contain numbers from 00 to 99; this form is called **packed BCD**: two BCD digits "packaged" into a byte, which reduces the memory footprint.
On the other hand, you can use an **unpacked format**, where, for example, the number 93 requires two bytes, 09 and 03.
Summarizing:

Way	max. num.	binary
BCD unpacked	09	00001001
packed BCD	99	10011001

The packed format allows you to make the best use of the space, while the BCD unpacked allows for quick conversion to ASCII, as you only need to add 3 (0011) in the top half byte.

BCD	binary	ASCII	binary
09	00001001	39	00111001

This is important as the exchange of data between different equipment is usually done using ASCII encoding; so, for example, if we want to send a message to a character LCD display or to a printer, we will have to send ASCII codes.

It is convenient to carry out counting and processing in binary and store data in this form, which saves space (a binary byte can contain up to 255, a packed BCD a maximum of 99) and convert to BCD or ASCII only at the time of sending it to the displays. So, an algorithm that does this function is commonly used by a lot of programs.

```
; hex1bcd3
; converte un byte hex in tre byte BCD (non packed)
; The byte to be converted arrives in W.
; The BCd result is saved in units, tens, hundreds;-----------------------
-------------------------------------
  include "pic12f508.inc"

  EXTERN  unita,decine,centinaia

HEX1BCD3   CODE

hex1bcd3:
      movwf  units      ; hex from W to units
      movlw  0          ; zeroes tens and
      movwf  Hundreds     hundreds
      movwf  0x64
HB0   movlw  units      ;
      subwf  Unit,W     ; 100 decimal
      skpc              ; w=100-unit skips
                          if carry=1

       Goto  HB1        ; otherwise switch to
      movwf  Unit       ; tens save in units
      incf   Hundreds   ; hundreds+1
      goto   HB0        ; Other Cycle
HB1   movlw  0x0A       ; 10 decimal
      subwf  United     ;
      skpc              ; skip if carry=1
       Goto  HB2        ; otherwise go to hundreds
      Movwf  Tens       ; save in units
      Incf   Units      ; tens+1
      Goto   HB1        ; other cycle
HB2   Return            ; end

   GLOBAL hex1bcd3

   END
```

As can be seen, the routine, presented in modular form, operates with simple subtractions:

- subtracts the number 100 from the hexadecimal; As many times as possible this is done, the hundreds counter is increased
- then subtract 10 and increase the number of tens accordingly.
- What's left at the end represents the units.

It is possible to perform binary-BCD conversions for multi-byte numbers. The AN526 has examples, while other examples can be easily retrieved from the WEB.

Obviously, it is also possible to make conversions in the opposite direction, from BCD to hex, or generate ASCII to be sent to a display or to a serial output.

```
;*********************************************************************
; hex1ascii2
; Converts one hex byte to two ASCII characters
; e.g. hex=DF ASCII = 'D', 'F'
; inbound hex in W
; Outbound ASCII in ASCIIH, ASCIIL
;*********************************************************************

 includes "pic12f508.inc"

  EXTERN ASCIIH, ASCIIL

  UDATA
Temp Res 1

HEX1ASCII2   TAILS

hex1ascii2:
     movwf   Temp       ; Save Hex
     CLRF    ASCIIH     ; Reset Output Logs
     CLRF    ASCIIL
     swapf   temp,W     ; High hex nibble in W low nibble
     andlw   0x0F       ; Low nibble only
     movwf   ASCIIH     ; save
     movlw   .10        ; Low BCD nibble only 4 bits
     subwf   ASCIIH,W
     movlw   0x30       ; If it's a number, fix
     skpnc
      movlw 0x37        ; if alphabetical adjust
     addwf   ASCIIH,f   ; and save
; according to ASCII, repeat previous
     operations movf  temp,W     ; hex in w
     andlw 0x0F         ; only low nibble
     movwf ASCIIL
     movlw .10 subwf
     ASCIIL,W movlw
     0x30 skpnc
      movlw 0x37
     addwf ASCIIL,f
     retlw 0

   GLOBAL hex1ascii2

   END
```

This algorithm uses simple steps to add the appropriate high nibble to the content of each nibble of the data to turn it into an ASCII character.
For example, the ABh byte will be rendered as two ASCII bytes containing 41h and 42h; A figure containing 63h will be rendered as 36h and 33h.
Since we have to "disassemble" the two nibbles of the data, which end up at the bottom of the result and we have to add the appropriate value of the ASCII encoding at the top, we use a specific opcode.

SWAPF

This opcode swaps the file in question, targeting the file itself or the W register.
By "swap" we mean that the high nibble (4 most significant bits) is **swapped** with the low nibble (4 least significant bits). Basically, if we have a byte composed like this:

bit	7	6	5	4	3	2	1	0
	Nibble High				**Low Nibble**			
Hex	1	1	0	0	1	0	1	0

After the swap, the situation becomes:

bit	7	6	5	4	3	2	1	0
	nibble alto				**nibble basso**			
Hex	1	0	1	0	1	1	0	0

The syntax is the usual one already widely seen:

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

Executing the statement does not change the flags of the STATUS.

This opcode is very useful in algorithms involving the use of BCD encodings.

The high nibble is reset with an AND, so that only the low nibble is retained, where, through the swap, the high nibble of the value to be converted had gone.
If we subtract 10, there is no carryover if it is a value between 0 and 9. In this case, we add 3 in the high nibble, so that we have the numeric digit in ASCII. If there is a Carry (to be precise, the borrow of subtraction), the datum is a letter between A and F.

Observe the play of the f and W "tails" on the instructions to direct the results to the right positions.
The sum for the ASCII adjustment is made by adding 30h for numbers and 37h for letters. To make this clearer point, let's look at the table below: the character is encoded in ASCII as a low binary nibble and a high nibble equal to 0011 for the numbers 0 to 9 and 0100 for the letters A to F (uppercase, lowercase starts with 0110).

Digit	Binary	Hex	ASCII	
			HEX	Bin
0	00000000	00	30	00110000

1	00000001	01	31	00110001
2	00000010	02	32	00110010
3	00000011	03	33	00110011
4	00000100	04	34	00110100
5	00000101	05	35	00110101
6	00000110	06	36	00110110
7	00000111	07	37	00110111
8	00001000	08	38	00111000
9	00001001	09	39	00111001
At	00001010	0A	41	01001010
B	00001011	0B	42	01001011
C	00001100	0C	43	01001100
D	00001101	0D	44	01001101
And	00001110	0E	45	01001110
F	00001111	0F	46	01001111

Notice that to switch from BCD to ASCII encoding for numbers, just add 3 in the high nibble.
For letters, on the other hand, it is necessary to add 37h. If we take the hexadecimal value 0Ah as an example, we get:

0Ah + 37 h= 41h

i.e **. the ASCII encoding of the letter A.** And so for the others.

One application: the simulation of the program.

Let's check through MPLAB's SIM simulator the functioning of some hexadecimal to BCD conversion algorithms.

Simulation is a method that allows you to follow the operation of a program without having any hardware connected.

A simulator tries to reproduce the behavior of the microcontroller through software.
They are the perfect tool to complete expensive emulators for large development teams, where buying an emulator for every developer is not financially feasible. Typical actions of the simulation are the step-by-step advancement of the program, the possibility of inserting breakpoints, i.e. checkpoints of the execution, **MPLAB** offers a simulator called **SIM** and can be set from the project menu as a debugging tool.

Simulating hardware without physically having it is not easy at all: to have sensible results you need a good knowledge of the connected hardware and the mechanisms of the simulation. Although some simulators may include peripherals outside the microcontroller, it is quite possible that the real hardware reserves a different behavior than the one assumed and this situation is not at all feasible in the simulation; Something higher is needed, which is emulation. Also, it doesn't matter how fast your PC is: there is no simulator on the market that can actually simulate the behavior of a microcontroller in real time, although it can come close. As a result, simulating external events can become a time-consuming exercise, as it is necessary to manually create "stimulus" files that insert external waveforms on the I/O pins into the simulation. For this reason, simulators are better suited for testing algorithms that run completely inside the microcontroller, such as a mathematical routine.
In case you need to get deeper into the hardware, you need to turn to an emulator.

Emulation is a method that allows you to verify the functioning of a program directly in the hardware in which the microcontroller is located, through a particular device, called ICE.

Basically, an emulator is a piece of hardware that behaves exactly like the real microcontroller IC, with all its features. It is the most powerful debugging tool, since microcontroller functions are emulated in real-time and non-intrusively.

Un ICE (In Circuit Emulator) It is a rather expensive piece of equipment, reserved for the professional sector.

For these reasons, Microchip has put aside its ICE (ICE200 and ICE4000) in favor of a different technique.

	To reduce the need for this, Microchip integrates an internal debug engine (IDE - In Circuit Debugger) into many chips, which eliminates the cost of ICE. This makes emulation possible even through economical tools such as Pickit. Unfortunately, none of the Baselines have this option and it is necessary to have a header, which is a special version of the chip that integrates the IDE. This involves an additional cost, not excessive, but which could only be appreciated by the professional.

If, however, we need to verify the operation of an algorithm that operates only inside the chip, making use of RAM memory registers and opcodes, simulation is still the method that allows the fastest response and an equally rapid correction of any errors, without requiring deep knowledge of the operation of the simulator itself.

We have retrieved from WEB some routines that convert a hexadecimal byte to BCD. For example, this one:

```
;*******************************************************
; 8-bit binary to BCD conversion
; bin    ; contains the binary value to convert
; bcdH   ; result
; bcdL
; temp
; temp1
; Executes in 86 instructions

bin2bcd      movlw     of 5'
             movwf     temp1
             CLRF      bcdL
             CLRF      bcdH
             rlf       bin,F
             rlf       bcdL,F
             rlf       bin,F
             rlf       bcdL,F
             rlf       bin,F
             rlf       bcdL,F
bin2bcd1     movfw     bcdL
             addlw     0x33
             movwf     Temp
             movfw     bcdL
             BTFSC     temp,3
             addlw     0x03
             BTFSC     temp,7
             addlw     0x30
             movwf     bcdL
             rlf       bin,F
             rlf       bcdL,F
             rlf       bcdH,F
             decfsz    temp1,F
             Goto      bin2bcd1
               return
```

The routine is written for the Midrange instruction set, which includes the **addlw** opcode, which does not exist in the Baseline set. We can, however, convert it, using **addwf**.

This means reversing the operators at various points:

14-bit core	12-bit core
	`Bin1Bcd2  movlw  0x33`
	`          addwf  bcdL,w`
`bin2bcd1  movfw  bcdL`	
`          addlw  0x33`	`          movwf  temp`
`          movwf  temp`	`          movfw  bcdL`
`          movfw  bcdL`	`          btfss  temp,3`
`          btfsc  temp,3`	`           goto  s30`
`          addlw  0x03`	`          movlw  0x03`
	`          addwf  bcdL,w`
`          btfsc  temp,7`	`S30       btfss  temp,7`
`          addlw  0x30`	`           goto  s00`
`          movwf  bcdL`	`          movwf  bcdL`
	`          movlw  0x30`
`          rlf    bin,F`	`          addwf  bcdL,w`
`          rlf    bcdL,F`	`S00       movwf  bcdL`
`          rlf    bcdH,F`	`          rlf    bin,F`
`          decfsz temp1,F`	`          rlf    bcdL,F`
`          goto   bin2bcd1`	`          rlf    bcdH,F`
`           return`	`          decfsz temp1,F`
	`           Goto  bin2bcd1`
	`           Retlw  0`

Work? We need some kind of system to be sure. We can try to mentally execute the instructions with the help of pen and paper, but there is a much more effective and less laborious method: using **MPLAB-SIM**.

First, let's write a source that contains the routine and the minimum of other information that allows the compiler to do its job:

```
;####################################################################
            LIST p=10F200
            #include <p10F200.inc>
            radix dec
;===================================================================
; No WDT
 __config _WDT_OFF
;
; RAM
     UDATA
     bcdH
     bcdL
     bin
     temp1
     temp
```

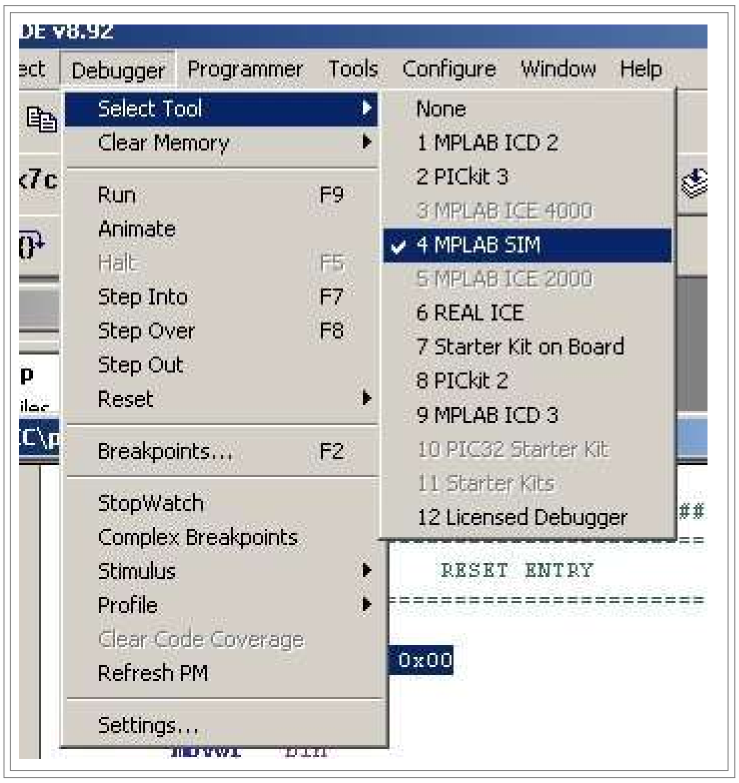
That's more than enough!
We choose a minimum Baseline that has just the right amount of memory needed; you don't need more, since the program to be tested only accesses RAM, without using any peripherals. And, since all Baselines have the same set of instructions, the algorithm will work on any other chip (including those from families higher than Baselines, which include in their set all the opcodes used here).
Since there is no hardware attached, the initial config only excludes the watchdog to prevent its intervention from disturbing debugging, resetting when the time runs out. The fact that we are using a simulator without real hardware attached, does not exclude that the internal functions are abolished: everything that exists in the chip being debugged is simulated as close as possible to the real behavior.

The actual program consists only of pre-loading a hexadecimal value to be converted and calling the procedure:

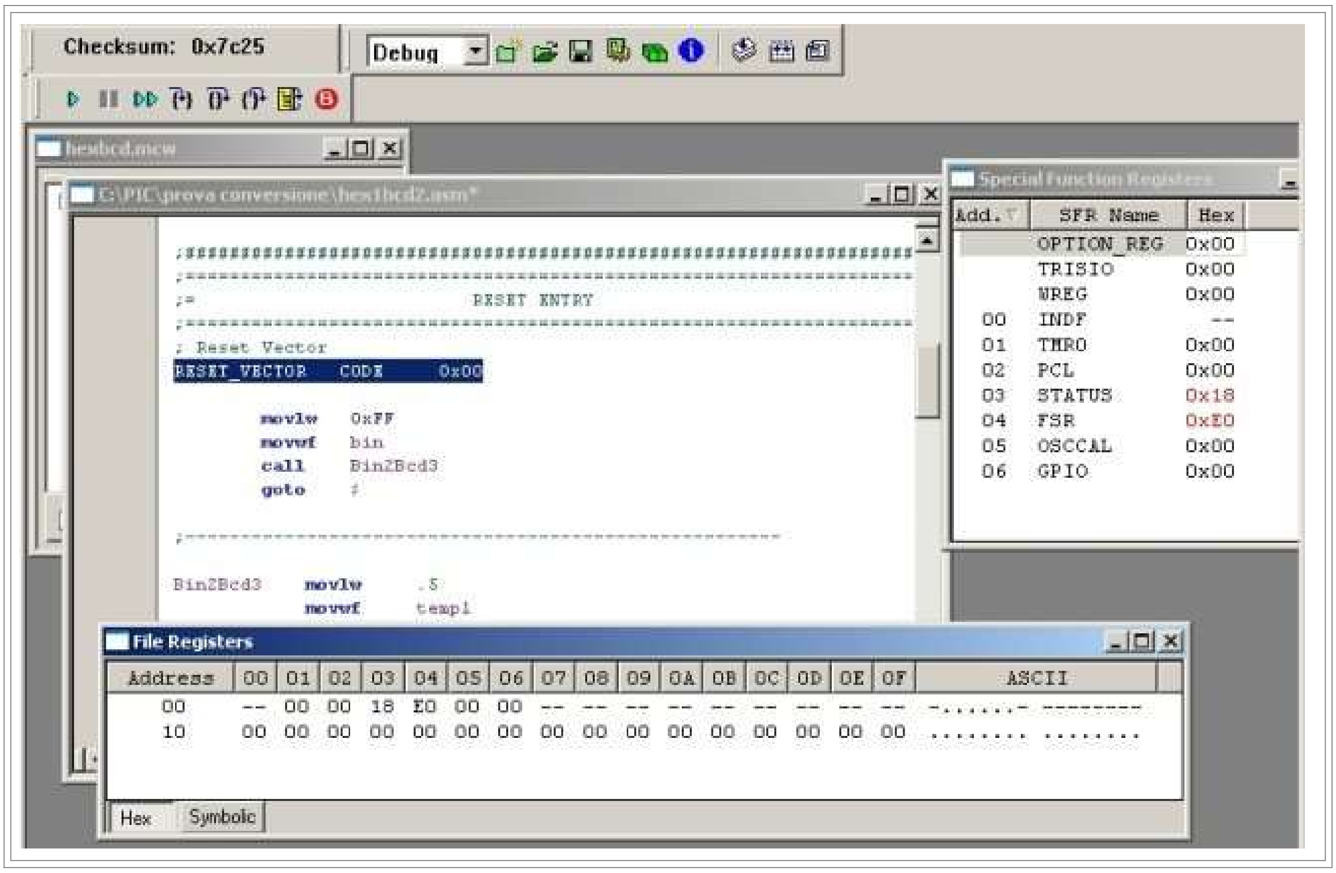
```
RESET_VECTOR CODE 0x00

   movlw 0xFF
   movwf  bin
   call Bin2Bcd3
   goto  $
```

Since we are not interested in a relationship with the clock, there is no need to adjust the internal oscillator (also because it would have no effect on the simulation, which uses its own clock, depending on the processor).

Now let's create the MPLAB project and from the main menu select as ***Debugger -> Select Tool -> MPLAB SIM***:



In addition, we open the ***Special Function Registers*** and ***File Registers*** windows, which display the SFR control logs and the RAM area, respectively. For this, simply call up the menu from ***View***:


Checksum: 0x7c25
Debug
hexbcd.mcw
G:\PIC\prova conversione\hex1bcd2.asm*
Special Function Registers
File Registers
Hex
Symbolic

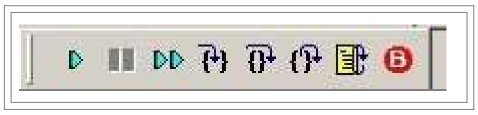
```
;##########################################################################
;==========================================================================
;=                          RESET ENTRY
;==========================================================================
; Reset Vector
RESET_VECTOR    CODE    0x00

            movlw   0xFF
            movwf   bin
            call    Bin2Bcd3
            goto    $

;------------------------------------------------------------

Bin2Bcd3    movlw   .5
            movwf   temp1
```

Add.	SFR Name	Hex
	OPTION_REG	0x00
	TRISIO	0x00
	WREG	0x00
00	INDF	--
01	TMR0	0x00
02	PCL	0x00
03	STATUS	0x18
04	FSR	0xE0
05	OSCCAL	0x00
06	GPIO	0x00

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
00	--	00	00	18	E0	00	00	--	--	--	--	--	--	--	--	--	-......- --------
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Windows are resizable and movable according to Windows standards.
The SFR window shows all the available control logs, with their position in memory and the value they take on during the execution of the program.
The File Registers window shows the contents of the RAM and the ASCII equivalence, if we have selected the ***Hex mode***; if we select the Symbolic mode, the opcodes will be presented.

We compile and, if there are no errors, we move on to execution. This is controlled by the buttons in the main menu:

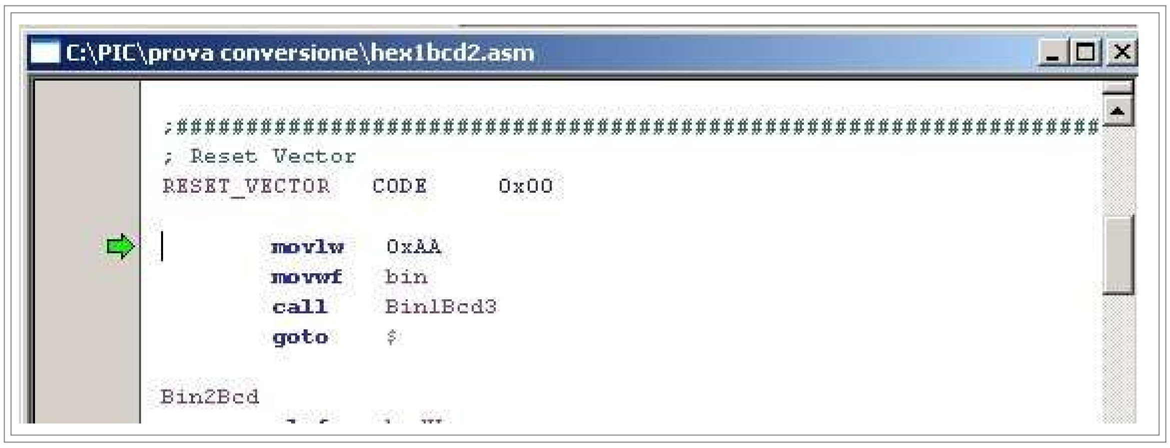


The functions are displayed by the window that opens automatically by pointing to an icon. From left to right:

	Run	Starts running the program at rated speed. Execution is stopped with ***Halt*** or by entering a Breakpoint
	Halt	Stop the execution. Illuminates if there is a ***Run run*** activated
	Animated	Launch the program by scrolling through the opcodes in slow motion. Execution is stopped with ***Halt*** or by entering a Breakpoint
m	***Step Into***	It advances the Program Counter one step, but does not enter the subroutines, which are executed at rated speed. This feature allows step-by-step, analyzing the instructions one by one, but skipping the display of program areas that would only get in the way, such as waste-time routines or other routines that you are sure will work and that you don't need to debug meticulously
	Step Over	Same as above, but it also displays what happens in the subroutines
	Step Out	Same as above, allowing an exit from the subroutines
	Reset	Simulate an MCLR by resetting the virtual chip. **It doesn't stop the execution** (as MCLR doesn't), **but it just returns the Program Counter to the Reset vector**
	Breakpoint	It manages the Breakpoints, i.e. the points at which you want to stop the execution.

More details on how the SIM works can be found in the MPLAB Help or manual.
Here we only add that the SIM, like any other simulator, has a certain number of limitations that, usually, are not an obstacle to debugging, but which, in special cases of large commitment of the chip's resources, must be considered.

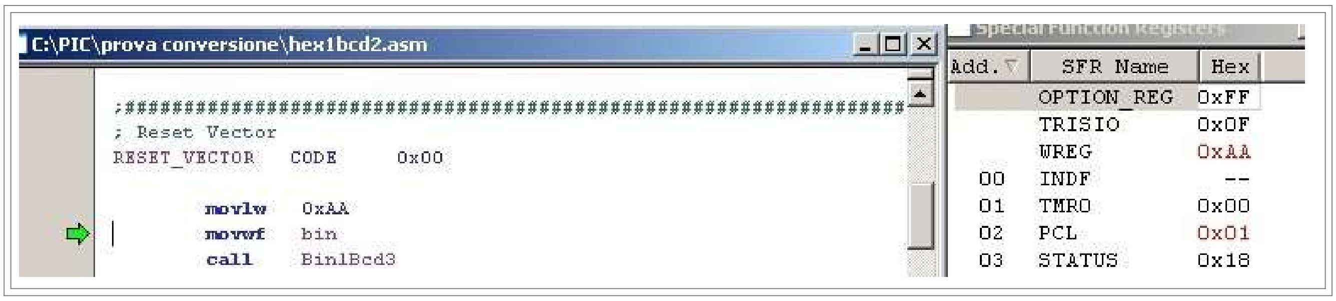
Once the source is compiled, we can immediately run it by pressing ***StepInto***: the debug window opens and the Program Counter advances one step with each press.



C:\PIC\prova conversione\hex1bcd2.asm

```
;######################################################################
; Reset Vector
RESET_VECTOR    CODE      0x00

        movlw    0xAA
        movwf    bin
        call     Bin1Bcd3
        goto     $

Bin2Bcd
```

An index in the form of a green arrow scrolls to the left of the listing indicating the line that will be executed.

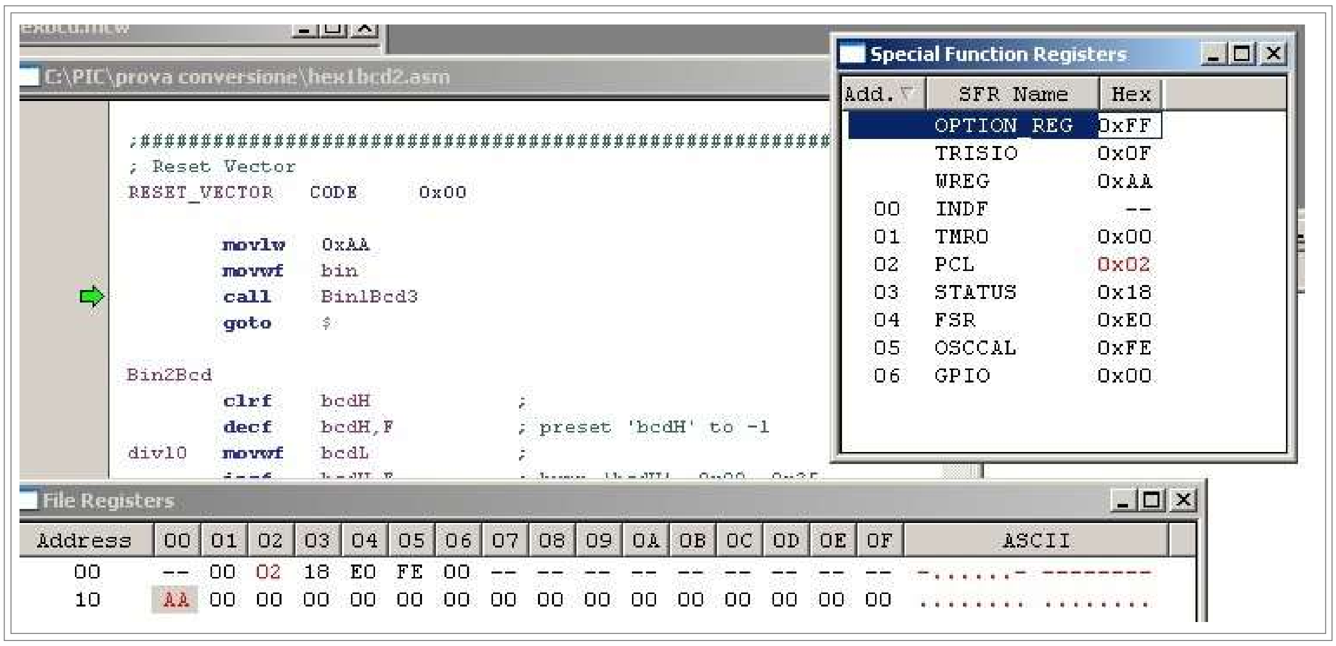


C:\PIC\prova conversione\hex1bcd2.asm

```
;######################################################################
; Reset Vector
RESET_VECTOR    CODE      0x00

        movlw    0xAA
        movwf    bin
        call     Bin1Bcd3
```

```
Add.   SFR Name     Hex
       OPTION_REG   0xFF
       TRISIO       0x0F
       WREG         0xAA
00     INDF           --
01     TMR0         0x00
02     PCL          0x01
03     STATUS       0x18
```

At each execution, the **SFR window** will indicate in red the change of the contents of the registers: for example, if the AA movlw line is executed , the ***Program Counter Low*** (**PCL**) will be switched from the reset vector (00) to the next location (01) while the WREG register will be loaded with AAh.



C:\PIC\prova conversione\hex1bcd2.asm

```
;######################################################################
; Reset Vector
RESET_VECTOR    CODE      0x00

        movlw    0xAA
        movwf    bin
        call     Bin1Bcd3
        goto     $

Bin2Bcd
        clrf     bcdH          ;
        decf     bcdH,F        ; preset 'bcdH' to -1
div10   movwf    bcdL          ;
```

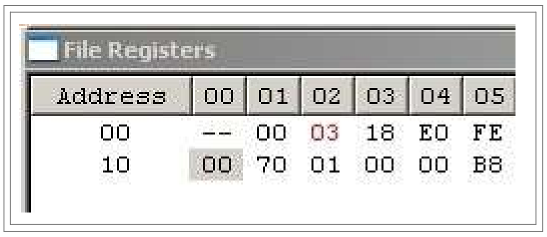
Special Function Registers

```
Add.   SFR Name     Hex
       OPTION_REG   0xFF
       TRISIO       0x0F
       WREG         0xAA
00     INDF           --
01     TMR0         0x00
02     PCL          0x02
03     STATUS       0x18
04     FSR          0xE0
05     OSCCAL       0xFE
06     GPIO         0x00
```

File Registers

```
Address  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F        ASCII
  00     -- 00 02 18 E0 FE 00 -- -- -- -- -- -- -- -- --  -......- --------
  10     AA 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ........ ........
```

Continuing the manual advance in steps, we execute the second line, **`movf bin`** and consequently we see the modification of the **PCL** and the **RAM 10h** memory location, label **`bin`**, in which the WREG is loaded.
Moving on, let's move on to the execution of the subroutine (the PCL will point to the start location of the subroutine, after the call). At the end, we observe how in RAM 11h and 12h (label **`bcdL`**, **`bcdH`**) we find 70h and 01h,

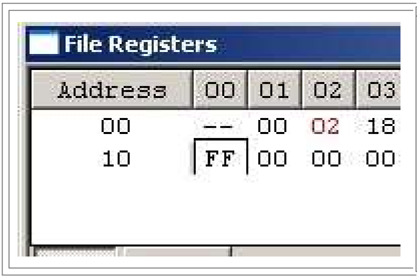

File Registers

Address	00	01	02	03	04	05
00	--	00	03	18	E0	FE
10	00	70	01	00	00	B8

since:

AAh = 170 dec

that is, in **`bcdL`** there are tens and units, while in **`bcdH`** the hundreds. It's a packed BCD and little endian.


File Registers

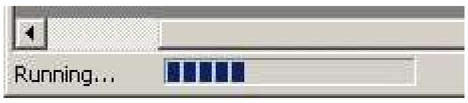
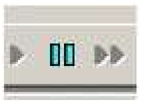
Address	00	01	02	03
00	--	00	02	18
10	FF	00	00	00

We can test other numbers, changing the initial value, which requires compilation.

But we can also manually modify the value that is loaded into the bin, by selecting the cell and introducing the new value.

A double click on the cell to be modified allows us to vary its value during the execution of the simulation; this allows you to change parameters in the registers without resorting to a recompilation or to test a different variable on the fly, impose a specific value on a bit of an SFR, etc.

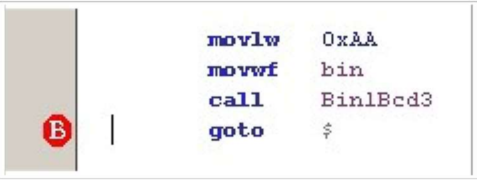
If, instead of the step button, we use the ***Run*** button, the program runs at a speed similar to what it would actually have in the chip. Since the execution of the instructions is NEVER BLOCKED, but only conducted in the loop closed on itself, we apparently see no effect.

	We can detect that the program is running from the ***Running*** bar that lights up in **the lower left corner** of the MPLAB window.
	We will be able to block the execution of the loop with the Halt button which is now activated

Once the program is stopped, we will be able to see in the debug window the cursor that indicates the next line running (in this case the one of the loop) and the situation of the registers.

If we use the ***Animate*** key, we see the instructions scroll at low speed, with the SFR and RAM windows adjusting to the results. The animation is stopped with the ***Halt key***.

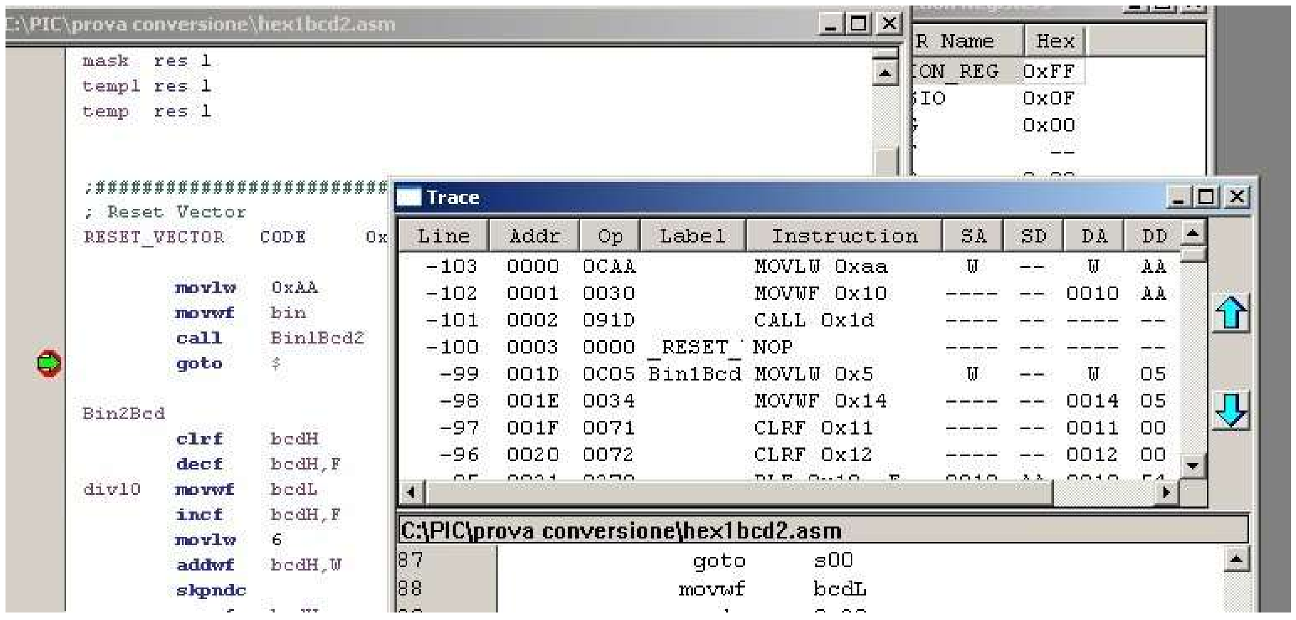
We can introduce execution checkpoints by imposing breakpoints: this is achieved by double-clicking the left button on the line where you want the block.



```
        movlw   0xAA
        movwf   bin
        call    Bin1Bcd3
B |     goto    $
```

If we double click on the loop line, the red breakpoint indicator will appear: by launching the program with a Run, it will stop the first time it encounters the breakpoint.

Breakpoints are also managed from the ***Debugger -> Breakpoints*** menu or the ***F2 hotkey.***



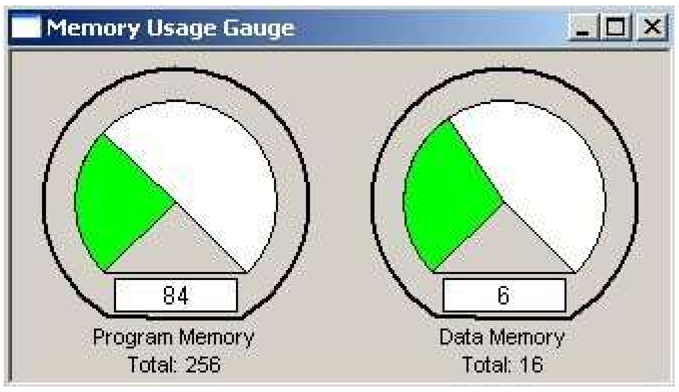
```
mask   res 1
templ  res 1
temp   res 1


;##########################
; Reset Vector
RESET_VECTOR    CODE      0x

          movlw   0xAA
          movwf   bin
          call    Bin1Bcd2
          goto    $

Bin2Bcd
          clrf    bcdH
          decf    bcdH,F
div10     movwf   bcdL
          incf    bcdH,F
          movlw   6
          addwf   bcdH,W
          skpndc
```

```
R Name      Hex
ON_REG      0xFF
IO          0x0F
            0x00
              --
```

```
 Line  Addr  Op    Label     Instruction   SA    SD  DA    DD
 -103  0000  0CAA            MOVLW 0xaa    W     --  W     AA
 -102  0001  0030            MOVWF 0x10    ----  --  0010  AA
 -101  0002  091D            CALL 0x1d     ----  --  ----  --
 -100  0003  0000  _RESET_   NOP           ----  --  ----  --
  -99  001D  0C05  Bin1Bcd   MOVLW 0x5     W     --  W     05
  -98  001E  0034            MOVWF 0x14    ----  --  0014  05
  -97  001F  0071            CLRF 0x11     ----  --  0011  00
  -96  0020  0072            CLRF 0x12     ----  --  0012  00
```

```
C:\PIC\prova conversione\hex1bcd2.asm
87                goto     s00
88                movwf    bcdL
```

The stop on the breakpoit is identified by the cursor that overlaps the symbol. If ***we have selected the*** Trace option from the ***View menu***, we open an additional window in which the trace of the executed instructions appears, in which important information appears, such as the number of cycles committed. The number of breakpoints available depends on the tool used and the processor chosen.

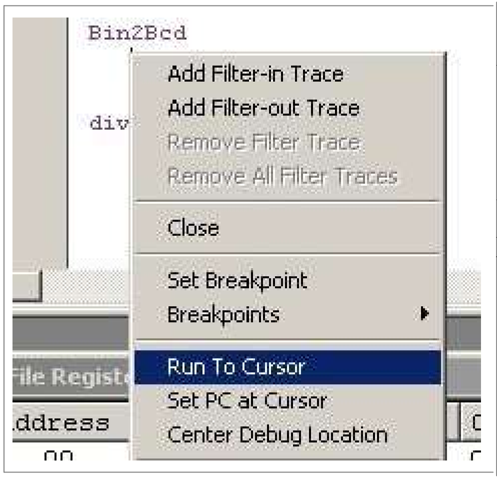


Without resorting to the manual, which is certainly not a friendly read, the MPLAB Help offers a good perspective for clarifying various points of the environment and also of the SIM (in fact, in addition to these views, numerous other functions are available to be able to verify the correctness than what was written in the source, even without having the hardware available).

For example, the Memory Usage Gauge window, always accessible from the View menu, which graphically and numerically represents the memory usage of the

	program

In the project given as an example, the source contains other conversion routines, with slightly different functions and based on different algorithms, which you can try by simply addressing the call line to them.
There is also a way other than recompilation, using the local menu of the row: just point to a line and with a right click you unroll the submenu.



Among many options, **Set PC at Cursor** moves the Program Counter to that line and allows step-by-step from that point.

It is therefore possible to execute a section of the program and then move to another non-consecutive line at any point in the program and resume execution from there.

Note that lines containing only labels are not pointable, since they do not contain opcodes; the next one will contain an instruction.

The simulator is a powerful tool, the use of which is little known. It should be noted that up to now we have written "blind" sources, i.e. programs that we have "run" in the mind and verified with it. Since the compiler indicates "syntax" problems, what is compiled and then written into the chip at the time of programming, works only and exclusively if there are no logical errors.
Unfortunately, due to the lack of knowledge of the possibilities offered by tools such as simulators and IDEs, it is common for the experimenter to write, compile and program, and then stand by and watch the chip not do what is intended, without knowing where to go to correct the problem.
If the compiler indicates formal errors, the simulator allows you to correct logic errors and is the essential operation to be carried out once the other causes of malfunction have been excluded (which, remember, are very often trivial hardware errors or even just false contacts on low-cost breadboards. The use of LPCuB or other similar development board is intended to minimize hardware problems, leaving the focus on programming).

The operation of writing programs blindly, even of high complexity, is possible as long as you use a structure composed of modules whose correct functionality has been verified. Modules, such as this one of the binary-BCD version, which can be debugged with great simplicity and effectiveness through the SIM: it is always possible that an error is hidden in the structure of the algorithms and, through emulation, this is likely to be clarified in a very short time.

Moreover, from a didactic point of view, step-by-step emulation allows you to fully understand what is happening inside the microcontroller, the result of the instructions, the variation of the registers, giving a greater awareness to the writing of the source.

Obviously, debugging a complex program is not a simple operation and requires a good "hand" with the simulator, which you can create over time by trying to pass the SIM i programs (or parts of them) that we have seen so far. In this sense, for example, it is interesting to check what happens to the Program Counter when we go to execute lookup tables of the retlw type.

Hex1Bcd2.asm

```
; ####################################################################
; Testing for Bin->BCD Routines

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

        radix     DEC

;==================================================================
;
; No WDT
  __config _WDT_OFF

;
; RAM
  UDATA
Bin    RES 1
bcdL res 1
bcdH res 1
Mask Res 1
temp1 res 1
Temp Res 1


; ######################################################################
; Reset Vector
RESET_VECTOR    TAILS    0x00

        movlw    0xAA
        movwf    bin
        call     Hex1Bcd2
        goto     $

Bin2Bcd
        CLRF     bcdH            ;
        decf     bcdH,F          ; preset 'bcdH' to -1
Div10   movwf    bcdL            ;
        incf     bcdH,F          ; bump 'bcdH', 0x00.. 0x25
        movlw    6               ; using "packed bcd" format
        addwf    bcdH,W          ; bcd "digit carry"?
        skpndc                   ; no, skip, else
        movwf    bcdH            ; fix 'bcdH'
        movlw    10              ; bcdL = bcdL - 10
        subwf    bcdL,W          ; borrow?
        bc       div10           ; no, branch, else
PrintIt
        movlw    " "             ; prep leading zero suppression
        movwf    mask            ; mask = 0x20 = " " (space char)
        swapf    bcdH,W          ; get hundreds, 0..2
        call     PutDigit        ; print " " or "1".."
        movf     bcdH,W          ; 2" get bcdH, 0..9
        call     PutDigit        ; print " " or "0".."
        movf     bcdL,W          ; 9" get bcdL, 0..9
        Goto     PutNumber       ; always print "0".." 9"
```

```
PutDigit
        andlw   0x0F            ;
        skpz                    ;
PutNumber
        Bsf     mask,4          ; mask = 0x30 = "0"
        iorwf   mask,W          ; wreg = " " or "0".."
                                9"
        retlw   0

;-------------------------------------------------------

Bin1Bcd2    movlw     .5
            movwf     temp1
            clrf      bcdL
            clrf      bcdH
            rlf       bin,F
            rlf       bcdL,F
            rlf       bin,F
            rlf       bcdL,F
            rlf       bin,F
            rlf       bcdL,F
bin1bcd21   movlw     0x33
            addwf     bcdL,w

            movwf     temp
            movfw     bcdL
            btfss     temp,3
             goto     s30
            movlw     0x03
            addwf     bcdL,w

S30         btfss     temp,7
             goto     s00
            movwf     bcdL
            movlw     0x30
            addwf     bcdL,w

S00         movwf     bcdL
            rlf       bin,F
            rlf       bcdL,F
            rlf       bcdH,F
            decfsz    temp1,F
            goto      bin1bcd21
              retlw   0

;---------------------------------------------------

Hex1Bcd2
        movwf   Bin
        CLRF    bcdL
        CLRF    bcdH

H1B20
        movlw   .100
        subwf   bin,W
        skpc            ; BTFSS  STATUS,C
         goto   H1B21
        incf    bcdH,F
        movwf   bin
        goto    h1b20

H1B21
        movlw   .10
        SUBWF   bin,W
        SKPC            ; BTFSS  STATUS,C
         Goto   h1b23
        Movwf   bin
        Incf    bcdL,F
        Goto    h1b21

; CONV ASCII
H1B23
        movlw   .48
        addwf   bin,F
        addwf   bcdL,F
        addwf   bcdH,F
        retlw   0



        END
```