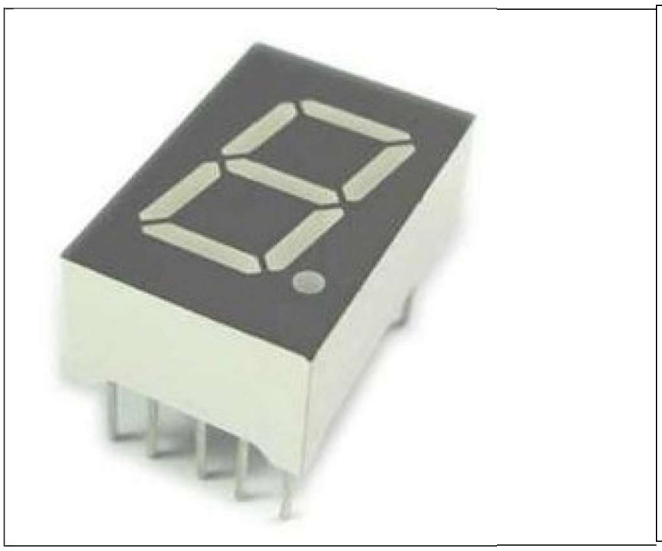
Knowledge of baseline

Controlling a 7-segment display - Lookup table

The purpose of the tutorial is to use a 7-segment display.

So far, we have seen how it is possible to control digital outputs using LEDs.
However, directly or with appropriate buffers, it is possible to use relays, motors, electromagnets, lamps and all kinds of loads. The concept is always common: with a certain level applied to the bit corresponding to the pin, the load is triggered; By programming the bit with the opposite value, the load is turned off. This makes it possible to introduce the "intelligence" of the processor, or rather, the intelligence of the programmer that is expressed through the program set in the processor, in all kinds of devices, from household appliances to toys, from communication equipment to industrial controls.

If we stay in the field of small currents and voltages, we can consider a rather commonly used device, namely the 7-segment display.

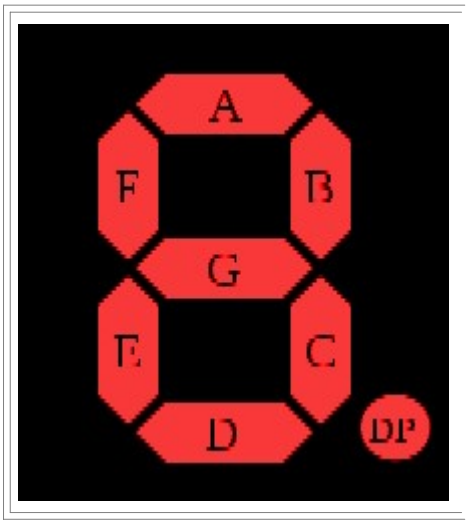


A 7-segment display is a set of LEDs, one or more for each segment, that can be powered individually.

They are arranged in the form of an '8': when all segments are lit, the display shows the number 8. By lighting up other combinations of segments, you will be able to represent the digits from 0 to 9, different letters of the alphabet, as well as a fair amount of other combinations.

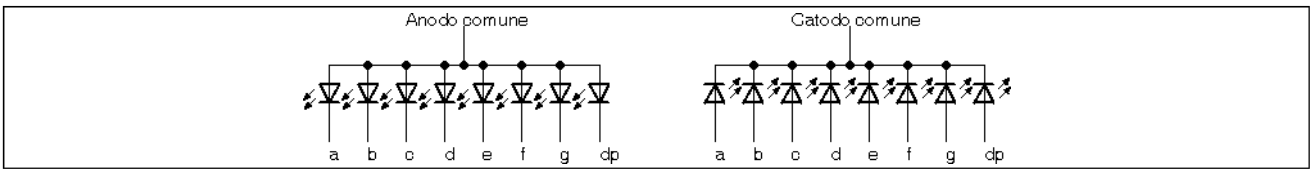
There are also 16-segment displays to represent alphanumeric digits, but they usually have a higher cost, also because, given the drop in production costs of an LCD display, the latter presentation system is preferred for more characters.

In any case, 7-segment displays still have a very wide field of application, since being bright in themselves, they allow them to be read clearly in ambient light conditions where LCDs could not give good results; They are widely used in instruments, clocks, reservation systems and, in general, where numerical information has to be displayed.

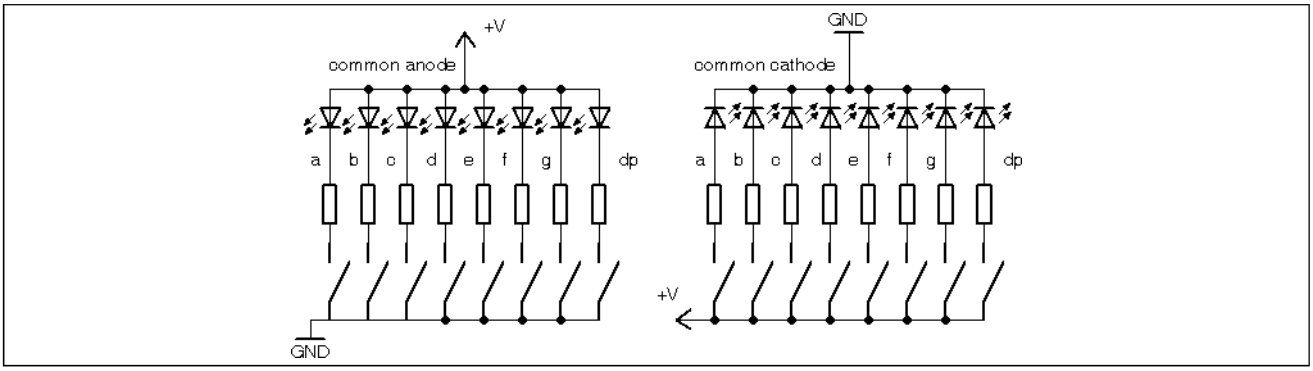


As a de facto standard, segments are denoted with the letters a to ***g***, while it is also possible to have an eighth element that acts as a decimal point (***dp***) in a numerical representation (the decimal point can be on the right, left, or even at the top, while you can have modules that collect more than one digit in a single block.

In the display, the LEDs are usually connected with a common end, so there are two types: LEDs with a common anode and LEDs with a common cathode.



In the common anode solution, this will be connected to the positive voltage, while the cathodes of each individual LED will be controlled by connecting them to the ground. The opposite is true for the common cathode.



If we think of replacing the switches in the diagram above with the output buffers of the microcontroller, it becomes clear how we can control the display from the program.
The logic will be "positive" for common cathode displays and "negative for others:

Pin	Common Anode	Common Cathode
0	on	off
1	off	on

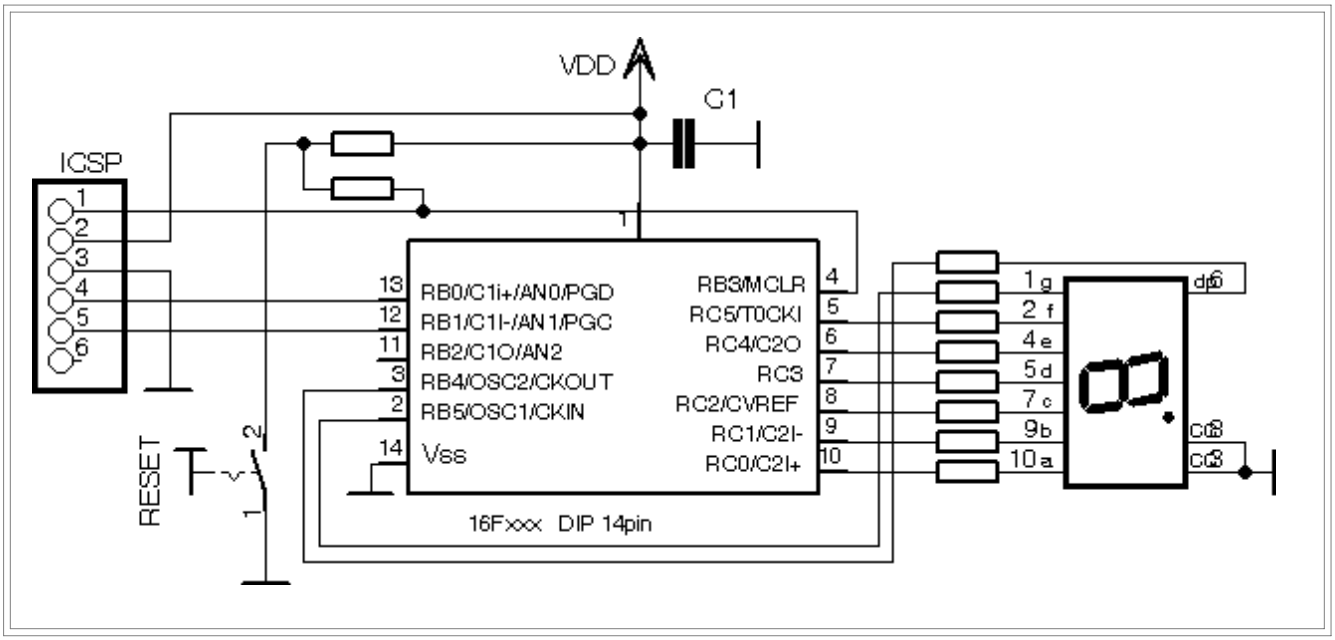
Each segment, since it is LED, needs a resistor to limit the current. Usually the current in the segments is less than 20mA and you can get bright displays even with 10mA or less (in the case of low current models, even just 1 or 2mA); Since the ins of the PIC can handle up to 25mA, the segments are directly controllable.

Our LPCuBoard makes available three displays, of the common cathode generator, and we can now see how to control one of them through the microcontroller with appropriate programming.

What we want to achieve

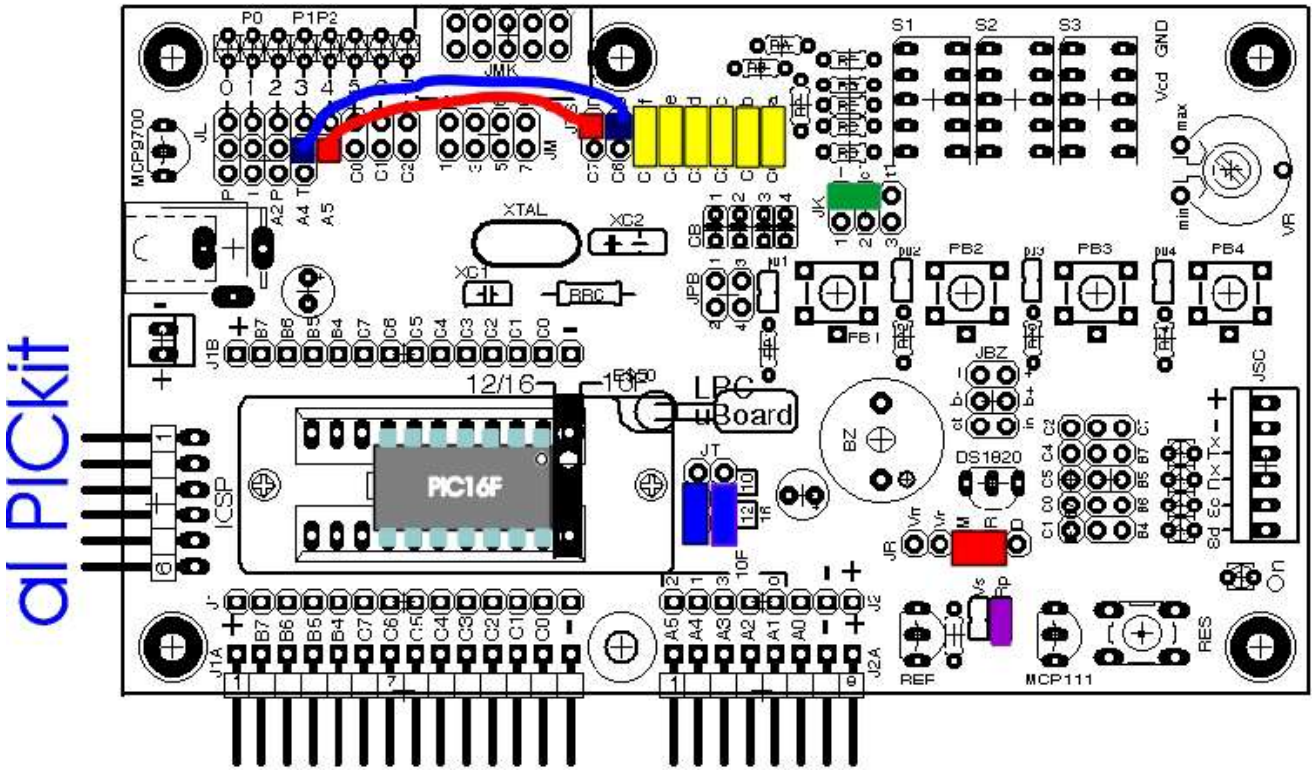
We want to control the 7-segment display so that the digits 0 to 9 and the letters A to F are presented in sequence.

Having to control 7 LEDs, you need to have 7 I/O pins available and this immediately puts out of play the chips in 8-pin packages, which, at most, can offer 5.
We therefore need to use a chip with 14 or more pins, such as the 16F505 or 526 we've already seen. Here is the wiring diagram:



Since the Ports are only composed of 6 bits, we use the entire **PORTC** (**RC0-RC5**) for the ***a-f*** segments and borrow **RB4** from the **PORTB** for the ***g segment*** and **RB5** for the dp decimal point.
The displays on the **LPCuBoard** are of the common cathode type and therefore the scheme is adapted for these, connecting the common cathode to ground (Vss).

These links are already present on the demo boards, for which the only action will be to insert the required jumpers. If we use a breadboard, we must perform the ones indicated. For the **LPCuBoard**, the connections are as shown in the following image.
The "green" jumper connects the common cathode of the S1 display to the Vss (gnd), so that the segments can be controlled as indicated above (pin=1, LED on).
Since 16F505/526 has 6-bit ports and the display needs at least 7 bits (plus one for the decimal point), we need to use part of **PORTC**, connected with the "yellow" jumpers, and two bits of **PORTB** that we connect with two flying jumpers (**PORTB4** connected to the ***g segment*** and **PORTB5** connected to the ***dp***).



We always observe the correct insertion of the chip on the socket and the position of the "blue" jumpers. The "red" and "purple" jumpers connect the ***RESET button*** to **`RB3`**.

What about the other chips?

Unfortunately, **8-pin PICs don't have enough lines to drive the display**. Despite this, we could still connect the segments through an intermediate element, i.e. an integrated "decoding" (CD4511 or similar), which allows you to manage the visible combinations in order to represent the numerical digits having only 4 lines as input (BCD-7 segments converter) or to an I/O expander, but these solutions complicate the hardware and software and end up having a higher cost than the use of a chip with a greater number of pins. For example, PIC12F508+CD4511 costs around €1.1, while a 14-pin 16F505 costs €0.86.

So, although we will see how to use I/O expansions, the solution is practically never cost-effective and should be left in the area of experimentation or last resources to be evaluated.

The source

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

- **Have the digits 0 to 9 and the letters A to F light up in succession on the display and repeat the cycle indefinitely.**

And how:

- **programming the PORTC and RB4/RB5 pins as digital outputs and then sending the right configurations to cause the desired segments to turn on.**

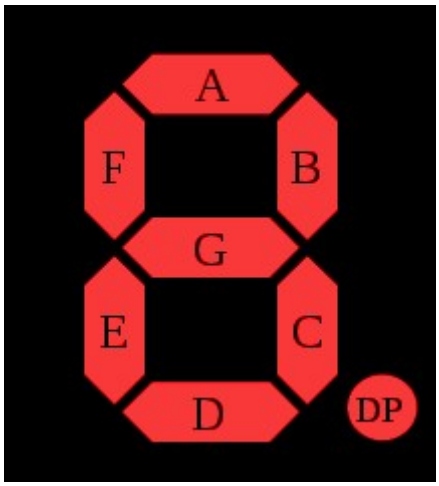
We observe that the key lies in the words "**the right configurations**": here we must not turn on LEDs randomly, but only those that determine the representation of the desired digits

---

A table?

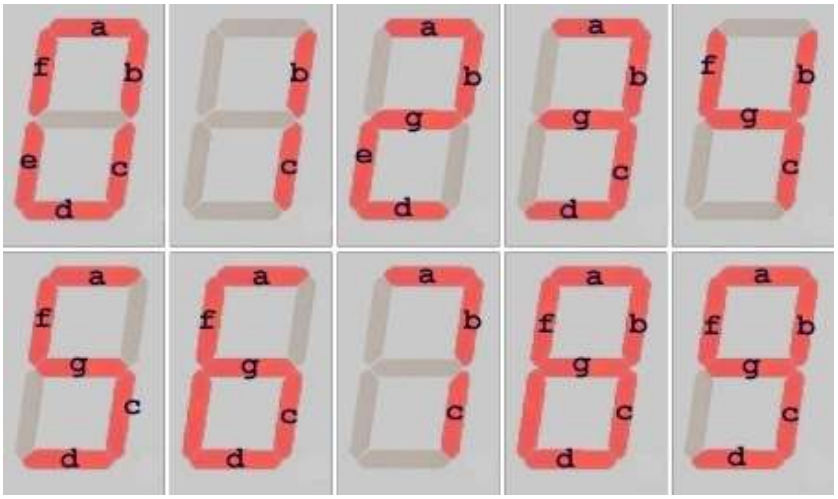
Let's take a look at the diagram of the links in the form of a table:

Segment	I/O
at	RC0
b	RC1
c	RC2
d	RC3
and	RC4
f	RC5
g	RB4



If we want to represent the digit 1, we will have to turn on segments b and c.
If we want to represent the digit 3 we will have to turn on the segments a, b, c, d, g. And so on.
It is obvious that if I want to display a symbol, I will have to turn on a specific series of LEDs and not others.

We can then draw up a table that relates the symbols to the segments to be lit:

Symbol	Segments							
	g	f	and	d	c	b	at	
0		x	x	x	x	x	x	
1					x	x		
2	x		x	x		x	x	
3	x			x	x	x	x	
4	x	x			x	x		
5	x	x		x	x		x	
6	x	x	x	x	x		x	
7					x	x	x	
8	x	x	x	x	x	x	x	
9	x	x		x	x	x	x	

The x's indicate the segments that need to be lit.
We can bring this table into the source and use it to control the I/O, in this form:

```
; segment data table - display Common cathode
;
segtbl
       andlw 0x0F          ; Low nibble only
       addwf pcl,f
       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
```

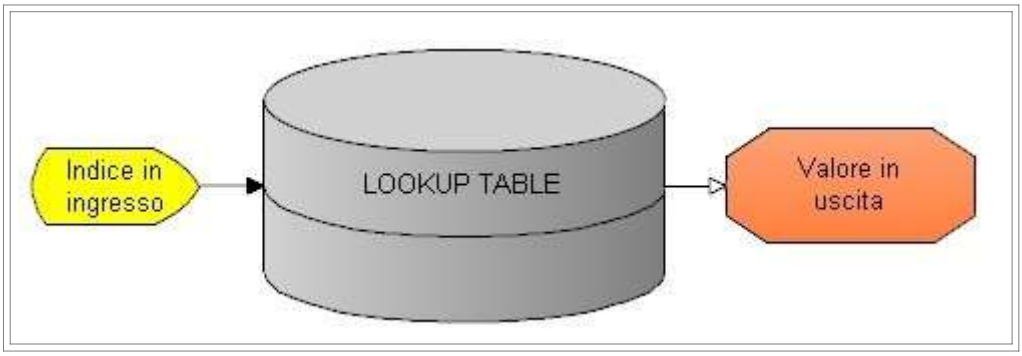
Notice that the x's in the table above correspond to bits at level 1 (to light up the corresponding LED, connected between pins and Vss). Segments that should remain off have a value of 0.

What we have just written is a so-called ***Lookup Table*** , also called ***a RETLW table***. Let's try to explain what it is.

LOOKUP TABLE

A ***Lookup Table*** (***LUT***) is a data structure, usually an array, used to replace computational operations with a simpler "***lookup***" operation based on an index or pointer.
The speed gain can be significant, because retrieving a value from the in-memory table is faster than performing calculations with long execution times.
The tables make it possible to associate each permissible combination of input data with a corresponding combination of output data. A classic application is the one we are analyzing, which is to convert the numbers from 0 to 9 (or 0 to F in hexadecimal) into masks for the command bits of seven-segment displays.

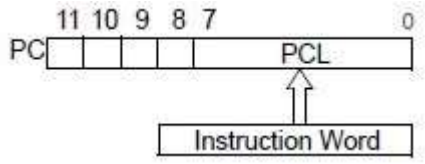
From a functional point of view, we can schematize the lookup table as follows:



You enter the table with an index number and exit with the corresponding value. With an analogy, you enter the library with the name of a book and leave with this book: the lookup table is the library catalog that allows you to retrieve the object by name.

These tables are part of the possibilities of the Baseline CIPs and the structure of the Baseline PICs, with the use of the **`retlw statement`**, which we have already encountered.
A ***retlw table*** consists of a list of **`retlw`** lines whose values represent what will be output; in our case, these are the command masks of the segments.
The entry to the table is made with the index number of the desired mask; In our case, the figure to be represented on the display.

The access to the individual rows of the table is done through the manipulation of the ***Program Counter***, to which is added the index that points to the desired row.
If we refer to the table above, it has 10 **retlw** lines, each of which corresponds to a digit from 0 to 9. Essentially, we "enter" with a value, such as 0, and come out with the mask **`b'00111111'`** in W, which we will use to command the segments needed to present that digit, and so on.
The mechanism by which this happens is not immediate, but it is not impossible to understand.
In Baselines, the ***Program Counter*** is a 12-bit register that contains the address of the instruction to be executed. On Reset, it is reset to zero, so that the execution starts from location 0. With each instruction, it is incremented by 1 (or two, depending on the opcode) and automatically marks the sequence of lines during execution.
We have already seen, however, that the ***PC*** can be manipulated by some instructions (**`goto`**, **`call`**, **`return`**, **`retlw`**) that modify its contents to make jumps, subroutine calls and return.



These actions are also automatisms of the internal management logic, but it is also possible to manually act on the value of the ***Program Counter***, through the part that is accessible from it, i.e. the **PCL** (***Program Counter Low***) register.

This register contains bits 0 to 7, while the remaining bits 8 to 11 are located in the **PCH** register which is not directly

Let's look at the table: it starts with the rows:

```
        andlw  0x0F          ; Delete Nibble High
        addwf   PCL,f        ; Program Counter
```

The first instruction is an **AND** bit wise that zeroes the high nibble of the byte contained in **W**. This content is the input "number" to the table and the AND is intended to limit it to only 16 combinations, leaving intact only the lowest 4 bits (***nibble***) (bit3-0) and zeroing the high nibble (bit7-4).
The second line is an arithmetic sum statement.

ADDWF

addwf stands for ***Add W** with **file***, which means that it sums the contents of **W** with that of the file in question. The syntax is the usual:

[label]	sp	addwf	, f/w	sp	object	sp	; Comment opt.

The instruction is of the "tailed" kind that we have already encountered, in that you can specify whether the result of the sum should be stored in W or in the file.

In the instruction sets of PICs there are also other sum opcodes, such as **addlw** is an acronym for ***Add literal with W***, i.e. it sums the content of **W** with that of the number in question to the statement. However, this opcode is not available in the Baseline set, but only in the PIC set of higher families.
Executing the sum statement alters some of the flags of the **STATUS,** including:

- **C – Carry** which becomes 1 if the sum exceeds FFh (255 decimal) For example:
  - **W=F0h** and **f=10h** : result **01h** with **C=1**
  - **W=F0h** and **f=02h** : result **F2h** with **C=0**

- **Z – Zero**, which becomes 1 if the sum yields a null result
  For example:
  - **W=FFh** and **f=01h** : result **00h** with **Z=1** (and C=0)
  - **W=F0h** and **f=22h** : result **12h** with **Z=0** (and C=1)

Flag analysis allows you to build algorithms for mathematical operations.

In our case, we add the "input number" to the table with the **PCL register** and this modifies the ***Program Counter***, i.e., it alters the address that corresponds to the next line to be executed.

Let's take a few examples: if the input number to the table is 0, the sequence of instructions is as follows:

```
; with W = 0
segtbl
      andlw 0x0F          ; W contains 0, so 00 & 0F = 00
      addwf PCL,f         ; PCL = PCL + W = PCL + 0 = PCL
      retlw b'00111111'   ; "0" -|-|F|E|D|C|B|A
```

Since the ***PC*** does not change by adding 0 to it, it points to the next line, which is executed. The result is a retlw "subroutine return" with the value `b'00111111'` in **W**.
If the input number is 1:

```
; with  W = 1
segtbl
      andlw 0x0F          ; W contains 1, so 01 & 0F = 01
      addwf PCL,f         ; PCL = PCL + W = PCL + 1
      retlw b'00111111'   ; "0" -|-|F|E|D|C|B|A
      retlw b'00000110'   ; "1" -|-|-|-|-|C|B|-
```

The ***PC*** has been modified and now does not point to the next line, but advances one and executes the next line, resulting in a "subroutine return" with the value `b'00000110'` in **W**.
And so on for all other values.

It should be noted that we have said that a "return from subroutine" results in it. So the table needs to be called as a subroutine, i.e. with a `call`:

```
      movlw indice        ; load in W the index for access to the
table
      Call  segtbl        ; Access the table
      Goto  Exec          ; Transfer Mask to I/O
```

After all, if you think about it, it's not too complex...

Limitazioni delle Lookup Table nei Baseline

In Baselines, due to limitations due to internal mechanisms, as we have already mentioned, calls to subroutines also have limitations.

In particular, when an instruction modifies the **PCL register**, bit 8 of the ***Program Counter*** is

Reset. This means that your ***PC*** will point within the first 256 bytes of the memory page it's on. It will be possible to switch to a different page by switching the **PA0 control bit** of the **STATUS** or with the **Assembler pageel, but, in any case, the table must be entirely in the first 256 bytes of the page.**
While there are tricks for subroutines, as mentioned in a previous tutorial, to overcome this limit, nothing can be done for retlw tables: you have to bend to the limitation.

The solution to adopt is to safely place the table at the top of the page. For example, at the top of page 1 (16F505 and 16F526 have their memory split into two pages):

```
; table area in page 1
    CODE    0x200

; 7 segment data table
segtbl
      andlw 0x0F          ; Low nibble only
      ....
```

To access the table on page 1, starting from page 0, which is the default to the POR, you will need to use the page switch.

Moreover, if it is clear that the tables cannot in any case exceed 256 elements - since the **retlw object** must be 8 bits, therefore between 0 and 255 (00h-FFh) - it should be remembered that the program memory of the Baselines is not very extensive and the use of large tables is possible only after a careful evaluation of the space needed for the other instructions.

Variations on the table

We have said that the AND limits the number of numbers entering the table to 16; We can then extend it to 16 lines, including all the digits of the hexadecimal count:

```
      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
```

It should be clear that the possibilities of the ratio of the input index number to the output mask are vast:

- The input index is normally linked to the symbol to be displayed, number or letter
- the output mask, made to adapt to the hardware used, controls the I/O For example, other

symbols that the 7-segment display can present:

```
      retlw b'01110110' ; "H" -|G|F|E|-|C|B|-
      retlw b'00000110' ; "P" -|-|-|-|-|C|B|-
      retlw b'00111000' ; "L" -|-|F|E|D|-|-|-
      retlw b'01000000' ; "-" -|G|-|-|-|-|-|-
      retlw b'00000000' ; " " -|-|-|-|-|-|-|-
      retlw b'00111000' ; "u" -|-|-|E|D|C|-|-
      retlw b'01100011' ; "°" -|G|F|-|-|-|B|A
```

If we had a command for a common anode display, where this is connected to the Vdd and the LEDs are lit at a low level on the I/O, just invert the contents of the table:

```
      retlw b'10001001' ; "H" -|G|F|E|-|C|B|-
      retlw b'10000110' ; "P" -|-|-|-|-|C|B|-
      retlw b'10000111' ; "L" -|-|F|E|D|-|-|-
      retlw b'10111111' ; "-" -|G|-|-|-|-|-|-
      retlw b'11111111' ; " " -|-|-|-|-|-|-|-
      retlw b'10111000' ; "u" -|-|-|E|D|C|-|-
      retlw b'10011100' ; "°" -|G|F|-|-|-|B|A
```

But, better than rewriting it, is to use an extra statement:

```
      movlw indice      ; load in W the index for access to the
table
      call  segtbl      ; Access the table
      xorlw 0xFF        ; invert value in W
      goto  exec        ; Transfer Mask to I/O
```

where the **XOR** with 1 reverses the state of the bit and then the **XOR with FFh** reverses all the bits of the **W register**. The possibilities of Lookup Tables are obviously not limited to this application and we will have the opportunity to see others during this course.

DT : A Different Way to Insert Tables

There is an MPASM compiler command that allows you to quickly define `retlw tables` like the ones you just saw. These are the dt statements. For example, the first table presented might look like this:

```
; segment data table - display Common cathode
;
segtbl:
      andlw 0x0F          ; Low nibble only
      addwf pcl,f

dt  0x3F, 6, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 7, 0x7F,
```

Binary values have been written in hexadecimal to limit the use of space on the line.

DT

The **DT** (Data Table) statement allows you to create **retlw** tables The syntax is simple

DT	sp	expr1,expr2, ...	sp	[; comment]

Basically, using the data in question, it compiles it as a set of **retlw** statements, an instruction for each element.
These can be values or expressions, which have a maximum width of 8 bits. This directive can be used in absolute or relocatable codes.
In general, **DT** is mainly recommended for PIC12/16, since PIC18 have characteristic instructions for accessing tables (**TBLRD / TBLWT**) and do not expressly require **retlw lists**.
The directive should not be confused with **DTM**, which has similar functions, but is used only in Enhanced Midrange and creates **movlw** lists instead **of retlw** lists.

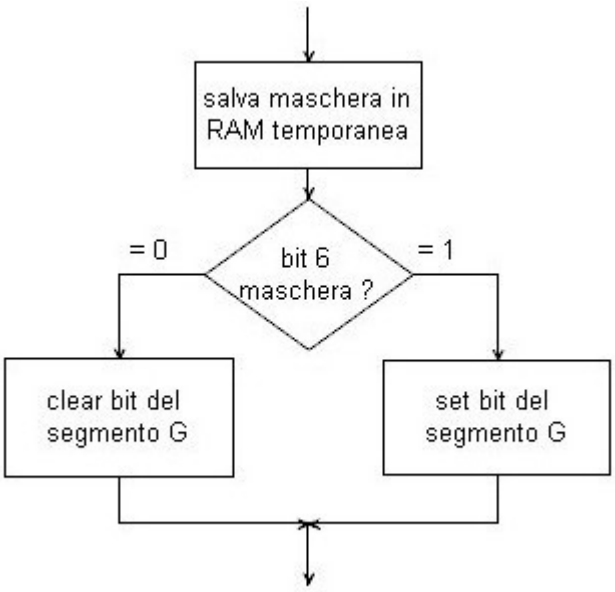
As usual, you can use either the lower case form dt or the upper case **form DT.**

Pass the table result to I/O

At the exit of the table we have in W a mask of 8 bits of which 6 can be passed directly to **PORTC**:

```
; switch the counter to the I/O
setout  movwf   d1          ; Save Mask
        movwf   PORTC       ; Segments A-F
```

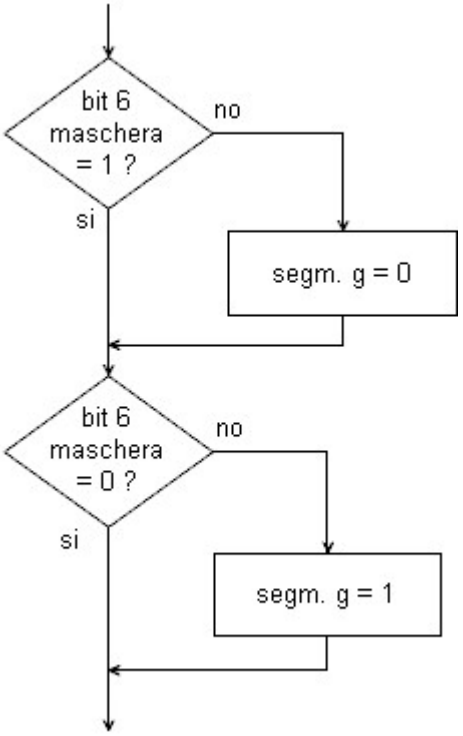
This acts on the **RC5-0** bits and thus on the ***a-f segments***.



Remember that, even if the number in W is 8 bits, writing to **PORTC** which is only 6 bits is perfectly adequate: the unimplemented bits will be overlooked and will not affect in the slightest.

It is now a matter of adapting the g-segment that is controlled by **RB4**.

An easy way to do this is described by the flow chart on the side.



From the point of view of instructions, a form that makes the most of the available opcodes is the one described in the diagram on the side.

Transformed into instructions, it becomes:

```
; adjusts the g-segment on RB4
   movwf   D1      ; save W in temp
   BTFSS   d1.6    ; G-segment?
    Bcf    segmg ; No - Off
   BTFSC   d1.6
    Bsf    segmg ; on
```

always with the use of a temporary memory location to save W during the various steps.

The btfss `and` btfsc `opcode clearance` allows you to choose the correct value for the pin that controls the segment

The time routine is the one already widely seen.

The dp

Since we have enough pins, we can also use the decimal point of the display. Bit 7 of the mask controls it through the state of bit RB5, by varying which the relative LED can be lit.

We can command it in this way: once the count has reached the digit 9, it starts again from 0, but the dp is turned on in order to indicate that the overflow has occurred.

We graft this action into the instructions that limit the count to Fh.

```
        incf    counter,f   ; counter = counter + 1
        movlw   0x10        ; > 0Fh?
        subwf   counter,w
         Bnc    wait        ; If <, wait
```

The counter comparison operation is carried out with the instruction , after loading in W the number to be compared with (10h = Fh+1). If the counter has exceeded Fh, it is reset and the count starts again, while the decimal point is lit.

```
; If >, turn on DP
overF   clrf    counter
        bsf     segmdp      ; dp on
```

After that, you add the waiting time and start the cycle all over again.

```
wait    movlw   LEDtime     ; waiting
        call    Delay10msW

        MOVF    counter,w   ; Retrieve Counter
        Goto    countloop   ; Other Count
```

Note the need to reload the counter in W, which serves as a pointer for the segment table.

The reset button restarts the cycle from the beginning.

The program

It is presented both in ordinary form (***5A_526***) and in modular form (***5Aw_526***), with the related projects.
We remind you that the compilation, in the first case, is carried out with the ***Build key***, in the second with ***Build All***.

The absolute version should need no further comment.

As far as the modular version is concerned, we use the already seen time routine in a suitable form and insert it as an external element:

```
        EXTERN    Delay10ms_W
```

The **CODE** Directive replaces **ORG:**

```
RESVEC   CODE    0x00
```

While memory assignment is done with the UDATA directive:

```
        UDATA
counter     res 1         ; counter
temp        res 1         ; temporaneo
```

We use the indirect resubmission of subroutine vectors, due to the limitations already described in Baselines:

```
; Reset Vector
RESVEC   CODE   0x00

; Internal Oscillator Calibration
        movwf   OSCCAL
        pagesel start
        goto    start
Dly10w:
        pagesel  Delay10ms_W
        goto     Delay10ms_W
Segmtbl:
        pagesel  segtbl
        goto     segtbl
```

A possible hitch

It should be noted that the compilation of the modular source that uses the Linker for the addition of external elements, under some conditions can have some oddities that it is worth mentioning. Here we are talking about the writing of the labels:

```
Dly10w  pagesel  Delay10ms_W
        goto     Delay10ms_W

Segmtbl pagesel  segtbl
        goto     segtbl
```

Although it is completely regular, it has generated error messages:

```
Dly10w  pagesel  Delay10ms_W
        goto     Delay10ms_W

Segmtbl pagesel  segtbl
        goto     segtbl
```

while the form indicated above, where the label is the only element of the line, has been filled in correctly. It should be noted that other labels along the same lines as the opcode did not create problems. The problem is relatively well known.

As a result, in general, where the indicated error appears, rather than looking for other possible causes, the first correction to be made is to separate the label on a different line.

---

20 pins

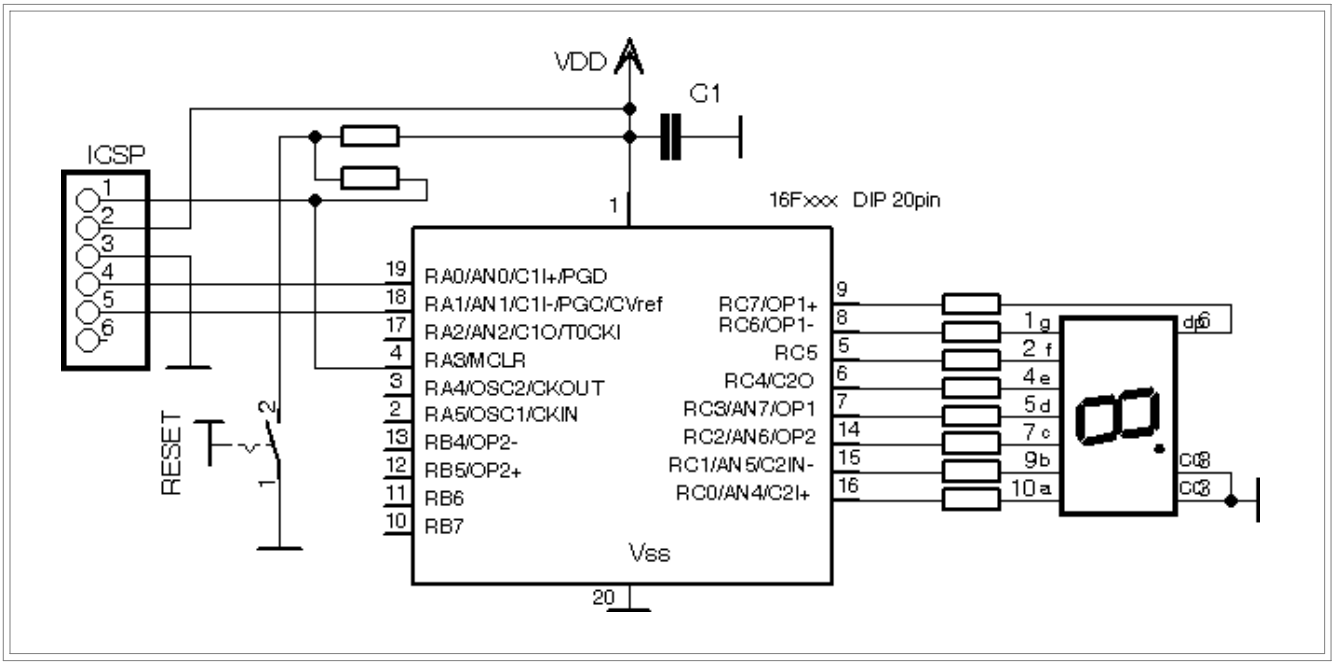
Let's take a little excursus to see the relationship between the number of pins, the possibility of managing peripherals and the program.

With a 20-pin chip, you have a larger number of I/O, organized in three ports, among other things:

DOOR			RA5	RA4	RA3	RA2	RA1	RA0
PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0
PORTC	RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0

Note that there are two full 8bit ports.
This microcontroller certainly occupies a larger surface area than those seen so far, but it is evident that the greater number of pins offers undoubted advantages when we have many things to connect to the micro. For example, here, we can connect the display to the **PORTC**, for segment and decimal point control, still having as many as 14 free I/O.

In Baselines, only **PIC16F527** has this feature:



But the advantage is not only in the number of pins available, but also in the simplification of the display management software: having a full 8-bit port available, the transition from the byte coming out of the table is immediate, not having to depend on pins of another port; All you need to do is transfer the data as it is, without any further processing.

```
; switch the counter to the I/O
setout  movwf  PORTC        ; segments a-g
```

It should be noted that, since **16F527** has many more function modules, it takes a greater number of operations to disable them and arrive at the use of pins as simple digital I/O.

However, **16F527** is a strange Baseline that differs a bit from the other members of the family in characteristics not otherwise present:

- While it has the 12-bit instruction set, it has three more
- has the ability to handle interrupts
- a 4-layer stack and resources that are not common in the family, such as ADCs, comparators, and even op-amps, a very unusual peripheral.

Basically, a component uncertain whether to be a Baseline or a Midrange. The cost is very limited (<1€) as for all Baselines and can, therefore, be considered in many applications.
on the other hand, it can be managed by Pickit3, but, unfortunately, only in MPLAB-X environment, while it is not possible from MPLAB-IDE.
So let's leave this chip for future analysis.

5A_526.asm

```
;*****************************************************************

;-----------------------------------------------------------------
;
;    Title          : Assembly & C Course - Tutorial 5A_526
;                     Counter with 7-segment display output.
;                     The figure presented on the
;                     display every 500ms. Once the overflow is reached,
;                     lit the decimal point.
;    PIC            : 16F526 or 16F505
;    Support        : MPASM
;    Version        : V.519-1.0
;    Date           : 01-05-2013
;    Hardware ref.  :
;    Author         :Afg
;
;-----------------------------------------------------------------
;
;  Pin use :
;  ________________
;       16F505/526 @ 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   segment  Dp
;    RB4/OSC2/CLKOUT           3: Out   segment  g
;    RB3/! MCLR/VPP            4: MCLR
;    RC5/T0CKI                 5:
;    RC4[/C2OUT]               6:
;    RC3                       7:
;    RC2[/CVref]               8: Out   segment  f
;    RC1[/C2IN-]               9: Out   segment  an
                                          d
;    RC0[/C2IN+]              10: Out   segment  d
;    RB2[/C1OUT/AN2]          11: Out   segment  c
;    RB1[/C1IN-/AN1]/ICSPC    12: Out   segment  b
;    RB0[/C1IN+/AN0]/ICSPD    13: Out   segment  at
;    Vss                      14: --
;
;   []16F526 only
; ################################################################
; Choice of #ifdef
  processor_16F526
         LIST      p=16F526         ; Processor Definition
         #include <p16F526.inc>
  #endif
  #ifdef____16F505
```

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

        Radix     DEC

; ##########################################################################
;                            CONFIGURATION
  #ifdef____16F526
; Internal Oscillator, 4MHz, No WDT, No CP, RB3=MCLR
 __config _IntRC_OSC_RB4 & _IOSCFS_4MHz & _WDTE_OFF & _CP_OFF &
_CPDF_OFF & _MCLRE_ON
  #endif
  #ifdef____16F505
; Internal oscillator, no WDT, no CP, MCLR
  __config _IntRC_OSC & _WDT_OFF & _CP_OFF & _MCLRE_ON
  #endif


; ##########################################################################
;                             RAM
; general purpose RAM
        CBLOCK 0x10          ; Start RAM
      counter area           ; Counter
      d1,d2,d3               ; ENDC Delay Counters

 ;*****************************************************************
;=================================================================
;            DEFINITION OF PORT USE
;
;P ORTC map
;| 5 | 4 | 3 | 2 | 1 | 0 |
;|-----|-----|-----|-----|-----|-----|
;| f | and | d | c | b | a |
;
#define segma PORTC,0 ; segment a #define
segmb PORTC,1 ; segment b #define segmc
PORTC,2 ; segment c #define segmd PORTC,3
;  segment  d  #define  segme  PORTC,4  ;
segment e #define segmf PORTC,5 ; Segment
F

;P ORTB map
;| 5 | 4 | 3 | 2 | 1 | 0 |
;|-----|-----|-----|-----|-----|-----|
;| dp | g |        |     |     |     |
;
;#define            PORTB,0    ;
;#define            PORTB,1    ;
;#define            PORTB,2    ;
;#define            PORTB,3    ; MCLR
#define      segmg PORTB,4     ; Segment E
#define     segmdp PORTB,5     ; Segment F
;
;*****************************************************************
;
```

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

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

 ; MOWF Internal Oscillator
        Calibration OSCCAL

; ########################################################################
;                         MAIN PROGRAM
Main:
; Reset Initializations
  #ifdef____16F526
; Disable CLRF Analog Inputs
                ADCON0

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

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

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

        CLRF    PORTB       ; Clear Latch of CLRF
        Ports   PORTC
        movlw   0           ; reset counter and
        W movwf Counter

Countloop:
        Call    segtbl      ; Convert number->mask 7 segm.

; switch the counter to the I/O
MovWF Setout    D1          ; Save MovWF
        Mask    PORTC       ; Segments A-F
```

```
        BTFSS   d1.6        ; G-segment?
         BCF    Segmg       ; off
        BTFSC   D1.6
         BSF    segmg       ; on

        incf    counter,f   ; counter = counter + 1
        movlw   0x10        ; > 0Fh?
        subwf   counter,w
         Bnc    wait        ; If <, wait
; if >, Light   Dp
overF   CLRF    Counter
        BSF     SEGMDP      ; dp on

wait    movlw   LEDtime     ; waiting
        call    Delay10msW

        MOVF    counter,w   ; Retrieve Other
        Goto    countloop   ; Count Counter


; 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|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|-|-|-|At

; ####################################################################
;                         SUBROUTINES

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

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