



---

## STATE MACHINE PART2

---

PIC®, MPLAB®, PICKit3® and ICD3® are registered trademarks of Microchip Technology Inc®.  
Proton Development Suite® or PDS® are a registered trademark of Crownhill Associates Limited©.

The Project has been developed and written by Alberto Freixanet.  
The document has been edited by John Drew.

### Introduction:

I present an application to implement the State Machine system discussed in part 1 of the tutorial, with a template written in Proton Basic language. This template is adapted for the compiler for the PIC18F25K20 of the Amicus18 board and easily adapted to other PICs®.

Remember the principles of this State Machine:

- 1- The State Machine is a large loop that can never stop.
- 2- You cannot use the 'GoTo' command to a destination that is outside the main code area of a module.

### SOME DEFINITIONS AND CONVENTIONS

The following describes the conventions I use in the compiler. The programs are intended to improve understanding of the code.

#### State Machine Commands

In addition to simplifying writing, these will improve the portability of our programs. The commands of the library are highlighted in **bold** in the IDE and end with double parentheses to assist recognition:

**StateInit()**, **StateInitEnd**(SyncOn), **StartDelayMS**(Value), **StateOut()**, **StateOutEnd()**, **ReturnSM**(Rson), **NextState**(Value), **IncState()**, **DecState()**, **SM\_State()**, **NextStateReturn**(StateX,StateY), **NextStateDelay**(Delay,NewState), **StartDelayMS**(Delay), **InitStateMachine()**, etc...

#### Global and Local Variables

Global variables are those that are available throughout a program. Alteration of one in a routine or function will result in the new value being available everywhere else in a program. Local variables are those that are only available within the procedure or function within which they are defined. Once outside the subroutine they are destroyed and no longer available. Languages such as Pascal and C use this approach.

Variables in programming languages are simply containers that contain values. Access to the variables and the content of them is done through their names, also known as their identifiers.

Variables are typed and created either at the beginning of a program or when first used. Variables are filled with values generally as a result of a statement.

## Local Variables

PDS does not support local variables. Even within the P24 version the 'local variables' are not destroyed and may be accessed outside the routine using a simple strategy. Local variables and parameters for procedures and functions use a significant amount of the RAM and stack. 8 bit PIC micros have small amounts of RAM and stack, so implementation of local variables is not always practical.

Some BASIC compilers support local variables in PIC micros but invariably their compiled programs are larger and less efficient.

The variables located in bank A, whose access is the fastest, will be used in my examples. They are called System Variables and must be used efficiently because their quantity is very limited.

As the compiler does not handle local variables I constructed a trick that I have been using for years and that works well. This system is used in all my libraries.

I have been able to determine that the best solution for me is to use blocks of 4 system Byte variables, 2 system Word variables or 1 system Dword variable declared as needed.

```
$ifndef _LVBS_Block1_
```

```
    $define _LVBS_Block1_
```

```
    Dim LVByte1 As Byte System ' Local Variable System Byte1
```

```
    Dim LVByte2 As Byte System ' Local Variable System Byte2
```

```
    Dim LVByte3 As Byte System ' Local Variable System Byte3
```

```
    Dim LVByte4 As Byte System ' Local Variable System Byte4
```

```
$endif
```

**LVByte1 => Local Variable System Byte number 1**

For a second block of local Bytes you can write.

```
$ifndef _LVBS_Block2_
```

```
    $define _LVBS_Block2_
```

```
    Dim LVByte5 As Byte System ' Local Variable System Byte5
```

```
    Dim LVByte6 As Byte System ' Local Variable System Byte6
```

```
    Dim LVByte7 As Byte System ' Local Variable System Byte7
```

```
    Dim LVByte8 As Byte System ' Local Variable System Byte8
```

```
$endif
```

## What does it mean?

The name of this block is **\_LVBS\_Block1\_**. If this block doesn't exist before (**\$ifndef**) then the following bytes are declared. If this block is written in several libraries then the compiler will take the first declared block for the variables without a compilation error.

Then in a module you can declare these local variables like this.

```
Dim ADC_Result As LVSByte1
Dim CounterValue As LVSByte2
Dim Var1 As LVSByte3
Dim UARTData As LVSByte4
```

For a Word local variable it will be written. (always 4 Bytes System)

```
$ifndef _LVWS_Block1_
    $define _LVWS_Block1_
        Dim LVSWord1 As Word System ' Local Variable System Word1
        Dim LVSWord2 As Word System ' Local Variable System Word2
```

```
$endif
```

```
Dim ADC_Result As LVSWord1
```

For a Dword local variable it will be written. (always 4 Bytes System)

```
$ifndef _LVDWS_Block1_
    $define _LVDWS_Block1_
        Dim LVSWord1 As Dword System ' Local Variable System Dword1
```

```
$endif
```

For a local Bit it will be written.

```
$ifndef _LVSBits_Block1_
    $define _LVSBits_Block1_
        Dim LVSBitsA As Byte System ' Local Variable System Byte
```

```
$endif
```

Then declare the Local Bits.

```
Dim F_UART_Error As LVSBitsA.0
Dim F_ADC_Error As LVSBitsA.1
```

Etc...

## User Flags

Some flags will also be needed to perform some routines. So I can differentiate a variable from a flag only by having an "F" in front: **F\_PushButton** other than **Pushbuttons** (the "Pushbuttons" variable could have several bits representing other pushbuttons).

## STRUCTURE OF THE TEMPLATE

A template already designed according to the standards of the compiler where everything is in its place will help the beginner in the realisation of its program.

### Declare the device:

**Device = 18F25K20**

### Compiler declares: (Only what is necessary)

**Declare Eeprom\_Address** = \$F00000 *'Not necessary if standard value.'*

**Declare Optimiser\_Level** = 2

**Declare Dead\_Code\_Remove** = On

**Declare Create\_Coff** = On

**Declare Watchdog** = On

*'Declare Float\_Display\_Type = Fast*

*'Declare Float\_Rounding = Off*

**Declare Bootloader** = Off

**Declare Show\_System\_Variables** = On

**Declare MemWrite\_Int\_Control** = 1

**\$if** 65537 < \_code

**Declare Access\_Upper\_64K** = On

**\$endif**

### Declare the Config Fuses:

**Config\_Start**

Etc...

**Config\_End**

### Declare the LCD: (Only if necessary)

**Declare LCD\_DTPin** = PORTB.4 *' LCD's Data lines (D4 to D7)*

**Declare LCD\_ENPin** = PORTB.3 *' LCD's EN line*

**Declare LCD\_RSPin** = PORTB.2 *' LCD's RS line*

**Declare LCD\_Interface** = 4 *' 4-bit interface to LCD*

**Declare LCD\_Lines** = 2 *' LCD contains 2 lines*

**Declare LCD\_DataUs** = 50 *' Time to wait after print a data To LCD*

**Declare LCD\_CommandUs** = 2000 *' Time to wait after print a command to LCD*

**Declare LCD\_Type** = Alphanumeric *' LCD type is alphanumeric*

**Symbol** \_\_LCD\_CharsLenght = 16 *' For the AMI18 LCD Shield*

### Declare the local variables:

**\$ifndef** \_SMLVBS\_Block1\_

**\$define** \_SMLVBS\_Block1\_

```

Dim SMLVSByte1 As Byte System ' Local Variable System Byte1
Dim SMLVSByte2 As Byte System ' Local Variable System Byte2
Dim SMLVSByte3 As Byte System ' Local Variable System Byte3
Dim SMLVSByte4 As Byte System ' Local Variable System Byte4

```

**\$endif**

The local variables for the main program (SM) have a different name so as not to destroy the local variables of the library.

Declare the local variables for the user:

Declare the constants for the user:

Declare the language for the strings of the project and libraries: (one only)

```

$define _English_
' $define _Spanish_
' $define _French_

```

The language is declared at compiler time.

All strings are written with a label at the end of the program in a file, in the upper part of the ROM. In order to access these strings a special command of the library is used:

**HRSLStrg**(Label Name, Number of Chars to Send, Number of CR to send)

As you can see, this command is very powerful.

There is the same command for the **LCD**.

**PrintStrg**(Label Name, Line Number, Position, Number of Characters)

**PrintStrg**(TXT0, 1, 1, **AllChars**) ' "STATES MACHINE"

Or simpler.

**PrintStrg**(TXT0, 1, 1) ' "STATES MACHINE"

Declare the dummy Array: (if there is an array(s) in your main code or library)

An array placed between BankA and Bank0 could cause write/read errors. For this reason an array of adjustable dimensions is built to occupy the upper part of BankA. It will not affect the System variables that will always be placed in BankA.

A Dummy Byte is used to show the starting point of the new array. Once the program is compiled the the position of the dummy byte is known. The value is set to: (for example)

**\$define AddressofMyDummyByte** 52

It is recompiled to position the new array.

Bits definition file of the Device: (not mandatory)

For this project I have written the following file.

Include "SYS\_18FXXK20.inc"

Many PIC® SFR (Special Function Register) configuration records can be read and / or written by the user. The more peripherals the PIC® has, the more registers it will have to configure. The use of the bits of the special registers are described in the manual of each PIC®. For example, to configure the read or write in the EEPROM memory there is the bit EEPGD that determines if you want access to the EEPROM or FLASH memory according to what is written (0 or 1). In the case of Pic18F25K20 it is defined as EECON1.EEPGD.

To avoid searching and/or writing the corresponding register I only name the bit of the function that interests me. I always forget which register it belongs to. Being a bit element of a Byte variable, I can define it as a flag. For example the EECON1 bit. EEPGD is always F\_EEPGD. But the compiler needs to know where this definition comes from, what record it is involved in. For that I have written some definitions for each Microchip microcontroller I use, for example for our case SYS\_18FXXK20.inc. By putting this file in the beginning of the program the compiler will search for the appropriate register. I did the hard work only once. Also consulting this file, you can know the functions of each bit of a particular record. This file is an adaptation of the internal registers defined in the PIC® manual.

Sometimes in new PICs®, Microchip® changes the position and name of the source register. With this system, when writing a new file, the references to these bits are not changed in the user program. It is a great advantage.

### State Variable Number:

It is better to **Define** every **State** with a Name.

In this way, each function of the different modules is recognised and if one or several modules are changed in the editor, it will not be necessary to modify all the values of the program's State Variables.

```
$define SM_Init 0           ' STATE00
$define SM_ButtonsOff 7     ' STATE07
$define SM_ReadButtons 8   ' STATE08
$define SM_LED0 9          ' STATE09
$define SM_LED1 10         ' STATE10
$define SM_LED2 11         ' STATE11
$define SM_LED3 12         ' STATE12
$define SM_LED4 13         ' STATE13
$define SM_LED5 14         ' STATE14
$define SM_LED6 15         ' STATE15
$define SM_LED7 16         ' STATE16
```

Same structure for the State Machine 2.

## USER FUNCTIONS (only Meta Macro, no Subroutine)

With meta-macros written with the pre-processor you can configure some time functions (not explained yet) with different timers. This code should be very short because these lines are executed before each module. Avoid writing subroutines.

### Enable the Library Commands included the STMachine00.Inc file

As I write the libraries, it does not occupy code space in the ROM if they are not called in the main program. But in some cases, these libraries need many local or/and different variables. Then many System variables would be consumed for no purpose. To avoid this problem they are disabled.

### Declare the Serial Communication for the State Machine Library

**\$define** \_EnableTerminal\_

This definition allows you to use the **HRSOut** command in the library codes.

The following line establishes the **Declare** for Serial Communication.

**\$define** \_HSerialBaud\_ 38400

It is not usual to use this form, but this allows me to manually modify the settings of each speed in my way.

### Choose the HRSOut command for the SM Library. (One only)

```
$define _HRSOut HRSOut
' $define _HRSOut HRSOut1
' $define _HRSOut HRSOut2
' $define _HRSOut HRSOut3
' $define _HRSOut HRSOut4
' $define _HRSOut HRSOut5
```

Depending on the UART number chosen by the user, all the **HRSOut** commands of all libraries are modified without changing any code at compiler time.

### Enable some State Machine Commands

Some functions are already prepared in the State Machine, the interrupt module, and more. Sites: Timers, virtual Delays, State Machine2.

### Include the Library for the States Machine System.

**Include** "STMachine00.inc"

### Insert the High Interrupt Handler & Low Interrupt Handler if necessary

```
'Interrupt Handler.
Declare Reminders = OFF
Interrupt_Handler:
Context Save
'-----
If F_TMR2IF = 1 Then           ' Check if this is a timer2 interrupt?
    $ifdef _Synchronize1mS_
        F_Timer1mS = 1           ' Set a Flag every 1mS
```

```

$endif
F_TMR2IF = 0           ' Clear the Timer2 interrupt flag for next time.
'-----
$if _defined(_Timer10mS_) Or _defined(_Timer1S_)
Inc Timer10mS
If Timer10mS = 10 Then
    Timer10mS = 0      ' 100Hz Timer
    $ifdef _Synchronize10mS_
        F_Timer10mS = 1
    $endif
    $ifdef _Timer1S_
        Inc Timer500mS
        If Timer500mS = 50 Then
            Timer500mS = 0
            Toggle F_Timer500mS  ' 1 Hz Timer, to toggle some leds.
        EndIf
    $endif
    ' Other Timers?
    ' Your code...
EndIf
$endif
'-----
$ifdef _SMVDelay_
' Delay 1mS for the States.
If F_SMVDelay = 1 Then      ' The VDelay is started.
    Dec SMDelayCounter      ' Decrements the SMDelayCounter every 1mS.
    If SMDelayCounter = 0 Then ' Check if delay counter reaches 0.
        F_SMVDelay = 0      ' Delay END, clear the Vdelay FLAG.
    EndIf
EndIf
$endif
'-----
$ifdef _SM2VDelay_
' Delay 1mS for the States2.
If F_SM2VDelay = 1 Then      ' The VDelay is started.
    Dec SM2DelayCounter      ' Decrements the SMDelayCounter every 1mS.
    If SM2DelayCounter = 0 Then ' Check if delay counter reaches 0.
        F_SM2VDelay = 0      ' Delay END, clear the Vdelay FLAG.
    EndIf
EndIf
$endif
'-----
$ifdef _Delay1_
' Count every 1mS.
If F_Delay1 = 1 Then          ' The Delay is started.
    Dec Delay1Counter          ' Decrements the DelayCounter every 1mS.
    If Delay1Counter = 0 Then  ' Check if delay counter reaches 0.
        F_Delay1 = 0          ' Delay End, clear the delay FLAG.
    EndIf
EndIf
$endif
'-----
$ifdef _Delay2_
' Count every 1mS.
If F_Delay2 = 1 Then          ' The Delay is started.
    Dec Delay2Counter          ' Decrements the DelayCounter every 1mS.
    If Delay2Counter = 0 Then  ' Check if delay counter reaches 0.
        F_Delay2 = 0          ' Delay End, clear the delay FLAG.
    EndIf
EndIf
$endif
'-----
EndIf
'-----
' Here the another user interrupt handler code....
'.....
'.....
'.....

```



```
'-----  
END_INT:  
'-----
```

```
Context Restore      ' Exit the interrupt handler and restore variables  
Declare Reminders = On
```

The **timer2** with **PR2** are used to generate an interrupt every **1mS**. It is the best system because you do not have to recharge the Timers inside the interrupt handler; which improves accuracy and simplifies code.

## EEPROM Data

You write here the data or strings that you want to write in the EEPROM memory.

## Library files

Insert the library files here.

## The Main Code: STMAIN:

Here initialize all variables, PORTs, etc ... of the PIC©. Initialize the State Machine.

## STATE MACHINE STARTS

The states machine really starts here.

**StateMachineInit:** It is the label to reset the State Machine.

**StateMachineStart:** It is the input of the State Machine. Each cycle starts here.

## User Functions:

The functions of the user could be activated or not by the pre-processor. This code must be related to a type of timing and should be very short.

## Running the Module Delay

If the virtual delay (**SMVDelay**) has been activated by its flag then the State Machine is derived and does nothing. No principal module is executed. But the **State Machine2** continues to run anyway. This system allows running some programs in the background when the main loop is waiting.

## State Machine Scheduler

Before arriving at the main scheduler, the value of the state variable of the second State Machine is checked. If this value is greater than 0 then the path forks to a second SM that we will study later.

The State Machine scheduler is responsible for making the jump to the corresponding module. When a supplementary program module is added, the new label must be added to the list.

```
' Update the LABELS when you add a new States Machine Module.  
BranchL StateIndex, [STATE00, STATE01, STATE02, STATE03, STATE04, STATE05, STATE06, _  
STATE07, STATE08]
```

The "**BranchL**" compiler command allows you to have control over a range error. If the value of the variable "**StateIndex**" does not correspond to a **LABEL** then it is skipped in this direction. In this case the programmer must write an error code and reset the system.

Here begins the users' code modules addressed by a structure that will ease the programmer's life.

[illegible]

Subroutines could be written after the Module.

### The Module header:

```
*****
'* Name      : STATE17 (SM_Toggle_LED5)
'* Purpose   : Toggle LED5 at 0,5 Hz
'* Input     : None
'* Output    : None
'* Notes     :
*****
```

The data that is written in the header is important as it informs us about the purpose of the code. This is very useful for maintenance reasons.

### Declaration of variables:

```
' Define the Global variables used in this state:
Dim ReturnACK As Byte
' Define the Local variables used in this state:
Dim KEYn      As SMLVSByte1
Dim KEY_Result As SMLVSByte2
Dim KEYnOld   As SMLVSByte3
Dim KEYCounter As SMLVSByte4
' Define the Symbols used:
Symbol BacklightTimeValue = 5 ' Backlight Set for 5 sec
```

It is not usual to do it this way but this system allows the user to control directly and quickly the variables used in the particular module.

This system will only work if the code that will use these variables/constants is in the descending lines. If these variables/constants are used in other higher modules then they have to be written at the beginning of the file as usual whereas local variables will always remain here.

### Entry of the Module: LABEL:

```
'=====
STATE17:
'=====
```

The Scheduler by a "GoTo" command will arrive at the module label.

### StateInit() Transitional state.

```
StateInit()
'##-INIT SECTION-#####
```

This command is a **switch** that lets you run the code underneath once and copy the StateIndex variable for future use.

### StateInitEnd()

```
'##-END INIT-#####
StateInitEnd()
```

This command closes the start section.

### Main Code: Static state.

```
':::-MAIN CODE-:::.....
```

```

' Write to the Config register: Enable signal output with frequency = 1 Hz.
DS1307_WriteControl(%10010000, ReturnACK)
If ReturnACK = 1 Then
    HRSLStrg(TXT16,1) ' "NACK DS1307 Write Control",1CR
    NextStateDelay(1000,SM_Error_NACK) ' Waiting 1000mS before to go to
    ' SM_Error_NACK State.
Else
    NextState(SM_ReadTime)
EndIf

```

In the Main section the PDS user can write the code. If the **StateIndex** is not altered then the code works like in an **infinite loop**.

STATE17: **While** 1 = 1 : [Module Code] : **Wend**

**NextStateDelay(1000,SM Error NACK)**

It allows starting a delay in **mS** when the Module is closed by ordering the **SM** to change state by (**SM\_Error\_NACK**), this **Delay** can be started before the execution of another module begins. This Delay is virtual because the PIC program counter does not wait. Codes continue to run without stopping. The Delay is running by interrupt and is another concept.

STATE17: **While** 1 = 1 : [Module Code, Change State =>] : **Wend** : **Delay** Y

**StartDelayMS(50)**

It allows starting a delay in **mS** every time the main program has been executed in a loop. And the result is as follows. This Delay is virtual because the PIC program counter does not wait. Codes continue to run without stopping. The Delay is running by interrupt.

STATE17: **While** 1 = 1 : **Delay**( X ) : [Module Code] : **Wend**

If the value in parenthesis is equal to **0** then the code of the macro is not written, accordingly the Delay does not exist.

**StateOut()** Transitional state.

This command verifies if the **StateIndex** variable has changed, if it is positive, the following code is executed to exit this **Module**. Sometimes you need to reinitialise values or send a message to the LCD for example.

Of course if **StateIndex** has not changed, the **SM** will only execute the main module code again.

## StateOutEnd()

This command closes the values used in this Module, in particular the virtual **Delay** if it has been activated.

## ReturnSM()

Return to the scheduler.

## Subroutines:

Some subroutine(s) corresponding to this module will have to be written. In this case, it will be written after the **ReturnSM ()** command. See an example in the STM00.bas file in STATE08.

## **Module Overview:**

Due to its structure, all the codes included in the Module could be stored in a .bas file as a library. All information is included to copy and paste into another project.

## **Example of Module: (Read a 4x4 Keypad)**

Complex code made simple thanks to **SM**.

```
'*****
'* Name      : STATE08 [SM_ReadKeypad]
'* Purpose   : Read all Buttons with InkeyX(50) with debounce code.
'* Input     : None
'* Output    : None
'* Notes     : The InkeyX(50) returns a "0" if NO Key pressed.
'=====
' Define the Global variables used in this state:
  Dim KEYCounter As Byte
  Dim KEYnOld As Byte
' Define the Local variables used in this state:
  Dim KEYn As SMLVSByte1
  Dim KEY_Result As SMLVSByte2
' Define the Symbols used:
  Symbol DebounceReadings = 4
'=====
STATE08:
'=====
  StateInit()
  '##-INIT SECTION-#####
  HRSLSstrg(TXT58,1)      ' "Waiting a KEY pressed!"
  CLL                     ' Clear the LCD
  PrintStrg(TXT90,2,1)    ' "WAIT KEY pressed!"
  KEYnOld = 0              ' Initialize parameters for debouncing.
  KEYCounter = 0
  NextState2(SM2_PrintLCDLine1) ' Start running the States Machine2 in
                               ' Background.
  '##-END INIT-#####
  StateInitEnd()
':::-MAIN CODE-:::
KEYn = InkeyX(50)          ' Read the Keypad with 50uS delay for filtering.

If KEYn > 0 Then
  If KEYnOld = KEYn Then    ' Some debounce system
    Inc KEYCounter
    If KEYCounter = DebounceReadings Then ' 4 + 1 reading necessary.
      KEYCounter = 0
      'KEYn = 0  1  2  3  4  5  6  7  8  9  10
      '          11 12 13 14 15 16
      KEY_Result = LookUp KEYn, [0, "7", "8", "9", "/", "4", "5", "6", "*", _
                                "1", "2", "3", "-", "C", "0", "=", "+"]

      Select KEY_Result
```

```

Case "0"
    NextState(SM_Set_LED0)
Case "1"
    NextState(SM_Clear_LED0)
Case "2"
    NextState(SM_Set_LED1)
Case "3"
    NextState(SM_Clear_LED1)
Case "4"
    NextState(SM_Set_LED2)
Case "5"
    NextState(SM_Clear_LED2)
Case "6"
    NextState(SM_Set_LED3)
Case "7"
    NextState(SM_Clear_LED3)
Case "8"
    HRSOut CR,LF
    HRSLStrg(TXT77,1)           ' "KEY 8 pressed!"
    NextState(SM_ButtonsOff)
Case "9"
    HRSOut CR,LF
    HRSLStrg(TXT79,1)           ' "KEY 9 pressed!"
    NextState(SM_ButtonsOff)
Case "/"
    HRSOut CR,LF
    HRSLStrg(TXT82,1)           ' "KEY / pressed!"
    NextState(SM_ButtonsOff)
Case "*"
    HRSOut CR,LF
    HRSLStrg(TXT83,1)           ' "KEY * pressed!"
    NextState(SM_ButtonsOff)
Case "-"
    HRSOut CR,LF
    HRSLStrg(TXT84,1)           ' "KEY - pressed!"
    NextState(SM_ButtonsOff)
Case "+"
    HRSOut CR,LF
    HRSLStrg(TXT85,1)           ' "KEY + pressed!"
    NextState(SM_ButtonsOff)
Case "="
    HRSOut CR,LF
    HRSLStrg(TXT89,1)           ' "KEY = pressed!"
    NextState(SM_ButtonsOff)
Case "C"
    NextState(SM_Toggle_LED5)   ' "START Toggle LED5", "STOP
                                ' Toggle LED5"
Case Else
    NextState(SM_ButtonsOff)    ' Come back reading buttons
EndSelect
EndIf
Else
    KEYCounter = 0              ' If another KEY is pressed then clear the counter.
EndIf
KEYnOld = KEYn
Else
    KEYCounter = 0              ' KEY depressed Then Clear the Counter.
EndIf

':::-EXIT MAIN-::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
    StartDelayMS(40)           ' Adjust some sample time 40mS.
'=====

```

## StateOut ()

[illegible]

```
StartBeep()           ' Generate a pulse (Beep) of 0,250 second for every
```

```
        ' Button pressed.
```

```
NextState2(SM2 Init)      ' Stop running the States Machine2 in Background.
```

[illegible]

StateOutEnd ()

### ReturnSM()

## What is this doing?

Some local variable have been declared and a constant for debouncing.

In the start section a message is sent to the terminal.

The LCD is cleared ready for another message.

Some variables are initialised.

A special command is made to start the States Machine2 to perform some background jobs (\*).

In main code, a 4x4 keyboard is read with the modified **InKeyX(y)** command from my library. (EMI compliant)

When reading a pressed key, count **5** readings (debounce) before continuing.

If the key is correct the SM will go to the different modules to handle the management of each function.

A delay is started to read the Keypad every 40 mS to perform the debouncing.

When the Module exits, corresponding to a pressed key, a beep sounds from the buzzer (running in Background).

Stop the function that the State Machine2 was doing in the background.

(\*) The background job prints the string **"STATE MACHINE2A"** to the LCD line 1.

## CONCLUSION

I have tried to explain the structure of my State Machine. We will see in the next tutorial the advanced functions.

# Alberto Freixanet

04 April 2017