



---

## STATE MACHINE PART3

---

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:

We are going to study the different advanced functions that allow us to facilitate the code.  
Please read carefully the pre-processor section in PDS manual.

### THE VIRTUAL DELAY: (SMVDelay)

In a standard Delay command, the program flow is stopped completely until the time is finished. With Virtual Delay the program flow does not stop and continues to execute the code written by the programmer.

Everything starts, as in normal Delay, by loading the Delay value on a counter and setting a delay flag too.

#### **StartDelayMS(40)**

```
$define StartDelayMS (pDelay)      '  
    $ifdef _SMVDelay_              '  
    $if pDelay > 0                  '  
    If F_NextStateVDelay = 0 Then  '  
        If StateIndex = StateIndexOld Then '  
            SMDelayCounter = pDelay      '  
            F_SMVDelay = 1                '  
        EndIf                            '  
    EndIf                              '  
$endif                              '  
$endif
```

This delay will only be activated when a value greater than zero is written. The “F\_NextStateVDelay = 0” condition prevents the loss of a delay that may be set by another priority. The same delay parameters are used for 2 different functions.

STATE17 : **While** 1 = 1 : [Module Code - **StartDelayMS(40)**] : **Wend**

When the delay is activated it goes directly to the beginning of the SM.

Code to bypass the scheduler:

```
$ifdef _SMVDelay_  
If F_SMVDelay = 1 Then  
    GoTo StateMachineStart ' Waiting delay for the States Machine1.  
EndIf  
$endif
```

Before the Scheduler the 'F\_SMVDelay' flag is controlled. In this way the **BranchL** command is not executed until the flag is zero.

Inside the interrupt handler the virtual delay counter is decremented every **mS**.

```
$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
```

### Another Virtual Delay (Priority Delay)

STATE17: **While** 1=1 : [Module Code ] : **Wend** : **VDelay X** : STATE18: **While** 1=1 : [Module Code ] : **Wend**

Often it is necessary to place a delay between 2 states (Modules) for example to give time to see a display. Then a Virtual Delay is introduced. This function is performed with the command:

**NextStateDelay(1000, SM\_LED5)**

The **NextState** command gives the order to go to another module "**SM\_LED5**" but with a delay of **1000** mS. It could be written in other ways.

**NextStateDelay(1000, IncState)**      ' Increment one State.

**NextStateDelay(1000, DecState)**      ' Decrement one State.

As the same delay is used as the previous one it must be done in a way that avoids mutual interference. This is done inside the "**StateOutEnd ()**" command, i.e. when leaving the state. The corresponding macro is:

```
$define NextStateDelay(pDelay,pNewState) '  
    $if pDelay > 0 '  
        SMDelayCounter = pDelay '  
        F_NextStateVDelay = 1 '  
        F_SMVDelay = 1 '  
    $endif '  
    $if pNewState = IncState '  
        Inc StateIndex '  
    $elseif pNewState = DecState '  
        Dec StateIndex '  
    $else '  
        StateIndex = pNewState '  
    $endif
```

## THE USER FUNCTION:

The user function is usually a code that works in the background using timing. Let's look at example #1 of the template.

To perform these functions we will use the interrupts and the State Machine.

**Purpose:** Blink one LED every second.

To perform this function we will need:

- A start command written in a module.
- A stop command written in a module.
- A timer running at 2 Hz by interrupt.
- Apply this Timing function to the LED.

- **A start command.**

The user must write the macros in the top of .bas file and the command will be inserted in the module code.

```
$define StartFunction1() F_UserFunction1 = 1
```

- **A stop command.**

```
$define StopFunction1() F_UserFunction1 = 0 : LED5 = 0
```

- **A timer running at 2 Hz by interrupt.**

```
$if __defined(_Timer10mS_) Or __defined(_Timer1S_)
Inc Timer10mS
If Timer10mS = 10 Then
    Timer10mS = 0                ' 100Hz Timer
    $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
```

Then this flag is created with the Toggle command.

```
Toggle F_Timer500mS
```

- **Apply this Timer function to the LED.**

Before the Scheduler, this code is written in the "User Functions" section.

```
' RUN User Function1: Toggle the LED5 every 1/2 second.
If F_UserFunction1 = 1 Then
    LED5 = F_Timer500mS
EndIf
```

With this indirect construction the flag “**F\_Timer500mS**” could be applied to several outputs.

```
If F_UserFunction3 = 1 Then
    LED9 = F_Timer500mS
EndIf
```

Then this code is executed at every loop of the State Machine, (Very high speed). The LED blinks (500mS = On, 500mS = OFF). See an example in the **STM02.bas** file.

## GENERATE A PULSE:

Another user function has been written to be able to generate a pulse of predetermined duration. This function requires a Virtual Delay2 generated by the interrupt handler. Let's look at example #1 of the template **STM02.bas** file.

**Purpose:** Generate a pulse.

To perform this function we will need:

- A start command written in a module.
- A stop command.
- A Virtual Delay by interrupt. (VDelay2)
- Apply this Timing function to a Buzzer or a LED9 for this case.

### • A start command.

Create a Macro that starts the **Delay2** and set the **Buzzer** or **LED9**. This macro must be written by the user in the top of the .bas file.

```
$define StartBeep() '
F_UserFunction2 = 1 '
LED9 = 1 '
StartDelay2MS(250)
```

The StartBeep () macro will run once only, be careful in its placement, see STM02.bas file in module 8.

### • A stop command.

The final end of the function is performed by Delay itself. As follows, this is the VDelay2 code in the interrupt handler.

```
$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
```

- **Apply the final Delay2 to the function Buzzer.**

Before the Scheduler, this code is written in the "**User Functions**" section.

```
' RUN User Function2: Clear LED9 after a delay2.
If F_UserFunction2 = 1 Then
    If F_Delay2 = 0 Then
        LED9 = 0 ' <= Clear the LED9 when VDelay2 finished.
        F_UserFunction2 = 0
    EndIf
EndIf
```

All user timing functions could have this same structure.

## Conclusion of the User Functions

As you have seen, the code inside the interrupt handler must be generic. The applications are done in the State Machine; but because the inside of an interrupt handler should be brief the user functions are written outside the handler.

The User Functions must use special Virtual Delay only.

## INDIRECT ADDRESSING:

### The Return State Option

This application is complex to perform. You could try to use a module as a subroutine when this module is in the normal flow of the SM. Normally this module has its own destination and is still valid if it is normally called with the standard command "NextState(StateX)". See an example in file **STM05B.bas**.

A special command has been built.

### **NextStateReturn(StateX, StateY)**

```
$define NextStateReturn(pState1,pState2) '
    StateIndex = pState1 '
    $ifdef _CheckReturnStateM_ '
    ReturnStateIndex = pState2 '
    F_ReturnState = 1 '
    F_EnableReturnState = 1 '
    $endif
```

First, write the destination module to be used as a subroutine (**pSate1**), then the "Return" destination module (**pSate2**).

The module used as a subroutine must be validated by "**RSOn**", thus avoiding writing useless code in all modules.

### **ReturnSM(RSon)**

```
$ifdef _CheckReturnStateM_
$define ReturnSM(pEnable) '
$if pEnable = RSOn '
GoSub mNewReturnState_Sub '
$endif
GoTo StateMachineStart
```

mNewReturnState\_Sub:

```

If F_StateOverride = 0 Then
    If F_ReturnState = 1 Then ' Option Return State enabled.
        StateIndex = ReturnStateIndex ' Change the State
        F_ReturnState = 0
    EndIf
EndIf
F_StateOverride = 0
Return
$else
' Nothing to do.
$define ReturnSM() GoTo StateMachineStart

$endif

```

In the subroutine module, in case of emergency, this linkage could be broken, for example, after a communication error. See an example in file **STM05B.bas**, in STATE06.

### NextStateOverride(StateN)

```

$define NextStateOverride(pNewState) '
    $if pNewState = IncState '
        Inc StateIndex '
    $elseif pNewState = DecState '
        Dec StateIndex '
    $else '
        StateIndex = pNewState '
    $endif '
    $ifdef _CheckReturnStateM_ '
        F_StateOverride = 1 '
    $endif

```

Then the parameters are deleted and the “**ReturnState**” link is not executed being valid in the new override destination.

## The General Return State Option

You could try to use one or more modules as a subroutine. In this case these destination modules are already prepared to receive different calls from any part of the program.

Like the previous system the return module is recorded in a special variable. This module that could be the code of a keyboard, would be read by multiple calls from the main program. See the **STM05D.bas** file.

Example: Checking if all Buttons are depressed.

```

STATE10:
'=====
    StateInit()
    '##-INIT SECTION-#####
    '##-END INIT-#####
    StateInitEnd()
'::-MAIN CODE-::::::::::::::::::::::::::::::::::::::::::::::::::
    GoSub ReadButtons2PORT_Sub ' Read all Buttons.

    ' Waiting All Buttons OFF (All Buttons have a Pull-up resistor).

```

```

If ButtonsVirtualPORT = 255 Then
    ReturnIndState() ' <= Load the Return State from the caller.
EndIf

':::-EXIT MAIN-::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
StartDelayMS(30)
'=====
StateOut()
'$$$$-EXIT SECTION-$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
'$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
StateOutEnd()
'=====
ReturnSM()

```

When all Buttons are depressed the State Machine goes to the module loaded in the special variable by the '**ReturnIndState()**' command.

This switch State Mode is invoked by the command:

**NextIndState**(Next Sate, Last State)

Example: **NextIndState**(SM\_ButtonsOFF, SourceState)

The state machine goes to the NextState [SM\_ButtonsOFF] and then to the caller module. In this example it is an equivalent of a "Gosub Module\_SM\_ButtonsOFF".

In other case the "SourceState" could be any other state if the module destination code is written for this.

This new indirect addressing will be studied in the next Tutorial Part 4.

## THE DEBUG STATE OPTION:

Thanks to the modular construction, it is easy to perform a control of the state machine as already mentioned in PART1.

This option is enabled at compile time with:

Existing macros are modified.

```

$ifdef _EnableTerminal_
    $define StateInit() '
    StateIndexOld = StateIndex '
    $ifdef _SMVDelay_ '
    F_NextStateVDelay = 0 '
    $endif '
    $ifdef _CheckReturnStateM_ '
    F_StateOverride = 0 '
    $endif '
    If F_StateInitMade = 0 Then '
    $ifdef _DebugStates_ '
    GoSub DebugStateIn_Sub '
    $else '
    $ifndef _DisableCRBeforeState_ '
    _HRSOut 13,10 '
    $endif '
    $endif '

```

```

        F_StateInitMade = 1
    $endif
$ifdef _DebugStates_
DebugStateIn_Sub:
    F_StateInitMade = 1
    $ifndef _DisableCRBeforeState_
        _HRSOut 13,10
    $endif
    _HRSOut "ST",Dec StateIndex,"in",13,10
    _HRSOut "From State",Dec StateIndexFrom,13,10
    Return
$endif

$else
    $define StateInit()
        StateIndexOld = StateIndex
    $ifdef _SMVDelay_
        F_NextStateVDelay = 0
    $endif
    $ifdef _CheckReturnStateM_
        F_StateOverride = 0
    $endif
    If F_StateInitMade = 0 Then
        F_StateInitMade = 1
    $endif

```

This code is added in the standard macro.

```

    _HRSOut "ST",Dec StateIndex,"in",13,10
    _HRSOut "From State",Dec StateIndexFrom,13,10

```

When the module is executed for the first time, the current state number is sent to the terminal and the state from which it comes.

In the same way when the module exits, the macro is modified.

```

' StateOut header End
$if _defined(_CheckReturnStateM_) Or _defined(_DebugStates_)
    $define StateOutEnd()
        GoSub DebugStateOut_Sub
    EndIf

$else
    $define StateOutEnd()
        F_StateInitMade = 0
    EndIf

$endif

$if _defined(_CheckReturnStateM_) Or _defined(_DebugStates_)
DebugStateOut_Sub:
    $ifdef _DebugStates_
        _HRSOut "Goto State",Dec StateIndex,13,10
        ' There is the First pass.
    $endif
    $ifdef _CheckReturnStateM_
        If F_EnableReturnState = 1 Then
            _HRSOut "Return State",Dec ReturnStateIndex,13,10 ' <=

```



```

        EndIf
    $endif
    _HRSOut "ST",Dec StateIndexOld,"Out",13,10 \ <=
    StateIndexFrom = StateIndexOld \ <=
$endif

$ifdef _CheckReturnStateM_
F_EnableReturnState = 0           ' First pass finished.
$endif

F_StateInitMade = 0
Return
$endif

```

This is a normal visualisation of the states in the terminal:

STATES MACHINE STM05

By A. Freixanet

Test1\_DS1307.bas file

Testing the DS1307 Clock Calendar

Using the library <DS1307-H.inc>

Sending Date & Time every minute To the terminal.

The Hserial is set for Proteus testing

Check the DS1307 acknowledge.

Check if the DS1307 Clock Calendar is connected?

The DS1307 Device is ready!

Enable DS1307 signal frequency [1Hz] to PORTB.0

Minutes Count: 01

Date = Friday 17/03/2017

Time = 09:49:58

Minutes Count: 01

Date = Friday 17/03/2017

Time = 09:49:58

An option allows separating information from all states with CR,LF.

Visualisation in the terminal with the "DebugStates" option enabled.

ST0in

From State0

STATES MACHINE STM05

By A. Freixanet

Goto State1

STAT0Out

ST1In

From State0

Test1\_DS1307.bas file

Testing the DS1307 Clock Calendar

Using the library <DS1307-H.inc>

Sending Date & Time every minute To the terminal.

The Hserial is set for Proteus testing

Goto State2

ST1Out

ST2In

From State1

Check the DS1307 acknowledge.

Check if the DS1307 Clock Calendar is connected?

The DS1307 Device is ready!

Goto State4

ST2Out

ST4In

From State2

Enable DS1307 signal frequency [1Hz] to PORTB.0

Goto State5

ST4Out

ST5In

From State4

Goto State6

Return State7      <= Return State

ST5Out

ST6In

From State5

Goto State9

ST6Out

ST7In

From State6

Goto State5

ST7Out

It indicates both the state from which it is leaving and the next state. If the option "ReturnState" has been activated then 'Return State' or the new destination must be State6 + State7. See the next State6.

The normal destination after the Modulo 6 is to go to the Module 9. But if the Return State option is activated the new destination will be State6 + State7.

It is a very powerful option.

## MAIN CODE SYNCRONIZED with INTERRUPT every 1mS OPTION:

In very sensitive cases where the code cannot be disturbed by an interrupt, I have provided an option to synchronize the main code to the 1mS interrupt.

**StateInitEnd(SyncOn)**

Wait for the interrupt to end on the first line of the main code with a While/Wend loop. In this case, the SM stops completely less than 1mS.

**Sync1mS()**

This a special command with the same code waiting for the interrupt in the middle of the Module Code.

Do not forget that in 1mS there are a little less than 16000 instructions with FOSC 64Mhz, thus removing the necessary code for the SM.

```
$define StateInitEnd(pEnable) '
    EndIf '
    $ifdef _SynchronizeBlmS_ '
        $if pEnable = SyncOn '
            F_SMMain1mS = 0 '
            While F_SMMain1mS = 0 '
                Wend '
            $endif '
        $endif
```

### A Module example: (STM00.bas)

STATE07:

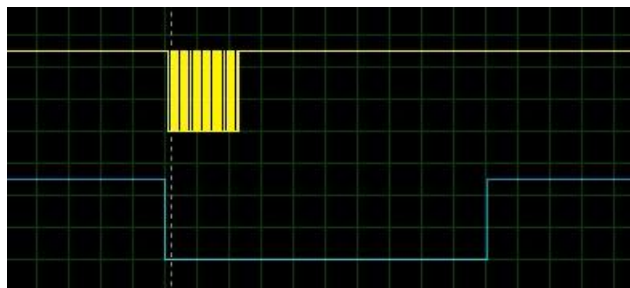
```
'=====
    StateInit()
    '##-INIT SECTION-#####
    HRSLStrg(TXT52,1) ' "Waiting All Buttons depressed!"
    '##-END INIT-#####
    StateInitEnd(SyncOn) '<= Module synchronized with timing 1mS.
':::-MAIN CODE-:::

    GoSub ReadButtonsPORT_Sub ' Init & Read all Buttons.

    ' Waiting All Buttons OFF (All Buttons have a Pull-up resistor).
    If ButtonsVirtualPORT = 255 Then
        NextStateDelay(30,SM_ReadButtons) ' Delay 30mS.
        NextState2(SM2_Message2) ' Call the State Machine2 in Background
    EndIf ' when VDelay is running.

':::-EXIT MAIN-:::
    StartDelayMS(30) ' 30 mS delay works like a debounce
'=====
    StateOut()
    '$$$-EXIT SECTION-$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
    '$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
    StateOutEnd()
'=====
    ReturnSM()
```

### STM05A.bas example: (DS1307 I2C calendar)



## Upper line of the Oscilloscope:

This is the signal I2C of the 'DS1307\_ReadTime(ReturnACK)' of 6 bytes length every second.

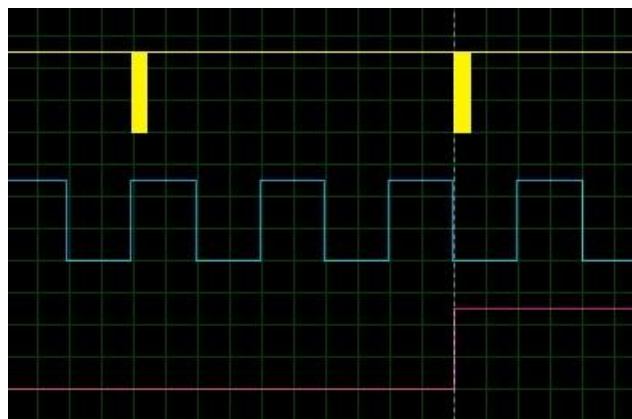
## Lower line of the Oscilloscope:

Signal synchronization generated by toggling an output by interrupt every 1mS.

According to the number of bytes of the I2C frame, the speed will be 100Khz or 400Khz for the I2C clock. In this example the clock is 400Khz so that the command 'DS1307\_ReadTime (ReturnACK)' of 6 bytes long is executed between two interruptions.

Usually interruptions do not affect hardware communications. But this provides a guarantee in case of software communications problems or another error.

## Synchronize Time & Date code STM05B.bas:



Line 2 (Blue colour) is the Interrupt Timing 1mS.

```
':::-MAIN CODE-::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
'-----
' Write the Date every minute only
If RTC_Minute <> RTC_Minute_Old Then
'-----
  #ifdef LCD#DTPORT
  High LCD_Backlight ' Set the Backlight to see the value on LCD.
  BacklightTime = BacklightTimeValue ' Backlight Set Time
  #endif
'-----
  Inc MinuteCounter
  If MinuteCounter = 60 Then
    MinuteCounter = 0
  EndIf
  HRSLStrg(TXT24,0) ' "Minutes Count: "
  HRSSOut Dec2 MinuteCounter,CR,LF
'-----
  Sync1mS() ' <= (DS1307_ReadDate) Command synchronized
            ' with timing 1mS.
'-----
  #ifdef _PinsTest_
  High Test2
  #endif
```

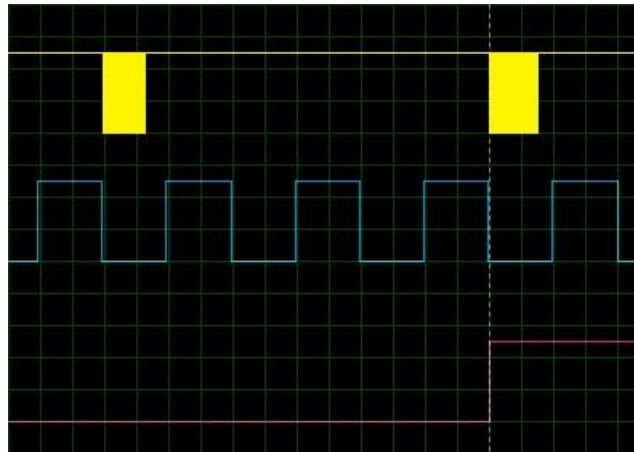
```

' Write the Date to test only
DS1307_ReadDate (ReturnACK)      ' Read the date from the DS1307
If ReturnACK = 1 Then
    HRSLSstrg (TXT19,1)          ' "NACK DS1307 Read Date"
    NextState (SM_Error_NACK)
    GoTo ERROR07EXIT             ' Bypass the code below.
EndIf
' Get the day's name into string RTC_DayName
DS1307_Get_DayName (RTC_DayOfWeek, RTC_DayName)
' Get the months's name into string RTC_MonthName
DS1307_Get_MonthName (RTC_Month, RTC_MonthName)

```

The synchronization of the Time & Date commands is perfect.

### Synchronize Time & Date code STM05A.bas:



The synchronization of the Time & Date commands is perfect.

### SECOND STATE MACHINE OPTION:

It is possible to perform some jobs in the background when the state machine is in standby caused by a Virtual Delay or busy state. This a programmer choice. The construction is absolutely identical. The virtual delay is entered as an option only. States are used from 100 and up. (STATE100)

Module **0** is not used for coding; it allows inhibiting the State Machine 2.

It is very easy to execute the operation of an SM2 module.

**NextState2**(SM2\_Function1)

This command positions the status indicator "**SM2\_Function1**" to access the module that will be executed after the calling module.

To stop, the execution of this module only places this command in an appropriate place.

**NextState2(0)**

Or write another indicator.

**NextState2**(SM2\_Function2)

## TIME COUNT OPTION:

As I said before, SM2 works in the same way as SM1. I have planned a new function, hoping it will be useful. Once an SM2 module has been started, execution can be stopped after running N times without main program intervention (SM1). See the STM02.bas file in module STATE101.

First we need to initialize a counter in the **Init Section** with this macro.

### **StateInit2End()**

```
$define StateInit2End() '
  $ifdef _TimeCounter2_ '
    SM2TimeCounter = 0 ' <=
  $endif '
  F_StateInit2Made = 1 '
EndIf
```

The SM2TimeCounter is cleared once the module is executed for the first time. A counter is added in the macro before exiting the main code. It is the best place. This way, the code will run 5 times.

### **StateOut2(5, SM2\_Init)**

SM2TimeCounter = 5, Goto Module => **SM2\_Init**

```
$define StateOut2(pCount, pIndex) '
$ifdef _TimeCounter_ '
  $if pCount > 0 '
    Inc SM2TimeCounter '
    If SM2TimeCounter = pCount Then '
      $if pIndex > 0 '
        State2Index = pIndex '
      $else '
        State2Index = 0 '
      $endif '
    EndIf '
  $endif '
$endif '
If State2Index <> State2IndexOld Then
```

## RUNNING THE SM2 IN BACKGROUND:

I am using the Amicus18 Board with the buttons and LEDs shield as I used in the Bootloader project. (STM00.bas file)

- **STATE07 – [SM\_ButtonsOff].**

This code allows for no auto repeat keys, It waits for the key to be released. In this way a single value of the pressed key is achieved. The key readings are performed every 30 mS.

When all keys are released then the module exits to the next destination with a delay of 30 mS. At this point during the delay, the State Machine 2 sends a message in the background.

STATE07:

```
'=====
StateInit()
'##-INIT SECTION-#####
HRSLStrg(TXT52,1)      ' "Waiting All Buttons depressed!"
'##-END INIT-#####
StateInitEnd(Sync1On)  ' <= Module synchronized with timing 1mS.
'::-MAIN CODE-:::::::::

GoSub ReadButtonsPORT_Sub  ' Init & Read all Buttons.

' Waiting All Buttons OFF (All Buttons have a Pull-up resistor).
If ButtonsVirtualPORT = 255 Then
    NextStateDelay(30,SM_ReadButtons)  ' Delay 30mS.
    NextState2(SM2_Message2)  ' <= Call the SM2 in Background
EndIf                                ' when VDelay 30 mS is running.

'::-EXIT MAIN-:::::::::
StartDelayMS(30)      ' 30 mS delay works like a debounce
'=====
StateOut()
'$$$-EXIT SECTION-$$$$$
'$$$$$
StateOutEnd()
'=====
ReturnSM()
```

#### • STATE09 – [SM\_LED0].

This module is used to activate or deactivate LED0 when the pushbutton SW0 is pressed. When the code ends and the module returns to the state 'SM\_ButtonsOff' an order is sent to the state machine 2 to execute a code in the background.

STATE09:

```
'=====
StateInit()
'##-INIT SECTION-#####
StartFunction1()  ' <= Blinking a LED (user function).
'##-END INIT-#####
StateInitEnd()
'::-MAIN CODE-:::::::::

HRSLStrg(TXT59,0)      ' "SW0 detected!"
If LED0 = 0 Then
    Set LED0
    HRSLStrg(TXT60,0)    ' ", Set"
Else
    Clear LED0
    HRSLStrg(TXT61,0)    ' ", Clear"
EndIf
HRSLStrg(TXT62,1)      ' " LED0"
```

```

NextState (SM_ButtonsOff)      ' Come back reading buttons

':::-EXIT MAIN-::::::::::::::::::::::::::::::::::::::::::::::::::
StartDelayMS (0) ' Delay between execution of the MAIN module
'=====
StateOut ()
'$$$-EXIT SECTION-$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
NextState2 (SM2_Message1) ' <= Call the State Machine2 when exits.
'$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
StateOutEnd ()
'=====
ReturnSM ()

```

## CONCLUSION

I finish the description of this State Machine hoping that it will help you develop your projects faster and with better program flow.

This system is designed for medium or large programs. A very high FOSC should be used without notable delays in the code. There would be no point in using a 4Mhz Xtal. That's why the PIC18FxxKxx series is the most interesting to use. Not forgetting that the K20 series can work at frequencies higher than 64Mhz. A State Machine running at 80Mhz would be fantastic (\*). This State Machine is ready for that.

(\*) The STM00.bas file has been tested with the Amicus18 Board with Xtal 16 & 20 Mhz (FOSC = 64Mhz & 80 Mhz).

Alberto Freixanet

04 April 2017