



AMICUS18®

www.protonbasic.co.uk



Powered by Proton Development Suite® Compiler of Crownhill Associates Limited©

STATE MACHINE PART5

A Clock Calendar full project with the DS1307 & the TC74

A simple multi-tasking System

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 a full project demo made with my Multi-Tasks State Machine.
It is essential to move from a linear programming approach to a structured method.

Multi-Tasking System:

I have previously introduced a very simple multi tasking system. This article substantially modifies the operation of the previously described state machine. We will study the new commands that will allow us to write the tasks very easily.

The multitasking system is reserved to run small programs in the background by lightening the main program of repetitive tasks and to simplify the writing of the user program. The idea is to run a maximum number of routines when the PIC has no external inputs or outputs or to delay the less urgent in favor of the urgent code.

It is not an RTOS system, because the tasks are simple and short. A task started cannot be interrupted by the TaskSwitcher. This very strict structure allows a programmer to obtain a very fast system. It could approach the speed of a Time Slicing system when the timing requirements are very important such as reading to the ADC every 4ms.

Note:

The program has been specially designed to work with ADC reading at 250Hz (every 4ms) so I have had to strictly comply with rules 1, 2, 3, 4, 6, 7, 10 and 11. You will see that the tasks are as short as possible. I've had to optimize all the codes, using a few tricks. For less demanding applications the task code could be longer and more standard.

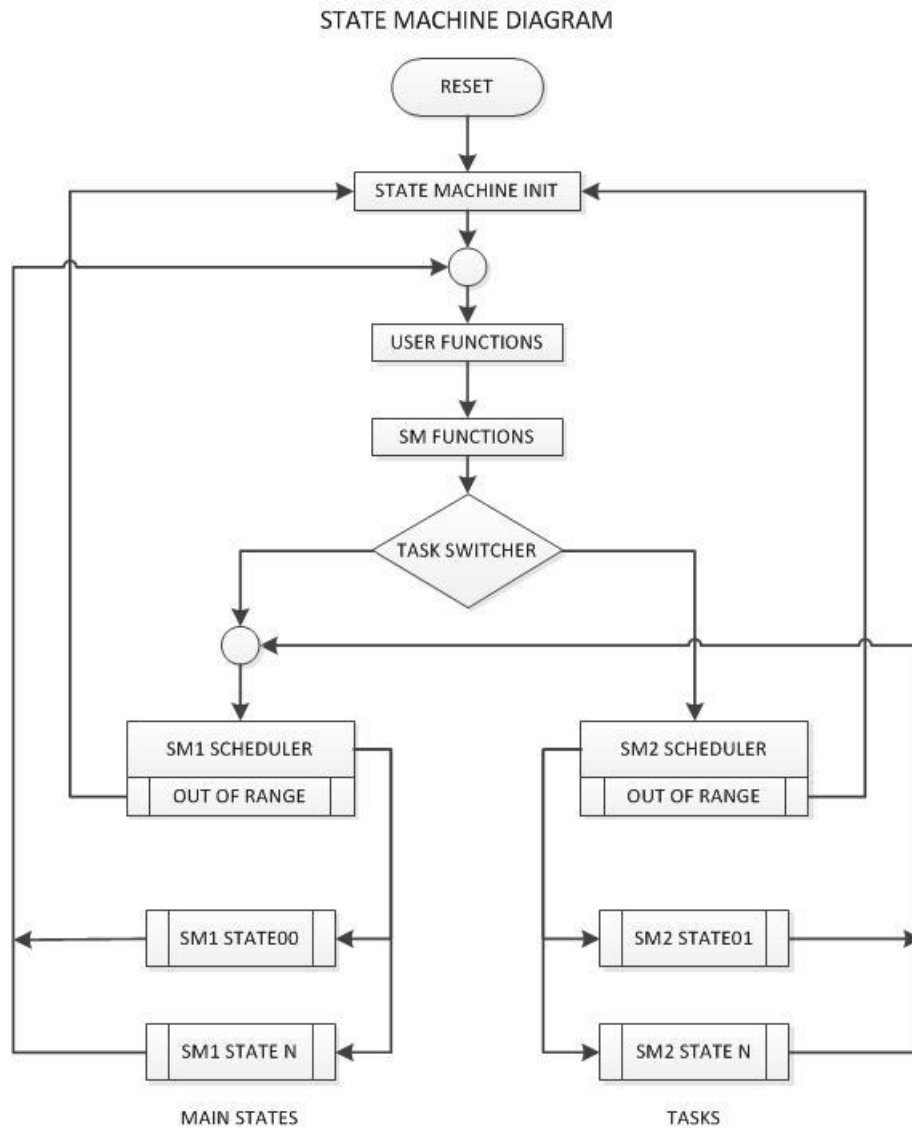
The multitasks system is reserved for the machine of states # 2. For that new rules have been defined.

New Rules:

- 1- This State Machine is a large loop that can never stop.
- 2- It is forbidden to use the DelayMS compiler command or some loops to follow rule # 1.
- 3- Timing rule: $(\text{Time Main SM1 State} + \text{Time Task max}) = (\text{ADC Timing} - 1\text{ms})$. The ADC result could be lost.
- 4- The State Machine must run at maximum speed to ensure the timing is correct. Take this into account when writing the code.
- 5- You cannot use the 'GoTo' command to a destination that is outside the static code area of a module.
- 6- The SM allows separation of the whole program into small autonomous modules. If a single-module code fails, it could be either a single problem with this same code or badly initialised parameters.
- 7- Do not delay the main code by inserting other long codes, better to use the Tasks System in the background or/and split the code into several modules.
- 8- If the ADC timing of the chosen ADC is fast, the tasks will have to run faster as well.
- 9- The Tasks Switcher distributes the tasks according to their priority.
- 10- Tasks are automatically deleted from the buffer after they are assigned.
- 11- A task (JOB) could run in one state only or could be split into several states.
- 12- Tasks are always interspersed within the main program loop. This way you will not miss the information that would be passed from a state to a task or back.
- 13- The SM would work better if the peripherals connected to the PIC will use the SPI bus. If using an LCD display, preferably use the full 8-bit bus.
- 14- The PDS user must organize all the tasks of the program.

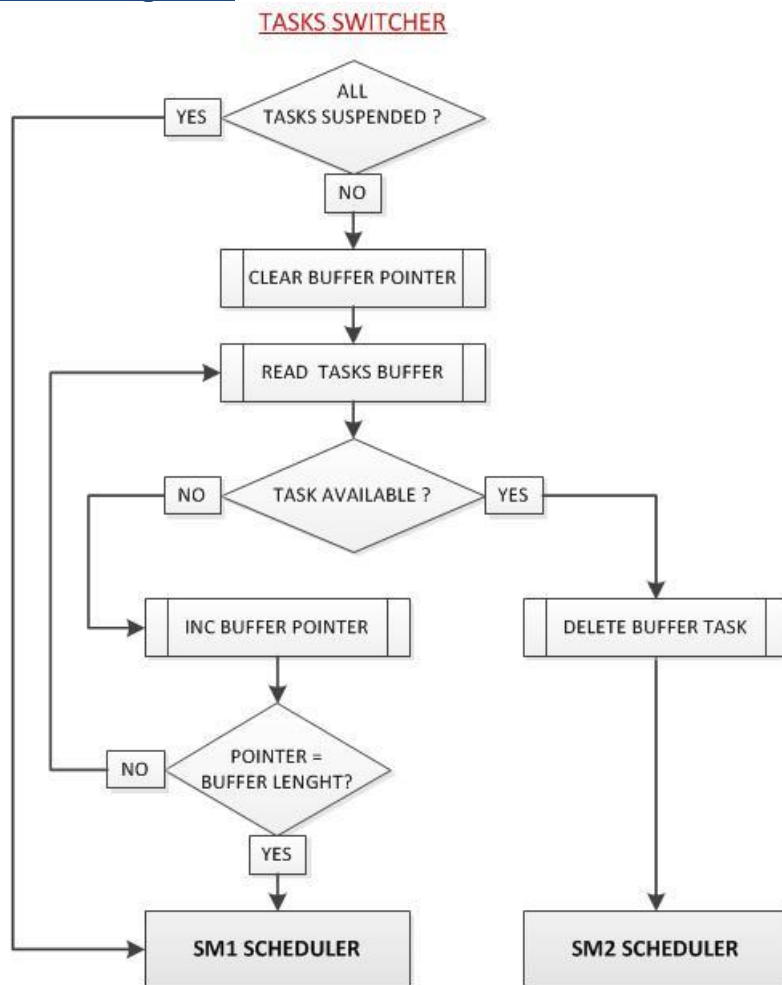
I remind you that the greatest quality of the state machine is the ease in which the code may be extended and/or maintained. The code may be considered as a set of separate mini-programs.

The State Machine Diagram:



In this way the tasks are always intertwined with the main program. Which means only one task runs at any one time.

The Tasks Switcher Diagram:



Task Switcher :

The system of tasks that I have designed is not the most sophisticated but it seems very robust, resisting interruptions every 1 ms.

Each time the Task Switcher goes into operation it starts loading Task0 if it is available and then executing the other tasks.

All tasks can be inhibited at once. You can not inhibit a task individually, you can only delete it.

Tasks should be written in a predetermined order for highest efficiency.

The most repetitive task would be the first in the list, followed by its associated tasks.

Then the tasks are written according to the urgency and in the order of execution. (very important)

The Task Switcher is very simple and controls only 1 parameter viz Task valid. When the task is loaded in the SM1 Scheduler it is deleted from the buffer, thus simplifying the general operation. See the SM2 code

Definition of a Task:

A task is a code that runs on state machine 2. This task should be as short as possible.

A task will only work once and the Task Switcher deletes it from the task lists once assigned.

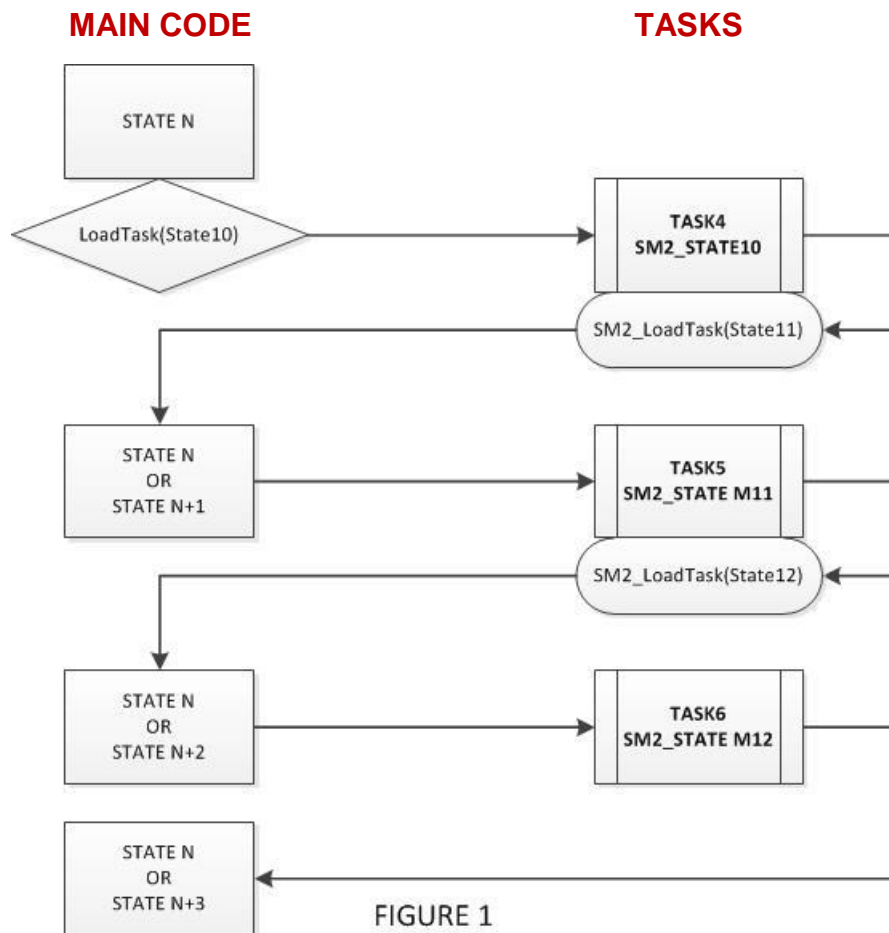


FIGURE 1

Figure 1: A background program divided into 3 tasks

Tasks are always intertwined with the main program (SM1).

To load a task, the **SM2_LoadTask(State #M)** command is used. Then all tasks will also be defined by their number and /or name:

```
$Define Task4 4    ' SM2_STATE10
$Define Task5 5    ' SM2_STATE11
$Define Task6 6    ' SM2_STATE12
```

The SM2_STATE10 state will be called from the main program by the command:
LoadTask(State10)

The SM2_STATE10 state will call the next task by the command:
SM2_LoadTask(State11)

The SM2_STATE11 state will call the next task with the command:
SM2_LoadTask (State12)

And the SM2_STATE12 state ends the job.

All tasks always end without any order because they are executed only once.

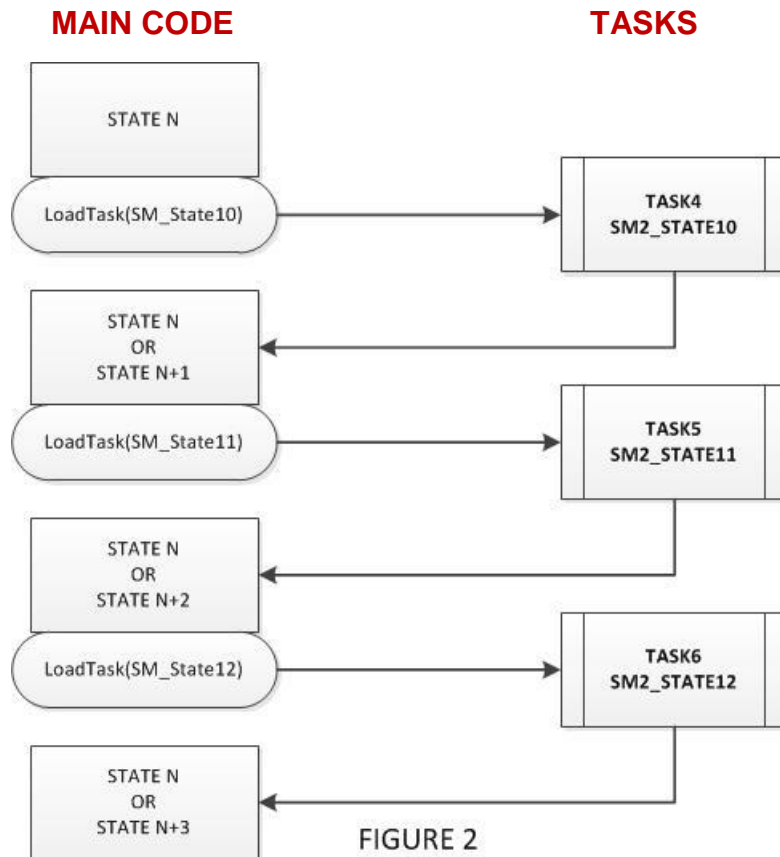
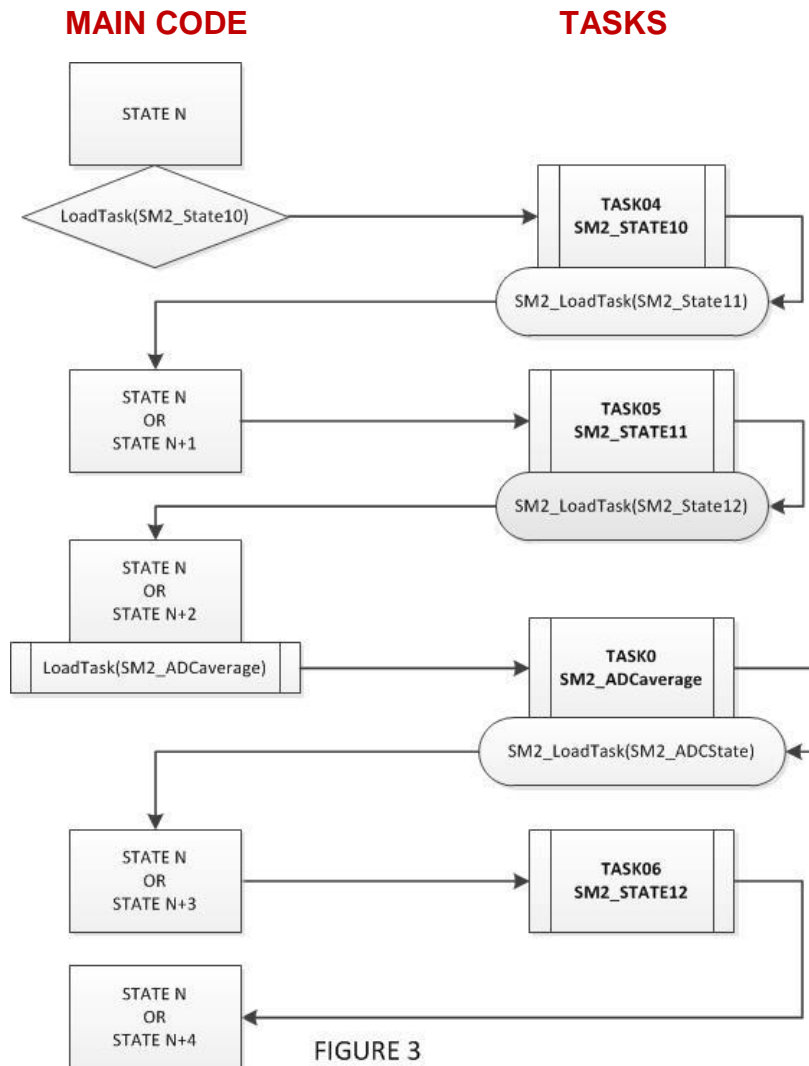


FIGURE 2

Example of a call to individual tasks.

These tasks are called individually from the main program (SM1).

Each SM2 state will be assigned a fixed task that will not be changed during the entire program, in the case of Task 0 the highest priority is assigned to the first state. The code should be simple and fast to execute. Good task organisation is very important.



A lower number task can be executed before a higher number task.

Distribution of Tasks:

States of Machine # 2

```

#define SM2_ReadTime 0
#define SM2_PrintLCDLine1 1
#define SM2_PrintLCDLine2 2
#define SM2_CheckAlarmState 3
#define SM2_ADCoverage 4
#define SM2_TC74SetNormalMode 5
#define SM2_TC74Read 6
#define SM2_TC74Close 7
#define SM2_ReadDate 8
#define SM2_Data1ToTerminal 9
#define SM2_Data2ToTerminal 10
#define SM2_TC74TempToTerminal 11
#define SM2_StateMax 12

```

The states (or Modules) of the SM1 or SM2 are not sorted in the order of execution, because it is the scheduler with the variable "state" that organises the order of execution.

This means that modules may be transferable between programs.

I have written Module 4 (SM2_ADCoverage) in fifth position, but it works first in the task list. The programmer will have to organize all tasks.

```
' Update the LABELS when you add a new State Machine 2 Module.
$define SM2_LABELS_LIST '
BranchL SM2StateIndex, [SM2_STATE00, SM2_STATE01, SM2_STATE02, _ '
                    SM2_STATE03, SM2_STATE04, SM2_STATE05, SM2_STATE06, _ '
                    SM2_STATE07, SM2_STATE08, SM2_STATE09, SM2_STATE10, _ '
                    SM2_STATE11]
```

It is very useful to assign a name to each state of SM2. To change the order of states, only the assigned numbers need to be routed.

For example, the value "SM2_PrintDateState" is loaded into the variable "SM2StateIndex" so that the "BranchL" command directs the Program Counter to the corresponding label. The "SM2_LABELS_LIST" macro allows you to update the new states without looking for lines of code in the entire file.

Organization of SM2 tasks:

Each SM2 state will be assigned a fixed task that will not be changed throughout the program.

This list is a reminder to compose the address commands of the different states.

```
' Define the Tasks Buffer length
$define SM2TasksBufferLenght SM2_StateMax + 2

' Define the Number of tasks & the corresponding starting state.
' The position of the Task defines the priority.
' All SM2 states must to have a Task number. (not the SM2_STATE00)
'-----
' User Tasks.
$define Task0 0      ' SM2_ADCoverage (04)           : (Higher priority)
$define Task1 1      ' SM2_ReadTime (0)
$define Task2 2      ' SM2_PrintLCDLine2 (2)
$define Task3 3      ' SM2_ReadDate (8)
$define Task4 4      ' SM2_PrintLCDLine1 (1)
$define Task5 5      ' SM2_CheckAlarmState (3)
$define Task6 6      ' SM2_Data1ToTerminal (9)
$define Task7 7      ' SM2_Data2ToTerminal (10)
$define Task8 8      ' SM2_TC74SetNormalMode (5)
$define Task9 9      ' SM2_TC74Read (6)
$define Task10 10    ' SM2_TC74Close (7)
$define Task11 11    ' SM2_TC74TempToTerminal (11)
'-----
' State Machine Tasks.
$define Task12 12    ' SM2 Virtual Delay
$define Task13 13    ' SM2 Delayed Task                : (Lower priority)
```

LoadDelayedTask (30000, SM2_TC74SetNormalMode)

This command, used basically in the main program (SM1), loads the state number (5) of the SM2 into the task buffer position (13) because it is a late task. Its position is predefined in the buffer as (13) by:

```
$define SM2DefDelayedState Task13      ' SM2 Delayed Task
```


SM2_NextStateDelay (250, SM2_TC74Read)

This command, used only in the task program (SM2), the status number (6) of the SM2 in the task buffer, has an already defined position (12). Its position in the buffer is (12) by:

```
$define SM2DefVDelayState Task12      ' SM2 Virtual Delay
```

Very important:

Tasks should be organized in order of priority and execution with associated tasks. For example:

```
$define Task1 1      ' SM2_ReadTime (0)
$define Task2 2      ' SM2_PrintLCDLine2 (2)
```

Task 1 and Task2 will be executed.

In the main program (SM1) you would write:

LoadTask (SM2_ReadTime)

In the program of tasks (SM2) would write:

SM2_STATE00: SM2_LoadTask(SM2_PrintLCDLine2)

SM2_STATE02: Nothing, the Task finished automatically. (SM2_PrintLCDLine2)

Task load command:

Once the task table is defined, the user will not have to remember the interleaving of Tasks and States of SM2. The command will do it all automatically if the PDS user has written all the tables correctly.

If you write to the state machine 1:

LoadTask(SM2_ReadTime)

If you write to the state machine 2:

SM2_LoadTask(SM2_ReadTime)

The correspondence is made with the following table.

```
$define LoadStateTable (pNewState) '
$if pNewState = 0      '
    SM2Task[Task1] = 0  '
$elseif pNewState = 1  '
    SM2Task[Task4] = 1  '
$elseif pNewState = 2  '
    SM2Task[Task2] = 2  '
$elseif pNewState = 3  '
    SM2Task[Task5] = 3  '
$elseif pNewState = 4  '
    SM2Task[Task0] = 4  '
$elseif pNewState = 5  '
    SM2Task[Task8] = 5  '
$elseif pNewState = 6  '
    SM2Task[Task9] = 6  '
$elseif pNewState = 7  '
    SM2Task[Task10] = 7  '
```

```

$elseif pNewState = 8      '
    SM2Task[Task3] = 8      '
$elseif pNewState = 9      '
    SM2Task[Task6] = 9      '
$elseif pNewState = 10     '
    SM2Task[Task7] = 10     '
$elseif pNewState = 11     '
    SM2Task[Task11] = 11    '
$else                      '
$error "NO SUCH STATE IN SM2" '
$endif

```

The equivalent macro is:

```

$define SM2_LoadTask(pNewState) LoadStateTable(pNewState)
LoadTask(SM2_ReadTime)

```

LoadTask(State number#00) => SM2Task[Task1] = 0, writes the value 0 in the Task array in position 1.

In the same way:

DeleteTask(SM2_ReadTime) => SM2Task[Task1] = 255, writes the value 255 in the Task array in the position 1. The value 255 means a deleted task.

```

$define DeleteTask(pNewState) DeleteStateTable(pNewState)

```

The correspondence is made in the following table:

```

$define DeleteStateTable(pNewState) '
$if pNewState = 0                    '
    SM2Task[Task1] = 255            '
$elseif pNewState = 1                '
    SM2Task[Task4] = 255            '
$elseif pNewState = 2                '
    SM2Task[Task2] = 255            '
$elseif pNewState = 3                '
    SM2Task[Task5] = 255            '
$elseif pNewState = 4                '
    SM2Task[Task0] = 255            '
$elseif pNewState = 5                '
    SM2Task[Task8] = 255            '
$elseif pNewState = 6                '
    SM2Task[Task9] = 255            '
$elseif pNewState = 7                '
    SM2Task[Task10] = 255           '
$elseif pNewState = 8                '
    SM2Task[Task3] = 255            '
$elseif pNewState = 9                '
    SM2Task[Task6] = 255            '
$elseif pNewState = 10               '
    SM2Task[Task7] = 255            '
$elseif pNewState = 11               '
    SM2Task[Task11] = 255           '
$else                                '
$error "NO SUCH STATE IN SM2"      '
$endif

```

Organization of the program:

The program has been reorganized so that the state machine is in separate files. The user program will be in 2 files (one code for SM1 and one for SM2). To update the System, you only have to enter the new files without modifying the main program.

SM PIC18F25K20.Inc

This file contains the data of the PIC and its compilation.

STM07.bas

The source file written by the user. Contains all the program corresponding to the machine of states # 1.

STMachine02.Inc

The library of commands specific to the SM other commands. This is very useful for the user, version 2.

STM07_SM2Code.bas

The source file written by the user. It contains all the program corresponding to the machine of states # 2 (Tasks). See my article in the WIKI.

Amicus_ADCbeta.Inc

Library for all commands corresponding to the Digital Analog Converter for the PIC18F25K20 and PIC18F25K22 for the Amicus18 board. I fixed some errors in macros.

DS1307-H.Inc / DS1307-S.Inc

Library for all the commands corresponding to the time/calendar circuit I²C of the DS1307. See my article in the WIKI.

LCD_ST7036.Inc

Library for all commands corresponding to the AMI18 LCD Shield (GEVO) compatible with the Amicus18 board. See my article in the WIKI.

HRSOut_K40.Inc

Library to add 2 stop bits to the HRSOut compiler command to be compatible with the CoolTerm_0 terminal used for my bootloader. It would not be necessary for another bootloader or programmer. Remove or disable this file if not using the CoolTerm_0.

ORG_SyncBlock_K20.Inc

Library to synchronize the ORG command of the compiler with the ERASE blocks. Not used at the moment.

TC74-H beta.Inc/ TC74-S beta.Inc

Library for all commands corresponding to the temperature sensor I²C TC74, adapted to the state machine. See my article in the WIKI.

SM1_Scheduler01.Inc

Code corresponding to the Scheduler of the machine of states # 1, version 1.

SM2_Scheduler01.Inc

Code corresponding to the Scheduler of the machine of states # 2, version 1.

STM_Strings07.Inc

File corresponding to the texts used for this development for indicative purposes. You may find this of value.

Characteristics of the project:

Hardware:

Amicus18 (PIC18F25K20) development board.

Clock Calendar clock device with DS1307 I²C.

Temperature system device with the TC74 I²C.

Ami18 LCD COG shield by EVO (ST7036) with White backlight LED.

4 pushbuttons for configuration settings.

1 Buzzer.

1 LED red or relay for alarm.

1 LED red for configuration.

My interface shield for buzzer and analog system for ADC.

Software:

Driver I²C for the DS1307 device, Bus speed 100 kHz.

Driver I²C for the TC74 device, Bus speed 100 kHz.

Driver for the LCD ST7036 bus 4 bits.

Contrast setting by software

Setting the clock: date, hour, minute.

Setting Alarm 1 daily: hour, minute.

Setting Alarm 2: date, hour, minute.

Output for Buzzer (pulsed for alarm, single pulse for button pressed).

Output for BackLight LED by PWM.

Output for Alarm (relay for example)

Output LED for configuration state (pulsed).

TimeOut for all configuration states.

Reading Time and ADC result every second and print to the LCD.

Checking the alarms every minute

Reading Date and Temperature every minute to the LCD and Terminal.

Reading ADC with configurable counter from 4ms started by interrupt.

Average for 2, 4, 8, 16, 32, 64 ADC samples.

Optimised average and maths calculations for ADC readings.

Debug:

All number states of SM1 of the user program.

ADC system, checking if there is a lost sample.

Configuration of the Amicus18 board:

Bootloader:

I am using the AGV Bootloader LSM V4.1 for the PIC18F25K20 for 64MHz and 80MHz. It is a very reliable and protected bootloader in the Boot Sector of the PIC. When choosing this bootloader the PROTON_START_ADDRESS is automatically changed. The Bootloader is available in the WIKI. You can find a copy in the "Bootloader" folder. You can use the Amicus18 bootloader available in the IDE as well.

ICSP:

To use another bootloader you need to program the firmware in the PIC using the ICSP bus. The board has some problems that need to be solved first.

- Place a 1N4148 diode in series with resistor R3 (1K or 2K7) (anode towards + VDD) of the PIC reset input.
- Insert a wire between the pin 20 (VDD) of the PIC and the pin VDD of the ICSP connector. Otherwise the PICKit3 / ICD3 will not see the PIC.

It would be possible to place a multilayer type 100nF ceramic capacitor between pins 20 and 8 of the PIC.

RB1:

The RB1 Bridge in position RB1.

XTAL:

The mounted xtal (16MHz) allows work up to 64MHz with the PLL.

If you need to work at a frequency of 80MHz, change the xtal to another of 20MHz and use the PLL.

See Les's article in the WIKI. To use the PIC with FOSC > 64MHz, I advise not to charge the outputs, preferably use a current less than 0.5mA.

I²C Clock for 80MHz:

There is an issue with the **HBus_Bitrate** Declare. When the **Xtal = 80** is used the Rate is divided by 4.

For Xtal 64MHz:

Declare HBus_Bitrate = 100

For Xtal 80MHz:

Declare HBus_Bitrate = 400

To have a 100 kHz result.

The Formula is: $F_{Clock} = 1 / ((SSPADD + 1) \times 4) / FOSC$

Program settings:

Declare the Xtal:

For the simulation in Proteus I am using: **Declare Xtal** = 16

For the Amicus18 board I am using: **Declare Xtal** = 64

For the Amicus18 plate with Declare Portal = 80 MHz I am using the option:

\$Define _Amicus18_80MHz_ at the beginning of the main program. This line will change all the parameters.

Declare the Watchdog:

The compiler Watchdog must be disabled.

Declare Watchdog = OFF

The Watchdog settings in the Config Fuses are confusing. With WDTEN = OFF, it does not mean that the WDT is disabled or rather it depends on the SWDTEN bit.

Although the watchdog is set in operation by the Config Fuses and the **SWDTEN = 1** bit for the states machine, a single **Clrwdt** instruction is really needed.

As the programmer must follow rule 1 (*This State Machine is a large loop that can never stop*), the SM always returns to the same site where the **Clrwdt** instruction is placed. It is the best watchdog system, the most reliable and allows you to save many lines of code. If the state machine is too slow or stopped, the watchdog will be triggered.

Only at the start of the program are several instructions placed.

Initialization of the program:

When initializing the program it would be very opportune to know the reason for the start. For that I have written a series of controls to know the reason for the System reset. It would be useful for all programs.

```
' At Power up control the state of the PIC.
' Check the Reset Flag bit
If RCONBits_RI = 0 Then
    RCONBits_RI = 1
    HRSOut "RESET", CR, LF
    F_Reset = 1 ' To see the error in the LCD display.
EndIf
' Check the Watchdog Time-out Flag bit
If RCONBits_TO = 0 Then
    HRSOut "WATCHDOG", CR, LF
    F_Watchdog = 1 ' To see the error in the LCD display.
EndIf
' Check the Brown-out Reset Status bit
If RCONBits_POR = 1 Then
    If RCONBits_BOR = 0 Then
        RCONBits_BOR = 1
        HRSOut "BROWN-OUT", CR, LF
        F_BrownOut = 1 ' To see the error in the LCD display.
```

```

    EndIf
EndIf
' Check the Power-on Reset Status bit
If RCONBits_POR = 0 Then
    RCONBits_POR = 1
    HRSOut "POWER-ON", CR, LF
EndIf
' Check the Stack pointer register
If STKPTRBits_STKFUL = 1 Then
    STKPTRBits_STKFUL = 0
    HRSOut "STACK FULL", CR, LF
    F_StackFull = 1 ' To see the error in the LCD display.
EndIf
If STKPTRBits_STKUNF = 1 Then
    STKPTRBits_STKUNF = 0
    HRSOut "STACK UNDERFLOW", CR, LF
    F_StackUnderflow = 1 ' To see the error in the LCD display.
EndIf

```

COMMANDS:

SM1 Commands:

The commands of the machine of states #1 are still valid, see the manuals Part2, Part3 and Part4.

SM1_State() – User command

Equivalent to the StateIndex value.

NextState(NewState) – User command

SM1 is informed of a new status change, equivalent to "Goto New State".

NextStateOverride(NewState) – User command

The SM1 is informed of the status change, replacing an equivalent command. It is used with the "Return State" function.

NextStateReturn(Next State,Return State) – User command

Indirect addressing:

The SM1 is informed of the state change (Next State) and has to return to the "Return State" state. It is used with the "Return State" function.

NextIndState(Next State,Return State) – User command

Indirect addressing:

The SM1 is informed of the state change (Next State) and has to return to the "Return State" state. It is used with the "**ReturnIndState (Delay)**" command.

DisableReturnState() – User command

Indirect addressing:

Disable the Return State command deleting all flag parameters.

IncState() – User command

Increments the variable "StateIndex" to change of state. It occupies fewer bytes than the (NextState (NewState) command).

DecState() – User command

Decrements the variable "StateIndex" to change of state. It occupies fewer bytes than the

Sync1mS() – User command

Synchronize the following code to the 1ms interrupt. It can be used in the middle of a program. This command can delay the user code of a maximum of 0.99 ms.

StartTimeOut(NextState, Delay) – User command

This command works in conjunction with "**NextState** (NewState)". It informs the SM1 that at the end of the TimeOut it will have to go to the state defined as NextState after a delay of seconds. Example:

```
StartTimeOut (SM_AskForConfig,10) ' Start a Time Out for 10 seconds.  
NextState (SM_ContrastSetup) ' Menu Configuration of the LCD contrast.
```

StateInit() – State Machine Template

The "Transitional Input State" is initialized. SM and debug functions are incorporated if the latter is activated.

StateInitEnd() – State Machine Template

It can incorporate a SM function if activated.

StateInitEnd(SyncOn) – State Machine Template

Synchronize the following code to the 1ms interrupt.

StateOut() – State Machine Template

Initializes the "Transitional Output State".

StateOutEnd() – State Machine Template

The "Transitional Output State" is initialized and debug functions are added if they are activated.

SM_Return() – State Machine Template

Return to the SM1 scheduler. SM functions are added if activated.

SM_Return(RSON) – State Machine Template

Return to the SM1 scheduler, used with the Return State function.

SMPulseOut(PorPin, Delay)

This new command replaces the old one for the Buzzer, it is a generic command for all PIC output pins, only for those that have been defined in the [RunPulsePinCode_Sub] subroutine.

LoadTask(New State)

This new command loads a status number (SM2 module) into the task list in a predetermined position which will be executed according to its priority by the Task Switcher.

LoadDelayedTask(Delay, New State)

This new command loads a new task defined by a status number (module) of the SM2 that will run only in a given time defined by a Delay.

DeleteDelayedTask()

It clears from the SM2 task list the status that was previously loaded.

DeleteTask(Numer of State)

This new command deletes a status number (module) already entered in the task memory of the SM2; Only the name of the state is written, the command will automatically search for the task number.

SuspendAllTasks()

This command suspends (does not erase) all the recorded tasks of the SM2.

ResumeAllTasks()

This command resumes all recorded tasks on the SM2. But it does not prevent the tasks from being written to the buffer.

SM2 Commands:

The Module **0** is used now for coding.

It is very easy to execute the operation of an SM2 module, only write the code in the space reserved for it.

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:

SM2_NextStateDelay(250, SM2_TC74Read)

This **NextState** command gives the order to load the "SM2_TC74Read" go to another module but with a delay of **250** ms. In this case corresponding to a case of the demo program, activate a sensor and wait 250ms to be active.

SM2_LoadTask(New State)

This new command loads a status number (module) into the task list in a predetermined position that will be executed according to its priority by the Task Switcher.

SM2_DeleteTask(Numer of State)

This new command deletes a status number (module) already entered in the task memory; Only the name of the state is written, the command will automatically search for the task number.

SM2_SuspendAllTasks()

This command suspends (does not erase) all the recorded tasks of the SM2.

SM2_ResumeAllTasks()

This command resumes all recorded tasks.

SM2_LoadTaskST(Number of Task, Number of State2)

This command loads a state of SM2 into a task that is not defined for this module. It is a special case, it can be used very carefully only after much experience in this state machine. This command could delete a default task. The reason for using this command would be to load a module in a position of highest priority. In this case you could set a high priority task blank (no defined state).

I will describe the code in as much detail as I can so that the beginner can understand how this project is very different to linear programming.

INTERRUPT HANDLER:

The Timer2 and PR2 are used to generate an interrupt every 1ms, because it is easy to configure and does not need any code to reload the timer in the interrupt handler. Several timings have been prepared to generate future functions. The interrupt handler can only be used for timers, counters and information flag. No code can be written in this space.

Some functions have been described in tutorial 4.

1 Hz interrupt: (by the DS1307)

Interruption generated by the 1Hz signal of the DS1307.

```
' The Timer0 interrupt is used at High to Low edge interrupt.
If INTCONBits_TMR0IF = 1 Then
    T0CONBits_TMR0ON = 0      ' Stop the Timer0.
    Nop
    TMR0L = 255                ' Reload the Timer0 for next interrupt.
    T0CONBits_TMR0ON = 1      ' Restart the Timer0.
    INTCONBits_TMR0IF = 0
    F_DS1307_1S = 1           ' External flag for DS1307 interrupt.
EndIf
```

On the LCD board the 1 Hz signal is not connected to the PIC.

There was no PORTB pin available to make an interrupt, but it could be connected to PORT4 corresponding to timer0 external input.

By correctly setting the Timer0 with a preset value of 255, with falling edge of the input pin, a direct interruption can be obtained at each pulse of the 1 Hz signal.

This signal is not used for the moment but the code is available for the PDS user.

New Timers:

The new Timers allow you to activate some flags for special functions, such as user functions or the reading of an analog input by the ADC module.

ADC Reading: (from Timer4mS)

In every Timer the ADC is started when the flag “F_DisableADCInt” is clear.

```
If F_DisableADCInt = 0 Then
    ADCON0bits_GO_DONE = 1          <= Start the AD conversion
EndIf
```

Waiting the interrupt bit of the ADC instead of the WR bit could save more than 10 uS time for the interrupt handler (very important).

```
Inc TaskADC_Timer          ' Increment every 1 mS.
If TaskADC_Timer = TaskADC_Interval Then
    TaskADC_Timer = 0
    If F_DisableADCInt = 0 Then
        ADCON0bits_GO_DONE = 1 ' Start an AD conversion.
    EndIf
EndIf

If PIR1Bits_ADIF = 1 Then
    PIR1Bits_ADIF = 0
    LoadTask(SM2_ADCAverage)
EndIf
```

A special code for debugging the ADC between interrupt and the average routine to control the possibility of a lost ADC value. The calculation routines must be synchronized with the interrupt “PIR1Bits_ADIF” bit, checking the value of the ADCCounter. When the code runs well, the ADC debugging code could be removed to improve the speed a little.

Input Aliasing filter:

When using an ADC, you should not forget to place an anti-aliasing filter (ADC sampling X 2) in the ADC input, rather than 4th order. The impedance of the signal source should be as low as possible so as not to interfere with the ADC's holding capacitor time.

The schematic of the filter is simply an indication. The values correspond to the 250Hz frequency of the ADC.

To measure a DC voltage, only a good RC filter (C = high value) is required.

PWM LCD Backlight Timer: (BackLightTimer)

```
Inc TaskBackLight_Timer           ' Increment every 1 mS.
If TaskBackLight_Timer = TaskBackLight_Interval Then
    TaskBackLight_Timer = 0
    High bLCD_BackLight
    Set F_PWMBackLight
    PWMBLCounter = PWMBLCounterValue
EndIf
' Count down the PWM BackLight.
' Do every mS.
If F_PWMBackLight = 1 Then
    Dec PWMBLCounter           ' Decrements the DelayCounter every 1mS.
    If PWMBLCounter = 0 Then ' Check if delay counter reaches 0.
        F_PWMBackLight = 0     ' Delay End, clear the delay FLAG.
        Low bLCD_BackLight
    EndIf
EndIf
```

TaskToggleFlag1 Timer: (TaskToggleFlag1_Timer)

```
' Toggle a flag for Blinking LEDs. Generic Timer.
Inc TaskToggleFlag1_Timer           ' Increment every 1 mS.
If TaskToggleFlag1_Timer = TaskToggleFlag1_Interval Then
    TaskToggleFlag1_Timer = 0
    Toggle F_ToggleSignal1 ' 1 Hz Timer, to toggle some leds.
EndIf
```

Task1 Second Timer: (Task1S_Timer)

```
' 1 Second flag to read the clock/calendar
Inc Task1S_Timer           ' Increment every 1 mS.
If Task1S_Timer = Task1S_Interval Then
    Task1S_Timer = 0
    F_TaskTimer1S = 1           ' For Main state05
EndIf
```

Pulse Delay: (Library Pulse command delay)

```
' Count every 1mS.
If F_PulseDelay = 1 Then ' The Delay is started.
    Dec PulseCounter       ' Decrements the DelayCounter every 1mS.
    If PulseCounter = 0 Then ' Check if delay counter reaches 0.
        F_PulseDelay = 0     ' Delay End, clear the delay FLAG.
    EndIf
EndIf
```

A delay is included in the library to generate a pulse on a PIC pin. We will see this later.

SM2 Virtual Delay: (SM2VDelayCounter)

```
' Virtual Delay 1mS for the SM2 States.
If F_SM2VirtualDelay = 1 Then ' The VDelay is started.
    Dec SM2VDelayCounter       ' Decrements the SM1VDelayCounter every 1mS.
    If SM2VDelayCounter = 0 Then ' Check if delay counter reaches 0.
        F_SM2VirtualDelay = 0     ' Delay END, clear the Vdelay FLAG.
        SM2Task[SM2DefVDelayTask] = SM2VDelayStateIndex ' Loading Task.
    EndIf
EndIf
```

SM2 Delayed State: (SM2TaskDelayCounter)

```
' Delay counter for the Delayed task.
If F_SM2_TaskDelay = 1 Then
    Dec SM2TaskDelayCounter
    If SM2TaskDelayCounter = 0 Then
        F_SM2_TaskDelay = 0
        SM2Task[SM2DefDelayedTask] = SM2DelayedState
    EndIf
EndIf
```

A new command allows you to execute a task after a programmed delay.

User Virtual Delay: (explained yet. Delay1,2,3)

```
' Generic User Delay
' 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
```

STATE MACHINE 1: (Scheduler: SM1 Scheduler00.inc)

The scheduler of the state machine 1 consists of 3 parts:

- 1- The Tasks Switcher reads the buffer tasks to send them to the state machine 2.
- 2- The SM1 Scheduler connects the addresses of the states according to the state variable.
- 3 - The security control of the Scheduler, warns of a failure in the number of states.

(1) Tasks Switcher:

SM2_TasksSwitcher:

```
If F_SM2_SuspendAllTasks = 0 Then
    '-----
    SM2TaskLoop = 0 ' Always read the Task0 first (highest priority).
    Repeat
        SM2StateIndex = SM2Task[SM2TaskLoop] ' Read any Task.
        If SM2StateIndex < 255 Then ' Did This Task suspended?
            '-----
            SM2Task[SM2TaskLoop] = 255 ' YES, this task is valid.
            ' Read Task done.
            GoTo StartStateMachine2 ' Go to SM2 Scheduler
            '-----
        EndIf
        ' There is not a task,
        ' Check next position of the buffer.
        Inc SM2TaskLoop
    Until SM2TaskLoop = SM2TasksBufferLenght
EndIf
' No Task in the Buffer.
```

The Task Switcher always starts reading the last areas of Task0. This way the tasks of higher priority will always work first. You cannot inhibit a task, you can only delete it by typing the value 255 into the buffer.

The task value, corresponding to the SM2 status number, is loaded into the SM2StateIndex variable used by the SM2 Scheduler to go to the corresponding state, and this task is then deleted from the buffer.

The flag "**F_SM2_SuspendAllTasks**" allows to suspend all tasks.

(2) SM1 Scheduler:

The task value, corresponding to the status number of SM1, is loaded into the StateIndex variable that the Scheduler uses to go to the corresponding state. State Machine 1 works as described in previous tutorials.

(3) Security of Scheduler SM1:

In case the programmer enters states and does not update the **SM1_LABELS_LIST**, an error will be generated in the terminal: "**SM1 STATE OUT OF RANGE:**" showing the wrong state number during the execution of the program.

STATE00: (Initialisation of parameters and/or Title of the project)

User command: **OpenADCAN4()**,

Configure the ADC of channel 4 of the PIC on pin PORTA.5. Note the reading is not yet activated.

User command: **SM_DisableADCInt()**,

Interrupts for the ADC are overridden. The ADC System does not work yet.

Temperature = 99

It is transmitted to the print job that the temperature reading is not ready.

```
' Deleting all Tasks
DeleteLoop = 0
Repeat
    SM2Task[DeleteLoop] = 255
    Inc DeleteLoop
Until DeleteLoop = SM2TasksBufferLenght
```

To initialize the system, you must cancel all tasks at startup.

User command: **SetBackLightTime(Max)**

In this module some sentences are sent to the LCD that need to be illuminated. This command activates the backlight without delay.

The AMI18 LCD Shield is a commercial product at www.picshop.nl, it is fully compatible with the Amicus18 Board. However the pin used for this function is PORTB.1. It is a badly chosen output because a PWM function cannot be used to vary the brightness of the LCD.

The function P1C could be used but the Timer2 is busy performing the timing of the interruption of the SM. There is no other solution than to perform my own PWM function.

Two levels of brightness are really needed; One maximum and another one to set the minimum brightness for the LCD.

Using the available means, it is possible to realise a PWM with a resolution of 10 steps (between 3 and 4 bits of resolution). This is sufficient for our application.

Two codes are needed in the interrupt routine.

- Activate the backlight every 10mS (available in the SM)
- Set a value of a 1ms resolution delay to perform the PWM.

```
' Start the PWM BackLight every 10mS
High LCD_BackLight
Set F_PWMBackLight
PWMBLCounter = PWMBLCounterValue

' Count down the PWM BackLight.
If F_PWMBackLight = 1 Then
    Dec PWMBLCounter      ' Decrements the DelayCounter every 1mS.
    If PWMBLCounter = 0 Then ' Check if delay counter reaches 0.
        F_PWMBackLight = 0 ' Delay End, clear the delay FLAG.
        Low LCD_BackLight
    EndIf
EndIf
```

Normally the order of the routines should be reversed in the interrupt routine, compensated by writing (BackLight Time = pTime + 1)

SetBackLightTime(Max) Macro:

```
$define SetBackLightTime(pTime) '
    $if pTime = Max              '
        BackLightTime = pTime + 1 '
        PWMBLCounterValue = 11    '
    $else                        '
        PWMBLCounterValue = 11    '
    $endif
```

If (pTime) value is different from the word "Max" the code corresponding to the time is not pasted in the code. This time value is no longer required to activate the backlight continuously.

The command activates the BackLight counter to 11. Since the counter has 10 stages to decrement, the PWMBLCounter could never reach 0 so the BackLight is always on.

SetBackLightTime(3)

The time counter is set to 3 seconds.

Library command: **HRSLStrg(TXT0,AllChars,1)**

This command is from the library of the State Machine available to the user of PDS. All project texts are written at the end of the .bas file with a Cdata table but the HRSSOut command writes its table at the beginning of the program.

For this reason a specific macro has been written. Special functions have been added to this occasion. Calling a **LABEL** can save many bytes of code if it is used more than once.

TXT0: Label of the String to print.

AllChars: Print all the characters of the String (1 to 255).

1: Carriage Return + Line Feed number is sent to the terminal (0 to 255)

Example:

HRSLStrg(TXT0,AllChars,1)

It may be written more simply as **HRSLStrg(TXT0, 1)** if it is assumed that all characters are written.

But you cannot write more code on the same line as it would generate a compile error.

SM command: **NextStateDelay(3000,IncState)**

This command has been described in an earlier chapter. As there is no more code to execute, the information is given to the SM to go to the next state with a delay of 3000 ms.

User command: **LCD_Clear()**

This macro corresponds to the following code. It is equivalent to Cls.

Print \$FE,1

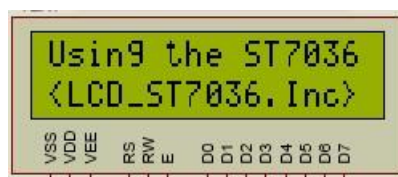
Library command: **PrintStrg(TXT0,1,1,AllChrs)**

This command performs the same function as above but applies to the Print command of the PDS.

Some parameters are initialised.

```
SM_UserFunctionFlags = 0
SM_UserSystemFlags = 0
AllButtonsBitsDefects = %11110000    ' NO button fails. Reset value.
ButtonFail = 0
```

STATE01: (Information of the project: LCD, ...)



Library command: **HRSE2PStrg(Address,AllChars,1)**

This command sends to the terminal the strings written by the **Edata** command in the EEPROM memory.

Reading the IDLOCS:

This code reads the contents of the USER ID in the PIC. In this case the version of this program is sent to the terminal.

User command: CheckAlarm1Avaible()

This command allows a user to check if the data of the Alarm1 has been written in the EEPROM memory.

```
$define CheckAlarm1Avaible() GoSub CheckAlarm1Avaible_Sub
```

```
CheckAlarm1Avaible_Sub:
    ALAvaible = ERead EEADR1_Alarm1Avaible
    If ALAvaible = $5A Then
        F_Alarm1Avaible = 1
    Else
        F_Alarm1Avaible = 0
    EndIf
    Return
```

The macro is written with a subroutine because it will be used more than once.

It is the same code for Alarm2.

```
$define CheckAlarm2Avaible()
```

Reading the Config Fuses:

The CONFIG2H byte of the Config Fuses is read to set the status of the Watchdog and BrownOut.

STATE02: (Check the DS1307 acknowledge)

A new global variable has been defined to be used in this module and later.

Dim ReturnACK As Byte

```
DS1307_Present(ReturnACK)
If ReturnACK = 1 Then
    HRSLStrg(TXT14,1)           ' "NACK DS1307 RTC device!"
    nextState(SM_Error_NACK)    ' The program demo cannot run.
Else
    HRSLStrg(TXT13,1)           ' "The DS1307 Device is ready!"
    nextState(SM_Enable1Hz)
EndIf
```

Before proceeding with the program, it is essential to know if the DS1307 circuit is working or connected properly. This function is performed by a command from the library that I wrote some time ago.

There are 2 possible answers.

If the acknowledge received is incorrect, the State Machine is sent to an error module.

nextState(SM_Error_NACK)

If not, it will execute the code in a new module

nextState(SM_Enable1Hz)

STATE03: (Config the DS1307 to Output the 1Hz signal)

```
DS1307_WriteControl(%10010000,ReturnACK)
If ReturnACK = 1 Then
```

```

    HRSLStrg(TXT16,1)      ' "NACK DS1307 Write Control"
    NextState(SM_Error_NACK) ' Go to SM_Error_NACK State.
Else
    AllButtons = 0
    StartTimeout(SM_AskForConfig,10) ' Start Time Out for 10Sec
    NextState(SM_ContrastSetup) ' Configuration of LCD contrast.
EndIf

```

The DS1307 circuit is configured to obtain 1Hz at the corresponding output that is sent to the PIC PORTA.4 to generate an interrupt. After that the SM is sent to a contrast adjustment menu. The contrast of this LCD is adjusted by software.

NextState(SM_ContrastSetup)

SM command: **StartTimeout(ASK_ForConfig,10)**

When entering any menu where a key or a keyboard is read, a timeout is necessary to return to the execution of the main program after some time without typing. The timeout value is 10 seconds in this example. The most interesting thing is that the SM could be sent to any destination as it is not a simple subroutine. The target module would then be "ASK_ForConfig" in case of a timeout after 10 seconds.

SM command: **NextState(SM_ContrastSetup)**

This command sends the SM to a new menu to adjust the contrast.

StateOut()

Depending on the case it is necessary to define the parameters before executing the next code. It is not always possible in the next module and will depend on its construction, especially when there is a loop.

STATE04: (Update Time & Date for the DS1307)

This module is responsible for writing the Date and Time data in the DS1307 circuit. It will be called when the date and time need to be updated. These macros have been updated in the library to introduce the Byte variables as well. Once the clock update is done, it will go to the main module (SM_ReadTimeDate).

DS1307_WriteDate(VDayOfWeek, VDay, VMonth, VYear, ReturnACK)

DS1307_WriteTime(VHour, VMinute, 0, ReturnACK)

The Minute parameter is always equal to 0 to write.

STATE05: (Running Time & Date)

It is in the main module of the program where everything happens.



Transitional Input State:

LoadTask(SM2_PrintLCDLine1)

Prints the data currently available on line 1 of the LCD to prevent this line from being blanked.

StopToggleConfig()

The LED blinks when the System is in configuration mode. This LED is cleared when you return to normal status (status 5).

The activation time of the BackLight is defined. The TimeOut is deactivated, it is not necessary in this code.

SetBackLightTime(5)

DisableTimeOut()

SM_EnableADCInt()

The ADC control system is initialized and interrupts are activated.

Static State:

The NACK errors of the clock and temperature circuits are controlled.

```

If F_ErrorNACK_DS1307 = 1 Then
    F_ErrorNACK_DS1307 = 0
    NextState (SM_DS1307Error_NACK)
    GoTo ERROR05EXIT
EndIf
If F_ErrorNACK_TC74 = 1 Then
    F_ErrorNACK_TC74 = 0
    NextState (SM_TC74Error_NACK)
    GoTo ERROR05EXIT
EndIf
```

Every Second:

The main function of this module is to know the time every second and the date every minute. The 1 Hz timing is generated by the interrupts system which is better than the 1Hz input from the DS1307 circuit via the PORTA4 pin because the Tasks Switcher can manage better the timing.

LoadTask(SM2_ReadTime)

Loading a Task to read the Time of the DS1307. The time is reached with the command: DS1307_ReadTime(ReturnACK) in the module (SM2_ReadTime), which always verifies the validity of the I2C communication by checking "ReturnACK".

To print the results on the LCD and also on the terminal the command is given to the SM2 to run this job after the 1 second interrupt.

Every Minute:

LoadTask(SM2_ReadDate)

This task is responsible for reading the date.

LoadTask(SM2_ChackAlarmState)

A request is done to execute a task every minute, in this case to check the activation status of the alarms.

LoadTask(SM2_Data1ToTerminal)

This task is responsible for sending the date and time to the terminal.

LoadDelayedTask(30000, SM2_TC74SetNormalMode)

Do not load all tasks at the same time every minute, a new deferred task is loaded in 30 seconds. You will begin reading the task by communicating with the temperature sensor TC74. As the sensor is to be activated and read, it will require 3 states of the machine of states # 2. Operation will be described below.

The activation time of the BackLight is set to 5 seconds:**SetBackLightTime(5)**

Next and every second, the time of the BackLight is decremented until deactivated. In this case, to make the LCD slightly visible, a PWM of 30% is set to illuminate the LCD: "PWMBLCounterValue = 3".

Checking the pushbuttons:

This is a function that we will see in module 8. Each minute the fault is reset to re-check the buttons.

```
AllButtonsBitsDefects = %11110000  
ButtonFail = 0
```

Reading the pushbuttons:

This device has 4 pushbuttons to perform the main tasks of the calendar. The reading of these pushbuttons cannot be executed at the same time as the date and time readings. It is important not to delay the SM. The reading is chosen at a time when no main task is performed. In this way the tasks are divided over time.

PushButton 1:

This button causes a jump to the LCD contrast adjustment module. A TimeOut of 10 sec is set in case no key is pressed. If the TimeOut is finished, the SM would return to this same destination.

StartTimeout(SM_ReadTimeDate,10)

To avoid problems with the pressed keys that are difficult to control in a menu, the command is given to check if the button has been raised before going to the destination. (Indirect Addressing)

NextIndState(SM_ButtonsOFF2,SM_ContrastSetup)

PushButton 2:

This button allows you to go to the date/time or alarms configuration module. A TimeOut of 10 sec is set in case no key is pressed. If the TimeOut is finished, the SM returns to this same destination.

PushButton 3:

This button lets you go to the alarm status check module and activate the daily alarm 2.

PushButton 4:

This button allows you to go to this module to disable the alarms.

Alarms:

The alarms are controlled. If one alarm is activated then the SM will go directly to the alarm module (SM_Alarm).

Transitional Ouput State:

No code

STATE06: (NACK error Module)

A warning of this error is sent to the LCD and the terminal. Any pushbutton may be pressed to interrogate the DS1307 circuit again. If it fails again then the SM would return to this module.

STATE07: (Configuration Time/Date or Alarms)



This menu allows you to choose the way to configure the calendar or alarms. It is a somewhat peculiar system to make the most of SM.

When calling this module you have to define: AllButtons = 0

This value is recognized as reading the keyboard with a TimeOut of 10 seconds.

StartTimeOut(SM_ReadTimeDate,10)

NextIndState(SM_ButtonsOFF,CallerState)

If the TimeOut is activated, it will return to the main module (5). The indirect address command is used to read the pushbuttons.

CallerState

It is the generic information to inform the SM that it must return to this same module after having read a pushbutton like a Gosub command.

STATE08: (Check if all Buttons are OFF)

The system of reading the pushbuttons or of a keyboard behaves in 3 parts.

- 1- Check if the push buttons are open to follow.
- 2- Read any tight buttons.

3- Wait until the push buttons are open to follow.

In this way a reading of a pushbutton or a keyboard will never fail.

Step 1: Check that the push buttons are open to follow.

GoSub ReadButtonsPORT_Sub *' Init & Read All Buttons.*

The code written in a subroutine will be read twice. In this first reading all the inputs of the pushbuttons are configured previously and then the value of all the bits input are read for module SW_ButtonsOFF.

In the second reading in Step 3, the inputs are only read for module SW_ButtonsOFF2. It is a very safe way to read a few inputs.

Then their values are controlled. According to the schematic of the LCD shield all the pushbuttons hold a resistor to VDD. If all the buttons are open the corresponding value will be %11111111 or 255 in decimal.

If ButtonsVirtualPORT = 255 **Then**

NextStateDelay(30,IncState) *' Delay 30mS to exit the state.*

EndIf

It verifies the value of the virtual port containing all the input bits. If all the pushbuttons are up, it goes to the next state with a delay of 30 ms. The other option, that is never contemplated, is to have a broken button stuck. In this case the program would be locked within an infinite loop in the SM.

To avoid this problem, and as an example every SM loop is counted and after passing 5 seconds a fault is generated. For that we need a little delay between readings.

StartDelayMS(30)

It is the code that will be used in 2 different modules.

CheckButtonsFail_Sub:

Inc ButtonsOffCounter

If ButtonsOffCounter = 167 **Then** *' <= 167 x 30mS = 5 Seconds*

ButtonsOffCounter = 0

CLL(2)

If ButtonsVirtualPORT.0 = 0 **Then**

ButtonsFail = 1

ElseIf ButtonsVirtualPORT.1 = 0 **Then**

ButtonsFail = 2

ElseIf ButtonsVirtualPORT.2 = 0 **Then**

ButtonsFail = 3

ElseIf ButtonsVirtualPORT.3 = 0 **Then**

ButtonsFail = 4

EndIf

NextState(SM_ReadTimeDate) *' Come back to Main Module.*

EndIf

Return



STATE09: (Read all Buttons)

Step 2: Read the pushbuttons

With the help of a counter (**KEYCounter**), a debounce system is built; Count 4 equal readings to generate a correct answer. Pressing a key loads a specific value into the "AllButtons" variable.

```
If F_SW1 = 0 Then      ' Check Button 1.  
    AllButtons = 1  
    IncState()
```

After reading a key, it goes to the next state.

Reading & checking the Buttons.

```
ReadButtonsPORT_Sub:  
    ANSEL = 0                      ' PORTA digital  
    TRISA = TRISA | %00001111    ' SW0 to F_SW3 for input.  
ReadButtons2PORT_Sub:  
    If ButtonFail > 0 Then  
        SetBit AllButtonsBitsDefects, ButtonFail - 1 'Disable PushButton  
    EndIf  
    ' All Buttons failed are saved in the AllButtonsBitsDefects buffer.  
    ButtonsVirtualPORT = PORTA | AllButtonsBitsDefects 'Read PushButtons  
    Return
```

SM command: CheckTimeOut()

This SM command checks if the Delay Time Out is finished and loads the new destination for the SM.

STATE10: (Check if all Buttons are OFF2)

Step 3: Wait for all push buttons to be open.

It is the same routine as step 1 with a small difference. Once the key has been lifted and given as valid, the SM is sent to a generic destination.

```
ReturnIndState(0) ' Load the Return State defined by the caller.
```

The SM is sent to the module that requested execute state 8 (Step 1). It could be any program module. Indirect addressing is used. The **0** means that there is not a delay.

The same code controls the fault of the pushbutton (pushbutton always pressed).

STATE11: (Get the value of Day of Week for calendar)



This is a menu to determine the day of the week.

Depending on the key pressed, a specific function is executed, incrementing, decrementing, validating the value or reading the keyboard.

In order to display a value at the beginning of the routine, it is necessary to send some parameters compatible with the format, for example:

```
AllButtons = 3           ' Like the Button 3 pressed (+)
VDayOfWeek = 0           ' Initialise parameter.
StartTimeOut(SM_ReadTimeDate,10) ' Start a Time Out for 10 seconds
NextState(SM_GetWeek)    ' Start configuration.
```

And you can read the word "**Sunday**" on the LCD.

When you go to another menu, you have to start a 10-second TimeOut with the command:

```
StartTimeOut(SM_ReadTimeDate,10) ' Start a Time Out for 10 seconds
```

STATE12: (Get the value of Day)

It is exactly the same structure as the previous one for Day.

STATE13: (Get the value of Month)

It is exactly the same structure as the previous one for Month.

STATE14: (Get the value of Year)

It is exactly the same structure as the previous one for Year.

STATE15: (Get the value of Hour)

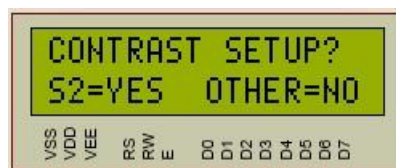
It is exactly the same structure as the previous one for Hour.

STATE16: (Get the value of Minute)

It is exactly the same structure as the previous one for Minute.

As previously determined, the date and time or an alarm will be written.

STATE17: (Contrast Setup)



This is the main menu for adjusting the contrast of the ST7036 LCD. You can see that there is a response different from the TimeOut as it comes from the Reset or the main menu. If key 1 has been pressed it goes to the setting menu.

```
If F_MenuContrastFromReset = 1 Then
  StartTimeOut(SM_AskForConfig,10) ' Start a Time Out
Else
  StartTimeOut(SM_ReadTimeDate,10) ' Start a Time Out
EndIf
```


STATE18: (Adjust the Contrast)

Adjust the contrast value in real time by writing the value on the LCD to see the contrast variation.



And at the end you press key 2 and go to another menu to record the value in the EEPROM.

STATE19: (Save the Contrast to eeprom)

In this new menu you can choose whether or not to record the contrast value in the EEPROM.

STATE20: (General Alarm Menu)



This menu allows for enabling alarm 1 or 2 and to go to the corresponding module to set the parameters.

STATE21: (Save Alarm parameters)

This menu allows a user to save the alarm parameters to the EEPROM memory.

STATE22: (Alarm Warning)

When an alarm is triggered from the SM2 in the background it goes directly to this module. To reset this alarm pressing any button is enough. As the alarms are routed twice, a subroutine has been generated.



```
ClearAlarms_Sub:
AllButtons = 0
If F_Alarm1 = 1 Then
    F_Alarm1 = 0
    EWrite EEADR1_Alarm1Avaible,[$FF] ' Erase the Alarm1(Not available)
    F_Alarm1Avaible = 0
    F_EnableAlarm1 = 0
EndIf
If F_Alarm2 = 1 Then
```

```

        F_Alarm2 = 0
        F_EnableAlarm2 = 0
    EndIf
    StopToggleBuzzer()
    Low bAlarm1Out          ' Clear the Alarm output
    Return

```

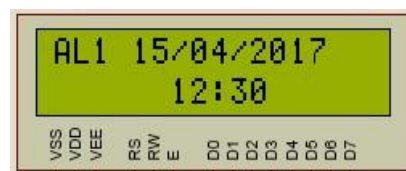
STATE23: (Display Alarm 2)

The status of the alarm 2 is displayed and the daily alarm is activated.
It is done by pressing push button number 3 in the main state “SM_ReadTimeDate”.



STATE24: (Display Alarm 1)

The status of the alarm 1 is displayed and the general alarm is activated.
It is done by pressing push button number 3 in the main state “SM_ReadTimeDate”.



STATE25: (Disable the Alarms)

This menu is called by the pushbutton 4 from the “SM_ReadTimeDate” and allows a user to disable alarms 1 or 2 individually.



STATE26: (Check the TC74)

This code read the status of the TC74 temperature sensor.

STATE27: (TC74 NACK)

This code organizes the NACK error of the TC74 sensor. Press any button to check the system again.

STATE MACHINE 2: (Scheduler: SM2 Scheduler00.inc)

SM2 Scheduler:

The task value, corresponding to the SM2 status number, is loaded in the SM2StateIndex variable that the Scheduler uses to go to the corresponding state.

Security of Scheduler SM2:

In case the programmer enters new states and does not update the **SM2_LABELS_LIST**, an error will be generated in the terminal: **"SM2 STATE OUT OF RANGE:"** and the status number, during the execution of the program.

All states of the State Machine number 2 are running the Tasks.

SM2 STATE00: (Read Time of the DS1307 calendar/Clock) [SM2_ReadTime]

The time is read, if the communication is correct, the next task [SM2_PrintLCDLine2] is loaded, otherwise an error is indicated by a flag (F_ErrorNACK_DS1307).

SM2 STATE01: (Print Strings on the LCD line 1) [SM2_PrintLCDLine1]

This task sends the LCD some information as they arrive:

- Failure of a button.
- Day, Month, Year.
- Alarm enabled.
- Month name or temperature.

The date and temperature are printed every minute.

Alarms, if activated, are controlled every minute

The error message of a fault is printed if it is necessary.

The programmed alarm indication is printed on the right side of the LCD.



SM2 STATE02: (Print Strings on the LCD line 2) [SM2_PrintLCDLine2]

This task sends the LCD some information as it arrives:

- Hour, Minute, Second.
- Day of the Week or ADC result in Volts.

The line is printed every second.

The ADC result is printed on the right side of the line 2.



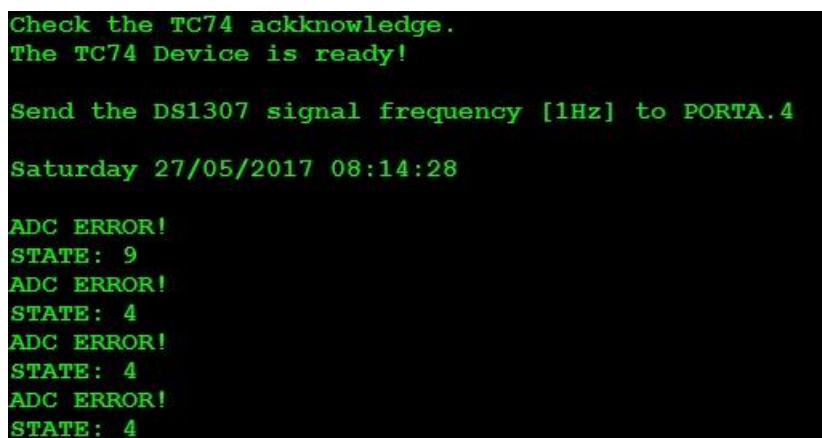
SM2_STATE03: (Check the Alarms) [SM2_CheckAlarmState]

This task checks the status of alarms every minute and resends a flag to the main program if an alarm has been detected.

SM2_STATE04: (Calculate the ADC values) [SM2_ADCCoverage]

Checks if the synchronization with the ADC reading is correct, otherwise the new value is rejected.

```
Inc ADCCounterOld
If ADCCounter <> ADCCounterOld Then
    HRSLStrg(TXT2,1)      ' Send "ADC ERROR!" to the Terminal.
    ADCCounter = 0
    ADCCounterOld = 0
    GoTo SM2_STATE04EXIT ' Don't calculate/print the erroneous value.
EndIf
ADCCounterOld = ADCCounter
```



```
Check the TC74 acknowledge.
The TC74 Device is ready!

Send the DS1307 signal frequency [1Hz] to PORTA.4

Saturday 27/05/2017 08:14:28

ADC ERROR!
STATE: 9
ADC ERROR!
STATE: 4
ADC ERROR!
STATE: 4
ADC ERROR!
STATE: 4
```

<= Error from SM2_STATE09
because the code execution time
is too long for the ADC timing.

Proteus simulation terminal

This task receives the read value of the ADC (ADC_ResultW) which is the equivalent of ADRESH / ADRESL. This value is inserted into a circular array of Word variables (NumberADCcavesages. Then in each ADC reading a new value is added to the array. And then the average of all values is calculated. This system needs more calculation but it is the best one that I have been able to prove.

To reduce the calculation time I have removed a few lines of repeated code.

Circular Buffer:

```
ADC_TrueAverage[TrueAverageHead] = ADC_ResultW ' Load the ADC value
Inc TrueAverageHead
TrueAverageHead = TrueAverageHead & (NumberADCcavesages - 1)
```

Averages:

I have tried several moving average system but I have always obtained some variations in the results (imprecision). The best system is to record each value in a circular array and perform the average at each reading. In this case you need to use the Maths32 compiler that takes a lot of time. The number of averages must be a power of 2 to simplify the circular buffer, 2, 4, 8, 16, 32 & 64.

```

AverageLoop = 0
ADCToPrint = 0
Repeat
    ADC_Average = ADC_TrueAverage[AverageLoop]
    ADCToPrint = ADCToPrint + ADC_Average <= Averages
    Inc AverageLoop
Until AverageLoop = NumberADCaverages

```

Using only a maximum of 64 averages you can use a variable Word (ADCToPrint) saving calculation time. (example for 3,3V power supply Amicus18 board)

```

ADC_DecimalResult = 1023 * NumberADCaverages <= Averages
ADCResult = (ADCToPrint * 3300) / ADC_DecimalResult
ADC_IntegerResult = ADCResult / 1000
ADC_DecimalResult = ADCResult // 1000

```

Also by checking the asm code, I realized that I could save many bytes by removing lines from the code in assembler. To perform this trick it is necessary that the compiler has already generated the Maths32 code. This is the reason for the false calculation, see the DUMMYCAL label. See the code tricks in SM2_State04.

SM2 STATE05: (Set the TC74 to Normal Mode) [SM2_TC74SetNormalMode]

The temperature sensor I2C TC74 is not an easy circuit to use, it has its difficulties. Once your exact protocol is known, this sensor works very well.

Before reading the TC74 sensor temperature, the sensor needs to be reconnected. This sensor usually switches to a standby state when there is noise on the I2C bus, it does not recognize the bus protocol or if it has been given the command to disconnect. This is the case because the I²C is also in dialog with the DS1307 clock chip. (See my article in the WIKI). For that it is disconnected after reading. Then an order is sent to the TC74 to move to a working state (Normal Mode). Note that the command is used:

SM2_NextStateDelay(250,SM2_TC74Read).

The TC74 sensor needs 240 ms to go from standby to read status. The state machine will load this new task after a time of 250 ms. At this time, it would be better not to read another sensor on the I2C bus that could alter the new status of the TC74.

This command from the TC74 library, which I wrote for the WIKI, has been modified to be compatible with the state machine.

SM2 STATE06: (Read the Temperature Sensor) [SM2_TC74Read]

Then the sensor temperature and bus control (ReturnACK) are read. The temperature and polarity have already been calculated by this command.

SM2 STATE07: (Put the Sensor in Standby Mode) [SM2_TC74Close]

The TC74 sensor is of the SMT type, and is mounted very close to the LCD whose BackLight heats it. Also it has a calendar by itself (see the datatsheet). A disable command is sent.

SM2_STATE08: (Read Date of the DS1307 calendar/Clock) [SM2_ReadDate]

The date is reached with the command: **DS1307_ReadDate**(ReturnACK) and again always verifying the validity of the I2C communication with "ReturnACK". Since the code is not in the first line of the module.



SM2_STATE09: (Send Time/Date to Terminal) [SM2_Data1ToTerminal]

Every minute, the date is sent to the terminal. Sending messages takes a lot of time due to UART communication. In this case so as not to retrace or lose the data of the ADC reading, the data is divided into 3 parts, recharging the same task with the counter trick. When this task is completed, an order is sent to the SM2_Data2ToTerminal task. Another solution would be to send the data at the speed of 230400 Bauds. This solution has been tested on the Amicus18 board and works well.

SM2_STATE10: (Send ADC to Terminal) [SM2_Data2ToTerminal]

Each minute, the value of the ADC is sent to the terminal.

SM2_STATE11: (Send temperature to Terminal) [SM2_TC74TempToTerminal]

Every minute + 30 seconds, the temperature is sent to the terminal. Sending messages takes a lot of time due to UART communication. In this case not to delay or lose the data of the reading of the ADC, the message is divided into 2 parts, recharging the same task with the counter trick.



AMICUS18 Board running at 80MHz:

I have been able to run the program with the Amicus18 board and the PIC18F25K20 at 80MHz with ADC readings at 333Hz (every 3mS). The calculation of Maths is done faster which allows more time for other tasks. It would be opportune to eliminate delay due to the LCD with 4-bit bus and I2C (100KHz) communications. It would be much better to use an 8-bit LCD and SPI communications. You could also configure the UART for 230400 baud. This way you could try to operate the ADC at higher frequencies. If you did not have to use the terminal (UART), which has been made for the demo, the benefits would be greater. It has not been tested.

The Amicus18 board prepared for 80 MHz with the guidelines given in a previous chapter works wonderfully. It is the ideal configuration for this state machine. I recommend using this configuration for the whole series PIC18FxxK20, using my bootloader LSM V4.1 for Xtal of 80MHz. In the "Bootloader" folder you will find different bootloaders for 80MHz.

CONCLUSION

The multitasking states machine is very easy to use.

I have tried to explain all the tricks that can be done with my state machine to carry out a project. Surely there will be more to discover, it will depend on you and your imagination. I think it is easier to program with this system with projects that allow it, of course.

I wish you all the best in your projects.

Enjoy the State Machine Multitasking System!

State Machine Part5.

Alberto Freixanet

05 June 2017



www.protonbasic.co.uk

Powered by Proton Development Suite® Compiler of Crownhill Associates Limited©

STATE MACHINE PART5

A Clock Calendar full project with the DS1307 & the TC74

A simple multi-tasking System

PIC®, MPLAB®, PICKIT3® and I/O
Proton Development Suite® or I/O



AmiPIC18 LCD Shield

ed and written by Alberto Freixanet.
ent has been edited by John Drew.

eds State Machine.
Structured method.

This article substantially
modifies the operation of the previously described state machine. We will study the new
commands that will allow us to write the tasks very easily.

