



STATE MACHINE PART1

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:

"**State Machine**" is a term used to describe a method for controlling a software controlled process.

References to read:

Application note DS40051c (c) 2003 of Microchip® Technology Inc.

An Introduction to 'State Machine' by Tim Box December 2003, read [more](#).

Using State Machines In Your Designs, read [more](#).

Finite State Machines for PICs, read [more](#).

A Simple State Machine, read [more](#).

State machines ease programming microcontrollers, read [more](#).

Finite-state machine, read [more](#).

Finite state machines, read [more](#).

And more...

Doing a web search would be a good start.

WHAT IS A STATE MACHINE?

According to Wikipedia: "State machine is called a behavior model of a system with inputs and outputs, where the outputs depend not only on the current input signals but also on the previous ones. The state machines are defined as a set of states that serves as an intermediary in this relation of inputs and outputs, causing the input signal history to determine, for each instant, a state for the machine, so that the output depends only state and current entries."

ADVANTAGES OF THE STATE MACHINE

State machines are an integral part of software programming. State machines make the code more efficient, easier to debug and help organise the flow of the program

1. The first advantage of using state machines is that it promotes good firmware design techniques.
2. Improvements and special features can be easily added to the code in later revisions or according to the evolution of the product.
3. Modules can be cut and pasted into other applications quickly and easily.
4. Other developers will be able to understand the code in order to maintain it. The jump number, well commented, can be an index for each of the software modules.
5. Another benefit of state machines is that built-in firmware naturally promotes the writing of modular type code.

STANDARD STATE MACHINE

State machines require a State Variable (SV). The state variable is a pointer that maintains control of the microcontroller's state and directs the flow of the program to the corresponding software module. With the state machine the 'GoTo' instruction has almost ceased to exist.

The State Variable can be modified in software modules (or states) by itself or by an external function.

GENERAL IDEA OF THE PROCESS

When you start deploying an application, think about what states are required for the application to work.

Once this is done, the first state must be identified

Then we must answer the following question:

What condition is needed to get out of this state and what state is next?

Depending on what happens in a particular state, the State Variable is modified with the goal of passing or jumping to the next state.

The implementation of a flow chart is suggested.

Finally you have to create the software modules for each state according to your flow diagram.

Blocking Code issue

If an application is a long/complex one, all the code inside the main loop is delayed, it is very likely that code may overflow, or repeat obsolete data or stop synchronizing with events.

Even if the code is interrupt-driven, it must still work out of the time that is supplied by an application or other module that executes in the polled loop.

The interrupt code must work within a very short period to ensure module execution can be completed within the interrupt to avoid interfering with the Delays in the main program. Interrupt code must be kept very short.

The main code must be broken down into smaller sub-tasks that can be performed from start-to-finish in an acceptable amount of time. Any sub-task should not block the rest of the system. To do this requires breaking up the application's tasks into smaller sub-tasks.

Consequently, the programmer should carefully use the appropriate libraries that do not delay the operation of the SM. In any case, the code of new libraries should be written according to the State Machine.

I propose a State Machine Model that does not use an automatic methodology like Time System Based as RTOS. The division of tasks and sub-tasks must be designed by the programmer. This model could be called a Static State Machine. The structure of a Module may be divided into Transitional Input State, Static State and Transitional Output State.

COMPILER HELP

In a State Machine there may be repetitive code that can be simplified, as a result the user does not have to write them repeatedly, thus avoiding wasted time and causing possible typographical errors when the code is compiled.

For example, using macros, I can replace the code "`Task_State = 1`" with "`NextState (1)`" which translated is equal to "`Go to New State 1`". The compiler would insert the code for you. In the same way "`Goto StateMachineInit`" could be written in the form "`ReturnSM ()`" which returns to the SM. This approach is shorter and easier to remember. Inside these macros you can also add special commands that allow you to make indirect jumps, thus adding new possibilities. (Part2)

Macros allow you to update/modify the SM code without having to retype the master code or the template again.

In Part2, 3 and 4 of this article we will see templates prepared for a simple and a complex State Machine that will facilitate the work of the beginning programmer. We will make them evolve according to our needs.

CONCLUSION

Incorporating this method to your programming style will improve the structure of your code. We will see some examples with templates that can be downloaded from the web for use with the Proton Development Suite® compiler for the Amicus18 board.

Alberto Freixanet
04 April 2017