

Finite State Machines

Making simple work of complex functions

David Gibson, SPLat Controls Pty Ltd Pty Ltd

Introduction

The State Machine, or more accurately, Finite State Machine (FSM), is a device and a technique that allows simple and accurate design of sequential logic and control functions. Whether you are designing computer programs, sequential logic circuits or electronic control systems, using State Machine methods you will be able to make your designs more sophisticated than before, and with ridiculous ease.

This article is intended as an easy to read introduction to the subject, and is based on over 20 years of on-the-job self-teaching. I will introduce you to the basic concepts of state machines from the point of view of engineering or programming. I'll then give you some programming examples using Basic and a low cost micro-PLC (the SPLat programmable controller), and expand the concepts into a simple model for multitasking in programmable controllers. For the hardware minded I've also covered hardware implementation of state machines.

One of the most fascinating things about FSMs is that the very same design techniques can be used for designing Visual Basic programs, logic circuits or firmware for a microcontroller. Many computers and microprocessor chips have, at their hearts, an FSM. In disciplines other than engineering and programming FSM concepts are used for pattern recognition, artificial intelligence studies, language and behavioural psychology. And yet, the basic concepts are easy to understand and immensely powerful.

What is a State Machine?

The idea behind the FSM is that a system such as a machine with electronic controls can only be in a limited (finite) number of states. Consider some simple systems that you encounter every day: a door may be open or closed; a light may be on or off; a light bulb may be on, off or broken; a cassette player may be playing, stopped, rewinding or fast forwarding.

Notice I refer to these things as systems. Perhaps it's a little strange to call a door a system. Not so if you are designing an automatic door controller! You might also wonder at the choice of two states for a light but three states for a light bulb. The point here is that you need to identify all those states of the system that are important for the task at hand. If you are designing an emergency lighting system then the "broken" state of the bulb is of vital importance to you.

Hence, for an automatic door operator "Open" and "Closed" are probably insufficient. In that case it might be appropriate to add "Opening:", "Closing", "Locked" and, heaven forbid, "Stuck".

The first step in any FSM design is to identify the significant states of the system; you need to include all the important states but avoid including unnecessary states. Hence, the door may be "Opening" or "Closing", but "¾ open" or "¼ closed" would most likely be overkill.

The State Transition Diagram

The main design tool for FSMs is the State Transition Diagram. Also called a state diagram or *bubble diagram*, this depicts the relationships between the system states and the *events* that cause the system to change from one state to another. Let's do a very simple example:-

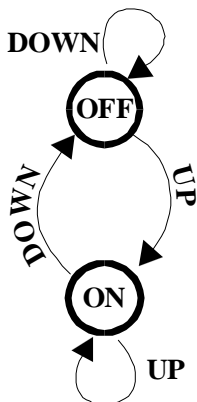


Figure 1

Consider a basic light switch. The bubble diagram for one is shown in Figure 1. It has two states, depicted by circles or bubbles. Those states are, as you have no doubt guessed, on and off. The states are joined by arrows that represents events causing changes of state. The event description is written along-side its arrow. Our light switch has two events: A press up and a press down. Pressing the switch up moves it to the on state, pressing it down moves it to the off state. Notice also that I've shown arrows for a press up when the switch is already on, and for a press down when it's already off. If you consider every possible state/event combination, even those that result in no state change, then the state diagram is *exhaustive*. An exhaustive state diagram, if properly thought out and accurately implemented should give no unexpected responses.

Let's make the light switch a little more interesting. Figure 2 is the state diagram for a push on/push off switch. This switch has only one event input (push) but two states (on and off). What happens when it is pushed depends on the state it's in before its pushed, which in turn depends on its past history. The state diagram accurately predicts the system's response to input events in the context of past history. This "context dependency" is what FSMs are really all

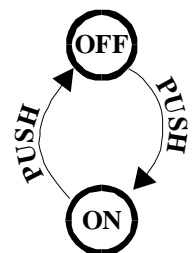


Figure 2

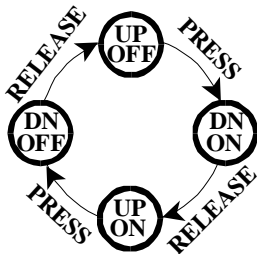
about. Put it another way: The FSM "knows" how to respond to a given input under various circum-

stances.

Static and Transitory input events

Let's investigate in more detail what happens when we operate our push on/push off light switch. When we "push" it we actually do two things: we depress it and then we release it.

Imagine that we design a physical FSM, say an electronic controller, that responds to the static input condition of the button, namely depressed or released, and that it takes the depressed condition to represent a "push" in Figure 2. This system will not be stable: No sooner has it responded to the button being depressed and turned the light on than it must respond again and turn the light off. Hence we will get a very rapidly flashing light.



The problem here is the distinction between a static input condition and a transitory event. A static input condition has duration in time; a transitory event has no duration

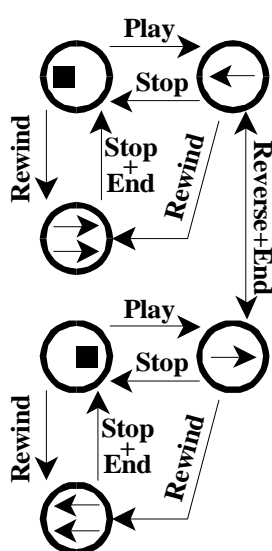
Figure 3 shows the push on/push off FSM re-designed to work with the static input conditions "Depressed" and "Released". It now has 4 states. You could also read the inputs as two different transitory events, namely "Depressing" and "Releasing", meaning the transitions between static input conditions.

Figure 3

An FSM can be designed to work with both static and transitory inputs. However, you must be absolutely clear as to which type you are working with, and design accordingly.

A more interesting example

Lets now look at a slightly less trivial example than a light switch. Consider the control functions for a car cassette player.



This is interesting because it must play the tape in both directions, and also fast wind/rewind the tape in both directions and automatically reverse the direction at the end of the tape. Our hypothetical player mechanism has two motors, one for movement left and one for movement right, and each motor can be driven fast or slow. The details of head selection, engaging the capstan speed control are not considered here, as they are implicit in the control decisions affecting the motors.

The user controls are four momentary pushbutton switches: "Play", "Reverse", "Rewind" and "Stop". Finally, there is an end of tape sensor switch. We will ignore the eject function for simplicity. We will also, for now, ignore the issue of static vs transitory input events.

The control has 6 states: Slow Left, Slow Right, Fast Left, Fast Right, Stop Left and Stop Right. There are two stop states because the system must know which of two possible directions to move in after it has stopped and the user hits play or rewind.

Figure 4

The bubble diagram is shown in Figure 4. I've used symbols rather than words in the state bubbles, to make them more compact. I've also used a double-headed arrow for events that toggle between two states, and placed two alternative events, separated by + signs, next to some arrows. The + sign should be read as "or", meaning either event leads to the state transition.

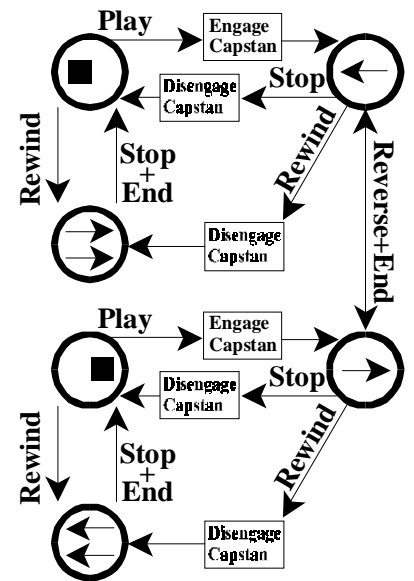


Figure 5

The State-Action-Decision diagram

In real-life it is often necessary for an FSM to perform some action in the process of changing from one state to another. Take our hypothetical tape player, and suppose that the capstan mechanism needs to be engaged whenever we enter a "Play" state and disengaged when we leave a Play state. I depict this by a rectangular box containing a description of the action (Figure 5).

Another trick I use is to include decision-making functions in a state transition. This can reduce the number of states required compared to a conventional state diagram. In effect we ignore certain external events until an event occurs that require immediate action. This eliminates states that might otherwise be needed to track those external events.

Take for example a fragment of the state diagram for a watering and nutrient dosing system for a hydroponic greenhouse. Imagine that we want to add a 10 second squirt of nutrient once per hour if, on the hour, the nutrient level is low. Figure 6 is my FSM for this function. There are two slightly different versions.

In Figure 6a there are two states, which I have simply numbered 0 and 1. State 0 is waiting for the 1 hour interval timer to expire. When the hour expires, we take a decision based on the measured nutrient concentration in the hydroponic solution (assume a sensor that gives a high/low signal). The decision is depicted by a diamond with an arrow leading into it, a question or implied question inside it, and two or more labelled arrows leading out of it, one for each possible answer to the question. If the concentration is low we start the nutrient pump and also a 10 second timer, then go to state 1 and wait for the 10 seconds to time out. At that time we turn off the pump and start the 1 hour timer before going to state 0. If the concentration is ok we simply start a new 1 hour timer cycle. Hence, we get a conditional 10 second squirt of nutrient every hour.

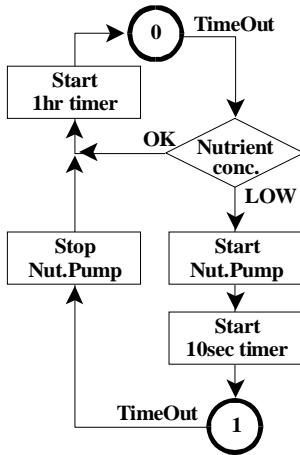


Figure 6a

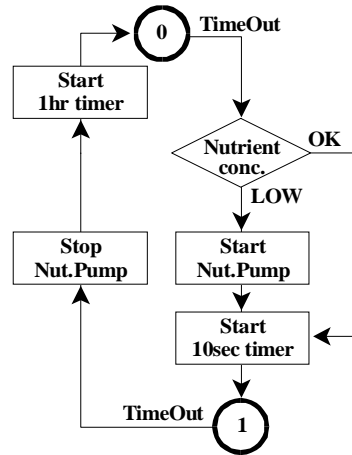


Figure 6b

line out of the decision diamond, the inaccuracy is removed (The one hour timer should now be changed to 1 hour minus 10 seconds). Notice also that in Figure 6b the pump is stopped even if it hasn't been started. The action is redundant, but keeps the diagram simple by minimising the number of paths required.

If you examine Figure 6a carefully you will realise that every time the nutrient pump is run, the total cycle time is 1 hour plus 10 seconds, so the one-hour cycle time will slip by 10 seconds. Probably not important in this application, but in some applications it could be fatal. In Figure 6b, by re-arranging the "OK" line out of the decision diamond, the inaccuracy is removed (The one hour timer should now be changed to 1 hour minus 10 seconds). Notice also that in Figure 6b the pump is stopped even if it hasn't been started. The action is redundant, but keeps the diagram simple by minimising the number of paths required.

If you examine Figure 6a carefully you will realise that every time the nutrient pump is run, the total cycle time is 1 hour plus 10 seconds, so the one-hour cycle time will slip by 10 seconds. Probably not important in this application, but in some applications it could be fatal. In Figure 6b, by re-arranging the "OK" line out of the decision diamond, the inaccuracy is removed (The one hour timer should now be changed to 1 hour minus 10 seconds). Notice also that in Figure 6b the pump is stopped even if it hasn't been started. The action is redundant, but keeps the diagram simple by minimising the number of paths required.

If you are familiar with conventional computer flow charts these diagrams should look familiar. You could consider them as charts with the added bubble symbol. The bubble signifies a place in the program flow where all action is suspended pending one or more external events.

FSM implementation in high level language

Let's now look at how an FSM can be implemented in a high level programming language. Listing 1 shows the Basic-language skeleton of an FSM with two events and two states. The skeleton consists of an outer Select Case structure that selects for the event, and for each event an inner Select Case structure that selects for current state. For each event/state combination there is a code segment that performs whatever actions are specified in the state-action-decision diagram and then sets the state to the new state.

If you want more detail, and a program you can actually run, I've coded the tape drive controller into two versions of Basic: QuickBasic 4.5 under DOS and Visual Basic 3. The full listings are provided in Appendix A (QuickBasic) and Appendix B (VB3).

The QuickBasic version uses keyboard keys to simulate events: P for play, S for stop, R for reverse and W for rewind. The end of tape sensor is the space bar. The program prints what it is doing, including its state, on the screen. You can also run this version under Visual Basic for DOS, and most likely QBasic (which I don't have to try).

The VB3 version has pretty on-screen buttons, but otherwise is much the same. Instructions for getting it going in VB3 are given in the listing.

Both versions use the same coding method for the actual FSM. The event is encoded as a single character ASCII string and then passed to the actual FSM.

Visual Basic for Windows is an inherently event triggered language. If the state number were made global (declared at the module level), then each of the button click event handlers could contain the code for that specific event. This would simplify the example program. I prefer to have the whole state machine inside one subroutine procedure, but that's just a question of personal

```
Select Case Event_Type
Case Event1
  Select Case State_Number
  Case 0
    'Action for event 1, state 0 goes here
    State = {New state after above action}
  Case 1
    'Action for event 1, state 1 goes here
    State = {New state after above action}
  End Select
Case Event2
  Select Case State_Number
  Case 0
    'Action for event 2, state 0 goes here
    State = {New state after above action}
  Case 1
    'Action for event 2, state 1 goes here
    State = {New state after above action}
  End Select
End Select
```

Listing 1

style.

FSM implementation in micro PLC

Now let's look at how the same state machine can be implemented in a programmable industrial controller. This example uses one of the SPLat (Simply Programmed Logic automation tool) boards that I helped design. The SPLat controllers were designed with state machine programming in mind, and contain a pair of instructions (BRANCH/TARGET) designed specifically for that purpose. These are similar to Basic's ON..GOTO, and give an n-way program branch depending on the numeric value of a variable (the state number). This contrasts sharply with "conventional" small PLC's that are programmed in a language that simulates relay circuits, making the power and reliability of FSM designs virtually impossible to achieve.

Listing 2 gives part of the skeleton for an FSM when coded for the SPLat controller. Unlike the Basic program this one has a separate subroutine for each event. This means that elsewhere in the program there will be some code that detects changes in external conditions, say sensor switches turning on or off, and calls the appropriate subroutine. Because the event selection is done in this way, the SPLat program, has no "outer" n-way selection. Listing 2 shows a skeleton event handler for an

```
;Entry point for event A
EventA      Recall State
            Branch      ;ON..GOTO in Basic
            Target EvAState0 ;go here in state 0
            Target EvAState1 ;go here in state 1
EvAState0   ;Do Event A state 0 stuff
            ...
            SetMem State,NewStateA0 ;Update state #
            Return
EvAState1   ;Do Event A state 1 stuff
            ...
            SetMem State,NewStateA1 ;Update state #
            Return
```

Listing 2

1. As discussed above, the SPLat program detects input events and then calls event specific subroutines within the actual FSM code.
2. The Basic programs have as their raw input transitory events, namely keyboard events or visual button clicks. In the real-world case, using SPLat, we have to add some code in the "front end" that detects significant changes in the input switches. Each of the 5 inputs is therefore pre-processed to detect off-to-on transitions. This is done by storing the old input value and comparing it with the new value. If there is an input change, and the new value is on, then we have an event. This is the software equivalent of edge triggering or AC coupling in hardware.

The SPLat program as written takes 351 bytes of program space, or 9% of the available space in a low cost SL88. That might give you some idea of the level of complexity that can be handled by an SL88.

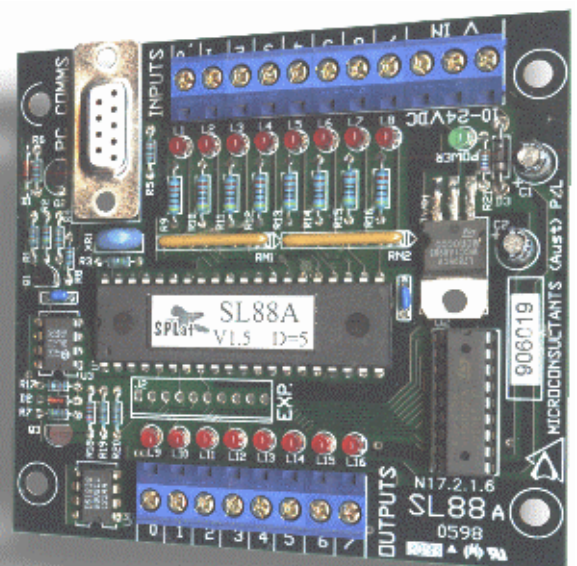
Proactive vs Reactive FSMs

The FSM implementations we have looked at so far have all been *reactive*, meaning they "sit around" passively waiting for an external event to come along. In programmed FSMs (this doesn't apply to hardware designs) it is also possible to write the program the other way around, making it what I call a *proactive* program.

A proactive program is written so that it goes out and looks for the events that, in its current state, are significant. The two styles of program are contrasted in Appendix D (proactive) and Appendix E (reactive). Both the SPLat programs will perform the same function with a similar complexity and program size. With a momentary push button switch connected to a SPLat controller the programs simulate the behaviour of a push on/push off switch. The button is connected to input 0 and controls output 0.

FSM with two states. It has a BRANCH with two TARGETs, one for each state. If the state number is 0 the BRANCH will pick the 0'th Target line and direct the program to program line EvAState0. If the state number is 1 the Branch will pick the next Target line and direct the program to program line EvAState1. At the two separate targets we place the code for handling the specific event/state combination, followed by an update of the state number and a subroutine Return instruction.

The tape player controller is fully coded for SPLat in Appendix C. The program structure differs somewhat from the Basic examples, although the overall principle is identical. The main differences are:



SPLat SL88 microPLC provides 8 inputs, 8 outputs, 8 timers and more. For more information fax us on International +61 3 9773 5091 or visit us at splatco.com

Both version of the program consist of the FSM itself and a “main loop” that drives it. In the case of the reactive FSM, the main loop has to test the input and “push” its current condition into the appropriate part of the FSM. In the case of the proactive FSM, the main loop simply calls the FSM and the FSM itself tests the inputs.

The reactive FSM is well suited to jobs where the “input” event to the FSM is generated inside the program itself, perhaps as a result of some internal decision making. This might for example be where one FSM generates an “output” that becomes the input to the next FSM in line.

The proactive FSM lends itself to jobs where it is directly processing external inputs, i.e. physical inputs on a micro-PLC. Hence in our example the proactive FSM translates to 53 bytes of object code versus 67 bytes for the reactive one.

Often, however, a real-world situation will lie somewhere in between, and at the end of the day it becomes a matter of judgement as to which is the better structure.

It is always possible to convert from one type to the other, or more importantly make FSMs that work in a combined mode. A reactive FSM can be made to simulate proactive behaviour on some inputs (events) by simply placing code to test those events in the main loop of the program.

A proactive FSM can be made to simulate reactive behaviour on some inputs (events). This would be needed where another FSM must trigger a response in the target FSM. What you do is simply have the originating FSM set a memory location (or variable if you are using a high level language) to a True (non-zero) value, and have the target FSM proactively test that memory location. This is called semaphoring, and the memory location is the semaphore (or flag). We say that the initiating FSM “sets” the semaphore and the target FSM tests or reads it. Usually the target FSM will reset the semaphore (set it back to zero) when it has read it, so it will only “process” it once. The originating FSM can then test the semaphore to see when it gets reset, and thus know it has been acted upon. This is called “handshaking”.

Counters, extended timers

Suppose we want to modify our push on/push off FSM so that when the light is on it takes 3 pushes of the button to turn it off. Figure 8 is the FSM for doing this, drawn “shorthand” to save space. The FSM has 8 states, because each press (depress + release) requires two states.

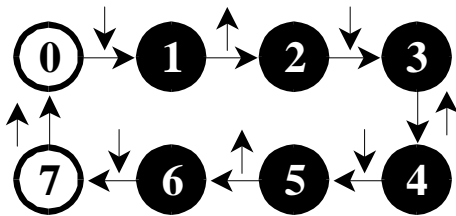


Figure 8

Suppose we wanted it to take 5 pushes, or 50 pushes to turn the light off. The state diagram would take a whole page and the program would become huge. Fortunately there’s a better way. Instead of using lots of states to do the counting we can use a memory variable. In Figure 9, memory variable N is used to count the number of presses while the light is on. It is initialised to 3 during the transition from state 0 to state 1. On each subsequent push (states 2 and 3), the counter is decremented and tested for zero. Depending on the outcome of the decrement/test we decide whether to turn the light off and revert to state 0, or to leave the light on and go back to state 2. This program could be extended to the capacity of a memory variable (255 counts in the case of SPLat) with the same FSM.

The same trick can be used to count other kinds of events, such as semaphore inputs from other FSMs, or timeouts of a timer. In some systems a timer expiring may lead to a semaphore being set for FSMs to process. For instance, in the main loop of your program you might have a section that tests a 1 second timer. If the timer has expired it sets a semaphore and restarts the timer. That semaphore becomes a 1-second heartbeat signal that can be used by several FSMs throughout the program. Each FSM counts the heartbeats independently, and at the same time each can each one be generating complex sequences of timed and/or untimed events.

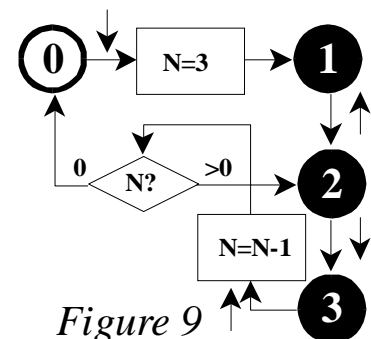


Figure 9

A common application for this technique is for protective time-outs. Say an FSM at some point initiates an external event, such as turning on a solenoid valve. It then looks for a completion signal from a limit switch, but also sets a 5 second time limit on the completion. If the completion signal fails to appear on time, the FSM goes into an alarm state and takes some corrective action. With the methods I’ve been showing you, you should be able to program such a function quite simply.

FSM as a task

Imagine, if you will, a program consisting of several proactive FSMs and a main loop that maintains a heartbeat timer and calls the FSMs in turn. The FSMs communicate with each other using semaphores. You now have the skeleton of an ex-

tremely powerful control system design. Such a system can do several things seemingly at once, handling various aspects of, for instance, a complex machine with ease, elegance and precision.

Within this kind of structure I sometimes think of the FSMs as *tasks*. By this I mean that each FSM is given the task of looking after some aspect of the overall program, such as scanning a keypad, operating a beeper when “asked” (sent a semaphore), driving a motor between its home position and a target point, or whatever. Each task can be written to “know” only about its own area of responsibility, communicating with other tasks only via semaphores and knowing nothing of how other tasks work. Some tasks may have only one state (in which case they don’t need Branch/Target instructions), others may have twenty or more. Imagine each task as a little gnome who does some part of his job, makes a note of where he’s up to (saves his state number), and then goes to sleep (executes a Return). Next time he wakes up he says “Now, where was I?” (retrieves his state number) and carries on from where he left off. When he’s done the next little job, or run out of things to do, he goes back to sleep.

I call the “going to sleep” and “waking up” of a task “Suspend” and “Resume”. A task will suspend itself when it has nothing to do, and gets resumed by the main loop some time later. Correction: A task **MUST** suspend itself (execute a Return instruction). It’s only when a task suspends itself that the main loop gets a chance to call (or “run”) the next task.

This view of the system I’m describing is called “non-pre-emptive multitasking”. Non-pre-emptive means that the operating system, which in our case is represented by the main loop, cannot interrupt a task that refuses to suspend itself. Windows 3.x is such a system. The alternative is a pre-emptive multitasking system, where the operating system is capable of pre-empting tasks (A bit like knocking our gnome on the head, poor fellow). Windows 95 is such a system.

It is possible to create quite sophisticated controls with simple, non-pre-emptive multitasking. Just remember the one golden rule, that a task may not have a tight loop without a suspend (Return) inside the loop. For example, the code in listing 3 is **bad**.

```
Wait      Input      6      ;test limit switch
          GoIfZ      Wait      ;keep testing if not on
HitLimit  . . .      ;here when we hit the limit switch
```

Listing 3. How not to do it!

If the limit switch *never* gets actuated your program will sit in the loop forever, waiting for it, and no other tasks will get run (which could be disastrous). Instead you should use something like the code in listing 4. Here the task suspends (executes a return) if the switch is not activated, so it will resume at the same state number, and perform the same test, next time it is called by the main loop.

```
WaitState Input      6      ;test limit switch
          RetIfZ      ;suspend if we've not yet hit the limit switch
HitLimit  . . .      ;here when we hit the limit switch
```

Listing 4. Can't hang up the whole system

FSM implementation in hardware

We have seen two or three software implementations of the same FSM function. Now lets see how we could also go about making the same FSM in hardware, i.e. using electronic circuitry. (This section is very hardware-intensive. If your interest is purely software, you can safely skip it.)

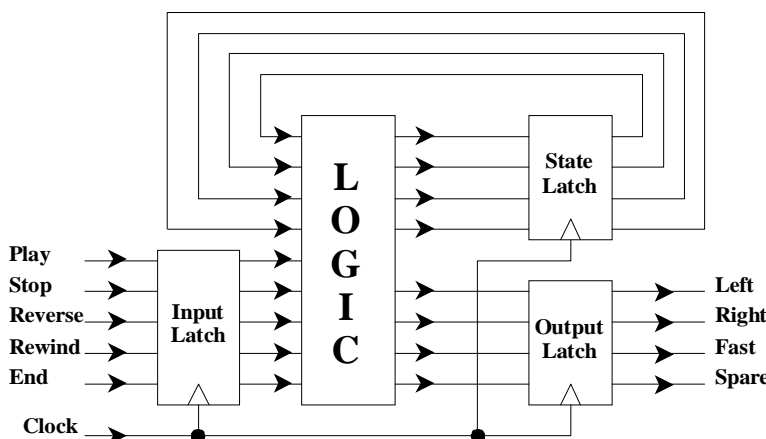


Figure 7

Figure 7 shows the general outline of a hardware implementation. It consists of a logic block and three latches. The input latch freezes the inputs. The state latch stores the state information. The output latch freezes the outputs. All latches are clocked by the same system clock.

The inputs to the logic block are the current state number, fed back via the state latch, and the current inputs that have been frozen by the input latch. The logic block determines the new state number and output pattern as a function of its inputs.

The reason the latches are needed is that the logic block, no matter how well designed, will not respond instantly to input changes. Even worse, when an input changes there is always a risk that the logic block outputs will not change directly to the correct new pattern, but will go through one or more intermediate patterns. Because there is feedback of the state number from the output of the logic block back to its input, if spurious output patterns were allowed back into the inputs the system could become entirely unpredictable.

Imagine the clock frequency is 1kHz, and that it takes the logic block 100µS to settle after its inputs change. At some point in time a clock pulse freezes the inputs to the logic block. For the next 100µS the output of the logic block is unpredictable, but it then settles down to its correct value. 900µS later the new values are clocked into the state and output latches at the same time as new input values are clocked into the input latch.

In this circuit I've allocated 4 lines (bits) to the state number. This is enough to represent 16 (2^4) states. Our tape controller only needs 6 states, which can be done in 3 bits ($2^3=8$). Why the extra bit?

The reason is that the input switches provide static, not transitory inputs. Our hardware design must take that into account. Remember the push on-push off FSM in Figures 2 and 3? Figure 2 required transitory inputs while Figure 3, with twice as many states could handle static inputs. By providing one extra state bit I've allowed for the doubling of the number of states required to handle static inputs.

Now, what goes inside the logic block? The purpose of the logic block is to generate, from every possible combination of current state and input, the correct new state number and outputs. This can be done either by a whole mess of logic gates, with a PLD (Programmable Logic Device) or with a lookup table in an EPROM.

EPROMs are well suited to the task, particularly for FSMs with a large number of inputs and a small number of outputs. The inputs are the address lines and the outputs are the data lines. For example, a 512K bit (64K byte) EPROM has 16 address lines and 8 data lines. 16 bit wide EPROMs will allow a different balance between input and output counts. Working out the EPROM pattern can be laborious for large FSMs. Some PLD design tools include facilities for direct pattern generation from state machine equations. In the past I have written programs in Basic that implement a given FSM, then have a "front end" to generate all possible input and state combinations and capture the resulting outputs and encode them to a file that can be burned into an EPROM.

PLDs are an attractive alternative. Simple PLDs consist of 8 or 10 output latches driven by a large number of configurable logic gates, and also have internal feedback paths in addition to 10 or more inputs. For small FSMs it's possible to determine the equations for each of the outputs (state and system outputs) using conventional logic analysis methods. However, it is far preferable to use a PLD compiler tool that supports state machine descriptions, both for the ease of initial design, because they allow changes to be made easily and most importantly because they leave you free to concentrate on the functionality of your design, not the mechanics.

Conclusion

Finite State Machines (FSM) provide an immensely powerful method of implementing sequential (past history dependent) control schemes both in hardware and software. By extending the concept of the FSM, and introducing reactive and proactive FSMs, I have shown how sophisticated multitasking controls can be programmed simply even into low cost programmable controllers such as Microconsultants' SPLat range.

Copyright Notice

This article is copyright property of SPLat Controls Pty Ltd. It may be reproduced in a maximum of 50 copies for non-commercial purposes, providing it is reproduced in its entirety including all references to SPLat's products. For mass circulation rights please contact SPLat Controls..
©1997-2000 SPLat Controls Pty Ltd

SPLat: World's most cost effective microPLCs

Notable features of the SPLat System include:

- **A simple, easy to learn programming language that is designed especially to support time/sequential control functions and Finite State Machines.**
- **A Windows based programming, development and commissioning package that allows interactive testing and includes an extensive programming tutorial.**
- **Off-the shelf products that will slash the cost of machine controls in the 1 to 1000/year range.**
- **Ability to make fully customized user programmable SPLat controllers for 250 to 50K/year range, bringing your product huge direct and indirect savings plus enhanced features.**

Should *your* product contain a SPLat?

For more information please contact us at

SPLat Controls Pty Ltd
2/12 Peninsula Blvd
Seaford VIC 3198, Australia

Ph +61 3 9773 5082, Fx +61 3 9773 5091
email dgibson@splatco.com
Or visit our web site splatco.com

(Address current June2000)

Appendix A: Tape control FSM in Quick Basic 4.5

If this program is saved to disk in ASCII format under the file name TAPE.BAS, you will be able to simply load it into QuickBasic and run it. If you are reading this on a Windows based web browser, you should be able to simple highlight the text, then copy and paste it into Notepad and from there save it as TAPE.BAS. To load TAPE.BAS under VBDOS, start up VBDOS, select File|Add File and browse to TAPE.BAS.

```
'Sample coding of tape player controller FSM in QuickBasic
```

```
'Main loop just looks for a key input to simulate  
'a real world event. When a key is found the  
'actual FSM is called as a Subroutine.
```

```
CLS
```

```
'===== The main loop =====
```

```
DO
```

```
  K$ = UCASE$(INKEY$)
```

```
  IF K$ <> "" THEN GOSUB TapeFSM
```

```
LOOP 'forever
```

```
'===== The FSM =====
```

```
'Input events are represented by key codes
```

```
'(characters) in K$. The codes are:
```

```
' P = Play button
```

```
' S = Stop button
```

```
' R = Reverse button
```

```
' W = Rewind button
```

```
' Space bar = end of tape sensor
```

```
TapeFSM:
```

```
  SELECT CASE K$
```

```
  CASE "P" 'Play
```

```
    PRINT "Play ";
```

```
    SELECT CASE State
```

```
    CASE 0 'Stop left
```

```
      PRINT "1 Motor slow left";
```

```
      State = 1
```

```
    CASE 3 'Stop right
```

```
      PRINT "4 Motor slow Right";
```

```
      State = 4
```

```
    END SELECT
```

```
  CASE "S" 'Stop
```

```
    PRINT "Stop ";
```

```
    SELECT CASE State
```

```
    CASE 1 'play left
```

```
      PRINT "0 Stop left";
```

```
      State = 0
```

```
    CASE 2 'rewind right
```

```
      PRINT "0 Stop left";
```

```
      State = 0
```

```
    CASE 4 'play right
```

```
      PRINT "3 Stop right";
```

```
      State = 3
```

```
    CASE 5 'rewind left
```

```
      PRINT "3 Stop right";
```

```
      State = 3
```

```
    END SELECT
```

```
  CASE "R" 'Reverse
```

```
    PRINT "Reverse ";
```

```
    SELECT CASE State
```

```
    CASE 1 'play left
```

```
      PRINT "4 Motor slow Right";
```

```
      State = 4
```

```
    CASE 4 'play right
```

```
      PRINT "1 Motor slow left";
```

```
      State = 1
```

```
END SELECT
CASE "W"      'Rewind
  PRINT "Rewind ";
  SELECT CASE State
  CASE 0  'Stop left
    PRINT "2 Motor fast Right";
    State = 2
  CASE 1  'play left
    PRINT "2 Motor fast Right";
    State = 2
  CASE 3  'Stop right
    PRINT "5 Motor fast Left";
    State = 5
  CASE 4  'play right
    PRINT "5 Motor fast Left";
    State = 5
  END SELECT
CASE " "      'End sensor
  PRINT "End ";
  SELECT CASE State
  CASE 1  'play left
    PRINT "4 Motor slow Right";
    State = 4
  CASE 2  'rewind right
    PRINT "0 Stop left";
    State = 0
  CASE 4  'play right
    PRINT "1 Motor slow left";
    State = 1
  CASE 5  'rewind left
    PRINT "3 Stop right";
    State = 3
  END SELECT
CASE ELSE
  BEEP
END SELECT
PRINT
RETURN
```

Appendix B: Tape control FSM in Visual Basic 3

Copy the lines between >>> and <<< to a file called TAPE.MAK and all following lines to a file called TAPE.FRM in the same Directory. The run Visual Basic 3 and Open Project TAPE,MAK.

Start of FRM file >>>>>>>>>

TAPE.FRM

ProjWinSize=152,402,248,215

ProjWinShow=0

<<<<<<<<<<< End of MAK file. Next line is start of FRM file.

VERSION 2.00

Begin Form Form1

```
Caption           = "Tape control"
ClientHeight      = 876
ClientLeft        = 792
ClientTop         = 1524
ClientWidth       = 3372
Height           = 1260
Left              = 744
LinkTopic         = "Form1"
ScaleHeight       = 876
ScaleWidth        = 3372
Top               = 1188
Width             = 3468
```

Begin CommandButton cmdStop

```
Caption          = "Stop"
Height           = 372
Left             = 2520
TabIndex         = 5
Top              = 0
Width            = 852
```

End

Begin TextBox Text1

```
Height          = 288
Left            = 0
TabIndex        = 4
Text            = "State info"
Top             = 600
Width           = 3372
```

End

Begin CommandButton cmdEOT

```
Caption          = "End of Tape Sensor"
Height           = 252
Left             = 0
TabIndex         = 3
Top              = 360
Width            = 3372
```

End

Begin CommandButton cmdRewind

```
Caption          = "Rewind"
Height           = 372
Left             = 1680
TabIndex         = 2
Top              = 0
Width            = 852
```

End

Begin CommandButton cmdReverse

```
Caption          = "Reverse"
Height           = 372
Left             = 840
TabIndex         = 1
Top              = 0
Width            = 852
```

End

Begin CommandButton cmdPlay

```
Caption          = "Play"
```

```

        Height      = 372
        Left        = 0
        TabIndex    = 0
        Top         = 0
        Width       = 852
    End
End
Option Explicit

Sub cmdEOT_Click ()
    TapeFSM "E"
End Sub

Sub cmdPlay_Click ()
    TapeFSM "P"
End Sub

Sub cmdReverse_Click ()
    TapeFSM "R"
End Sub

Sub cmdRewind_Click ()
    TapeFSM "W"
End Sub

Sub cmdStop_Click ()
    TapeFSM "S"
End Sub

Sub TapeFSM (Cmnd As String)
Static State
'===== The FSM =====
'Input events are represented by key codes
'(characters) in K$. The codes are:
' P   = Play button
' S   = Stop button
' R   = Reverse button
' W   = Rewind button
' Space bar = end of tape sensor

TapeFSM:
    Text1.Text = ""
    Select Case Cmnd
    Case "P"      'Play
        Text1.Text = Text1.Text & "Play      "
        Select Case State
        Case 0    'Stop left
            Text1.Text = Text1.Text & "1 Motor slow left"
            State = 1
        Case 3    'Stop right
            Text1.Text = Text1.Text & "4 Motor slow Right"
            State =
        End Select
    Case "S"      'Stop
        Text1.Text = Text1.Text & "Stop      "
        Select Case
        Case 1    'play left
            Text1.Text = Text1.Text & "0 Stop left"
            State = 0
        Case 2    'rewind right
            Text1.Text = Text1.Text & "0 Stop left"
            State = 0
        Case 4    'play right
            Text1.Text = Text1.Text & "3 Stop right"
            State = 3
        Case 5    'rewind left

```

```

    Text1.Text = Text1.Text & "3 Stop right"
    State = 3
End Select
Case "R"      'Reverse
Text1.Text = Text1.Text & "Reverse "
Select Case
Case 1  'play left
    Text1.Text = Text1.Text & "4 Motor slow Right"
    State = 4
Case 4  'play right
    Text1.Text = Text1.Text & "1 Motor slow left"
    State = 1
Case 5  'rewind left
End Select
Case "W"      'Rewind
Text1.Text = Text1.Text & "Rewind  "
Select Case State
Case 0  'Stop left
    Text1.Text = Text1.Text & "2 Motor fast Right"
    State = 2
Case 1  'play left
    Text1.Text = Text1.Text & "2 Motor fast Right"
    State = 2
Case 3  'Stop right
    Text1.Text = Text1.Text & "5 Motor fast Left"
    State = 5
Case 4  'play right
    Text1.Text = Text1.Text & "5 Motor fast Left"
    State = 5
End Select
Case "E"      'End sensor
Text1.Text = Text1.Text & "End      "
Select Case
Case 1  'play left
    Text1.Text = Text1.Text & "4 Motor slow Right"
    State = 4
Case 2  'rewind right
    Text1.Text = Text1.Text & "0 Stop left"
    State = 0
Case 4  'play right
    Text1.Text = Text1.Text & "1 Motor slow left"
    State = 1
Case 5  'rewind left
    Text1.Text = Text1.Text & "3 Stop right"
    State = 3
End Select
Case Else
    Beep
End Select
End Sub

```

Appendix C: Tape control FSM coded for SPLat micro-PLC

```
;Tape controller FSM for SPLat (Simply Programmed Logic automation tool)
;Assign inputs, outputs and data memory
;Inputs (assume user controls are simple momentary make push button switches):
iPlay      equ      0          ;Play button
iStop      equ      1          ;Stop button
iReverse   equ      2          ;Reverse button
iRewind    equ      3          ;Rewind button
iEnd       equ      4          ;End of tape sensor switch

;Outputs
oLeft      equ      0          ;Drive left motor
oRight     equ      1          ;Drive right motor
oFast      equ      2          ;on for fast, off for slow

;Data memory
State      equ      10         ;State number

;The following memories are used to detect transitions
;in the control inputs by comparing the new condition
;with the remembered old condition.
OldPlay    equ      0          ;Play button
OldStop    equ      1          ;Stop button
OldReverse equ      2          ;Reverse button
OldRewind  equ      3          ;Rewind button
OldEnd     equ      4          ;End of tape sensor

;===== Main Loop =====

;The main loop simply sits constantly looking for a change
;on any input. When it detects a change from off to on, it
;calls the corresponding event in the main FSM.

Loop
    Input      iPlay          ;Read Play button
    Push
    Push
    Push
    Recall     OldPlay
    XOR
    AND
    SWAP
    Store      OldPlay
    Push
    GoIfZ      DonePlay
    Gosub     PlayEvent

DonePlay
    Input      iStop          ;Read Stop button
    Push
    Push
    Push
    Recall     OldStop
    XOR
    AND
    SWAP
    Store      OldStop
    Push
    GoIfZ      DoneStop
    Gosub     StopEvent

DoneStop
    Input      iReverse
```

```

        Recall    OldReverse
        XOR                               ;True if Old<>New
        AND                               ;True if (Old<>New) AND New = True
        SWAP                               ;Get new to X
        Store     OldReverse               ;Update old value
        Push                               ;duplicate processed input value
        GoIfZ     DoneReverse
        Gosub     ReverseEvent
DoneReverse
        Input     iRewind
        Push
        Push
        Push
        Recall    OldRewind
        XOR                               ;True if Old<>New
        AND                               ;True if (Old<>New) AND New = True
        SWAP                               ;Get new to X
        Store     OldRewind               ;Update old value
        Push                               ;duplicate processed input value
        GoIfZ     DoneRewind
        Gosub     RewindEvent
DoneRewind
        Input     iEnd
        Push
        Push
        Push
        Recall    OldEnd
        XOR                               ;True if Old<>New
        AND                               ;True if (Old<>New) AND New = True
        SWAP                               ;Get new to X
        Store     OldEnd                  ;Update old value
        Push                               ;duplicate processed input value
        GoIfZ     DoneEnd
        Gosub     EndEvent
DoneEnd
        Goto Loop                          ;End of main loop

;=====
;The main FSM. This has one entry point as a subroutine
;for each event. In each instance it retrieves the state number,
;does a Branch and ends up at a section of code which is
;unique to each event/state combination.

;The BRANCH instruction uses the number in the X register (which gets
;there via a Recall from memory), to direct the program to one of 'n'
;Target instructions that follow the BRANCH.

PlayEvent    ;Called when Play button is pressed
        Recall    State                ;Retrieve the state number
        Branch                               ;Test number, go to n'th target
        Target     Play0                ;Target (next) instruction at Play0 if State = 0
        Target     Play1                ;Target (next) instruction at Play1 if State = 1
        Target     Play2                ;Target (next) instruction at Play2 if State = 2
        Target     Play3                ;Target (next) instruction at Play3 if State = 3
        Target     Play4                ;Target (next) instruction at Play4 if State = 4
        Target     Play5                ;Target (next) instruction at Play5 if State = 5

Play0        ;Play while stopped left
        Off        oRight
        On         oLeft
        Off        oFast
        LoadX      1                    ;New state number=1
        Store     State                ;Save it for next time
        Return

Play1        ;Play while doing slow left

```

```

Return                ;Do nothing

Play2                ;Play while going fast right
Return                ;Do nothing

Play3                ;Play while stopped right
Off                  oLeft
On                   oRight
Off                  oFast
LoadX                4
Store                State
Return

Play4                ;Play while going slow right
Return                ;Do nothing

Play5                ;Play while going fast left
Return                ;Do nothing

StopEvent
Recall              State
Branch
Target              Stop0
Target              Stop1
Target              Stop2
Target              Stop3
Target              Stop4
Target              Stop5

Stop0                ;Stop while stopped left
Return

Stop1                ;Stop while doing slow left
Off                  oRight
Off                  oLeft
LoadX                0
Store                State
Return

Stop2                ;Stop while going fast right
Off                  oRight
Off                  oLeft
LoadX                0
Store                State
Return

Stop3                ;Stop while stopped right
Return

Stop4                ;Stop while going slow right
Off                  oLeft
Off                  oRight
LoadX                3
Store                State
Return

Stop5                ;Stop while going fast left
Off                  oLeft
Off                  oRight
LoadX                3
Store                State
Return

ReverseEvent
Recall              State
Branch

```

```
Target Reverse0
Target Reverse1
Target Reverse2
Target Reverse3
Target Reverse4
Target Reverse5
```

```
Reverse0 ;Reverse while stopped left
Return
```

```
Reverse1 ;Reverse while doing slow left
Off      oLeft
On       oRight
Off      oFast
LoadX    4
Store    State
Return
```

```
Reverse2 ;Reverse while going fast right
Return
```

```
Reverse3 ;Reverse while stopped right
Return
```

```
Reverse4 ;Reverse while going slow right
Off      oRight
On       oLeft
Off      oFast
LoadX    1
Store    State
Return
```

```
Reverse5 ;Reverse while going fast left
Return
```

```
RewindEvent
```

```
Recall    State
Branch
Target    Rewind0
Target    Rewind1
Target    Rewind2
Target    Rewind3
Target    Rewind4
Target    Rewind5
```

```
Rewind0 ;Rewind while stopped left
Off      oLeft
On       oRight
On       ofast
LoadX    2
Store    State
Return
```

```
Rewind1 ;Rewind while doing slow left
Off      oLeft
On       oRight
On       ofast
LoadX    2
Store    State
Return
```

```
Rewind2 ;Rewind while going fast right
Return
```

```
Rewind3 ;Rewind while stopped right
```

```

Off      oRight
On       oLeft
On       oFast
LoadX    5
Store    State
Return

Rewind4  ;Rewind while going slow right
Off      oRight
On       oLeft
On       oFast
LoadX    5
Store    State
Return

Rewind5  ;Rewind while going fast left
Return

EndEvent
Recall   State
Branch
Target   End0
Target   End1
Target   End2
Target   End3
Target   End4
Target   End5

End0     ;End while stopped left
Return

End1     ;End while doing slow left
Off      oLeft
On       oRight
Off      oFast
LoadX    4
Store    State
Return

End2     ;End while going fast right
Off      oRight
Off      oLeft
LoadX    0
Store    State
Return

End3     ;End while stopped right
Return

End4     ;End while going slow right
Off      oRight
On       oLeft
Off      oFast
LoadX    1
Store    State
Return

End5     ;End while going fast left
Off      oLeft
Off      oRight
LoadX    3
Store    State
Return

```

Appendix D: Push-on, push off light control, proactive version

;This version takes 53 bytes of code when translated.

```
State      EQU      10      ;allocate storage for state number
```

```
Loop      Gosub      PushPush
          Goto      Loop
```

```
;=====
```

```
PushPush Recall      State
```

```
      Branch
      Target      State0
      Target      State1
      Target      State2
      Target      State3
```

```
State0      Input      0      ;Test input
            RetIfZ      ;R/ Not of interest
            On          0      ;Turn on output
            Loadx      1      ;Update state number
            Store      State
            Return
```

```
State1      Input      0      ;Test input
            RetIfNZ     ;R/ Not of interest
            Loadx      2      ;Update state number
            Store      State
            Return
```

```
State2      Input      0      ;Test input
            RetIfZ      ;R/ Not of interest
            Off         0      ;Turn on output
            Loadx      3      ;Update state number
            Store      State
            Return
```

```
State3      Input      0      ;Test input
            RetIfNZ     ;R/ Not of interest
            Loadx      0      ;Update state number
            Store      State
            Return
```

Appendix E: Push-on, push off light control, reactive version

;This version takes 67 bytes of code when translated.

```
State      equ      10          ;allocate storage for state number

Loop       Input    0          ;test input
           GoIfZ    Off        ;G/ input is off
           Gosub    Down       ;Call FSM at "Button down" entry point
           Goto     Loop

Off        Gosub    Up         ;Call FSM at "Button up" entry point
           Goto     Loop

;=====The actual FSM =====
Down       Recall   State
           Branch
           Target   Down0
           Target   Down1
           Target   Down2
           Target   Down3

Down0      On       0
           LoadX    1
           Store    State
           Return

Down1      Return

Down2      Off      0
           LoadX    3
           Store    State
           Return

Down3      Return

Up         Recall   State
           Branch
           Target   Up0
           Target   Up1
           Target   Up2
           Target   Up3

Up0        Return

Up1        LoadX    2
           Store    State
           Return

Up2        Return

Up3        LoadX    0
           Store    State
           Return
```