Finite State Machines

Written and illustrated by James Trevelyan, School of Mechanical Engineering, based on lectures by Peter Kovesi, School of Computer Science and Software Engineering.

The Finite State Machine (FSM) is an abstract concept that helps us analyse a mechatronics control system in a structured way. At the same time it is also a useful design tool that allows us to approach the design in a systematic manner. It is particularly useful for sequential logic and control functions. Finite state machines are widely used in the design of software, logic, PLCs, electronic controls and mechanisms. Software finite state machines find applications in artificial intelligence, pattern recognition, linguistics and even in human behavioural models, though here some people would say that their use is misguided.

Table of Contents

Defining States

A finite state machine is simply a machine that has a finite number of states. We need to contrast this with machines that can have an infinite number of states. For example, a sensor that can read the temperature of a process producing an analog voltage is an example of a machine with an infinite number of states. The number of possible output voltages is infinite. On the other hand, the temperature sensor that only indicates if the temperature is too high or too low has two output states. Many mechatronics systems have elements of both: some parts have infinite numbers of states while other parts operate as finite state machines with a limited number of possible states.

A cassette player, for example, can be in one of several discrete "operating" states:

> Stop: tape is stopped
>
> Play: tape moves slowly past the 'reading' head at constant speed, tape is pressed against the reading head.
>
> Rewind: tape moves quickly backwards, not necessarily at constant speed, tape is not pressed against the reading head.
>
> fast forward: tape moves quickly forwards, not necessarily at constant speed, tape is not pressed against the reading head.
>
> Pause: tape motion in play mode is temporarily stopped, but the tape remains pressed against the reading head.
>
> Record: tape moves slowly past the 'reading' head at constant speed, tape is pressed against the reading head, but the 'write' coil is energized changing the magnetic patterns stored on the tape surface layer. Possibly, a separate 'erase' head is also energized as well.

Certain events cause a machine to transfer from one state to another. For example, when the cassette player is rewinding a tape it will transfer from the "rewind" state to the "stop" state when a sensor indicates that the end of the tape has been reached.

The first step in designing or analysing a finite state machine is to find all the possible states that it can be in. This is not as easy as it seems. An automatic door is a good example. It can be "open" or "closed". However, it does not change between the "open" and "closed" states instantaneously: it passes through some other states to achieve that. If we look closely at a typical automatic door we might observe two intermediate states. From "closed" it transfers to an "opening" state in which the motor is opening the door at a steady speed. Shortly before reaching the fully open position it transfers to an "opening slowly" state in which the motor is running slower so that the door does not slam into the end of its rail. After a short time it transfers into the "open" state and stays there until the "close" button is pressed or perhaps an automatic timer produces a signal starting a similar sequence of states to close the door.

Sometimes it is possible to identify states that do not actually exist or are not required. We call these "redundant" states. It takes practice and it is best to learn by experience.

State Transition Diagram

The second step is to draw a state transition diagram. This diagram shows the relationships between each state. It shows how the machine transfers from one state to another in response to events. It is possible to read the diagram to reveal the sequence of states and events that define the machine's operational

behaviour. A state transition diagram is an invaluable design tool and a means of communicating the control system design to other people.

Let's take a simple light switch as an example to start with. There are two states: "on" and "off". There are two possible events: lifting the switch lever up and pressing the switch lever down by finger movement. (Note that we observe the British and Australian convention: the switch is up in the "off" position and down in the "on" position.)

The state diagram looks like this.

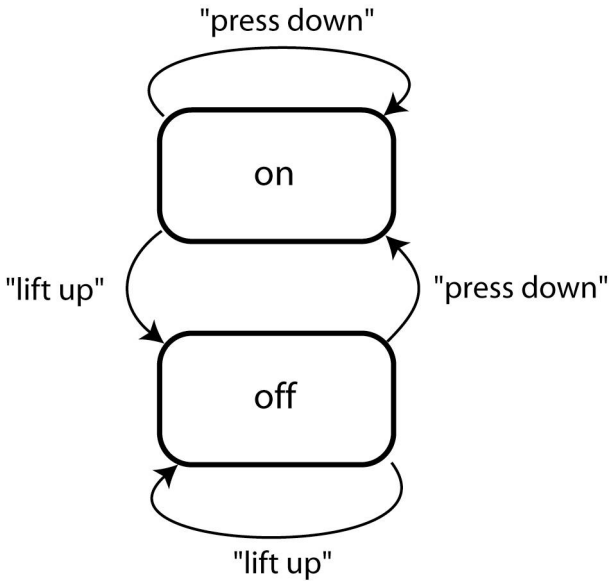

Figure 1: State diagram for simple electric light switch

Notice we have included all possible events in both states. Pressing the switch down in the "on" position produces no state change but we still show an arrow that returns to the "on" state. A state diagram that shows every possible combination of state and event is called an exhaustive state diagram. Most of the time we only show events that actually cause state transitions. If we were to show every possible state transition and event we may end up with a diagram that is far too complicated to read. However, for safety critical systems where reliability is very important we need to analyse every possible event in every state.

Notice also that the term "pressing the switch down" or " lifting the switch up" implies a discrete event when the finger force just overcomes spring pressure and causes the switch suddenly to move to its final position. The electrical state of the switch only changes when the spring pressure has been overcome and the sudden final movement occurs. If the finger exerts insufficient force, the switch lever moves, but not enough to cause a state transition.

Now let's take a different kind of light switch. A pushbutton light switch is pressed to turn the light on and pressed again to turn the light off. Once again the state transition only occurs when mechanical spring pressure has reached a certain critical point. In this example there is only one event: pressing the switch button. Notice that the same event causes a transition in both directions!

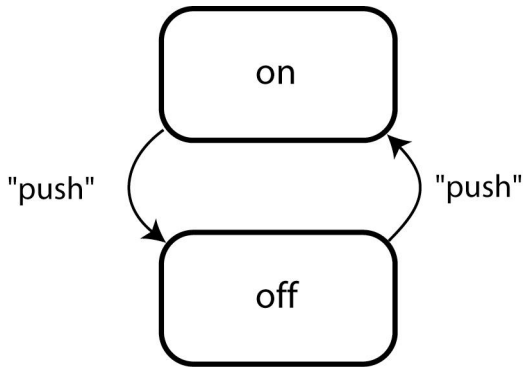

Figure 2: Push button light switch using transient "push" events.

Transient and Persistent Events

Now we need to distinguish between "static" and "transient" events. A transient event happens in an infinitely short instant of time. The static event is persistent for a finite time. Let's look at this simple pushbutton state diagram again assuming that the "push" event exists for a finite period of time. As soon as the push event occurs there is a state transition from "off" to "on". However, the push event still persists. Now that we are in the "on" state the push event causes an immediate transition to the "off" state. Thus, our state machine will enter an endless loop, oscillating from one state to the other, until the end of the "push" event. This is often known as a "race" condition.

We can overcome this problem by introducing additional states. The state diagram immediately below shows how this can be done. We have introduced "off wait" and "on wait" states and introduced a "release" event. The "push" event is always followed by a "release" event.

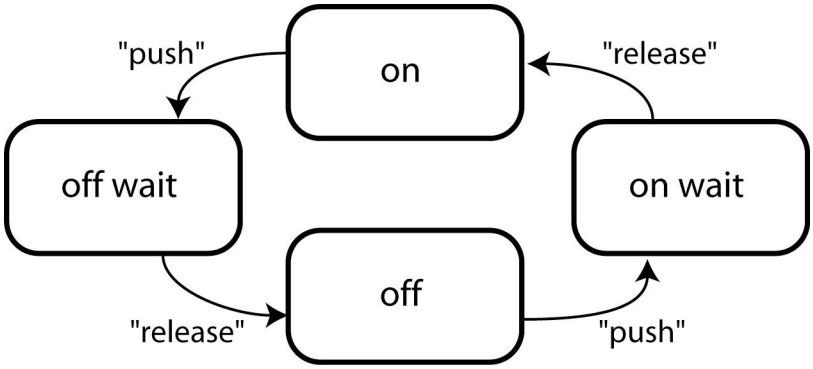

Figure 3: Expanded state diagram for push button light switch
using persistent events "push" and "release".

To avoid this kind of problem we must declare events to be transient or static (persistent or transient). Notice that we can introduced a small "sub-state" diagram that transforms a pair of persistent events such as "push" and "release" into a transient "push once" event. In the diagram below, there is an immediate transition from "on" to "off". The existence of the "on" state is therefore transient, and this becomes the transient event that can be used in the simple state diagram shown in figure 3.

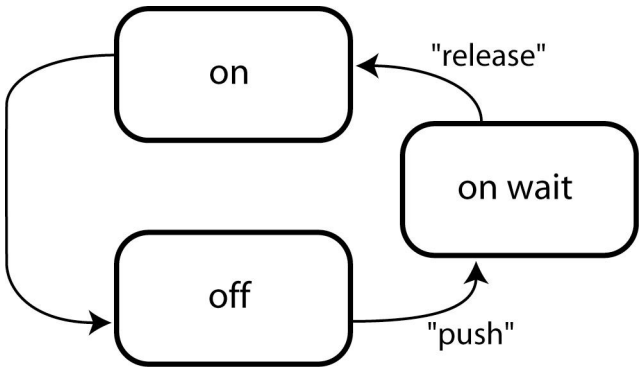

Figure 4: Sub-state diagram that converts persistent "push" and "release" events into a transient event "on" determined by the duration of the "on" state. Since the state will immediately change (unconditionally) from "on" to "off", the "on" state has to be transient.

Exercises 1

Define the state transition diagram for the "close" button on a typical window on a computer screen. On my computer the button changes its appearance as soon as I press the mouse button down with the pointer over the button. However, the window does not close immediately. The window only closes if I release the mouse button with the arrow still over the button. If I drag the pointer away from the button, holding the mouse button down, the "close" button of the window simply changes its appearance back to the "up" position and the window remains open. Draw a state transition diagram to represent this behaviour.

Note that there is no single right answer. A state transition diagram represents one person's way of seeing the design of a finite state machine. Different people can conceive of the same design using different state transition diagrams.

Exercise 2

Dismantle an old pushbutton pen: the type with a button at the end which you press to make the ball point appear at the other end and press again to make it disappear. This is a mechanical example of a two state machine. Draw the state transition diagram and draw sketches to show how the mechanism implements the state transition diagram.

Cassette Player Example

Now, as a final worked example let's return to the cassette player.

We will define the following set of controls for an English-language version:

"Stop", "Play", "Reverse direction", "Rewind", "Fast forward"

The machine starts in the "stop" state. When we press the "play" button (thereby generating a transient "play" event) the machine transfers to the "play" state.

At this stage we have to pause our analysis and realise that there are two possible "play" states. The machine can be playing the tape in the forward direction or in the reverse direction depending on which side of the tape we want to listen to. Therefore we need to define two play states: "play side A" and "play side B". If your cassette player is similar to mine the direction of play depends on the previous play

direction before we pressed the stop button. Therefore, we also need two "stop" states: "stop A" and "stop B" so that the machine can remember which side of the tape it was playing last.

In the same way, the "rewind" state depends on which side of the tape we are playing.

We can resolve this problem in two ways. We can either introduce a second state diagram to define the direction in which we are playing the tape or we can simply have a separate state for each kind of operation when we play different sides of the tape. Both alternatives provide equally valid interpretations of the machine behaviour.

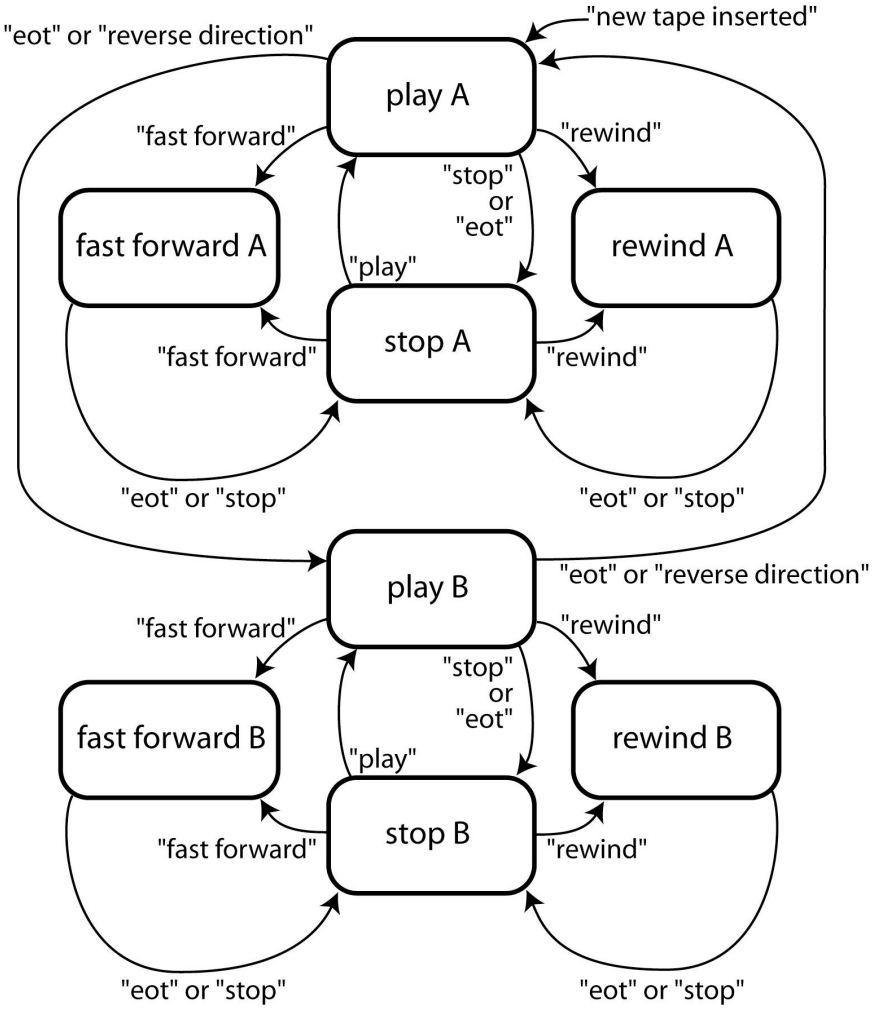

Figure 5: State transition diagram for simple cassette player

Exercise 3

Modify the state transition diagram of figure 3 to allow the "reverse direction" event to change the tape direction while the tape is stopped.

Exercise 4

Modifiy the state transition diagram of figure 3 to allow the "reverse direction" button to change the tape direction *and* start the tape playing in the new direction if it was stopped.

Exercise 5

Examine a real tape cassette player, either video or audio. Determine the operating states and draw a state transition diagram to describe the response of the machine to the different controls (assuming that the button controls generate transient events). If possible, remove the cover and examine the cassette player in its different states and carefully observe how the mechanical parts operate. (If you do not have access to a tape cassette player, perform the same exercise for a CD player.)

Decomposition Techniques

These examples demonstrate that state transition diagrams for relatively simple machines can become quite complex. It is useful to decompose more complex machines into a series of simpler independent state machines, each with its own state transition diagram. This is analogous to the way we decompose software into upper-level modules and lower-level modules to hide complexity.

Timer

Just as an example we can decompose a timer used in many ways in typical state machines.

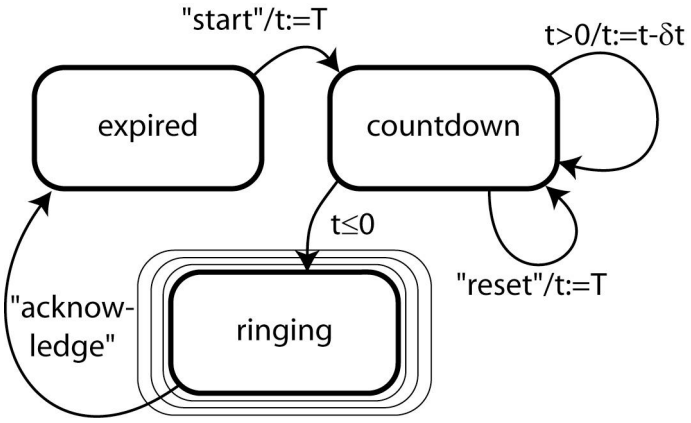

Figure 6: Transition diagram for timer state machine. The machine starts in the "expired" state. The countdown time T is transferred to a countdown register t when the timer is set with the "start" event. In the "countdown" state, the countdown register gradually decreases by δT at each "tick". Most state machines are synchronised to an external clock and execute once every "tick" of the clock. (Transient events last just one tick) The timer passes to the "ringing" state when the countdown timer reaches zero. It will remain in this state until the "acknowledge" signal is received.

Notice how actions that are performed along state transition arcs are defined following the '/' after the event that triggers the arc.

To our knowledge there is no standard way of representing multiple finite state machines that operate simultaneously other than by defining a full state transition table for the entire group of machines. This can generate enormous complexity. A state transition diagram summarises a state transition table in graphical form. UML (universal modelling language) is widely used by software engineers yet it cannot easily accommodate multiple state machines. This is a practical engineering issue that needs to be addressed. In the meantime the best we can do is to draw state transition diagrams clearly with explanatory notes to show how different state machines interact.

Select

Another simple example that is worth exploring is the typical selection mechanism used in computer software. A single mouse click and selects any object that was previously selected and selects the object over which the mouse pointer has been placed. If another click occurs within a short time the system launches the application handling the object that has been selected.

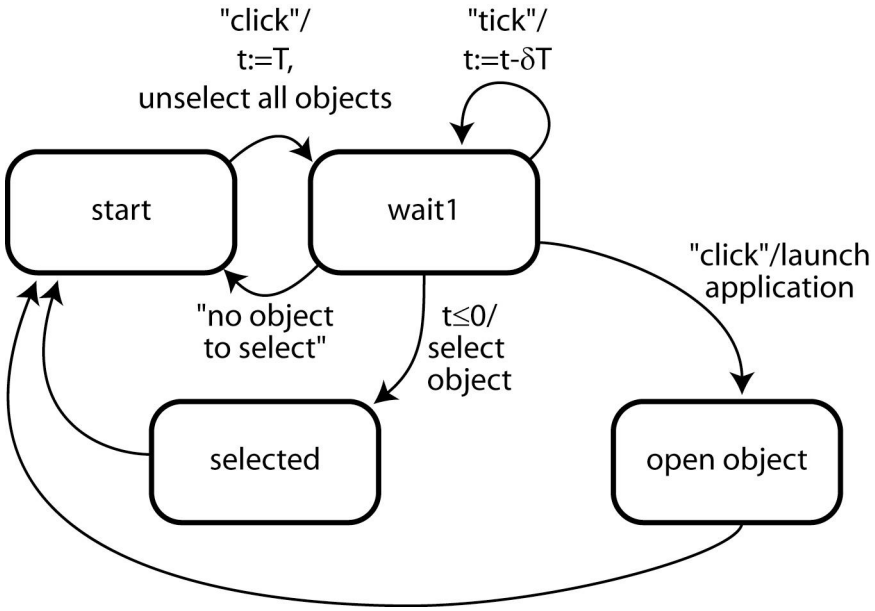

Figure 7: State transition diagram for single click or double-click selection. Are there redundant states?

Prelude and postlude actions

Another way to describe actions that occur along state transition arcs is to introduce the concept of "prelude" and "postlude" actions.

In the two previous examples we see that certain actions need to be performed when a given state is entered. As we enter the "wait1" state (above) we deselect all other objects and set the timer value T. These are the "prelude" actions for the "wait1" state. Launching the application that handles the object selected is then the "prelude" action of the "open object" state. Notice that we leave this state immediately returning to the "start" state. Postlude actions are performed as we leave a state. However, postlude actions tend to be used less than prelude actions because there are typically multiple pathways on leaving a given state, each with different postlude actions.

State transition diagrams have their limitations. The software package Visual Studio, for example, has no less than 150 menu events leading off from the startup state alone. This kind of state machine is far too complex to represent with conventional state transition diagrams. Decomposition into groups of states is essential.

Hardware Implementation Schemes

There are two slightly different classical hardware implementations of finite state machines. Modern software techniques enable us to implement highly complicated state machines using Field Programmable Gate Arrays (FPGAs) and Very Large Scale Integrated circuits (VLSI). Central processor circuits for computers are large finite state machines. However the same technologies allow us to design purpose-built

machines for specific applications. Application Specific Integrated Circuits (ASICs) are cheaper to mass-produce than FPGAs but the latter have much lower setup costs for small production runs.

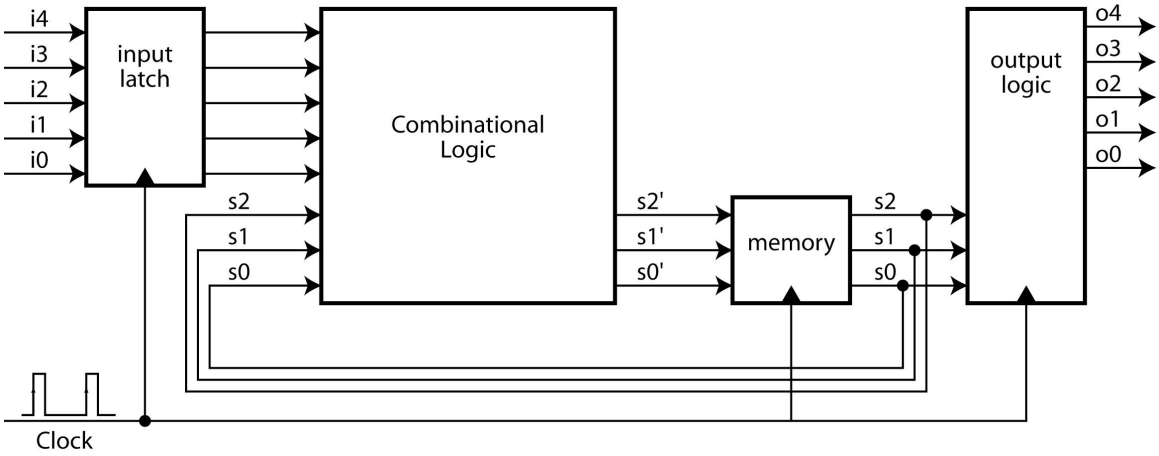

Figure 8: Moore machine architecture. Inputs i0, i1, i2… are captured at each clock pulse and, together with state values s0, s1, s2… form inputs to combinational logic that computes the new state values s0', s1', s2'… The new state values become the actual state when they are latched at the start of the next clock pulse. Output logic computes the output values o0, o1, o2… from the state values.

The Moore machine architecture is shown in figure 8. It consists of an input latch, combination logic, state memory and output logic. An external clock synchronizes the input latch, state memory and output logic: outputs appear immediately after each clock pulse and remain unchanged until the next clock pulse. Practical limitations on combination logic speed mean that a small but finite time elapses between presenting input signals immediately after the clock pulse and the arrival of the next state value at the inputs to the state memory. Output signals are computed from the state value.

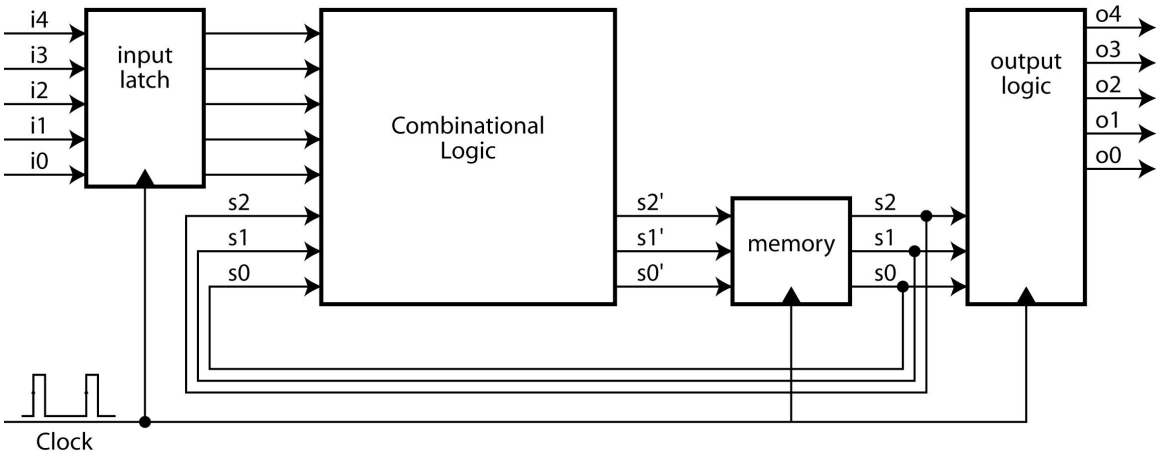

Figure 9: Mealy machine implementation. Inputs i0, i1, i2… are captured at each clock pulse and, together with state values s0, s1, s2… form inputs to combinational logic that computes the output values o0, o1, o2… and the new state values s0', s1', s2'… The new state values become the actual state when they are latched at the start of the next clock pulse.

The Mealy machine has an almost identical structure except that the outputs are computed from combination logic at the same time as the next state value. The Moore machine requires at least as many states as there are possible output combinations. However the Mealy machine requires fewer states.

It is also important to note that the timer state machine cannot easily be represented unless we consider that every possible timer value is an additional state in its own right. In practice, of course, these state machines are usually represented by software implemented in a microprocessor rather than by binary hardware devices.

State Transition Tables

As the state machine operates at clock ticks spaced at time δt numbered k=0, 1, 2, 3, … k, k+1, k+2, … we can represent the current state as S_k and the current input as I_k and the current output as O_k.

The current state is computed from the previous state and the current input values:

$S_k = \mathbf{f}_s(S_{k-1}, I_k)$ where $\mathbf{f}_s$ is known as the state transition function. This is also written as $\mathbf{f}_s$:SxI→S.

The output values are computed similarly:

$O_k = \mathbf{f}_o(S_{k-1}, I_k)$ where $\mathbf{f}_o$ is known as the output function.

This is also written as $\mathbf{f}_o$:S→O (in the case of the Moore machine) or $\mathbf{f}_o$:SxI→O (in the case of the Mealy machine).

We can define a state and input alphabet, typically a binary alphabet (0, 1) and then define $\mathbf{f}_s$ and $\mathbf{f}_o$ in the form of a look-up table, or state transition table as the following example shows.

Exercise 6

What does the following finite state machine represent? There is a single input value (0 or 1) and a single output value (0 or 1). On each state transition arc the input values and output values appear as input/output.

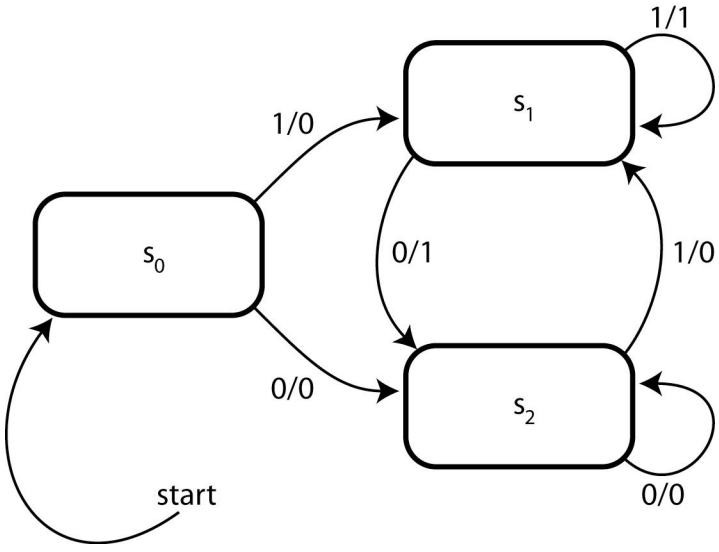

Figure 10: What is this finite state machine?

Represent states with two binary bits:

s_0 0 0

s_1 0 1

s_2 1 0

Then fill in the following state transition table. The first two rows have been filled in for you…..

Current State		Input	New State		Output
State bit 0	State bit 1	i_0	State bit 0	State bit 1	o_0
0	0	1	1	0	0
0	0	0	0	1	0
0	1	0			
0	1	1			
1	0	0			
1	0	1			

Hint: make an input sequence such as 00111001000011110000 and work out the state changes and the output values.

Creating and Designing Finite State Machines

Logic circuits, application-specific integrated circuits, PLCs, computer software, pneumatic hardware, fluidic devices and mechanisms all provide convenient ways to implement finite state machines. There are many design tools especially for the more complex state machines such as those found in large-scale integrated circuits and telephone exchanges.

Software tools such as MATLAB's Simulink and other discrete event simulation languages are very useful for modelling finite state machines. If you follow careful design procedures, simulation should not necessary except for the most complex designs.

In this course many projects will require you to implement finite state machines in software and the following notes provide some helpful guidance on how to do this reliably.

Implementing Finite State Machines in Software

Software finite state machines can provide a highly reliable code by following certain guidelines. The following flowcharts illustrate how the code should be structured. This structure can be accommodated in almost any programming language.

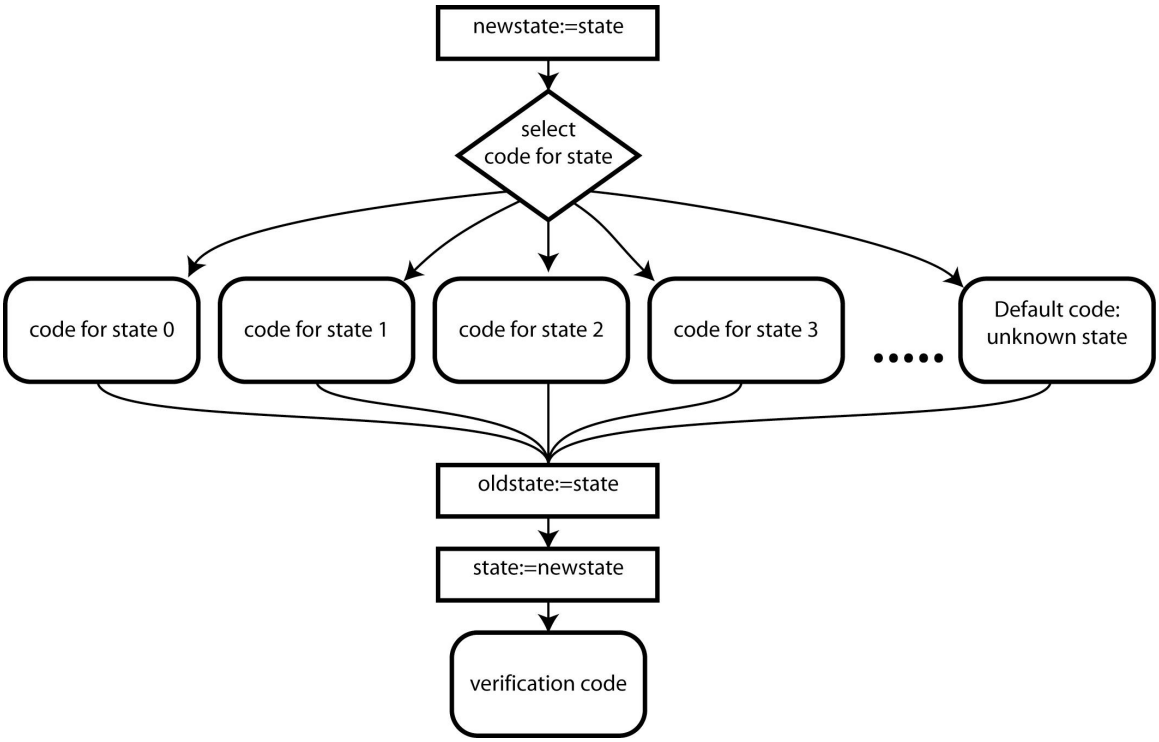

Figure 11: Finite State Machine implementation in software: overall flowchart.

In figure 11 we can see that the code is structured in the form of a "switch" statement (in C, C++ or Java) with a case for every state. A "case" structure should be used in LabVIEW. Note that there is also a default state to cover the case of an unknown value for the variable "state". This can happen if there is a mistake in the code so the default code is simply used to detect the fact that this kind of error has occurred. Never use a default case for legitimate state values. This is lazy coding and leads to disasters.

The local variables "oldstate" and "newstate" are used within the code for each state to detect conditions under which the prelude or post lewd actions should be executed. This is shown in the following flowchart. Note that the values of "state" and "oldstate" must be retained between successive executions of the code. They cannot be allocated to temporary local storage unless that storage is guaranteed to be preserved.

The last section of the code is verification code. This code is included to ensure that the module operates correctly. Finite state machine code is usually critical to all other aspects of the operating software so it is essential to confirm that operates correctly. The verification code, at the least, will record every state change to a log file or some other form of permanent storage. In the case of a high reliability application the verification code could incorporate a state transition table that defines all legitimate state transitions so that it can flag an error if an illegitimate state transition occurs. Note that recording data to a log file can be time-consuming, at least in some real-time applications, and separate notes provide some useful techniques for recording data in time- and space-efficient manner.

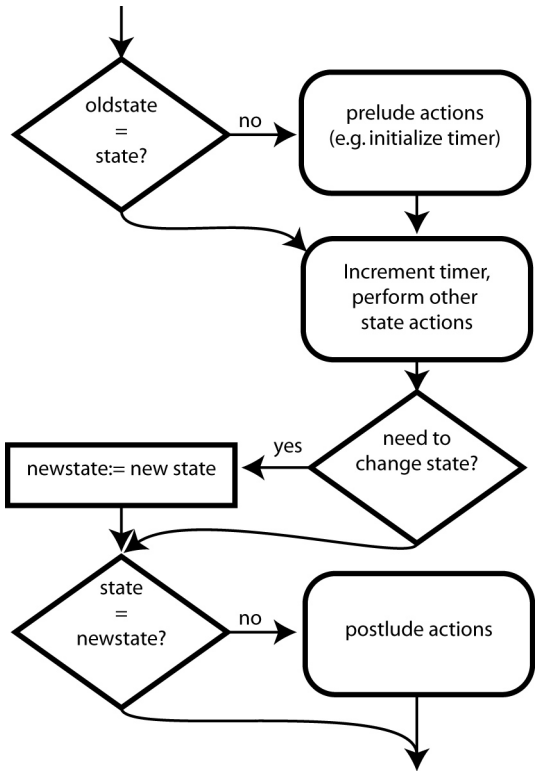

Figure 12: Flowchart template for each state

The code for each state follows a template and consists of four separate sections. At the beginning the code checks to see whether the state value has changed on the previous time the code was executed and, if so, executes prelude actions such as initialising timer values. Similarly, at the end, the code executes postlude actions if the state is about to be changed. In between the code performs actions specific to this particular state and no other state. This is the most important feature of the code template. By having separate code for every distinct state changes can be made to one state without having to be concerned that these changes may have side-effects in the other states. After this the code checks to see whether the state should be changed and if so sets the new state value into the variable "newstate".