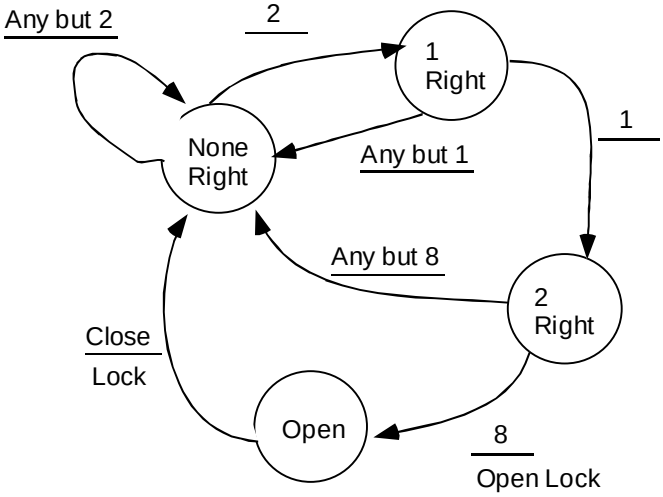
State Machines

While event driven programming by itself will allow you to successfully tackle a set of embedded software problems, it really comes into bloom when it is combined with a concept known as state machines. This combination can tackle just about any problem that you can throw at it while retaining the simplicity and design orientation of event driven programming.

State machines are formally referred to as Finite State Machines (FSM) or Finite State Automata (FSA). These 'machines' are constructs that are used to represent the behavior of a reactive system. Reactive systems, as opposed to transformational systems, are those systems whose inputs are not all ready at a single point in time. Hopefully, this sounds a lot like the places where event driven programming would be useful because the two concepts go hand in hand. Events are the driving forces in state machines.

At it simplest, a finite state machine consists of a set of states and the definition of a set of transitions among those states. At any point in time the state machine can be in only one of the possible states. It moves between the states, transitions, in response to events. It is probably easiest to understand state machines by examining the graphical depiction, known as a Finite State Diagram (FSD) or State Transition Diagram (STD), of a simple example.



The finite state diagram shown above describes the behavior of a combination lock whose combination is 2-1-8. The FSM can be in one of four possible states: NoneRight, OneRight, TwoRight, and Open. The arrows between the state bubbles represent the transitions, labeled with the event that triggers that transition. The bubbles represent the states. While in a state, the FSM is waiting for an event to cause it to transition.

The lock FSM transitions from NoneRight to OneRight when the digit '2' is entered. When in the OneRight state, entering a '1' causes a transition to the TwoRight state. Any other entry returns the FSM to NoneRight. This pattern continues until we are ready to examine the transition from TwoRight that is triggered by an entry of '8'. This transition is labeled in two parts. The upper part calls out the event that triggers the transition and the lower part describes an action associated with taking the transition. In this case, the action is to release the lock. The actions are one of the ways in which state machines actually *do* something.

When the state diagram appears to be complete, the next step is to test it. In this phase we imagine sequences of events and examine how the system described by the FSM would behave. It is pretty easy to convince yourself that for any sequence of three digits, the FSM will only unlock for 2-1-8. But what about four-digit sequences? Let's try 2-1-8-1. The FSM ends up in the NoneRight state with the lock locked, that's good. From this example sequence it is pretty easy to extrapolate that any four-digit sequence that begins with the correct combination will leave the lock locked. What about 1-2-1-8? That leaves the lock open, as will any other four-digit sequence that ends in 2-1-8!

We are now faced with a decision. We could accept this behavior, though it is probably not what we expected. Or we could modify the design in some way to eliminate the undesired response. For our

purposes here, the decision about how to treat the situation is of less importance than the fact that we found a possible error in our design. Notice, we just tested our design and found a potential flaw before writing the first line of code! This is one of the real strengths of a methodical use of state machines. The design can be tested easily and repeatedly before any code is written.

A State Machine in Software

While there are a number of ways that you can go about implementing a state machine in software, the most straightforward approach is probably a series of nested IF-THEN-ELSE statements. Using this approach, there is a series of IF clauses that test for the possible machine states. Within each of those state tests there is another series of nested IF clauses that handle the events that could occur while in that state. This combination of IF statements are wrapped inside a function that maintains a static local variable to track the current state of the machine. This state machine function takes at least one parameter that passes the most recent event to the state machine. For the lock example that we were just looking at, the state machine function, implemented in C, might look like:

```
void LockStateMachineIF( unsigned char NewEvent)
{
     static unsigned char CurrentState = NoneRight;
     unsigned char NextState;

     if( CurrentState == NoneRight)
     {
           if( NewEvent == KeyEqual2) /* Key == '2' ? */
                NextState = OneRight;
           /* no else clause needed, we are already in NoneRight */
     }else if( CurrentState == OneRight)
     {
           if( NewEvent == KeyEq1)    /* Key == '1' ? */
                NextState = TwoRight;
           else
                NextState = NoneRight; /* Bad Key go back to none */
     }else if( CurrentState == TwoRight)
     {
           if( NewEvent == KeyEq8) /* Key == '8' ? */
           {
                NextState = Open;
                OpenLock();
           }
           else
                NextState = NoneRight; /* Bad Key go back to none */
     }else if( CurrentState == Open)
     {
           NextState = NoneRight;
           LatchLock();
     }
     CurrentState = NextState; /* update the current state variable */
     return;
}
```

The main function to run this state machine would look something like:

```
void main(void)
{
     unsigned int KeyReturn;
     while(1)
     {
           KeyReturn = CheckKeys(); /* check for events */
           if( KeyReturn != NO_KEYS)
                LockStateMachineIF(KeyReturn); /* run state machine */
     }
}
```

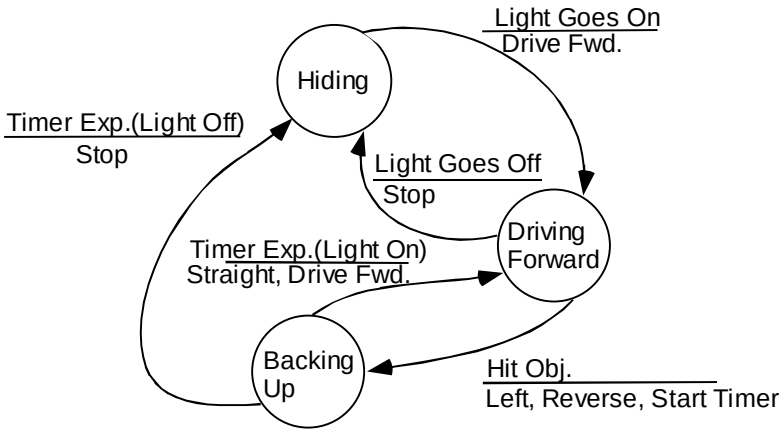
This program would run forever ('while (1)'), checking for key input. Every time that the event checker (CheckKeys()) found that there had been a new key pressed, the LockStateMachine() function would be called with a parameter that indicated *which* key had been pressed. In this case, the event:service pair would be CheckKeys:LockStateMachineIF.

The nested IF clause approach works well for problems like this where there are only a few states and only one or two paths out of each state. As the state machine becomes more complex it will be easier to read and often generate more efficient code if the problem is expressed using a nested CASE structure, rather than the nested IF-THEN-ELSE clauses. As an example of this structure, the following function duplicates the lock state machine function using a nested switch():case structure in C.

```
void LockStateMachineCASE( unsigned char NewEvent)
{
     static unsigned char CurrentState;
     unsigned char NextState;
     switch(CurrentState)
     {
           case NoneRight :
                switch(NewEvent)
                {
                      case '2':
                            NextState = OneRight;
                            break;
                      default:       /* we are already in NoneRight */
                            break;
                }
                break;
           case OneRight :
                switch(NewEvent)
                {
                      case '1':
                            NextState = TwoRight;
                            break;
                      default:       /* anything else sends us back */
                            NextState = NoneRight;
                            break;
                }
                break;
           case TwoRight :
                switch(NewEvent)
                {
                      case '8':
                            NextState = Open;
                            OpenLock();
                            break;
                      default:
                            NextState = NoneRight;
                            break;
                }
                break;
           case Open :
                NextState = NoneRight;
                LatchLock();
                break;
     }
     CurrentState = NextState;
     return;
}
```

While the C code for this simple function is noticeably longer than that for the nested IF clause version, the actual code generated by the compiler is comparable (within about 10%). The clarity of the code is improved by the labeling of the event cases as well as the explicit default case. As the complexity of the state machine grows, both in terms of the number of states and number of active events, the clarity and efficiency of the CASE structure becomes stronger. For anything more complex than the simplest state machines, the CASE structure will be preferable.

As another example, let's cast the cockroach behavior from the Event Driven Programming section as a state machine. The events and actions remain the same, this is simply another way of representing the problem. In this case you should come up with a state diagram that looks something like:



Here, the behavior of the 'roach' is described as one of three states (Hiding, Driving Forward and Backing Up) and the same set of events are used to trigger transitions between these states.

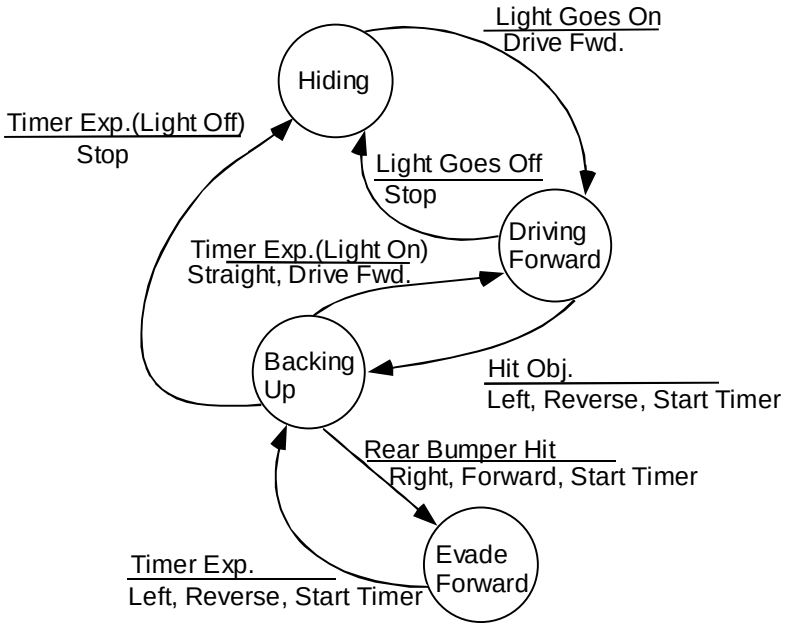
The diagram also introduces another feature of state machines: Guard Conditions. Notice that, while backing up, the Timer Expires event triggers one of two possible transitions, depending on the current state of the light. The current state of the light is the Guard Condition on those two transitions. In order to take either of the transitions the event must occur *and* the guard condition must be true.

It is important to be very careful whenever a single event will take one of two transitions based on guard conditions. The thing to be careful of is that the two guard conditions on the transitions are complements of one another. If this were not the case, it would be possible to have the event occur and not respond to it because neither guard condition was met.

As drawn, this state machine captures the specifications without the ambiguity that we discovered in the Event Driven Programming section. Notice that the light Goes Off event is only responded to if the state machine is in the Driving Forward state. In this way, if the light goes out while backing up, it will continue to back up for the three seconds. Only when the timer expires will it either resume going forward or stop, based on the current state of the light.

The state machine representation has another benefit over the purely events and services approach. In the events and services solution, we needed to simulate a Light Off event in order to get the desired behavior. This required that the scope of the variable holding the last state of the light be expanded beyond the LightGoesOn event checker. That is not necessary in the state machine solution.

If we make the problem just a little more complex, we can really see how the state machine representation becomes useful. Let's add a response to the back bumper that will only be active while we are backing up. Much like the response to the front bumper, it should change direction (go forward) for a period of time and turn (right this time). When this maneuver is complete, it should return to the BackingUp State. Adding this to the state machine requires adding one new state (EvadingFwd), one new event (rear bumper hit), one new action (wheels right; motor forward) and two new state transitions.



In this revised state machine, there are two distinct responses to the Timer Expires event. If we are backing up, we should straighten the wheels & go forward or stop, depending on the guard condition. If we are evading forward, we should always set the wheels left and go into reverse. The response to the event depends on what we are currently doing. This is a situation that would be messy to handle using only events and services. A state machine is an excellent way to capture that type of behavior.

While this state machine might not be eventual final design, notice that it won't stop while evading forward, it does provide an easily understood representation of the behavior. That is one of the key strengths of using state diagrams. They allow you to capture the behavior of the design very early in the process, easily present it to others and immediately begin testing it.

The thing to be emphasized at this point is that we have described the software to control a machine with an interesting behavior in very short order. As importantly, the actual code that would need to be written to implement the individual functions described is relatively simple. The combination of focusing on events and a state machine representation allows us to easily decompose the problem into relatively simple sub-problems (test bumper, test for light goes on, drive forward…) that are then easier to code without errors. The state machine captures the complexity of the desired design behavior in an easily understood framework. Filling in the framework with the code from the simple sub-problems will give us a working program in a much shorter time than would be possible without the event-driven/state machine paradigm.