# Finite State Machines in Embedded Applications

J. L. Garbini

A program that sequences a series of actions, or handles inputs differently depending on what mode its in, is often implemented as a finite state machine. A *state* is a condition that defines a prescribed relationship between inputs and outputs, and between inputs and subsequent states. A *finite state machine* is an algorithm that can be in a finite number of different states.

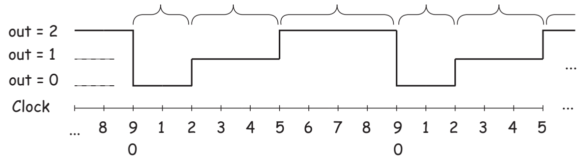
For example, consider the control algorithm for an elevator operating between two floors. The elevator has four possible states: Stopped on floor-1, Stopped on floor-2, Moving up, and Moving down. Inputs include: 1) the buttons that are pushed in the elevator car and on each floor and 2) limit switches indicating that the car has reached either floor. The outputs are the commands to the lift motor, to the elevator doors, and to the indicator displays in the car and on the floors. The outputs and the transition from one state to another depend on the current state and inputs.

Technically, a state machine for which the outputs are functions of both the current state and the inputs is called a Mealy machine. A state machine for which the outputs are functions of only the current state is called a Moore machine.

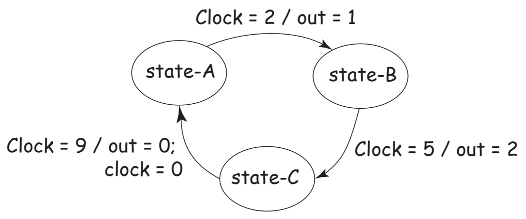
An advantage of using state machines is that the necessary logic can be represented graphically in a state transition diagram. A state transition diagram shows the input/output relationships and the conditions for transitions between states. A skeleton of code that implements any state transition diagram can be standardized.

Let's examine the state transition diagram for a simple example, and see how it might be coded. This system contains three **states** (A, B, and C). Its only input is the sequential count of a **clock** (0, 1, 2, . . . ). And, its outputs are a variable called **out**, and the **clock** (which the algorithm may reset to 0). The clock increments at a fixed rate. Potential state transitions are evaluated at each clock count.

The state machine operates as follows: The system stays in **state-A** until **clock=2**, then it makes **out = 1**, and changes to **state-B**. It stays in **state-B** until **clock=5**, then makes **out = 2**, and changes to **state-C**. Finally, it stays in **state-C** until **clock=9**, then makes **out = 0**, resets the clock (**clock=0**), and changes back to **state-A**. The process repeats indefinitely, producing a periodic output of 9 clock counts. A plot of the output would look like this:



This complicated natural language specification of the system operation can be represented very simply in the following state transition diagram.



The arrows between states are commonly labeled as:

$$\left\langle \begin{matrix}\text{Event that caused}\\ \text{the transition}\end{matrix} \right\rangle \Big/ \left\langle \begin{matrix}\text{Output(s) as a result}\\ \text{of the transition}\end{matrix} \right\rangle$$

Often the information in the state transition diagram is described the form of a *state transition table*:

| Current State | Inputs | Outputs | | Next State |
|---|---|---|---|---|
| | Clock | Out | Clock | |
| state-A | 2 | 1 | nc | state-B |
| state-B | 5 | 2 | nc | state-C |
| state-C | 9 | 0 | 0 | state-A |

nc = no change

As shown, the table lists all possible transitions between states, the conditions that cause the state transitions, and the corresponding outputs.

Now, how can this be efficiently coded? The listing on the following page illustrates one possibility.[1] You will need to study this code carefully. Be sure that you understand all of the C constructs. Some of them are tricky!

Each state is implemented as a separate C function. The heart of the program is the "Main State Transition Loop" (Note: just three lines of code!) This infinite loop calls the function corresponding to the current state. The variable **curr_state** keeps track of which state is current. The loop also causes a wait for one clock period, increments **Clock**, and then repeats.

The primary task of each state function is to determine if the current state should be changed. If no change is needed, the function does nothing. If the state is to be changed, the function sets **curr_state** to the new state, and alters the outputs appropriately.

A function, **initializeSM()**, is included to initialize the state machine.

At first, this may appear to be unnecessarily complicated for this simple example. However, the same code can be expanded easily (by adding more state functions) to implement a state machine of any complexity, with an unlimited number of states, inputs, and outputs.

[1] See also: Gomez, M., "Embedded State Machine Implementation", *Embedded Systems Programming* December 2000, p. 40-50.

```
/* State Machine Example */

#include <stdio.h>

/* Prototypes */
void stateA(void);
void stateB(void);
void stateC(void);
void initializeSM(void);
void wait(void);

/* Define an enumerated type for states */
typedef enum {STATE_A=0, STATE_B, STATE_C} State_Type;        Define a new data type

/* Define a table of pointers to the functions for each state */
static void (*state_table[])(void)={stateA, stateB, stateC};  An array of pointers to named functions!

/* Global Declaration */
static State_Type curr_state;        /* The "current state" */    Global variables, including the
static int   Clock;                                                current state, clock, and output
static int   out;

void main(void)
{
/* Initialize the State Machine*/
 initializeSM();                             Initialize State Machine (once)

/* The is the main state transition loop */
 while (1) {
  state_table[curr_state]();                 Call the current state
  wait();                                    Wait a fixed time interval     Main State Transition
  Clock++;                                   Keep track of clock count      Loop  (loop forever)
 }
}

void stateA(void)
{
 if( Clock == 2 ) {       /* Change State? */
  curr_state = STATE_B;   /* Next State */         State- A Function
  out = 1;                /* New output */
 }
}

void stateB(void)
{
 if( Clock == 5 ) {       /* Change State? */
  curr_state = STATE_C;   /* Next State */         State- B Function
  out = 2;                /* New output */
 }
}

void stateC(void)
{
 if( Clock == 9 ) {       /* Change State? */
  Clock = 0;                                       State- C Function
  curr_state = STATE_A;   /* Next State */
  out = 0;                /* New output */
 }
}

void initializeSM(void)
{                                                  State Machine
 curr_state = STATE_A;                             Initialization Function
 out = 0;
 Clock = 1;
}
```