04.4 Finite state machines

A program that sequences a series of actions, or handles inputs differently depending on what mode it's 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:

1. stopped on floor-1,
2. stopped on floor-2,
3. moving up, and
4. 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 each floor.

The outputs are the commands

1. to the lift motor,
2. to the elevator doors, and
3. 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. 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

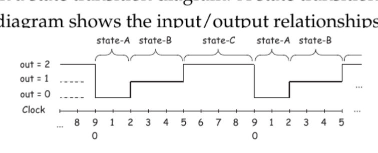

Figure 04.1:

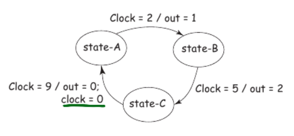

Figure 04.2:

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 variable `Clock` (0, 1, 2, …). Its outputs are a variable `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 A until `Clock == 2`, then it sets `out = 1`, and changes to B. It stays in B until `Clock == 5`, then sets `out = 2`, and changes to C. Finally, it stays in C until `Clock == 9`, then sets `out = 0`, resets the clock (`Clock = 0`), and changes back to A. The process repeats indefinitely, producing a periodic output of 9 clock counts. A plot of the output would look like that of Fig. 04.1.

This complicated natural language specification of the system operation can be represented very simply in a state transition diagram, such as that of Fig. 04.2.

The arrows between states are commonly

Table 04.1: state transition table with ○: no change.

when state is	and input is	then output		and make state
	Clock	out	Clock	
A	2	1	○	B
B	5	2	○	C
C	9	0	0	A

labeled as:

⟨ event that caused the transition ⟩ / ⟨ output(s) as a result of the transition ⟩

Often the information in the state transition diagram is described in the form of a state transition table, such as that of Table 04.1.

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.[2] You will need to study this code carefully. Be sure that you understand all the C constructs. Some of them are tricky!

2. See also Gomez (2000).

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 in the following to initialize the state machine.

```
/* 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 an array of pointers to each state function */
static void (*state_table[])(void) = {
  stateA, stateB, stateC
};

/* Global variable declaration */
static State_Type curr_state; // The "current state"
static int Clock;
static int out;

void main(void) {
  /* Initialize the state machine */
  initializeSM();

  /* Main state transition loop */
  while (1) {
    state_table[curr_state](); // call cur. state fnct.
    wait(); // wait fixed time interval
    Clock++;
  }
}

/* SM initialization function */
void initializeSM(void) {
  curr_state = STATE_A;
  out = 0;
  Clock = 1;
}

/* State functions */
void stateA(void) {
  if( Clock == 2 ) {       // change state?
    curr_state = STATE_B; // next state
    out = 1;              // new output
  }
}

void stateB(void) {
  if( Clock == 5 ) {       // change state?
    curr_state = STATE_C; // next state
    out = 2;              // new output
  }
}

void stateC(void) {
  if( Clock == 9 ) {       // change state?
    Clock = 0;            // reset clock
    curr_state = STATE_A; // next state
    out = 0;              // new output
  }
}
```

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.