ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/273636602

How to code finite state machines (FSMs) in C. A systematic approach

Technical Report · November 2014

DOI: 10.13140/2.1.4147.9200

CITATION
1

READS
14,645

1 author:

Lluís Ribas-Xirgo
Autonomous University of Barcelona

75 PUBLICATIONS **142** CITATIONS

SEE PROFILE

All content following this page was uploaded by Lluís Ribas-Xirgo on 17 March 2015.

The user has requested enhancement of the downloaded file.

How to code finite state machines (FSMs) in C. A systematic approach

Lluís Ribas-Xirgo,
Universitat Autònoma de Barcelona (UAB),
Bellaterra, Barcelona.
October-November 2014

State machines are used in a number of applications, from games to embedded systems. They are used to control elements like characters in games or devices with embedded systems.

A state machine is a model of a machine that reacts to any change at inputs from outside and generates outputs accordingly. The way it reacts to input changes depends on its state.

Take, for instance, a "useless machine" (you bet they exist!) that switches itself off every time someone switches it on. The control of such a machine can be modeled (you guessed it!) by a state machine. If the switch is off, it stays idle. If the switch is on, it moves a finger to push it back to the off position. Then, it returns the finger to the original position and sets itself idle again. So it has two main states: being idle and switching itself off, and one input: switch position, and one output to make the finger move.

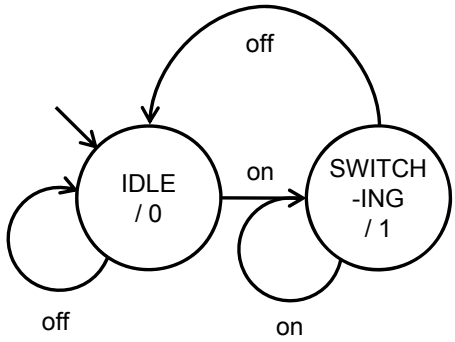


Figure 1. Useless machine controller FSM

There are many ways to code the control of this machine. Typically, you may translate the behavior into a control flow like this:

```
int main() {
  int get_switch_position(); /* Input function defined elsewhere     */
  void move_finger();        /* Output function defined elsewhere    */

  while( 1 ) { /* Infinite loop, as the controller is permanently on */
    while( get_switch_position() == 0 ) {
      move_finger( 0 );
    } /* while */
    while( get_switch_position() == 1 ) {
      move_finger( 1 );
    } /* while */
  } /* while */
} /* main */
```

However, it's much better you code into a state-based control-flow, with states of the machine explicitly stored in a variable. This way, you can easily link the state machine with the program code.

```
1: int main() {
2:   int get_switch_position(); /* Input function defined elsewhere     */
3:   void move_finger();        /* Output function defined elsewhere    */
4:   enum { IDLE, SWITCHING } state;
5:
6:   state = IDLE; /* Initially, the device's switch is off             */
7:   while( 1 ) { /* Infinite loop, as the controller is permanently on */
8:     switch( state ) {
9:       case IDLE:
10:        move_finger( 0 );
11:        if( get_switch_position() == 1 ) { state = SWITCHING; }
12:        break;
13:      case SWITCHING:
14:        move_finger( 1 );
15:        if( get_switch_position() == 0 ) { state = IDLE; }
16:        break;
17:    } /* switch */
18:  } /* while */
19:} /* main */
```

What's more, by using this coding pattern, it's possible to change from one state to another upon any condition, and, last but not least, in embedded systems, it allows verification methods other than simulation and makes it easy to compute execution costs.

Take, for instance, that you can compute the worst-case execution time (WCET) of a state-machine transition by carefully profiling each possible case. You can, then, use this information to determine the WCET of the controller, thus the longest reaction time to any input change or event. That's critical in real-time systems.

In the programs above, I/O functions `get_switch_position()` and `move_finger()` have not been defined, as they depend on the processor this program is going to be run, and on the device sensors and actuators.

There are many choices you can take about the processor: Arduino, Raspberry Pi, BeagleBone, and so on. Indeed, you can use any computer (including portables, tablets, smartphones, …), with I/O functions interacting with users through some graphical user interface. For the first kind of processors you will need to build the actual device (a prototype), and the latter can be used to interface with it or with a device simulator, i.e. with a program that simulates the physical elements of the useless machine, either autonomously or by interacting with some user. Picture, for example, a user clicking on an image of a switch on the screen to turn it on and a program that automatically turns it back to the off position.

In the following pages, you will see how more complex state machines can be systematically coded into any imperative language, such as C. And C is one of the most used for embedded systems.

1. Introduction

The following state machine corresponds to a four-state machine (S_0, S_1 and S_2) which changes states upon function *C*() returning true. The initial state is S_0 and there is an outgoing arc from S_1 that stops the machine when *C*(1, END). Each state is linked to some output update, which is represented by a call to a function *Out*() that takes the state index as argument.

It is assumed that the machine doesn't change state if no conditions *C*() are satisfied, i.e. all states have a loop over themselves.

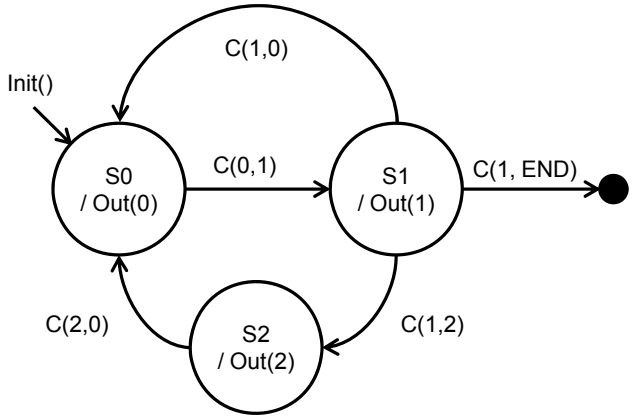


Figure 2. Sample FSM with 3 states and a sink one, representing a final stop (big black dot)

To translate a simple state machine into the corresponding C program you can use the model given below.

```
int main() {
  enum { S0, S1, S2, Sstop } state;

  state = S0;
  while( state != Sstop ) {
    switch( state ) {
      case S0:
        Out( 0 );
        if( C(0, 1) )   { state = S1;    }
        break;
      case S1:
        Out( 1 );
        if( C(1, 0) )   { state = S0;    }
        if( C(1, 2) )   { state = S2;    }
        if( C(1, END) ) { state = Sstop; }
        break;
      case S2:
        Out( 2 );
        if( C(2, 0) )   { state = S0;    }
        break;
    } /* switch */
  } /* while */
  return 0;
} /* main */
```

Note that there is an initializing section where the state variable is set to the initial one and that the whole state machine code is run inside a loop.

In this case, the machine can reach a stop state. In many cases, though, machines run indefinitely, so the condition at the while instruction is always true, i.e. `while(1) { … }`.

1.1. Rules

To translate a FSM graph into a C program you have to:

(1) enumerate all states,
```
enum { S0, S1, S2, Sstop } state;
```
(2) set the state variable to the initial state,
```
state = S0;
```
(3) set the condition of the `while` loop properly, and
```
while( state != Sstop ) {
```
(4) include a `case` for every state symbol in the enumeration, but for the stop state.
```
case S0:
  break;
case S1:
  break;
case S2:
  break;
```

For every state case, you have to:

(1) perform all actions within the state, i.e. computing outputs, and
```
Out( 0 );
```
(2) include an `if` to test each outgoing arc condition.
```
if( C(1, 0) )   { state = S0;    }
if( C(1, END) ) { state = Sstop; }
if( C(1, 2) )   { state = S2;    }
```

As you see, it is quite a straightforward procedure.

It is important to recall that, in the model above, functions `C()` and `Out()` have to be replaced by corresponding condition evaluation and output updating procedures. Actually, they can be function calls, but to begin with, they will be expressions and instructions.

1.2. Example

One of the uses of state machines is that of recognizing sequences, for instance, to identify an incoming message and validate its data. Let's do a machine that says "HI" every time it gets a "HI" messages. This machine gets a stream of characters (one per cycle) and generates a stream of the same kind with dots or 'H's followed by 'I's. For the sake of simplicity, we assume that the input stream does never contain concatenated "HI"s.

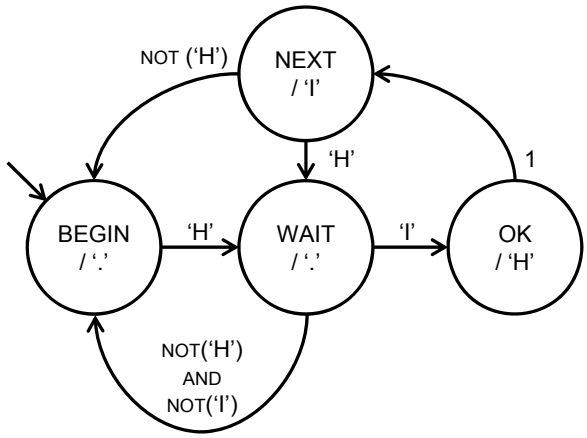


Figure 3. FSM to accept sequences of character string "HI" on input

The machine responds with an 'H' followed by an 'I' each time it receives "HI" from the input, thus it takes two states (OK and NEXT) to generate the output sequence in response to a "HI". Note that the arc from OK to NEXT is labeled with 1 (always true) because OK is used to output 'H' and nothing else. As input string is said not to contain two consecutives "HI", losing the input character at OK state doesn't affect the machine's function.

The corresponding program is listed here:

```
int main() {
  char read_input(); /* character input function defined elsewhere */
  void write_output( char c ); /* character output function */
  enum { BEGIN, WAIT, OK, NEXT } state;
  char input, output;

  state = BEGIN;
  while( 1 ) { /* It is running continuously */
    input = read_input();
    switch( state ) {
      case BEGIN:
        output = '.';
        if( input == 'H' ) { state = WAIT; }
        break;
      case WAIT:
        output = '.';
        if( input == 'I' ) { state = OK; }
        if( input != 'H' && input != 'I' ) { state = BEGIN; }
        break;
      case OK:
        output = 'H';
        state = NEXT;
        break;
      case NEXT:
        output = 'I';
        if( input == 'H' ) { state = WAIT; }
        if( input != 'H' ) { state = BEGIN; }
        break;
    } /* switch */
    write_output( output );
  } /* while */
  return 0; /* Unreachable! */
} /* main */
```

The former code should be completed by defining the I/O functions and will run until execution is interrupted (e.g. by typing [CTRL]+[C] on the console keyboard).

In this case, two variables are added to hold the value of the current input character and for the output character at the present state. By doing so, the previous model is extended by decomposing the FSM cycle into three phases:

(1) Read inputs;
```
input = read_input();
```
(2) Compute outputs at current state and next state, and
```
switch( state ) { ... }
```
(3) Write outputs.
```
write_output( output );
```

This organization prevents from multiple readings and writings in a single iteration. In this example, it prevents from taking two characters from input stream in cases with multiple accesses to the current input character (lines 17-18 and 26-27). Note that, this would happen if instead of taking the value from variable `input` a call to `read_input()` is done: two calls, two characters read in a single step.

Finally, note that the `return` from `main()` will never be executed because of the infinite loop. It is left in the code for consistency with the model given before. We will see that, in the end, this instruction can be reached when executing improved versions of FSM programs.

1.3. Tips

When programming it is important to **keep a naming convention** so the result source code is easy to understand and to manipulate. Therefore, we enumerate states with proper symbol names, e.g. by using state names beginning with "S_".

Include a default case to avoid unwanted state variable values let the machine stuck because of being at an undefined state. As this is an unrecoverable error, it's better to stop the execution after it. To do so, the main loop has to be changed as shown:

```
1:  int main (){
2:    /* … */
3:    int errcod; /* error code */
4:    /* … */
5:    errcod = 0;
6:    while( errcod == 0 ) {
7:      /* … */
8:      switch( state ) {
9:        /* … */
10:       default:
11:         errcod = -1;
12:     } /* switch */
13:     /* … */
14:   } /* while */
15:   return errcod; /* Now, it is eventually reachable! */
16: } /* main */
```

Check that no two different arc outgoing conditions are true at the same time, i.e. **insert a verification of determinism**.

Indeterminism happens when there is more than one possible next state, i.e. it cannot be determined which transition should be fired. Of course, with the given programming model, the state machine will set the next state to be the one included in the last listed `if` instruction, but that's not what it should be. It's much better to set a specific error code and stop the state machine.

Checking for indeterminism can be done by using a variable, e.g. `firings`) that is set to zero at the beginning of the transition (i.e. of the iteration) and increased by 1 every time an `if` condition is satisfied. At the end, if `firings` is zero (stay at the same state) or one, it's OK. But if it's more than one, there's an indeterminism.

The corresponding, reshaped, program is listed here:

```
int main() {
  /* … */
  enum { S_BEGIN, S_WAIT, S_OK, S_NEXT } state;
  char input, output;
  int errcod, firings;

  state = S_BEGIN;
  errcod = 0;
  while( errcod == 0 ) {
    input = read_input();
    firings = 0;
    switch( state ) {
      case S_BEGIN:
        output = '.';
        if( input == 'H' ) { state = S_WAIT; firings = firings +1; }
        break;
      case S_WAIT:
        output = '.';
        if( input == 'I' ) { state = S_OK; firings = firings +1; }
        if( input!='H'&&input!='I' ) { state = S_BEGIN; firings = firings +1; }
        break;
      case S_OK:
        output = 'H';
        state = S_NEXT;
        break;
      case S_NEXT:
        output = 'I';
        if( input == 'H' ) { state = S_WAIT; firings = firings +1; }
        if( input != 'H' ) { state = S_BEGIN; firings = firings +1; }
        break;
      default:
        errcod = -1;
    } /* switch */
    if( firings > 1 ) { errcod = firings; }
    write_output( output );
  } /* while */
  return errcod;
} /* main */
```

All these tips do not affect the normal behavior of the program but make it robust.

During development they help to detect programming logic errors and/or specification errors. For instance, forgetting a case in the state switch instruction or wrongly writing two overlapping case conditions on different transitions from the same state.

During execution, a program that includes all those former elements would rarely fail due to a flaw in the specifications or because of memory bit upsets (imagine the code is embedded on hardware that is exposed to radiation that may spuriously change a bit in the variables). Instead, it will detect (most of) errors and, eventually, perform appropriate error-recovering procedures.

1.4. Summary

The template for programming a FSM in accordance with what's been said is given below.

```
1:  int main() {
2:    /* … */
3:    enum { S_0 = 0, S_1 = 1, …, S_stop = N } state;
4:    /* … */
5:    int errcod, firings;
6:
7:    state = S_0;
8:    errcod = 0;
9:    while( errcod == 0 && state != S_stop ) {
10:     /* READ INPUTS */
11:     firings = 0;
12:     switch( state ) {
13:       case S_0:
14:         /* … */
15:         if( C(0,0) ) { state = S_0; firings = firings +1; }
16:         if( C(0,1) ) { state = S_1; firings = firings +1; }
17:         if( C(0,2) ) { state = S_2; firings = firings +1; }
18:         /* … */
19:         if( C(0,N) ) { state = S_stop; firings = firings +1; }
20:         break;
21:       case S_1:
22:         /* … */
23:         if( C(1,N) ) { state = S_stop; firings = firings +1; }
24:         break;
25:       /* … */
26:       default:
27:         error_code = -1;
28:     } /* switch */
29:     if( firings > 1 ) { errcod = firings; }
30:     /* UPDATE OUTPUTS */
31:   } /* while */
32:   return errcod;
33: } /* main */
```

It may look like a long code that can be minimized, and it's true! However, we should not mix the mechanical translation from transitions graphs (stage 1 of the FSM programming process) with possible further code optimizations (stage 2).

It is also recommendable to include self-loop conditions (line 15) so to verify that the machine stays at the same state only under the expected conditions and to make the determinism test complete. Note that you can also detect unexpected self-loops when `firings==0`.

2. Resettable FSMs

A reset button is something we expect from any electronic product to bring it into a known state. Usually from one state in which the product is useless to one where users gain control over it.

For plugged devices, we can force a reset by unplugging them. For battery-operated ones, things are more difficult, though. Anyhow, the ideal case is to have a neat reset mechanism.

A possible form of that is including a specific input reset signal that cause state machines return to the initial state. The model will look like the left transition graph below.

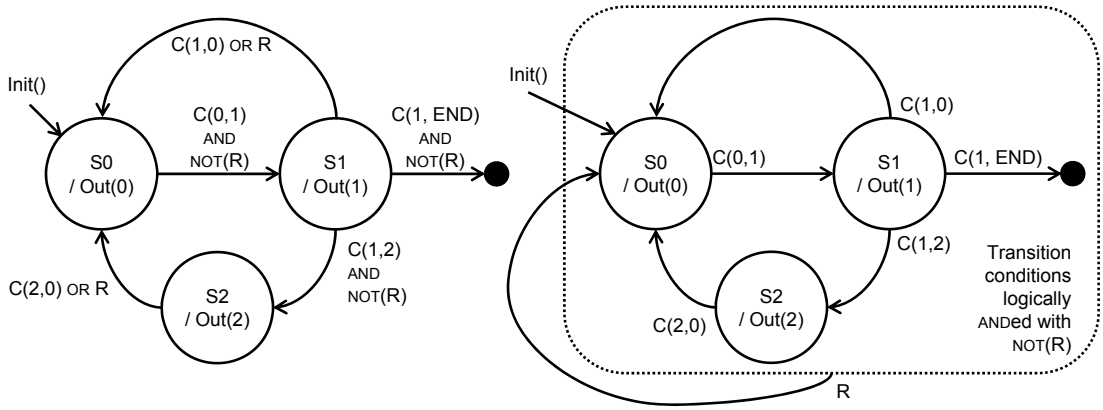


Figure 4. FSMs with reset input R, flat (left) and hierarchical (right)

On one hand, the reset signal has to be taken into account in all states, i.e. when *R* is present, next state has to be the initial one. On the other hand, the rest of the transitions can only be done if *R* is absent, i.e. the transition conditions are logically ANDed with NOT(*R*).

Given the highest priority of *R* and that it affects all states and transitions, this signal has to be processed before the output and next state computation stage and, in case there's reset, prevent it from executing.

Following the example of the detection of "HI" sequences in the input stream, it is possible to use character 'R' as a resetting signal. Therefore, instead of adding a new input signal, the machine uses the character stream input to get the reset indicator.

The corresponding program for the resettable FSM of the example is listed here:

```
1:  int main() {
2:    /* … */
3:    enum { S_BEGIN, S_WAIT, S_OK, S_NEXT } state;
4:    char input, output;
5:    int errcod, firings, reset;
6:
7:    state = S_BEGIN;
8:    errcod = 0;
9:    while( errcod == 0 ) {
10:     input = read_input();
11:     if( input == 'R' ) { reset = 1; } else { reset = 0; }
12:  /* if( reset ==  1  ) { state = S_BEGIN; } /* IMMEDIATE RESET */ */
13:     firings = 0;
14:     switch( state ) {
15:       case S_BEGIN:
16:         output = '.';
17:         if( input == 'H' ) { state = S_WAIT; firings = firings +1; }
18:         break;
19:       case S_WAIT:
20:         output = '.';
21:         if( input == 'I' ) { state = S_OK; firings = firings +1; }
22:         if( input!='H'&&input!='I' ) { state = S_BEGIN; firings = firings +1; }
23:         break;
24:       case S_OK:
25:         output = 'H';
26:         state = S_NEXT;
27:         break;
28:       case S_NEXT:
29:         output = 'I';
30:         if( input == 'H' ) { state = S_WAIT; firings = firings +1; }
31:         if( input != 'H' ) { state = S_BEGIN; firings = firings +1; }
32:         break;
33:       default:
34:         errcod = -1;
35:     } /* switch */
36:     if( firings > 1 ) { errcod = firings; }
37:     if( reset  == 1 ) { state  = S_BEGIN; } /* DELAYED RESET */
38:     write_output( output );
39:   } /* while */
40:   return errcod;
41: } /* main */
```

Note that, in this case, upon reset, the FSM returns to S_BEGIN in the next iteration. This is the behavior corresponding to the FSM models shown in Fig. 4. However, sometimes is interesting to immediately return to the initial state, without execution of the code for the current state. For that, you have to comment out line 37 and remove comment marks from line 12 in the listing above.

To distinguish between the two options, we can call the first one “delayed reset”, and the second, “immediate reset”. The choice entirely depends on the machine’s desired behavior.

3. Simulation of FSMs

With reset, we can bring a FSM to the initial state, for simulation, we need to bring it to some stop state, to be able to control its execution fully, i.e. to start, reset and stop it.

As with reset insertion, adding a stop input can be done either by adding a new specific input signal or by checking for a stop indicator in an existent input. Following the example given in this text, we shall use character 'Q' (from "quit simulation") as a stop indicator in the input character stream.

The corresponding code is the following one.

```
1:  int main() {
2:    /* … */
3:    enum { S_BEGIN, S_WAIT, S_OK, S_NEXT } state;
4:    char input, output;
5:    int errcod, firings, reset, quit;
6:
7:    quit = 0;
8:    state = S_BEGIN;
9:    errcod = 0;
10:   while( errcod == 0 && quit == 0 ) {
11:     input = read_input();
12:     if( input == 'Q' ) { quit  = 1; }
13:     if( quit == 0 ) {
14:       if( input == 'R' ) { reset = 1; } else { reset = 0; }
15:       firings = 0;
16:       switch( state ) {
17:         case S_BEGIN:
18:           output = '.';
19:           if( input == 'H' ) { state = S_WAIT; firings = firings +1; }
20:           break;
21:         case S_WAIT:
22:           output = '.';
23:           if( input == 'I' ) { state = S_OK; firings = firings +1; }
24:           if( input!='H'&&input!='I' ) { state=S_BEGIN; firings=firings+1; }
25:           break;
26:         case S_OK:
27:           output = 'H';
28:           state = S_NEXT;
29:           break;
30:         case S_NEXT:
31:           output = 'I';
32:           if( input == 'H' ) { state = S_WAIT; firings = firings +1; }
33:           if( input != 'H' ) { state = S_BEGIN; firings = firings +1; }
34:           break;
35:         default:
36:           errcod = -1;
37:       } /* switch */
38:       if( firings > 1 ) { errcod = firings; }
39:       if( reset  == 1 ) { state  = S_BEGIN; }
40:       write_output( output );
41:     } /* if */
42:   } /* while */
43:   return errcod;
44: } /* main */
```

While the program above is functionally correct, it mixes simulation control (the stop indicator 'Q' would not exist in reality) with FSM behavior.

The behavior of a FSM must be encapsulated in a set of data structure and functions, and the main one in charge of the simulation. As in object-oriented programming, a FSM can be modeled by a set of data that includes state, inputs and outputs and a couple of related methods or functions, namely the one to initialize the data set and the one to perform a single step or state transition:

```
1: struct {
2:   enum { fsm_BEGIN, fsm_WAIT, fsm_OK, fsm_NEXT } state;
3:   char input, output;
4: } fsm_data;
5:
6: int fsm_init() {
7:   fsm_data.state = S_BEGIN;
8:   return 0;
9: } /* fsm_init */
10:
11:int fsm_step() {
12:  int e, f, r; /* e: errcod, f: firings, r: reset */
13:
14:  e = 0;
15:  f = 0;
16:  if( fsm_data.input == 'R' ) { r = 1; } else { r = 0; }
17:  switch( fsm_data.state ) {
18:    case fsm_BEGIN:
19:      fsm_data.output = '.';
20:      if( fsm_data.input == 'H' ) { fsm_data.state = fsm_WAIT; f = f +1;
   }
21:      break;
22:    case fsm_WAIT:
23:      fsm_data.output = '.';
24:      if( fsm_data.input == 'I' ) { fsm_data.state = fsm_OK; f = f +1; }
25:      if( fsm_data.input != 'H' && fsm_data.input != 'I' ) {
26:        fsm_data.state = fsm_BEGIN; f = f +1;
27:      } /* if */
28:      break;
29:    case fsm_OK:
30:      fsm_data.output = 'H';
31:      fsm_data.state = fsm_NEXT;
32:      break;
33:    case fsm_NEXT:
34:      fsm_data.output = 'I';
35:      if( fsm_data.input == 'H' ) { fsm_data.state = fsm_WAIT;  f = f +1; }
36:      if( fsm_data.input != 'H' ) { fsm_data.state = fsm_BEGIN; f = f +1; }
37:      break;
38:    default:
39:      e = -1;
40:  } /* switch */
41:  if( f  > 1 ) { e = f; }
42:  if( r == 1 ) { fsm_data.state = fsm_BEGIN; }
43:  return e;
44:} /* fsm_step */
45:
```

Take into account that access to FSM variables has changed, as now they are packed into a single, global variable `fsm_data`.

Note that, in the former code, all state names have been changed their prefixes from "S_" to "fsm_" so to be in accordance with the rest of the naming convention.

The main function is now only devoted to initialize the set of data and to repeat the `fsm_step()` execution until the machine reaches a stop state, runs into an error or simulation is stopped.

```
46: int main() {
47:   int errcod, quit;
48:
49:   fsm_init();
50:   quit = 0;
51:   errcod = 0;
52:   while( errcod == 0 && quit == 0 ) {
53:     fsm_data.input = read_input();
54:     if( fsm_data.input == 'Q' ) { quit  = 1; }
55:     if( quit == 0 ) {
56:       errcod = fsm_step();
57:       write_output( fsm_data.output );
58:     } /* if not quitting */
59:   } /* while */
60:   return errcod;
61: } /* main */
```

This code organization enables stop the simulation whenever needed. For instance, after some error has been detected or, if we are lucky, when all tests have been successfully passed.

For instance, an input sequence "mHImmmRHImmmRHImmRHImRHIRmmHIHmRHIHImmQ" shall test as many as 7 cases, from a generic case to the unexpected case of concatenated "HI"s (yes, you should test your FSM under all possible conditions!). The cases in the given input sequence are separated with control character 'R' to cause the state machine go to the S_BEGIN state. The last character is a 'Q' to stop the simulation of the FSM execution.

By the way, the expected output is "…HI….HI….HI…HI..HI..H….HI…HI…", so it can be compared to the actual output in an automatic fashion to easily detect the differences.

Note that it is very interesting to prepare such kind of input sequences and expected output pairs even before inception of the state machine. This is a sort of specification that helps in the FSM design and, for sure, can be used to verify it.

3.1. Rules

In order to have a structured simulator of a FSM you should follow the model presented here. Hence, to have a C program of a FSM simulation you have to:

(1) Use a specific prefix for all FSM related objects, such as “fsm_”
This is particularly useful when other data structures and functions are also used in the program.

(2) Create a globally accessible data structure for the FSM
The structure must contain, at least, the enumerated type for the state variable. Additionally, it has to include variables to hold the values at the inputs and at the outputs. Follow a naming convention like the one given in the example, which consists of appending a “_data” at the end of the machine’s name.

```
1:  struct {
2:    enum { fsm_BEGIN, fsm_WAIT, fsm_OK, fsm_NEXT } state;
3:    char input, output;
4:  } fsm_data;
5:
```

(3) Prepare a initializing function
The least thing it does is accessing the machine’s data structure to set the state variable to some start state (usually, there’s only one, like in the example given). It is highly recommended to set also the input and output variables to some default values. These values can be either the ones after a reset or, in case they are not known or unset, indicators of such a case, i.e. values different from the regular values of the variables and, particularly, from the ones expected or set in the initial state. As with the name of data structure, the initializing function must be named accordingly e.g. ended by “_init”.

```
6:  int fsm_init() {
7:    fsm_data.state  = fsm_BEGIN;
8:    fsm_data.input  = '\0'; /* Unknown (void) input: NULL character */
9:    fsm_data.output = '*';  /* Unset output: asterisk */
10:   return 0;
11: } /* fsm_init */
12:
```

Note that initialization of FSM variables in lines 8 and 9 were not included in the previous version of `fsm_init()`.

(4) Pack the model behavior into a function that implements a single step
Name this function accordingly e.g. with postfix "_step" on it. This function must return an integer indicator of error detection or 0, if everything has gone smoothly.

```
11:int fsm_step() {
12:  int e, f, r; /* e: errcod, f: firings, r: reset */
13:
14:  e = 0;
15:  f = 0;
16:  if( fsm_data.input == 'R' ) { r = 1; } else { r = 0; }
17:  switch( fsm_data.state ) {
18:    case fsm_BEGIN: /* … */ break;
22:    case fsm_WAIT:  /* … */ break;
29:    case fsm_OK:    /* … */ break;
33:    case fsm_NEXT:  /* … */ break;
38:    default:        e = -1;
40:  } /* switch */
41:  if( f  > 1 ) { e = f; }
42:  if( r == 1 ) { fsm_data.state = fsm_BEGIN; }
43:  return e;
44:} /* fsm_step */
45:
```

(5) Implement a "simulation motor"
The typical pattern of a FSM simulator follows the model given by the example of the previous `main()` function:

```
46:int main() {
47:  int errcod, quit;
48:
49:  fsm_init();
50:  quit = 0;
51:  errcod = 0;
52:  while( errcod == 0 && quit == 0 ) {
53:    fsm_data.input = read_input();
54:    if( fsm_data.input == 'Q' ) { quit  = 1; }
55:    if( quit == 0 ) {
56:      errcod = fsm_step();
57:      write_output( fsm_data.output );
58:    } /* if not quitting */
59:  } /* while */
60:  return errcod;
61:} /* main */
```

Note that the simulator motor can have an input on its own instead of sharing the input of the FSM, like in the example given.

In a nutshell, the FSM simulator is decomposed into a data structure, `fsm_data`, two functions, `fsm_init()` and `fsm_step()`, and a simulator motor that implements the `fsm_step()` calling loop.

Here, "fsm" is used for the FSM, but any other name could be used. In the example given, you could use a name like "sayHi" ;-)

3.2. Simulation monitoring

To monitor what's going on during simulation you need to see inputs and outputs, but also iteration count as well as the state and other internal data of the simulated FSM. Therefore, the simulation motor must include an iteration counter, `tick`, and a call to a function `fsm_monitor()`, that outputs the internal data:

```
46: int main() {
47:   int errcod, quit, tick;
48:
49:   tick = 0;
50:   fsm_init();
51:   quit = 0;
52:   errcod = 0;
53:   while( errcod == 0 && quit == 0 ) {
54:     printf( "sim: cycle = %07i\n", tick );
55:     fsm_monitor();
56:     fsm_data.input = read_input();
57:     if( fsm_data.input == 'Q' ) { quit  = 1; }
58:     if( quit == 0 ) {
59:       errcod = fsm_step();
60:       write_output( fsm_data.output );
61:       tick = tick +1;
62:     } /* if not quitting */
63:   } /* while */
64:   return errcod;
65: } /* main */
```

Typically, the least thing `fsm_monitor()` does is to print the state of the FSM. Here is an example for the study case:

```
1:  int fsm_monitor() {
2:    printf( "mon: { state:" );
3:    switch( fsm_data.state ) {
4:      case fsm_BEGIN: printf( "BEGIN" ); break;
5:      case fsm_WAIT:  printf( "WAIT"  ); break;
6:      case fsm_OK:    printf( "OK"    ); break;
7:      case fsm_NEXT:  printf( "NEXT"  ); break;
8:      default:        printf( "UNKNOWN!");
9:    } /* switch */
10:   printf( " }\n" );
11:   return 0;
12: } /* fsm_monitor */
```

Of course, you can include any piece of information you may need like, for instance, the rest of the fields of the `fsm_data` structure:

```
1:  int fsm_monitor() {
2:    printf( "mon: fsm_data = { state: " );
3:    switch( fsm_data.state ) {
4:      case fsm_BEGIN: printf( "BEGIN" ); break;
5:      case fsm_WAIT:  printf( "WAIT"  ); break;
6:      case fsm_OK:    printf( "OK"    ); break;
7:      case fsm_NEXT:  printf( "NEXT"  ); break;
8:      default:        printf( "UNKNOWN!");
9:    } /* switch */
10:   printf( ", input: %c", fsm_data.input );
11:   printf( ", output: %c", fsm_data.output );
12:   printf( " }\n" );
13:   return 0;
14: } /* fsm_monitor */
```

In the example, monitoring input and output is not really necessary, but the former `fsm_monitor()` function is given only for illustrative purposes.

Ideally, I/O of simulation monitoring and control should be independent from FSM I/O. For instance, they can be done through different windows on the screen. However, for the sake of simplicity, we shall use the standard console for both. This is why it is so important to make it clear to the user that input stream is shared and which shown messages correspond to simulation control and which others to simulated machine.

In order to do so, system input/output (I/O) operations must be differentiated from monitoring ones. In the example, `fsm_monitor()` functions prefix text lines with “mon:”. Also it is highly recommendable to modify `read_input()` and `write_output()`to print lines with a “fsm:” prefix.

For instance an input “HImmQ” would generate the following dialogue (bold typeface corresponds to user input):

```
sim: cycle = 0000000
mon: fsm_data = { state: BEGIN, input: , output: * }
fsm: input = ? H
fsm: output = .
sim: cycle = 0000001
mon: fsm_data = { state: WAIT, input: H, output: . }
fsm: input = ? I
fsm: output = .
sim: cycle = 0000002
mon: fsm_data = { state: OK, input: I, output: . }
fsm: input = ? m
fsm: output = H
sim: cycle = 0000003
mon: fsm_data = { state: NEXT, input: m, output: H }
fsm: input = ? m
fsm: output = I
sim: cycle = 0000004
mon: fsm_data = { state: BEGIN, input: m, output: I }
fsm: input = ? Q
```

Note that simulation with monitoring can easily generate lots of data and that, for more complex state machines, it is impossible to be as exhaustive as in the example given.

However, what if an erroneous behavior is found after a long sequence of inputs? Stop the simulation, make changes and repeat the procedure until the error discovery cycle or try and go directly to the state and inputs of the cycle previous to the one where the error was discovered? Next section will discuss about that.

3.3. Extra feature: Hold, change and resume

The previous code organization enables stopping the simulation whenever needed. However, sometimes it is interesting to hold the simulation and resume it after eventual changes on the state machine's data structure.

To include the extra feature, a new input must be defined. In our example, we will take profit of the same input character stream, as for 'Q'. In this case, we use 'D'. When 'D' is given, an event to enter a *debugger* is set.

In this context, a *debugger* is a function that allows modifying any variable of a given state machine and, particularly, that of its state. Following the example, also the input character can be updated:

```
int fsm_debug() {
  int state_no;
  printf( "dbg: fsm_data.state [0:BEGIN, 1:WAIT, 2:OK, 3:NEXT] = " );
  scanf( "%u", &state_no );
  switch( state_no ) {
    case 0:  fsm_data.state = fsm_BEGIN; break;
    case 1:  fsm_data.state = fsm_WAIT;  break;
    case 2:  fsm_data.state = fsm_OK;    break;
    case 3:  fsm_data.state = fsm_NEXT;  break;
    default: printf( "dbg: fsm_data.state not changed!\n");
  } /* switch */
  printf( "dbg: fsm_data.input = " );
  scanf( " %c\n", &fsm_data.input );
  fsm_data.input = toupper( fsm_data.input ); /* #include <ctype.h> */
  return 0;
} /* fsm_debug */
```

Including this debugging option requires a slight modification in the code for the simulation motor:

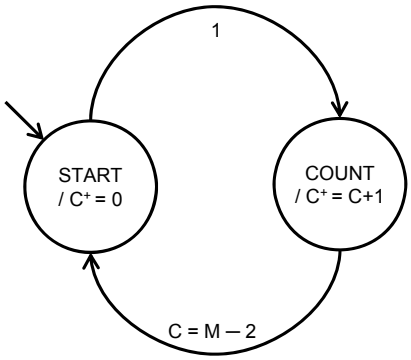
```
46: int main() {
47:   int errcod, quit, tick, debug;
48:
49:   tick = 0;
50:   fsm_init();
51:   quit = 0;
52:   debug = 0;
53:   errcod = 0;
54:   while( errcod == 0 && quit == 0 ) {
55:     printf( "sim: cycle %07i:\n", tick );
56:     fsm_monitor();
57:     fsm_data.input = read_input();
58:     if( fsm_data.input == 'D' ) { debug = 1; } else { debug = 0; }
59:     if( debug == 1 ) { quit = fsm_debug(); }
60:     if( fsm_data.input == 'Q' ) { quit  = 1; }
61:     if( quit == 0 ) {
62:       errcod = fsm_step();
63:       write_output( fsm_data.output );
64:       tick = tick +1;
65:     } /* if not quitting */
66:   } /* while */
67:   return errcod;
68: } /* main */
```

The simulation motor is thus complete, with capability to stop (quit), monitor and modify (debug) simulated FSM. For this, the FSM requires: a data structure to hold all of its internal variables, `fsm_data` and functions to perform its initialization, `fsm_init()`; a single transition, `fsm_step()`; its monitoring, `fsm_monitor()`, and eventual internal variable updates, `fsm_debug()`.

4. Extended FSMs

Finite state machines can have more complex states, with additional variables that hold internal values along machine execution cycles, exactly the same as with the state variable.

A simple, recurrent example is the one of a counter. The EFSM of a counter from 0 to *M*–1 has a start and a counting state, but requires an additional variable to store the actual counter:



This counter is said to be autonomous, as it does not take any input from the outside, i.e. it performs its function as time goes by or, in other words, along a(n infinite) sequence of control loop iterations. The code for such EFSM could be:

```
1:  struct {
2:    enum { c_START, c_COUNT } state;
3:    int C, M;
4:  } c_data;
5:
6:  int c_init() {
7:    c_data.state = c_START;
8:    c_data.C     = -1; /* Unset value: -1 */
9:    c_data.M     =  8; /* Maximum cycle count */
10:   return 0;
11: } /* c_init */
12:
13: int c_step() {
14:   int e; /* e: errcod */
15:
16:   e = 0;
17:   switch( c_data.state ) {
18:     case c_START: c_data.C = 0;
19:                   c_data.state = c_COUNT;
20:                   break;
21:     case c_WAIT:  if( c_data.C == c_data.M – 2 ) { c_data.state = c_START; }
22:                   c_data.C = c_data.C + 1;
23:                   break;
24:     default:      e = -1;
25:   } /* switch */
26:   return e;
27: } /* c_step */
28:
```

(To keep the code short, the “firings control” has been taken out, as well as the reset mechanism.)

In the previous listing, lines 21 and 22 correspond to actions taken at the WAIT state of the counter. Note that, differently from the previous example, the computation of next values for state and variable *C* are not permutable. If next value for *C* is computed first, then the computation for next state will use the next value of *C* instead of the current value.

In this case, though, the order of execution matters. To make all these computations truly order-independent two sets of data can be used for the state machine: the one for current state and the one for the next state. They can be put in a 2-position vector of state machine data sets, with two symbols to identify the current (CURR) and the next (NEXT) set. Following the example of the counter, the updated program is:

```
1:  enum period_e { CURR = 0, NEXT = 1 };
2:
3:  struct {
4:    enum { c_START, c_COUNT } state;
5:    int C, M;
6:  } c_data[2];
7:
8:  int c_init() {
9:    c_data[CURR].state = c_START;
10:   c_data[CURR].C     = -1; /* Unset value: -1 */
11:   c_data[CURR].M     =  8; /* Maximum cycle count */
12:   return 0;
13: } /* c_init */
14:
15: int c_step() {
16:   int e; /* e: errcod */
17:
18:   e = 0;
19:   switch( c_data[CURR].state ) {
20:     case c_START: c_data[NEXT].C = 0;
21:                   c_data[NEXT].state = c_COUNT;
22:                   break;
23:     case c_WAIT:  c_data[NEXT].C = c_data[CURR].C + 1;
24:                   if( c_data[CURR].C == c_data[CURR].M – 2 ) {
25:                     c_data[NEXT].state = c_START;
26:                   } /* if */
27:                   break;
28:     default:      e = -1;
29:   } /* switch */
30:   return e;
31: } /* c_step */
32:
```

Note that `c_init()` sets the values for the current data set and that it is in `c_step()` where the values for the next period are computed. As they are stored into the NEXT position of the c_data vector and all computations are done in terms of values in the CURR positions, the order of execution of instructions at state cases is now irrelevant.

The simulator motor does not essentially change from the previous one but for adding the updating of current values at the end of every simulation step:

```
33: int main() {
34:   int errcod, quit, tick, debug;
35:   char control;
36:
37:   tick = 0;
38:   c_init();
39:   quit = 0;
40:   debug = 0;
41:   errcod = 0;
42:   while( errcod == 0 && quit == 0 ) {
43:     printf( "sim: cycle %07i:\n", tick );
44:     c_monitor();
45:     control = sim_read_input();
46:     if( control == 'D' ) { debug = 1; } else { debug = 0; }
47:     if( debug == 1 ) { quit = c_debug(); }
48:     if( control == 'Q' ) { quit  = 1; }
49:     if( quit == 0 ) {
50:       read_input();
51:       errcod = c_step();
52:       write_output();
53:       c_data[CURR] = c_data[NEXT]; /* move to next step by */
54:       tick = tick +1;              /* making it current    */
55:     } /* if not quitting */
56:   } /* while */
57:   return errcod;
58: } /* main */
```

Functions `c_init()`, `c_monitor()`, and `c_debug()` must be adapted to update the values of `c_data[CURR]`.

View publication stats