State Machines

Alexander Nelson
September 23, 2019

University of Arkansas - Department of Computer Science and Computer Engineering
Time-ordered behavior – Outputs depend on order of input
Many real-world computation challenges involve time-ordered behavior
Many programming languages are built to perform sequential computation
Good at data processing, not meant for time-ordered behavior

C follows a sequential computational model
Each statement executed one after another
Event-driven programming – Input events drive programs/threads
Typically a main loop that listens for events, and calls “callback” function when event is detected

Can be implemented using hardware interrupts
State Machines

Computation model to capture time-ordered behavior
State Diagram – drawn model of the state machine

State machine for turnstile operation (or Aldi shopping cart)
Initial State

Initial State – Defines the behavior of the system on startup
Initializes variables, sets outputs, prepares for input
Typically depicted as $S_0$ or with external arrow pointing in
State – a description of system status while waiting to execute a transition

Each state has a set of actions and transitions

Actions – modifications to internal variables or outputs on entry to state

e.g. S0 turns off LED0, S1 turns on LED0
Transitions – Change in status of the internal system
Reflected by transition from one state to another

Transition can re-enter the same state
Example: vending machine – add coin, but still not enough to buy
would stay in the same state with updated “value” variable
Deterministic FSM – Transitions in a state must be **mutually exclusive**

i.e. An FSM is only in one state at a time

Non-deterministic FSM – An input can lead to one, more than one, or no transition

Can use powerset construction algorithm to build a deterministic FSM from a non-deterministic FSM

Because of this, we will only look at deterministic FSMs for this course
In C, transition conditions are evaluated sequentially
e.g. if(A0 && A1) ... else if(!A0 && !A1) ... else()
This process takes a non-zero amount of time

Tick – time between evaluation of conditions
Ticks should be faster than input can change to avoid missed input

If all conditions evaluate false, state implicitly transitions to itself
Good practice to make explicit
To implement FSM in C, one needs three components:

- State variable declaration
- Tick function
- Main loop that calls tick function
 Enums are useful for defining states
 e.g enum elevatorStateList = {E_INIT_STATE,
 BUTTON_PRESSED, ...}

 These labels are descriptive, can be used for switch statements in
 Tick function
Tick function executes different set of code depending on current state

switch/case statement with enum helps conditional execution

Each state will evaluate transition conditions and actions
Default Case

If you are using a switch/case statement for the tick function, the default case shouldn’t ever be called. Good practice to use it as a way to reinitialize system in case state variable is corrupted.
The main() function is the entry point of the C program
This can be thought of as the initial state

Can use this state to initialize inputs/outputs/variables needed by the state machine
Set state to an entry state after initialization
Maintaining Variables

Often, states will want to share variables & have variables exist between ticks
Example: Vending machine

Initial state sets value = 0
Count state adds value of coin to value
Purchase state reduces value by cost of item
Refund state gives coins back until value == 0
How can these variables be maintained?
How can these variables be maintained?

Global variables
Disadvantage – Can be modified outside of the tick function

Static variables
Disadvantage – Can only be modified in tick function

Choose variable scope that makes sense in your application
What about Mealy FSMs

Mealy FSM – allows actions on transitions as well as on states

(a) Moore actions only

(b) Mealy actions too

(c) Wrong
Capturing the behavior of a system is difficult
For complex systems, the task can be daunting

Process:

1. Start by defining obvious states
   • List actions if known
2. Add transitions between states for given behaviors
3. Check behavior of FSM & iterate
   • Will discover additional transitions or states
   • Think about edge cases & errors
Testing an FSM

Testing all inputs of an FSM may be intractable

**Test Vectors** – input combinations for testing

Test each of the state transitions at least once
Make sure border cases are represented

Include additional test vectors of each type
Black-box/White-box testing

Black-box testing – Only check for valid output from test vector
White-box testing – Examine proper state transitions & internal variables during operation

White-box testing more likely to find issues
Requires higher overhead in developing test mechanisms
The book defines the process of defining a FSM, and writing it into C as the **capture/convert process**

The process is always the same:

1. Capture – Define the process in terms of an FSM
2. Convert – Represent the FSM through C code
3. Iterate – Any changes that need to made should start at the FSM level, not the C level
There are more formal definitions of FSMs & other types of modes (UML state machines)
Dataflow models – Good for digital signal processing (DSP) applications

Mealy & Moore models are good enough for defining most embedded applications
Synchronous FSMs
Time-Interval Behavior

Time-Interval behavior – Events must be separated by specific intervals of time

Examples:

- Blinking LED
- Stop Light
- Servo-motors
FSMs can easily be extended to Synchronous FSMs

FSM Tick() function takes a small amount of time
Instead, Tick() function can be set to a specific rate (e.g. 100ms)

Actual Tick() functionality should happen at the beginning of clock period, and take a small amount of time
Synchronous FSM Intervals

Syncronous FSMs are often used for two different behaviors:

- **Sampling inputs**
  Example: Take temperature reading every 5 seconds

- **Measure time between inputs**
  Example: Tire rotation to detect speed of vehicle
Example Code

```
B0 = 1;
```

```
B0 = 0;
B1 = 0;
```
Example Code

TickFct_State_machine_1() {
    switch(SM1_State) { // Transitions
        case -1:
            SM1_State = SM1_s1;
            break;
        case SM1_s1:
            if (A0==0) {
                SM1_State = SM1_s2;
            } else if (A0==1) {
                SM1_State = SM1_s3;
            }
            break;
        case SM1_s2:
            if (A0==00) {
                SM1_State = SM1_s3;
            } else if (A0==1) {
                SM1_State = SM1_s1;
            }
            break;
        case SM1_s3:
            if (A0==0) {
                SM1_State = SM1_s1;
            } else if (A0==1) {
                SM1_State = SM1_s2;
            }
            break;
        default: // ADD default behaviour below
            SM1_State = SM1_s1;
    } // Transitions
}

switch(SM1_State) { // State actions
    case SM1_s1:
        B0=1;
        break;
    case SM1_s2:
        B1 = 1;
        break;
    case SM1_s3:
        B0 = 0;
        B1 = 0;
        break;
    default: // ADD default behaviour below
        break;
} // State actions
unsigned char SM1_Clk;
void TimerISR() {
    SM1_Clk = 1;
}

int main() {
    const unsigned int periodState_machine_1 = 500;
    TimerSet(periodState_machine_1);
    TimerOn();

    SM1_State = -1; // Initial state
    B = 0; // Init outputs

    while(1) {
        TickFct_State_machine_1();
        while(!SM1_Clk);
        SM1_Clk = 0;
    } // while (1)
} // Main
What about multiple intervals?

Some systems will have different interval requirements

Example:
Stoplight turns Green after 25 seconds, yellow after 20 seconds, red after 4 seconds

How do you handle this?
Counter Variables

Choose a tick frequency that is evenly divides into all tasks
Greatest Common Divisor – Good choice for tick frequency e.g. 1 second for above example
Trigger events based on counter variables

Example:

```c
void tick()
{
    i += 1;
    if(i==25){Green = 1; Red = 0;}
    else if(i == 45){ Green = 0; Yellow = 1;}
    else if(i == 49){ Yellow = 0; Red = 1; i = 0}
}
```
Concurrent State Machines
Tasks

Task – Unique continuously executing behavior
e.g.

- Flashing LED
- Sample Sensor Input at given frequency
- Refresh LCD Display

Concurrent Tasks – Tasks that execute during the same time window
Many embedded systems are composed of concurrent tasks

Handling concurrency is difficult:

- Mostly single-core CPUs
- Limited resources for task switching
- Full OS not common
Concurrent State Machines

Block Diagram – Used to demonstrate systems composed of concurrent SMs

Each state machine has its own set of states (including initial state)
Each state machine controls separate output
Figure 5.1.1: LedShow.
Shared Input/Variables

Some systems require sharing between concurrent tasks
Example: Smart Stop Light

Concurrent separate state machines:

- Camera to detect presence
- Stoplight controller
Why separate tasks instead of one big state machine?

“Separation of concerns” – Let each task take care of its behavior
Build system from multiple concerns – Abstract some details when assembling big system
Exercise:
Capture the Smart Stoplight State Machines

What variable(s) need to be shared between machines?
Reading can occur from multiple tasks
Read frequency depends on:

- Missing data important?
- Write frequency
- When is the variable needed?

Only one task should be responsible for writing shared variable
Easy to create undefined behavior if variable overwritten before used
Conversion process for Concurrent Tasks is similar to single task

1. Create a separate Tick function for each task
2. If they have separate tick frequency, clock divide
Serialization of concurrent tasks is called multi-tasking. Execution of each in every period is called “round-robin” task execution.

Round-Robin execution requires that all tasks complete quickly. Why?
Sequential Code in Multi-Task

A sequential block can be considered as a single state SM with a self-transition.
Block needs to run to completion in the given time-period.

Figure 5.3.3: Converting the sequential code task to a single-state synchSM.

```c
CountFour

S0

unsigned char cnt;

cnt = 0;
for (i = 0; i < 8; i++) {
    if (get_bit(A, i) {
        cnt++;
    }
}
B1 = (cnt >= 4);
```
Concurrent SM Variable Scope/Lifetime

Task variable scope/lifetime depends on usage

- Shared variables should have global scope
- Local variables that need to persist should be declared static
- Local variables that can reset can either be static or auto

What is the benefit of static for variables that don’t need to persist?
Keeping Distinct Behaviors Distinct

Attempting to merge distinct behaviors results in complicated programs
Difficult to understand what each portion does

Exercise: Design state machine that: Plays a tone for 1 second, and blinks an LED every 100ms for 1s after A0 is pressed
Communicating between tasks is essential to synchronize data

Communication types:

- **Synchronous** – Task must sample global variable at certain frequencies
- **Handshake** – Request/Acknowledge pattern with two 1-bit global variables
- **Queues & Message Passing**
Handshake

Behavior:

- Requester raises Req flag
- Servant task performs action and raises Ack flag
- Requester lowers req flag to acknowledge receipt
- Servvent lowers ack flag
Queues & Message Passing

Message Passing – Tasks communicate through packets (messages) of data/commands

1https://rubyplus.com/articles/4761-Ruby-Basics-Message-Passing
Queues & Message Passing

Message should persist until receiver can read message
i.e. Global variable not message passing – Can be overwritten
before read

Need data structure where messages can be placed
Queue – Data structure with FIFO behavior

First In First Out

Messages are pushed to the back of the queue by sender
Messages are popped from the front of the queue by receiver
Example queue for pushing/popping integers

A four-integer queue
Queues w/ State Machine

DetectButton
unsigned char b;

detectButton{
  unsigned char btn;
  switch ((A&0x3E) >> 1) {
    case 0x00: btn = 0; break;
    case 0x01: btn = 1; break;
    case 0x02: btn = 2; break;
    case 0x04: btn = 3; break;
    case 0x08: btn = 4; break;
    case 0x10: btn = 5; break;
    default: btn = 0; break;
  }
  return btn;
}

Period: 100 ms

DetectSequence
Period: 200 ms

const unsigned char seq[3] = {5,2,3};
unsigned char i;

b = DB_GetBtn();

(c==0)
QPUSH(b)

(b==0)

(b==0)

b = DB_GetBtn();

Start

Init
B1 = 0;
i = 0;

(bn_g==0)&&(i==2)/B1=!B1;

Wait

！QEmpty()

Check

！(i==2)/i++;

！QEmpty()
c = QPop();

c == seq[i]

Match
Queue Implementation in C

Required Contents

- Data Item – Can be primitive or struct
  This is the message that is being passed
- Outer queue structure – Can be array/linked list/circular buffer of messages
- QFull() – Returns boolean value if queue is full
- QEmpty() – Returns boolean value if queue is empty
- QPush() – Checks Queue not full, push new item into queue
- QPop() – Check queue not empty, pops new item from queue
- QPrint() – Prints queue, for debugging purposes
Push/Pop

Push/Pop add or remove contents from the queue
If using array structure, pop will require shifting contents to the front
Circular buffer keeps array structure, but maintains pointers to head & tail
Interrupts may disrupt push/pull operations and corrupt data items

**DisableInterrupts()** - Prevent the current operations from being interrupted
Should be only the critical portion of code
– Called atomicity

**EnableInterrupts()** – Allows interrupts again