

Chapter 3 - State Machines

Section 3.1 - Time-ordered behavior

Time-ordered behavior is system functionality where outputs depend on the order in which input events occur. For example, an electronic lock may require a user to press and release button A1, then A0, then A2 to unlock a door. A toll booth may raise a toll gate when the booth operator presses button A0, then keep the gate up as long as a car is detected by sensor A1 near the gate. For each system, the essential behavior involves not just input/output values but also the *order* of those input/output values over time.

Like most programming languages, C was not designed for time-ordered behavior. C uses a **sequential instructions computation model**, wherein statements (instructions) in a list are executed sequentially (one after another) until the list's end is reached. A sequential instructions model is good for capturing algorithms that transform given input data into output data, known as **data processing behavior**. However, the sequential instructions model is poorly-suited for capturing time-ordered behavior.

For example, consider a system on a carousel (merry-go-round) that increments B whenever the carousel rotates once, detected by a sensor that briefly pulses A0 for each rotation. The following RIMS code captures the system's behavior.

P

Participation Activity

3.1.1: Pulse counting code for a carousel

Start

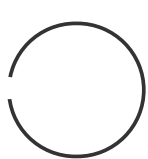
```
#include "RIMS.h"

void main()
{
    B = 0;
    while (1) {
        while (!A0);
        B = B + 1;
        while (A0);
    }
}
```

Carousel (top-view)

Sensor

A0: 1



01/ PERSONAL USE ONLY

The code's statements like `while (!A0);` may look unusual to a beginning embedded programmer. The statement loops as long as A0 is 0, and the immediate ; means no statements execute within the loop. Because C isn't designed to capture such time-oriented behavior, the code is slightly awkward, but understandable. However, the code becomes less understandable as more time-oriented behavior is introduced, such as having a button A1 that resets B, as shown below.

Figure 3.1.1: Extended carousel code: Becoming hard to understand due to sequential instructions model not made for capturing time-oriented behavior.

```
#include "RIMS.h"

void main()
{
    B = 0;
    while (1) {
        while (!A1 && !A0);
        if (A1) {
            B = 0; // Reset
        }
        else {
            B = B + 1;
            while (A0);
        }
    }
}
```

Glenn Lopez
glennlopez@gmail.com
ProgrammingEmbeddedSystemsR10
01/13/17 02:47 am
PERSONAL USE ONLY

The code is becoming harder to understand. The text itself is simple, but how the code interacts with the inputs over time is not obvious, and requires plenty of mental execution to understand the system's behavior. With even more time-oriented behavior introduced, the code may become a spaghetti-like mess whose behavior is extremely hard to understand.

The lesson is this: Capturing time-ordered behavior directly into C's sequential instructions computation model is challenging. Instead, a computation model better suited for capturing time-ordered behavior is needed. State machines, introduced in another section, is one such model.

Glenn Lopez
glennlopez@gmail.com
ProgrammingEmbeddedSystemsR10
01/13/17 02:47 am
PERSONAL USE ONLY

P

Participation
Activity

3.1.2: Time-ordered behavior.

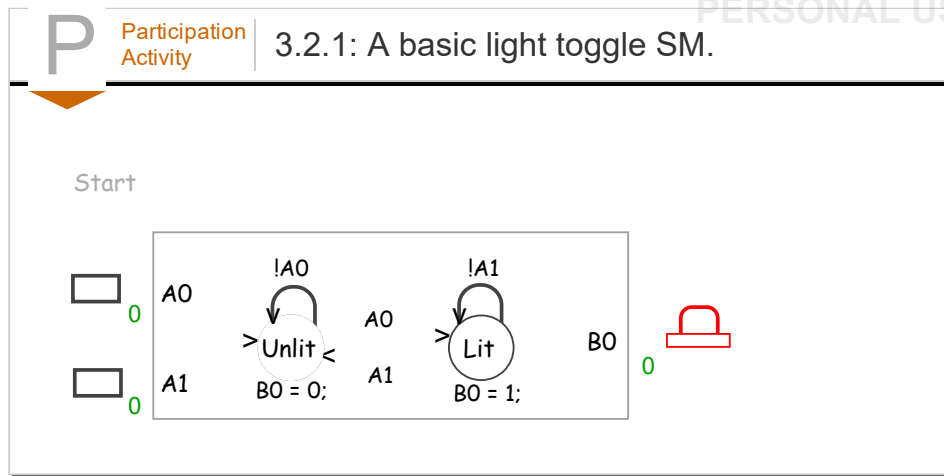
Indicate whether each is more time-ordered behavior or data-processing behavior.

#	Question	Your answer
1	Raise a toll-gate, wait for a car to pass, lower the toll gate.	Time-ordered
		Data-processing
2	Find the maximum value in a set of 100 integers.	Time-ordered
		Data-processing
3	Given two four-bit inputs, compute their sum and average as two four-bit outputs.	Time-ordered
		Data-processing
4	When a person is detected approaching the front of a door, automatically open the door until sensors in front of and behind the door no longer detect anybody.	Time-ordered
		Data-processing
5	A wrong-way system has 10 sensors on a freeway offramp. If a car is driving the wrong way, the sensors will detect a car in the opposite order as normal. The system should flash a "Wrong way" sign and notify the police.	Time-ordered
		Data-processing

Section 3.2 - State machines

A **state machine** is a computation model intended for capturing time-ordered behavior. Numerous kinds of state machines exist. Common features of state machines are a set of inputs and outputs, a set of states with actions, a set of transitions with conditions, and an initial state. A drawing of a state machine is called a **state diagram**.

Consider a simple time-oriented system that turns on a light (by setting $B0 = 1$) if a user presses a button ($A0$ is 1 when pressed). The light stays on even after the button is released. Pressing a second button ($A1$ is 1) turns the light off. The following figure captures that behavior as a state machine.

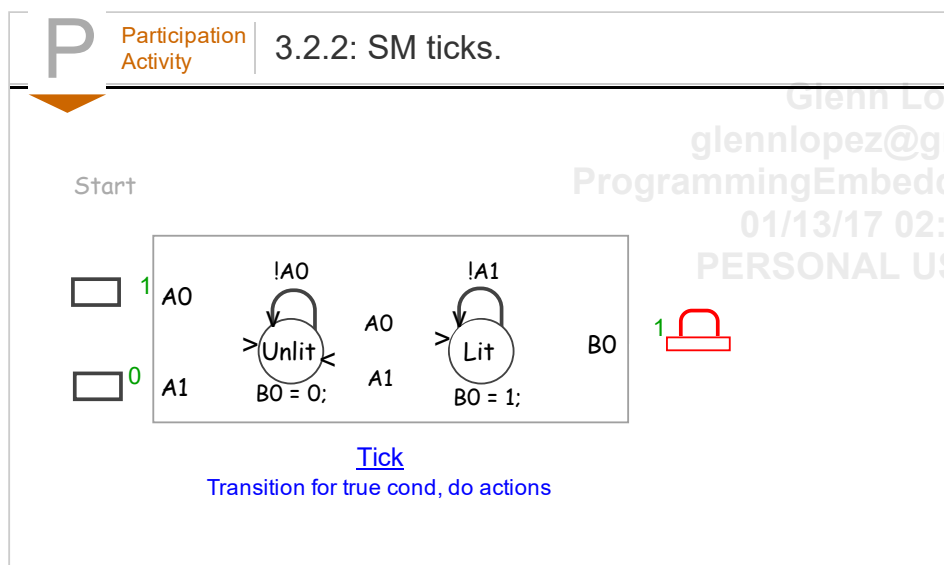


The above animation shows a state machine with inputs A0 and A1 and output B0 (each 1 bit), and the states Unlit (with action B0 = 0) and Lit (with action B0 = 1). The state machine has four transitions with conditions !A0, A0, !A1, and A1, and the initial state is Unlit (denoted by the special "initial transition" arrow).

A system described by a state machine executes as follows. At any time, the system is "in" some state, called the **current state**. Upon starting, the transition to the initial state is taken and that state's actions are executed once. The following process then occurs, called a **tick** of the SM:

- A transition T leaving the current state and having a true condition is taken
- Transition T's target state has its actions executed once and becomes the current state

The ticking process repeats. Each tick takes a tiny but non-zero (perhaps nearly-infinitesimally small) amount of time, during which no event is assumed to occur. Ticks are assumed to occur at a much faster rate than input events, so no input events are missed. The following animation illustrates SM ticking.



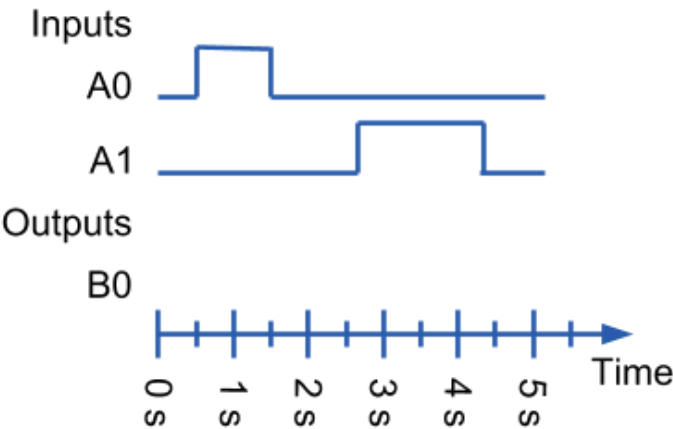
For the above example, upon startup the system takes the transition to state Unlit and executes $B0 = 0$ once. For subsequent ticks, if $A0$ is 0 then the system takes the transition back to state Unlit and executes $B0 = 0$ again. At some time, if $A0$ is 1 then the system takes the transition to state Lit and executes $B0 = 1$. The system stays in that state until $A1$ becomes 1, at which time the system takes the transition back to state Unlit.



Participation Activity

3.2.3: Tracing execution of an SM.

Given the following timing diagram and the above light on/off SM, determine the value of $B0$ at the specified times.



#	Question	Your answer
1	0 s	1
		0
2	1 s	1
		0
3	2 s	1
		0
4	3 s	1
		0
5	4 s	1
		0

6	5 s	1
		0

For the above input sequence, the light was on from about 0.5 seconds when A0's button was pressed, until about 2.5 seconds when A1's button was pressed.

If none of the current state's transitions has a true condition for a given tick, an implicit transition back to the state itself is taken (thus causing the state's actions to execute each such tick). Good practice, however, is to have an explicit transition point back to the same state with the proper condition, rather than relying on the implicit transition from a state to itself. The above system has an explicit transition with condition !A0 from Unlit back to Unlit, for example, making very clear what happens when in state Unlit and A0 is 0.

For a state machine to be precisely defined, transitions leaving a particular state should have **mutually exclusive** transition conditions, meaning only one condition could possibly be true at any time (otherwise, which of two transitions with true conditions should be taken? The state machine becomes **non-deterministic** in that case). For example, state Unlit transitions have conditions A0 and !A0, only one of which can possibly be true at any time.

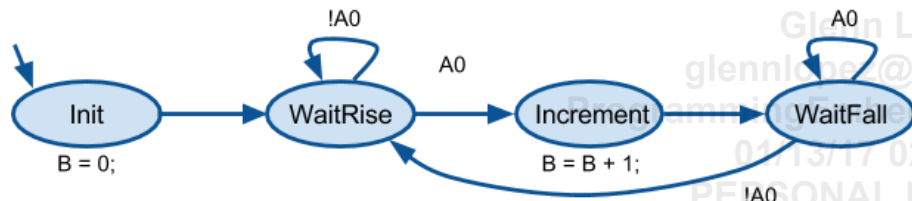
A transition may be specified to have a condition of "true" or 1, meaning the transition should always be taken. That transition of course should be the only one leaving a particular state, else mutual exclusivity would not exist. A common shorthand notation omits the "true"; a transition with no condition is known to have a true condition.

A state may have multiple actions, such as B0 = 1; B1 = 1; or may have no actions at all. A state's actions execute once each time that a tick takes the state machine to that state, even if a transition points back to the same state.

We will use a particular form of a state machine model, referred to in this material just as an **SM**, intended for creating C programs that support time-ordered behavior. An SM uses declared C variables rather than explicit inputs and outputs; the lone exception is the use of RIMS' implicitly-declared A and B input and output variables though. The SM's state actions consist of C statements, and the SM's transitions consist of C expressions. Variable values (such as B0 = 1) persist between ticks.

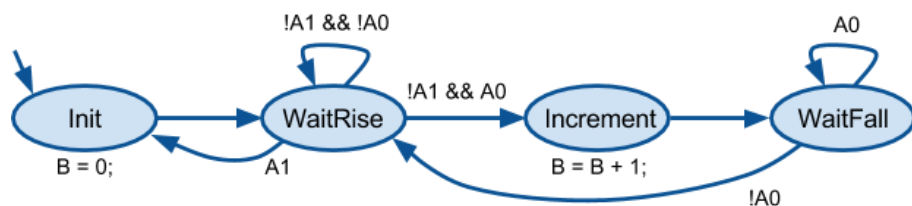
Using an SM, the earlier section's pulse counting system for a carousel can be captured as follows. Note how the SM more clearly defines the time-oriented behavior, versus the earlier section's awkward C code.

Figure 3.2.1: Pulse counting SM.



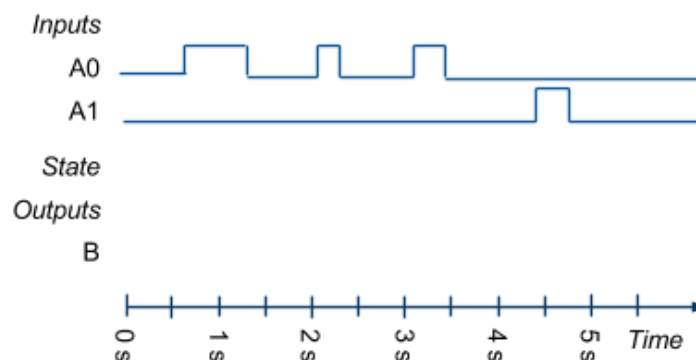
A reset behavior that returns the state machine to the initial state when A1 is pressed can be easily added, as shown below.

Figure 3.2.2: Pulse counting with reset SM.



The following timing diagram shows a sample sequence of inputs to the above SM.

Figure 3.2.3: Sample timing diagram for pulse counter with reset.



P

Participation
Activity

3.2.4: Tracing the pulsing counting SM.

Given the above SM input sequence, type the SM's current state at the specified times. At time 0 ms, the answer is: Init.

#	Question	Your answer
1	0 s	<input type="text"/>
2	0.5 s	<input type="text"/>
3	1 s	<input type="text"/>
4	1.5 s	<input type="text"/>
5	What is the integer value of B at time 4 s?	<input type="text"/>
6	What is the integer value of B at time 5 s?	<input type="text"/>

Try 3.2.1: Trace the current state.

Complete the above timing diagram by showing the current state and also the B0 signal value.

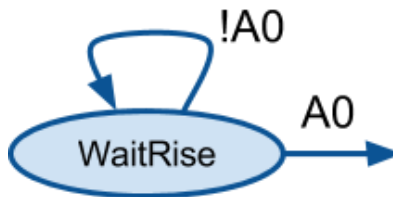
Notice that the SM model and the C code from an earlier section have the same behavior. However, the SM more explicitly captures the desired time-ordered behavior. This straightforwardness can be further seen by trying to extend the SM.

Try 3.2.2: Extend pulse counter to detect a threshold.

Extend the pulse counter SM to set B7 = 1 to indicate when the value of B reaches 99 (for the carousel system, such an indication may tell the ride operator to do a routine safety check). Note that 99 requires only the lower 7 bits of B, so B7 can be used for such indication. When 99 is reached, counting stops until a rising even on A1 resets the count.

The following pattern is common in SMs for detecting a *rising* edge of a signal. The SM stays in the state while A0 is 0. When A0 becomes one, the transition is taken to another state. Detecting a falling edge is similar, with the condition swapped.

Figure 3.2.4: Pattern to detect rising edge of a bit signal.



Glenn Lopez

glennlopez@gmail.com

ProgrammingEmbeddedSystemsR10

01/13/17 02:47 am

PERSONAL USE ONLY

Glenn Lopez

glennlopez@gmail.com

ProgrammingEmbeddedSystemsR10

01/13/17 02:47 am

PERSONAL USE ONLY

P

Participation
Activity

3.2.5: State machines.

#	Question	Your answer
1	An SM tick consists of:	Executing the current state's actions and then transitioning to the next state.
		Transitioning to the next state and executing that state's actions an unknown # of times.
		Transitioning to the next state and executing that state's actions once.
2	How many SM ticks occur per second?	0
		1
		Many
		Infinite
3	In the SM model, which is true about ticks and input events?	Input events may occur faster than ticks.
		Two events may occur between ticks.
		An event may occur in the middle of a tick.
		Ticks occur faster than events.

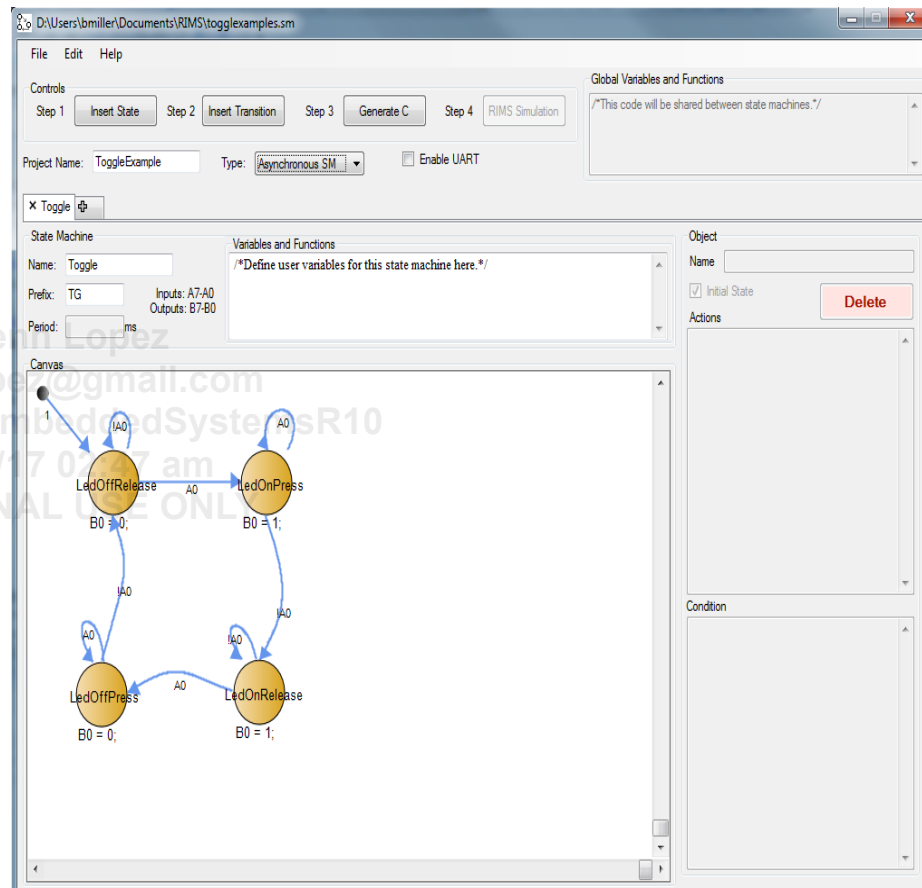
Exploring further:

- [Wikipedia: Finite state machine](#)
- [Wikipedia: State diagram](#)

Section 3.3 - RIBS

The **RIBS** (Riverside-Irvine Builder of State machines) tool supports graphical state diagram capture of SMs.

Figure 3.3.1: RIBS.



A user can insert states and insert transitions between states. The user can click on a state and write C code for that state's actions (in the text box on the right). Likewise, the user can click on a transition and write C conditions (bottom-right text box). The SM begins by transitioning to an initial state (from the shown black dot). Pressing "Generate C" automatically generates C code for RIBS. Pressing "RIMS Simulation" automatically starts RIBS and runs the generated C code on RIBS, where the user can set input switches and observe output LEDs, while RIBS highlights the currently-executing state.

Try 3.3.1: Light toggle SM in RIBS.

Run the RIBS tool and create the light toggle state machine that was introduced in an earlier section. Select the "Asynchronous SM" option from the 'Type' dropdown menu (these items will be described later). Press "Generate C" and then press "RIMS Simulation", which causes the RIMS tool to automatically execute. With both the RIBS and RIMS tools visible on your screen, click on the A0 and A1 switches on RIMS and see how the current state changes in RIBS.

Glenn Lopez

glennlopez@gmail.com

ProgrammingEmbeddedSystemsR10

01/13/17 02:47 am

PERSONAL USE ONLY

Try 3.3.2: Three-LED sequencer in RIBS.

Capture an SM for a three-LED sequencer. Initially B0's LED is on. When A0 rises (changes from 0 to 1), B0 turns off and B1 turns on. When A0 rises again, B1 turns off and B2 turns on. When A0 rises again, B2 turns off and B0 turns on again. And so on. Try capturing the behavior in C directly first, and then instead capture it as an SM (use a distinct state for each distinct output situation). Test the SM using RIBS.

Try 3.3.3: Three-LED sequencer extended in RIBS.

Extend the three-LED sequencer by reversing the sequence if A1=1, else following the earlier sequence if A1=0. Try extending the C code first (hard!), and then the SM.

Glenn Lopez

glennlopez@gmail.com

ProgrammingEmbeddedSystemsR10

01/13/17 02:47 am

PERSONAL USE ONLY

Glenn Lopez

glennlopez@gmail.com

ProgrammingEmbeddedSystemsR10

P

Participation
Activity

3.3.1: RIBS.

#	Question	Your answer
1	RIBS draws a user's inserted state as a circle.	True
		False
2	To insert a transition, RIBS requires a user to click on the end state and then the start state.	True
		False
3	In RIBS, the user indicates the initial state by drawing a transition from no state to a state.	True
		False
4	RIBS translates the SM to C and then runs that C code in RIMS.	True
		False

Glenn Lopez

glennlopez@gmail.com

ProgrammingEmbeddedSystemsR10

01/13/17 02:47 am

PERSONAL USE ONLY

P

Participation Activity

3.3.2: RIBS.

Simulate

Pause

Insert state

Ins

LoHi

+

A0 0

A1 0

A2 0

A3 0

A4 0

A5 0

A6 0

A7 0

A = 0

Lo

B0 = 0;

/ B=0; !A0

!

A0

A0

!

A0

Use test vectors

Export to RIMS

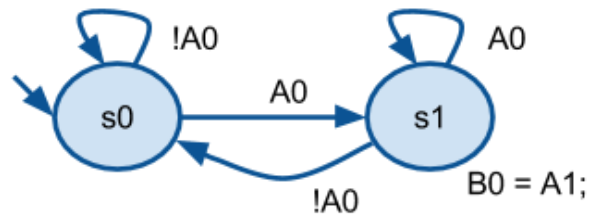
Paste design he

Section 3.4 - Implementing an SM in C

Because microprocessors typically have C compilers but not SM compilers, implementing an SM in C is necessary. Using a standard method for implementing an SM to C enhances the readability and correctness of the resulting C code. The following illustrates such a method for the given SM named Latch (abbreviated as LA), which saves (or "latches") the value of A1 onto B0 whenever A0 is 1.

Figure 3.4.1: Method for implementing an SM in C.

SM name: Latch (abbrev: LA)



```

#include "RIMS.h"

enum LA_States { LA_SMStart, LA_s0, LA_s1 } LA_State;

void TickFct_Latch()
{
    switch(LA_State) { // Transitions
    case LA_SMStart: // Initial transition
        LA_State = LA_s0;
        break;
    case LA_s0:
        if (!A0) {
            LA_State = LA_s0;
        }
        else if (A0) {
            LA_State = LA_s1;
        }
        break;
    case LA_s1:
        if (!A0) {
            LA_State = LA_s0;
        }
        else if (A0) {
            LA_State = LA_s1;
        }
        break;
    default:
        LA_State = LA_SMStart;
        break;
    } // Transitions

    switch(LA_State) { // State actions
    case LA_s0:
        break;
    case LA_s1:
        B0 = A1;
        break;
    default:
        break;
    } // State actions
}

void main() {
    B = 0x00; // Initialize outputs
    LA_State = LA_SMStart; // Indicates initial call

    while(1) {
        TickFct_Latch();
    }
}

```

While the code may seem imposing, the code follows a simple pattern:

- State variable declaration
- Tick function
- main loop that calls tick function

State variable declaration: The first line creates a new enum data type `LA_States` defined to have possible values of `LA_SMStart`, `LA_s0` and `LA_s1`. That same code line declares global variable `LA_State` to be of type `LA_States`. **enum** is a C construct for defining a new data type, in contrast to built-in types like `char` or `short`, whose value can be one of an "enumerated" list of values. The programmer provides the enumeration list, as in `{ LA_SMStart, LA_s0, LA_s1 }` above. In C, each item in the list becomes a new constant, with the first item having value 0, the second item having value 1, etc.

Tick function: The SM's **tick function** carries out one tick of the SM. This SM's tick function is named `TickFct_Latch()`. For the current state `LA_State`, the tick function's **first switch statement** takes the appropriate transition to a new current state; if `LA_State` is `LA_SMStart`, the transition is to the SM's initial state. The **second switch statement** then executes the actions for the new current state.

main loop that calls tick function: `main()` first initializes outputs; good practice for any embedded program, whether implementing an SM or not, is to start the main function by initializing all outputs. `main()` then sets the current state to a value of `LA_SMStart`. `main()` then enters the normal infinite "while (1)" loop, which just repeatedly calls the function `TickFct_Latch()` to repeatedly tick the Latch SM.

Try 3.4.1: Follow an SMs execution in C.

Run the above code in RIMS. Press "Break" and then repeatedly press "Step" (setting A0 accordingly), observing how the code executes each tick of the state machine.

The transition switch statement's default case should never actually execute, but good practice is to always include a default case for a switch statement, in case something bad happens. For a state machine, the default case should be included for safety if the state variable ever somehow gets corrupted. The default case commonly has a transition to the initial state, causing the SM to start over rather than getting stuck.

Capturing behavior as an SM and then converting to C using the above method typically results in more code than capturing behavior directly in C. However, *more* code does not mean *worse* code. The C code generated from an SM may be more likely to be correct, may be more easily extensible and maintainable, and has other major benefits that will be seen later.

Figure 3.4.2: Carousel SM in C.





```

#include "RIMS.h"

enum CR_States { CR_SMStart, CR_Init, CR_WaitRise, CR_Increment, CR_WaitFall }

void TickFct_Carousel()
{
    switch(CR_State) { // Transitions
        case CR_SMStart: // Initial transition
            CR_State = CR_Init;
            break;

        case CR_Init:
            CR_State = CR_WaitRise;
            break;

        case CR_WaitRise:
            if (A1) {
                CR_State = CR_Init;
            }
            else if (!A1 && !A0) {
                CR_State = CR_WaitRise;
            }
            else if (!A1 && A0) {
                CR_State = CR_Increment;
            }
            break;

        case CR_Increment:
            CR_State = CR_WaitFall;
            break;

        case CR_WaitFall:
            if (!A0) {
                CR_State = CR_WaitRise;
            }
            else if (A0) {
                CR_State = CR_WaitFall;
            }
            break;

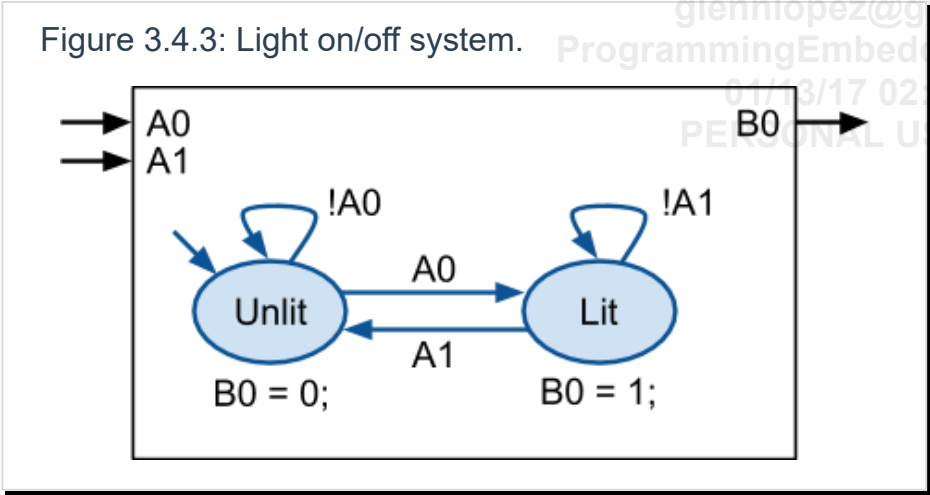
        default:
            CR_State = CR_SMStart;
            break;
    } // Transitions

    switch(CR_State) { // State actions
        case CR_Init:
            B = 0;
            break;
        case CR_WaitRise:
            break;
        case CR_Increment:
            B = B + 1;
            break;
        case CR_WaitFall:
            break;
        default:
            break;
    } // State actions
}

void main() {
    B = 0x00; // Initialize outputs
    CR_State = CR_SMStart; // Indicates initial call
    while(1) {
        TickFct_Carousel();
    }
}

```

For the following question sets, consider the SM below.



P

Participation
Activity

3.4.1: Method for implementing an SM in C.

Strictly adhere to the above SM to C implementation method, including naming conventions. Use `LT_States` as the enumerated type name, and `TickFct_LightToggle` as the tick function name.

#	Question	Your answer
1	Provide the enumerated type definition following the above SM to C method.	<input type="text"/>
2	<code>main()</code> 's <code>while(1)</code> loop will consist of what one statement?	<input type="text"/>

P

Participation
Activity

3.4.2: C code for SM.

#	Question	Your answer
1	The tick function's first switch statement will execute the current state's actions.	True
		False
2	The first switch statement's first case will include: case LT_SMStart: LT_State = LT_unlit;	True
		False
3	The first switch statement's second case will be for the transitions going to state Unlit.	True
		False
4	The tick function's second switch statement will include: case (LT_Unlit): if (!A0) { B0 = 1; }...	True
		False
5	The SM to C method uses an if-else statement for multiple transitions, but could have just used multiple if statements.	True
		False
6	If a state has no actions, the state should be omitted from the second case statement.	True
		False
7	The break statements could be removed from the switch statements without changing behavior, but should be included for clarity.	True
		False
8	A transition from a state back to that same state can be omitted from the first switch statement without changing the SM's behavior.	True
		False

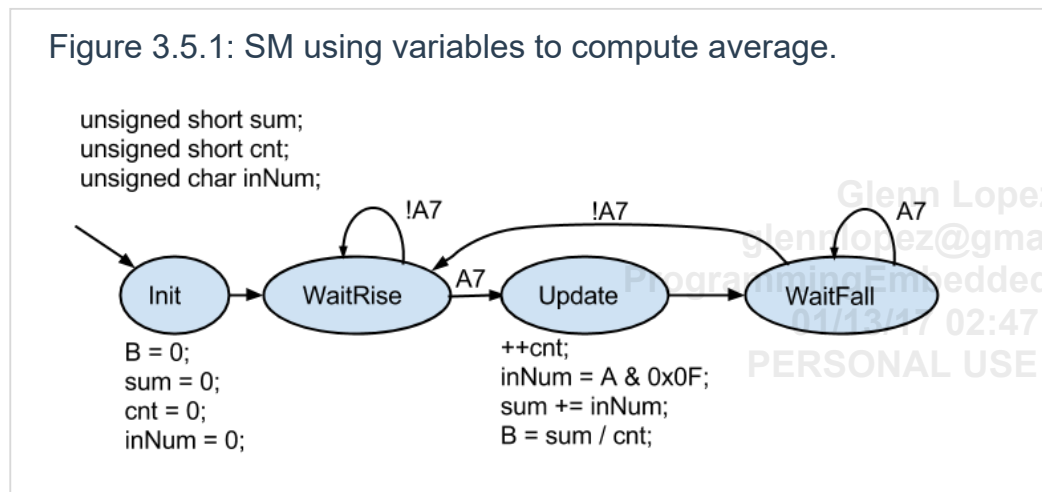
Try 3.4.2: SM to C for the light toggle system.

Strictly following the above method for implementing an SM in C, implement the above light toggle SM in C, and test in RIMS.

Section 3.5 - Variables, statements, and conditions in SMs

Variables

An SM can have variables declared at the SM scope (i.e., not within a state), which can be accessed by all actions and conditions of the SM. For example, the following SM uses several variables to output the average of 4-bit numbers that appear on A3..A0 when A7 rises.



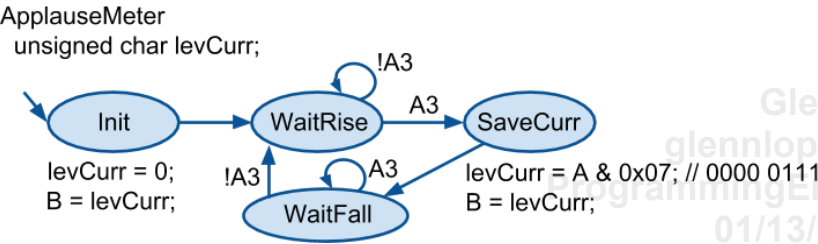
Variable sum maintains the sum of all 4-bit numbers seen so far, defined as a short to reduce likelihood of overflow. Variable cnt counts the number of times A7 has risen. Variable inNum stores A3..A0, that variable used just to improve code readability. (Note: A reset for the above system, perhaps using A6, is omitted for brevity).

In RIMS, the input/output variables like A0 or B can be thought of as having been declared as unsigned char variables.

While the variables could be initialized when declared (e.g., unsigned short sum=0;), good practice is to create a state named Init to carry out initializations of variables and also outputs. In this way, not only are all initializations in one place, but the system can be re-initialized merely by transitioning back to the Init state.

Example 3.5.1: Applause meter.

Consider an applause-meter system intended for a game show. A sound sensor measures sound on a scale of 0 to 7 (0 means quiet, 7 means loud), outputting a three-bit binary number, connected to RIM's A2-A0. A button connected to A3 can be pressed by the game show host to save (when A3 rises) the current sound level, which will then be displayed on B. The system's behavior can be captured as an SM with a variable, as shown.



When converting to C, the SM variables can be implemented as global variables above the SM's tick function.

P

Participation
Activity

3.5.1: SM variables.

#	Question	Your answer
1	A programmer can declare variables within each SM state.	True
		False
2	An SM variable maintains its value across SM ticks.	True
		False
3	Failing to write an SM variable in a particular state causes that variable to become 0.	True
		False
4	When implementing an SM in C, the SM's variables should be declared in the main() function.	True
		False

Try 3.5.1: System to count instances driving without seatbelt.

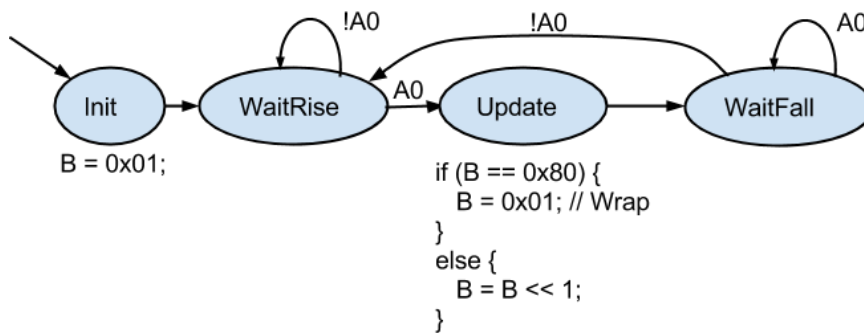
Design a system for an automobile that counts the number of times the car was put into drive (A0 is 1) while the driver's seatbelt was not fastened (A1 is 0). Once the car starts driving, do no further counting until the car is taken out of drive (A0 is 0). While the car is not in drive, a mechanic can view the count by holding a button (A2 is 1) causing the count to appear on B. B is normally 0.

Statements

The statements that can appear in actions can include more than just assignment statements. Any C statements can appear, such as if-else statements, loops, and function calls. However, for this material's purposes, good practice is to ensure that actions don't wait on an external input value, such as `while (!A0) {}`. Waiting behavior should be captured as states and transitions so that all time-ordered behavior is visible at the transition level. Also, a state with actions that wait could cause the SM to violate the basic assumption that SM ticks occur faster than events so that no events are missed.

The following example changes RIMS' output LED pattern from 00000001 to 00000010 to 00000100, etc., each time A0 rises. When 10000000 is reached, the pattern wraps back to 00000001.

Figure 3.5.2: SM using if-else statement to output a shifting pattern.



Try 3.5.2: Extend the applause meter SM.

Extend the above applause meter SM so that if A4 becomes 1 while in the WaitRise state, the system outputs the maximum value seen so far. When A4 is returned to 0, the system outputs the most-recently-saved value as before. Use a variable for the most-recently-saved value and another for the max value, and an if-else statement.

Try 3.5.3: SM with loop.

Create an SM that waits for A0 to rise, upon which the SM counts the number of 1s on A1..A7 and outputs the count on B. That count stays on B until A0 rises again. Use the GetBit function from an earlier section, and a for loop, as the actions of a state that counts the 1s on A1..A7.

Participation Activity

3.5.2: SM statements.

#	Question	Your answer
1	A state's actions may include a for loop.	True
		False
2	A state's actions may include a function call.	True
		False
3	The statement if (A0) {...} should not appear in a state's actions.	True
		False
4	The statement while (A0) {...} should not appear in a state's actions.	True
		False

Conditions

Conditions in an SM should be C expressions. The following rule is important:

- *Exactly one—no more, no fewer—of a state's exiting transitions should have a condition that evaluates to true at a given time.* In this way, the next state for each tick is precisely and clearly defined.

A common error is to create transitions leaving a state whose conditions are not mutually exclusive, like one transition with condition A0 and another with condition A1, both of which could be true simultaneously. Another common error is to create transitions such that sometimes no transition has a true condition, like one transition with condition A0 and another with condition !A0 && A1, which fail to cover the situation !A0 && !A1. Technically, neither situation is actually an error. In the first case, the SM

13/01/2017

Programming Embedded Systems

becomes non-deterministic, because the model does not define which of two true transitions will be taken. In the second case, the SM will implicitly take a transition back to the same state, but explicit transitions are preferable for clarity.

For convenience, the SM's in this material use a condition named **other** to indicate the transition that should be taken if none of the state's normal transition conditions are true. For example, the following system opens a swinging door when a person approaches the front (A0 is 1) and nobody is directly behind the door (A1 is 0). Note that the transition that remains in the WaitPerson state is simply "other" rather than being `!(A0 && !A1)`, which clutters the SM and detracts the readers attention away from the more important transition `A0 && !A1` from WaitPerson to Open. Once opening the door, the system keeps the door open as long as the person is detected near the door as indicated by `A0 || A1`, again using "other" for the opposite condition.

Figure 3.5.3: Door opener system using condition other.

When implementing an SM in C, the "other" transition may be implemented as a last else branch (with no expression) in the state's transitions if-else code.

P

Participation Activity

3.5.3: SM conditions.

Remember that transition conditions are expressions, *not* statements, so should *not* end with a semicolon. For each, write the most direct answer, e.g., write "A0 and A1 are true" as `A0 && A1` (in that order, without parentheses, without `== 1`, etc.)

#	Question	Your answer
1	Write the condition for a transition that should be taken if either A0 or A1 is true.	<div></div>
2	Write the condition that detects that A is greater than or equal to 99.	<div></div>
3	Write the condition that detects that A2A1A0 is 010. Use individual bit variables A2, A1, A0.	<div></div>

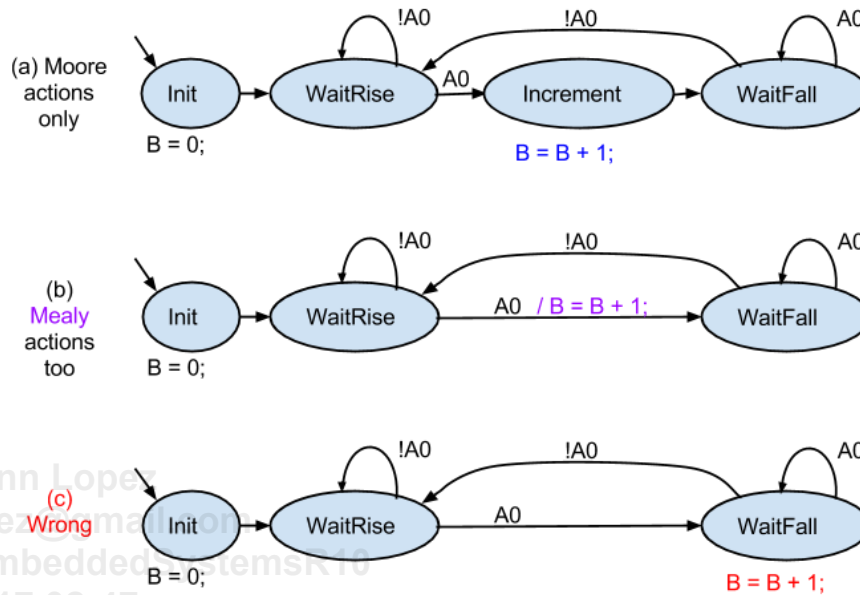
4	A designer intended to have one transition taken if A0 is 1; otherwise, a second transition is taken if A1 is 0 and a third taken if A1 is 1. The designer wrote the conditions A0, !A1, and A1. Fix the <i>second</i> condition.	
5	A designer intended to have one transition taken if exactly one of A0 or A1 is 1, and a second transition taken if both are 0s. The designer wrote the conditions as (A0 && !A1) (!A0 && A1) and as !A0 && !A1. A third transition is missing; write its condition.	
6	A designer has two transitions leaving a state. One transition's condition is (A1 A2 A3). The second transition's condition is "other". What expression does the other represent? Write the expression with the fewest changes to the first expression.	
7	A designer has three transitions leaving a state. One transition's condition is (!A0 && !A1). A second transition's condition is (A0 && A1). A third transition's condition is "other". What expression does the other represent? Start with: !((!A0 ... Don't try to simplify.	

Section 3.6 - Mealy actions

The earlier state machine model associates actions with states only, known as a **Moore-type state machine**. A **Mealy-type state machine** allows actions on transitions too. A Mealy-type SM can make some behaviors easier to capture.

For example, the following figure parts (a) and (b) show SMs that increment variable cnt once for each rising A0. The Mealy SM increments on the transition that detected the rise, and thus avoids the need for a separate state. Note that Mealy actions are shown graphically following a transition's condition and a /.

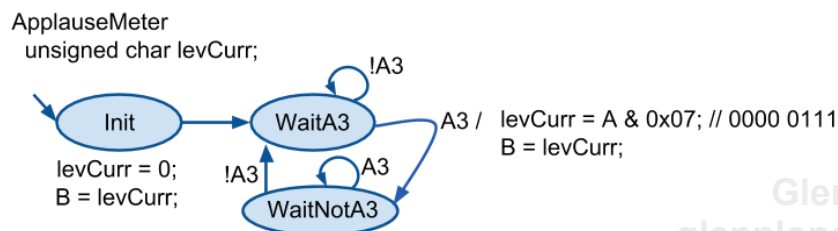
Figure 3.6.1: Mealy SMs can reduce states and more intuitively represent some behavior.



Note that the SM in (c) is wrong, incrementing B repeatedly while A0 is 1. Removing the A0 transition from WaitFall would still be wrong because the SM model would have an implicit transition back to the state if no other transition is true.

The following implements an applause meter system (from an earlier section), using a Mealy action on the transition that detects a rising A3, thus preventing having an additional state.

Figure 3.6.2: Applause meter using a Mealy action.



When translating to C, a transition's actions appear in the transition switch statement, in the appropriate if-else branch.

Try 3.6.1: MealySMToggle.

Capture a toggle light system using an SM. A button connects to A0. B0 connects to a light, initially off. Pressing the button (rising A0) changes the light from off to on. Another press changes the light from on to off. And so on. First try capturing with a Moore SM. Then try with an SM having Mealy actions, and notice that fewer states are required.

P

Participation
Activity

3.6.1: Mealy actions.

#	Question	Your answer
1	A Mealy action occurs at which point during an SM tick?	While checking whether a transition's condition is true or false.
		While taking a transition to the next state.
		Upon entering a new state.
		Before an SM's tick.
2	Can a Mealy action include an if-else statement?	No, only one statement is allowed.
		No, only assignments statements are allowed.
		Yes, but the if and else must both assign the same variable.
		Yes.
3	Integer variable X is 0. A transition with action $X = X + 1$ points to a state with action $X = X + 2$. If that transition is taken during an SM tick, what is X after the tick?	0
		1
		2
		3

Glenn Lopez

glennlopez@gmail.com

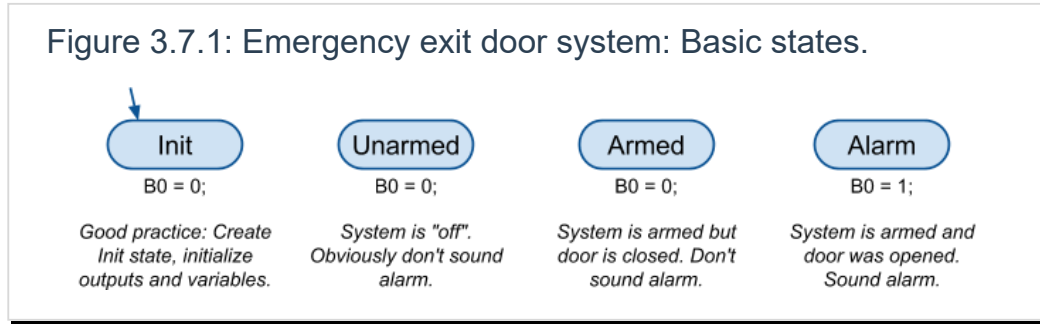
Section 3.7 - How to capture behavior as an SM

Capturing behavior as an SM is an art. The following 3-step process may help.

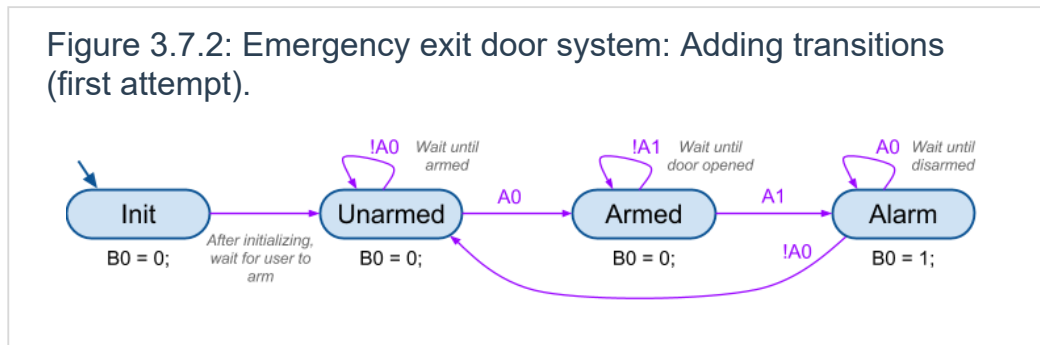
Consider an emergency exit door that sounds an alarm when armed (switch A0 is 1)

and then opened (door A1 is 1).

Step 1: A first step is to list the system's basic states, adding actions if known:



Step 2: A second step of the process is to add transitions to each state to achieve the desired behavior. Clearly the first state to enter after initialization is the Unarmed state, so we add a conditionless (true) transition from Init to Unarmed. We add a transition to stay in Unarmed while A0 is 0, and another transition to go to Armed when A0 is 1. We add a transition to stay in Armed while the door is closed (A1 is 0), and another transition to go to Alarm if the door is opened. Finally, we add a transition to stay in Alarm while the system is still armed (A0 is 1), and when disarmed (A0 is 0) we could go to either Init or Unarmed; Unarmed seems reasonable so we add a transition to there.



Step 3: The third step is to mentally check the behavior of the captured SM. This step may result in adding more transitions, or even more states. Minimally, for each state, we should check that *exactly* one transition's condition will be true, modifying the conditions or adding transitions if necessary. Even more importantly, we should also determine whether the SM's behavior is as desired, by mentally executing the SM and thinking what input sequences might occur. In doing so for the above SM, we consider the possibility that a user might want to disarm the system while armed, and notice that is not possible in the SM. Thus, we may add transitions, such as !A0 from Armed to Unarmed (requiring that the other two transitions be modified, otherwise the state's outgoing conditions would not be mutually exclusive).

When mentally executing, one might focus on a particular state, and then for each input not explicitly on the transitions one might ask: "Does that input's value matter for this state's behavior?" For example, the above system has two inputs, A0 and A1. For state Unarmed, does input A1's value matter? It does not for that state. For state Armed, does input A0 matter? It does, so we need to add transitions. For state Alarm, does A1's value matter? It does not; once the door has been opened, the alarm continues sounding whether the door stays open or closed.

Capturing time-ordered behavior is hard. Good designers will spend much time mentally executing their SM and thinking of possible input sequences that need to be addressed. Refining an SM many times is commonplace.

P

Participation
Activity

3.7.1: Capturing behavior as an SM.

#	Question	Your answer
1	The above process suggests creating one state at a time, creating that state's actions and all the states outgoing transitions, before moving on to create another state.	True
		False
2	Step 3 for the emergency door example determined that a transition with condition A0 is needed from Armed to Unarmed.	True
		False
3	Upon adding a transition !A0 from Armed to Unarmed, the transition A1 from Armed should be changed to to A1 && A0.	True
		False

Try 3.7.1: Exit only doorway.

A doorway is for exit only. Sensors A0, A1, A2 detect a person passing through. A proper exit causes A2A1A0 to be 000, then 100, then 010, then 001, then 000. Any other sequence causes a buzzer to sound (B0=1) until 000.

Design an SM using RIBS that captures the doorway behavior, and simulate using RIMS.

Try 3.7.2: Automatic door.

An automatic door at a store has a sensor in front (A0) and behind (A1). The door opens (B0=1) when a person approaches from the front, and stays open as long as a person is detected in front or behind. If the door is closed and a person approaches from the behind, the door does not open. If the door is closed and a person approaches from the bfront but a person is also detected behind, the door does not open, to prevent hitting the person that is behind.

Design an SM using RIBS that captures the automatic door behavior, and simulate using RIMS.

Try 3.7.3: Dimmer light system.

A dimmer light system has increase (A0) and decrease (A1) buttons. B sets the light intensity, 0 is off, 255 is the maximum intensity, and:

- Pressing both buttons does nothing.
- Pressing both buttons immediately turns the light off.

Design an SM using RIBS that captures the dimmer light behavior, and simulate using RIMS.

Try 3.7.4: Amusement ride.

An amusement park ride has sensor mats on the left (A0) and right (A1) of a ride car. A ride operator starts the ride by pressing a button (A7); each unique press toggles the ride from stopped to started (B0=1) and vice-versa. If anyone leaves the ride car and steps on a sensor mat, the ride stops and an alarm sounds (B1=1). The alarm stops sounding when the person gets off the sensor mat. The only way for the ride to restart is for the operator to press the button again. The ride never starts if someone is on the sensor mat.

Design an SM using RIBS that captures the amusement ride behavior, and simulate using RIMS.

Section 3.8 - Testing an SM

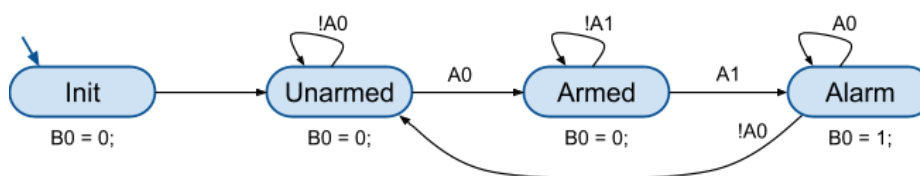
Testing time-ordered behavior requires generating a good set of input scenarios, where a **scenario** is a sequence of inputs that should cause a particular sequence of state. Such testing is in contrast to merely generating a good set of input combinations for a system lacking internal states, each input combination known as a **test vector**.

A testing process for time-ordered behavior may consist of:

1. Decide what *scenarios* to test, including normal cases as well as border cases.
2. Devise a sequence of test vectors to test each scenario.

Consider the following SM for an emergency exit alarm system.

Figure 3.8.1: Emergency exit door system to be tested.



The following lists some scenarios, and corresponding test vectors.

Table 3.8.1: A few test scenarios for the emergency door system.

Scenario.	Starting state in (.).	Expected final state	Expected output (B0)	Test vectors (A1A0)
1	(Init): <i>Alarm should not sound.</i>	Unarmed	0	00
2	(Unarmed): Arm system: <i>Alarm should not sound.</i>	Armed	0	01
3	(Armed): Open door. <i>Alarm should sound.</i>	Alarm	1	11
4	(Alarm): Unarm: <i>Alarm should stop.</i>	Unarmed	0	10
5	(Unarmed): Arm, open door, close door. <i>Alarm should continue to sound.</i>	Alarm	1	00, 01, 11, 01
6	(Alarm): Unarm, arm, unarm: <i>Alarm should not sound.</i>	Unarmed	0	00, 01, 00

Each row indicates the assumed starting state (same as previous row's final state), describes the system's input scenario and the expected system behavior, lists the expected final state and output, and finally lists the test vectors that will generate the scenario. For example, the scenario 2 starts in state Unarmed, then arms to the system, which should cause a change to state Armed and output B0=1. Arming the system is achieved by A1A0 being 01, which is the test vector. Scenario 5 carries out a longer sequence, requiring several test vectors.

Scenario 6 arms and then unarms the system. Testing with the given test vectors will yield the correct final output of 0, but the final state will be Armed rather than Unarmed. Thus, testing uncovers a problem, namely that Armed is missing a transition with condition !A0 going back to Unarmed .

As with testing systems with combinational behavior, testing a system with time-ordered behavior should involve normal cases and border cases. Border cases specifically test unusual situations, like all inputs changing from 0s to 1s simultaneously, or an input changing back and forth from 0 to 1 repeatedly.

When testing an SM, test vectors should ensure that *each state and each transition is executed at least once*. Furthermore, if a state's action code has branches, then test vectors should also ensure that every statement is executed at least once. Even more

ideally, *every path* through the SM would also be tested, but achieving complete path coverage is hard because huge numbers of paths may exist.

In testing terminology, **black-box testing** refers to checking for correct outputs only, as in checking the value of B0 in the above example. In **white-box testing**, internal values of the system are also checked, such as the current state. One can see the advantage of white-box testing in the above example, where the output was correct but the state was not. Further black-box testing might discover the above problem too, but white-box is more likely to detect problems. Of course, white-box testing is harder because a mechanism is necessary to access internal values.

Designing good test vectors can take much effort, both to formulate the scenarios, and to properly create the test vectors. Testing is as important as capturing the SM. Good practice is to plan to spend nearly as much time for testing a system as for designing a system. New programmers rarely follow this practice, believing testing is just a "sanity check" step at the end of design.

P

Participation
Activity

3.8.1: Creating test vectors for the emergency door system.

Match the test vectors with the desired test scenario for the above emergency door system. Assume each set of test vectors is applied immediately after starting the system, so the starting state is always Unarmed. Each vector is for A1A0, so 01 means A1 is 0 and A0 is 1.

01, 11, 01, 11

01, 00, 10

01, 00, 01, 00

01, 10

Drag and drop above item	
	Arm the system, open the door, close the door.
	Arm the system, disarm, arm, disarm.
	Arm the system, disarm the system, open the door.
	Arm the system, then simultaneously open the door and disarm the system.

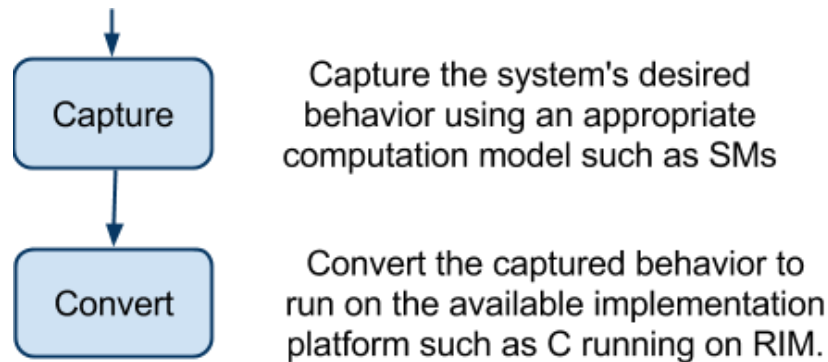
Reset

Section 3.9 - Capture/convert process

The above sections described a two-step process that is common in disciplined embedded programming. The first step is to **capture** the desired behavior using a computation model appropriate for the desired system behavior, such as SMs. The second step is to **convert** that captured behavior into an implementation, such as C that will run on a microprocessor. The conversion is typically very structured and

automatable. The capture/convert process will be used in subsequent chapters for more complex behavior and can result in code that is more likely to be correct, that is more maintainable, and that has many other benefits compared to behavior that is captured directly in C's sequential instruction model. *The capture/convert design process is perhaps one of the most important concepts in disciplined programming of embedded systems.*

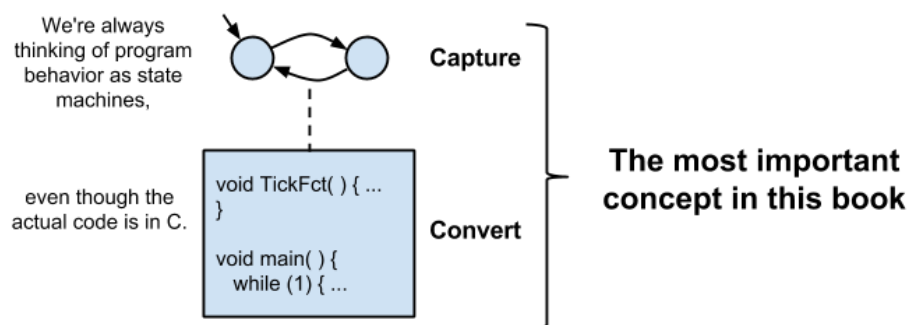
Figure 3.9.1: Capture, convert.



Even in the absence of a tool like RIBS, programmers can (and should) capture time-ordered behavior as an SM, typically drawing the SM on paper first, and then converting to C. All modifications are done by changing the SM (capture), and then updating the C code (convert). An experienced programmer can work with SMs in C without always having to see a state diagram, and can see or draw a state diagram from C code.

The concept of thinking of program behavior as state machines, even though the actual code is in C, is the most important concept in this book.

Figure 3.9.2: Thinking in state machines.



Try 3.9.1: Reverse engineering C code to an SM.

For the below C code, draw the corresponding SM state diagram.

```
#include "RIMS.h"

unsigned char x;

enum EX_States { EX_SMStart, EX_S0, EX_S2, EX_S1 } EX_State;

void TickFct_Example() {
    switch(EX_State) { // Transitions
        case EX_SMStart:
            EX_State = EX_S0;
            break;
        case EX_S0:
            if (1) {
                EX_State = EX_S1;
            }
            break;
        case EX_S2:
            if (A3) {
                EX_State = EX_S2;
            }
            else if (!A3) {
                EX_State = EX_S1;
            }
            break;
        case EX_S1:
            if (!A3) {
                EX_State = EX_S1;
            }
            else if (A3) {
                EX_State = EX_S2;
                x = A & 0x07;
                B = x;
            }
            break;
        default:
            EX_State = EX_SMStart;
            break;
    }

    switch(EX_State) { // State actions
        case EX_S0:
            x = 0;
            B = x;
            break;
        case EX_S2:
            break;
        case EX_S1:
            break;
        default:
            break;
    }
}

void main() {
    EX_State = EX_SMStart; // Initial state
    B = 0; // Init outputs
    while(1) {
        TickFct_Example();
    }
}
```

Note that the RIBS tool does not run a C compiler on the C code in actions/conditions/declarations, but rather just generates a new C program that contains that code. Any syntax errors will only be determined upon running a C

compiler on that program, requiring a RIBS user to correlate the error message to the SM code.

Try 3.9.2: C syntax error in RIBS.

For any working example in RIBS having an action in a state, introduce a C syntax error by removing the semicolon after an action. Save, generate C, then press "RIMS Simulation". Note the error message that is generated, and strive to correlate that with the RIBS SM. Fix the error and this time introduce an error by adding a semicolon after a transition condition, and try running again.

The SM model in this chapter is a basic state machine model specifically intended to aid the capture of time-ordered behavior and for conversion to C. Many other state machine models exist, such as **UML state machines**. Some state machine models have a more formal mathematical basis, but translation to C is more cumbersome and the code harder to maintain. Another common category of computation model involves dataflow models, which are well suited to digital signal processing applications but are beyond our scope.

P

Participation
Activity

3.9.1: Capture/convert process.

#	Question	Your answer
1	"Capture" refers to describing desired system behavior in the most appropriate computation model.	True
		False
2	"Convert" refers to describing desired system behavior directly in C.	True
		False
3	A system has two 4-bit inputs, and continually sets a 5-bit output to their average. The system is best captured as an SM, then converted to C.	True
		False
4	Because an SM can be converted to C, then C directly supports the SM computation model.	True
		False

Exploring further:

- [Wikipedia: UML state machine](#)
 - statecharts.org
 - [Wikipedia: Kahn process networks](#)
-