



The Embedded Systems Experts

## How to Code a State Machine in C or C++

By Miro Samek

*A state machine is any object that behaves differently based on its history and current inputs. Many embedded systems consist of a collection of state machines at various levels of the electronics or software.*

This article provides an introduction to state machines fundamentals plus practical advice for implementing them in C or C++. I hope that these simple techniques can become more common, so that we can more often easily see the state-machine structure right from the source code.

### State Machines Reduce Spaghetti Code

As I pointed out in "*State Machines for Event-Driven Systems*", the main challenge in programming reactive (event-driven) systems is to correctly identify the appropriate piece of code to execute in response to a given event. In all but the most trivial reactive systems, the response depends both on the nature of the event and, more importantly, on the history of past events in which the system was involved.

From the programming perspective, this dependence on context very often leads to deeply nested if-else or switch-case constructs. Most reactive programs start out fairly simple and well structured, but as features are grafted on, more and more flags and variables are

introduced to capture the relevant event history. Then ifs and elses must be added to test the increasingly complex logical expressions built out of the various variables and flags (aka spaghetti), until no human being really has a good idea what part of the code gets executed in response to any given event.

And here is where state machines come in. When used correctly, state machines become powerful "spaghetti reducers" that drastically reduce the number of execution paths through the code, simplify the conditions tested at each branching point, and simplify transitions between different modes of execution. All these benefits hinge on the concept of "state." As it turns out, the behavior of most reactive systems can be divided into a relatively small number of non-overlapping chunks (states), where event responses within each individual chunk depend only on the current event, but no longer on the sequence of past events. In this model, change of behavior (that is, change in response to any event) corresponds to change of state (state transition). Thus, the concept of state becomes a succinct representation of the relevant system history.

In terms of coding, this means that instead of recording the event history in a multitude of variables, you can use just one "state variable" that can take only a limited number of a priori known values. By crisply defining the state of

the system at any given time, a state machine reduces the problem of identifying the execution context to testing just one state variable instead of many variables (recall the Visual Basic Calculator sample application I discussed in "State Machines for Event-Driven Systems"). Actually, in all but the most basic state-machine implementations (such as the nested switch statement), even the explicit testing of the state variable disappears from the code, which reduces spaghetti further still. In addition, switching between different execution contexts is vastly simplified as well, because you need to reassign just one state variable instead of changing multiple variables in a self-consistent manner.

### Memoryless Finite State Machines

Strictly speaking, the idealistic model just described corresponds to traditional finite state machines (FSMs) that don't have memory and must assume a new state for every change in behavior. For example, consider a simple time bomb, which will be our toy project for this episode (see Figure 1).



Figure 1. Time bomb user interface: (a) setting; (b) timing; and (c) blast.

The lifecycle of the bomb starts with users setting up the desired timeout from 1 to 10 seconds by pushing the UP ("+") and DOWN ("-") buttons. The setting ends when users push the ARM button, at which time the bomb becomes armed and starts ticking. Every TICK of the internal clock (occurring once per second)

decrements the timeout. The bomb blows up when the timeout reaches zero.

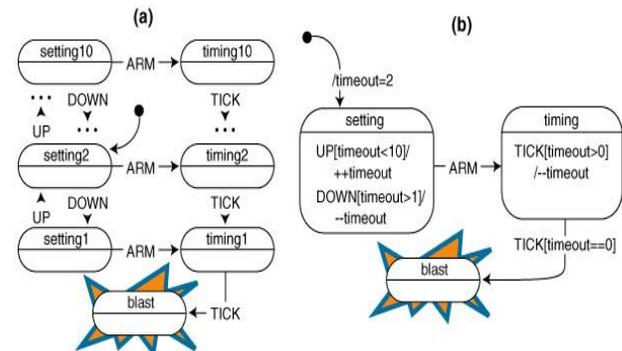


Figure 2. (a) Memoryless Time Bomb state machine; (b) equivalent extended state machine with extended state variable timeout.

Figure 2(a) shows a traditional (memoryless) FSM that implements the behavior of the time bomb. The diagram consists of 21 states: setting1 through setting10, timing1 through timing10, and the blast state. The alphabet of the state machine (all events that it recognizes) consists of four events: UP, DOWN, ARM, and TICK.

**Exercise:** Code the state diagram depicted in Figure 2(a) in C using the simple nested-switch-statement technique with a scalar state variable used as the discriminator in the first level of the switch, and the event-type in the second.

### Extended State Machines

Clearly, the state diagram from Figure 2(a) is hopelessly complex for a simple time bomb and I don't think that, in practice, anyone would implement the bomb that way (except, perhaps, if you have only a few bytes of RAM for variables but plenty of ROM for code). I suppose that most people would come up with a solution akin to Figure 2(b), which is obviously much more compact, but at the price of giving

the state machine memory in form of the timeout counter. Using the UML notation, Figure 2(b) shows how the timeout counter is initialized to 2 in the initial transition and then tested and modified in transitions UP, DOWN, and TICK.

The state diagram in Figure 2(b) is an example of an extended state machine, in which the complete condition of the system (called the "extended state") is the combination of a qualitative aspect—the state—and the quantitative aspects—the extended state variables (such as the timeout counter). In extended state machines, a change of a variable does not always imply a change of the qualitative aspects of the system behavior and, therefore, does not always lead to a change of state.

The obvious advantage of extended state machines is that they let you apply the underlying formalism to much more complex problems than is practical with the basic (memoryless) FSMs. For example, extending the timeout limit of the time bomb from 10 to 60 seconds would require adding 100 new states to the memoryless FSM, but would not complicate the extended state machine at all (the only modification required would be changing the test in transition UP). This increased flexibility of extended state machines comes with a price, however, because of the complex coupling between the qualitative and the quantitative aspects of the extended state. The coupling occurs through guard conditions (or simply guards), which are Boolean expressions evaluated dynamically based on the value of extended state variables. Guard conditions affect the behavior of a state machine by enabling actions or transitions only when they evaluate to true (and disabling them when they evaluate to false). In the UML notation, guards are shown in square brackets

immediately following the corresponding event (for example, `TICK[timeout == 0]`).

The need for guards is the immediate consequence of adding memory (extended state variables) to the state-machine formalism. Used sparingly, extended state variables and guards make up an incredibly powerful mechanism that can immensely simplify designs—just compare Figures 2(a) and (b). But don't let the fancy name ("guard") and the innocuous UML notation fool you. When you actually code an extended state machine, the guards become the same ifs and elses that you wanted to eliminate by using the state machine in the first place. Add too many of them, and you'll find yourself back in square one (spaghetti), where the guards effectively take over handling all the relevant conditions in the system.

Indeed, abuse of extended state variables and guards is the primary mechanism of architectural decay in designs based on state machines. Usually, in the day-to-day battle, it seems very tempting (especially to programmers new to state-machine formalism) to add yet another extended state variable and yet another guard condition (another if or an else) rather than to factor out the related behavior into a new qualitative aspect of the system—the state. From my experience, the likelihood of such architectural decay is directly proportional to the overhead (actual or perceived) involved in adding or removing states. (That's why I don't particularly like the popular state-table technique of implementing state machines, because adding a new state requires adding and initializing a whole new column in the table.)

One of the main challenges in becoming an effective state-machine designer is to develop a sense for which parts of the behavior should be

captured as the qualitative aspects (the state) and which elements are better left as the quantitative aspects (extended state variables). In general, you should actively look for opportunities to capture the event history (what happened) as the state of the system, instead of a simple record of events stored in extended state variables. For example, the Visual Basic calculator (see "*State Machines for Event-Driven Systems*") used an extended state variable `DecimalFlag` to record that the user entered the decimal point (to avoid entering multiple decimal points in a number). However, a better solution is to observe that entering a decimal point really leads to a distinct state "entering the fractional part of a number," in which the calculator ignores decimal points.

This solution is superior for a number of reasons. The lesser reason is that it eliminates one extended state variable and the need to initialize and test it. The more important reason is that the state-based solution is more robust because the context information is used very locally (only while entering the fractional part of a number) and is discarded as soon as it becomes irrelevant. Once the number is correctly entered, it doesn't really matter for the subsequent operation of the calculator whether that number had a decimal point. The state machine moves on to another state and automatically forgets the previous context. The `DecimalFlag` extended state variable, on the other hand, lays around well past the time the information becomes irrelevant (and perhaps outdated). Herein lies the danger, because you must not forget to reset `DecimalFlag` before entering another number, or the flag will incorrectly indicate that indeed the user once entered the decimal point, but perhaps this happened in the context of the previous number.

Capturing behavior as the qualitative state has its disadvantages and limitations, too. First, the state and transition topology in a state machine must be static and fixed at compile time, which can be too limiting and inflexible. Sure, you can easily devise state machines that would modify themselves at runtime (this is what often actually happens when you try to refactor spaghetti into a state machine). However, this is like writing self-modifying code, which was done in the early days of programming, but was quickly dismissed as a generally bad idea. Consequently, state can capture only static aspects of the behavior that are known a priori and are unlikely to change in the future. For example, it is fine to capture the entry of a decimal point in the calculator as a separate state "entering the fractional part of a number," because a number can have only one fractional part, which is both known a priori and is not likely to change in the future. However, as in Figure 2(a), capturing each time-unit processing in the time bomb as a separate state leads to rather elaborate and inflexible designs. This points to the main weakness of the qualitative state, which simply cannot store too much information (such as the wide range of timeouts). Extended state variables and guards are thus a mechanism for adding extra runtime flexibility to state machines.

## State Machines versus Flowcharts

Newcomers to state-machine formalism often confuse state machines with flowcharts. The UML specification isn't helping in this respect because it lumps activity graphs in the state-machine package. Activity graphs are essentially elaborate flowcharts.

The most important difference between state machines and flowcharts is that the state machines perform actions only in response to explicit events (they are entirely event driven).

In contrast, flowcharts do not need to be triggered by events; rather, they transition from node to node in their graph automatically upon completion of activities. Graphically, compared to state diagrams, flowcharts reverse the sense of vertices and arcs. In state diagrams, the processing is associated with the arcs (transitions); whereas in flowcharts, it is associated with the vertices.

You can compare a flowchart to an assembly line in manufacturing because the flowchart describes the progression of some task from beginning to end (for example, transforming source code input into machine code output by a compiler). A state machine generally has no notion of such a progression. A time bomb, for example, is not in a more advanced stage when it is in the timing state, compared to being in the setting state—it simply reacts differently to events. A state in a state machine is an efficient way of specifying a particular behavior, rather than a stage of processing.

The distinction between state machines and flowcharts is especially important because these two concepts represent two diametrically opposed programming paradigms: event-driven programming (state machines) and transformational programming (flowcharts). You cannot devise effective state machines without constantly thinking about the available events. In contrast, events are only a secondary concern (if at all) for flowcharts.

## How to Structure State Machine Code

To illustrate basic guidelines for structuring state-machine code, I'll walk you quickly through an implementation of the time bomb FSM from Figure 2(b). I'm going to use the pointer-to-member-function technique that I introduced in *"State Machines for Event-Driven Systems"*, and subsequently extended with entry and exit

actions in *"Introduction to Hierarchical State Machines (HSMs)"*.

```

1: class Bomb : public Fsm
2: {
3:     int myTimeout;           // extended state
                                variable
4: public:
5:     Bomb() : Fsm((State)&Bomb::initial) {}
                                // ctor
6:     void initial(Event const *e); // initial
                                pseudostate
7:     void setting(Event const *e); // state
                                handler
8:     void timing(Event const *e); // state
                                handler
9:     void blast(Event const *e);  // state
                                handler
10: };

```

Listing 1. Declaration of the Bomb FSM from Figure 2(b).

Listing 1 shows the first step of the implementation, in which you derive the Bomb FSM from the Fsm superclass described in *"State Machines for Event-Driven Systems"*. You declare extended state variables as data members of the derived class (Listing 1, line 3) and you map each state from the diagram in Figure 2(b) to a state-handler method (lines 7-9). The FSM from Figure 2(b) has three states, so you end up with three state-handler methods, each with the same signature declared in the Fsm superclass. Additionally, you also must declare the initial pseudostate handler (line 5). Finally, you define a default constructor using the initial pseudostate handler as the argument to construct the superclass Fsm (line 5).

The actual implementation of the state-handler methods based on the diagram shown in Figure 2(b) is straightforward and consists of applying just a few simple rules. Take, for instance, the

definition of the state-handler Bomb::setting()  
(Listing 2, lines 6-26).

```

1: void Bomb::initial(Event const *e)
2: {
3:   myTimeout = 2;
4:   initialTran((State)&Bomb::setting); //
initial transition
5: }
6: void Bomb::setting(Event const *e)
7: {
8:   switch (e->sig)
9:   {
10:    case UP_SIG:
11:     if (myTimeout < 10)
12:     {
13:      ++myTimeout;
14:     }
15:     break;
16:    case DOWN_SIG:
17:     if (myTimeout > 1)
18:     {
19:      --myTimeout;
20:     }
21:     break;
22:    case ARM_SIG:
23:     tran((State)&Bomb::timing);
24:     break;
25:   }
26: }
27: void Bomb::timing(Event const *e)
28: {
29:   switch (e->sig)
30:   {
31:    case ENTRY_SIG:
32:     SetTimer(locHwnd, 1, 1000, 0); //
start ticking every 1000 ms
33:     break;
34:    case TICK_SIG:
35:     if (myTimeout > 0)
36:     {
37:      --myTimeout;
38:     }
39:     else

```

```

40:     { // timeout expired
41:       tran((State)&Bomb::blast);
42:     }
43:     break;
44:    case EXIT_SIG:
45:     KillTimer(locHwnd, 1); // don't
leak the timer!
46:     break;
47:   }
48: }

```

Listing 2. Definition of the Bomb FSM from Figure 2(b).

First, you look up this state in the diagram and follow around its state boundary. You need to implement all transitions originating at this boundary, entry and exit actions (if present), as well as all internal transitions enlisted in this state. State setting has only one transition, ARM, that originates at its boundary, as well as two internal transitions, UP and DOWN, which both have guards. (Just a reminder: Internal transitions are different from self transitions because the latter cause execution of exit and entry actions, while the former never trigger state exit or entry.)

To code a state transition, you intercept the trigger (ARM\_SIG in this case, see Listing 2, line 22), enlist all actions associated with this transition (here there are none), then designate the target state as the argument of the tran() method inherited from the Fsm superclass (line 23).

You code the internal transitions in a similar way, except that you don't call the tran() method. If the transition has a guard (as, for example, the transition UP does), you first test the guard condition inside an if (...) statement (Listing 2, line 11), and you place the transition actions inside the true branch of the if (line 13).

Listing 2 demonstrates some more examples of coding other state-machine elements. One of the more interesting cases is the response to the TICK event in state timing, which, depending on the guard [`myTimeout > 0`], is either an internal transition or a regular transition to state blast (note that the event TICK appears twice in state timing with two complementary guards). Please note that this implementation supports entry and exit actions, which are used in the state timing to initialize and cleanup the timer that provides the TICK events.

**Exercise:** Download the time bomb Windows application that's included with the standard QP distributions here and execute it—it's really harmless; see Figure 1(c). Subsequently, add the "defusing" feature to the time bomb, which lets users defuse the armed bomb by entering a secret binary code. The code should be entered as a sequence of the UP ("`+`") and DOWN ("`-`") buttons ("`++--`", for example) followed by pressing the ARM button.

## Specifying State Machines in C and C++

Now, after you have seen how you could code a state machine in C++, let's look at Listing 2 again, but this time not so much as an implementation of a state machine, but as its specification. In fact, I challenge you to invent any other textual notation for state machines that would be more precise, expressive, and succinct than Listing 2 is.

Which brings me to the main point of this article: C++ (or C) is not just an implementation language; it can also be a powerful specification language for state machines. You should keep this in mind all the time while implementing state machines. This perspective helps you (and others) to readily see the state-machine

structure right from the code and easily map the code to state diagrams and vice versa.

The key is the way you break up the code. Instead of splitting the code ad hoc, you should partition it strictly into elements of state machines—that is, states, transitions, actions, guards, and choice points (structured if-else). You should also construct complete state-handler methods, by which I mean state-handler methods that directly include all state-machine elements pertaining to a given state, so that you could, at any time, unambiguously draw the state in a diagram. The following implementation of the `Bomb::timing()` state handler illustrates a problematic way of partitioning the code:

```
void Bomb::timing(Event const *e)
{
    switch (e->sig)
    {
        ...
        case TICK_SIG:
            onTick(e); // internal transition?
            break;
    }
}

void Bomb::onTick(Event const *e)
{
    if (myTimeout > 0)
    {
        --myTimeout;
    }
    else
    {
        // timeout expired
        tran((State)&Bomb::blast);
    }
}
```

Although correct in principle, this state handler is an incomplete specification of state timing, because the action `Bomb::onTick()` hides guard conditions and the transition to state blast.

Worse, the state handler `Bomb::timing()` is misleading because it suggests that the `TICK_SIG` signal triggers an internal transition, which it does not.

## End Around Check

Because my focus has been on the basics, I limited the discussion to simple nonhierarchical state machines. However, all arguments and guidelines apply equally well to hierarchical state machines (HSMs). Please refer to "*Introduction to Hierarchical State Machines (HSMs)*" for code examples pertaining to HSMs.

Most design automation tools internally represent state machines in textual format. One example of such a published notation is the "ROOM linear form" described by Selic, et al., in *Real-Time Object Oriented Modeling*.<sup>1</sup> Interestingly, except for the C/C++ switch and break statements, the ROOM notation is essentially identical to the state-handler methods just outlined.

I often wonder if a computer program can ever have enough structure. It seems that the more discipline you bring to bear on writing code, the more code you seem to get written. And the code works better in the bargain.

Here, I tried to convince you that state machines are more than just fancy diagrams—they are an excellent tool for better structuring event-driven code. Actually, the rules of mapping between state diagrams and code are

so simple that, with just a bit of practice, you will forget that you are laboriously translating a state diagram to code or vice versa. Rather, you will directly code states and transitions in C or C++, just as you directly code classes in C++ or Java. At this point, you'll experience a paradigm shift because you'll no longer struggle with convoluted if-else spaghetti and gazillions of flags. You'll find yourself working at a higher level of abstraction: directly with states, transitions, events, guards, and other state-machine elements. For embedded-systems developers, this paradigm shift can be even more important than the transition from procedural to object-oriented programming.

## Endnotes

1. Selic, Bran; Gullekson, Garth; and Ward, Paul T. *Real-Time Object Oriented Modeling*. John Wiley & Sons, 1994. [back]

*Related webinars and topics can be found at:*

<https://barrgroup.com/Embedded-Systems/How-To/State-Machines-Event-Driven-Systems>

<https://state-machine.com/products/>

<https://barrgroup.com/Embedded-Systems/How-To/Introduction-Hierarchical-State-Machines>