MARTIN GOMEZ

Embedded State Machine Implementation

Turning a state machine into a program can be pretty straightforward if you follow the advice of a skilled practitioner.

Many embedded software applications are natural candidates for mechanization as a state machine. A program that must sequence a series of actions, or handle inputs differently depending on what mode it's in, is often best implemented as a state machine.

This article describes a simple approach to implementing a state machine for an embedded system. Over the last 15 years, I have used this approach to design dozens of systems, including a softkey-based user interface, several communications protocols, a silicon-wafer transport mechanism, an unmanned air vehicle's lost-uplink handler, and an orbital mechanics simulator.

State machines

For purposes of this article, a *state machine* is defined as an algorithm that can be in one of a small number of states. A *state* is a condition that causes a prescribed relationship of inputs to outputs, and of inputs to next states. A savvy reader will quickly note that the state machines described in this article are Mealy machines. A Mealy machine is a state machine where the outputs are a function of both present state and input, as opposed to a Moore machine, in which the outputs are a function only of state.[1] In both cases, the next state is a function of both present state and input. Pressman

One of the advantages of a state machine is that it forces the programmer to think of all the cases and, therefore, to extract all the required information from the user.

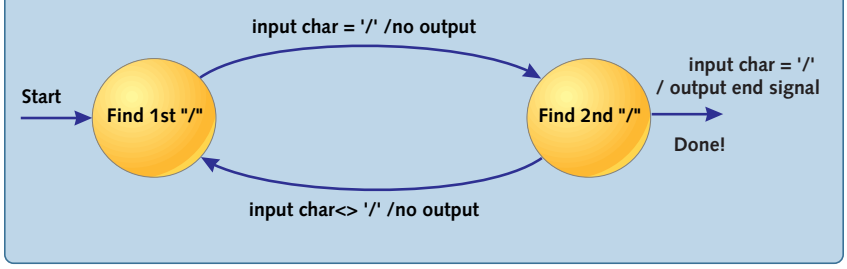
has several examples of state transition diagrams used to document the design of a software product.[2]

A simple example of a state machine is a protocol parser. Suppose you had to parse a string of text, looking for the token "//". Figure 1 shows a state machine to do that. In this example, the first occurrence of a slash produces no output, but causes the machine to advance to the second state. If it encounters a non-slash while in the second state, then it will go back to the first state, because the two slashes must be adjacent. If it finds a second slash, however, then it produces the "we're done" output.

The state machine approach I recommend proceeds as follows:

- Learn what the user wants
- Sketch the state transition diagram
- Code the skeleton of the state machine, without filling in the details of the transition actions
- Make sure the transitions work properly
- Flesh out the transition details
- Test

FIGURE 1 A simple state machine to parse a character string, looking for "//"



An example

A more illustrative example is a program that controls the retraction and extension of an airplane's landing gear. While in most airplanes this is done with an electrohydraulic control mechanism (simply because they don't have a computer on board), cases exist—such as unmanned air vehicles—where one would implement the control mechanism in software.

Let's describe the hardware in our example so that we can later define the software that controls it. The landing gear on this airplane consists of a nose gear, a left main gear, and a right main gear. These are hydraulically actuated. An electrically driven hydraulic pump supplies pressure to the hydraulic actuators. Our software can turn the pump on and off. A direction valve is set by the computer to either "up" or "down," to allow the hydraulic pressure to either raise or lower the landing gear. Each leg of the gear has two limit switches: one that closes if the gear is up, and another that closes when it's locked in the down position. To determine if the airplane is on the ground, a limit switch on the nose gear strut will close if the weight of the airplane is on the nose gear (commonly referred to as a "squat switch"). The pilot's controls consist of a landing gear up/down lever and three lights (one per leg) that can either be off, glow green (for down), or glow red (for in transit).

Let us now design the state machine. The first step, and the hardest, is to figure out what the user really wants the software to do. One of the advantages of a state machine is that it forces the programmer to think of all the cases and, therefore, to extract all the required information from the user. Why do I describe this as the hardest step? How many times have you been given a one-line problem description similar to this one: don't retract the gear if the airplane is on the ground.

Clearly, that's important, but the user thinks he's done. What about all the other cases? Is it okay to retract the gear *the instant* the airplane leaves the ground? What if it simply bounced a bit due to a bump in the runway? What if the pilot moved the gear lever into the "up" position while he was parked, and subequently takes off? Should the landing gear then come up?

One of the advantages of thinking in state machine terms is that you can quickly draw a state transition diagram on a whiteboard, in front of the user, and walk him through it. A common notation designates state transitions as follows: <event that caused the transition>/<output as a result of the transition>.[2] If we simply designed what the user initially asked us for ("don't retract the gear if the airplane is on the ground"), what he'd get would look a bit like Figure 2. It would exhibit the "bad" behavior mentioned previously.

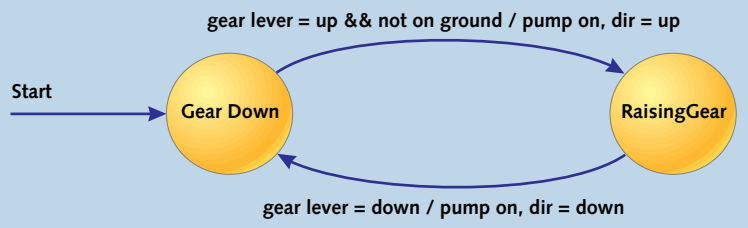
Keep the following in mind when designing the state transition diagram (or indeed any embedded algorithm):

- Computers are very fast compared to mechanical hardware—you may have to wait
- The mechanical engineer who's describing what he wants probably doesn't know as much about computers or algorithms as you do.

RUPERT ADLEY

How will your program behave if a mechanical or electrical part breaks? Provide for timeouts, sanity checks, and so on.

FIGURE 2 A state machine fragment that does only what the user requested



Good thing, too—otherwise you would be unnecessary!

- How will your program behave if a mechanical or electrical part breaks? Provide for timeouts, sanity checks, and so on

We can now suggest the following state machine to the user, building upon his requirements by adding a few states and transitions at a time. The result is shown in Figure 3. Here, we want to preclude gear retraction until the airplane is definitely airborne, by waiting a couple of seconds after the squat switch opens. We also want to respond to a rising edge of the pilot's lever, rather than a level, so that we rule out the "someone moved the lever while the airplane was parked" problem. Also, we take into account that the pilot might change his mind. Remember, the landing gear takes a few seconds to retract or extend, and we have to handle the case where the pilot reversed the lever during the process. Note, too, that if the airplane touches down again while we're in the "Waiting for takeoff" state, the timer restarts—the airplane has to be airborne for two seconds before we'll retract the gear.

Implementation

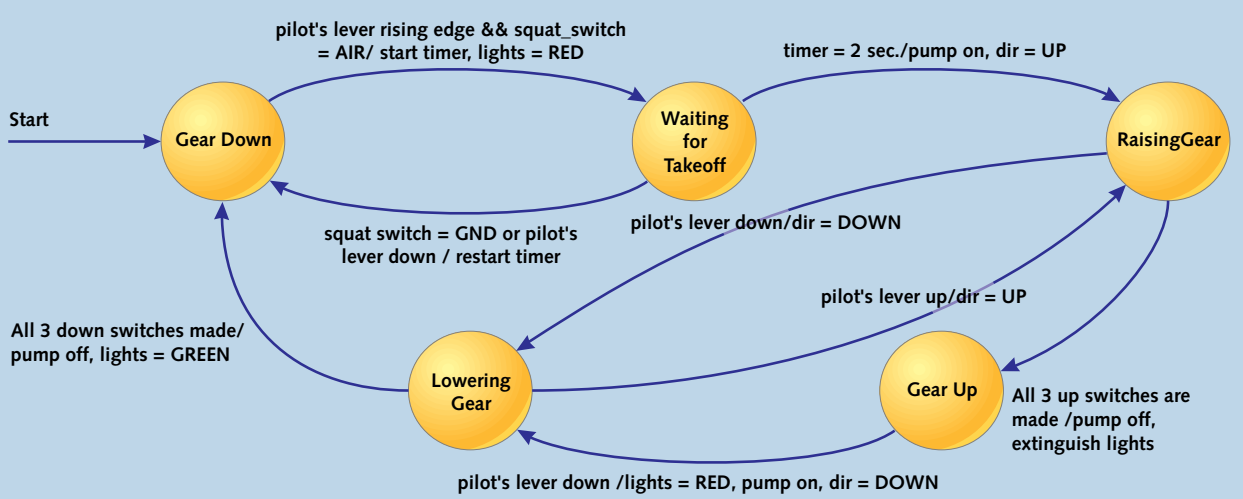
This is a good point to introduce a clean way to code a finite state machine. Listing 1 is my implementation of the state machine in Figure 3.

Let's discuss a few features of the example code. First, you'll notice that the functionality of each individual state is implemented by its own C function. You could just as easily implement it as a switch statement, with a separate `case` for each state. However, this can lead to a very long function (imagine 10 or 20 lines of code per state for each of 20 or 30 states.) It can also lead you astray when you change the code late in the testing phase—perhaps you've never forgotten a `break` statement at the end of a `case`, but I sure have. Having one state's code "fall into" the next state's code is usually a no-no.

To avoid the `switch` statement, you can use an array of pointers to the individual state functions. The index into the array is the `curr_state`, which is declared as an enumerated type to help our tools enforce correctness.

For simplicity, all the hardware I/O (reading switches, turning pumps on and off, and so on) is represented as a simple variable access. It's assumed that these variables are "magic addresses" that are associated with the hardware through invisible means. The next obvious thing is that the

FIGURE 3 The result of finding out what the user really wants



LISTING 1 Landing gear implementation

```
typedef enum {GEAR_DOWN = 0, WTG_FOR_TKOFF, RAISING_GEAR, GEAR_UP,
LOWERING_GEAR} State_Type;

/* This table contains a pointer to the function to call in each state.*/
void (*state_table[])() = {GearDown, WtgForTakeoff, RaisingGear, GearUp,
LoweringGear};

State_Type curr_state;

main()
{
   InitializeLdgGearSM();

   /* The heart of the state machine is this one loop. The function
      corresponding to the current state is called once per iteration. */
   while (1)
   {
      state_table[curr_state]();
      DecrementTimer();

      /* Do other functions, not related to this state machine.*/
   }
};

void InitializeLdgGearSM()
{
   curr_state = GEAR_DOWN;
   timer = 0.0;

   /* Stop all the hardware, turn off the lights, etc.*/
}

void GearDown()
{
   /* Raise the gear upon command, but not if the airplane is on the ground.*/
   if ((gear_lever == UP) && (prev_gear_lever == DOWN) && (squat_switch == UP))
   {
      timer = 2.0;
      curr_state = WTG_FOR_TKOFF;
   };

   prev_gear_lever = gear_lever; /* Store for edge detection.*/
}

void RaisingGear()
{
   /* Once all 3 legs are up, go to the GEAR_UP state.*/
   if ((nosegear_is_up == MADE) && (leftgear_is_up == MADE) && (rtgear_is_up
    == MADE))
   {
```

Listing 1 continued on p. 46

code doesn't do much at this point. It just transitions from state to state. This is an important intermediate step, and you shouldn't skip it lightly. It would be a good idea to add some `#ifdef DEBUG`'d print statements that cause each function to print "I'm in state xxx" as well as the value of the inputs.

Half the battle is in the code that induces the state transitions, that is, detecting that the input events have occurred. Once the code is cycling through states correctly, the next step is to fill in the "meat" of the code, namely that which produces the output. Remember that each transition has an input (the event that caused it), and an output (the hardware I/O, in our example). It is often helpful to write this down as a state transition table. Keller and Shumate use a format similar to the one in Table 1, to which I've added an explicit listing of all the inputs and outputs.[3] Here, we'll fill out the state transitions.

There is one line per state transition in the state transition table. In our relatively simple example, the "action" code is a very direct mapping from the state transition table. This, as you can imagine, is not always the case.

In coding a state machine, try to preserve its greatest strength, namely, the eloquently visible match between the user's requirements and the code. It may be necessary to hide hardware details in another layer of functions, for instance, to keep the state machine's code looking as much as possible like the state transition table and the state transition diagram. That symmetry helps prevent mistakes, and is the reason why state machines are such an important part of the embedded software engineer's arsenal. Sure, you could do the same thing by setting flags and having countless nested `if` statements, but it would be much harder to look at the code and compare it to what the user wants. The code fragment in Listing 2 fleshes out the `RaisingGear()` function.

LISTING 1, continued Landing gear implementation

```
        curr_state = GEAR_UP;
    };

    /* If the pilot changes his mind, start lowering the gear.*/
    if (gear_lever == DOWN)
    {
        curr_state = LOWERING_GEAR;
    };
}

void GearUp()
{
    /* If the pilot moves the lever to DOWN, lower the gear.*/
    if (gear_lever == DOWN)
    {
        curr_state = LOWERING_GEAR;
    };
}

void WtgForTakeoff()
{
    /* Once we ve been airborne for 2 sec., start raising the gear.*/
    if (timer <= 0.0)
    {
        curr_state = RAISING_GEAR;
    };

    /* If we touch down again, or if the pilot changes his mind, start over.*/
    if ((squat_switch == DOWN) || (gear_lever == DOWN))
    {
        timer = 2.0;
        curr_state = GEAR_DOWN;

         /* Don't want to require that he toggle the lever again
            this was just a bounce.*/
         prev_gear_lever = DOWN;
    };
}

void LoweringGear()
{
    if (gear_lever == UP)
    {
        curr_state = RAISING_GEAR;
    };

    if ((nosegear_is_down == MADE) && (leftgear_is_down == MADE) &&
        (rtgear_is_down == MADE))
    {
         curr_state = GEAR_DOWN;
    };
}
```

The beauty of coding even simple algorithms as state machines is that the test plan almost writes itself. All you have to do is to go through every state transition.

Notice that the code for `RaisingGear()` attempts to mirror the two rows in the state transition table for the Raising Gear state.

One guideline to keep in mind is "Avoid hidden states." A hidden state is what you get (or more honestly, what I got) when, out of laziness, you try to add a conditional substate rather than explicitly add a state. For instance, if your state function has code that says "Do *this* in one mode but do *that* in another mode," I'd start to wonder if that state shouldn't be split into two. This choice would depend on how much stuff was contained in the conditional. If your code is handling the same input in different ways (that is, triggering different state transitions) depending on a mode flag, then that's a hidden state. To implement it that way reduces the advantage that the state machine conveys.

As an exercise, you may want to expand the state machine we've described to add a timeout to the extension or retraction cycle, because our mechanical engineer doesn't want the hydraulic pump to run for more than 60 seconds. If the cycle times out, the pilot should be alerted by alternating green and red lights, and he should be able to cycle the lever to try again. Another feature to exercise your skills would be to ask our hypothetical mechanical engineer "Does the pump suffer from having the direction reversed while it's running? We do it in the two cases where the pilot changes his mind." He'll say "yes," of course. How would you modify the state machine to stop the pump briefly when the direction is forced to reverse?

LISTING 2 The RaisingGear() function

```
void RaisingGear()
{
   /* Once all 3 legs are up, go to the GEAR_UP state.*/
   if ((nosegear_is_up == MADE) && (leftgear_is_up == MADE) && (rtgear_is_up
     == MADE))
   {
      pump_motor = OFF;
      gear_lights = EXTINGUISH;
      curr_state = GEAR_UP;
   };

   /* If the pilot changes his mind, start lowering the gear.*/
   if (gear_lever == DOWN)
   {
      pump_direction = DOWN;
      curr_state = GEAR_LOWERING;
   };
}
```

Testing

The beauty of coding even simple algorithms as state machines is that the test plan almost writes itself. All you have to do is to go through every state transition. I usually do it with a highlighter in hand, crossing off the arrows on the state transition diagram as they successfully pass their tests. This is a good reason to avoid "hidden states"—they're more likely to escape testing than explicit states. Until you can use the "real" hardware to induce state changes, either do it with a

TABLE 1 State transition table

Current State	Inputs					Outputs				
	3 are up	3 are down	Squat switch	Timer	Pilot's lever	Pump on/off	Dir	Timer	Lights	Next State
Gear down	X	X	Air	X	Rising edge	NC	NC	Start	RED	Waiting for takeoff
Waiting for takeoff	X	X	X	2 sec	X	ON	UP	stop	NC	Raising gear
Waiting for takeoff	X	X	GND	X	Down	NC	X	Restart	NC	Gear down
Raising gear	Made	X	X	X	X	OFF	NC	NC	OFF	Gear up
Raising gear	X	X	X	X	Down	NC	DN	NC	NC	Lowering gear
Gear up	X	X	X	X	Down	ON	DN	X	RED	Lowering gear
Lowering gear	X	X	X	X	Up	NC	UP	NC	NC	Raising gear
Lowering gear	X	Made	X	X	X	OFF	NC	NC	GREEN	Gear down

NC = No change in output X = Don't care input

Once the user's requirements are fleshed out, I can crank out a state machine of this complexity in a couple of days. They almost always do what I want them to do.

source-level debugger, or build an "input poker" utility that lets you write the values of the inputs into your application.

This requires a fair amount of patience and coffee, because even a mid-size state machine can have 100 different transitions. However, the number of transitions is an excellent measure of the system's complexity. The complexity is driven by the user's requirements: the state machine makes it blindingly obvious how much you have to test. With a less-organized approach, the amount of testing required might be equally large—you just won't know it.

It is very handy to include print statements that output the current state, the value of the inputs, and the value of the outputs each time through the loop. This lets you easily observe what ought to be the Golden Rule of Software Testing: don't just check that it does what you want—also check that it doesn't do what you don't want. In other words, are you getting only the outputs that you expect? It's easy to verify that you get the outputs that you expected, but what else is happening? Are there "glitch" state transitions, that is, states that are passed through inadvertently, for only one cycle of the loop? Are any outputs changing when you didn't expect them to? Ideally, the output of your `printf`s would look a lot like the state transition table.

Finally, and this applies to all embedded software and not just to that based on state machines, be suspicious when you connect your software to the actual hardware for the first time. It's very easy to get the polarity wrong—"Oh, I thought a '1' meant raise the gear and a '0' meant lower the gear." On many occasions, my hardware counterpart inserted a temporary "chicken switch" to protect his precious components until he was sure my software wasn't going to move things the wrong way.

Crank it

Once the user's requirements are fleshed out, I can crank out a state machine of this complexity in a couple of days. They almost always do what I want them to do. The hard part, of course, is making sure that I understand what the user wants, and ensuring that the user knows what he wants—that takes considerably longer! **esp**

Martin Gomez is a software engineer at the Johns Hopkins University Applied Physics Lab, where he is presently developing flight software for a solar research spacecraft. He has been working in the field of embedded software development for 17 years. Martin has a BS in aerospace engineering and an MS in electrical engineering, both from Cornell University. He may be reached at martin.gomez@jhuapl.edu.

References:

1. Hurst, S.L. *The Logical Processing of Digital Signals*. New York: Crane, Russak, 1978.
2. Pressman, Roger A. *Software Engineering: A Practitioner's Approach, 3rd Edition*. New York: McGraw-Hill, 1992.
3. Shumate, Kenneth C. and Marilyn M. Keller. *Software Specification and Design: A Disciplined Approach for Real-Time Systems*. New York: John Wiley & Sons, 1992.