A recent EDN article dealt with a number of programming tips for embedded microcontrollers. One of those tips, using state machines, is a programming approach that is especially useful, yet probably seldom gets used by most programmers. State machines create a structured, easily maintained approach to programming. As we’ll see, state machines even lend considerable structure to assembly language programs.

What Is A State Machine?
A state machine (more formally, a finite state machine), as used in this article, is a mathematical structure in which the next state is determined only by the previous state and by the inputs to the machine. Although we’ll be talking about state machines as they apply to computer programs, they are also useful in designing computer logic and a logic diagram reinforces the definition (Figure 1).

Such a machine can have up to $2^N$ states and is synchronous, since the states only change on the clock edges.

**Example - A Robot That Turns to Avoid Obstacles**
The state diagram provides the framework needed to think through the state machine design. This is usually shown as a collection of circles, with each circle representing a state, and with arrows representing the transitions. Symbols above these arrows, representing the input variable values making the transition possible, are also shown. A simple example will illustrate.

Let’s say that we have a robot like the one in the block diagram of Figure 2. It has sensors that can tell whether there’s an obstacle ahead and whether the obstacle is on the left or the right. The
robot’s only available responses are to turn left or right or to move straight ahead. The robot possesses no capability to stop, slow down, or reverse.

![Figure 2. Block Diagram of the Robot’s Electronics](image)

The robot’s ability to sense which side the obstacle is on is important because its diversionary maneuver (turning to one side or the other) depends on this additional information (Figure 3).

![Figure 3. Robot Approaching Obstacle](image)

Let’s identify the states. The robot can move straight ahead, to the left, or to the right. Those are the states of the machine. In some state machines there is also an initial state and a termination state, but for this example, we assume the robot is initialized to the “straight ahead” state on power-up and continues through the three states until it is manually powered down. We arbitrarily assign the state names A, B, and C (Table 1).

<table>
<thead>
<tr>
<th>Table 1. State Assignment and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
</tbody>
</table>
Now we’ll begin drawing the state diagram. This consists of the three circles corresponding to states A, B, and C. After drawing these, we write in the transitions that result from the inputs. Let’s assign names and values to the inputs. $x_o$ is 0 if there is no obstacle ahead and 1 otherwise. $x_s$ is 0 if the obstacle is on the left and 1 if the obstacle is on the right.

An obstacle on the left should cause the robot to turn right to avoid the obstacle. Similarly, an obstacle on the right should cause the robot to turn left. And we will always want to steer straight ahead if there is no obstacle. So we can immediately draw the transition arrows based on this information (Figure 4).

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Move straight ahead</td>
</tr>
<tr>
<td>B</td>
<td>Turn left</td>
</tr>
<tr>
<td>C</td>
<td>Turn right</td>
</tr>
</tbody>
</table>

*State diagram*

Now we’ll begin drawing the state diagram. This consists of the three circles corresponding to states A, B, and C. After drawing these, we write in the transitions that result from the inputs. Let’s assign names and values to the inputs. $x_o$ is 0 if there is no obstacle ahead and 1 otherwise. $x_s$ is 0 if the obstacle is on the left and 1 if the obstacle is on the right.

An obstacle on the left should cause the robot to turn right to avoid the obstacle. Similarly, an obstacle on the right should cause the robot to turn left. And we will always want to steer straight ahead if there is no obstacle. So we can immediately draw the transition arrows based on this information (Figure 4).

*Figure 4. The State Diagram for the Simple Robot*

Note that each state has an arrow emanating from it for each input-initiated transition. Since there are four input combinations (two binary input variables), there should be four arrows leaving each
state bubble. However, in Figure 4, some arrows represent two possible combinations, since a “don’t care” condition, represented by a “d” in the diagram, counts twice. For example, the arrow labeled “\(x_0 = 0, x_s = d\)” at State A represents both “\(x_0 = 0, x_s = 0\)” and “\(x_0 = 0, x_s = 1\)”.

Note also that this particular state machine has the peculiar property that the next state depends only on the two input variables, not on the previous state. Thus, any time \(x_0 = 0\), the state machine transitions to State A, any time \(x_0 = 1\) and \(x_s = 1\) the next state is State B, and any time \(x_0 = 1\) and \(x_s = 0\), the next state is State C.

In such a case, we just need a combinatorial logic circuit. Such a situation does not require any machine memory. But the machine needs to be a little more sophisticated than formulated so far, and the state machine approach will indeed be needed.

**The Indecisive Robot**

What happens when the robot’s been turning left to avoid an obstacle on the right and suddenly its sensors tell it there’s an obstacle on the left? Can such a sudden change happen? And what should be the robot’s response in that instance?

Actually, this is a common phenomenon with such sensor systems. Think about what happens when the robot is approaching an obstacle head-on, as in Figure 5. The left and right obstacle sensors generate signals of almost the same analog amplitude in that case and, if the system is set up to do a compare of these two amplitudes and send a simple obstacle-on-left/obstacle-on-right digital signal, as in our example, this binary signal from the comparator will change frequently.

![Figure 5. The Robot Approaching Obstacle Head-On](image)

**Unstable situation**

The result, if we stick with the Figure 4 machine, can be an unstable situation, where the robot switches erratically between steer-left/steer-right conditions, possibly crashing into the obstacle as the result of its seeming indecision. This is particularly a problem when the robot is fast-moving and the robot’s control loop update rate is marginal.

Recognizing that this undesirable condition is the result of a head-on condition, one way to avoid the problem is to modify the state machine so that the machine can’t switch directly from State B to State C, or vice versa. If the robot starts a left-turn, it must continue turning left until it no longer sees an obstacle, which will mean that it’s then in State A. Likewise with a right turn. The revised
state diagram is shown in Figure 6.

This revised version of the state machine eliminates the direct transitions from left turn to right turn and from right turn to left turn. Instead, the transitions from States B and C are always back to those respective states as long as $x_0 = 1$. So, once the robot begins making a turn in a particular direction, it continues in that direction until it no longer sees an obstacle ($x_0 = 0$), at which point the state machine goes to State A.

**State Assignments**

So far we have only given the states labels. We will eventually need to associate a number with each state. We could arbitrarily number the states, for example, State A could be represented by the binary 00, State B by 01, and State C by 10.

Sometimes it’s advantageous to use the output values of the state machine to represent the states. Let’s say that the steering motor driver expects the following commands:

- A 01 to steer the front wheels right
- A 10 to steer the front wheels left
- A 00 to maintain the front wheels in a straight-ahead position

In that case, we might wish to make the state assignments based on those desired outputs, as in Figure 7.
Programming the State Machine

Programming this state machine is straightforward. Let’s assume that the two state bits, \( S_1 \) and \( S_0 \), and the two input bits, \( x_O \) and \( x_S \), have been packed into a control byte (Figure 8):

![Figure 8. Control Byte Containing State and Input Bits](image)

We’ll call this byte “Control”. Most C compilers allow the Switch / Case type of construct. The Switch statement would choose a Case destination based on the value of the Control byte. Up to 16 cases could be accommodated (corresponding to the two state bits and the two input bits) although only 12 cases would be needed for this example, since the \( S_1 = S_0 = 1 \) state is not used.

Assembly language program

However, it’s also possible to use this state idea in assembly language. In fact there’s a very simple way of doing it in assembly language and that is through the use of jump tables. And the state machine approach provides considerable structure to the assembly language program.

This jump table approach is similar for all assembly languages. To demonstrate its use, a TI MSP430 instruction set will be used. In the MSP430 instruction set, the jump instruction (JMP) is two bytes long. The program of Figure 9 assumes that a subroutine called SensorInputs has already determined whether there is an obstacle in the robot’s path and on which side of the robot the obstacle exists. This information is updated at each iteration of the control loop.
The subroutine Pack takes the \( x_0 \) and \( x_s \) inputs from the SensorInputs subroutine, along with the current state of the state machine, and packs these together in the format of Figure 8.
Case6
; In State B with \( x_O = 0 \) and \( x_S = 1 \) => Move to State A
clr &NextState
jmp ReturnFromStateMachine

Case7
; This is an illegal case, since the Current State = 3
jmp Trap

Case8
; In State A with \( x_O = 1 \) and \( x_S = 0 \) => Move to State C
mov #1,&NextState
jmp ReturnFromStateMachine

Case9
; In State C with \( x_O = 1 \) and \( x_S = 0 \) => Stay in State C
jmp ReturnFromStateMachine

Case10
; In State B with \( x_O = 1 \) and \( x_S = 0 \) => Stay in State B
jmp ReturnFromStateMachine

Case11
; This is an illegal case, since the Current State = 3
jmp Trap

Case12
; In State A with \( x_O = 1 \) and \( x_S = 1 \) => Move to State B
mov #2,&NextState
jmp ReturnFromStateMachine

Case13
; In State C with \( x_O = 1 \) and \( x_S = 1 \) => Stay in State C
jmp ReturnFromStateMachine

Case14
; In State B with \( x_O = 1 \) and \( x_S = 1 \) => Stay in State B
jmp ReturnFromStateMachine

Case15
; This is an illegal case, since the Current State = 3
jmp Trap

ReturnFromStateMachine
bic.b #3,&P1OUT
xor.b &CurrentState,&P1OUT
mov &NextState,&CurrentState
jmp Top

Trap
jmp Trap

Pack
; Pack creates a single packed byte, called Control, from the inputs
; \( x_O \) and \( x_S \) and the state \( S_1,S_0 \):
; \( \text{Control} = 8^*x_O + 4^*x_S + 2^*S_1 + S_0 \)
mov &xO,&Control
ral &Control
add &xS,&Control
ral &Control
add &CurrentState,&Control
Figure 9. The MSP430 State Machine Program

Note that the output values depend on the current state and change when the state assignment changes. Note also that, while there are 16 states in the Figure program, only 12 of them are valid states. The remaining four (corresponding to the state $S_1S_0=11$) are directed to an infinite loop called Trap in this program.

One other thing to note is that the Control variable is always twice the state assignment, due to the fact that each JMP instruction in the jump table consists of two bytes, not one.

Is the State Machine Approach Better?
If you think about what would have been required to program the robot in a traditional sequence of IF-THEN sequences (for C programs) or bit-test/conditional-jump sequences in assembly language, the benefit of the state machine approach may seem underwhelming. After all, the state machine program approach is probably longer than any such conditional test approach.

But once you use the state machine approach a few times and get used to its form, you’ll probably agree that it has some real advantages. Those advantages generally include:

1. Structure – the program is organized in a way that makes it easy to read and understand, unlike a traditional IF-THEN spaghetti code sequence.
2. Maintenance – the program’s response to a set of machine inputs is immediately understandable, so that it can more easily be modified in future releases.
3. Troubleshooting – testing for execution of a particular set of inputs/outputs can be determined by setting breakpoints for particular program sequences corresponding to that state.

The fact that the state machine approach often leads to longer programs is an attribute of many programming techniques that produce structure. And while the program is longer, it is nevertheless fairly efficient in execution time – only one of the 16 short sequences is actually executed each time through the loop.

Conclusion
The state machine approach to programming control loops takes a little getting used to, but is a big help in structuring such programs. The example given was fairly simple, so the state machine’s benefits were not as obvious as they would have been in more complicated situations. I think you’ll find, for typical real-world applications, that it will become an essential part of your embedded-controller programming toolkit.
Dan Harres retired from Boeing Company in 2011, where he was a Technical Fellow in the area of fiber optics and microcontrollers. He has since started an educational robotics company called Bitstream Technology, which is developing a line of products to teach young people about electronics. More about Dan Harres.

If you liked this and would like to see a weekly collection of related products and features delivered directly to your inbox, click here to sign up for the EDN on Systems Design newsletter.