Program Structures
for Embedded Systems

In your introductory programming course, you learned the basic program structures, assignments, loops, tests, and so on, associated with the language used for the course. If you go back and look at those programs, you will likely see that your program was always in control of when the input occurred. Your code looked for input at specific places in your program and waited for it to appear. While your program was looking for a particular kind of input, be it a mouse click or a key press, it wasn’t doing any other work. And, when it was working on a computation, it was unable to detect inputs. If, at a particular place in your program, you were looking for a mouse click and a key was pressed, generally, the key press was ignored (at least at that instant). While this approach to programming is functional in many situations, it doesn’t work very well for most mechatronics systems, where there will be multiple possible sources of input which may become active at times you cannot predict and where it is unacceptable to ignore any of the inputs. This chapter introduces two complementary programming structures that are extremely useful in designing and implementing software for mechatronic systems. By the time you have mastered these topics, you should be able to:

1. identify the events that will be relevant to a given system,
2. draw a state diagram to graphically capture the behavior of a system,
3. understand how to deal with noise on signals that indicate the occurrence of events,
4. write functions to implement event checking routines, service routines, and state machines.

5.1 BACKGROUND

As an example of a mechatronic system, consider the case of a DVD player. On a typical DVD player, there are a number of buttons on the front of the player, sensor inputs for disc present, and for disc tray closed, as well as commands that can be received from the remote control. The program for the embedded computer controlling the DVD player must be able to respond to any of those inputs at any time and in any sequence. For example, while running fast forward, you can stop, play, pause, or any number of other actions can be carried out. The program for the DVD player can’t predict if the next command from the remote control will come before reaching the end of the disc, for example, so a program structure that can only manage one input at a time would fail in this application.

What we need is a structure that will allow the program to respond to a multitude of inputs, any of which may require attention at unpredictable times and in an unpredictable sequence. While this problem is very common in embedded systems, like the DVD player, it is also very common in modern desktop computers. Think about using a word processor.
While you are typing, the program is accepting your keys and entering the characters into your document. You can also click with the mouse at any place on the screen. If the mouse pointer is at another place in your document, the point of insertion moves. If it is in the menu bar, a menu will drop down. If it is over a formatting button, it may change the format of the next character entered. Your word processor, and most other programs running under Windows, MacOS, X-Windows, or any other modern graphical user interface (GUI), must also be capable of dealing with the same kind of variable input that the DVD player must handle. All of these operating systems, and many embedded applications, have adopted a common program structure to deal with the need to respond to many asynchronous inputs. This program structure is generally called event driven programming.

5.2 EVENT DRIVEN PROGRAMMING

Under the event driven programming model, the program structure is divided into two groups: events and services. An event represents the occurrence of something interesting in your application. A service is the set of actions that are performed in response to an event. While events most often originate from outside your program, such as a mouse click, events can also be generated by other parts of the program, such as the expiration of a timer. The program executes by constantly checking for possible events, and when an event is detected, it executes the associated service.

In order for this approach to work, the event checking routines must run continuously and often. This implies that the services must execute quickly so that the program can get back to checking for other events. To meet this requirement, a service cannot go into a state where it is waiting for some long or indeterminate time. The most common example of this would be a WHILE loop, where the condition for termination was not under program control, such as a switch closure. This kind of program structure, an indefinite loop, is referred to as blocking code. In order for the event driven programming model to work, you must only write nonblocking code.

So, how do you deal with the need to wait for the switch to close? You make the switch closure an event. Then, rather than waiting in a loop for the switch to close, you program a service that performs a desirable action when the switch closes. Until the switch closure event occurs, however, your program continually scans for all the other possible events. In this way, you could also react to another event, if necessary, while you were waiting for the switch to close.

5.3 EVENT CHECKERS

An event, by its nature, occurs at a discrete point in time, instantaneously. At one instant, the event has not yet occurred, at the next, it has. In the case of a switch, the opening or closing of the switch represents an event. The switch being open or closed does not represent an event. The event is marked by the transition from open to closed (Figure 5.1).

![Figure 5.1](image)

An event represents a transition in a variable.
The event checkers, then, are small pieces of code that test for the occurrence of an event. In order to test for an event when the input is something like a switch, whose value is continuously available, the event checker must have some record of the past. An event has only occurred when the current value is different from the past value. In pseudo-code, the test would look something like the following.

IF switch is closed now AND switch was open last time

THEN SwitchClosed event has occurred.

To complete the event checker, the code must update the value kept as the last state of the switch. The variable used for the last state of the switch must be able to retain its value between successive calls to the event checker. In C, this might look like Listing 5.1.

```c
unsigned char CheckSwitchClosed( void)
{
    static unsigned char LastState = OPEN;
    unsigned char CurrentState;
    unsigned char EventStatus;

    CurrentState = GetSwitchState();
    if((CurrentState != LastState) &&
       (CurrentState == CLOSED)) /* the event test */
       EventStatus = TRUE;
    else
       EventStatus = FALSE;

    LastState = CurrentState; /* update state variable */

    return(EventStatus);
}
```

**LISTING 5.1**
An event checking routine in C.

In examining this function, notice that the LastState variable is local because it is only needed within this function. It is also declared with the static modifier so that it will retain its value between function calls. This is not required for the other two variables, which are also only needed within the function, but do not need to retain their values between function calls. As a reminder: when a static variable has an initializer, the initialization only takes place once (before your program starts running), not every time the function is entered (as is the case for nonstatic variable initializations).

The use of the CurrentState variable also demonstrates an important concept. By using the CurrentState local variable, we read the switch state in only one place. We will need the state of the switch in two places. We need the switch state first to test for an event and, then, later to update the LastState variable. By reading the switch state only once and assigning the result to a local variable, we avoid the possibility that the switch state might change between the time that we test it to check for an event and the time that we update the LastState variable. If this were to happen, we would miss an event since the test would fail, but the LastState variable would be updated with a new value.

This basic function outline can be used for most event checkers that test for events that can be represented as the transition between two discrete states. Events can also be triggered by variables that can take on a range of values. In that case, the event would be defined
as the transition across some threshold value. An example of this might be a light sensor that would generate an event when the light level rose above some threshold value. If you apply this same basic function outline to the light sensor [by substituting a test of above/below a threshold for the call to GetSwitchState()], you will probably find that you get a string of ON-OFF-ON-OFF events as the light level rises from below the threshold to above the threshold. To understand why that might be, consider the plot shown in Figure 5.2.

Any measured variable that represents the state of some real world (analog) signal will have a certain amount of noise associated with it. As the measured value of the variable approaches the threshold, the noise will alternate the value over the threshold, then below the threshold. This will continue until the combination of the signal plus noise stays completely above the threshold. While the combination of signal plus noise is dancing back and forth across the threshold, a sequence of spurious events is being generated. While this might be desirable in some cases, most of the time we would like to be able to respond to the signal while ignoring the noise. Ideally, we’d like such a transition, from low to high, to generate only a single event.

One approach to this problem would be to go back to the source of the signal and reduce the noise (Chapters 14 and 15). This would most likely be done with filtering to reduce the noise level to an undetectable level. This solution requires additional hardware and has the potential to adversely affect the time response of the desired signal. There is another approach that allows us to add a small amount of additional software complexity to eliminate the effect of the noise with no additional hardware.

We can use hysteresis (a different response to rising and falling signals) to overcome the chatter. If we modified our test to implement the correct amount of hysteresis at the switch point, we could eliminate the multiple events associated with the noise. If the hysteresis band was slightly larger than the amplitude of the noise, we would get a single clean transition as the signal passes through the hysteresis band from the low threshold to the high threshold (Figure 5.3).

The way to implement hysteresis in software is to change the threshold from a single fixed value to a variable that can take on one of two different values. For a signal that starts below the desired trip-point, the value is initially set at the higher of the two threshold values. As the signal plus noise rises, it will eventually exceed this threshold, generating an event. The service response to this detected event is to lower the threshold to the lower value and then do whatever else this event requires. Changing the threshold to a lower value, at this point in time, will prevent the noise superimposed on the signal from immediately triggering another event. To generate another event, the signal would need to fall by the width of the hysteresis band to get below the new threshold. Since we set the spacing between the two thresholds to be larger than the amplitude of the noise, this noise will not, by itself, generate additional
spurious events. When the falling signal plus noise does get below the new (lower) threshold, an event signifying that the light has gone off will be generated, and the threshold will then be raised to its higher value. The hysteresis behavior can be encapsulated into a GetLightState function that will implement the hysteresis and create a two-valued system like the switch. This will allow us to use the GetLightState function in the same basic event checking template that we used for the switch. In C, this might look like Listing 5.2.

```c
unsigned char GetLightState( void)
{
    static unsigned char Threshold = HI_THRESHOLD;
    unsigned char LightState;

    if (LightValue >= Threshold) /* above the threshold? */
    {
        LightState = LIGHT_ON;
        Threshold = LO_THRESHOLD;
    } else /* must be below */
    {
        LightState = LIGHT_OFF;
        Threshold = HI_THRESHOLD;
    }
    return(LightState);
}
```

LISTING 5.2
Implementing hysteresis in software, an example in C.

Here, again, we use a static local variable to maintain internal information that needs to be preserved between successive calls to the function. In this case, it is the threshold value. Whenever the current light value is above the threshold, the light is considered on and the threshold is forced to its lower value. Changing the threshold to the lower value allows the light level to drop somewhat (due to noise) and still be considered on. Only when the light level drops below the new lower threshold will the light be declared off and the threshold raised back to its higher value. Similarly, if the light level then rises slightly (due to noise), it will still be below the new higher value and the state will remain off. It is worthwhile noting again that the LightValue is read only once per call to the function. This prevents any confusion that might be caused by a rapidly changing light value that was sampled twice within a single function call.
The solution of adding hysteresis to eliminate chatter is not limited to software implementations. In Chapter 11, we will introduce an electronic functional block that can implement hysteresis in hardware.

5.4 SERVICES

Now that we can recognize events, what are services? Services are simply the actions that you want your program to perform when an event has been detected. As an example, think about a mobile robot with bump sensors. The event detector would check for bumping into something, and the service would execute the response. In this case, the response would most likely be to stop or reverse the drive motors and begin to move in another direction.

Services should be very compact functions that initiate the required action and quickly return. This allows the program to get back to checking for other events. The core assumption in event driven programming is that you can check for events quickly enough so that none are missed. This can only happen if both the event checkers and the service routines execute quickly. Neither event checkers nor service routines should enter into an indefinite loop (i.e., blocking code).

If you find yourself wanting to code a WHILE loop for something other than stepping through an array, you probably need to add an additional event. The way to handle a situation like that is to have a service routine that starts the activity going and an event checker to detect the end condition. Using the mobile robot as an example, you might want to back up for a second after a bump in order to move away from the obstacle. You might, improperly, code that as the response to the bump event (Listing 5.3).

```c
void BumpResponse( void)
{
    unsigned int StartTime = GetTime();
    DriveMotors(REVERSE);
    while(((GetTime() - StartTime) < BACKUP_TIME))
    {
        driveMotors(STOP); /* wait for backup time to expire */
    }
}
```

LISTING 5.3
A poor example: a response routine that implements blocking code.

If you did this, you would not be able to respond to any other inputs during the time that you were waiting for the backup timer to expire. This would mean that if you hit something while backing up, you would ignore it. A much better approach to this problem would be to write a response routine that started the motors in reverse and also started a timer to run for the backup time. Then you would need to write a new event checker to check for when the backup timer had expired (Listing 5.4).

The response routine for BackupTimeoutCheck() would take care of stopping the motors and initiating any other evasive maneuvers. This approach preserves the ability of the software to perform other checks while it waits for the backup time to expire.

5.5 BUILDING AN EVENT DRIVEN PROGRAM

The main body of an event driven program will consist of calls to the initialization routines followed by an endless loop to check for events and execute the service routines. While there are software frameworks available to simplify the event checking and calling of the services, they are simply optimizations of a structure that looks something like Listing 5.5.
void main(void)
{
    DoInitializations();
    while(1)
    {
        if(EventChecker1())
            ServiceRoutine1();
        if(EventChecker2())
            ServiceRoutine2();
    }
}

Listing 5.5
The main() function for a program using an event driven framework.

The bulk of your programming effort will be spent designing and coding the event checkers and service routines. The emphasis on short event checking and service routines has the added benefit of making the design and implementation of each of these routines simpler and less prone to errors. This implies that, with reasonable care, the major problems to be debugged will be at the design level, (i.e., choosing what the events will be and what should be done in response to those events) not at the code, or implementation, level.

By emphasizing the design phase as a place for debugging, we put ourselves in the enviable position of not needing either hardware or software to get started debugging! Usually, the debugging phase is the longest and most difficult, so getting such an early start will likely get us to a working program sooner than if we had tackled the problem without using event driven programming.
5.6 AN EXAMPLE

You are to design the software for a simple machine that, on a small scale, mimics the behavior of a cockroach. At the simplest level, this means that the cockroach runs straight ahead while in the light and stops when in the dark. If the cockroach encounters an object, it turns left, reverses direction, and runs backwards for 3 seconds. If, at the end of that time, it is not in the dark, it should begin running straight ahead again.

Identifying the basic events and services is pretty straightforward.

<table>
<thead>
<tr>
<th>Event</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Light Goes On</td>
<td></td>
</tr>
<tr>
<td>Drive Forward</td>
<td></td>
</tr>
<tr>
<td>Light Goes Off</td>
<td></td>
</tr>
<tr>
<td>Stop</td>
<td></td>
</tr>
<tr>
<td>Contact Object</td>
<td></td>
</tr>
<tr>
<td>Turn left, reverse for 3 sec.</td>
<td>Service</td>
</tr>
</tbody>
</table>

Notice the wording of the event descriptions. An event represents a transition (an instantaneous change), not a state (a continuing condition), and each of these event descriptions is written to capture the transition. The most common mistake made by programmers when beginning to use this methodology is to write event checking routines that actually detect the state of a sensor or input. The result is a continuous stream of “events” being detected and the corresponding actions being executed. In the example above, “Light Goes On” is an event that happens when the light transitions from off to on, not a state of the light being on. You need to be sure that your event checkers are testing for a transition in the variable so that they are checking for events.

Another interesting aspect is the service associated with contacting an object. How do you handle the “reverse for 3 sec.” service? As in the earlier example, we shouldn’t just sit in the service routine for 3 seconds. So we need to introduce a new, internal, event: Reverse Timer Expires. Adding this new event and a little more thought yields:

<table>
<thead>
<tr>
<th>Event</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Light Goes On</td>
<td></td>
</tr>
<tr>
<td>Drive Forward</td>
<td></td>
</tr>
<tr>
<td>Light Goes Off</td>
<td></td>
</tr>
<tr>
<td>Stop</td>
<td></td>
</tr>
<tr>
<td>Contact Object</td>
<td></td>
</tr>
<tr>
<td>Turn left, drive in reverse, set Reverse Timer for 3 sec.</td>
<td></td>
</tr>
<tr>
<td>Reverse Timer Expires</td>
<td></td>
</tr>
<tr>
<td>Turn straight, simulate Light Goes Off event</td>
<td></td>
</tr>
</tbody>
</table>

Now, when the internal event Reverse Timer Expires occurs, the machine should straighten its wheels and simulate a Light Goes Off event. The simulation of a Light Goes Off event must modify the “last state” variable of the light.\(^1\) In this way, if the light is still on, it will trigger a Light Goes On event and begin driving forward. This structure appears to be entirely governed by the events and services and should behave as the specifications demand. To be sure, we can start testing its behavior. No hardware is required, simply the hypothesis for a sequence of events. The designer can, in his/her head or on a piece of paper, work through how the machine would behave.

---

\(^1\)This will require that the scope of the state variable be expanded beyond the event checking function, an undesirable state of affairs. In the State Machines section, we will examine how to eliminate this.
As an example of debugging the design, consider what appears to be a reasonable simplification of the service for the Reverse Timer Expires event. What would happen if, rather than simulate Light Goes Off, we simply performed Drive Forward? It seems like a reasonable simplification, let's test it. What would happen if, while we were backing up, we ran into the dark? The Light Goes Off event would fire and we would stop. Then, when the Reverse Timer Expires, we would Drive Forward out of the dark, not exactly the behavior that we were looking for. Here, we tested a possible simplification and found a potential error before we even started coding.

In working through this process, it will become clear that there is a possible conflict between the design and the specifications. Consider this sequence:

Contact Object
While Reverse Timer is running, Light Goes Off

In the design described above, as soon as the light goes off, the motors will stop. A closer look at the specification shows that this may not be the correct behavior.

The specifications state: “If the cockroach encounters an object, it turns left, reverses direction, and runs backwards for 3 seconds.” Running backwards for 3 seconds may, if the machine runs into the dark during that time, conflict with the specification that the machine “stops when in the dark.”

The process of evaluating the proposed software design has uncovered an ambiguity in the specifications. Without the emphasis on design that a methodology like this provides, it is likely that this ambiguity would not have been uncovered until the customer actually saw the behavior and said “That's not what I wanted.” One of the real strengths of an event driven programming model is that it allows the designer to begin to evaluate and test the program design very early in the process.

5.7 SUMMARY OF EVENT DrIVEN PROGRAMMING

The event driven programming model represents a way of thinking about software that lends itself very nicely to situations where there are many inputs, and the timing of events on the inputs is unpredictable. It supports the decomposition of the problem into a set of relatively simple and short event checking and service routines. The behavior of the resulting program is defined by the events that are tested for and the services carried out in response to the events. It is an approach that supports the testing and evaluation of program designs before implementation (coding).

While the event driven programming model alone is capable of dealing with many problems, it may fall short when tackling many of the more complex problems that you may encounter. However, when combined with the concept of state machines, event driven programming is capable of tackling almost any problem, independent of complexity.

5.8 STATE MACHINES

While event driven programming by itself will allow you to successfully tackle a wide variety of embedded software problems, it really comes into bloom when it is combined with a concept called state machines. This combination can tackle just about any problem that you can throw at it while retaining the simplicity and design focus of event driven programming.

State machines are formally referred to as finite state machines (FSM) or finite state automata (FSA). These “machines” are constructs that are used to represent the behavior of a reactive system. Reactive systems are those whose inputs are not all active at a single point
in time. Hopefully, this sounds a lot like the situations where event driven programming would be useful because the two concepts go hand in hand. Events are the driving forces in state machines.

At its simplest, an FSM consists of a set of states and a set of defined transitions between those states. At any point in time, the state machine can be in only one of the possible states. States embody the concept of time duration. Time is passing while the state machine is in a state. For example, the cockroach we explored in Section 5.6 may be in a state of driving forward for an indefinite period of time. As a result, states are most commonly associated with gerund words (e.g., pumping, driving, filling, waiting). In contrast, transitions between the states are treated as instantaneous and occur in response to events. It is probably easiest to understand state machines by examining the graphical depiction, known as a finite state diagram (FSD) or state transition diagram (STD), of a simple example (Figure 5.4).

The FSD shown in Figure 5.4 describes the behavior of an electronic keypad lock whose combination is 2-1-8. The FSM can be in one of the four possible states: NoneRight, 1Right, 2Right, and Open. The arrows between the state bubbles represent the transitions, labeled with the event that triggers that transition. The bubbles represent the states. While in a state, the FSM is waiting (gerund word) for an event to cause it to transition.

The lock FSM transitions from NoneRight to 1Right when the digit “2” is entered. When in the 1Right state, entering a “1” causes a transition to the 2Right state; any other entry returns the FSM to NoneRight. This pattern continues until we are ready to examine the transition from 2Right that is triggered by an entry of “8.” This transition is labeled in two parts. The upper part (“numerator”) calls out the event that triggers the transition and the lower part (“denominator”) describes an action associated with taking the transition. In this case, the action is to open the lock. The actions are one of the ways in which state machines actually do something.

When the state diagram appears to be complete, the next step is to test it. In this phase, we imagine sequences of events and examine how the system described by the FSM would behave. It is pretty easy to convince yourself that for any sequence of three digits, the FSM will only unlock for 2-1-8. But what about four-digit sequences? Let’s try 2-1-8-1. The FSM ends up in the NoneRight state with the lock locked. That’s good! From this example sequence, it is pretty easy to extrapolate that any four-digit sequence that begins with the correct combination will leave the lock locked. What about 1-2-1-8? That leaves the lock open, as will any other four-digit sequence that ends in 2-1-8!

We are now faced with a decision. We could accept this behavior, though it is probably not exactly what we expected. Or we could modify the design in some way to eliminate the undesired response. For the purposes of this discussion, the decision about how to treat the
situation is of less importance than the fact that we found a possible error in our design. Notice, we just tested our design and found a potential flaw before writing the first line of code! This is one of the real strengths of a methodical use of state machines. The design can be tested easily and repeatedly before any code is written. This is another example of "design time" debugging vs. "implementation time" debugging.

5.9 A STATE MACHINE IN SOFTWARE

While there are a number of ways that you can go about implementing a state machine in software, the most straightforward (though generally not the best) approach is a series of nested IF-THEN-ELSE statements. Using this approach, there is a series of IF clauses that test for the possible machine states. Within each of those state tests, there is another series of nested IF clauses that handle the events that could occur while in that state. This combination of IF statements are wrapped inside a function that maintains a static local variable to track the current state of the machine. This state machine function takes at least one parameter that passes the most recent event to the state machine. For the lock example that we were just considering, the state machine function, implemented in C, might look like Listing 5.6.

```
void LockStateMachineIF( unsigned char NewEvent)
{
    static unsigned char CurrentState = NoneRight;
    unsigned char NextState;

    if( CurrentState == NoneRight)
    {
        if( NewEvent == KeyEqual2) /* Key == '2'? */
            NextState = OneRight;
        /* no else clause needed, we are already in NoneRight */
    }else if( CurrentState == OneRight)
    {
        if( NewEvent == KeyEq1) /* Key == '1'? */
            NextState = TwoRight;
        else
            NextState = NoneRight; /* Bad Key go back to none */
    }else if( CurrentState == TwoRight)
    {
        if( NewEvent == KeyEq8) /* Key == '8'? */
            NextState = Open;
            OpenLock();
        else
            NextState = NoneRight; /* Bad Key go back to none */
    }else if( CurrentState == Open)
    {
        NextState = NoneRight;
        LatchLock();
    }
    CurrentState = NextState; /* update the current state variable */
    return;
}
```

LISTING 5.6

C implementation of the combination lock state machine using a nested IF structure.
The main function to run this state machine would look something like Listing 5.7.

```c
void main(void)
{
    unsigned int KeyReturn;
    while(1)
    {
        KeyReturn = CheckKeys(); /* check for events */
        if( KeyReturn != NO_KEY )
            LockStateMachine(KeyReturn); /* run state machine */
    }
}
```

LISTING 5.7
The main() function that would be used with the state machine function of Listing 5.6.

This program would run forever ["while (1)"], calling the event checking routine CheckKeys() to check for key input. Every time that the event checker found that there had been a new key pressed, the LockStateMachine() function would be called with a parameter that indicated which key had been pressed. In this case, the event:service pair would be CheckKeys:LockStateMachine.

Although the nested IF clause approach works for problems like this, where there are only a few states and only one or two paths out of each state, as the state machine becomes more complex, it will be more robust, easier to read, and often generate more efficient code if the problem is expressed using a nested SWITCH:CASE structure, rather than the nested IF-THEN-ELSE clauses. As an example of this structure, the following function (Listing 5.8) duplicates the lock state machine function using a nested SWITCH:CASE structure in C.

```c
void LockStateMachineCase( unsigned char NewEvent )
{
    static unsigned char CurrentState = NoneRight;
    unsigned char NextState;
    switch( CurrentState )
    {
        case NoneRight :
            switch( NewEvent )
            {
                case '2':
                    NextState = OneRight;
                    break;
                default: /* we are already in NoneRight */
                    break;
            } break;
        case OneRight :
            switch( NewEvent )
            {
                case '1':
                    NextState = TwoRight;
                    break;
                default: /* anything else sends us back */
                    NextState = NoneRight;
                    break;
            } break;
    }
}
```
Listing 5.8

C implementation of the combination lock state machine using a SWITCH:CASE structure.

While the C code for this simple function appears noticeably longer than that for the nested IF clause version, the resulting machine code generated by the compiler is comparable. The clarity of the code is improved by the labeling of the event cases as well as the explicit default case. As the complexity of the state machine grows, both in terms of the number of states and number of active events, the clarity and efficiency of the SWITCH:CASE structure becomes more evident. For anything more complex than the simplest state machines, the SWITCH:CASE structure will be preferable.

5.10 The Cockroach Example as a State Machine

As another example, let's cast the cockroach behavior from the Event Driven Programming section as a state machine. As the events and actions remain the same, this is simply another way of representing the problem. In this case, you might come up with a state diagram that looks something like Figure 5.5.

Here, the behavior of the cockroach is described as being in one of the three states (Hiding, Driving Forward, and Backing Up), and the same set of events are used to trigger transitions between these states.

The diagram also introduces another feature of state machines: guard conditions. Notice that, while backing up, the Timer Expires event triggers one of the two possible transitions, depending on the current state of the light. The current state of the light is the guard condition on those two transitions. In order to take either of the transitions, the event must occur and the guard condition must be true.

It is important to be very careful whenever a single event will take one of two transitions based on guard conditions. In particular, the two guard conditions on the transitions
must not overlap one another. If this were not the case, it would be possible to have the event occur and the response would be indeterminate, since both guards had been met.

As drawn, this state machine captures the specifications without the ambiguity that we discovered in the Event Driven Programming section. Notice that the Light Goes Off event is only responded to if the state machine is in the Driving Forward state. In this way, if the light goes out while backing up, it will continue to back up for 3 seconds. Only when the timer expires will it either resume going forward or stop, based on the current state of the light.

The state machine representation has another benefit over a pure events and services approach. In the events and services solution, we needed to simulate a Light Goes Off event in order to get the desired behavior. This required that the scope of the variable holding the last state of the light be expanded beyond the Light Goes On event checker. That is not necessary in the state machine solution.

If we make the problem just a little more complex, we can really see how the state machine representation becomes useful. Let’s add a response to the back bumper that will only be active while we are backing up. Much like the response to the front bumper, the cockroach should change direction (go forward) for a period of time and turn (to the right this time). When this maneuver is complete, it should return to the Backing Up State. Adding this to the state machine requires adding one new state (Evading Forward), one new event (Rear Bumper Hit), one new action (Wheels Right; Motor Forward), and two new state transitions (Figure 5.6).

In this revised state machine, there are two distinct responses to the Timer Expires event. If we are Backing Up, we should straighten the wheels and go forward or stop, depending on the guard condition. If we are Evading Forward, we should always set the wheels left and go into reverse. The response to the event depends on what we are currently doing—that is, what state we are in when the event occurs. This is a situation that would be messy to handle using only events and services. A state machine is an excellent way to capture that type of behavior.

While this state machine might not be the final design (e.g., it won’t stop while Evading Forward), it does provide an easily understood representation of the behavior. That is one of the key strengths of using state diagrams: they allow you to capture the behavior of the design very early in the process, easily present it to others, and immediately begin testing it.

The thing to emphasize at this point is that we have described the software to control a system with an interesting behavior in very short order. Equally importantly, the actual code that would need to be written to implement the individual functions described is relatively simple. The combination of focusing on events and a state machine representation allows us
to easily decompose the problem into relatively simple subproblems (test bumper, test for light goes on, drive forward ...) that are then easier to code without errors. The state machine captures the complexity of the desired design behavior in an easily understood framework. Filling in the framework with the code from the simple subproblems will give us a working program in a much shorter time than would be possible without the event driven/state machine paradigm.

5.11 HOMEWORK PROBLEMS

5.1 Work with a DVD player and develop an exhaustive set of events that you think the software must sense and respond to.

5.2 Identify the events and draw an STD for the behavior of a DVD player. Limit yourself to the events generated by the DVD player (ignore the remote control).

5.3 Given a function GetRoomTemperature() that returns the room temperature and a variable, SetPoint, write a pair of functions in pseudo-code called IsTemperatureTooHot() and IsTemperatureTooCool() that return TRUE-FALSE values and together implement a hysteresis band around the SetPoint temperature.

5.4 Work with an answering machine to develop the list of events that the machine's software must respond to.

5.5 You are writing code to implement cruise control for a car. Given a function GetVehicleSpeed() that returns the speed of a vehicle, write a single function TestAccelDecel() in pseudo-code that returns, NeedAccelerate, NeedDecelerate, or SpeedOK based on the speed relative to a set point DesiredSpeed. To provide noise immunity, implement hysteresis around the switch point.

5.6 In an application, the state of a switch is determined by reading a variable called PortA. The state of the switch is indicated by the state value of bit 4 within the byte variable PortA. In pseudo-code, write an event checking function TestSwitch() that returns Opened, Closed, or NoChange depending on what has happened since the last time TestSwitch() was called.

5.7 Work with a microwave oven to develop the list of events that the oven's software must respond to.
5.8 Figure 5.7 shows a simple microwave oven. Draw an STD that captures the behavior described by the following description as follows.

(a) The Open button opens the door, stops cooking, and holds time.
(b) The Clear button clears the timer. If cooking was active, it is disabled.
(c) The Start button starts cooking for whatever time has been set. While cooking, the timer decrements to zero and turns off cooking when it reaches zero.
(d) The Popcorn button forces a time of 2 minutes to be set into the timer.
(e) The Def/Light button forces the power level to 50%.
(f) Time is set by twisting the dial and then pressing the button at the center of the dial. No action takes place until the button is pressed. At that point, the time on the dial is entered into the timer.
(g) There is a switch connected to the door to show whether it is open or closed.

5.9 Assume that the switches from the microwave oven in Problem 5.8 are available in a single variable (Switches), that the timer can be set with a function SetTimer(), that the timer can be read with a function IsTimerExpired(), and that the power level can be set with a function SetPower(). In pseudo-code, write routines for the required event checkers, state machine function, and main() to implement your design from Problem 5.8.

5.10 Draw the STD for a soft drink machine that accepts nickels, dimes, and quarters and sells drinks that cost $0.75 each.

5.11 Draw a state diagram for modified combination lock like that shown in Figure 5.4. Make your lock such that it will not accept unlock if four or more digits are entered that end with the combinations that end in 2-1-8 sequence.

5.12 There are many common behaviors that can be described using state machines. As an example, identify the relevant events, and draw a state diagram to describe the behavior of a "Shy Party Guest" (a guest who does not initiate conversations but participates when someone else initiates).

FURTHER READING

