Bitter, Rick et al "State Machines"
LabVIEW Advanced Programming Techniques
Boca Raton: CRC Press LLC, 2001
Every programmer is looking for a faster, easier way to develop software. One tool that has been used by programmers in multiple software platforms is the state machine. A state machine is a very flexible tool that can be used by LabVIEW programmers to make their code easier to maintain, easier to document, and easier to reuse. In addition, state machines allow the program to change the way it executes based on inputs and results of the application. This chapter will discuss state machine fundamentals and provide examples of how to implement different versions of state machines.

3.1 INTRODUCTION

State machines revolve around three concepts: the state, the event, and the action. No state machine operates effectively without all three components. This section will define all three terms and help you identify meaningful states, events, and actions in your programming. Major mistakes programmers make when working with state machines is not defining appropriate states. We will begin with the concept of state.

“State” is an abstract term, and programmers often misuse it. When naming a state, the word “waiting” should be applied to the name of the state. For example, a state may be waiting for acknowledgment. This name defines that the state machine is pending a response from an external object. States describe the status of a piece of programming and are subject to change over time. Choosing states wisely will make the development of the state machine easier, and the robustness of the resulting code much stronger. Relevant states allow for additional flexibility in the state machine because more states allow for additional actions to be taken when events occur.

Events are occurrences in time that have significant meaning to the piece of code controlled by the state machine. An event that is of significance for our previous example is the event “Acknowledgment Received.” This external occurrence will inform the state machine that the correct event has occurred and a transition from states is now appropriate. Events can be generated internally by code controlled by the state machine.

Actions are responses to events, which may or may not impact external code to the state machine. The state machine determines which actions need to be taken when a given event occurs. This decision of what action needs to be taken is derived from two pieces of information: the current state and the event that has occurred. This pair of data is used to reference a matrix. The elements of this matrix contain the action to perform and the next state the machine should use. It is possible, and
often desirable, for the next state to be equal to the current state. Examples in this section will demonstrate that it is desirable to have state changes occur only when specific actions occur. This type of behavior is fairly typical in communications control. Unless a specific sequence of characters arrives, the state should not change, or perhaps the state machine should generate an error.

The state machine itself always makes state changes. The current state is not normally given to code external to the state machine. Under no circumstances should external code be allowed to change the current state. The only information external code should give to the state machine is an event that has occurred. Changing state and dictating actions to perform is the responsibility of the state machine.

3.1.1 State Machines in LabVIEW

A state machine, in simple terms, is a case structure inside a While loop, as shown in Figure 3.1. The While loop provides the ability to continuously execute until the conditional operator is set “false.” The case statement allows for variations in the code to be run. The case that is selected to run can be, and usually is, determined in the previous iteration of the While loop. This allows for a relatively simple block of code to make decisions and perform elegant tasks. In its simplest form, a state machine can be a replacement for a sequence structure. In more complex forms of the state machine, the resulting structure could be used to perform the operations of a test executive. The topic of state machines is covered in a number of places, including the National Instruments LabVIEW training courses; however, there is not a lot of depth to the discussions. This chapter will describe the types of state machines, the uses, the pitfalls, and numerous examples.

When used properly, the state machine can be one of the best tools available to a LabVIEW programmer. The decision to use a state machine, as well as which type to use, should be made at an early stage of application development. During the design or architecting phase, you can determine whether the use of a state machine is appropriate for the situation. Chapter 4 discusses how to approach application development including the various phases of a development life cycle.
3.1.2 When to Use a State Machine

There are a number of instances where a state machine can be used in LabVIEW programming. The ability to make a program respond intelligently to a stimulus is the most powerful aspect of using state machines. The program no longer needs to be linear. The program can begin execution in a specified order, then choose the next section of code to execute based on the inputs or results of the current execution. This can allow the program to perform error handling, user-selected testing, conditional-based execution, and many other variations. If the programmer does not always want the code to be executed in the same order for the same number of iterations, a state machine should be considered.

An example of when a state machine is useful to a programmer is in describing the response to a communications line. An automated test application that uses UDP to accept commands from another application should be controlled with a state machine. Likely, states are “waiting for command,” “processing command,” and “generating report.” The “waiting for command” state indicates that the application is idle until the remote control issues a command to take action. “Processing command” indicates that the application is actively working on a command that was issued by the remote application. “Generating report” notes that the state for command has completed processing and output is pending.

Events that may occur when the code is executing are “command received,” “abort received,” “error occurred,” and “status requested.” All of these possibilities have a different action that corresponds to their occurrence. Each time an event occurs, the state machine will take a corresponding action. State machines are predictable; the matrix of events, actions, and states is not subject to change.

The purpose of the state machine is to provide defined responses to all events that can occur. This mechanism for control is easily implemented, is scalable for additional events, and always provides the same response mechanism to events. Implementing code to respond to multiple events without the state machine control leads to piles of “spaghetti code” that typically neglect a few events. Events that are not covered tend to lead to software defects.

3.1.3 Types of State Machines

There are a number of different styles of state machines. To this point there is no defined convention for naming the style of a state machine. In an effort to standardize the use of state machines for ease of discussion, we propose our own names for what we feel are the four most common forms of state machines in use. The four styles are the Sequence, the Test Executive, the Classical, and the Queued style state machines. Discussions of the four types of state machines, as well as examples of each type, follow in Sections 3.3 to 3.6.

3.2 Enumerated Types and Type Definitions

In the introductory chapters on LabVIEW programming, we stated that an “enumerated type control” is similar to a text ring. The enumerated type control is basically
a list of text values associated with a numeric value. The main difference for the enumerated control is that the string is considered part of the data type. The enumerated control can be found in the “List & Ring” section of the Control palette. The default data representation of the enumerated control is an unsigned word. However, its representation can be changed to unsigned byte or unsigned long by popping up on the control and selecting Representation. When an enumerated type control is copied to the back panel, the result is an enumerated constant. Using the “numeric” section of the Function palette can also create an enumerated constant.

Enumerated constants can make state machines easier to navigate and control. When an enumerated type is connected to a case structure, the case indicator becomes the enumerated text instead of a numeric index. When a programmer is using a case structure with a number of cases, navigation of the case structure becomes difficult if the input is a numeric constant. When the user clicks on the case structure selector, a list of case numbers is shown. It is difficult for the user to determine which case does what, requiring the user to go through a number of cases to find the one that is desired. A benefit of using an enumerated constant with the case structure is readability. When someone clicks on the selector of a case structure controlled by an enumerated-type input, the lists of cases by name are shown. If the enumerated constant values are used to describe the action of the case, the user can easily find the desired case without searching through multiple cases. Similarly, you can use strings to drive state machine structures to enhance readability. However, this is only available if you are using LabVIEW 5.0 or later. When enumerated or string constants are used with a state machine, there are a few added advantages. When the state machine passes the next state to execute to the case structure, the state to be executed becomes obvious. When a state branches to subsequent states, the user can see by the constant which state will execute next. If numerics are used, the person going through the code will have to go through the states to see what is next.

A second advantage involves the maintenance of existing code. When numeric inputs are used to control the state machine, a numeric constant will point to whichever state corresponds to the defined index. A better way to aid in modifications is the use of enumerated types. When the order of states is changed, the enumerated constants will still be pointing to the state with the matching name. This is important when you change the order of, add, or remove states. The enumerated constants will still point to the correct state. It should be noted that in the event a state is added, the state needs to be added to the enumerated constant in order to make the program executable; however, the order of the enumerated constant does not have to match the order of the case structure. This problem does not exist when using string constants to drive the case structure. This leads into the next topic, type definitions used with state machines.

3.2.1 Type Definitions Used with State Machines

A “type definition” is a special type of control. The control is loaded from a separate file. This separate file is the master copy of the control. The default values of the control are taken from this separate file. By using the type definition, the user can
use the same control in multiple VIs. The type definition allows the user to modify
the same control in multiple VIs from one location.

The benefit of using type definitions with state machines is the flexibility
allowed in terms of modifications. When the user adds a case in the state machine,
each enumerated type constant will need to have the name of the new case added to
it. In a large state machine, this could be a very large and tedious task. In addition,
there are many opportunities for mistakes. If the enumerated constant is created from
a type definition, the only place the enumerated type needs to be modified is in the
control editor. Once the type definition is updated, the remaining enumerated con-
stants are automatically updated. No matter how hard we all try, modifications are
sometimes necessary; the use of type definitions can make the process easier.

3.2.2 Creating Enumerated Constants and Type Definitions

Selecting Enumerated Type from the List & Ring section of the Tools pallete creates
the enumerated control on the front panel. The user can add items using the Edit
Text tool. Additional items can be added by either selecting Add Item Before or After
from the popup menu, or pressing Shift + Enter while editing the previous item.
The enumerated constant on the code diagram can be created by copying the control
to the code diagram via “copy and paste,” or by dragging the control to the code
diagram. Alternative methods of creating enumerated constants include choosing
Create Constant while the control is selected, and selecting the enumerated constant
from the Function pallete.

To create an enumerated-type definition, the user must first create the enumerated-
type control on the front panel. The user can then edit the control by either double-
clicking on the control or selecting Edit Control from the Edit menu. When the
items have been added to the enumerated control, the controls should be saved as
either a Type Definition or a Strict Type Definition. The strict type definition forces
all attributes of the control, including size and color, to be identical. Once the type
definition is created, any enumerated constants created from this control are auto-
matically updated when the control is modified, unless the user elects to not auto
update the control.

3.2.3 Converting between Enumerated Types and Strings

If you need to convert a string to an enumerated type, the Scan from String function
can accomplish this for you. The string to convert is wired to the string input. The
enumerated type constant or control is wired to the default input. The output becomes
the enumerated type constant. The use of this method requires the string to match
the enumerated type constant exactly, except for case. The function is not case-
sensitive. If the match is not found, the enumerated constant wired to the default
input is the output of the function. There is no automatic way to check to see if a
match was found. Converting enumerated types into strings and vice versa is helpful
when application settings are being stored in initialization files. Also, application
logging for defect tracking can be made easier when you are converting enumerated
types into strings for output to your log files.
There are two methods that can be used to ensure only the desired enumerated type is recovered from the string input. One option is to convert the enumerated output back to a string and perform a compare with the original string. The method needed to convert an enumerated to a string is discussed next. A second option is to use the **Search 1-D Array** function to match the string to an array of strings. Then you would use the index of the matched string to typecast the number to the enumerated type. This assumes that the array of strings exactly matches the order of the items in the enumerated type. The benefit of this method is that the **Search 1-D Array** function returns a –1 if no match is found.

If the programmer wants to convert the enumerated-type to a string value, there is a method to accomplish this task. The programmer can wire the enumerated type to the input of the format into string function in the Function palette. This will give the string value of the selected enumerated type.

### 3.2.4 **Drawbacks to Using Type Definitions and Enumerated Controls**

The first problem was mentioned in Section 3.2.2. If the user does not use type definitions with the enumerated type constant, and the constant needs to be modified, each instance of the constant must be modified when used with a state machine. In a large state machine, there can be large number of enumerated constants that will need to be modified. The result would be one of two situations when changes to the code have to be made: either the programmer will have to spend time modifying or replacing each enumerated control, or the programmer will abandon the changes. The programmer may decide the benefit of the changes or additions do not outweigh the effort and time necessary to make the modifications. This drawback limits the effectiveness of the state machine since one of the greatest benefits is ease of modification and flexibility.

The programmer needs to be careful when trying to typecast a number to the enumerated type. The data types need to match. One example is when an enumerated type is used with a sequence-style state machine. If the programmer is typecasting an index from a While or For loop to an enumerated type constant, either the index needs to be converted to an unsigned word integer, or the enumerated type to a long integer data type. The enumerated data type can be changed in two ways. The programmer can either select the representation by right-clicking on the enumerated constant and selecting the representation, or by selecting the proper conversion function from the Function palette.

If the programmer needs to increment an enumerated data type on the code diagram, special attention needs to be paid to the upper and lower bounds of the enumerated type. The enumerated values can wrap when reaching the boundaries. When using the increment function with an enumerated constant, if the current value is the last item, the result is the first value in the enumerated type. The reverse is also true; the decrement of the first value becomes the last value.
3.3 SEQUENCE-STYLE STATE MACHINE

The first style of state machine is the sequence style. This version of the state machine is, in essence, a sequence structure. This version of the state machine executes the states (cases) in order until a false value is wired to the conditional terminal. There are a couple ways in which to implement this style of state machine. The first and simplest way is to wire the index of the While loop to the case statement selector. Inside each case, a Boolean constant of “true” is wired to the While loop conditional terminal. The final case passes a false Boolean to the conditional terminal to end execution of the loop. Figure 3.2 shows a sequence machine using the index of a While loop. This type of state machine is more or less “brain dead” because, regardless of event, the machine will simply go to the next state in the sequence.

A second way to implement the sequence-style state machine is to use a shift register to control the case structure. The shift register is initialized to the first case, and inside each case the number wired to the shift register is incremented by one. Again, the state machine will execute the states in order until a false Boolean is wired to the conditional terminal of the While loop. This implementation of a state machine is modified for all of the other versions of state machines described in this chapter. Figure 3.3 shows a simple sequence machine using a shift register.

FIGURE 3.2

FIGURE 3.3
3.3.1 **When to Use a Sequence-Style State Machine**

This style of state machine should be used when the order of execution of the tasks to be performed is predefined, and it will always execute from beginning to end in order. This state machine is little more than a replacement for a sequence structure. We generally prefer to use this type of structure instead of sequences because it is far easier to read code that uses shift registers than sequence locals. The single biggest problem with the sequence structure is the readability problems caused by sequence locals. Most programmers can relate to this readability issue.

The biggest benefit is gained when the sequence-style state machine is implemented with enumerated-type constants. When enumerated types are used, the code becomes self-documenting (assuming descriptive state names are used). This allows someone to see the function of each action at a glance.

3.3.2 **Example**

When writing test automation software there is often a need for configuring a system for the test. Generally, there are number of setup procedures that need to be performed in a defined order. The code diagram in Figure 3.4 shows a test that performs the setup in the basic coding procedure. This version of the VI performs all of the steps in order on the code diagram. The code becomes difficult to read with all of the additional VIs cluttering the diagram. This type of application can be efficiently coded through the use of the sequence-style state machine.

There are a number of distinct steps shown in the block diagram in Figure 3.4. These steps can be used to create the states in the state machine. As a general rule, a state can be defined with a one-sentence action: setup power supply, set system time, write data to global/local variables, etc. In our example, the following states can be identified: open instrument communications, configure spectrum analyzer, configure signal generator, configure power supply, set display attributes and variables to default settings, and set RF switch settings.

Once the states have been identified, the enumerated control should be created. The enumerated control is selected from the List & Ring group of the Control palette. Each of the above states should be put into the enumerated control. You should not be concerned with how long the state name is as long as it is readable. The label on the case statement will go as wide as the case statement structure.

There are two main factors to consider when creating state names. The first is readability. The name should be descriptive of the state to execute. This helps someone to see at a glance what the states do by selecting the Case Statement selector. The list of all of the states will be shown. The second factor to consider is diagram clutter or size. If enumerated constants are used to go to the next state, or are used for other purposes in the code, the size of the constant will show the entire state name. This can be quite an obstacle when trying to make the code diagram small and readable. In the end, compromises will need to be made based on the specific needs of the application.

After the enumerated control has been created, the state machine structure should be wired. A While loop should be selected from the Function palette and placed on the diagram with the desired “footprint.” Next, a case structure should be placed
inside the While loop. For our example we will be using the index to control the state machine. This will require typecasting the index to the enumerated type to make the Case Selector show the enumerated values. The typecast function can be found in the Data Manipulation section of the advanced portion of the Function palate. The index value is wired to the left portion of the typecast function. The enumerated control is wired to the middle portion of the function. The output of the typecast is then wired to the case structure. To ensure no issues with data representations, either the representation of the enumerated control or the index should be changed. We prefer to change the index to make sure someone reading the code will see what is being done. Since the index is a long integer, it will need to be converted
to an unsigned word to match the default representation of the enumerated control.
The Conversion functions are part of the numeric section of the function pallete.

Now that the enumerated control has been wired to the case structure, the additional states can be added to match the number of states required. With the structure in place, the code required to perform each state should be placed into the appropriate case. Any data, such as instrument handles and the error cluster, can be passed between states using shift registers. The final and possibly most important step is to take care of the conditional terminal of the While loop. A Boolean constant can be placed in each state. The Boolean constant can then be wired to the conditional terminal. Since the While loop will only exit on a false input, the false constant can be placed in the last state to allow the state machine to exit. If you forget to wire the false Boolean to the conditional terminal, the default case of the case statement will execute until the application is exited.

At this point, the state machine is complete. The diagram in Figure 3.5 shows the resulting code. When compared to the previous diagram, some of the benefits of state machines become obvious. Additionally, if modifications or additional steps need to be added, the effort required is minimal. For example, to add an additional state, the item will have to be added to the enumerated control and to the case structure. That’s it! As a bonus, all of the inputs available to the other states are now available to the new state.

### 3.4 TEST EXECUTIVE STYLE STATE MACHINE

The test executive-style state machine adds flexibility to the sequence-style state machine. This state machine makes a decision based on inputs either fed into the machine from sections of code such as the user interface, or calculated in the state being executed to decide which state to execute next. This state machine uses an initialized shift register to provide an input to the case statement. Inside each case, the next state to execute is decided on. An example of this state machine is shown in Figure 3.6.
3.4.1 **WHEN TO USE A TEST EXECUTIVE STYLE STATE MACHINE**

There are a number of advantages to this style of state machine. The most important benefit is the ability to perform error handling. In each state, the next state to execute is determined in the current state. If actions were completed successfully, the state machine will determine what state to execute next. In the event that problems arise, the state machine can decide to branch to its exception-handling state. The next state to execute may be ambiguous; there is no reason for a state machine to execute one state at a time in a given order. If we wanted that type of operation, a sequence state machine or a sequence diagram could be used. A test executive state machine allows for the code to determine the next state to execute given data generated in the current state. For example, if a test running in the current state determined that the Device Under Test (DUT) marginally makes spec, then the state machine may determine that additional tests should be performed. If the DUT passes the specified test with a lot of margin, the state machine may conclude that additional testing is not necessary.

The user can make one of the cases perform dedicated exception handling. By unbundling the status portion of the error cluster, the program can select between going to the next state to execute or branching off to the Error State. The Error State should be a state dedicated to handling errors. This state can determine if the error is recoverable. If the error is recoverable, settings can be modified prior to sending the state machine back to the appropriate state to retry execution. If the error is not recoverable, the Error State, in conjunction with the Close State, can perform the cleanup tasks involved with ending the execution. These tasks can include writing data to files, closing instrument communications, restoring original settings, etc. Chapter 6 discusses the implementation of an exception handler in the context of a state machine.

3.4.2 **RECOMMENDED STATES FOR A TEST EXECUTIVE STATE MACHINE**

Test executive state machines should always have three states defined: Open, Close, and Error. The Open state allows for the machine to provide a consistent startup and initialization point. Initialization is usually necessary for local variables, instrument
communications, and log files. The existence of the Open state allows the state machine to have a defined location to perform these initialization tasks.

A Close state is required for the opposite reason of that of the Open state. Close allows for an orderly shutdown of the state machine’s resources. VISA, ActiveX, TCP, and file refnums should be closed off when the state machine stops using them so that the resources of the machine are not leaked away.

When this type of state machine is developed using a While loop, only one state should be able to wire a false value to the conditional terminal. The Close state’s job is to provide the orderly shutdown of the structure, and should be the only state that can bring down the state machine’s operation. This will guarantee that any activities that must be done to stop execution in an orderly way are performed.

The Error state allows for a defined exception-handling mechanism private to the state machine. This is one of the biggest advantages of the test executive style over “brain dead” sequence-style machines. At any point, the machine can conclude that an exception has occurred and branch execution to the exception handling state to record or resolve problems that have been encountered. A trick of the trade with this type of state machine is to have the shift register containing the next state use two elements. This allows for the Error state to identify the previous state and potentially return to that state if the exception can be resolved.

The Error state should not be capable of terminating execution of the state machine; this is the responsibility of the Close state. If your exception-handling code determines that execution needs to be halted, the Error state should branch the state machine to the Close state. This will allow for the state machine to shut down any resources it can in an orderly manner before stopping execution.

### 3.4.3 Determining States for Test Executive State Machines

When working with a test executive machine, state names correlate to an action that the state machine will perform. Each name should be representative of a simple sentence that describes what the state will do. This is a guideline to maximize the flexibility of the state machine. Using complex or compound sentences to describe the activity to perform means that every time the state is executed, all actions must be performed. For example, a good state description is, “This state sets the voltage of the power supply.” A short, simple sentence encapsulates what this state is going to do. The state is very reusable and can be called by other states to perform this activity. A state that is described with the sentence, “This state sets the power supply voltage, the signal generator’s output level, and sends an email to the operator stating that we have done this activity,” is not going to be productive. If another state determines that it needs to change the power supply voltage, it might just issue the command itself because it does not need the other tasks to be performed. Keeping state purposes short allows for each state to be reused by other states, and will minimize the amount of code that needs to be written.

### 3.4.4 Example

This example of the state machine will perform the function of calculating a threshold value measurement. The program will apply an input to a device and measure the resulting output. The user wants to know what level of input is necessary to obtain
an output in a defined range. While this is a basic function, it shows the flexibility of the text executive-style state machine.

The first step should be performed before the mouse is even picked up. In order to code efficiently, a plan should already be in place for what needs to be done. A flowchart of the process should be created. This is especially true with coding state machines. A flowchart will help identify what states need to be created, as well as how the state machine will need to be wired to go to the appropriate states. A flowchart of the example is shown in Figure 3.7.

Once the test has been laid out, the skeleton of the state machine should be created. Again, the While loop and case statement need to be placed on the code diagram. An enumerated control will need to be created with the list of states to be executed. Based on the tasks identified in the flowchart, the following states are necessary: Instrument Setup, Measure Output, Compare to Threshold, Increase Input, Decrease Input, Error, and Close. A better approach is to combine the Increase and Decrease Input states into a Modify Input state that will change the input based on the measurement relationship to the desired output. However, this method makes a better example of state machine program flow and is used for demonstration purposes.

Once the enumerated control is created, an enumerated constant should be made. Right-clicking on the control and selecting create constant can do this. The Instrument Setup state should be selected from the enumerated list. This is the initial state.
to execute. The user needs to create a shift register on the While loop. The input of the shift register is the enumerated constant with the Instrument Setup state selected. The shift register should then be wired from the While loop boundary to the Case Statement selector. Inside each state an enumerated constant needs to be wired to the output of the shift register. This tells the state machine which state to execute next. Once the structure and inputs have been built, the code for each state can be implemented.

The Instrument Setup state is responsible for opening instrument communications, setting default values for front panel controls, and setting the initial state for the instruments. One way to implement the different tasks would be to either break these tasks into individual states or use a sequence-style state machine in the Initialize state. We prefer the second method. This prevents the main state machine from becoming too difficult to read. The user will know where to look to find what steps are being done at the beginning of the test. In addition, the Initialize state becomes easier to reuse by putting the components in one place.

After initializing the test, the program will measure the output of the device. The value of the measurement will be passed to the remainder of the application through a shift register. The program then goes to the next state to compare the measurement to a threshold value. Actually, a range should be used to prevent the program from trying to match a specific value with all of the significant digits. Not using a range can cause problems, especially when comparing an integer value to a real number. Due to the accuracy of the integer, an exact match cannot always be reached, which could cause a program to provide unexpected results or run endlessly.

Based on the comparison to the threshold value, the state machine will either branch to the Increase Input state, Decrease Input state, or the Close state (if a match is found). Depending on the application, the Increase or Decrease state can modify the input by a defined value, or by a value determined by how far away from the threshold the measurement is. The Increase and Decrease states branch back to the Measure Output state.

Although not mentioned previously, each state where errors can be encountered should check the status of the error Boolean. If an error has occurred, the state machine should branch to the Error state. What error handling is performed in this state is dependent on the application being performed. As a minimum, the Error state should branch to the Close state in order to close the instrument communications.

Finally, there should be a way to stop execution. You should never assume a program will complete properly. There should be a way for the program to “time out.” In this example, the test will only execute up to 1000 iterations of the Measure Input state. One way to implement this requirement is to do a comparison of the While loop index. Since the Initialize state is only executed once, the state is negligible. That leaves three states executing per measurement (Measure, Compare, and the Change). The Measure Output state can compare the loop index to 3000 to verify the number of times the application has executed. If the index reaches 3000, the program can either branch to the Close state directly or set an error in the error cluster. By using the bundling tools, the program can set the error Boolean to “true,” set a user-defined code, and place a string into the description. The program can indicate that the test timed out or give some other descriptive error message to let the user know that the value was never found. Another way to implement this "time
“out” is to use shift registers. A shift register can be initialized to zero. Inside the Measurement state, the program can increment the value from the shift register. This value can be compared to the desired number of cycles to determine when the program should terminate execution. Figure 3.8 shows the completed state machine. The code is also included on the CD accompanying this book.

3.5 CLASSICAL-STYLE STATE MACHINE

The classical state machine is taught to computer programming students, and is the most generic of state machine styles. Programmers should use this type of state machine most frequently, and we do not see them often enough in LabVIEW code. The first step to using the classical state machine is to define the relevant states, events, and actions. Once the triad of elements is defined, their interactions can be specified. This concludes the design of the state machine, and coding may begin to implement the design. This section will conclude with a design of a state machine for use with an SMTP mail VI collection. This design will be used to implement a simple mail-sending utility for use with LabVIEW applications.

Step One is to define the states of the machine. States need to be relevant and should be defined with the word “waiting.” Using the term “waiting” helps frame the states correctly. State machines are not proactive; they do not predict events about to happen. The word “waiting” in the state name appropriately describes the state’s purpose.

Once the states for the machine are defined, then the events that are to be handled need to be defined. It is usually better to err on the side of having too many states than too few. Additional states allow for future expandability and general bullet-proofing of the design.

3.5.1 WHEN TO USE A CLASSICAL STYLE STATE MACHINE

Classical state machines are a good design decision when events that occur are coming from outside the application itself. User mouse-clicks, messages coming from a communications port, and ActiveX event handling are three examples. Since
these events may come into the application at any moment, it is necessary to have a dedicated control structure to process them. LabVIEW 5.0 introduced menu customization for LabVIEW applications. Generally, the programmer needs to generate a polling loop to determine which, if any, menu selections were made by a user. State machines fit well into handling these types of user interactions. For example, if a user menu selection would make other menu selections not meaningful, the state machine could be used to determine what menu items needed to be disabled; for example, if your file menu had the option for application logging and a selection for level of logging. If the user determined that he did not want the application to generate a log file of its activities, then setting the level of logging detail is no longer meaningful. The state machine handling the menu polling loop would make the determination that logging detail is not useful and disable the item in the menu.

3.5.2 Example

One of the requirements of this example is to read information from a serial port searching for either user inputs or information returned from another application or instrument. This example will receive commands from a user connected through a serial port on either the same PC or another PC. Based on the command read in from the serial port, the application will perform a specific task and return the appropriate data or message. This program could be a simulation for a piece of equipment connected through serial communications. The VI will return the expected inputs, allowing the user to test their code without the instrument being present. The user can also perform range checking by adjusting what data is returned when the program requests a measurement.

For this style of state machine the states are fairly obvious. There needs to be an Initialize state that takes care of the instrument communication and any additional setup required. The next state is the Input state. This state polls the serial port until a recognized command is read in. There needs to be at least one state to perform the application tasks for the matched input. When a command is matched, the state machine branches to the state developed to handle the task. If more than one state is necessary, the first state can branch to additional test states until the task is complete. When the task is completed, the state machine returns to the Input state. Finally, there needs to be an Error state and a Close state to perform those defined tasks.

The first step is to identify what commands need to be supported. If the purpose of the test is to do simulation work, only the commands that are going to be used need to be implemented. Additional commands can always be added when necessary. For our example, the VI will support the following commands: Identity (ID?), Measurement (Meas), Status (Status), Configure (Config), and Reset (RST). For our example, only one state per command will be created.

Once the commands are identified, the state machine can be created. As in the previous example, the input of the case statement is wired from an initialized shift register. Inside each state, the next state to execute is wired to the output of the shift register. This continues until a false Boolean is wired to the conditional terminal of the While loop. The main structure of this style of state machine is shown below.
The most important state in this style of state machine is the Input state. In our example, the list of commands is wired to a subVI. This subVI reads the serial port until a match is found in the list. When a match is found, the index of the matched command is wired out. This index is then wired to an Index Array function. The other input to this function is an array of enumerated type constants. The list is a matching list to the command list. The first state to execute for a given command should be in the array corresponding to the given command. A quick programming tip: when using this method, the index of the match should be increased by one. Then in the match array of enumerated constants, the first input should be the Error state. Since the match pattern function returns a –1 when no match is found, the index would point to the zero index of the array. This can allow the program to branch to the Error state if no match is found. Then, each command in order is just one place above the original array. The code for this state is shown on page 147.

The VI will continue to cycle through reading the serial port for commands and executing the selected states until the program is finished executing. There should be a way to stop the VI from the front panel to allow the VI to close the serial communications.

In this example, we are simulating an instrument for testing purposes. Using the Random Number Generator function and setting the upper and lower limits can use the measurement outputs to perform range checking. The state can be set up to output invalid data to check the error-handling capabilities of the code as well. This is a nice application for testing code without having the instrument available.

This next example will focus on developing a Simple Mail Transfer Protocol (SMTP) VI. Communications with a mail server are best handled through a state machine. The possibilities of errors and different responses from the server can make development of robust code very difficult. A state machine will provide the needed control mechanism so that responding to the various events that occur during a mail transfer conversation can be handled completely.

Before we begin the VI development, we need to get an understanding of how SMTP works. Typically, we learn that protocols containing the word “simple” are anything but simple. SMTP is not very difficult to work with, but we need to know the commands and responses that are going to present themselves. SMTP is defined in Request For Comments (RFC) 811, which is an Internet standard. Basically, each command we send will cause the server to generate a response. Responses from the server consist of a three-digit number and text response. We are most concerned with the first digit, which has a range from two to five.

The server responses that begin with the digit two are positive responses. Basically, we did something correctly, and the server is allowing us to continue. A response with a leading three indicates that we performed an accepted action, but the action is not completed.

Before we can design the state machine, we need to review the order in which communications should occur and design the states, events, and actions around the way things happen. When designing state machines of any kind, the simplest route to take is to thoroughly understand what is supposed to happen and design a set of states around the sequence of events. Exception handling is fairly easy to add once the correct combinations are understood.

©2001 CRC Press LLC
Figure 3.9 shows the sequence of events we are expecting to happen. First, we are going to create a TCP connection to the server. The server should respond with a “220,” indicating that we have successfully connected. Once we are connected, we are going to send the Mail From command. This command identifies which user is sending the mail. No password or authentication technique is used by SMTP; all you need is a valid user ID. Servers will respond with a 250 code indicating that the user is valid and allowed to send mail. Addressing the message comes next, and this is done with “RCPT TO: <email address>.” Again, the server should respond with a 250 response code. To fill out the body of the message, the DATA command is issued which should elicit a 354 response from the server. The 354 command means that the server has accepted our command, but the command will not be completed until we send the <CRLF>.<CRLF> sequence. We are now free to send the body of the message, and the server will not send another response until we send the carriage return line feed combination. Once the <CRLF>.<CRLF> has been sent, the server will send another 250 response. At this point we are finished and can issue the QUIT command. Servers respond to QUIT with a 220 response and then disconnect the line. It is not absolutely necessary to send the QUIT
command, we could just close the connection and the server would handle that just fine.

As we can see, our actions only happen when we receive a response from the server. The likely events we will receive from the server are 220, 250, and 354 responses for “everything is OK.” Codes of 400 and 500 are error conditions and we need to handle them differently. Several interactions with the server generate both 250 and 220 response codes, and a state machine will make handling them very easy. Our action taken from these events will be determined by our current state. The control code just became much easier to write.

Our event listing will be 220, 250, 354, >400, and TCP Error. These values will fit nicely into an enumerated type. Five events will make for a fairly simple state machine matrix. We will need states to handle all of the boxes in the right column of Figure 3.9. This will allow us to account for all the possible interactions between our application and the mail server.

Surprisingly, we will only need states for half of the boxes in the right column of Figure 3.9. When we receive a response code, the action we take will allow us to skip over the next box in the diagram as a state. We just go to a state where we are waiting for a response to the last action. The combination of Event Received and Current state will allow us to uniquely determine the next action we need to take. This lets us to drive a simple case structure to handle the mail conversation, which is far easier to write than one long chain of SubVIs in which we will have to account for all the possible combinations. The table summarizes all of the states, events, and actions.

We have an action called “Do Nothing.” This action literally means “take no action” and is used in scenarios that are not possible, or where there is no relevant action we need to perform. One of the state/event pairs, Waiting For Hello and 354Received, has a Do Nothing response. This is not a possible response from the server. A response code in the 300 range means that our command was accepted, but we need to do something to complete the action. TCP connections do not require

<table>
<thead>
<tr>
<th>State/Event</th>
<th>200 Received</th>
<th>250 Received</th>
<th>354 Received</th>
<th>&gt;400 Received</th>
<th>TCP Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waiting For Hello</td>
<td>Waiting For Address/ Send from</td>
<td>Waiting For Hello/ Do Nothing</td>
<td>Waiting For Hello/ Do Nothing</td>
<td>Waiting For Hello/ QUIT</td>
<td>Waiting For Hello/ QUIT</td>
</tr>
<tr>
<td>Waiting For Address</td>
<td>Waiting For Address/ Do Nothing</td>
<td>Waiting For Data/ Send Recpt</td>
<td>Waiting For Address/ Do Nothing</td>
<td>Waiting For Address/ QUIT</td>
<td>Waiting for Address/ QUIT</td>
</tr>
<tr>
<td>Waiting For Data</td>
<td>Waiting For Data/ Do Nothing</td>
<td>Waiting</td>
<td>Send Body/ Send Data</td>
<td>Waiting For Data/ Do Nothing</td>
<td>Waiting for Data/ QUIT</td>
</tr>
<tr>
<td>Waiting To Send Body</td>
<td>Waiting To Send Body/ Do Nothing</td>
<td>Waiting To Send Body/ Send Body</td>
<td>Waiting To Quit/ Send Body</td>
<td>Waiting To Send Body/ QUIT</td>
<td>Waiting To Send Body/ QUIT</td>
</tr>
<tr>
<td>Waiting To Quit</td>
<td>Waiting To Quit/ Do Nothing</td>
<td>Waiting To Quit/ QUIT</td>
<td>Waiting To Quit/ QUIT</td>
<td>Waiting To Quit/ QUIT</td>
<td>Waiting To Quit/ QUIT</td>
</tr>
</tbody>
</table>
any secondary steps on our part, so this is not likely to happen. We will be using an array for storing the state/event pairs, and something needs to be put into this element of the array. Do Nothing prevents us from getting into trouble.

![State Machine Diagram](image)

**FIGURE 3.10**

You can see from the table that there is a correct path through the state machine and, hopefully, we will follow the correct path each time we use the SMTP driver. This will not always be the case, and we have other responses to handle unexpected or undesirable responses. For the first row of the state table, TCP errors are assumed to mean that we cannot connect to the mail server, and we should promptly exit the state machine and SMTP driver. There is very little we can do to establish a connection that is not responding to our connection request. When we receive our 220 reply code from the connection request, we go to the Waiting for Address state and send the information on who is sending the e-mail.

The waiting for Address state has an error condition that will cause us to exit. If the Sending From information is invalid, we will not receive our 250 response code; instead, we will receive a code with a number exceeding 500. This would mean that the user name we supplied is not valid and we may not send mail. Again, there is little we can do from the SMTP driver to correct this problem. We need to exit and generate an error indicating that we could not send the mail.

Developing the state machine to handle the events and determine actions is actually very simple. All we need is an internal type to remember the current state,
a case statement to perform the specific actions, and a loop to monitor TCP communications. Since LabVIEW is going to remember which number was last input to the current state, we will need the ability to initialize the state machine every time we start up. Not initializing the state machine on startup could cause the state machine to think it is currently in the Wait to Quit state, which would not be suitable for most e-mail applications.

**Figure 3.10** shows the state/action pair matrix we will be using. The matrix is a two-dimensional array of clusters. Each cluster contains two enumerated types titled “next state” and “action.” When we receive an event, we reference the element in this matrix that corresponds to the event and the current state. This element contains the two needed pieces of information: what do we do and what is the next state of operation.

To use the matrix we will need to internally track the state of the machine. This will be done with an input on the front panel. The matrix and current state will not be wired to connectors, but will basically be used as local variables to the state machine. We do not want to allow users to randomly change the current state or the matrix that is used to drive the machine. Hiding the state from external code prevents programmers from cheating by altering the state variable. This is a defensive programming tactic and eliminates the possibility that someone will change the state at inappropriate times. Cheating is more likely to introduce defects into the code than to correct problems with the state machine. If there is an issue with the state machine, then the state machine should be corrected. Workarounds on state machines are bad programming practices. The real intention of a state machine is to enforce a strict set of rules on a code section is of behavior.

Now that we have defined the matrix, we will write the rest of the VI supporting the matrix. Input for Current State will be put on the front panel in addition to a Boolean titled “Reset.” The purpose of the reset Boolean is to inform the state machine that it is starting and the current internal state should be changed back to its default. The Boolean should not be used to reset the machine during normal operation, only at startup. The only output of the state machine is the action to take. There is no need for external agents to know what the new state of the machine will be, the current state of the machine, or the previous state. We will not give access to this information because it is not a good defensive programming practice. What the state machine looks like to external sections of code is shown in **Figure 3.11**.

The “innards” of the state machine are simple and shown in **Figure 3.12**. There is a case statement that is driven by the current value of the reset input. If this input is “false,” we index the state/event matrix to get the action to perform and the new state for the machine. The new state is written into the local Variable for Current state, and the action is output to the external code. If the reset Boolean is “true,” then we set the current state to Waiting for Hello and output an action, Do Nothing. The structure of this VI could not be much simpler; it would be difficult to write code to handle the SMTP conversation in a manner that would be as robust or easy to read as this state machine.

Now that we have the driving force of our SMTP sending VI written, it is time to begin writing the supporting code. The state machine itself is not responsible
for parsing messages on the TCP link, or performing any of the actions it dictates. The code that is directly calling the state machine will be responsible for this; we have a slave/master relationship for this code. A division of labor is present; the SMTP VI performs all the interfaces to the server, and gets its commands from the state machine. This makes readability easier because we know exactly where to look for problems. If the SMTP VI did not behave correctly, we can validate that the state machine gave correct instructions. Assuming the state machine gave correct instructions, the problem is with the SMTP VI.

State machines work well for dealing with protocols such as SMTP. SMTP sends reply codes back, and the reply codes may be the same for different actions. The 220 reply code is used for both quitting and starting the mail conversation. If you were not using a state machine to determine what to do when you receive a 220 from the server, “tons” of temporary variables and “spaghetti code” would be needed instead. The matrix looks much easier to work with. Instead of following code and tracking variables, you look at the matrix to determine what the code should be doing.

### 3.6 QUEUED-STYLE STATE MACHINE

As the name suggests, the queued-style state machine works with an input queue. Prior to entering the state machine, a queue or input buffer is created. As the state machine executes, the state that has executed is removed from the queue during execution of the state machine. New states can be added to or removed from the
queue based on what happens during execution. The execution of the queued-style state machine can complete by executing the close state when the queue is empty. We recommend always using a Close state as the last element of the queue. This will enable the program to take care of all communications, VISA sessions, and data handling. There is a way to combine these methods through the use of the Default state in the case statement.

There are two ways to implement the queue. The first method is using the LabVIEW queue functions. The Queue palette can be found in the Synchronization palette in the Advanced palette of the Function palette (are you lost yet?). [Functions>>Advanced>>Synchronization>>Queue]. The VIs contained in this palette allow you to create, destroy, add elements, remove elements, etc. For use with the state machine, the program could create a queue and add the list of elements (states to execute) prior to the state machine executing. Inside the While loop, the program could remove one element (state) and wire the state to the case selector of the case structure. If an error occurs, or there is a need to branch to another section of the state machine, the appropriate elements can be added to the queue. The addition can be either to the existing list, or the list could be flushed if it is desired to not continue with the existing list of states.

The use of the LabVIEW Queue function requires the programmer to either use text labels for the case structure, or to convert the string labels to corresponding numeric or enumerated constants. One alternative is to use an array of enumerated types instead of the Queue function (again, string arrays would work fine). The VI can place all of the states into an array. Each time the While loop executes, a state is removed from the array and executed. This method requires the programmer to remove the array element that has been executed and pass the remaining array through a shift register back to the beginning of the state machine, as shown in Figure 3.8.

### 3.6.1 When to Use the Queued-Style State Machine

This style of state machine is very useful when a user interface is used to query the user for a list of states to execute consecutively. The user interface could ask the user to select tests from a list of tests to execute. Based on the selected items, the program can create the list of states (elements) to place in the queue. This queue can then be used to drive the program execution with no further intervention from the user. The execution flexibility of the application is greatly enhanced. If the user decides to perform one task 50 times and a second task once followed by a third task, the VI can take these inputs and create a list of states for the state machine to execute. The user will not have to wait until the first task is complete before selecting a second and third task to execute. The state machine will execute as long as there are states in the buffer. The options available to the user are only limited by the user interface.

### 3.6.2 Example Using LabVIEW Queue Functions

This first example will use the built-in LabVIEW Queue function. In this example, a user interface VI will prompt the user to select which tests need to be executed. The selected tests will then be built into a list of tests to execute, which will be
added to the test queue. Once the test queue is built, the state machine will execute
the next test to be performed. After each execution, the test that has been executed
will be removed from the queue. This example is not for the faint of heart, but it
shows you how to make your code more flexible and efficient.

The first step is creating the user interface. The example user interface here is
a subVI that shows its front panel when called. The user is prompted to select which
tests to execute. There are checkboxes for the user to select for each test. There are
a number of other methods that work as well, such as using a multiple selection
listbox. The queue can be built in the user interface VI, or the data can be passed
to another VI that builds the queue. We prefer to build the queue in a separate VI
in order to keep the tasks separated for future reuse. In this example, an array of
clusters is built. The cluster has two components: a Boolean value indicating if the
test was selected and an enumerated type constant representing the specific test.
There is an array value for each of the options on the user interface.

The array is wired into the parsing VI that converts the clusters to queue entries.
The array is wired into a For loop in order to go through each array item. There are
two case statements inside the For loop. The first case statement is used to bypass
the inner case statement if the test was not selected (a false value). The second case
statement is a state machine used in the true case to build the queue. If a test is
selected, the VI goes to the state machine and executes the state referenced by the
enumerated type constant from the input. Inside the specific cases the appropriate
state name (in string format) is added to the output array. In some instances multiple
cases may be necessary to complete a given task. In these instances, the cases to
eexecute are all added to the output array. This is why the string value of the
enumerated type input is not simply added to the queue. Using the state machine
allows a selected input to have different queue inputs. You would be tied to the name
of the enumerated type if the Format into String function was used. When all of the
array items have been sorted, a close state string is added to the end of the array to
allow the main program to close the state machine.

The final stage of the VI is to build the queue with the inputs from the output
string array. The first step is using the Create Queue function to create a named
queue. The queue has a reference ID just like a VISA instrument. The ID is then
passed into a For loop with an output array of strings. Inside the For loop, each
string is put into the queue using the Insert Queue Element VI. When the VI
completes execution, the reference ID is passed back to the main program. The
queue-building VI is shown in Figure 3.13.

Now that the queue is built, the actual test needs to be created. The main VI
should consist of a state machine. The main structure of the state machine should
be a While loop with the case structure inside. Again, each case, except the Close
state, should wire a “true” Boolean to the condition terminal of the While loop.
The only trick to this implementation is the control of the case statement. In the
beginning of the While loop, the Remove Queue Element VI should be used to get
the next state to execute. Once the state executes, the While loop will return to the
beginning to take the next state from the queue. This will continue until the Close
state is executed and theWhile loop is stopped. In the Close state, the programmer
should use the Destroy Queue VI to close out the operation.

©2001 CRC Press LLC
There is one final trick to this implementation: the wiring of the string input to the state machine. There are two ways to accomplish this task. The first is to create the case structure with the string names for each state. One of the states will need to be made the Default state in order for the VI to be executable. Since there are no defined inputs for a string, one of the cases is required to be “default.” We would suggest making the default case an Error state since there should not be any undefined states in the state machine. If you do not want to use strings for the state machine, the second option is to convert the strings into enumerated-type constants. The method required to perform this action is described in Section 3.2.4. The enumerated constant can then be used to control the state machine. The main diagram is shown in Figure 3.14.

3.6.3 Example Using an Input Array

A second version of the queued-style state machine involves using an array of states to execute instead of the LabVIEW Queue functions. We will use the same example application to illustrate the interchangeability of the methods. The application can use the same user interface. This time, instead of creating an array of strings based
on the user inputs, the array of the enumerated types used in the user interface will be built. This array will then be passed to the main state machine. The programmer should make sure to add the Close State constant to the end of the array to prevent an endless loop. As a backup plan, the user should also make the Close state the default state. This will force the Close state to execute if the array is empty. The VI to build the state array is shown in Figure 3.15.

![Figure 3.15](image)

At the beginning of the While loop, the first state is taken off of the array of states by using the Index Array function. This value is then directly wired to the case structure input. The array is also passed to the end of the While loop. At the end of the While loop, the state that was executed is removed. Using the Array Subset function performs the removal. When the state array is wired to this function, with the index value being set to 1, the first element is removed from the array. This is continued until the Close state is executed, or until the array is empty. The diagram of the main VI is shown in Figure 3.16.

![Figure 3.16](image)
3.7 **DRAWBACKS TO USING STATE MACHINES.**

There are very few drawbacks to state machines, and we will go through those instances here. The first issue we have found with state machines is the difficulty following program flow. Due to the nature of state machines, the order of execution can change due to many factors. The code becomes difficult to debug and trace errors. This is especially true with time-critical applications where execution highlighting is not an option. Documentation is crucial for reading and debugging tests using state machines.

For applications where there are only a couple of tasks that are done sequentially, a state machine can be overkill. Creating an enumerated control for the case statement, setting up Error and Close states, and creating the necessary shift registers can be more work than is necessary. This is the case only in very simple sequences where there will not be major changes or additions. If there is a possibility of expanding the functionality of the VI, a state machine should be used. The benefits and issues of using a state machine should be considered during the architecting stage of an application.

3.8 **RECOMMENDATIONS AND SUGGESTIONS**

As is the case with most programming tasks, there are a number of ways to “skin a cat.” While this is true, there are a number of methods of “skinning a cat” that can make life easier. Using a sharp knife is one of them. Seriously, though, there are a number of ways to make state machines easier to use and modify. The following are tips or suggestions for using state machines.

3.8.1 **DOCUMENTATION.**

The programmer should spend time documenting all code; however, this is especially true when using state machines. Since the order of the execution changes, thorough documentation can help when debugging. An additional reason to document is for when you attempt to reuse the code. If it has been a while since you wrote the VI, it may take some time to figure out how the code executes and why some inputs and outputs are there. Some of the code written in LabVIEW strives to abstract low-level interactions from the higher levels. Good documentation can help ensure that the programmer does not have to go through the low-level code to know what is required for the inputs and outputs. Chapter 4, Application Structure, also discusses some documenting methods available in LabVIEW.

3.8.2 **ENSURE PROPER SETUP**

Since state machines can change the order of execution, special care should be taken to ensure all equipment is in the proper state, all necessary inputs have been wired, all necessary instruments are open, etc. You should try to make every state a stand-alone piece of code. If you are taking measurements from a spectrum analyzer, and the instrument needs to be on a certain screen, you must make sure to set the instrument to that screen. There is no guarantee that previous states have set the
screen unless the order of execution is set. If there is a chance that a prior state will not execute, the necessary precautions must be taken to avoid relying on the prior state to perform the setup.

3.8.3 Error, Open, and Close States

When creating a state machine, there are three states that should always be created. There should be an Error state to handle any errors that occur in the program execution. If you are not using enumerated types or text labels for the states, you should make the Error state the first state. This way, when states are added or removed, the location of the Error state will always remain the same. An additional benefit to making the Error state the first state is when a Match Pattern function is used to select the state to execute. When no match is found a –1 is returned. If the returned value is incremented, the state machine will go to the Zero state. The Error state can be as simple as checking and modifying the error cluster and proceeding to the Close state or to an Error state that can remove certain states and try to recover remaining portions of the execution. The Close state should take care of closing instruments, writing data, and completing execution of the state machine. This is especially important when performing I/O operations. For example, if a serial port is not closed, the program will return an error until the open ports are taken care of. The Open State should handle instrument initialization and provide a single entry point to the state machine.

3.8.4 Status of Shift Registers

Most state machines will have a number of shift registers in order to pass data from one state to another, unless local or global variables are used. National Instruments suggests that local and global variables be used with caution. Depending on the purpose of the state machine, care needs to be taken with regard to the initial values of the shift registers. The first time the state machine runs, any uninitialized shift registers will be empty. The next time the state machine runs, the uninitialized shift registers will contain the value from the previous execution. There are times that this is desirable; however, this can lead to confusing errors that are difficult to track down when the register is expected to be empty. This method of not initializing the shift register is an alternative way to make a global variable. When the VI is called, the last value written to the shift register is the initial value recalled when it is loaded.

As a rule of thumb, global variables should generally be avoided. In state machine programming, it is important to make sure the machine is properly initialized at startup. Initializing shift registers is fairly easy to do, but more importantly, shift register values cannot be changed from other sections of the application. The biggest problem with global variables is their global scope. When working in team development environments, global variables should be more or less forbidden. As we mentioned earlier, a state machine’s internal data should be strictly off-limits to other sections of the application. Allowing other sections of the application to have access to a state machine information can reduce its ability to make intelligent decisions.
3.8.5 **Typecasting an Index to an Enumerated Type**

This was mentioned earlier, but this problem can make it difficult to track errors. When the index is being typecast into an enumerated type, make sure the data types match. When the case structure is referenced by integers, it can be much more difficult to identify which state is which. It is far easier for programmers to identify states with text descriptions than integer numbers. Use type definitions to simplify the task of tracking the names of states. Type definitions allow for programmers to modify the state listing during programming and have the changes occur globally on the state machine.

3.8.6 **Make Sure You Have a Way Out**

In order for the state machine to complete execution, there will need to be a “false” Boolean wired to the conditional terminal of the While loop. The programmer needs to make sure that there is a way for the state machine to exit. It is common to forget to wire out the false in the Close state which leads to strange results. If there is a way to get into an endless loop, it will usually happen. There should also be safeguards in place to ensure any While loops inside the state machine will complete execution. If there is a While loop waiting for a specific response, there should be a way to set a timeout for the While loop. This will ensure that the state machine can be completed in a graceful manner.

It is obvious that the state machine design should include a way to exit the machine, but there should only be one way out, through the Close state. Having any state being able to exit the machine is a poor programming practice. Arbitrary exit points will probably introduce defects into the code because proper shutdown activities may not occur. Quality code takes time and effort to develop. Following strict rules such as allowing only one state to exit the machine helps programmers write quality code by enforcing discipline on code structure design.

3.9 **Problems**

This section gives a set of example applications that can be developed using state machines. The state machines are used to make intelligent decisions based on inputs from users, mathematical calculations, or other programming inputs.

3.9.1 **The Blackjack Example**

To give a fun and practical example of state machines, we will build a VI that simulates the game of Blackjack. Your mission, if you choose to accept it, is to design a VI that will show the dealer’s and player’s hands (for added challenge, only show the dealer’s up card). Allow the player to take a card, stand, or split the cards (if they are a pair). Finally, show the result of the hand. Indicate if the dealer won, the player won, there was a push, or there was a blackjack. Obviously, with an example of this type, there are many possible solutions. We will work through the solution we used to implement this example. The code is included on the CD included with this book.
The first step to our solution was to plan out the application structure. After creating a flowchart of the process, the following states were identified: Initialize, Deal, User Choice, Hit, Split, Dealer Draw, and Result State. The Initialize state is where the totals are set to zero and the cards are shuffled. Additionally, the state sets the display visible attributes for the front panel split pair's controls to “false.” The flowchart is shown in Figure 3.17.

![Flowchart Diagram](image)

**FIGURE 3.17**

The shuffling was performed by the following method. A subVI takes an input array of strings (representations of the cards) and picks a card from the array at random to create a new array. The cards are randomly chosen until all of the cards are in the new array. The VI is shown in Figure 3.18.

The next state to define is the Deal Cards state. This state takes the deck of cards (the array passed through shift registers) and passes the deck to the Deal Card VI. This VI takes the first card off the deck and returns three values. The first is the string value of the card for front panel display. The second output is the card value. The final output is the deck of cards after the card that has been used is
removed from the array. This state deals two cards to the dealer and to the player. The sum of the player's cards is displayed on the front panel. The dealer’s up card value is sent to the front panel; however, the total is not displayed.

The User Choice state is where the player can make the decision to stand, hit, or split. The first step in this state is to evaluate if the user has busted (total over 21) or has blackjack. If the total is blackjack, or the total is over 21 without an ace, the program will go directly to the Result state. If the player has over 21 including an ace, 10 is deducted from the players total to use the ace as a one. There is additional code to deal with a split hand if it is active.

The Split state has a few functions in it. The first thing the state does is make the split displays visible. The next function is to split the hand into two separate
hands. The player can then play the split hand until a bust or stand. At this point, the hand reverts to the original hand.

The Hit state simply calls the Deal Card VI. The card dealt is added to the current total. The state will conclude by returning to the User Choice state. The Dealer Draw state is executed after the player stands on a total. The dealer will draw cards until the total is 17 or greater. The state concludes by going to the Result state. The Result state evaluates the player and dealer totals, assigning a string representing a win, loss, or tie (push). This state exits the state machine. The user must restart the VI to get a new shuffle and deal.

As can be seen by the code diagram of the VI shown in Figure 3.19, the design requirements have been met. There are a number of ways to implement this design; however, this is a “quick and dirty” example that meets the needs. The main lesson that should be learned is that by using a state machine, a fairly intricate application can be developed in a minimal amount of space. In addition, changes to the VI should be fairly easy due to the use of enumerated types and shift registers. The programmer has a lot of flexibility.

3.9.2 The Test Sequencer Example

For this example, there is a list of tests that have been created to perform evaluation on a unit under test. The user wants to be able to select any or all of the tests to run on the product. In addition, the user may want to run the tests multiple times to perform overnight or weekend testing. The goal of this example is to create a test sequencer to meet these requirements.

The first step is to identify the structure of the application that we need to create. For this problem, the queued state machine seems to be the best fit. This will allow a list of tests to be generated and run from an initial user interface. With a basic structure identified, we can create a flowchart to aid in the design of the state machine. The test application will first call a User Interface subVI to obtain the user-selected inputs. These inputs will then be converted into a list (array) of states to execute. For this example each test gets its own state. After each state executes, a decision will need to be made. After a test executes, the state machine will have to identify if an error has occurred. If an error was generated in the state that completed execution, the state machine should branch to an error state; otherwise, the state that executed should be removed from the list. In order to exit the testing, an Exit state will need to be placed at the end of the input list of states. In this Exit state, the code will need to identify if the user selected continuous operation. By “continuous operation” we mean repeating the tests until a user stop. This option requires the ability to reset the list of states and a Stop button to allow the user to gracefully stop the test execution. The flowchart is shown in Figure 3.20.

The first step is to design the user interface. The user interface for this example will incorporate a multiple select listbox. This has a couple benefits. The first benefit is the ability to easily modify the list of tests available. The list of available tests can be passed to the listbox. The multiple select listbox allows the user to select as many or as few tests as necessary. Finally, the array of selected items in string form is available through the Attribute node. The list of tests can then be
used to drive the state machine, or be converted to a list of enumerated constants corresponding to the state machine. In addition to the multiple select listbox, there will need to be a Boolean control on the user interface to allow the user to run the tests continuously, and a Boolean control to complete execution of the subVI. By passing the array of tests into the User Interface VI and passing the array of selected items out, this subVI will be reusable.

The next step is to build the state machine. The first action we usually take is to create the enumerated type definition control. This will allow us to add or remove items in the enumerated control in one location. The next decision that needs to be made is what to do in the event there is no match to an existing state. This could be a result of a state being removed from the state machine, or a mismatch between the string list of tests to execute and the Boolean names for the states. There should be a default case created to account for these situations. The default case could simply
be a “pass-through” state, essentially a Do Nothing state. When dealing with strings, it is important to acknowledge that these types of situations can occur, and program accordingly. The code diagram of the Test Sequencer VI is shown in Figure 3.21.

**FIGURE 3.21**

Once the enumerated control is created, the state machine can be built. After performing an error check, the array of states is passed into a While loop through a shift register. The conditional terminal of the While loop is indirectly wired to a Boolean created on the front panel to stop the state machine. This will allow the program to gracefully stop after the current test completes execution. What we mean by “indirectly” is that the Boolean for the stop button is wired to an AND gate. The other input of the AND gate is a Boolean constant that is wired out of each state in the state machine. This allows the Close state or the Stop button to exit execution. One important item to note on the code diagram is the sequence structure that is around the Stop button. This was placed there to ensure the value of the button was not read until the completion of the current state. If the sequence structure was not used, the value of the Stop button would have been read before the completion of the given state. If the user wanted to stop the state machine, and the user pressed the button, the state machine would finish the current test and perform the next test. Only after reentering the state machine would the “false” be wired to the conditional terminal of the While loop.

Inside the While loop, the Index Array function is used to obtain the first state to execute by wiring a zero to the index input. The output of this function is wired to the case structure selector. This will now allow you to add the cases with the Boolean labels.

The Next_State subVI is the most important piece of code in the state machine. This subVI makes the decision of which state to execute next. The first step in the code diagram is to check the current state in the queue. This is the state that has
just executed. This value is compared to the error state enumerated constant. If this is a match, the state machine proceeds to the Close state to exit execution. This is the method for this application to exit the state machine after an error if no error handling has been performed. After verifying that the Error state was not the last state to execute, the error cluster is checked for errors. Any errors found here would have been created during the test that last executed. If there is an error, the Error state enumerated constant is wired to the output array. This will allow any error handling to be performed instead of directly exiting the state machine. If no error has occurred, the Array Subset function will remove the top state. Wiring a one to the index of the function performs this action. If there are no more states to execute, an empty array is passed to the shift register. The next iteration of the state machine will force the error state (which was made the default state) to execute. The code diagram for the Next-State VI is shown in Figure 3.22.

The first state in this state machine is the Error state. The Error state in this example will perform a couple of functions. The Error state can have code used to perform testing, or clean up functions in the case of an error. This will allow the user to be able to recover from an error if the testing will still be valid. The second function is resetting of the states if continuous sequencing is selected. The first step is to make this case the default case. This will allow this case to execute if the input array is empty or doesn’t match a state in the state machine. If an error occurred, the error cluster will cause the remainder of the state array to be passed to the Next State subVI. If no error occurred, the VI will wire the original array of states to a Build Array function. The other input of this function is an enumerated constant for any state in the state machine except the Error state.

You may be asking yourself why any state would be added to the new queue. The reasoning behind this addition was to allow the sequencer to start back at the beginning. The Error state is only entered when there is an error or when the queue is empty. Since the next state VI uses the Array Subset function to obtain the array of states to be wired to the shift register, the first state in the list is removed. The reason the Error state constant cannot be used is the first check in the Next State subVI. If the Error state is on the top of the array, the subVI will think that an error has occurred and has been dealt with. The VI will then proceed to the Close state.
The remainder of the test sequencer is relatively straightforward. Each state passes the test queue from the input of the state to the output. The error cluster is used by the test VIs and is then wired to the output of the state. Finally, a “True” Boolean constant is wired to the output of each state. This is to allow a “False” Boolean to be wired out of the Close state. The other states have to be wired to close all of the tunnels. Additional functions can be added to the sequencer such as a front panel indicator to show what state is currently executing, an indicator to show the loop number being executed, and even results for each test displayed in an array on the front panel. The sequencer can be modified to meet the needs of the application. The test sequencer is a simple (relatively speaking) way to perform test executive functionality without a lot of overhead.

3.9.3 The PC Calculator Example

The goal is to create a VI to perform as the four-function calculator that comes on most computer desktops. For this example, the higher-level functions will not be added. Only the add, subtract, multiply, and divide functions will be implemented. The idea is to use the classical-style state machine to provide the same functionality.

Again, the first step is to identify the form and function of the application. There needs to be a user interface designed to allow the user to input the appropriate information. For this example, an input section designed to look like the numeric keypad section of a keyboard is designed. In addition to the input section, there needs to be a string indicator to show the inputs and results of the operations. Finally, a Boolean control can be created to allow a graceful stop for the state machine. The state machine is controlled via the simulated numeric keypad.

Boolean controls will be used for the keys on our simulated keypad. The Boolean controls can be arranged in the keypad formation and enclosed in a cluster. The labels on the keys can be implemented by right-clicking on the control and selecting “Show Boolean Text.” The text tool can then be used to change the Boolean text to the key labels. The “True” and “False” values should be changed to the same value. The text labels should be hidden to complete the display. The buttons should be “False” as the default case. Finally, the “Mechanical Action” of the buttons will need to be modified. This can be done by right clicking on the button and selecting the mechanical action selection. There is a possibility of six different types of mechanical actions. The default value for a Boolean control is “Switch when Pressed.” The “Latch when Released” selection should be selected for each of the buttons. This will allow the button to return to the “False” state after the selection has been made. The front panel is shown in Figure 3.23.

After the cluster is created, the cluster order needs to be adjusted. Right-clicking on the border of the cluster and selecting “Cluster Order” can modify the cluster order. When this option is selected, a box is shown over each cluster item. The box is made up of two parts: The left side is the current place in the cluster order; the right side is the original order value. Initially, the values for each item are the same. The mouse pointer appears like a finger. By clicking the finger on a control, the value displayed on the top of the window frame is inserted into the left side of the cluster order box. The controls can be changed in order, or changing the value shown
on the top window frame can change the value of each in any order. When you are finished modifying the cluster, the “OK” button needs to be pressed. If a mistake has been made or the changes need to be discarded, the “X” button will reset the values of the cluster order.

For our example, the numbers from one to nine will be given the cluster order of zero to eight, respectively. The zero is selected as the ninth input, and the period is the tenth input. The Divide, Add, Multiply, Subtract, and Equal keys are given the 11th to the 15th cluster inputs, respectively. Finally, the “Clear” key is given the 16th and final cluster position. The order of the buttons is not important as long as the programmer knows the order of the buttons, since the order is related to the position in the state machine.

The code diagram consists of a simple state machine. There is no code outside of the state machine except for the constants wired to the shift registers. Inside the While loop, the cluster of Boolean values from the control is wired to the Cluster to Array function. This function creates an array of Boolean values in the same order as the controls in the cluster. This is the reason the cluster order is important. The Search 1-D Array function is wired to the output of the Cluster to Array function. A “True” Boolean constant is wired to the element input of the search 1-D array function. This will search the array of Boolean values for the first “True” Boolean. This value indicates which key was pressed.

When the Search 1-D Array function is used, a no match results in a −1 being returned. We can use this ability to our advantage. If we increment the output of the Search 1-D Array function, the “no match” case becomes a zero. The output of the Increment function is wired to the case statement selector. In the zero case, when no match is found, the values in the shift registers can be passed through to the output without any other action being taken. This will result in the state machine continually

FIGURE 3.23
monitoring the input cluster for a keypress, only performing an action when a button is pressed. The code diagram of the state machine is shown in Figure 3.24.

![Figure 3.24](image)

For this state machine, there are four shift registers. The first is used for the display on the front panel. The initial input is an empty string. The resulting value of the display string is sent to the display after the case structure executes. Inside the case structure, the inputs decide how to manipulate the string. There will be more discussion of this function after the remainder of the shift registers are discussed. The second shift register is a floating-point number used to hold the temporary data for the calculations. When one of the operators is pressed, the value in the display is converted to a number and wired to this shift register. At the beginning of execution, after computing the function, or after a clear, the intermediate value shift register is set to 0. When the user presses one of the operators, the third shift register is used to hold the value of the selected operator. After the equal sign is pressed, the operator shift register is cleared. The final shift register is used to hold a Boolean constant. The purpose of this constant is to decide whether to append new inputs to the existing display, or to start a fresh display. For example, when the user inputs a number and presses the plus key, the current number remains in the display until a new button is pushed. When the new button is pushed, the display starts fresh.

The easiest way to make the discussion clearer is to describe the actions performed in the states. As stated earlier, the zero state does no action. This is the state when nothing is pressed. States 1–11 are the inputs for the numbers and decimal point. In these states there is a case statement driven by the value in the final shift register (Boolean constant). If the value is “True,” the value of the input is sent to the display discarding any previous values in the display. If the value is “False,” the input key value is appended to the data already in the display. In each of these cases
a “False” is wired to the shift register since the only time the value needs to be
“True” is when the display needs to be cleared.

In states 12 through 15, the display string is converted to a floating-point
number. This number is wired to the temporary data shift register. The string value
of the display is also wired back to the display shift register. A “True” is wired to
the Boolean shift register to force the next input to clear the display. Finally, the
value of the operator selection is wired to the operator shift register in order to be
used when the Equal button is pressed. Speaking of the Equal button, this is the
16th state. This state has a case structure inside. The case structure selector is wired
to the operator shift register. There are four cases, one for each of the operators.
The display string is converted to a floating-point number, and is wired into the
case structure. The previous input is taken from the shift register and is also wired
to the case structure. Inside each case, the appropriate function is performed on the
inputs with the result being converted to a string and wired to the display output.
The temporary data shift register and the operator shift register are cleared. The final
step in this case is to wire a “True” to the Boolean shift register to clear the display
when a new input is selected. The final state is for the Clear button. This state
clears all of the shift registers to perform a fresh start.

There are only two other components to this example: the Quit button that is
wired to the conditional terminal of the While loop allowing the user to stop the
application without using the LabVIEW Stop button, and a delay. The delay is
needed to free-up processor time. The user would not be able to input values to the
program if there was no delay because the state machine would run continuously.
A delay of a quarter second is all that is necessary to ensure that the application
does not starve out other processes from using the processor.

BIBLIOGRAPHY


*G Programming Reference,* National Instruments