This chapter introduces a robust design pattern called the state machine that can be used to tackle many common LabVIEW coding problems. The state machine translates state transition diagrams into LabVIEW code that is readable, scalable, and maintainable. State machines are robust, flexible, and compact. They provide storage for persistent data and handle errors gracefully and efficiently.

4.1 State Machine Elements

The state machine architecture organizes a VI into states and transitions. The states are the general functions and abilities of the VI—the things the VI can do. Transitions map out the ways that the VI can move from one state to another.

The states are represented by individual cases in a case structure. Each case contains all of the code needed to execute the functions of its state. The case structure is placed within a loop, driving the VI from one state to another. Each state decides which state to move to next. Typically, the states are listed in a type-defined enumerated constant, and this constant is stored in a shift register on the main loop. Each loop iteration, which is the value of the enumerated constant, is read from the shift register and fed to the selector terminal of the case structure to choose the next state. When a state is completed, it places the name of the next state on the shift register. Each state is responsible for telling the state machine where to go to next.

Each state must also decide if the main loop continues or terminates. This is typically done by creating a Boolean output tunnel on the main case structure and wiring it to the continuation terminal of the loop. Each state in the case structure wires a value to this tunnel to either stop the state machine or keep it running.

Let’s take a look at the standard elements of a classic state machine in more detail. A basic schematic of the classic state machine is shown in Figure 4.1. In order to discuss the various elements of the state machine, we have to name them. Unfortunately, there are very few widely accepted terms in the LabVIEW community used to discuss state machine elements. This means that we have the freedom to invent our own names for use in this book. The terms “lobby” and “alley” were coined for this book. The term
“data highway” is a commonly accepted term in the LabVIEW community. The term “guard clause” is well-established in text-based programming, but it has not been widely used in the LabVIEW community.

Here is a glossary defining the various parts of a classic state machine:

**Alley** This is the portion of the state machine between the right edge of the case structure and the right edge of the main loop. The alley is often used for indicators and for logic connected to the stop terminal of the main loop.

**Data Highway** This is the group of wires that run from one end of the main loop to the other. The data highway connects pairs of shift registers and runs through every case in the main case structure. The data highway provides persistent storage in the state machine. This is a commonly used term in the LabVIEW community.

**Error Shift Register** This shift register is usually found near the bottom of the main loop. It passes warnings and errors from one iteration of the loop to the next.

**Execution Timer** This provides timing to regulate the loop iteration rate. It does not provide overall timing for the main application.

**Guard Clause** The guard clause sends the state machine into shutdown if an error is detected or if other user-defined conditions occur, such as hitting a stop button. If there is no reason to shut down the state machine, the next state is passed to the selector of the main case structure.

**Input Area** This area contains inputs to the main loop of the state machine. It is commonly used for initializing shift registers, reinitializing front panel controls, and queue creation.

**Lobby** This is the portion of the state machine between the left edge of the main loop and the left edge of the case structure. The lobby is commonly used for error
checking, for polling a stop button, as a location for inputs that need to be monitored every loop iteration, or for inputs needed by multiple states.

**Main Case Structure** The case structure is the heart of the state machine. The selector of the case structure accepts an input that designates the next state, typically an enumerated constant. It contains the code for each state of the state machine. Each state is represented by a case in the case structure.

**Main Error Wire** The main error wire is the error-handling backbone of the state machine. This wire runs from one end of the main loop to the other. The error wire connects a pair of shift registers, and it runs through both the guard clause and the main case structure. The main error wire should be sent to an error-handling routine after the state machine finishes.

**Main Loop** This is the While loop that drives the state machine. The While loop contains shift registers for the data highway, the state constant, and the main error wire. The conditional terminal is wired to an output of the main case structure. Each case must send either a True or False value to the conditional terminal.

**Output Area** This area contains the outputs of the main loop of the state machine. It is commonly used for closing and destroying references and for error handling.

**State Constant** This is the constant that determines the next state to be executed. It is typically a type definition enumerated constant or a string constant. The state constant is placed on a shift register to pass it on to the next iteration of the main loop.

**State Shift Register** This shift register stores the value of the state constant. It allows each state to determine which state will be executed next.

### 4.2 State Machine Structure

The area to the left of the main loop is the input area of the state machine. Code in the input section that is wired to the main loop executes before the loop starts running. Wires enter the loop through tunnels or shift registers. If a wire enters the loop at a non-indexing tunnel, the value on the wire at the beginning of loop is constant throughout the life of the loop. If a wire terminates at a shift register, the value on the wire initializes the shift register. The input area is a natural place to put code that initializes shift registers, sets front panel objects to known starting states, or creates queues (covered in Chapter 13). It is also a natural place to put items that are constant for the duration of the loop.

The main loop is the engine that drives the state machine from one state to the next. The main loop has shift registers devoted to storing important data known as the data highway. It also has a shift register dedicated to transmitting error information and the value of the next state, usually in the form of an enumerated constant that lists the states of the state machine.

In Figure 4.1, the state shift register is initialized with the Initialize state. This ensures that the Initialize state will be the first state executed. The data highway and error shift registers are also initialized. Depending on the nature of the information
Chapter 4 Classic State Machines

on the data highway, it may or may not be necessary to initialize the data highway shift registers. The error shift register, though, always should be initialized in a top-level VI. Uninitialized shift registers retain their data from one run of the VI to the next as long as LabVIEW remains open on the host computer. This may be desirable in an action engine sub-VI, as we will see in Chapter 5, but it is usually undesirable for a top-level VI. If you do not initialize the error shift register when a top-level VI starts, an error from the last run may be read from the uninitialized shift register and enter the VI during the first loop iteration. Initializing the error shift register prevents this from happening.

The most common way of initializing the error shift register is to use the error wire emanating from start-up code placed in the input area of the loop. This technique uses the dataflow of the error wire to ensure that the start-up code executes before the loop begins to run. Figure 4.1 has no start-up code with error wires, so it uses an Error Ring constant to transmit a clean error cluster to the shift register. The help file for the Error Ring constant is shown in Figure 4.2.

When the main loop iterates, the state and error information are passed to the guard clause. The guard clause checks for errors and shuts down the state machine

---

**Defining a Custom Error Code from the Error Ring**

Creating a custom error code is useful if you want to define a single error code or overwrite a single built-in error code. If you have multiple custom errors you want to use in the Error Ring, use the Error Code Editor dialog box. Complete the following steps to create a custom error from the Error Ring for a one-time use.

1. Add the Error Ring to the block diagram.
2. Click the drop-down arrow \( \rightarrow \) to display the Select Error dialog box.
   - To define a custom error code, enter an error code from the following ranges of error codes specifically reserved for custom error messages: -8999 through -8000, 5000 through 9999, and 50000 through 599,999.
   - To overwrite an existing error code message, enter the existing error code. Creating a custom error message for an already-defined error code is useful when you want to return the error code for a common error case but you want to provide specific information about this particular instance of the error. For example, you can use error code 7, "File not found," and replace that with a description about the specific file that was not found.
5. Enter a description in the Error Description (Optional) field. The description field accepts the same % codes as the Format Into String function. For example, for the custom error message "At time %T, the file %s was not found", the configured Error Ring adjusts to include inputs for a timestamp and a string, respectively, as displayed in the following figure.

6. Use the radio buttons to configure the Error Ring to return either an error or a warning.
7. Remove the checkmark from the Include Call Chain checkbox if you do not want to include the call chain information in the error cluster. If you include the call chain, the error description displays the chain of callers from the current VI to the top-level VI, which is useful when determining where in the VI hierarchy the error occurred. (Real-Time Module) Exclude the call chain to reduce jitter in real-time applications.
8. Click OK.

After you configure the Error Ring, you can adjust the configuration without returning to the dialog box.

- Click the two icons on the Error Ring to toggle the error/warning setting and the include/exclude call chain setting, respectively.
- Right-click the Error Ring and select Generate Error, Generate Warning, Include Call Chain, or Exclude Call Chain from the shortcut menu to affect the relevant configuration.
- Right-click the Error Ring and select Visible Items: Error Explanation Text from the shortcut menu to show or hide the error description.

**Figure 4.2:** Help file for the Error Ring constant
The guard clause in Figure 4.1 checks the status of the main error wire. If the error wire is clear, the guard clause passes the state information to the selector terminal of the main case structure. This can be seen in the No Error case of the guard clause shown in Figure 4.1.

If an error is detected by the guard clause, the Shutdown state is sent to the selector of the main loop. This is illustrated in the Error case of the guard clause shown in Figure 4.3. The guard clause is executed every loop iteration, since it is in the lobby of the state machine. This allows the state machine to detect and respond to errors one loop iteration after the error occurs.

The guard clause is in the lobby of the state machine. The lobby contains code that executes every loop iteration before the main case structure executes. This makes it a good location for code that must be updated frequently to provide inputs for the main case structure. Be careful, though, code in the lobby executes before the main case structure only if dataflow demands it.

The main case structure is the heart of the state machine. Each state is represented by a case in the case structure. The code necessary to execute the state is placed in the case representing that state. Each case of the case structure also determines which state to execute next and places the appropriate value of the state constant on the state shift register.

Each state must also decide whether to continue or terminate the main loop. The Shutdown state, shown in Figure 4.3, wires a Boolean True to the conditional terminal of the main loop using an output tunnel in the main case structure. Every case of the case structure must wire a value to this output tunnel. One way to do this would be to wire a Boolean False constant to this tunnel in every case other than the Shutdown state. A simpler method is to enable the Use Default If Unwired option for the output tunnel, since the default value for a Boolean is False.
Wiring Tip: Be careful with the *Use Default If Unwired* option. The visual cue on the output tunnel is easy to miss and might lead to confusion, degrading readability. Stopping a loop is one of the acceptable uses of this option. When a reader flips through the cases of the case structure, the Boolean True in the *Shutdown* state jumps out because it is the only value explicitly wired to the tunnel. This increases the readability of the code and is a common enough use that the danger of confusion is minimal.

Code in the alley of the state machine executes when the code in the main case structure is finished. The code in the alley is executed every loop iteration, so it is a good location to put indicators that must be frequently updated from data generated by multiple states. Code in the alley of the state machine executes after the main case structure only if dataflow demands it.

With proper dataflow dependence, code in the lobby executes before the main case structure, and code in the alley executes after the main case structure. Code in the lobby often creates input tunnels to the main case structure, while code in the alley often flows from output tunnels from the main case structure. Input tunnels to a case structure can be used or ignored by the individual cases in the structure, but every case must output values to every output tunnel in a case structure.

Sometimes this can lead to difficulties. The *Shutdown* case shown in Figure 4.3 sends a Boolean True to the stop terminal of the of the main loop. Even though there will be no more state transitions, *something* must be wired to the enumerated constant output tunnel or the VI will not run. The *Shutdown* state was chosen because it is the least confusing choice in this situation.

State machines often require loop timing control. One choice for controlling the loop speed is the *Wait Until ms Multiple* function, which was discussed in Chapter 1. This function provides a variable waiting time, ensuring that the loop period is equal to the wired value in milliseconds (if possible). If the duration of the code within the loop exceeds the time target, the period will be set to the nearest multiple of the time target. This function also “puts LabVIEW to sleep” during the waiting period. The CPU is released by LabVIEW to perform other tasks during the wait period.

When the main loop has ended, the code in the output section of the state machine is executed. This is a natural place to put error-handling code. It is also a good place to close references, release resources, and do whatever else is needed to shut down the VI gracefully.

State machines often require storage for important data. This storage is provided by the data highway. The data highway is a wire (or group of wires) that connects pairs of shift registers at the top of the main loop. The shift registers ensure that data on the data highway is stored from one loop iteration to the next. The data highway runs through the main case structure, making it available to every state in the state machine.
4.3 Placing Elements in the State Machine

Once you have the main elements of the state machine in place, how do you decide where to place the controls, indicators, and constants needed by the state machine? Often the best location for a particular item will be obvious, but sometimes there may be more than one reasonable choice. Here are some guidelines to consider while making the decision of where to place controls, indicators, and constants on the block diagram.

Two questions should be considered when deciding where to place an item in a state machine. How many states need to access the item? Does the item need to be stored? The answers to these two questions will help. Some general guidelines are shown in Figure 4.4. The input area of the loop executes before the main loop starts, so it is a good place to put start-up code and initialization data.

The lobby and alley of the state machine are executed every iteration of the main loop and can be accessed by every state in the state machine. The lobby is a good place for controls that change frequently and are needed by multiple states. The alley is a good place for indicators that need to be updated by multiple states. If a control or indicator is only used by a single state, place it within that state.

If data is needed by multiple states and must be stored, consider placing it on the data highway. The data highway runs through the case structure and can be accessed by all states. The shift registers of the data highway provide storage for the data from one loop iteration to the next.

The output area of the state machine only executes after the state machine finishes. This makes it an ideal place for clean-up code, error handling, or controls and indicators that deal with data output by the state machine.

Figure 4.4: Rules of thumb for placement of controls, indicators, constants, and data in a state machine
4.4 Building a State Machine From Scratch: The Multitest VI

Now it’s time to build our first complete state machine. Take a moment to review the specification for the Multitest VI given in Chapter 2 and the state transition diagram shown in Figure 2.2. This diagram will be the blueprint for the Multitest state machine.

Defining the States and the State Enumerated Constant

The first step in building a state machine is to identify the states, all of the possible transitions, and the conditions or triggers that cause the transitions. The states can be read directly from the state transition diagram in Figure 2.2: Initialize, Wait, Run Test 1, Run Test 2, and Shutdown. Using the methods described in Chapter 1, create a type-defined enumerated constant listing the states. It should look similar to the one shown in Figure 4.5. Note that the pull-down menu below the menu bar shows Type Def., indicating that the control is being saved as a type definition.

Figure 4.5: The type definition of the Multitest state enumerated control
**Wiring Tip:** Always save enumerated state controls as type definitions. Enumerated state constants will be used many times throughout a given state machine, and by saving them as type definitions, you help protect your code from changes during development. If states need to be added or deleted, you will only need to change the type definition file, and all of the state constants in the program will be automatically updated.

**Build the Front Panel**

Build the front panel of the VI as shown in Figure 2.1. In the programs in this book, this step is trivial since screenshots of the front panels are provided in the functional specifications. In real-life projects, this step may require significant design work to build a satisfactory and professional user interface.

**Lay Out the Skeleton of the State Machine**

Once the front panel has been built, lay out the skeleton of the state machine. Place the main loop, the main case structure, and the guard clause on the block diagram. Make the main loop large enough to fill most of the screen, but leave enough room at the left and right edges for initialization code and clean-up code. The goal is for the entire state machine to fit on the screen without scrolling.

**Style Guide:** Always strive to have the state machine fit entirely on one computer screen. This makes the code much more readable. If that is not possible, arrange the code so that the user only needs to scroll in one direction: left/right or up/down.

Place the `Wait Until Next ms Multiple` function in the lower portion of the main loop to control the execution timing. Right-click on the `millisecond multiple` input and select `Create... Constant`. Set the constant to 100. At this point, the block diagram should look something like Figure 4.6.

**Create and Initialize the Shift Registers**

The `Multitest` VI will not need a data highway, so no shift registers are required at the top of the loop. There will be a shift register for the state constant located in the middle of the loop, and a shift register for the error cluster located near the bottom of the loop.

Drop a copy of the enumerated state control on the block diagram to the left of the loop. When controls are dropped directly onto the block diagram instead of the front panel, they are automatically converted to constants. Wire the state constant to the left edge of the main loop, creating an input tunnel. Right-click the tunnel, and select `Replace with Shift Register` to convert it to a shift register. Set the state constant to the `Initialize` state.
To create the error shift register, create a *Reinitialize To Default* invoke node for the STOP button as described in Chapter 1. Move the invoke node to the left of the main loop and wire it to the left edge of the loop, creating an input tunnel. Repeat the procedure for converting the tunnel to a shift register. The block diagram should now look like Figure 4.7.

Wiring the *Reinitialize To Default* invoke node of the STOP button to the main-loop error shift register accomplishes two important goals. First, it ensures that the STOP button is set to False before the main loop begins. Second, it initializes the error shift register and overwrites any error information from previous runs of the VI.

**Wire the Guard Clause and the Main Case Structure**

Wire the error shift register on the left edge of the main loop to the selector of the guard clause. The edges of the structure should change color: green for the *No Error* case and red for the *Error* case. Pass the error wire through the guard clause to create an output tunnel, wiring the error output tunnel in both cases. Wire the state shift register to the left edge of the guard clause as well, creating another input tunnel. In the *No Error* case, wire this tunnel directly to the other side of the guard clause, creating an output tunnel. In the *Error* case, right-click on the output tunnel and select *Create . . Constant* to automatically create a state constant. Select the *Shutdown* state. Take the opportunity to document both cases of the guard clause with free labels.
Figure 4.7: Creating and initializing the shift registers for the Multitest VI

**Style Guide:** It is good LabVIEW practice to use a free label in each case of every case structure in a VI.

Now run the state wire from the output of the guard clause to the selector of the main case structure. Right-click on the case menu at the top of the case structure and select *Add Case for Every Value*. This will automatically create a case for every state in the enumerated constant.

The Multitest VI will be polling the STOP, RUN TEST 1, and RUN TEST 2 buttons in the *Wait* state, so place them there for now.

**Build the Shutdown State**

Select the *Shutdown* state and drop a Boolean True constant in the lower-right corner. Wire the constant to the conditional terminal of the main loop. Right-click on the Boolean output tunnel of the case structure and select *Use Default if Unwired*. This will cause the other states to send a Boolean False (the default value of a Boolean) to the loop conditional terminal.

Drop another state constant in the *Shutdown* state and set it to *Shutdown*. Wire it to the edge of the case structure, creating an output tunnel. Then wire the tunnel to the state shift register.

Wire the error line across the main case structure to the output tunnel and then to the right-hand shift register. Drop a *General Error Handler* function outside
Build the \textit{Initialize State}

The \texttt{RUN TEST 1} and \texttt{RUN TEST 2} buttons will be reinitialized in the \textit{Initialize} state. To do this, create \texttt{Reinitialize To Default} invoke nodes and place them in the \textit{Initialize} state. Wire their error input and output terminals to the error tunnels of the main case structure, as shown in Figure 4.9. Wire a state enumerated constant to the state output tunnel and set it to the \textit{Wait} state.

Look at the case selector and check to see if the \textit{Initialize} case is the default case. If it is not, right-click on the selector and select \textit{Make This The Default Case}.

Build the \texttt{Run Test 1} and \texttt{Run Test 2} States

The \texttt{Run Test 1} and \texttt{Run Test 2} states simulate tests by displaying a dialog window with the text \texttt{Run Test 1} and \texttt{Run Test 2}. This is done using the LabVIEW \texttt{One Button Dialog} function (see the help file in Figure 4.10). The completed \texttt{Run Test 1} state is shown in Figure 4.11, and the completed \texttt{Run Test 2} state is shown in Figure 4.12. Both of these states transition back to the \textit{Wait} state when the user closes the dialog window.
Figure 4.9: The Initialize state of the Multitest state machine

One Button Dialog Function

**Owning Palette:** Dialog & User Interface VIs and Functions

**Requires:** Base Package

Displays a dialog box that contains a message and a single button.

- **message**
- **button name** ("OK")
- **true**

Add to the block diagram | Find on the palette

- **message** is the text to display in the dialog box.
- **button name** is the name displayed in the dialog box button. The default is **OK**.
- **true** contains a value of TRUE when you click the button.

Figure 4.10: Help file for the One Button Dialog function

Build the **Wait State**

The **Wait** state is the most awkward and complicated state in the Multitest VI. The function of the **Wait** state is a demanding one: It must poll all front panel objects and perform whatever calculations are necessary to determine the next state transition.
Figure 4.11: The Run Test 1 state of the Multitest state machine, where the test is simulated by a one-button dialog and after the completion of the test the state machine returns to the Wait state.

Figure 4.12: The Run Test 2 state of the Multitest state machine, where the test is simulated by a one-button dialog and after the completion of the test the state machine returns to the Wait state.
Chapter 4  Classic State Machines

The Multitest VI has relatively few front panel objects, so this task isn’t very difficult. But as the number of front panel objects increases, polling all of the objects becomes more unmanageable. Fortunately, LabVIEW provides event structures to perform this task cleanly and easily. This topic will be addressed in Chapter 10.

In the Multitest VI, the front panel objects are all Booleans. This makes monitoring the front panel easier, since Booleans only have two values: True and False. One way to monitor a set of Boolean controls is to use a conditional ladder. A conditional ladder is a series of conditional statements that tests each control in turn. A conditional ladder would read something like “If STOP was pressed choose the Shutdown state, else if the RUN TEST 1 button was pressed choose the Run Test 1 state, else if the RUN TEST 2 button was pressed choose the Run Test 2 state, else go back to the Wait state.”

A conditional ladder can be implemented in LabVIEW using the Select function with the help file shown in Figure 4.13. The Select function accepts three inputs: a Boolean selector input, a True input, and a False input. The data types of the True input and the False input must match. If the Boolean selector input is True, the Select function passes the value of the True input. If the Boolean selector input is False, the Select function passes the value of the False input. The Select function is demonstrated in Figure 4.14. This VI outputs x if x ≥ y and outputs y otherwise.

Select functions can be connected in a cascade to form a conditional ladder, as shown in Figure 4.15. By following the dataflow, you can confirm that this cascade of Select VIs behaves in the same way as the conditional ladder described previously.

**Select Function**

*Owning Palette:* Comparison Functions

*Requires:* Base Package

Returns the value wired to the t input or f input, depending on the value of s. If s is TRUE, this function returns the value wired to t. If s is FALSE, this function returns the value wired to f.

The connector pane displays the default data types for this polymorphic function.

**Example**

![Select VI](select.png)

- t is the value that this function returns if s passes a TRUE value. t and f must be of the same type, but they can have different numeric representations.
- s determines whether the function returns the value of t or f in s? tf. If you wire an error cluster to s and an error occurs, the error cluster passes a TRUE value to the function. Otherwise, the error cluster passes a FALSE value to the function.
- f is the value that this function returns if s passes a FALSE value. t and f must be of the same type, but they can have different numeric representations.
- s? tf is the value wired to t if s is TRUE. s? tf is the value wired to f if s is FALSE.

**Example**

Refer to the Select VI in the labview\examples\general\functions\Comparison directory for an example of using the Select function.

**Figure 4.13:** Help file for the Select function
Figure 4.14: Using the Select function

If \( x \) is greater than or equal to \( y \) output \( x \), else output \( y \)

Figure 4.15: The Wait state of the Multitest state machine, where the Wait state uses a selector ladder to poll the front panel controls and to decide which state to go to next.

If the STOP button is True, the top Select passes the Shutdown state to the shift register. If the STOP button is False, the result of the middle Select is passed on. The middle Select checks the status of the Run Test 1 button. If it has been pressed, the Run Test 1 state is passed on to the shift register. The bottom Select function passes on the Run Test 2 state if the RUN TEST 2 button has been pressed. Finally, the bottom Select passes on the Wait state if none of the buttons have been pressed.

This method is effective, but it becomes awkward for large number of Boolean controls and even more awkward if non-Boolean controls need to be monitored. Another drawback is that all of the front panel objects need to be polled every loop iteration. This is not an efficient use of computer resources.