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 a 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: set up 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 pallete. Each of the above states should be put into the enumerated control. 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 pallete. 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

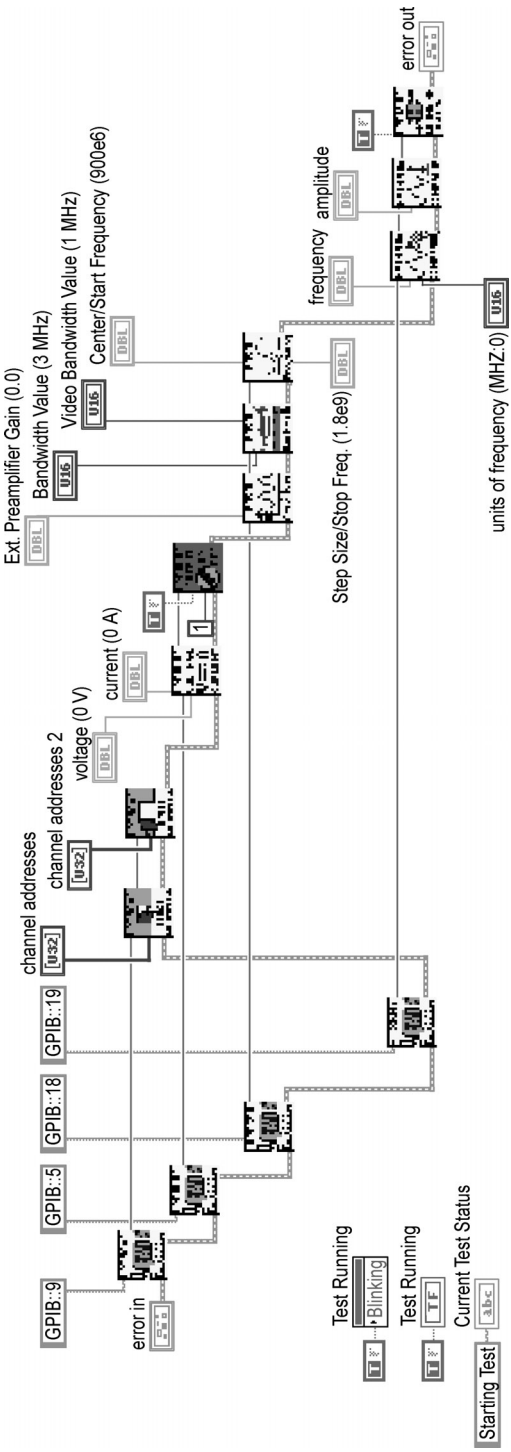
Ext. Preamplifier Gain (0.0)
Bandwidth Value (3 MHz)
Video Bandwidth Value (1 MHz)
Center/Start Frequency (900e6)
frequency
amplitude
error out
channel addresses
channel addresses 2
voltage (0 V)
current (0 A)
Step Size/Stop Freq. (1.8e9)
units of frequency (MHZ:0)
GPIB::9
GPIB::5
GPIB::18
GPIB::19
error in
Test Running
Blinking
Test Running
Current Test Status
Starting Test

FIGURE 3.4

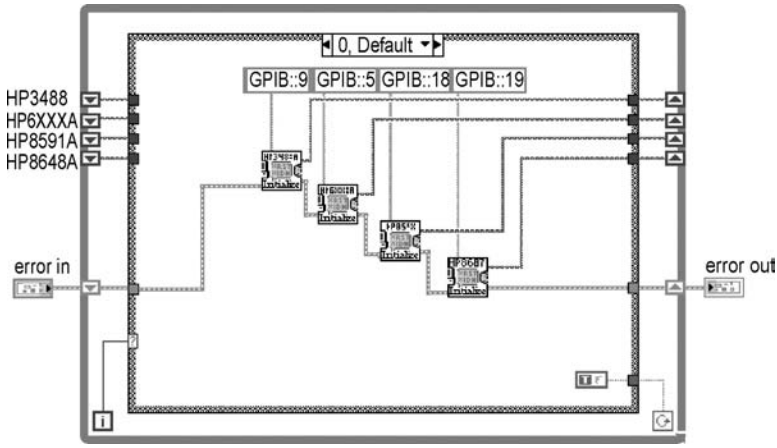

FIGURE 3.5

see what is being done. As 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. Because the While loop will exit only 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,

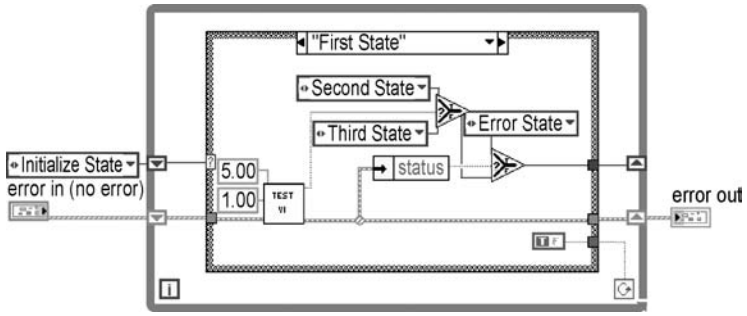

FIGURE 3.6

the next state to execute is decided on. An example of this state machine is shown in Figure 3.6.

3.4.1 The LabVIEW Template Standard State Machine

The template state machine provided by LabVIEW is a test executive styled machine. In order to add one to an application, select "New …" under the file menu. The state machine template is listed in the Design Patterns set as shown in Figure 3.7. The template state machine is shown in Figure 3.8. This stock state machine covers all

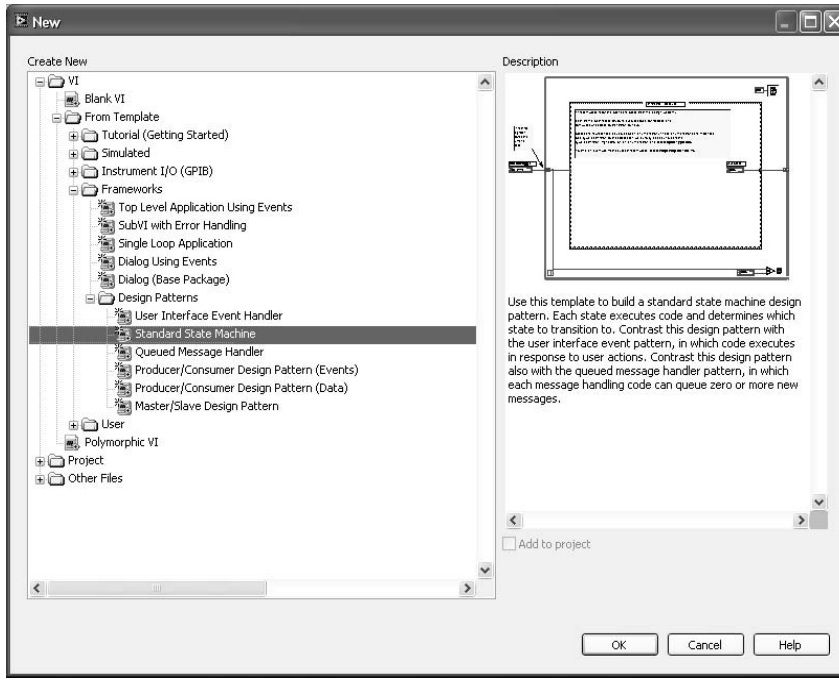

FIGURE 3.7

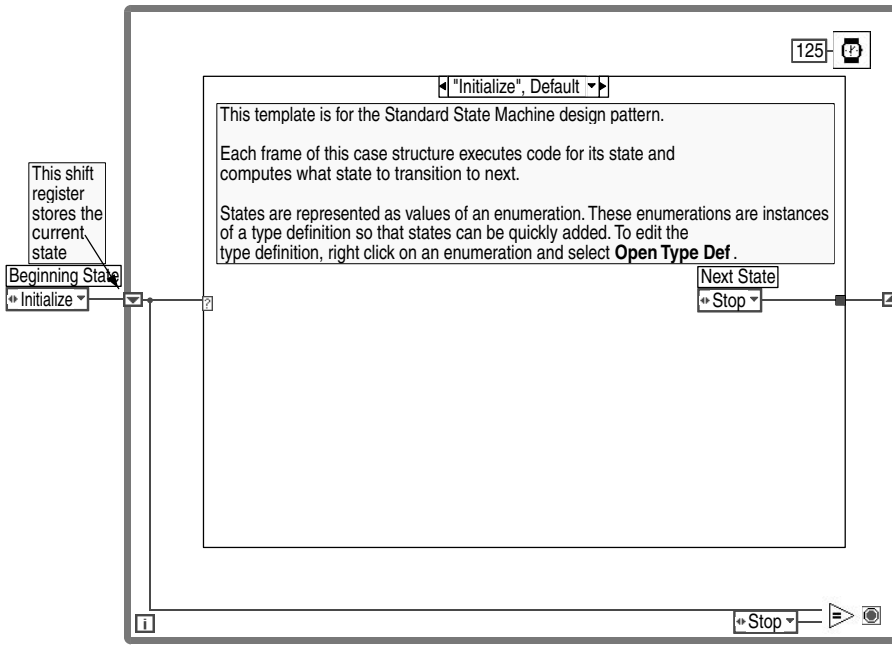

FIGURE 3.8

FIGURE 3.9

the basics but typically requires some modifications before being used. Key features of the template state machine are the use of enumerated types for state definitions, a delay, and defined initialize and stop states.

Right clicking on the enumerated type is shown in Figure 3.9. This shows we were in error; the template state machine isn't based on type definitions, not enumerated types. As stated above, editing the type definition for the state machine will cleanly propagate any changes in the state list through the state machine. The state machine's feedback register is also initialized.

The state machine compares the current state to "stop." If the state machine is in Stop, the While loop will be fed a true and exit, otherwise it will keep going. The next state is determined inside the case structure. It is also possible to eliminate this comparison and feed a constant Boolean true or false from inside the case statement. The template implementation is a bit easier to implement as

opposed to adding and wiring a Boolean to each state. The only disadvantage is there is an additional comparison every iteration. The state is compared to stop, and the Boolean result is compared to true. In general, the performance is not going to be impacted by this.

The 125-ms wait located in the upper right hand corner of the state machine is there to provide some pacing. There are scenarios where delaying action between states is needed, such as processing user interface input. In general, we tend to delete this part of the template machine. If there is a need for particular states to have delays, put the delays inside the relevant states.

The template state machine implements all the features of a test executive styled machine that are needed.

3.4.2 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 considerable 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.

3.4.3 Recommended States for a Test Executive-Style 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 — in the case of the template state machine, only one state in the comparison should end execution of the state machine. 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.

If the error condition is resolvable, the error state can set the error code to 0 and the error indication to false. Sometimes it is advantageous to put the error information into the error string so it can be fed out of the state machine. A good error handler state may make it impossible to tell an error has occurred from outside the state machine.

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. If necessary, the error state can close off any resources related to the error so they do not cause an additional error in the close state. This will allow for the state machine to shut down any resources it can in an orderly manner before stopping execution.

3.4.4 Determining States for Test Executive-Style 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 and 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.

3.4.5 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. Although 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 3.10.

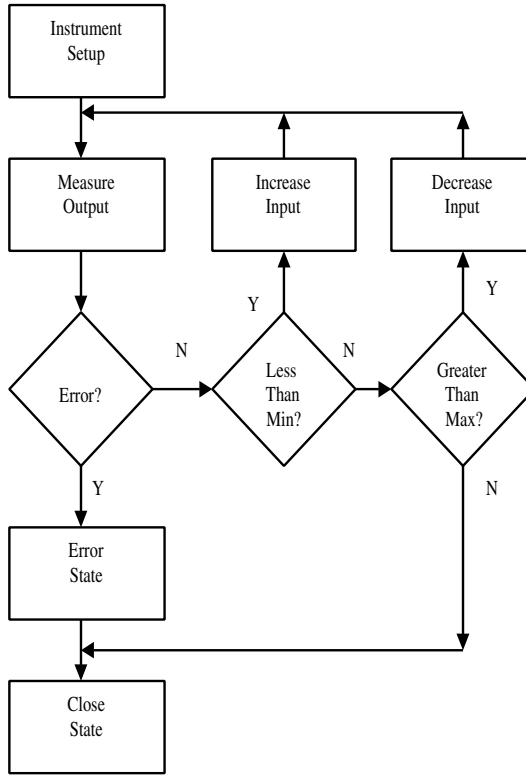

FIGURE 3.10

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.

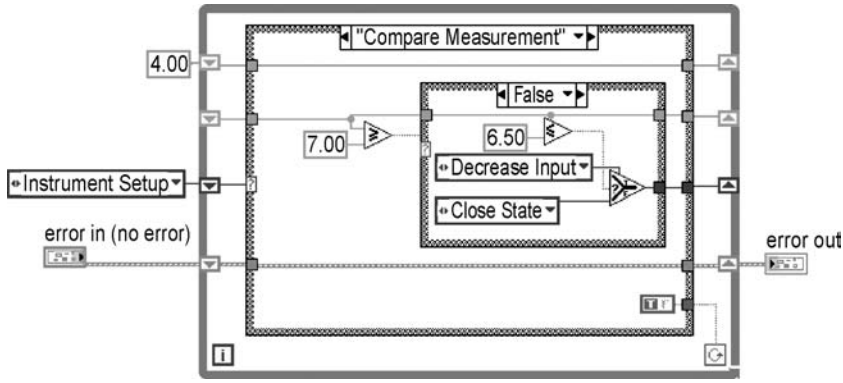

FIGURE 3.11

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. As 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 3.11 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 in most scenarios should be defined with the word "waiting." Using "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. The events are typically driven by external elements to the state machine. The states that are defined for the state machine should not be considered when it comes time to determine what events are going to occur. It will be common for a state to handle one, maybe two events. Anytime an event other than the one or two it was designed to handle arrives it becomes an error condition.

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 .NET event handling are three examples. As 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. Classical style state machines used to be the best solution for user events. LabVIEW 7 introduced the event handling structure. The event structure would appear to easily outperform the classical state machine. The event structure itself is lacking in terms of an error state, but it is easy enough to add one. Add a hidden error cluster to the front panel, you can trigger an event by setting the error cluster's values. The event structure can have a frame dedicated to handling a set value in the error cluster.

As an example, if a user menu selection would make other menu selections not meaningful, the state machine, now an event structure, could be used to determine what menu items need to be disabled, if your file menu had the option for application logging and a selection to determine the level of logging. Typically, application logging is defined in "steps": one step might log everything the application does for debugging purposes, one level might only track state changes, and one will only log critical errors. When the user determines that he did not want the application to generate a log file, then setting the level of logging detail is no longer meaningful. The event structure handling the menu events 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 the 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 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. Because 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 in Figure 3.12.

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.

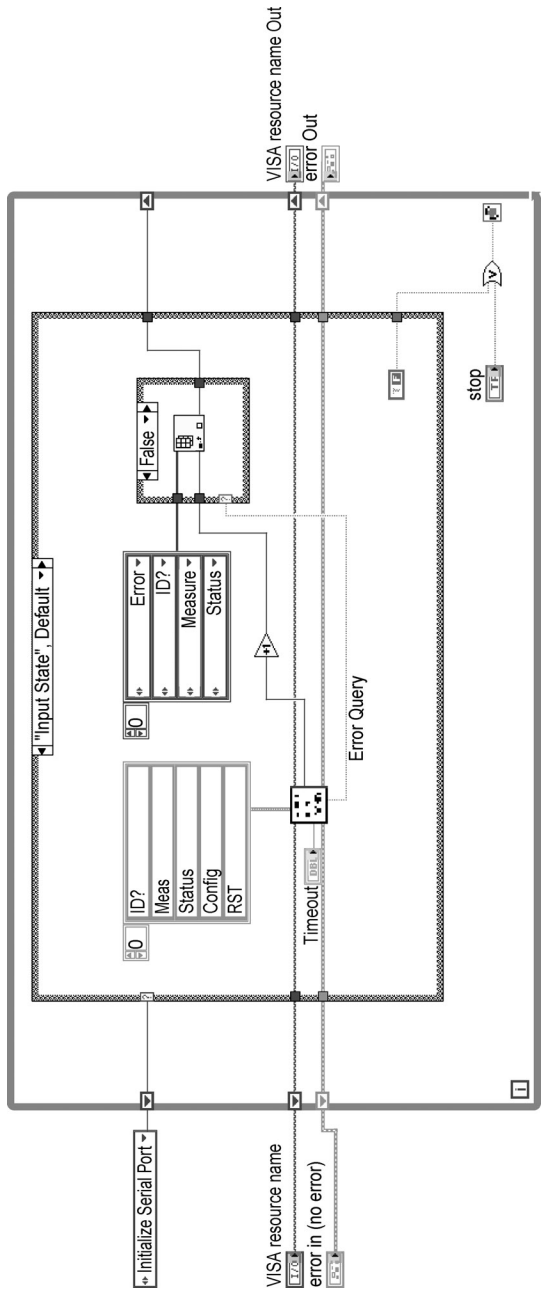
Initialize Serial Port
VISA resource name
error in (no error)
"Input State", Default
ID?
Meas
Status
Config
RST
Timeout
Error Query
False
Error
ID?
Measure
Status
stop
VISA resource name Out
error Out

FIGURE 3.12

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.

Figure 3.13 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. (See Table 3.1.)

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

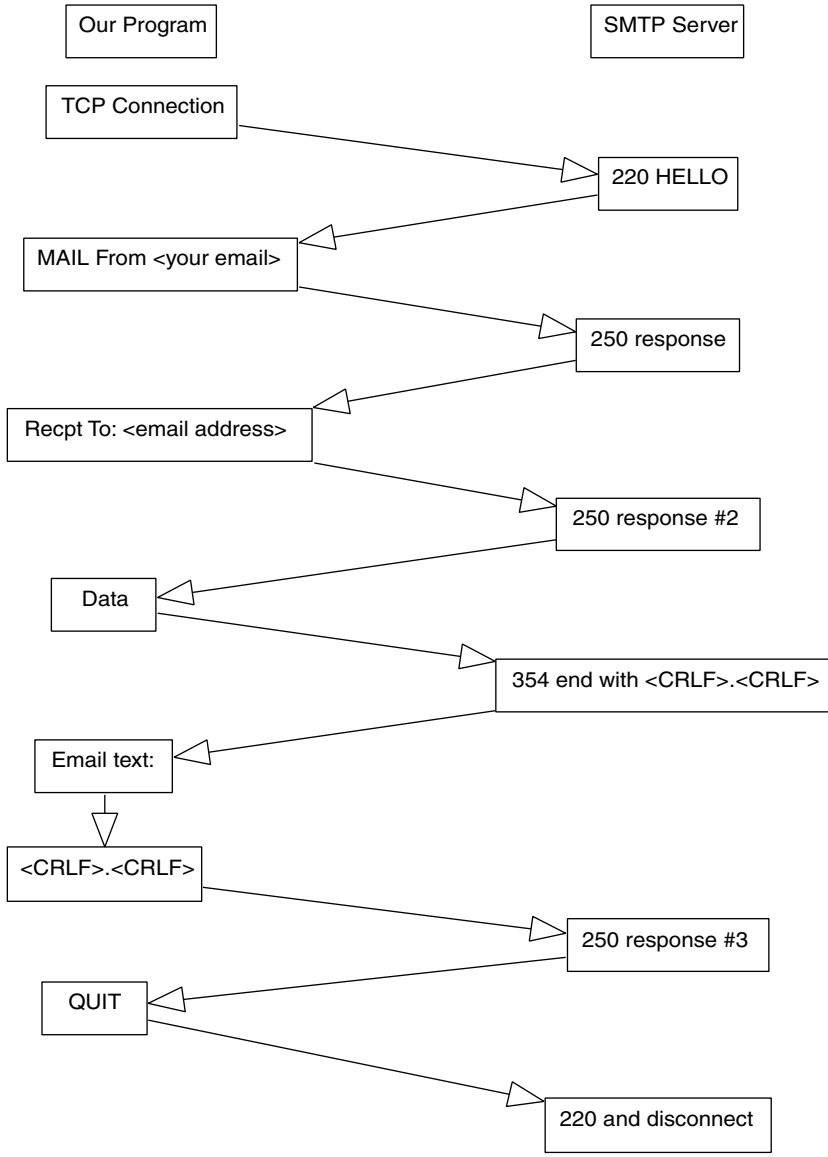
Our Program
SMTP Server
TCP Connection
220 HELLO
MAIL From <your email>
250 response
Recpt To: <email address>
250 response #2
Data
354 end with <CRLF>.<CRLF>
Email text:
<CRLF>.<CRLF>
250 response #3
QUIT
220 and disconnect

FIGURE 3.13

TABLE 3.1
Event Matrix

State/Event	200 Received	250 Received	354 Received	>400 Received	TCP Error
Waiting For Hello	Waiting For Address/ Send from	Waiting For Hello/ Do Nothing	Waiting For Hello/ Do Nothing	Waiting For Hello/ QUIT	Waiting For Hello/ QUIT
Waiting For Address	Waiting For Address/ Do Nothing	Waiting For Data/ Send Recpt	Waiting For Address/ Do Nothing	Waiting For Address/ QUIT	Waiting for Address/ QUIT
Waiting For Data	Waiting For Data/ Do Nothing	Waiting Send Body/ Send Data	Waiting For Data/ Do Nothing	Waiting for Data/ QUIT	Waiting For Data/ QUIT
Waiting To Send Body	Waiting To Send Body/ Do Nothing	Waiting To Send Body/ Do Nothing	Waiting To Quit/ Send Body	Waiting To Send Body/ QUIT	Waiting To Send Body/ QUIT
Waiting To Quit	Waiting To Quit/ Do Nothing	Waiting To Quit/ QUIT	Waiting To Quit/ QUIT	Waiting To Quit/ QUIT	Waiting To Quit/ QUIT.

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 3.13. This will allow us to account for all the possible interactions between our application and the mail server.

Surprisingly, we will need states for only half of the boxes in the right column of Figure 3.13. 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 determine uniquely the next action we need to take. This lets us 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 354 Received, 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 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.

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. As 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 3.14 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 of behavior on a code section.

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

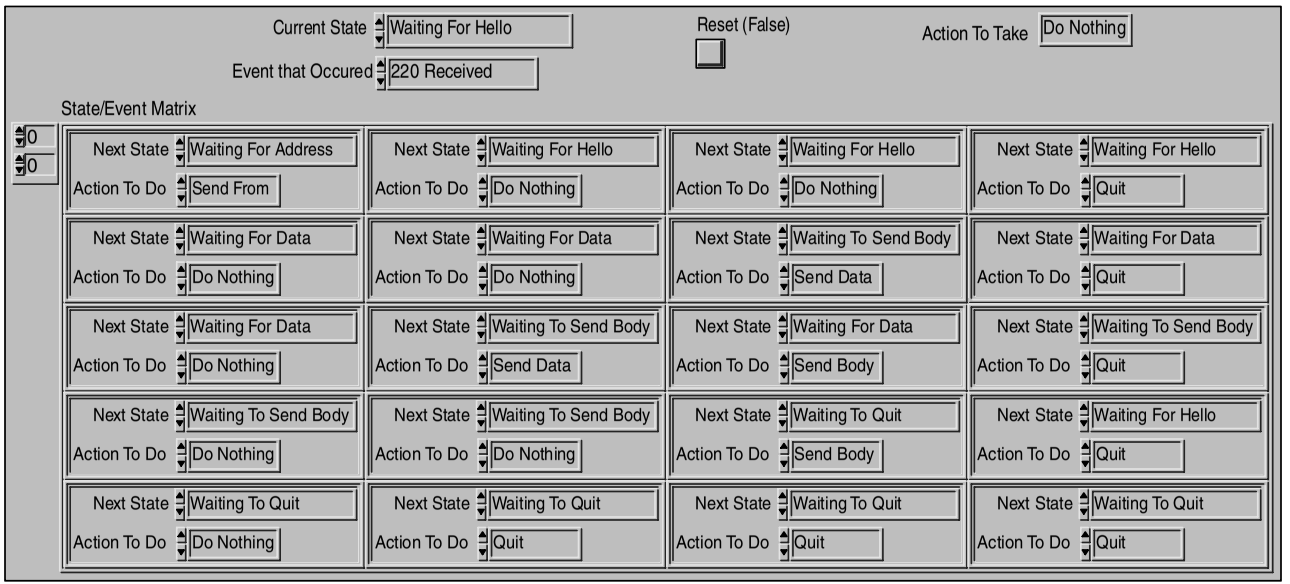
Current State
Waiting For Hello
Reset (False)
Action To Take
Do Nothing
Event that Occured
220 Received
State/Event Matrix
0
0
Next State
Waiting For Address
Action To Do
Send From
Next State
Waiting For Hello
Action To Do
Do Nothing
Next State
Waiting For Hello
Action To Do
Do Nothing
Next State
Waiting For Hello
Action To Do
Quit
Next State
Waiting For Data
Action To Do
Do Nothing
Next State
Waiting For Data
Action To Do
Do Nothing
Next State
Waiting To Send Body
Action To Do
Send Data
Next State
Waiting For Data
Action To Do
Quit
Next State
Waiting For Data
Action To Do
Do Nothing
Next State
Waiting To Send Body
Action To Do
Send Data
Next State
Waiting For Data
Action To Do
Send Body
Next State
Waiting To Send Body
Action To Do
Quit
Next State
Waiting To Send Body
Action To Do
Do Nothing
Next State
Waiting To Send Body
Action To Do
Do Nothing
Next State
Waiting To Quit
Action To Do
Send Body
Next State
Waiting For Hello
Action To Do
Quit
Next State
Waiting To Quit
Action To Do
Do Nothing
Next State
Waiting To Quit
Action To Do
Quit
Next State
Waiting To Quit
Action To Do
Quit
Next State
Waiting To Quit
Action To Do
Quit

FIGURE 3.14

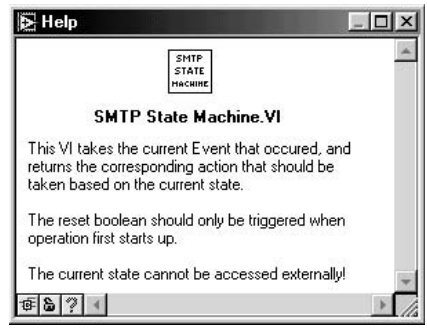

FIGURE 3.15

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 3.15.

The "innards" of the state machine are simple and shown in Figure 3.16. 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

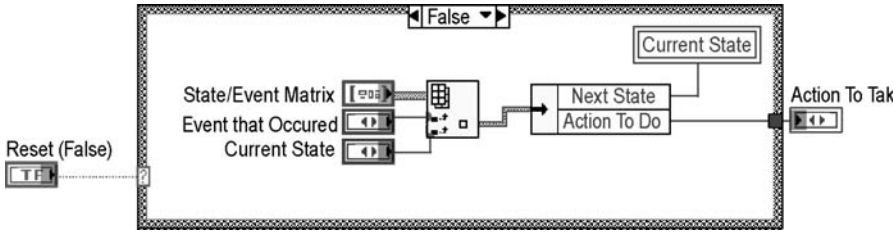

FIGURE 3.16

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 3.11.

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 execute 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

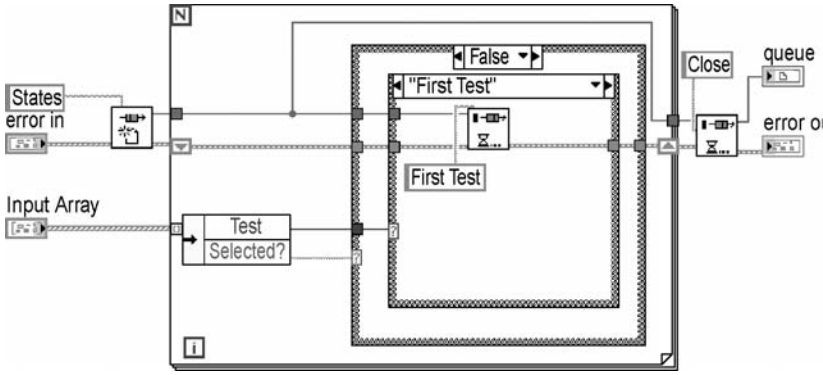

FIGURE 3.17

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 3.17.

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 conditional 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 the While loop is stopped. In the Close state, the programmer should use the Destroy Queue VI to close out the operation.

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. Because 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 as 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 3.18.

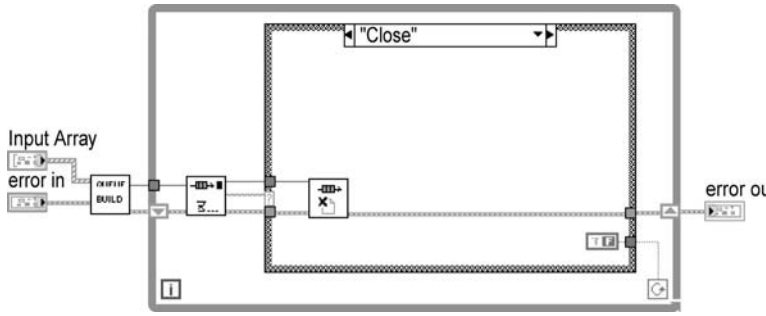

FIGURE 3.18

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 3.19.

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 3.20.

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

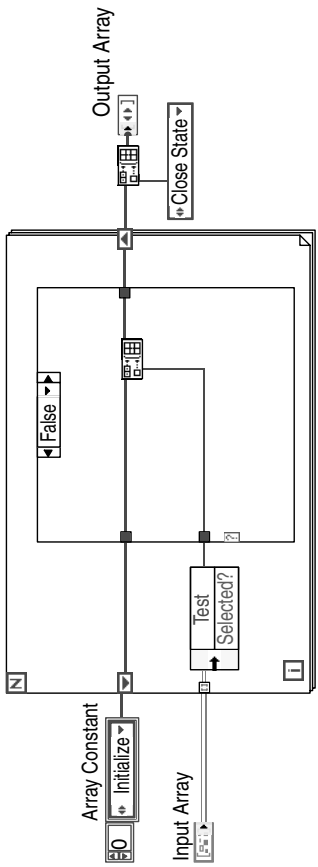

FIGURE 3.19

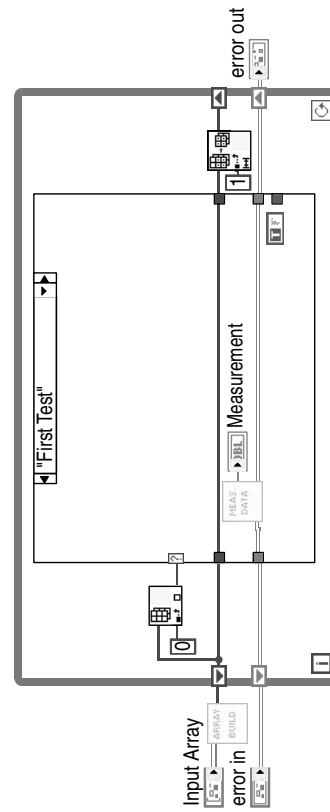

FIGURE 3.20

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 solve a problem. Although this is true, there are design patterns that can make life easier. This section will outline some of the design tools and methodologies that we have found to help implement state machines.

3.8.1 Documentation

The programmer should always spend time documenting all code; however, this is especially true when using state machines. Because 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 also discusses some documenting methods available in LabVIEW.

3.8.2 Ensure Proper Setup

As 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. If the value of the shift register is not known at the time the state machine is started, it should be quantifiable during the open state — that's one of the reasons it's there.

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 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/EXAMPLES

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 3.21.

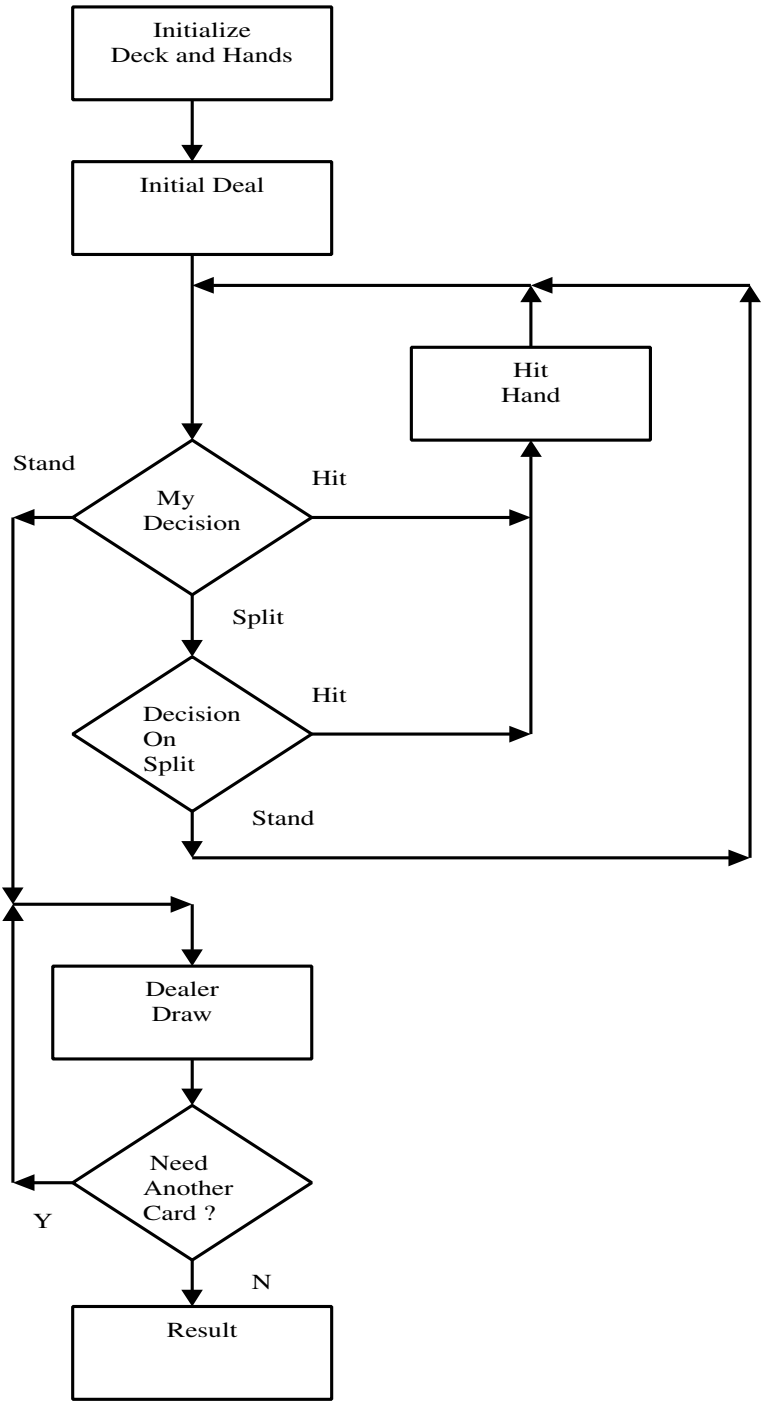
Initialize
Deck and Hands
Initial Deal
Hit
Hand
Stand
My
Decision
Hit
Split
Decision
On
Split
Hit
Stand
Dealer
Draw
Need
Another
Card ?
Y
N
Result

FIGURE 3.21

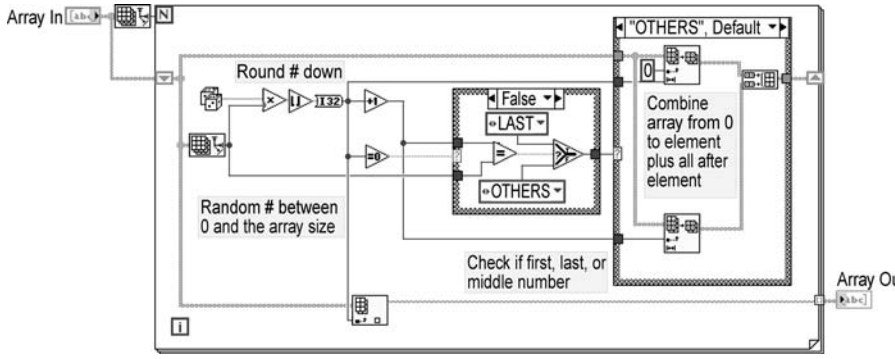

FIGURE 3.22

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 3.22.

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 player's 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 3.23, 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

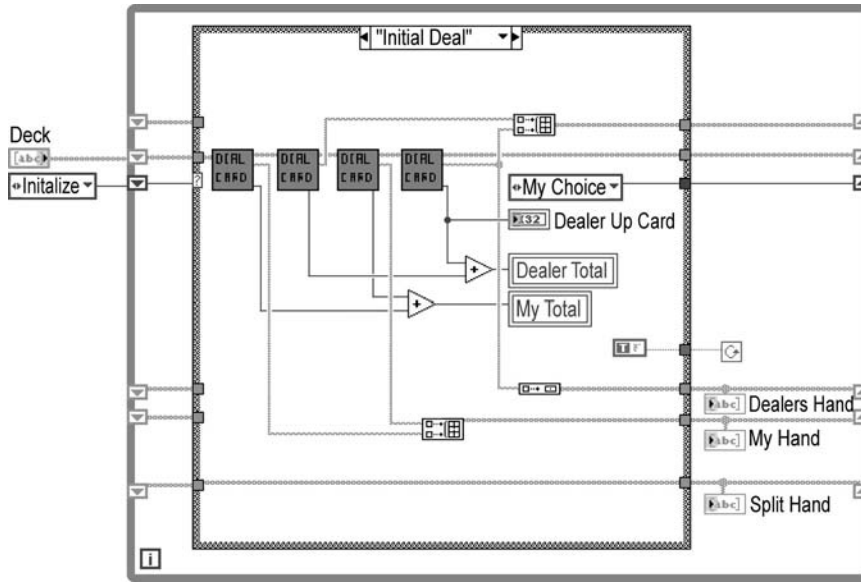

FIGURE 3.23

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

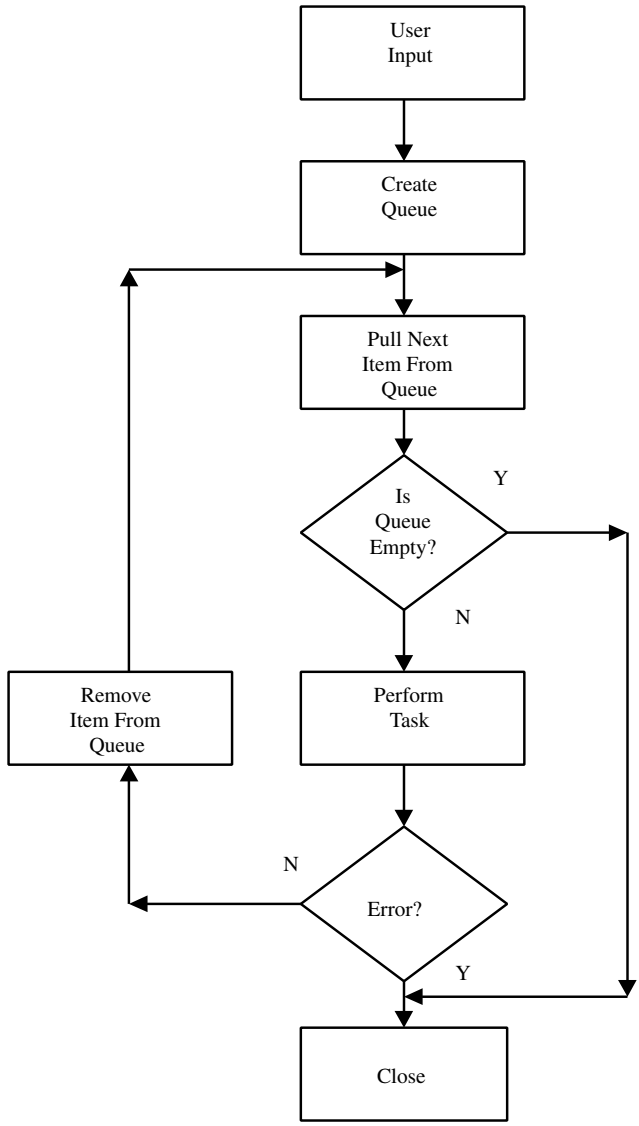

FIGURE 3.24

operation" we mean repeating the tests until a user stops. 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 3.24.

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 3.25.

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

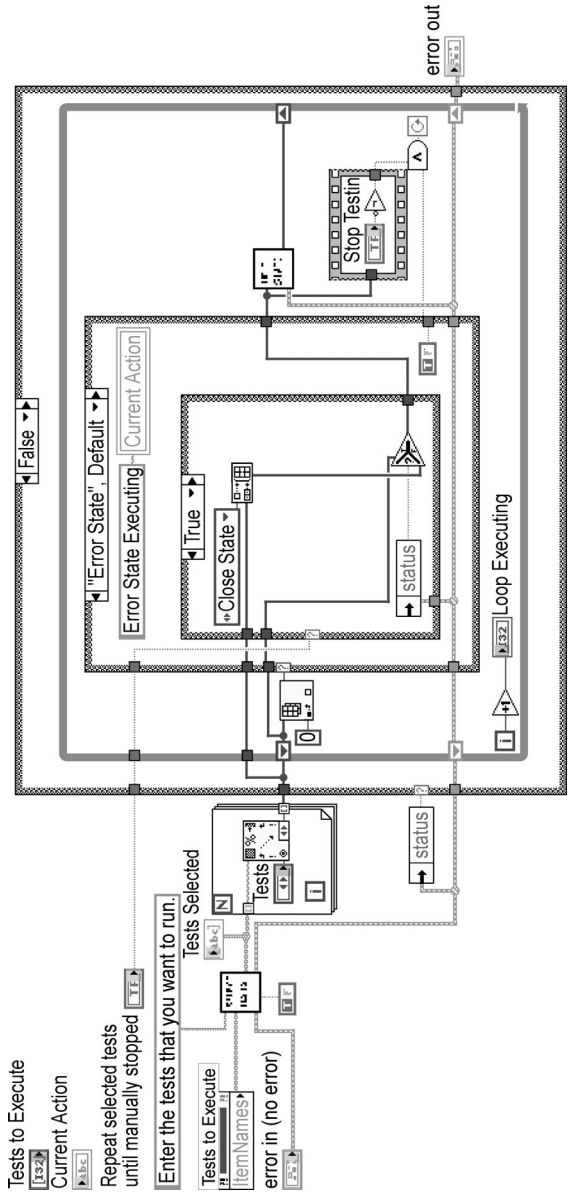
Tests to Execute
Current Action
Repeat selected tests until manually stopped
Enter the tests that you want to run.
Tests Selected
Tests to Execute
itemNames
error in (no error)
Tests
status
False
"Error State", Default
Error State Executing
Current Action
True
Close State
status
Stop Testin
Loop Executing
error out

FIGURE 3.25

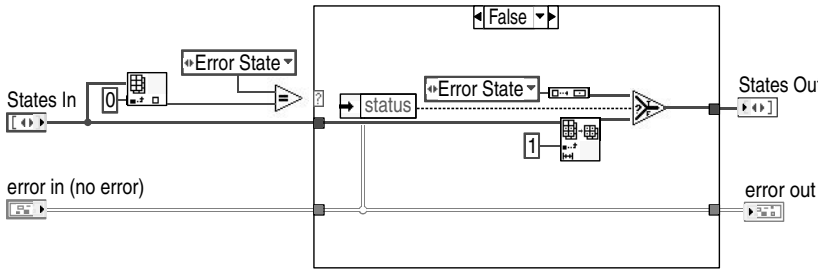

FIGURE 3.26

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 3.26.

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. Because 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 3.27.

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

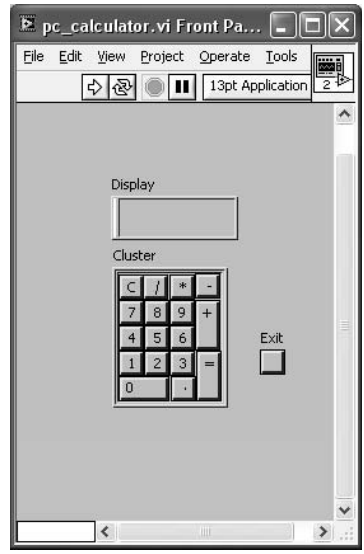

FIGURE 3.27

16th and final cluster position. The order of the buttons is not important as long as the programmer knows the order of the buttons, as 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 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 3.28.

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 tem-

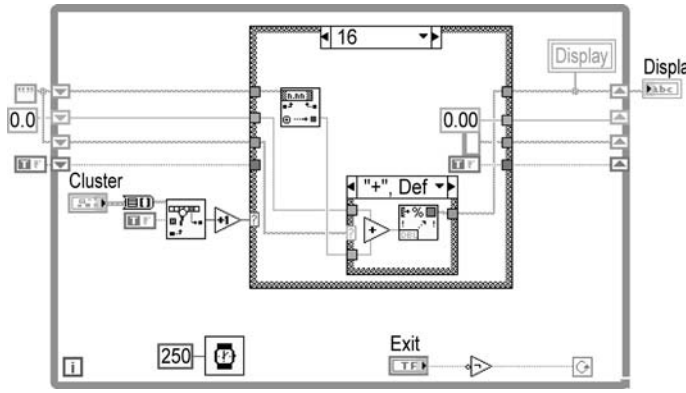

FIGURE 3.28

porary 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 because 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

LabVIEW Graphical Programming. Gary W. Johnson, McGraw-Hill, New York, 1997.

G Programming Reference, National Instruments, Austin, TX, 1999.

LabVIEW with Style — A Guide to Better LabVIEW Applications for Experienced LabVIEW Users. Gary W. Johnson and Meg F. Kay, Formatted for CDROM included with *LabVIEW Graphical Programming,* second ed., Austin, TX, January 12, 1997.

4 Application Structure

This chapter provides insight into developing well-structured applications, and will be particularly helpful for those applications that are relatively large. Several topics will be discussed that are important to the success of a software project. First, the various issues that must be considered before development can begin will be looked at. Then, the role of structure, or framework, of applications and its importance will be explained. The sections that follow will elaborate on software models, project administration, and the significance of documentation.

The three-tiered approach will then be presented as a framework for well-structured applications, stressing the importance of strict partitioning of levels. This topic will include the main, test, and driver levels of an application. Some of the features discussed in the book to this point have involved the LabVIEW Project. We will now take a look at the project and some of its features. The chapter will conclude with a summary example.

4.1 PLANNING

Complex architectures are not needed when the application being developed is simple. It is relatively easy to throw together a program in LabVIEW for performing specific functions on a small scale. But when the application becomes large in size, several design considerations should be taken into account before coding can begin. The following issues, among others, need to be considered: flexibility, extensibility, maintainability, code reuse, and readability.

Flexibility and extensibility impact the ability of an application to adapt to future needs. The ability to add functionality after the application has been released should be designed into the code. It is almost inevitable that requirements will change after the program is released. The architecture of large applications needs to be designed with the ability to make additions. For example, the end user may demand additional functionality to meet new requirements. If the application was not designed with the capacity to evolve, incremental enhancements can prove to be very difficult. The needs of the user evolve over time, and a well-designed application can easily adapt.

Maintainability of code is necessary for applications so that needed modifications can be made easily. The concept of allowing for change in functionality holds true for the ability to maintain and modify code easily. For example, if a power supply that is being used in the current test setup will not be used in another test rack, you may need to change to a different model. How easily your code can be modified to

reflect this change in the test setup is material. The amount of work involved in the alteration depends on how the code is structured.

Code reuse is required for cycle-time reduction on future projects. This attribute is often overlooked because programmers focus on accomplishing the goal of the current project. The time it takes to complete future projects can be reduced if even small pieces of the code can be reused. When software is written in a way that it cannot be reused, efforts are duplicated unnecessarily. Both time and money can be saved when a project is developed with reuse as a design goal.

The ability of the software to provide abstraction is also significant because it improves code readability. Not everyone interacting with the program needs the same level of abstraction. Someone who will use the application, but knows nothing about programming, does not need or wish to see the low-level data manipulation of the program. Operators want an easy user interface that will allow them to use the application for their specific purpose. On the other hand, the person in charge of writing and maintaining the application needs to see all levels of the program. Abstraction allows the programmer to conceal subsections of the application from those who would not benefit from seeing it. Drivers abstract the I/O so it is easier to understand the test level. The test level abstracts the logic so the main level is easier to read.

The concepts presented in this chapter are a good starting point for beginning a project. "Plans are nothing; planning is everything," is a quote by Dwight D. Eisenhower that is applicable to software design. Without adequate planning, large applications are not likely to be successful. Planning provides a roadmap for development and helps minimize the occurrence of unexpected events. You can plan with contingencies depending on the results of the design stages.

Inadequate planning is more likely to result in problems. When designing an application, detailed knowledge of the system — instruments, software requirements, feature sets, etc. — plays a significant role in building a successful application.

4.2 PURPOSE OF STRUCTURE

The topics discussed on application structure may be applied to programming languages other than LabVIEW. Architecture and process are two elements that are important in all languages. The structure of the program or framework that is used is important for future additions, modifications, and maintenance. If the correct process is taken in designing the software system, the application can change as the needs of the user change. These things should be taken into account in the early stages of the development process. Systematically approaching the development of an application means deciding on a process.

The importance of heuristics as discussed by Rechtin and Maier should also be considered. Several rules of thumb that guide the development process will be pointed out as the three-tiered approach is described. These are suggestions and ideas that have been learned through experience.

As is the case in any programming language, the programmer must take the time to understand the nature of the task at hand and what the purpose of the project is. By this, we mean the project requirements should be well defined. There should

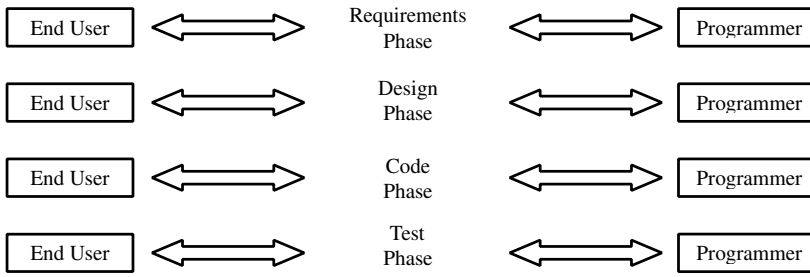

DIAGRAM 4.1

be a clear goal or result for the project. Defining the requirements is one of the first stages in the development of any application. Some people believe that a big portion of the project is completed once the detailed requirements are defined and documented. An example is writing an application to monitor process control. The user requirements might include the number of items to monitor as well as the frequency of the sampling. You must decide what instruments will be used, the type of communications (serial, GPIB, etc.), what specific measurements are needed, where the data will be sent, and who the end users of this application are. These are considered derived requirements. Together, these are some of the more general items that would be included in the enumerated list of requirements.

The requirements are key deliverables for a software project. One of the most common reasons software projects run over budget and beyond due dates involves not having the requirements defined. There is a tendency to add new features into the application late in the development cycle. If the requirements keep changing, it will be difficult to adhere to limited schedules and budgets. You must lock in the requirements to prevent "feature creep."

When the requirements are very loosely defined or not defined at all, the end result has many possibilities. Failing to document the needs of the customer can prove to be very costly. A lot of time and money will be wasted in making changes to fit customer needs. It can be avoided if time is spent earlier in the process. If requirements are not in writing, contract payments may not be made.

The end user of the program plays a key role in development. If the end user of the program is the person writing the code, the requirements do not have to be defined as well because that person will know what is needed. When the code is being written for someone else, they must be consulted at several stages in the process in addition to the early requirements phase. The saying, "You never really understand a person until you consider things from his point of view," holds true here. (See Diagram 4.1 on user involvement.)

4.3 SOFTWARE MODELS

There are many software models that exist, but only the waterfall model and the spiral model will be described in this section. These are two common software models that are widely used in the development process of a software project. They

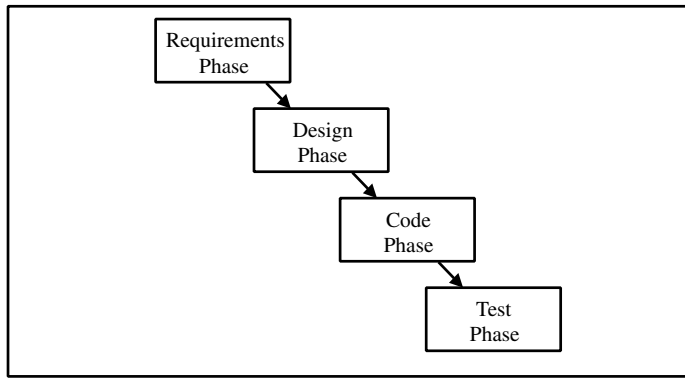

DIAGRAM 4.2

are called "lifecycle models" because they can be applied from the beginning to the end of the application's existence. Both models have several variations, but only the basic models will be presented.

4.3.1 The Waterfall Model

The waterfall model has been widely used for some time. It is a classic model that consists of distinct phases in the project. In its simplest form, the waterfall model contains the following phases: requirements, design, coding, and testing. The waterfall model is depicted in Diagram 4.2. The modified versions that are in use sometimes include more than one step in each phase.

Documentation plays an important role in the waterfall model. In its purest form, the focus is kept on one phase at a time. Each previous phase is thoroughly documented and approved before proceeding to the next step.

In the requirement phase, the capability and functionality needs are defined in detail. There are many requirements that have to be outlined, such as hardware requirements, software requirements, backward and forward compatibility requirements, etc. There are also user requirements and requirements that are derived.

The design phase consists of deciding on the structure of the application in detail. This is the stage where you decide how you will implement the requirements. It includes developing the design or architecture of the application at a high level, and performing the description of logic to accomplish the objective. This chapter focuses mainly on the design phase of the project.

The coding phase includes the actual implementation and software development. As the name suggests, the actual programming is done in this step. The plans made in the design phase are stepping stones for programming. When working in a team environment where several people are involved, good coordination is necessary. The program can be separated into modules and integrated later.

The testing phase attempts to find and fix all the bugs in the code, and includes integration. The point of this phase is to determine if the specifications and requirements were met as outlined in the earlier stages. The importance of testing is not always emphasized as much as it should be. No matter how much time is spent on

testing, finding all of the faults is very difficult. Some faults are hard to find and may eventually slip through the cracks. Generally, test plans will be developed to verify and validate the code's conformance to requirements.

The waterfall model heavily stresses the importance of the requirements phase to eliminate or reduce problems early. The requirements must be explicitly outlined in this model before work can begin on the next phase. Keep in mind that defining detailed requirements will not always translate into a good application structure. However, it does bring to attention the important phases that are involved in application development. This model is aimed at getting the project completed in one pass. Returning to a previous phase to make changes using this model can become costly because one phase is to be completed before the next phase begins.

4.3.2 The Spiral Model

The second model is the spiral model, which is essentially an iterative development process. In this model, software is developed in cycles that include the phases described previously. Each iteration either fixes something from the previous one, or adds new features or functionality to the application. The importance that is stressed in this model is that the significant issues are discovered and fixed early in the development process. A goal for a deliverable can be defined for each iteration of the project. The spiral model is depicted in Diagram 4.3.

Each release or version includes going through planning, evaluation of risks, design and code, and software assessment. Planning consists of establishing the goals, constraints, and alternatives that are available. The potential issues and alternatives are analyzed in the second stage. The design and coding stage involves implementation of the design where the application is actually developed and tested. Finally, the application is evaluated during software assessment.

The evaluation of risks related to the project is crucial in the spiral model. You start with the most important risk and continue through one development cycle, working to eliminate that risk. The next cycle begins with the next important issue.

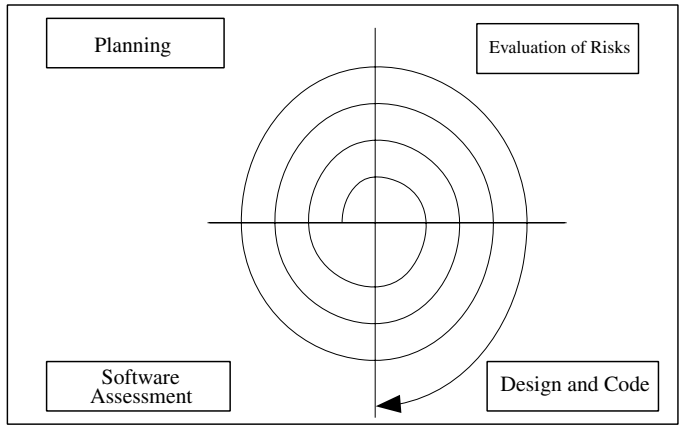

DIAGRAM 4.3

Iterations continue until all issues have been resolved, or the requirements for the finished project have been fulfilled.

The spiral model is based on the concept of incremental development during each iteration. The highest priority items, consisting of risks or features, are addressed and implemented first. Then, the project is reevaluated, and the highest priority item is defined and implemented in the next iteration. The first release of an application can consist of the first loop, and following versions add new features as more iterations are made.

The spiral model is best when the requirements are not fully defined and development work must begin quickly. Iterative models create an early version of the application for demonstration purposes and further refinement. If the risks cannot be identified easily, this model may not work very well.

4.3.3 Block Diagrams

Using a particular model will not guarantee success, but nevertheless following a model provides an orderly roadmap for the development process. Following one model strictly may not be the best course. There should be sufficient flexibility to take the specific circumstances into consideration. Sometimes it is enough to keep in mind that there are different phases in the models and adapt them to the current project.

In either model, block diagrams are useful tools that can assist in the design of the application. When using a top-down design approach, the block diagram, or hierarchy, helps get the structure defined. It also assists in separating tasks for the project and developing timelines for completion. In this way the developer can get the big picture of how the application is laid out and how to progress in the coding phase. The top-down design is more suitable for large applications where work can begin on the user interface and main level first. Work can then continue down to the driver level. In the bottom-up design, the drivers that will be needed for the program are worked on first. At most, a small team should be responsible for the architecture of the project. This facilitates management and prevents pulling the project in various directions.

4.3.4 Description of Logic

A large application must be designed before the coding phase can begin. The requirements determine what the application must do. The design or architecture determines how the application will actually do it. The design phase of a project includes developing the architecture as well as description of logic for the implementation.

The architecture consists of the framework that will be used for the application, taking code reuse, readability, and flexibility into consideration. This can be done by creating flow diagrams or designing the VI hierarchy for LabVIEW applications. The software architecting is followed by definition of the logic that will be used to perform the actual coding.

In the description of logic, the designer must describe the logic of each VI in the hierarchy. The description of logic should capture the intention of the VIs and provide documentation at the same time. This will prevent rework and reduce the

time it takes to develop the application. A description should include both the purposes of the VI and how it will accomplish its objective. The purpose should simply be one sentence describing what the VI will do. When several sentences are needed to describe the action of the VI, the code can possibly be broken down into subVIs. This will increase readability and maintainability of the code. For example, "This VI will configure the signal generator for test A," illustrates the intent of the VI. When describing how the objective will be accomplished, include what VIs will be called and why they will be called.

The coding phase will utilize the description of logic for development of the application. This is followed by the test phase where the code and the description of logic are validated. A test plan must be used to verify the description of logic. This is done by developing test cases designed to test different portions of the description of logic.

4.4 PROJECT ADMINISTRATION

A single programmer project can have its process managed by the programmer. Management of the development process may not be a big problem in this situation. However, projects are often worked on by teams composed of several members. When more than one person works on an assignment, the need for a team leader arises. Someone has to drive the project and control the direction of its progress. The whole development process needs management. Decisions must be made on how and whether to continue to the next phase.

One team member must assume the role of project manager. Without someone to focus on the overall goal of the project, the goals of the members can become divergent. When a team works on all phases of the application, the team leader becomes the lead designer and the one who ensures that the process is moving in the right direction. When separate teams are working on each phase of the application, a project manager is needed to work with all the teams involved. The project manager has to make sure that the designers, programmers, and testers work together and are synchronized. Clear roles have to be assigned to the individual members.

Projects have constraints and risks regarding cost, schedule, and performance. The project manager has to practice techniques to control the risks and constraints. Scheduling is a crucial aspect of the administration of a project. Some stages have to be finished before work can begin on the next stage. Goals and milestones have to be achieved in a timely manner. The project manager works with all who are involved and is made aware of problems as they arise. Resources can be shifted where necessary to assist in problem resolution and to meet schedules. Deadlines are strategic issues that must be dealt with in the appropriate manner. In some cases it might be preferable to be over budget and on time than to be late but within budget constraints. In other cases it is better to be late with a high-quality and high-reliability product.

The administrator should have a good understanding of the complete system. If the project manager is involved in the early conception and requirements stages, then this person will have a better grasp of the purpose of the application. Better

decisions can be made on the priorities of the task at hand and how to resolve conflicts. Information must be acquired, evaluated, interpreted, and communicated to the group members as necessary.

Software projects are more difficult to manage than other types of projects for several reasons. It is difficult to make estimates on the project size, schedules, scope, and resources needed. Software projects can fail due to inaccurate estimates on any of these aspects. Planning plays a key role in project management.

4.5 DOCUMENTATION

When a software application is being developed, the proper documentation is often overlooked. Many times, the documentation process will begin only after all the coding has been completed. This results in insufficient reports on the procedures followed and the actual code written. When you return to write documentation after completing the project, you tend to leave out design decisions that are important to the development. Then, the record keeping becomes more of a chore and fails to serve its intended purpose.

Good documentation will allow someone not involved in the development to quickly identify and understand the components of the project. It also provides a good design history for reference purposes on future software projects. Accounts should be kept at all of the phases in the development cycle. The requirements documents are significant because they will guide the rest of the phases. The design phase documentation serves as a reference for the coding phase.

Documentation during the coding phase, or Description of Logic, is critical. Major points help understand what the code is supposed to do. Comments that are included with the code help identify the different segments and the purpose of each segment. They aid in the maintenance, modification, and testing of the code. Updating the code becomes easier for someone who was not involved in the development process. The original programmers may be reallocated, transferred, or may even leave the company. Then, problems can arise for those who use the program and have to make modifications.

4.5.1 LabVIEW Documentation

LabVIEW has some built-in functions to help in the documentation of code. As with other programming languages, comments can be included in the appropriate places with the code. This allows anyone to look at the diagram and get a better understanding of the code. When modifications have to be made, the comments can help identify the different areas in addition to their functionality.

LabVIEW allows the programmer to enter descriptions for front panel controls and indicators. When Show Help has been activated from the Help menu, simply place the cursor over the control or indicator to display its description. To enter the description, pop up on the control and select Data Operations from the menu. Then select Description and a window appears that will allow you to type in the relevant information. These descriptions will assist anyone who is using the application to identify the purpose of the front panel controls and indicators

Descriptions can also be added for each VI that is developed, using the Show VI Info selection under the Windows menu. You can include relevant details of the VI, inputs, and the outputs. When the Show Context Help is activated from the Help menu, this VI information will appear in the Help window if you place the cursor over the icon. Help files can also be created and linked to LabVIEW in an on-line form. They have to be created in Windows format and compiled before they can be used in LabVIEW.

4.5.2 Printing LabVIEW Documentation

You can also select Print Documentation from the File menu and LabVIEW will allow you to customize the way you want to print the documentation. The VI Info that was entered will be included. There is a feature that gives you the ability to print documentation to an HTML file. This file can then be published easily on the Web. Options for saving files in RTF format or as plain text files are also available. The user can select this from the Destination drop-down menu in the window after Print Documentation has been selected. The pictures of the code diagram and front panels can be saved as PNG, JPEG or GIF formats.

4.5.3 VI History

Another way to document LabVIEW applications is to use the Show History selection under the Windows menu. This will allow the programmer to write what changes are made each time the VI is modified. The VI history provides a good reference when trying to track down what changes were made, why they were made, and when they were made. You can force LabVIEW to prompt the user to input comments into the VI History when changes are made. This is a good practice to incorporate in the development process. Select Preferences from the Edit menu, and then select History from the drop-down box. You can then select the appropriate checkbox so that LabVIEW will prompt for comments each time the VI is saved.

Some firms may desire to be ISO 9000 compliant, which requires more effort. The items covered in this chapter are intended to help in the documentation process for those not requiring ISO 9000. The basic documentation will include how to use a VI, will describe the inputs and outputs, and will discuss the necessary configurations for the user. ISO 9000 requires controlled master copies of all documents to ensure that only the newest version is distributed at any time. In addition, a record must be kept of the controlled documents and the location of their storage.

4.6 THE THREE-TIERED STRUCTURE

Once the requirements are defined and the major design decisions are made, the programmer is ready to work on the structure of the application. An application should be divided into three tiers. The first tier is referred to as the “Main Level.” The Main Level consists of the user interface and the test executive. The second level is the “Test Level” or the “Logical Level.” The Test Level is responsible for performing any logical and decision-making activities. The lowest level is referred

to as the "Driver Level." The Driver Level performs all communications to instruments, devices under test, and to other applications.

Before we look at each of these levels in more detail, we shall identify the benefits of using the three-tier approach. First, this strict partitioning of levels and functions maximizes code reuse. Specific functions or code can be immediately identified and reused because VIs in each level have a defined scope. Drivers can be reused when the need to communicate with another application or instrument arises. Test and measurement VIs can be reused when that test has to be performed. The user interface can also be reused with minor modifications for a different application.

The reuse of code is further simplified with the use of a state machine. State machines work well when the three-tiered approach is applied. State machines and the variations that exist are discussed in Chapter 3. Any state within the state machine can be reused by simple copy-and-paste methods.

A second benefit of using the three-tiered approach is that the maintenance time of the code is minimized. Maintenance and modifications are often necessary after the completion of an application. The application design should therefore plan ahead for changes and make them easy to apply. Because distinct layers exist, modifications can be made quickly and efficiently. VIs that need modification can be identified and located easily. The code that has to be changed can be pinpointed along with the interdependencies with little effort. When this is done, the modifications can be made where needed.

Another notable benefit of the strict partitioning and three-tier approach is the abstraction that is gained. Each level provides an abstraction for the layer below it. The Driver Level abstracts the vague commands used in instrument communication. The Driver Level provides an abstraction for the Test Level. The Main Level then provides an abstraction for the subroutines and measurements by supplying an easy-to-understand user interface. The user interface is an abstraction that hides or disguises all the lower levels involved.

The NI Test Executive serves as the Main Level for an application. It supplies the User Interface function that allows you to select the sequence of tests that you want to perform. The Test Executive can be customized to match the specific needs of the situation. The Test Executive also has the structure already defined, reducing the responsibility of the programmer.

Figure 4.1 is a diagram of a VI hierarchy that uses the three-tiered approach and depicts the strict partitioning of different levels. A quick glance at the diagram reveals the three distinct layers in the application. The Main Level, the Mid-Level, and the Driver Level can be distinguished easily in this example. If Test 2, shown in the hierarchy, has to be used in another program, it can easily be cut and pasted into a new application. Maintenance of the code is easy because changes can be made to a specific section. Also, note how each level abstracts the level directly below it.

Now look at Figure 4.2; it displays the VI hierarchy of an application that does not utilize the three-tiered approach. Code reuse is diminished in this case because the tests are no longer stand-alone VIs. Modifications and maintenance become difficult because changes made in one location may affect other things. Changes made in Driver 1 can affect the Main VI, Test 1 VI and the Driver 2 VI. The

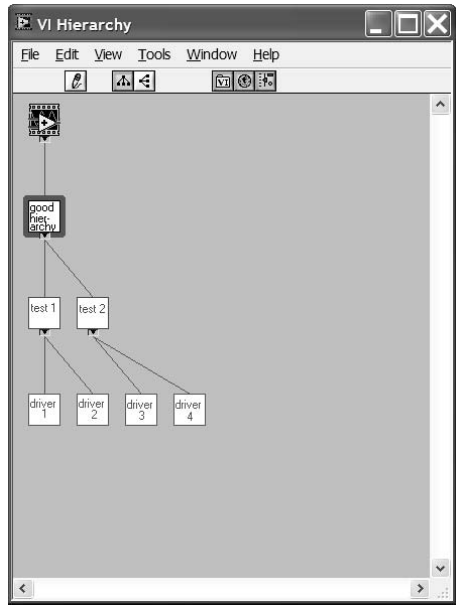

FIGURE 4.1

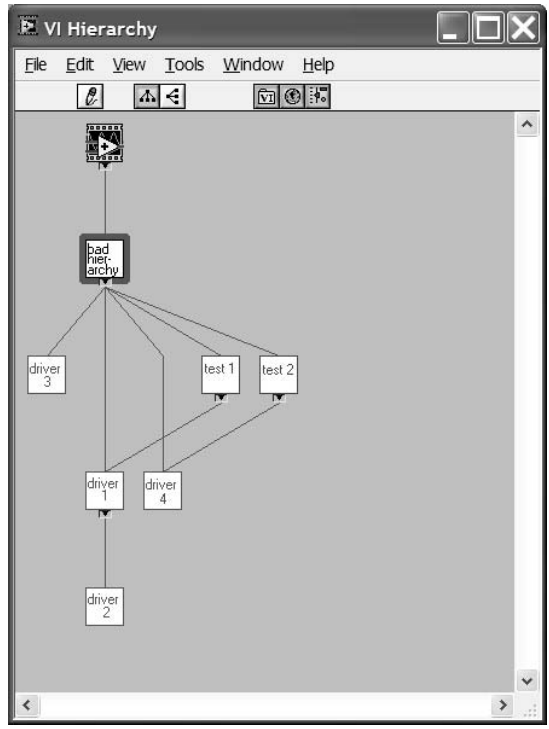

FIGURE 4.2

dependencies are harder to track when a definite structure is not used for the program. Locating a specific section of code will take longer because drivers and tests are mixed. Also note that there is no abstraction below the user interface.

4.7 MAIN LEVEL

Let's first look at the Main Level, which serves as the user interface and test executive. The Main Level should consist of a single VI. Only Test Level VIs are allowed to be called from this first tier. The Test Level will then call the needed drivers for the specific operations. The Main Level should avoid calling drivers because the abstraction benefits are diminished. Reuse is diminished when specific sections cannot be differentiated for copying and pasting methods. Furthermore, the code panel and hierarchy also become difficult to read and maintain. The use of application tiers aids reusability and readability.

The Main Level VI provides user interface functions. It supplies the needed structure for adding application-specific tests, and offers flexibility for changes. National Instruments offers an application called TestStand to be used as a test executive/sequencer. If you are using TestStand, you will already have the extent of partitioning available to gain the benefits of the three-tiered approach. You will be supplying the test and driver level VIs to incorporate into the framework of the executive.

4.7.1 User Interface

The user interface is part of the main level. The user interface is significant because it is the means by which interaction and control of the program occur. LabVIEW provides various tools for designing an effective front panel. Its graphical nature gives it an edge over other programs when it comes to the user interface. ActiveX and .NET controls can now be used in addition to the basic controls. This section will provide some tips and examples for developing effective interfaces.

As TestStand already has a user interface, you would no longer have the responsibility of creating one. TestStand allows the operator to select the test sequence and control execution of the sequence through the interface. The results of the test sequence are shown in a display that also indicates pass or fail status.

4.7.1.1 User Interface Design

The user interface should be designed with the target operators in mind. The interface should not be designed solely to fulfill the functional requirements, but it should also be user friendly. Unless the programmer is the one using the application, it can be the only interaction the operator has with the program. The Main Level user interface should allow the operator to select settings that are variable. Keep in mind that the user inputs may have to be validated. Unexpected inputs will cause the program to behave in an unexpected manner. Variable inputs by the user may include choosing measurements to perform, inputting cable loss parameters, selecting device addresses, adding file storage tags, selecting processes to monitor, etc. These vari-

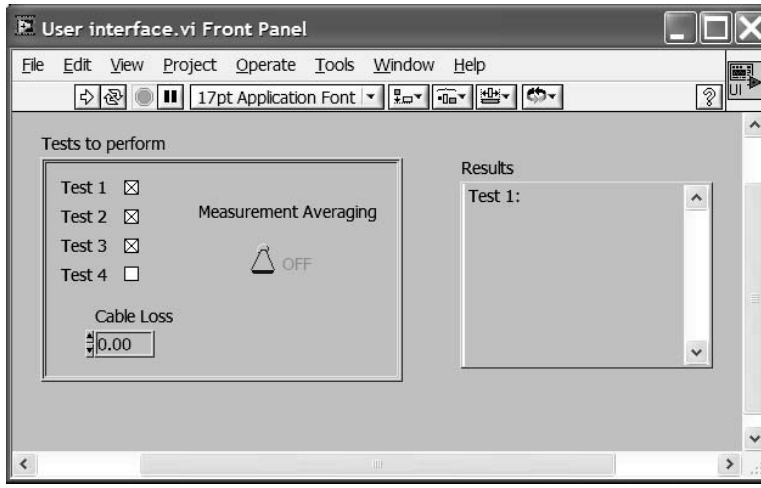

FIGURE 4.3

ables are then passed to the Test Level, ultimately dictating the program flow. Figure 4.3 is an example of a simplified user interface.

Consider using clusters to group-related controls and indicators. Not only does it place the related controls together, but it also reduces the number of wires on the code diagram. When you are trying to manage large amounts of data, the code diagram can get confusing with all of the wires of data being passed to VIs. Clusters can be unbundled as needed, with only one wire representing the group of controls otherwise. Even if you are not using clusters, try to use a frame decoration to group the controls for the user.

The user interface should utilize controls and displays that are appropriate for the situation. If you are entering a cable loss, for example, you should use a digital control with the appropriate precision rather than a knob. This is more practical because the value can be read and changed easily. Using descriptive labels is a good way to differentiate the various front panel controls. Try not to clutter the main user interface with controls or displays that are not needed. The person using the program can easily get confused or lost if too many controls, indicators and decorations are used on the front panel of the user interface. The use of menus will help reduce the clutter, and will give the interface a nice appearance. Use buttons if the function will be used frequently; otherwise, use menus. Dialog controls are also good for user interface functions.

Remember to give the user a way to cancel or abort execution. This is easy to overlook, but is very important to an operator. Users need a way to stop in the middle of execution without having to use the Abort Execution button on the toolbar.

Graphs and charts are useful for displaying data. Sometimes just a glance at a graph can reveal a number of things, but graphs, charts, and other graphics should be used only as needed. Graphing while acquiring data will not only slow the execution of the program, but will take up more memory. Memory concerns will grow as the number of VIs written for the application grows.

4.7.1.2 Property Node Examples

You can use your imagination to develop a professional user interface. The user interface can be simplified by using the Tab control to group similar functions. Another way to simplify the user interface is to allow the user to see only the available options. For example, let's say that you are creating a VI to get user inputs to set up your test system. One portion of the setup requires you to define how to communicate with your unit under test (UUT). There are several options for communications depending on the type of unit.

For our example we are communicating with a UUT through serial communications. There are several options for communicating serially. You could use the computer serial port, a terminal server or even a GPIB to RS232 controller box. If you are using a terminal server to communicate with your UUT you would want to know the IP address and port you are connecting to. You would not want to see a GPIB address or COM port number. By using property nodes, you can control what the user sees based on the chosen input.

Figure 4.4 shows the user interface for the above example that utilizes property nodes. There are 3 tabs for entering information. On the UUT Communications tab there is a control to select the method of communication used. Based on this selection the appropriate options are displayed. In this case, the Terminal Server option was selected. To the right of the selector control are the inputs for IP address and port number.

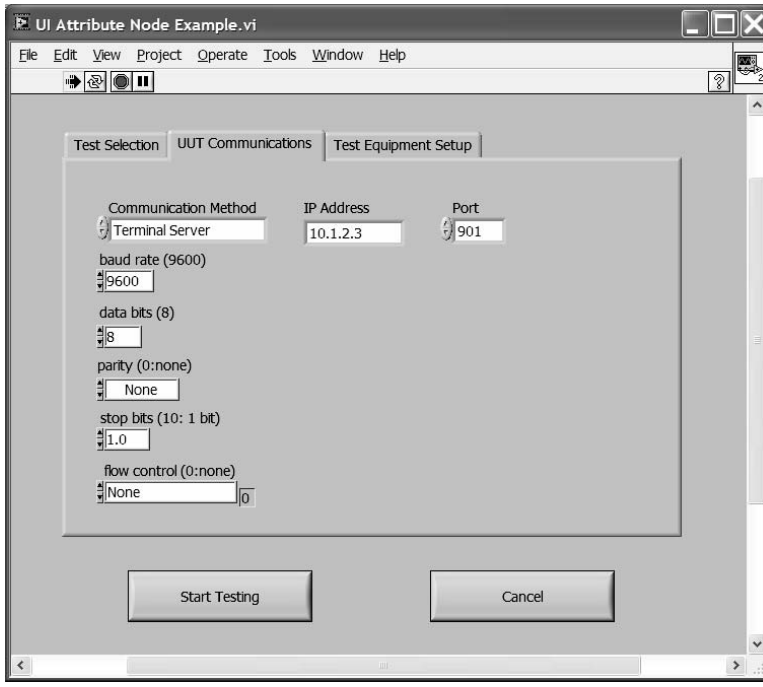

FIGURE 4.4

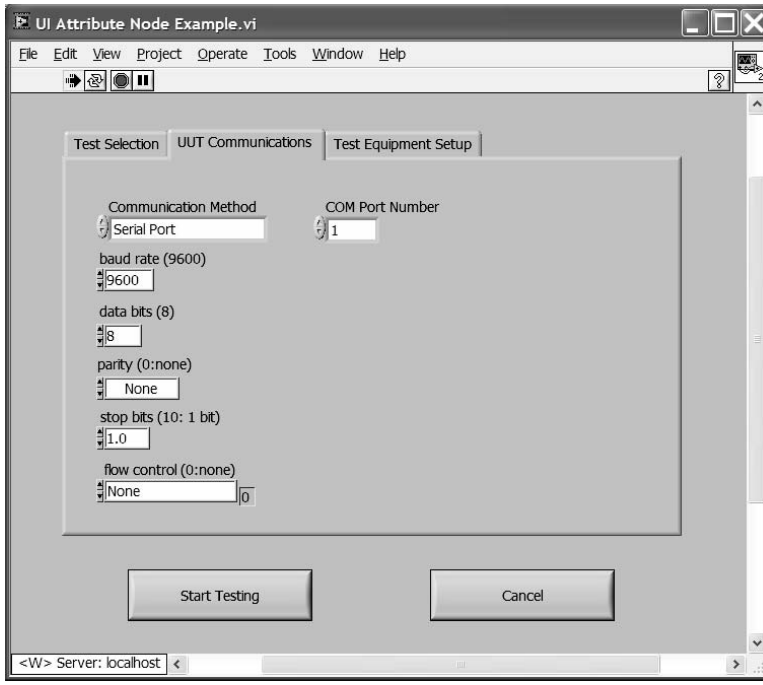

FIGURE 4.5

Figure 4.5 shows what appears when the serial port option is selected. Figure 4.6 captures the code diagram of the example VI. Property nodes are used to make the controls visible or to hide them. Property nodes can be created by popping up on the control from the code diagram. Select Create from the menu, then select Property Node. The available options for the selected control will be available. The Visible property was used in the example. Some common properties that can be manipulated include Visible, Disabled, Key Focus, Blinking, Position, Bounds, Caption, and Caption Visible.

Using Property nodes once again, Figure 4.7 is an example of a menu and submenus structure that is simple to implement. The front panel shown has the main menu that appears on the user interface panel. A Single Selection Listbox is used from the List & Rings palette. Figure 4.8 shows the submenu that appears when the first item, Select Tests, is selected from the listbox. A Multiple Selection Listbox becomes visible and allows the operator to select the tests that have to be executed. When all the needed settings have been completed, the user can hit the Start Testing button to begin execution of the tests.

Figure 4.9 is an illustration of the code diagram for this example. The case structure is driven by the main menu selection. The structure is placed inside a main While loop that will repeat until the Start Testing button is pressed. The Visible Property node is used to make the submenus appear when a selection is made on the main menu. Note that you must first set the Visible Property node for all of the submenus to “false” before the While loop starts.

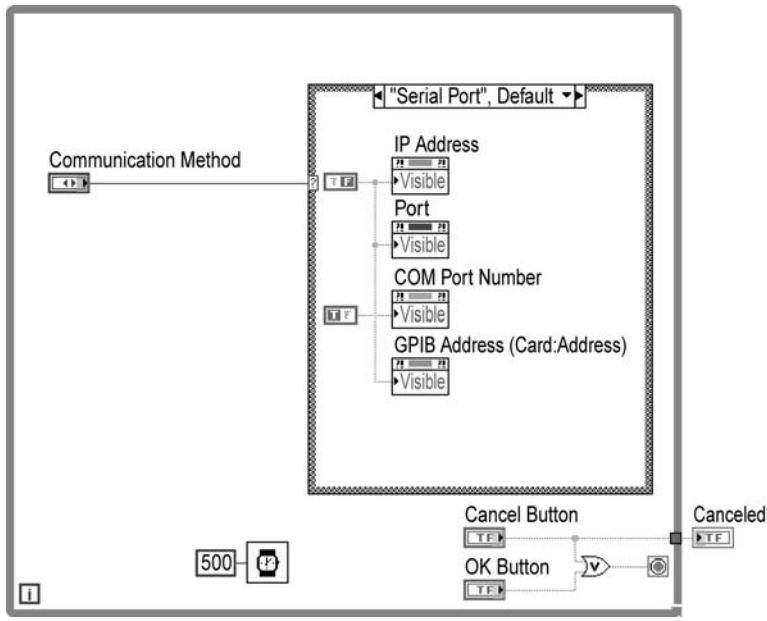

FIGURE 4.6

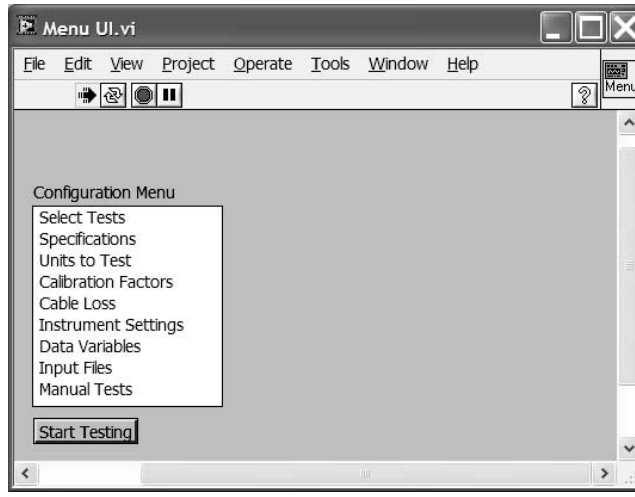

FIGURE 4.7

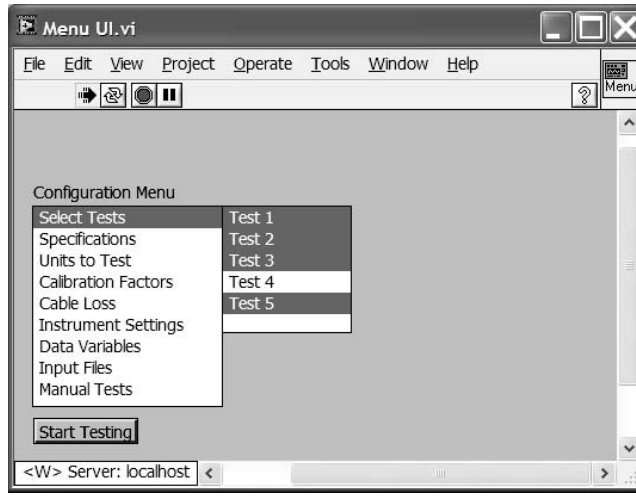

FIGURE 4.8

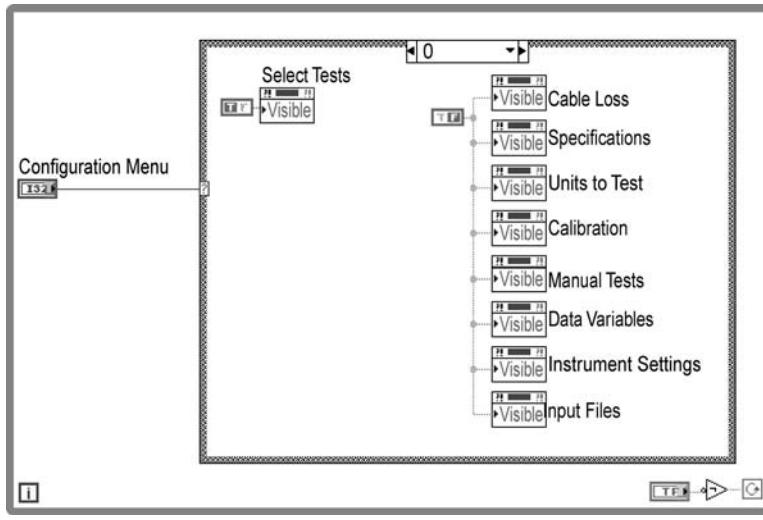

FIGURE 4.9

4.7.1.3 Customizing Menus

LabVIEW run-time menus can be customized to suit specific needs by using the Menu Editor. The menu contents can be modified by selecting the Run-Time Menu item from the Edit menu. The drop-down box allows the programmer to select either the default, minimal, or custom menus to appear during VI execution. The default selection displays the menus that are normally available while the program is not executing. The minimal selection is a subset of the default that appears during run-time. The custom selection requires the programmer to create a new menu structure

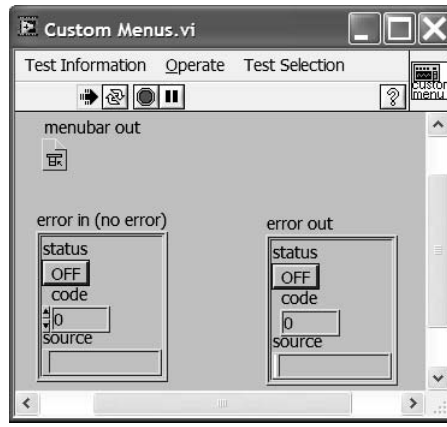

FIGURE 4.10

and save it as a real-time menu (*.rtm) file. Once a real-time menu file is created, it can be used for multiple VIs. A shortcut key combination can be specified for each menu item that is created. The on-line help explains how to customize menus, including how to add User Items.

Figure 4.10 is an example of how custom menus appear when a VI is executing. Three main menus are displayed on the front panel: Test Information, Operate, and Test Selection. The Operate menu is an application item that is normally available if the menu is not customized. The Menu Editor allows you to utilize application items in addition to user items.

Figure 4.11 illustrates how the custom menus may be used programmatically once they have been created. The Current VI's menu bar returns a refnum for the current VI. This VI is available in the Menu subpalette of the Application Control palette. This refnum is passed to the Get Menu Selection VI, which is available in the same subpalette. The Get Menu Selection VI returns the menu item that was selected, as well as the path of the selection in the menu structure. In this diagram, the Get Menu Selection VI is used to monitor selections that are made through the custom menus. The menu selection is then wired to a case structure that takes the appropriate action depending on the selection that is made. The Begin Testing case

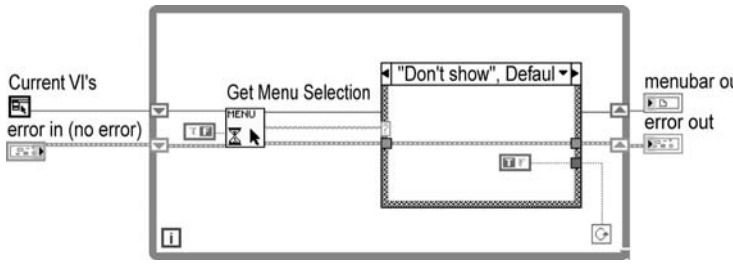

FIGURE 4.11

that is shown corresponds to a menu item in the Test Information menu from the previous figure. When Begin Testing is selected the While loop terminates and the VI completes execution. By utilizing other VIs in the Menu subpalette, a programmer can dynamically insert, delete, or block menu items.

4.7.2 Exception-Handling at the Main Level

Error handling is one element of the project that is often overlooked or not well implemented. Planning for the possibility of something going wrong is difficult, but necessary. A well-designed program will take into account that errors can and do occur. Building exception handling into a program has several benefits. It is a way to notify the operator something has gone wrong that needs attention. It is also very useful for troubleshooting and debugging purposes, as well as for finding out where and why the problem occurred.

There are different ways to control the error situations that can arise. One way is to let the program attempt to correct the problem and continue execution. For errors that cannot be corrected, the application may complete tests not dependent on the failed subsection. Another possibility would be to halt execution of the program and notify the user via a dialogue box, e-mail, or even a pager.

Error handling is an important task that should be managed in the Main Level. This forces all errors to be dealt with in one central place, allowing them to be managed better. The Main Level controls program flow and execution. The Main Level should also determine the course of action when faults occur. When errors are handled in several locations, or as they occur, program control may have to be passed to lower levels and may be difficult to troubleshoot. Also, when errors are handled in more than one location, the code for the handling may have to be repeated.

When a state machine is used, this significant task is made easy because one state is assigned specifically for error handling. When errors are generated, the state machine jumps to the Error state to determine the course of action. Based on the severity of the fault that has occurred, the Error state in the Main Level will decide what will be done. If the error is minor, other states that might be affected will be parsed and the remaining will be executed. If it is a major fault, the program will perform all closing duties and terminate execution in the normal manner while notifying the user of the error. Chapter 6 discusses exception handling in more detail.

Handling execution based on pass or fail criteria should also be considered. If you are using TestStand the user can specify the course of action when a test fails. You can continue to the next test, stop execution of the whole sequence, or repeat the same test again. Dependencies can be created for the individual tests. A dependency, once created, will execute a test based on the result of another test. The result can be defined as either pass or fail.

4.8 SECOND LEVEL — TEST LEVEL

The Test Level is called by the Main Level. The VIs in this level should be written on a stand-alone basis to allow reuse. Each Test Level VI should perform one test or action only. The code should be broken up so that each test that needs to be

performed can be written as a separate VI. When multiple tests are combined in one VI, they are not easily reused because either the extra tests that are not needed would have to be removed, or the extra tests must be executed unnecessarily. These second tier VIs are basically test and measurement subroutines, but can also include configuration and dialog VIs.

Writing each test exclusively in its own VI facilitates reuse in cases where the measurement subroutine has to be executed more than one time. An example of this is making temperature measurements at multiple pressure levels. When a temperature is measured, it will vary with the pressure conditions. A VI that performs a temperature measurement can be written and called as many times as needed to test at the different pressures. Note that the efficiency of the VI is maximized when the pressure is set outside of the temperature measurement VI, and a call is made to it as many times as needed.

The measurement subroutine VIs should perform the initialization of the instruments and any configuration needed to make the measurement. This may include setting RF levels, selecting the necessary instrument fields, or placing the device under test in the appropriate state. These initialization steps must be taken within the VI because the previous condition of the devices may not be known. This is especially true when using a state machine because the program jumps from one state to another; the order of execution is not necessarily predetermined.

When a state machine is being used, only one test or measurement VI should be placed in each state. The benefit of this is that when a particular test needs to be performed, the program executes only the associated state. Placing more than one test in one state causes the additional tests to be executed even if they are not needed. This results in an application that takes more time to run. It also results in loss of flexibility of the state machine. An example of a single test in each state of the state machine is shown in Figure 4.12. The state shown will be executed whenever the particular test has to be performed; the purpose of this state is clearly defined. This method reduces clutter and makes the code diagram self-explanatory.

Flowcharts can assist in the implementation of subroutines and Test Level VIs. They help in defining the execution flow of the VI and the specific decisions that it must make. Flowcharts are especially helpful in LabVIEW because it is a dataflow-based programming language, similar to a flowchart. Once the flowchart is formed,

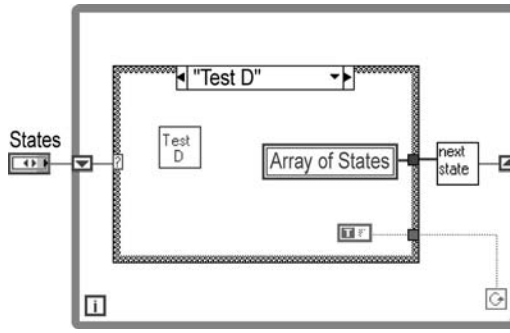

FIGURE 4.12

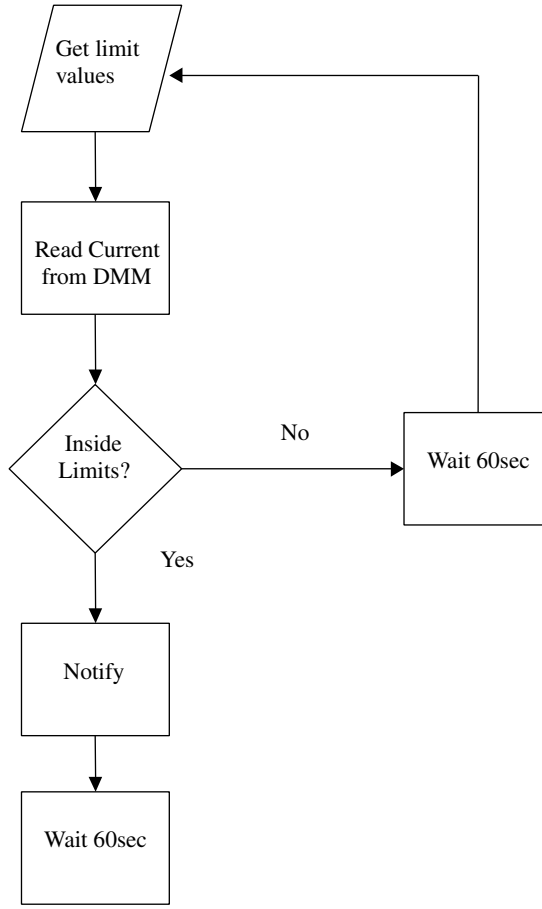

FIGURE 4.13

it is relatively easy to code it in LabVIEW. Figure 4.13 is an example of a simple flowchart. It is checking to see if the current draw from a source is within the limits. This is easily coded in LabVIEW because of the similarities between the two. If you compare the flowchart with the actual implementation in Figure 4.14, the similarities and ease of conversion become apparent. The VI reads the current every 60 seconds to check if the value is within the specified limits. When the value is outside the limits, it will terminate execution and the front panel indicator will notify the user. The flowchart and the LabVIEW VI perform the same function.

4.9 BOTTOM LEVEL — DRIVERS

The Driver Level is an abstraction that makes the Test Level easier to understand. It conceals little-known or unclear GPIB, serial or VXI commands from the user. This level performs any communications necessary to instruments and devices being used. Drivers can be classified into measurement, configuration and status categories.

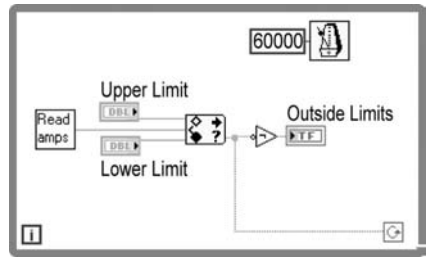

FIGURE 4.14

An efficient way to write drivers is to write each measurement command to one VI each, group configuration commands logically into VIs, and write each status command to one VI. As an example of this driver architecture, examine the HP8920A driver set by National Instruments.

Simply put, measurement drivers are used to perform a measurement. One VI should be used to perform one measurement to maximize the reuse of the VI. By writing measurement drivers in this manner, the driver can be called in the same application for different cases, or in a different application where the same measurement needs to be performed. If more than one measurement is grouped into a single VI, either one of the measurements must be stripped out for reuse or other measurements will have to be performed unnecessarily.

Configuration drivers set up the instrument to make a measurement or place it in a known state at the start of an application. Configuration commands can be grouped logically in VIs. When a measurement has to be performed, usually more than one configuration command is needed to prepare the instrument. Sometimes many parameters have to be configured for a single test. Writing one configuration command to a VI would create difficulty in maintenance because of the number of VIs that will result. Grouping the configuration commands according to the type of measurement will minimize the number of VIs on the Driver Level. In addition, memory space can also be used more efficiently by following this style.

Drivers can range from very simple to very complicated, depending on the instruments being used. A driver for a power supply might need only a few parameters and commands, but an instrument like a communication analyzer might have upwards of 100 different commands. In this case you must group configuration commands to reduce the number of VI drivers that need to be written.

Status drivers simply check the state of an instrument by reading status registers. These are usually written as needed, one to a VI. An example of a status driver is a register that holds the current state of a particular measurement. One bit will be set if the measurement is under progress, and cleared when it is finished. You must ensure that the measurement is finished before reading the value so the status register is checked. Remember that drivers can be created for other types of communication needs as well.

If you need to use TCP, DDE, ActiveX, or PPC, you can use a similar logic when developing the lower layer of the application. When VIs are created to perform a specific action, configuration, or status inquiry, they can be reused easily.

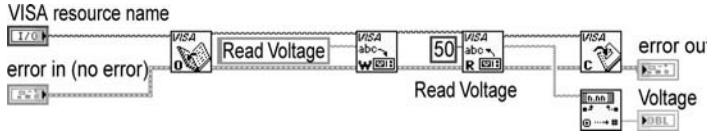

FIGURE 4.15

Figure 4.15 demonstrates a simple driver that can be used to read the voltage from a voltmeter. It has been written so that it can be called whenever the voltage needs to be read. The instrument handles are opened and closed inside the driver.

4.10 STYLE TIPS

We have seen numerous applications in the past few years that do not incorporate good practices that could increase the efficiency, readability, maintainability, and reuse of the code that has been developed. Most of this chapter covers topics to assist the reader in developing successful applications by revealing the programming style that has been effective for the authors. This section was intended to provide more tips on programming, this time by uncovering inefficient programming styles and common pitfalls that can be avoided.

4.10.1 Sequence Structures

The first inefficient programming style is a result of the overuse of stacked sequence structures in the code diagram. Stacked sequence structures were described in Chapter 1 in detail. The main purpose of the sequence structure is to control the execution order of the code. Code that must execute first is placed in the first frame, and then pieces of the code are placed in the appropriate frame in the order that execution is desired. If your code is data-dependent, however, sequence structures are not needed. The execution order is forced by dataflow; VIs will execute when the data they need becomes available to them.

Overuse of sequence structures is the consequence of not utilizing the structures as they were intended. We have seen VIs that contained sequence structures with 50 or more frames. When the architecture or design phase of the application is omitted, an application with so many frames can be an outcome. It signals that perhaps the VI is performing too many actions and that subVIs are not being used sufficiently. When too many actions are performed, the code that is developed is no longer modular. Lack of modularity hampers the ability to reuse code. Consider using the three-tiered structure approach to your application if your sequence structures have too many frames. The use of subVIs for tests and subroutines will reduce the need for many frames while increasing the ability to reuse code. The frames in the sequence structure can be easily modularized into VIs. At the same time, the code will become more readable and maintainable. By developing the description of logic for the VIs during the design phase, you can determine what each VI should do as part of the whole application.

Figure 4.16 displays a stacked sequence with only four frames, but notice how the wires are already beginning to degrade readability of the code. The sequence

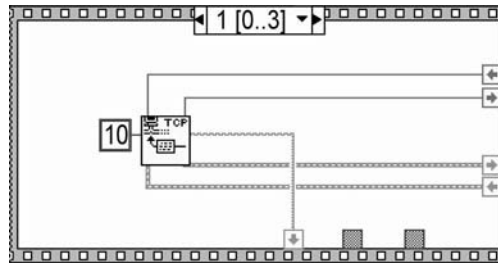

FIGURE 4.16

locals are not easy to follow as the data is being passed from one frame to the next. Code reuse is also becoming difficult in this example. Now imagine what the code would look like if there were 20 or 30 frames in the sequence structure.

If your VIs are data-dependent, you do not have to use sequence structures. For example, execution order can be forced through VIs that utilize error I/O with error clusters. The need for excessive sequence locals may indicate that execution order can be forced simply through dataflow. When many locals are used, problems arise in remembering which local is passing what data. It also degrades readability because of the wiring that is needed to support them. You must; however, be aware of any race conditions in your code.

4.10.2 Nested Structures

Nested case structures, sequence structures, For loops, and While loops are sometimes necessary in the code diagram to accomplish a task. However, creating too many levels of nested structures can lead to inefficient code that lacks modularity and readability. The arguments presented previously on the use of stacked sequence structures apply to the use of nested structures.

Try to avoid nesting your structures more than three levels deep. When too many levels of nesting are used, the code becomes difficult to read. Data wires being passed into and out of the structures are not easy to follow and understand. When case structures are being used, you must look at each case to determine how the data is being handled. This, along with the use of sequence locals or shift registers, For loops, and sequence structures, adds to the readability problem.

Figure 4.17 shows the code diagram of a VI that utilized nested case structures four levels deep. Although the case structures are nested only four levels, it is difficult for anyone looking at the code to determine how the final result was actually obtained. You have to look at all the possible true and false combinations to figure out how the data is being manipulated. Imagine if this VI had more than four levels, or if there were more than just the two true and false cases used in each nest. The readability would be degraded further, while code reuse would be impossible.

Utilizing too many levels may also be a signal that your VI is performing too many actions. When too many actions are being performed, the resulting code has no modularity. This hinders the capability to reuse your code. The use of subVIs can reduce the need for excessive nesting of structures, as well as improve code reuse.

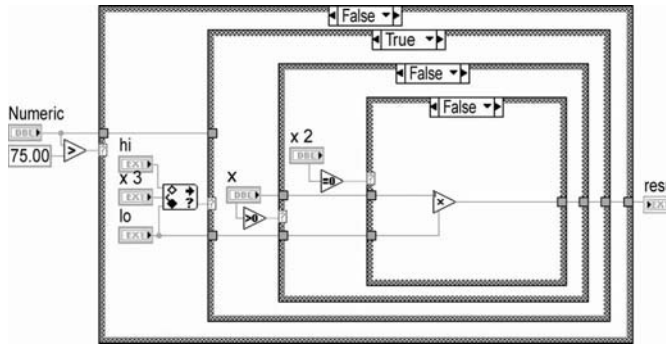

FIGURE 4.17

4.10.3 Drivers

Another bad programming style that we have seen is that drivers are sometimes underused. When communication with an external device or program is being performed, the I/O operation is executed in the Test Level, or even in the Main Level, instead of utilizing a driver. The concept of drivers is not fully understood by some LabVIEW programmers. A question that was posed at one user group meeting was, "Why do I need drivers when I can simply look up the command syntax and perform the I/O operation where it is needed?"

There are definite advantages that can be gained by creating and using drivers. The abstraction that drivers provide is a notable benefit for the application. The actual communication and command syntax is hidden from those who do not need or wish to see this code. This also improves the readability of the code when these obscure operations are not mixed with the Main and Test Level VIs.

The use of drivers also facilitates reuse of code. When drivers are not used, the actual code that performs the communication is difficult to reuse because it is part of another VI. Cutting and pasting part of the code is not as easy as inserting a new VI. However, when drivers are written to perform specific actions, they can be reused easily in any application by inserting the driver VI. Drivers must be developed in a way that will simplify its reuse. A thorough discussion on drivers and driver development is presented in Chapter 5.

Figure 4.18 demonstrates some of the reasons why drivers should be used. The VI shown is performing both instrument communications and other activities, using the results obtained. Compare this diagram to the driver shown in Figure 4.15 earlier. Notice that the instrument communications could have been placed in a separate VI, exactly as was done in the driver in Figure 4.15. Abstraction, readability, and reuse could have been improved through the use of a driver.

4.10.4 Polling Loops

Polling loops are often used to monitor the occurrence of particular events. Other parts of the code are dependent on the execution of this event. When the event takes

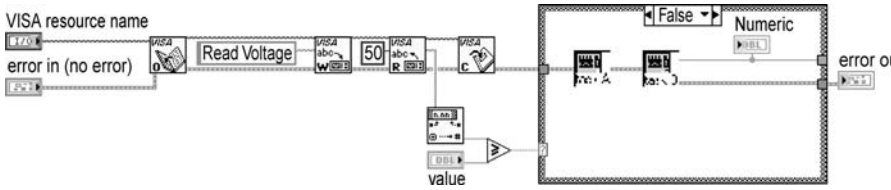

FIGURE 4.18

FIGURE 4.19

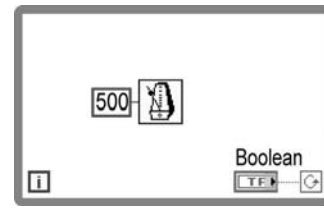

FIGURE 4.20

place, the dependent code executes in the appropriate manner. Using polling loops to monitor an event may not be the best way to accomplish this goal, however.

Tight polling loops can use all of the available CPU resources and degrade the performance of a system noticeably. If you are working on a Windows platform, you can use the System Monitor to view the kernel processor usage while you are performing activities on the computer. We can try a simple exercise to demonstrate this point. Open a new VI and copy the simple VI diagram shown in Figure 4.19. Set the Boolean to "true," run the VI, and monitor the processor usage; almost 100% of the processor will be used for the simple polling loop shown. What happens if we introduce a simple delay in the same polling loop? Use the Wait until Next ms Multiple in the loop with a 500-millisecond delay as shown in Figure 4.20, and monitor the processor usage again. The resources being used are significantly lower when a delay is introduced. Polling loops will certainly reduce the efficiency of your application.

If you are using polling loops, try to use delays where tight polling loops are not necessary. When loops are used for the user interface, the operator will not perceive a delay of 250ms. If you are using polling loops to synchronize different parts of your code, consider using the Synchronization VIs that are available in the Synchronization palette. These include Notification, Queue, Semaphore, Rendezvous, and Occurrences VIs.

4.10.5 Array Handling

The manner in which arrays are handled can affect the performance of an application considerably. Special care should be taken when performing array operations with For loops. A scalar multiplication of an array is a good example for demonstrating the methods available to perform this action. Figure 4.21 illustrates one way to perform the multiplication. The array is passed into the For loop, where the element

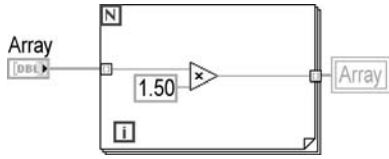

FIGURE 4.21

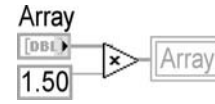

FIGURE 4.22

is multiplied by a constant of 1.5, and then passed out. The correct result is acquired; however, the method chosen to perform the multiplication is very inefficient. The same result could have been acquired without using the For loop. Figure 4.22 shows that the array could simply have been multiplied by the constant without the For loop. The first method is inefficient because it requires the array to be broken down into its elements, then each element of the array must be multiplied by the constant separately, and, finally, the array must be rebuilt with the results.

Whenever possible, you should try to avoid passing arrays into loops to perform necessary operations. Passing large arrays will result in longer execution times for applications, as well as the use of more memory during execution. Both the speed and performance of your application will be affected. The Show Buffer Allocations function can help you find places where arrays are being allocated. The function can be found under the Profile folder under the Tools menu. This function is described in more detail in Chapter 2.

4.11 THE LABVIEW PROJECT

Building, maintaining and compiling source code for an application can be a difficult task at times. The location of source files must be tracked, a list of supplemental files must be maintained, and build specifications need to be created for the application. You must also be able to handle the complex task of multiple developers working on the same project. All of these tasks can now be accomplished using the LabVIEW Project. We will discuss some of the project features and terminology. Some functions were also discussed in Chapter 2. The Shared Variable is discussed in depth in Chapter 7.

4.11.1 Project Overview

A project is a grouping of LabVIEW VIs, controls and build specifications. Other files relating to the application, such as documentation and support files, can be stored in a project as well. All the information related to the project is stored in a project file. The project file has a .lvproj extension. The project file includes references to the files contained in the project; build information, deployment information, and configuration information. The project file is an XML file that can be viewed by opening it with a text editing application. Figure 4.23 shows a project file that has been opened in WordPad.

Fortunately you do not have to edit a project in a text file. In LabVIEW, you can open up a project in the Project Explorer. The Project Explorer window is a

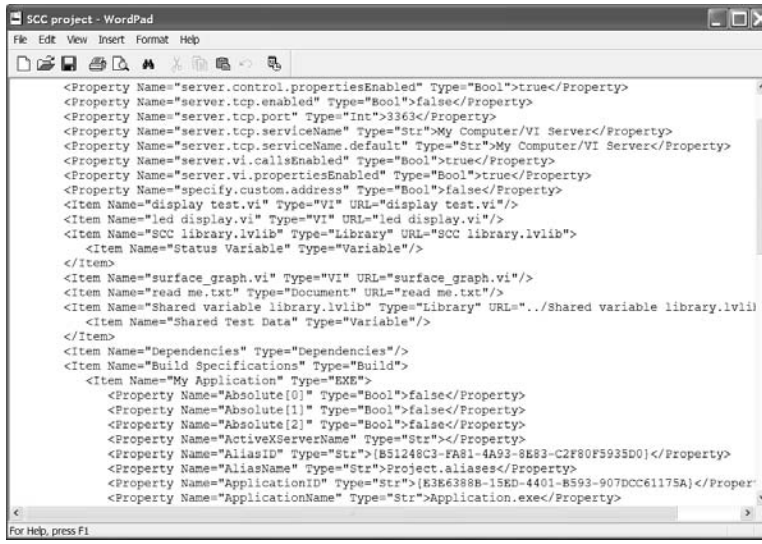

FIGURE 4.23

separate window that is loaded when a project is opened. The Project Explorer window is similar to the Windows Explorer. The window contains a graphical listing of all the items stored in the project. Figure 4.24 shows the Project Explorer window for the project file in the previous illustration. You can execute all project related functions through the Project Explorer window. A discussion of some of these functions will follow.

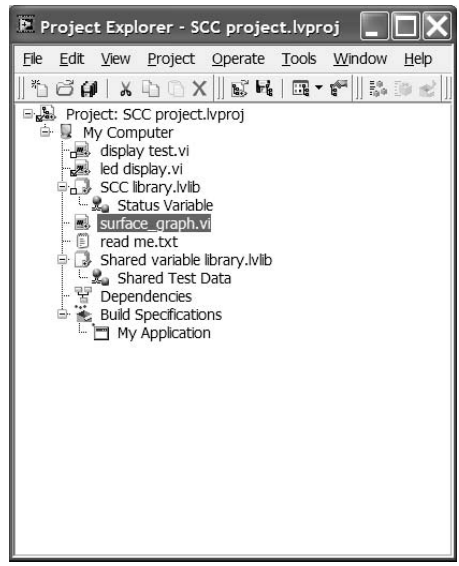

FIGURE 4.24

4.11.2 Project File Operations

To start using a project you will need to create a new project. This can be done by selecting New Project under the File or Project menu. There is also an option to create an empty project in the New dialog box (comes up when you select New from the File menu). If any VIs are currently open, you will see a dialog box asking you if you want to add the currently open files to the new project. Once you address this dialog a Project Explorer window will pop up. By default an empty project will display the name of the new project with the My Computer listed below. My Computer is the target of the project. By default the local computer is the target. Additional targets can be added by right clicking on the project name, which is called the project root, and selecting Targets and Devices from the New category. Here you will have the option to add an existing target or device, an existing target or device on a remote subnet, or a new target or device. This is the only way you can distribute an application to an FPGA, Real Time or PDA target.

Each target will start with two items; Dependencies and Build Specifications. The Dependencies category includes items that are required by the VIs in a target. The Build Specifications category includes build configurations for source distributions, executable files (EXE), installers, shared libraries (DLL), source distributions and zip files. Only source distributions are available without the application builder or LabVIEW Professional Development System installed.

Once your project is opened you are able to start adding files. To add a file to your project you can right click on the target (My Computer) where the item will be added. You will have a lot of options including adding a new or existing VI, folder, Variable and library. You can add non-LabVIEW files such as documents or spreadsheets to the project as well. As with all operations on the project, you can access the operations through the shortcut menu as well as through the main menus. In this instance you can add files through the File menu. There is also the ability to use the New dialog box to add a file to the project. To insert the new item into the project, you need to click on the Add to Project checkbox. The New dialog box is shown in Figure 4.25.

To remove a file from a project you can right-click on the file and select remove. Removing a file from the project does not remove the item from the file system. The reference to the file is simply removed from the project file. The file itself was never physically a part of the project. This is different than the LabVIEW LLB where the VIs are actually part of the LLB file. Because of this method of using links to a file for a project, there needs to be some care taken with the files in the file system. If you move a VI that is part of a project to a new location through the Windows Explorer, the project will not initially find the file. When the project is opened it will try to find the file in the original location, and then start searching elsewhere for the file. You do have the option to browse for the file as well. There is a risk of the project finding a version of the file that you do not want used. Once the project finds the file, the location is updated in the project file. You will have to save the project; otherwise the location change will be lost. As the information is stored in a file, you could manually update the information as well.

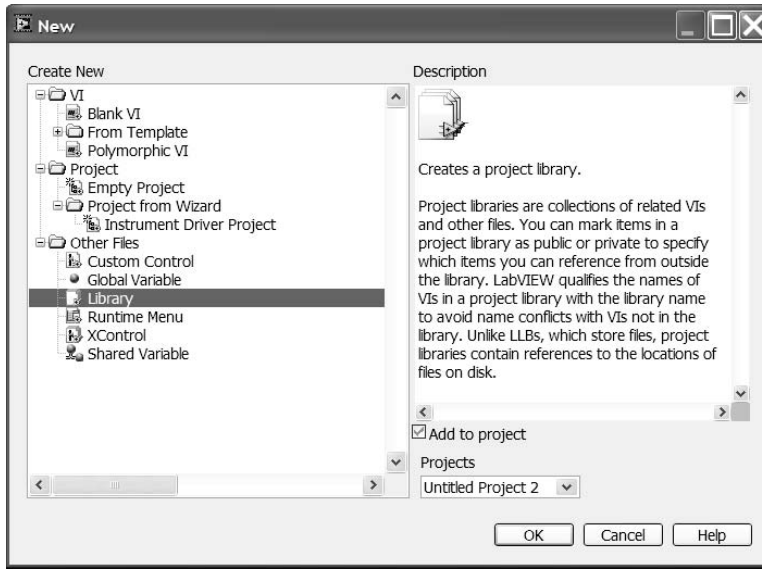

FIGURE 4.25

To add a Shared Variable you use the same procedure as with other files. A Shared Variable is similar to a global variable in that you obtain and pass data to other parts of your program or other VIs locally or over the network without having to connect the data by wires. How to setup a Shared Variable is described in Chapter 2. An in-depth look at the Shared Variable is offered in Chapter 7.

The Project Explorer supports drag and drop functionality. You can add a new file to a project by selecting the VI icon from the corner of a front panel or block diagram and dragging the icon to the Project Explorer. This is equivalent to the methods for adding files already discussed. The process works in the other direction as well. If you are editing a VI you can click on the VI or variable name and drag it to the VI you are editing. The item you had selected will be inserted in your VI. On Windows and Mac operating systems you can select an item or folder and drag it to a target in the Project Explorer.

Finally, you will probably want to use a file that you have inserted into the project. You can open the item by double-clicking on the item name in the Project Explorer. If the item is not a LabVIEW item, the appropriate application will launch in order to open the file.

4.11.3 Project Library

A LabVIEW project library is an object that contains links to several types of objects including VIs, type definitions, variables and other files. A project library is access through the Project Explorer or the project library window. A library maintains the links to the objects it contains in a library file. A library file has a .lvlib extension. A project library should not be confused with a LLB file. A LLB file is a collection

of VIs that are physically a part of the LLB. A project library owns files that are stored in individual files on disk. The files do not move from their original location.

Managing a large application can be difficult. You have to keep track of the hundreds of files that are in the application. The files needed may also include external code, project documentation and support files. By using a project library you have the ability to organize all of the files needed for an application in a single hierarchy. This has the benefit of being able to set permissions on groups of files in a single action. This also will make distributing the files easier. In order to distribute a project library you can either distribute the actual library file along with the corresponding files that the library owns. You can also create a zip file that contains the entire library. The zip file function is part of the Application Builder application.

One problem that pretty much all LabVIEW developers have had at one time or another is the issue of linking to a VI that has the same name as the original VI, but is not the correct VI. It could be a VI that was already in memory that gets saved in a calling VI, or when LabVIEW cannot find a VI and finds a VI with the same name in the wrong location. The project library eliminates this issue. By using XML namespaces, a project library can guarantee that a VI is correct. The VI is stored using the filename and the project library filename. This filename information is known as a qualified name.

To help illustrate the concept of the namespace, an example will be presented. Let's say you are creating a VI to communicate with two pieces of equipment through serial communications. Let us also assume one piece of equipment is connected through the serial port and one is connected through a terminal server. Previously you may have created a library of VIs for communicating through the serial port and a library of VIs for communicating through the terminal server. Now in your test VI you want to insert a VI created for checking the equipment status for both pieces of equipment. You never envisioned using both serial communications and the terminal server, so the status VI you created for both libraries has the same name. As the namespace for each of the VIs is different you could insert both VIs in your test program. The names that LabVIEW would use for these VIs would be in the following (URI) format: project library name:VI name.

To create a new library you would right-click on your target in the Project Explorer and select Library from the New menu item. This will create an untitled project library. You can either start adding files or save the library first. To save the library you can right-click on the library and select Save. The Save function will save the library in the location of your choosing with a .lvlib extension. If you have an existing folder in a project, you can right-click on the folder and select Convert to Library.

The library settings can be configured through the Project Library Properties dialog. This dialog can be launched by right-clicking on a library in the Project Manager and selecting Properties. The dialog is shown in Figure 4.26. The dialog has 3 sections. A General Settings section gives you access to the library protection level, the default palette and default icon. You can also set a version number for your library. The documentation section is where you can add a VI description as well as link to a help file. The Item Settings section gives lets you set the access scope for the entire library or for each individual file.

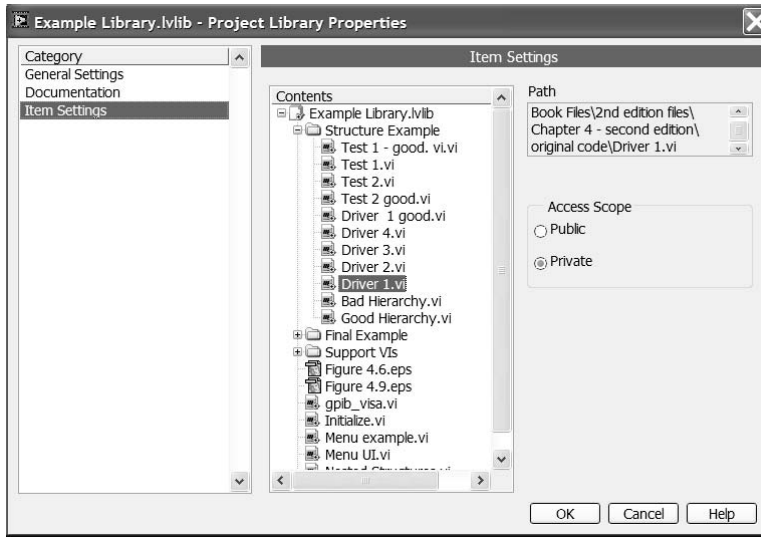

FIGURE 4.26

The access scope lets you set the Project Library or files as either public or private. A public VI is a standard VI that can be opened directly or called by other VIs. A private VI can only be called by VIs that are located in the same project library. If a folder in a project library is marked as private, all VIs contained in that folder will also be private. By using private VIs you can make sure the VI can only be run where they were intended. Figure 4.27 shows a Project Manager view of a library that contains some private VIs. The Private VIs have a key next to their icon.

Setting the protection level in the Project Library Properties window can protect a library or VI. You can set the protection level to Unlocked, Locked or Password-protected. When a project library is locked, items cannot be added or removed. Items that are set to private cannot be viewed. If the library is set to password-protected, the library can only be edited when the password is entered. This also holds true for the viewing of private files. They can only be viewed after the password has been entered.

To add files to a project library you can right-click on the library and select the item you want to add from the menu. You can also drag and drop an item in the project to the project library. A VI that is added to a library will need to be saved in order to properly link to the library. If you add a VI to a library that calls subVIs, you will need to add the subVIs to the project as well. The subVIs are not automatically loaded when the calling VI is loaded. You should also be aware that a VI can be linked to only one project library. If you want to link a VI to a new library you will need to break the connection by selecting Disconnect VI from Library from the File menu.

4.11.4 Project File Organization

The larger the application, the more difficult it can be to manage all the files in the Project Explorer. You may want to take the time to organize the project so that

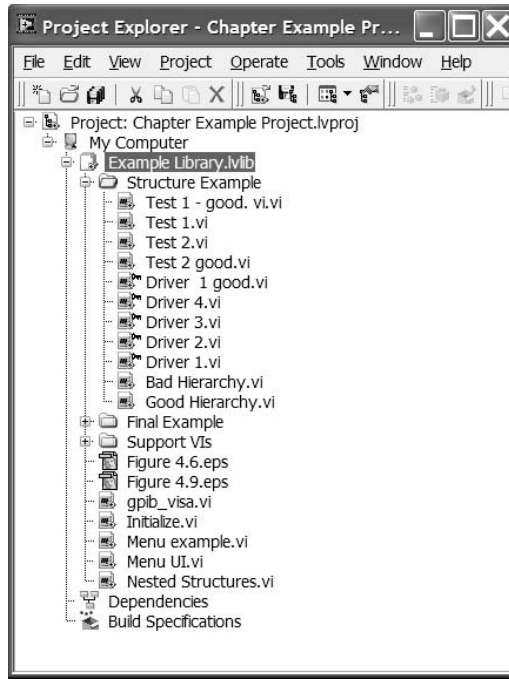

FIGURE 4.27

related items are in common folders. This can make maintenance easier by being able to work on items that are logically grouped together.

To add a new folder to a project or library you can right click on the project item and select Folder from the New option. If you want to add a directory from your files to the project you can right click on the item and select Add Folder. LabVIEW will create a folder in the project that has the same name as the folder on your computer. The files and subfolders that were contained in the original folder will then be added to the project. As a reminder, the actual files on your computer do not change location, the project maintains links to the files on your machine.

The link between the Project Explorer and your machine is not an active link. If you change files in the folder or the folder location, the changes will not be reflected in the Project Explorer. If you want the changes to be available in the project as well, you will need to manually update the project.

National Instruments recommends that you create a separate directory for each project. You should include your project and library files in the directory. If all of your project files are stored in a common directory it will be easier to locate and manage files on your computer.

4.11.5 Build Specifications

While storing and editing your application in a Project file has many benefits, ultimately you will need to get the code you have written to an end user or to a test

system. There are several ways to distribution your LabVIEW code. You could build a source distribution, an executable, a shared library or a zip file. These options are configured and executed through Build Specifications. A Build Specification can only be created through the Project Explorer.

Once you have a project started you may decide you need to make a copy of the code in another directory in order to make some experimental changes. You might also want to give a copy of the code you have been working on to a coworker to tryout or debug. Regardless of why you want to make a copy of your code, you need to be able to pull all the needed files for a project together. This is where the Source Distribution comes in. The Source Distribution is similar to the Development or Application Distribution operations in earlier versions of LabVIEW. You can save all VIs to a new location. You can choose to include vi.lib, user.lib and instr.lib files. You can customize VI properties, apply passwords and remove diagrams.

The first step in creating a Source Distribution is to open a Project. Under project you can right click on the Build Specifications and select Source Distribution. A dialog box will open with three categories of inputs. The first is the Distribution Settings. Here you will select your Packaging Option. You have the choice of distributing the files to a single location, a single location with the same hierarchy that is under My Computer in the Project Explorer and you have a custom option. If you select custom, you will be able to select where all of the files are installed. You can create multiple locations for files. One benefit of distribution of the files in this method is the application will keep track of the file locations, so you will not get a broken arrow after the files are moved.

The second option for Source Distribution is the Source File Settings. Here you will have options for every item in the Project Hierarchy based on the type of item. If you select a VI, you will have the option to include the file, which destination it is going to and password options. You will also have the option to customize the VIs settings, including removing the code diagram. Once you have configured your file settings you can preview your distribution. The preview will show you where the files will be distributed.

Once you are done you can select build to create the specification. Your new build specification will appear in the Project Explorer window. You can right click on the Source Distribution to Build (distribute), remove the distribution or edit the settings.

Source Distributions are great for making copies of your code and sharing code with other developers, but you might be creating an application for a customer. Even if you are the end user, you may not want to purchase a full LabVIEW license for each test machine you are using. These examples are both excellent reasons to build an application. A LabVIEW application is an executable program with an .EXE extension (.APP for Mac users). What if your end user is programming in another language? This would require building a shared library (DLL) so that the end user can reuse your functions. You can accomplish both of these tasks by using Build Specifications.

The application or shared library requires only that the LabVIEW runtime engine be installed on the target computer. The runtime engine includes the files necessary to run your application. The runtime engine can be installed with your application

or can be downloaded from the National Instruments website. To be able to build an application or shared library you need to have either the Professional Development version of LabVIEW or an add-on package. The creation of an executable or shared library is similar to the procedure for creating the Source Distribution. An explanation on how to build an application using the Project Explorer is given in Chapter 2.

The final option is for the Build Specifications in the ZIP file. Here you can add all or some of the Project files to a compressed Zip file at a specific location. You can insert comments into the build specification. The addition of source files is simply a side-by-side window that allows you to add or remove files in a Project to the destination Zip file.

4.11.6 Source Code Management

Working on large projects often requires using source code management software (SCM). SCM applications allow you to maintain versions of your code as well as be able to prevent other users from modifying your code while you are working on it. The project works with several SCM applications for the management of your project files. Chapter 2 discusses SCM management through the project in detail.

4.12 SUMMARY

Developing an application requires good planning and a design process that should be followed. Following a formal process helps avoid costly mistakes and revisions to the code at the end. One software model will not be suitable for everyone. However, following the requirements, design, code, and test phases will aid in developing applications. The structure of the application, which is decided on during the design phase, is an essential piece of the process. It will determine many crucial aspects of the program. The three-tiered approach, as described, embodies the desired characteristics of most applications: the ability to make future modifications, the ability to add features, ease of maintenance, the ability to reuse code, and layers of abstraction. When the strict partitioning of levels is used in conjunction with the state machine, all the characteristics are further enhanced.

A summary example will help in applying the topics and ideas presented in this chapter. Suppose that Company A is involved in the sale and production of Widget A. Let's follow the steps that would be required to develop an application that would be used in testing the widgets to determine if they meet specifications.

This first step involves defining the requirements for the test program. The goal of this application must be defined before beginning to code. After discussing the program with the appropriate people you enumerate the following requirements:

1. Parameters H, W, and D are to be tested using instruments H, W, and D, respectively.
2. The operator will be a factory technician and will need the flexibility to select individual tests as needed.
3. The program should alert the operator when one of the widgets is out of specification limits.

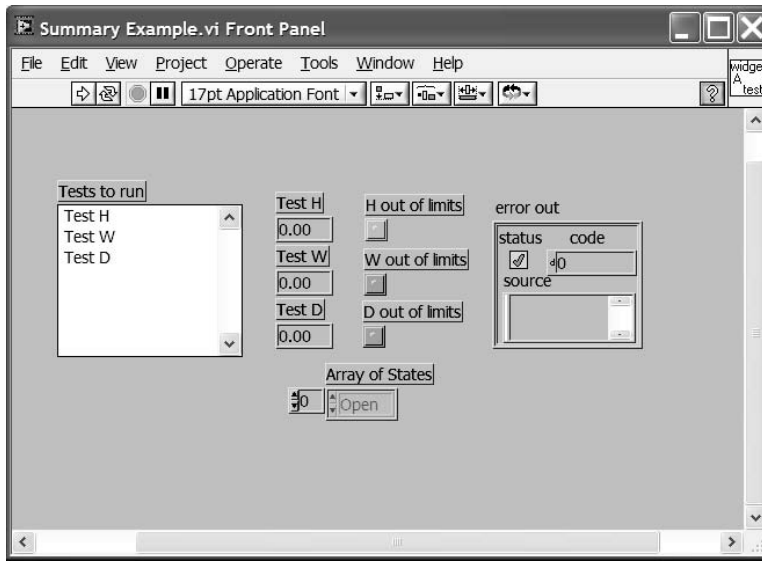

FIGURE 4.28

4. There should be provisions for the addition of tests; measuring Parameter Z using Instrument Z is in the foreseeable future.
5. Company A is planning to produce Widget B next year, which will be tested for H, W, and several other parameters.

The next step is to decide on the structure of the program. We will utilize the three-tiered approach and take advantage of the benefits it provides. The user interface of this application should be simple but flexible enough to provide the operator the level of control needed. Figure 4.28 shows what the User Interface looks like for this application.

The Main Level will use the state machine. It will also abstract the lower levels for the operator. The following states will be needed for this application: Open (to open all communication handles to instruments), Error, Initialize (to put all instruments into a known configuration), Test H, Test W, Test D, and Close (to close all communication channels). Figure 4.29 shows this state machine.

Each test is contained in its own VI, and each state consists of a single test. The Test Level is composed of Test H, Test W, and Test D. Each test VI calls the necessary drivers to perform the measurement. If the operator selects a single test to perform, the other tests will not be executed unnecessarily. An array of states will be built using the selections made by the operator. The first state that will be executed is the Open, and the last state is the Close.

The application takes into account that the needs may evolve over time. If additional tests have to be added, that can be done quickly. Suppose we were asked to add Test M to the current program. All we have to do is follow a few steps to get the needed functionality. First we have to modify the state machine to include the

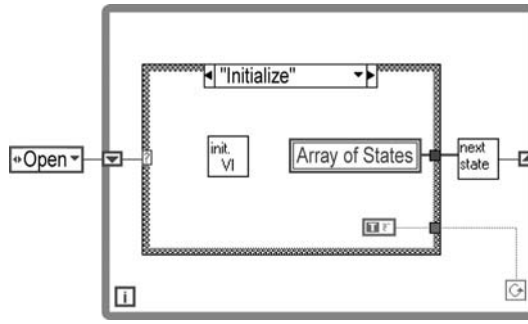

FIGURE 4.29

extra state for this Test M. Then we have to modify how the array of states is built to include the new state if the test has been selected by the operator. Next, we can modify the user interface to include the new test in the list for selection. We would also have to add a display for the measured value, and an LED that would indicate when a widget fails Test M.

Reuse is also made simple by the strict partitioning of levels. When the company begins to produce Widget B, Tests H and W can be reused. Tests H and W are stand-alone test VIs and call the appropriate drivers for performing the tests. If we decide to write another application to test Widget B, all we have to do is place the test VIs in the new application. If the new application were to use the state machine also, then we can copy and paste entire states.

The VI hierarchy for this example is shown in Figure 4.30. The strict partitioning of levels is illustrated by the distinct layers in the hierarchy. At the top is the Main

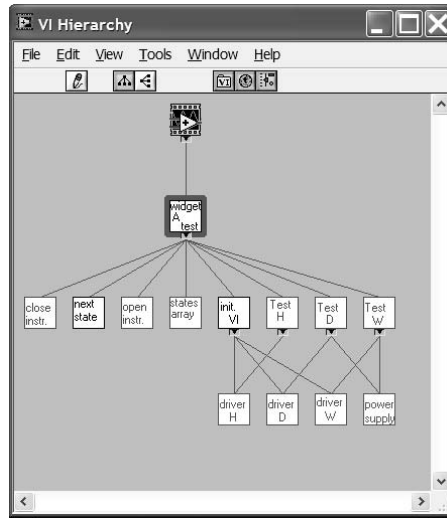

FIGURE 4.30

Level, which controls program execution. The second layer depicts the Test Level VIs. The bottom layer consists of the drivers used to test the widgets. The middle layer VI icons are in blue, and the Driver Level icons are in red. This was done purposely to distinguish the layer that the VI belongs to.

BIBLIOGRAPHY

G Programming Reference, National Instruments, Austin, TX, 1999.

The Art of Systems Architecting. Eberhardt Rechtin and Mark W. Maier, CRC Press, Boca Raton, FL 1997.

5 Drivers

This chapter discusses LabVIEW drivers. A driver is the bottom level in the three-tiered approach to software development; however, it is possibly the most important. If drivers are used and written properly, the user will benefit through readability, code reuse, and application speed.

LabVIEW drivers are designed to allow a programmer to direct an instrument, process, or control. The main purpose of a driver is to abstract the underlying low-level code. This allows someone to instruct an instrument to perform a task without having to know the actual instrument command or how the instrument communicates. The end user writing a test VI does not have to know the syntax to talk to an instrument, but only has to be able to wire the proper inputs to the instrument driver.

The following sections will discuss some of the common communication methods that LabVIEW supports for accessing instruments and controls. After the discussion of communication standards, we will go on to discuss classifications, inputs and outputs, error detection, development suggestions, and, finally, code reuse.

The standard LabVIEW driver will be discussed first. This standard driver is the basis for most current LabVIEW applications. In an effort to improve application performance and flexibility, the Interchangeable Virtual Instrument (IVI) driver was created. IVI drivers will be described in depth later in this chapter.

5.1 COMMUNICATION STANDARDS

There are many ways in which communications are performed every day. Communication is a method of sharing information. People can share information with each other by talking, writing messages, sign language, etc…. Just as people have many different ways to communicate with each other, software applications have many ways to communicate with outside entities. Programs can talk to each other, to instruments, or to other computers. The following communication standards are just some of the methods LabVIEW uses to communicate with the outside world.

5.1.1 GPIB

The General Purpose Interface Bus (GPIB) is a standard method of communication between computer/controller and test equipment. The GPIB consists of 16 signal lines and 8 ground return lines. The 16 signal lines are made up of 8 data lines, 5 control lines, and 3 handshake lines. The GPIB interface was adopted as a standard (IEEE 488). The maximum GPIB data transfer rate is about 1Mbyte/sec. A later

version of the standard with added features was defined in 1987. This standard is the ANSI/IEEE 488.2. This enhancement to the standard defines how the controller should manage the bus. The new standard includes definitions of standard messages for all compliant instruments, a method for reporting errors and other status information, and the protocols used to discover and configure GPIB 488.2 instruments connected to the bus.

HS488 is a standard that NI has created. This standard is an extension of the IEEE 488 standard and increases the GPIB data transfer rate. By using HS488 controllers and compatible instruments, the data transfer rate can be increased up to 8 Mbytes/sec. The biggest benefit of the higher data transfer rate is the use of instruments that return large data sets. Instruments such as oscilloscopes and spectrum analyzers send large amounts of data to the control computer. The HS488 standard allows you to increase your test throughput.

There are two types of GPIB commands. There are device-dependent messages and interface messages. Device-dependent messages contain programming instructions, data measurements, and device status. Interface messages execute the following operations: initializing the bus, configuring remote or local settings on the device, and addressing devices.

Many current instrument manufacturers have standardized remote commands. This allows the user of an instrument to learn how to program an instrument in a shorter period of time and makes instruments more interchangeable. In order to try to make programming instruments easier, the SCPI (Standard Commands for Programmable Instrumentation) command set was developed. SCPI commands are standardized for basic functions that almost all instruments support. There are a number of instruments on the market that are not SCPI compliant. These instruments have their own command sets, and formats. This can make writing automation software difficult. One example is the T-BERD PCM analyzer. This instrument is not SCPI compliant. If you wanted to reset the instrument, you would have to search through the reference manual for the command. In this instance, to reset the instrument, you would have to write "FIRST POWER UP" to the instrument. Not only is the command not obvious, but it would require the developer to spend time hunting down commands. Figure 5.1 illustrates the GPIB driver.

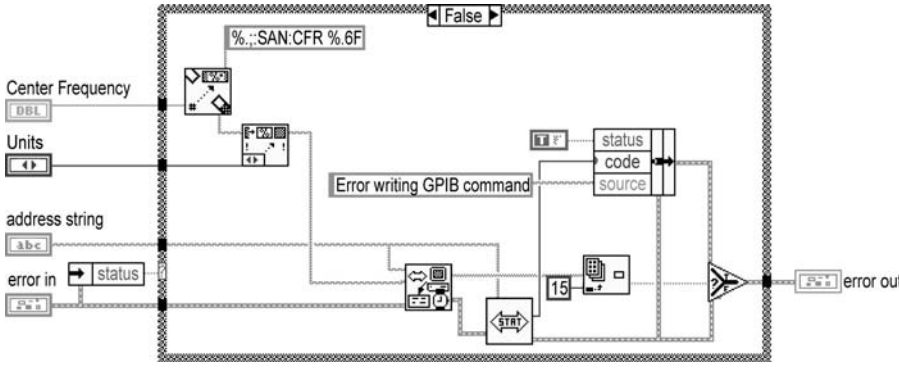

FIGURE 5.1

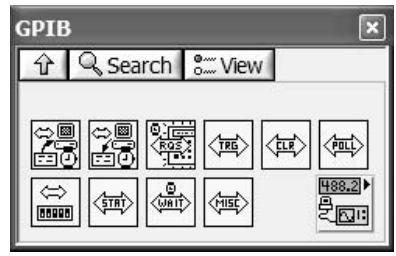

FIGURE 5.2

In the Instrument I/O section of the Functions palette there is a subpalette that contains GPIB drivers. The GPIB palette contains the traditional GPIB 488 commands. On the GPIB palette there is a subpalette, GPIB 488.2, which contains GPIB 488.2 commands. The VIs from these subpalettes can be used in conjunction with a GPIB 488.2 instrument. If the instrument you are using is not GPIB 488.2 compliant, you can only use the VIs in the traditional GPIB palette.

The primary VIs in the GPIB palette are GPIB Read and GPIB Write. These two VIs are the basis for any program using GPIB instruments. There are also VIs used to wait for a service request from the instrument (Wait for GPIB RQS), obtain the status of the GPIB bus (GPIB Status), and initialize a specific GPIB bus (GPIB Initialization). Among the remaining GPIB VIs, there is a GPIB Miscellaneous VI. This VI allows you to execute a low-level GPIB command. The GPIB palette is shown in Figure 5.2.

The GPIB 488.2 palette contains additional functions. The GPIB functions are broken into five categories: single device functions, multiple device functions, low-level I/O functions, bus management functions, and general functions. The single device functions are VIs that communicate with a specific instrument or device. Some of the functions include Device Clear, Read Status, and Trigger. The multiple device functions communicate with several devices at the same time. The VIs define which devices to communicate with through an array of addresses that are input. This category of VIs includes VIs to clear a list of devices, enable remote, trigger a list of VIs, and VIs to perform serial or parallel polls of the devices.

Low-level I/O VIs allow you to have more control over communications. The VIs in this category include functions to read or write bytes from a device, send GPIB command bytes, and configure a device in preparation to receive bytes. The Bus Management functions are VIs used to either read the status of the bus or to perform functions over the entire GPIB. The VIs in this category include VIs to find all listeners on the GPIB, to reset the system, to determine the state of the SRQ line, and to wait until an SRQ is asserted. Finally, the general functions are used to make an address or to set the timeout period of the GPIB devices. The GPIB 488.2 palette is shown in Figure 5.3.

5.1.2 Serial Communications

Serial port communications are in wide use today. One of the advantages of serial communication versus other standards like GPIB is availability: every computer has

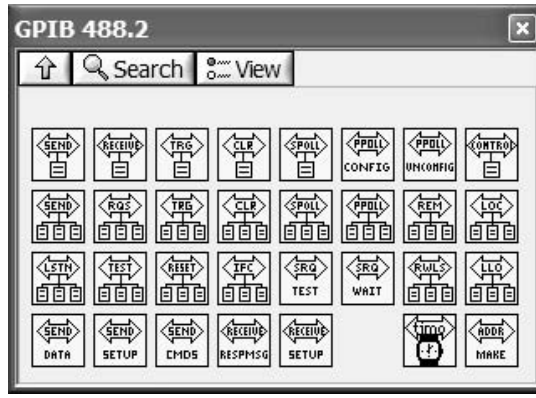

FIGURE 5.3

a serial port. Another benefit to serial communications versus GPIB is the ability to control instruments at a greater distance. The serial standard allows for a longer cable length.

The most common serial standard is RS-232C. This protocol requires transmit, receive, and ground connections. There are other lines available for handshaking functions, but they are not necessary for all applications. Macintosh serial ports use RS-422A protocols. This protocol uses an additional pair of data lines. Due to the additional data lines, the standard is capable of transmitting longer distances and faster speeds reliably. There are other serial protocols available, but these are the most widely used at this time.

The serial port VIs are in the Instrument I/O section of the Functions palette. This subpalette consists of VIs used to read data from the serial port, write data to the serial port, initialize the serial port, return the number of bytes available at the serial port, and to set a serial port break. The Serial Port Initialize VI allows you to configure the serial port's settings. In order to have successful communications between a serial port and a device, the settings of the port should match the device settings. The settings available are buffer size, port number, baud rate, number of data bits, number of stop bits, data parity and a flow control cluster. This flow control cluster bundles together a number of parameters, including a number of handshaking settings. The Serial VI palette is shown in Figure 5.4.

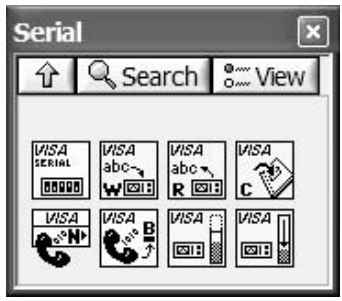

FIGURE 5.4

A programmer using the serial standard must ensure that the serial write does not overflow the buffer. Another issue is making sure all of the data is read from the serial port. There are a number of LabVIEW built-in functions designed to configure the buffer size and to query the number of bytes available at the serial port. Figure 5.5 shows a VI

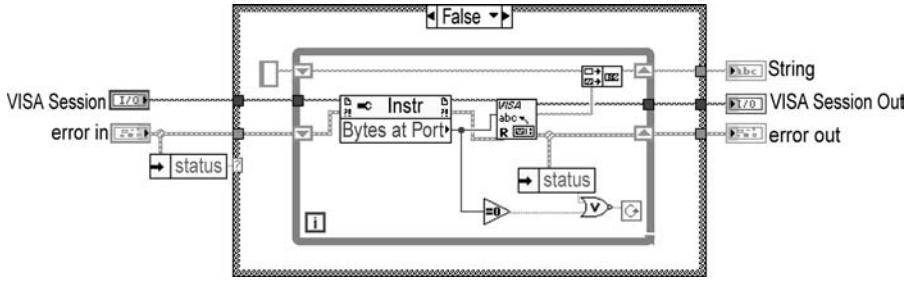

FIGURE 5.5

written to read information from the serial port. This VI performs the read until all of the desired data has been read.

There was a change in LabVIEW serial port communications starting in LabVIEW 7. Whereas the older version of serial port VIs still work and can be found in C:\Program Files\National Instruments\LabVIEW 8.0\vi.lib\Instr\serial.llb, the main serial port VIs in the Serial palette were converted to VISA. In addition to the VISA serial VIs, the legacy serial port VIs were also built on VISA communications. This change requires that VISA be installed with LabVIEW even if your application is only doing serial communications.

There are additional serial port standards, which would require a separate discussion. These standards are the Universal Serial Bus (USB) and Firewire (IEEE 1394). USB allows you to plug devices into a common port, and gives you the ability to "hot swap" instruments. There are a number of hardware devices available that are USB capable. In addition, National Instruments builds devices to take advantage of this technology, including a GPIB-to-USB Controller. This external box connects to the PC through the USB port and allows the user to connect up to 14 GPIB instruments without having to have a GPIB port on the PC. This is especially useful when using a laptop computer without I/O slots; the controller can plug into the USB port. You can also find boxes available for adding serial ports to your computer through a USB converter. The new serial ports look just like a standard serial port to your computer. USB supports transfer speeds of 1.5 to 12Mbps.

Firewire is actually the Macintosh implementation of the IEEE 1394 standard. Firewire is a name brand like Coke or Kleenex. Firewire allows hot-swapping of devices and can daisy-chain up to 16 devices. The main benefit to Firewire is speed. The 1394a Firewire standard boasts speeds of 100, 200, and 400 Mbits/Sec. Revisions to the IEEE 1394 standard (1394b) allow for speeds of 800Mbps, 1.6 Gbps and 3.2 Gbps. Currently, there are only 800Mbps devices commercially available. The 1394b standard also increases the allowable cable length from 4.5m to 100m.

5.1.3 VXI

VME Extensions for Instrumentation (VXI) is a standard designed to support instrument implementation on a card. VME is a popular bus architecture capable of data rates of 40MB/s. VXI combines the speed of the VMEbus with the easy-to-use command set of a GPIB instrument. The goal of VXI instrumentation is to produce

a small, cost-reduced hardware system with standardized configuration and programming. The VXI Plug&Play standards promote multi-vendor interchangeability by standardizing the instrument commands for all VXI instruments. By implementing instruments on cards, the size necessary to implement a test station can be greatly reduced. The ability to implement a number of instruments in a small frame allow the test developer to create a test site in places that were not practical before, freeing up resources for other applications. The VXI standard also gives the user the flexibility of custom solutions. Cards can be made and utilized to implement solutions that are not available off the shelf. The VXI VIs are contained in a subpalette of the Instrument I/O palette.

NI advises against using VXI for new applications. The availability of VXI code is only to maintain compatibility with existing applications. The push is to use VISA functions for instrument communications going forward.

5.1.4 LXI

Due to users' desire to be able to setup existing and new devices without having to use special cables or controllers as well as to be able to reduce cost (specialized VXI modules increase the cost due to lower volume), a new standard is being developed. The LXI standard uses LAN (Ethernet) as the system backbone. This has several benefits including the reduction in cost (no card cages or interface cards), speed, and availability (every computer has a LAN port and many newer instruments already have LAN support as well). This standard is still in the definition phase, but has the support of the major instrument manufacturing companies. I think we will be hearing more from LXI in the future.

5.1.5 VISA Definition

Virtual Instrument Software Architecture (VISA) is a standard Application Programming Interface (API) for instrument I/O communication. VISA is a means for talking to GPIB, VXI, or serial instruments. VISA is not LabVIEW specific, but is a standard available to many languages. When a LabVIEW instrument driver uses VISA Write, the appropriate driver for the type of communication being used is called. This allows the same API to control a number of instruments of different types. A VI written to perform a write to an instrument will not need to be changed if the user switches from a GPIB to a serial device. Only the resource name must be modified where Instrument Open is used.

Another benefit of using VISA is platform independence. Different platforms have different definitions for items, like the size of an integer variable. The programmer will not have to worry about this type of issue; VISA will perform the necessary conversions. Figure 5.6 is a side-by-side comparison of GPIB and a VISA driver.

As is seen in Figure 5.6, the main work in a VISA application is in the initialization. GPIB communications require the address string to be passed everywhere a driver is called. If there were a change in the instrument, like using a serial instrument instead of a GPIB instrument, a large application would require consid-

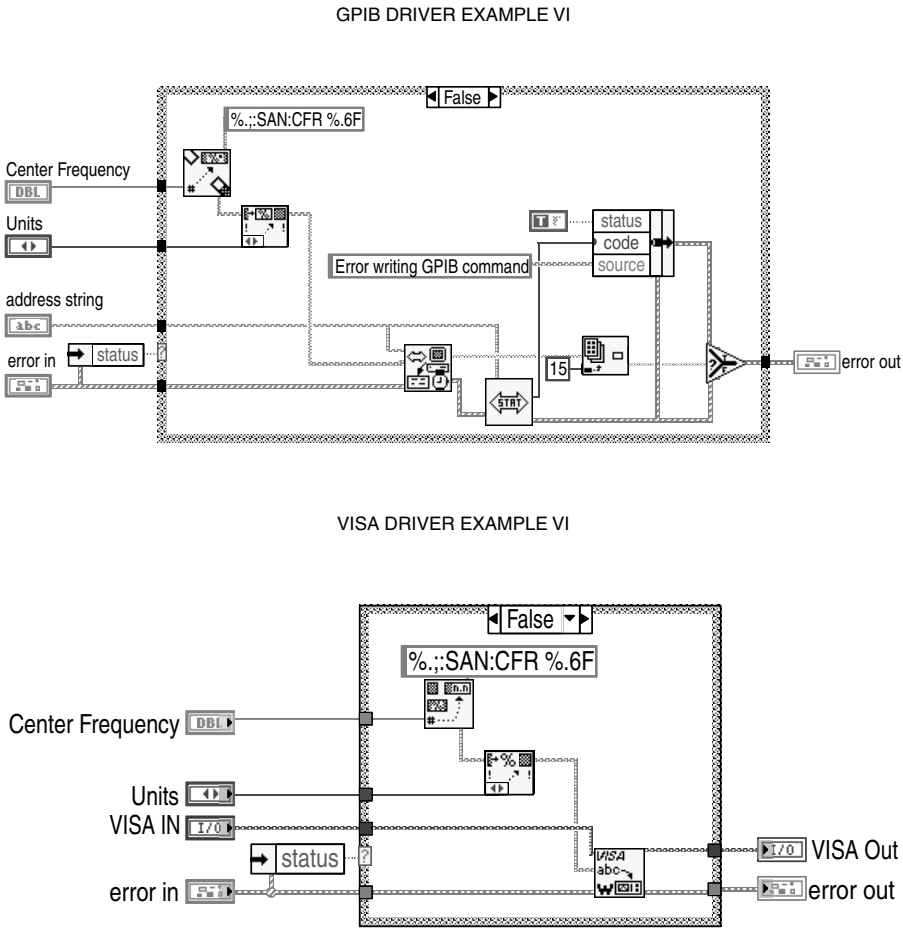

FIGURE 5.6

erable changes. All the drivers would have to be changed. An application using VISA would require changing only the input to the VISA Open VI. The resulting instrument reference would still be valid for the VISA drivers, requiring no change. VISA drivers offer flexibility.

The VISA driver VIs are located in the Instrument I/O section of the Functions palette. The VISA subpalette contains a wide range of program functions. In the main palette are the standard VISA driver VIs. These VIs allow you to open a communication session, read and write data, assert a trigger, and close communications. In addition to the standard VISA VIs, there are a number of advanced VISA functions. These are contained in the VISA Advanced subpalette and three subpalettes on the Advanced palette.

The first subpalette on the VISA Advanced palette is the Bus/Interface subpalette. It contains VIs used to deal with interface-specific needs. There are VIs to set the serial buffer size, flush the serial buffer, and send a serial break. The VISA GPIB

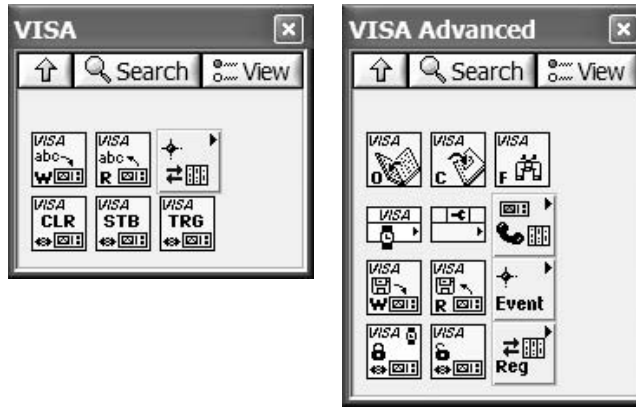

FIGURE 5.7

Control REN (Remote Enable) VI allows you to control the REN interface line based on the specified mode. The VISA VXI CMD or Query VI allows you to send a command or query, or receive a response to a previously sent query based on the mode input.

The next subpalette is the Event Handling palette. The VIs in this palette act on specified events. Examples of events are triggers, VXI signals, or service requests. Finally, the Register Access subpalette allows you to read, write, and move specified-length words of data from a specified address. The Low-Level Register Access subpalette allows you to peek and poke specified bit length values from specified register addresses. The VISA palette is shown in Figure 5.7.

5.1.6 DDE

Dynamic Data Exchange (DDE) is a method of communication between Windows applications. This communication standard is no longer supported in the current versions of LabVIEW. The following text discusses support in earlier versions of LabVIEW.

In DDE communications, there is a server and a client application. The DDE client is the program that is requesting data or sending a command to the DDE server. Assuming both applications are open, the client first establishes communication with the server. Connections are called "conversations." The client can then request the server to send or modify any named data. The client can also send commands or data to the server. A client can either request data or request to be advised of data changes for monitoring purposes. Like the other forms of communication, when all tasks have been completed, the client must close communication with the server.

LabVIEW can act as the server or the client. One example of LabVIEW acting as a client would be a VI that obtains data from an Excel spreadsheet or writes the data to the spreadsheet. If LabVIEW is acting as a server, another Windows program could open and run a VI, taking the data obtained to perform a task.

The DDE VIs are in the Communications palette. There are VIs for opening and closing conversations, and performing advise functions, requests, and executions. In addition to the DDE function drivers, there is a subpalette contained in the DDE palette. This subpalette contains the DDE server functions. These functions are used to register and unregister DDE service and items. There are also VIs used to set and check items.

5.1.7 OLE

OLE, like DDE, is no longer supported in the current versions of LabVIEW. The following text discusses support in earlier versions of LabVIEW.

Object Linking and Embedding (OLE), or automation, is the ability to place objects from other software programs into another application. This ability allows both the expansion of the program's abilities and the ability to manipulate data in another application. An example of this would be taking a movie clip (AVI file) and embedding it in a Word file. Even though Word has no idea what a movie clip is, it can display it in the word processing environment. OLE is a method by which objects can be transferred between applications.

OLE works with objects using a standard known as the Component Object Model (COM). The COM standard defines common ways to access application objects to determine if an object is in use, is error reporting, or if there is object exchange between applications, and a way to identify objects to associate them with specific applications. OLE is a superset of the ActiveX standard and uses the same VIs. There is an in-depth discussion of ActiveX with examples in Chapter 8.

5.1.8 TCP/IP

There are three main protocols for communication across networks: Transmission Control Protocol (TCP), Internet Protocol (IP), and User Datagram Protocol (UDP). TCP is built on top of IP. TCP breaks the data into packets for the IP layer to send. TCP also performs data checking to ensure the data arrives at its destination in a singular, complete form. TCP/IP data consists of 20 bytes of IP information, followed by 20 bytes of TCP information, followed by the data being sent. The TCP/IP protocol can be used on all platforms of LabVIEW and BridgeVIEW.

Every computer on an IP network has a unique Internet address. This address is a 32-bit integer, usually represented in the IP dotted-decimal notation. The address is separated into 8-bit integers separated by decimal points. The Domain Name Service (DNS) system is a database of IP addresses associated with unique names. For instance, a user looking up the National Instruments Web site (www.ni.com) will be routed to the appropriate IP address that corresponds to the name. This process is known as "hostname resolution."

There are a number of standards using TCP/IP that can be implemented using LabVIEW. Telnet, SMTP, and POP3 are a few applications built using the TCP/IP protocol. Telnet can be used for providing two-way communications between a local and remote host. POP3 and SMTP are used to implement mail applications.

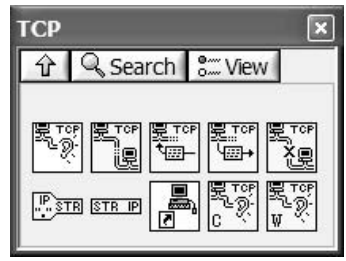

FIGURE 5.8

With TCP/IP, the configuration of your computer depends on the system you are working on. With Windows, UNIX, and Macintosh Version 7.5 and later, TCP/IP is built in. For earlier versions of Macintosh Operating systems, the MacTCP driver needs to be installed.

The TCP palette is located in the Communication section of the Function palette. The VIs in the TCP palette allow you to open and close connections. Once the connection is opened, you can read and write data through the VIs in the TCP palette. There are also VIs to create a listener reference and wait on listener. The IP to string function allows you to convert an IP address to a string. There is an input to this function to specify if the address is using dot notation. A function to convert a string to an IP address is also available. The VIs in this palette are shown in Figure 5.8.

5.1.9 DataSocket

DataSocket is a programming technology that facilitates data exchange between applications and computers. Data can easily be transferred between applications over an Internet connection. DataSocket is built using TCP/IP and ActiveX/COM technologies. The DataSocket server can reside on the local machine or on another machine on the network. You can read data using DataSocket http, ftp, and local files. DataSocket can also read in live data through a dstp (DataSocket transfer protocol) connection. You also have the ability to control your LabVIEW application through a Web interface by using CGI functions with DataSocket. The Shared Variable in LabVIEW 8 is replacing the DataSocket functionality. Support of DataSocket will remain, but new applications should start transitioning to the new variable. The Shared Variable is discussed in detail in other chapters.

The DataSocket VIs are in a subpalette of the Communication section of the Function palette. The DataSocket VIs work in the same way VISA or other standard LabVIEW VIs operate. There are VIs for opening and closing connections. The Open function will open communication based on the URL input and the access mode input. The URL input must be one of the above-mentioned protocols. The output of the Open function is a DataSocket reference. This reference is used in the same manner as a typical connection refnum. The remaining VIs use this reference to perform actions on the desired information. You can then read or write a string, Boolean, integer, or a double value. If you want to read or write arrays of these data types, the necessary VIs are available in the DataSocket Write and the DataSocket

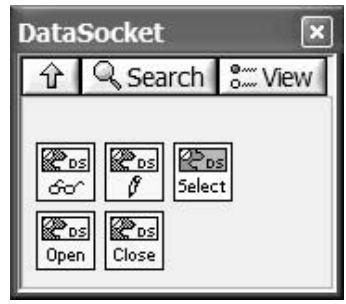

FIGURE 5.9

Read subpalettes. The Advanced subpalette gives you the ability to read or write variants. In addition to the variant functions, there are also low-level functions for performing DataSocket communication. These functions include VIs to connect and update data. Finally, there is a VI to control the DataSocket server programmatically. You should also be able to access the DataSocket server from your Start menu under the National Instruments DataSocket name. The DataSocket function palette is shown in Figure 5.9.

If you want to perform live data updates, you first need to determine if the DataSocket server is running on the local machine. The typical format for a local write data to a DataSocket server is dstp://localhost/test. This assumes that "test" is the label for the data you are writing to the server. If you are using a local server, the DataSocket server will need to be launched through the function in the DataSocket Advanced subpalette. Then, you will need to open a DataSocket connection with Write Attribute selected. You can then write the data you want to share to the DataSocket server. If you are running the DataSocket server on another computer, the machine address will need to be in the DSTP address.

To read the data from the server, you will again need to determine if the server is local or on a remote machine. Once you have the server name resolved, and have a connection open to the server with the read attribute, you can use the Read DataSocket VIs to read the data in. You will need to use the Update data VI if you want to read new data after it has been written to the server.

To read and write static data, the process is the same. The only difference is the URL used to connect to the DataSocket. Examples of a VI used to generate live data to the DataSocket server, and a VI to read the data from the DataSocket server, are shown in Figure 5.10. This example includes additional attributes. This allows items like time and date stamps to accompany the data that is being transferred. The DataSocket server is launched on the same PC as the Data Write VI. There are additional examples in the LabVIEW on-line reference.

5.1.10 Traditional DAQ

Data acquisition (DAQ), in simple terms, is the action of obtaining data from an instrument or device. In most cases, DAQ is performed using plug-in boards to collect data. These plug-in boards are made by a number of manufacturers, including

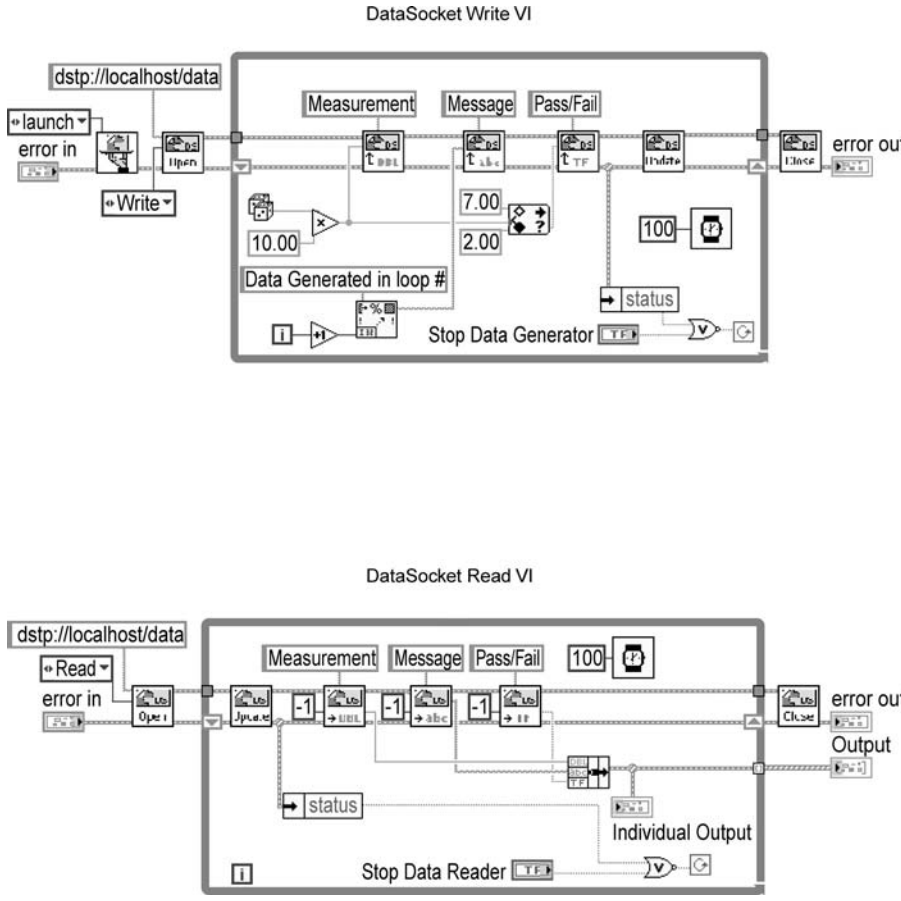

FIGURE 5.10

National Instruments. These DAQ boards perform a variety of tasks, including analog measurements, digital measurements, and timing I/O. One convenience is the ability to obtain boards for PC, Macintosh, and Sun workstations.

One of the benefits of using National Instruments boards is the availability of NI-DAQ drivers for the boards. Although other manufacturers' boards are compatible with LabVIEW, the DAQ library will most likely not be compatible with the board. Most board manufacturers do provide their own drivers for their equipment; some even have drivers written in LabVIEW. Even if the code is not written in LabVIEW, DLLs can be implemented by using the Call Library function. Code Interface Nodes (CINs) can be used to implement drivers written in C source code.

The Data Acquisition subpalette is a part of the Functions palette. The Data Acquisition palette is made up of six subpalettes: the Analog Input VIs, Analog Output VIs, Digital I/O VIs, Counter VIs, Calibration and Configuration VIs, and Signal Conditioning VIs. The Data Acquisition subpalette is shown in Figure 5.11. Each of the subpalettes is comprised of a number of VIs of varying complexity and

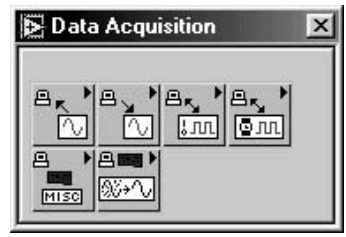

FIGURE 5.11

functionality. There are four levels of DAQ VIs. They are Easy VIs, Intermediate VIs, Utility VIs, and Advanced VIs. As a rule, the Utility VIs are stored in their own subpalette. The Advanced DAQ VIs are also stored in their own subpalette. The main difference between the Easy VIs and the Intermediate VIs is the ability of the Easy VIs to run as stand-alone functions. These VIs call the higher-level VIs to perform the task. The Easy VIs allow you to pass in the device number and channel numbers. The VIs will also perform error-handling functions to alert you if an error has been encountered.

Let's look at the Analog Input subpalette. The palette consists of the four types of VIs described above. The Easy VIs include functions to acquire one or multiple waveforms from an analog input. There are also functions for acquiring samples at the designated channels. The Intermediate VIs allow you to configure the hardware and associated settings, start an acquisition, read the buffered data, make single scan acquisitions, and clear the analog input task. The Analog Input palette contains two subpalettes. The first subpalette contains the Utility VIs. These VIs include functions to initiate a single scan, a waveform scan, or a continuous scan. The second palette contains the Advanced functions. The Advanced functions palette contains VIs to perform configurations, read the buffer, set parameters, and control analog input tasks. We could devote a number of chapters on DAQ functions, but the DAQ functions are described in great detail in the NI Data Acquisition Basics manual. We will not attempt to cover material that is concisely covered already.

5.1.11 NI-DAQmx

Whereas the traditional DAQ (Legacy) VIs have been used successfully to automate data acquisition applications for more than a decade, support of new capabilities and functional improvements resulted in the creation of DAQmx. DAQmx is a superset of the DAQ Legacy VIs. You can still do all the functions that you used to be able to do, but now you have additional features such as multithreaded execution, additional driver functionality and configuration applications like DAQ assistant and express VIs. DAQmx is available for Windows and Linux operating systems. NI-DAQmx Base was created to provide a subset of DAQmx functionality for MAC OSX, RTX and Pocket PC operating systems.

The first benefit of DAQmx is the support for newer devices. New devices for data acquisition are continually created with added functionality over older model devices. In order to support the new functionality new drivers will need to be created.

The new drivers will be created for DAQmx only. Although the support for traditional NI-DAQ will continue, no new drivers will be created. In order to continue to develop applications it is advised to use DAQmx in order to make sure the code will continue to be supported.

In most cases DAQmx improves application speed. This is due to a couple of new features. First, the legacy drivers ran only in a single thread. DAQmx now supports multithreaded execution speeding applications that can do two or more acquisition tasks in parallel. The second improvement in speed is by application design. Now with more control over operations such as reserving resources and configuration so that the user application can be designed to perform these operations only when needed to reduce expensive overhead.

Finally, DAQmx tools can make application development easier. The ability to use the express VI for configuring an acquisition task can shorten the amount of time needed to get a test running. This can be valuable when the test program will change often, and having to recode a VI each time a new test is needed could be tedious. The DAQ Assistant is an application that can make coding easier by walking you though each step of building an acquisition task step by step. An example will now be shown for a simple read of information from an analog voltage input from a DAQ card.

The first step for the example is to set up the hardware. The hardware setup is done in the measurement automation explorer (MAX). Under the Devices and Interfaces folder all available hardware should be shown. Any installed DAQ cards should show up here. If you are developing the application at your desk before installing it on your acquisition system, your device will not be here. You can add a device with the drivers for your hardware, or you can create a simulated device in order to be able to develop and test your application. You can do this by right clicking on the folder and selecting Create New. Here you can select the appropriate drivers. Once you have the device selected you can configure settings such as initial settings and connector type. For this example we have created a simulated DAQ card (PCI-6220) at DEV1. The MAX window is shown in Figure 5.12.

Now on the code diagram the DAQ Assistant will be selected from the input palette under the Express Palette. The initial DAQ Assistant interface that automatically launches is shown in Figure 5.13. For this example we are going to select Voltage under the Analog Input heading. After selecting the type of acquisition the input screen gives you the options to set up what hardware channels to use. The available information is based on what has been already set up in MAX. Here I am able to select specific analog input lines on DEV1 (the simulated 6620 card).

Now that you have completed the initial setup the DAQ assistant opens a window for configuring the channel parameters. This interface is shown in Figure 5.14. Here you can add, remove and test the channels to make sure you have all the settings needed. There is another tab on the bottom of the window. This tab will show the connection diagram. Here you can select each channel and see what wires are connected based on the defined DEV1 connector in MAX. This window is shown in Figure 5.15. Once configuration is complete the resulting DAQ block will be on the code diagram. The block is the standard Express VI block with the blue border. Now you can wire controls to any inputs you want to be able to change. The output

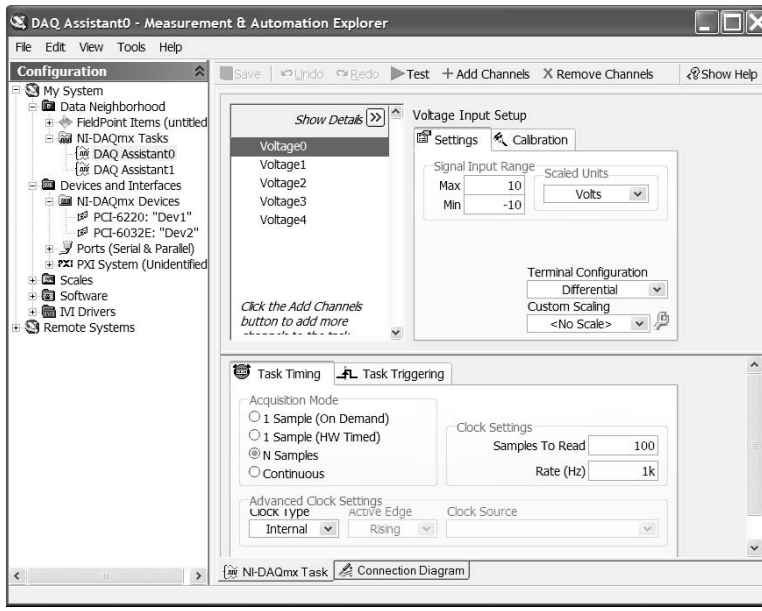

FIGURE 5.12

FIGURE 5.13

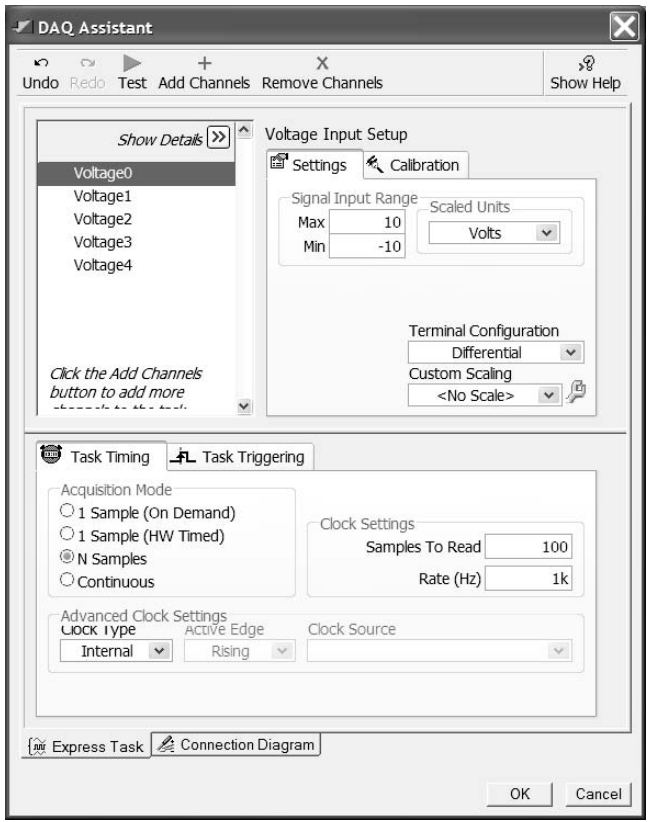

FIGURE 5.14

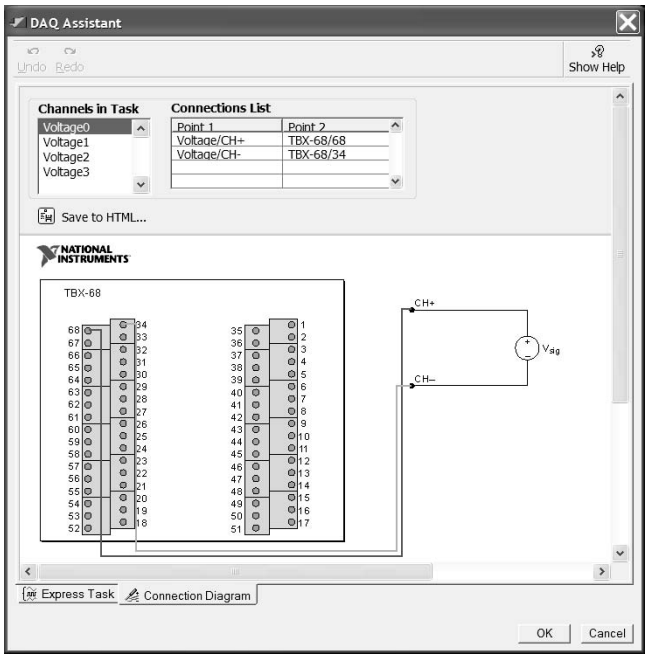

FIGURE 5.15

waveform can be connected to the DAQ Assistant block output. Running the VI produces a graph of the input channels.

Some people do not like to use the express VIs because the code is hidden from the user. You have two options. You can wire the application using the DAQmx subVIs the same way you would use the Legacy DAQ VIs or you can configure the application with the DAQ assistant above and choose to generate DAQmx code by right clicking on the icon. Figure 5.16 shows a simple single channel analog input acquisition. Notice the DAQmx VIs are polymorphic. A polymorphic VI is a VI that can perform different functions based on an input value. In this case there is a VI for creating a channel. In this instance analog input is selected, so the VI will create an analog input voltage channel. If the programmer changed the text below to read analog output voltage (AO Voltage) an analog output voltage channel would be created. This setting can be changed by right clicking on the text selector at the bottom of the VI as is shown in Figure 5.17. This is another way to make programming easier. The same code can be used for different applications by changing the functionality of the polymorphic VIs without having to insert a new VI.

5.1.12 File I/O

File input and output is a type of driver that people do not often think of. The ability to read data from a file and write data to a file in many ways is similar to reading data from and writing data to a GPIB instrument. You require a means to identify the file you want to communicate with. Instead of a GPIB address you have a file path. You also need to be able to transfer data from one place to another. Instead of passing data between the computer and the GPIB instrument, you are passing data between the LabVIEW program and a file. The File I/O functions are very similar to instrument or communication drivers.

The File I/O VIs can be found in the File I/O section of the Function palette. This subpalette contains a number of file functions as well as subpalettes containing VIs pertaining to binary files, file constants, configuration files, and advanced file functions. The standard file I/O functions include VIs for opening/creating a file, reading data from a file, writing data to a file, and closing a file. In addition to these functions, there are VIs for writing and reading data from a spreadsheet file, writing or reading characters from a file, and reading lines from a file. The File I/O palette is shown in Figure 5.18.

There are two remaining functions that are included with the standard file I/O functions. The first VI allows you to build a file path. This VI creates a new file path by appending the file name or relative path from the string input to the base path. The default value of the base path is an empty path. The result is the combined file path. If there is a problem in one or both of the inputs, the VI will return "not-a-path." The second function takes a file path and breaks it apart. The last section of the path is wired out as a string filename. The remainder of the path is wired out as a path. The VI will output an empty string and "not-a-path" if there is an invalid input. The binary file VIs allow you to read and write 1- or 2-D arrays of data to a byte stream file. The byte stream file can be in a signed word format or a single precision format. The configuration file palette contains VIs used to read and modify

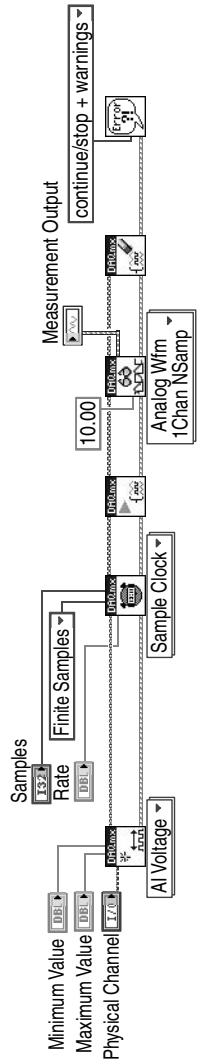
Measurement Output
continue/stop + warnings
Error
10.00
Analog Wfm
1Chan NSamp
Finite Samples
Sample Clock
Samples
Rate
AI Voltage
Minimum Value
Maximum Value
Physical Channel

FIGURE 5.16

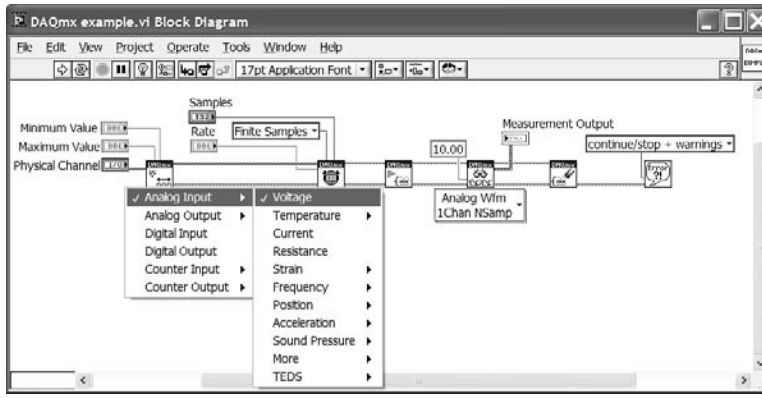

FIGURE 5.17

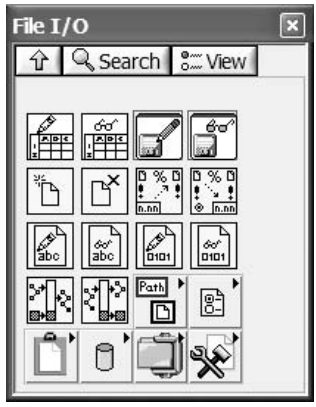

FIGURE 5.18

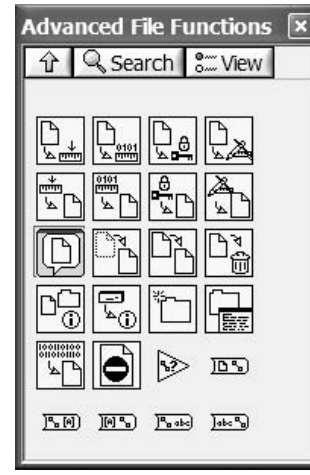

FIGURE 5.19

information in the configuration files. The File Constants palette contains VIs that allow you to access the current directories, paths, or VI library directories. In addition to these functions, there are constants that can be used to create inputs to the file I/O VIs.

The Configuration Files Palette contains VIs used to read from and write to INI formatted files. These VIs can be very useful when requesting configuration information from a user to set up the code execution. Using these VIs you can save the settings that the user had entered to be loaded the next time the application is run so that the user does not have to reenter all the information. There could even be options to save configurations in the event there may be multiple setups needed.

The Advanced palette contains VIs that perform a number of file-related tasks. The Advanced palette is shown in Figure 5.19. The File Dialog function displays the file dialog box for the user to select a file. The output is the path of the file

selected. The Open File VI allows you to specify a datalog type. There is a function used to find the offset of the end of file (EOF). The seek function allows you to begin a file in a position other than the beginning of the file. There are VIs used to set access rights for a specified file, as well as to find out information on the file, directory, or volume.

There is a set of five VIs in the Advanced palette that performs actions on directories. There is a VI that allows you to move a file or directory. There are also VIs that allow you to copy a file or directory, as well as delete a file or directory. The New Directory function allows you to create a directory at the specified path. The List Directory function lists all of the file names and directory names that are found in the directory path.

The final set of functions in the Advanced palette are VIs used to convert between strings and paths. The functions can perform the functions on a single string or an array of strings. There is also a VI that converts a refnum to a path. These VIs are useful when converting string paths created by the user in a user interface to a file path to perform file functions.

We will now give a quick example of how to read and write data when dealing with datalog files. The first step is to create the data type used for storing the data. For this example we will be recording three distinct values per datalog value. The first is the index of the data. This is simply the value of the For loop index used to create the data. The second item in the data cluster is the data. The data for this example is simply random numbers generated between 0 and 10. The final data type used for the cluster is a date and time stamp. This value is written as a string. To summarize, our data type consists of an integer, a real number, and a string.

The first step is to create the code to perform the data generation. The For loop executes 100 iterations. Inside the For loop, the loop index, the test data, and the time and date string are bundled into a cluster. This cluster is wired to the output of the For loop, where auto indexing is enabled. When all the data has been collected, the New File VI is used. The File Path contains the name and location of the file you are writing the data to and will be needed when you want to retrieve the data. The file path is the only required input. There are a number of other inputs to the VI that can be wired, or left as default. To write and read datalog files, you will need to wire a copy of the data format to the datalog type. Wiring the actual data to the input, or wiring a constant with the same data type, can do this. The other inputs are permissions, group, deny mode, and overwrite. The overwrite input for our example will be given a "true" value. This allows the program to overwrite an existing file with the same name as specified in the file path input. If the input were "false," the program would error out when trying to create a new file that already exists.

Once the file is created, the next step is to write the data out. The Write File VI is used to send the collected data to the datalog file. The inputs of the Write File VI include convert eol (end of line), header, refnum, positive mode, positive offset, error in, and the data. The only required inputs are the refnum and data inputs. The data from the For loop is wired to the data input. The final step of this subVI is to close the file using the Close File VI.

The next step is to create a VI to read the data back from the file. In this VI, the Open File function is used to create a connection to the file. The File Path input

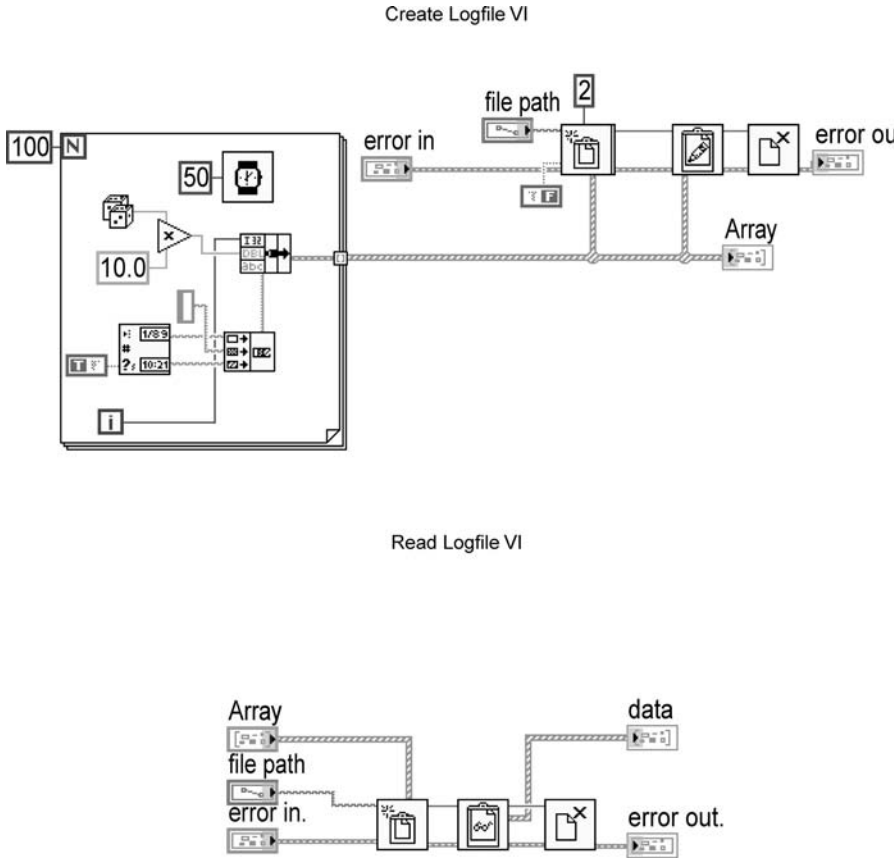

FIGURE 5.20

is used to point the VI to the datalog file. In addition to the file path, the data type is wired to the Datalog Type input. This data type needs to match the data type of the cluster we wrote to the file. This allows you to read the information back in the appropriate format. In addition to the datalog type and file path, you can set the open mode and deny mode for the file. This allows you to determine the file permissions. Once the file is opened, you need to use the Read File function. This VI is used to acquire the data from the file, and write the data to an indicator. Again, the final step is to close the file. The code diagram for the Datalog Write VI and the Datalog Read VI is shown in Figure 5.20.

5.1.13 Code Interface Node and Call Library Function

LabVIEW has the ability to execute code written in C as well as to execute functions saved in a DLL. There are two methods for calling outside code. The programmer can call code written in a text-based language like C using a Code Interface Node (CIN). The programmer also has the ability to call a function in a DLL or shared library through the use of the Call Library function. A short description of each will follow.

The CIN is similar in some respects to a subVI. The CIN is an object on the block diagram of a VI. The programmer can enter inputs required to execute a function, and wire the outputs of the CIN to the remainder of the program. The main difference is a subVI is code written in the G language to perform a function, whereas the CIN executes text-based code to perform the function. The CIN is linked to compiled source code. When the execution of a block diagram comes to the CIN, LabVIEW calls the executable code, returning the final outputs to the VI.

There are a number of reasons for using the Code Interface Node. One benefit is the ability to use existing code in your LabVIEW program. If a function is already written in C, you have the ability to integrate the code into your LabVIEW program to reduce development time. Another benefit to using a CIN is to expand the functionality of LabVIEW. Certain system functions that do not have corresponding LabVIEW functions can be implemented using code written in C. This can help a programmer to perform low-level programming with LabVIEW's graphic-based interface. A final consideration for using CINs is speed. Whereas LabVIEW is fast enough for most programming tasks, certain time-critical operations such as data acquisition and manipulation can be done more efficiently in a programming language like C. The use of the CIN allows the programmer to use the right tool for the right job.

The ability to use prewritten code is a key to reducing development time. Functions to perform many Windows functions have already been written. These functions are typically written in C, and are stored in Dynamic Link Libraries (DLLs). LabVIEW can call these Windows functions in two ways. The first way is through the use of a Code Interface Node. An easier method for calling DLL functions is through the use of the Call Library function. The main difference between calling C code in a CIN and using the Call Library function to call a DLL is the integration of the source code. When using a DLL, the code remains in its library; it is not copied into the executable files of the application. The other obvious difference is the fact that DLLs are Windows-specific, whereas the Code Interface Node can be used across platforms.

For more information on the Code Interface Node, the Code Interface Reference Manual can be found on National Instruments' Web site. The PDF file covers how to integrate a CIN on any platform. For information on using DLLs, there is an application note on the NI Web page. Application Note 087, "Writing Win32 Dynamic Link Libraries (DLLs) and calling them from LabVIEW," discusses the methods for using DLLs.

5.2 DRIVER CLASSIFICATIONS

There are three main functions a driver performs. The three types correspond to the three main purposes of a driver: configure an instrument, take a measurement, or check the status. These three main types of drivers will be discussed below. When creating driver VIs, National Instruments recommends a standard format the drivers should follow. Driver libraries should contain the following functions: Initialize, Configure, Action/Status, Data, Utility, and Close.

5.2.1 Configuration Drivers

The first type of driver is a Configure VI. These VIs should open or close communications with the instrument, initialize the instrument, or configure the instrument for the desired use. The Initialize driver first performs the initial communications. This should include opening a VISA session if VISA is being used. The Initialize driver can also perform instrument setup and initial configurations. This can allow the instrument to begin in a known or standard state. The Configuration Instrument drivers send the necessary commands to the instrument to place the instrument into the state required to make the desired measurements. There may be a number of configuration VIs for a particular instrument, logically grouped by function or related purpose. The Close driver closes the instrument communication, the VISA handle, and any other required items to complete the testing process. It is important to close the instrument communications, especially when doing serial and TCP communications. When a serial port is open, no other applications can use the port. If the port is not closed, the port is inaccessible until LabVIEW is closed. With TCP, when you connect to another machine, the port on that machine will stay open unless you close the session or the session time out.

5.2.2 Measurement Drivers

Measurement drivers are used to take measurements or read specific data from the instrument. The user should be aware that a data driver does not always require reading data from an instrument. The data driver could also be used to provide data to an instrument, like sending a waveform to a signal generator. It is important to note that only one measurement should be taken per driver. This is done to promote reusability as well as to ensure the application speed is not compromised by taking unneeded measurements.

5.2.3 Status Drivers

The action/status drivers are used to start or stop a specified process, check errors, and general instrument-related information. One example would be a VI written to start and stop a Bit Error Rate (BER) test or a waveform capture from a spectrum analyzer. Another example is checking a status register to find out if a test that has been initiated is completed so the result can be read from the instrument. The VI would not change any of the instrument configurations, only the initiation or termination tasks are performed. As checking the status of an instrument can require the instrument to be reset, a set of utility drivers should also be designed. The utility drivers are used to perform tasks such as reset, self-test, etc.

5.3 INPUTS/OUTPUTS

An important aspect of a driver is the interface with the calling VIs. There are a number of standard inputs and outputs for drivers. The Error In and Error Out clusters are the most important I/Os in a driver. These clusters have three components. For

the Error In cluster, the first control is a status Boolean control; a "true" indicates there is an error. The second is a numeric control to display an error code. The final control is a source string. This string can indicate where an error occurred. There are two primary reasons for using the Error In and Out clusters. The first reason is obviously error handling. If an error has already occurred in a program, the Error In cluster will pass this information to the driver, preventing the execution of the intended task. The error cluster can also pass error information out of the driver if an error occurred while the driver was executing. A discussion of error handling is described in the following section.

The second reason for using the Error In and Out clusters is flow control. The wiring of the Error Out of one VI to the Error In of another forces the order of execution because of data dependency. For example, an instrument needs to be configured prior to taking a measurement. Wiring the Error Out of the configuration driver to the Error In of the Measurement driver forces the order of execution.

The other required inputs are the instrument communication handles. Depending on the communication VIs being used, a number of different inputs could be used. We suggest using VISA standards in your drivers. This will allow the same driver format regardless of what type of communication is used to address your instrument or device. The standard method for wiring the connector pane has the VISA session in and out in the top left and right positions, respectively. The Error In and Out are in the bottom left and right positions, respectively. This consistency of location makes connections easier to wire and find.

For readability and ease of use, the programmer should use as few inputs and outputs to a driver VI as possible. The use of clusters should be avoided unless the information is packaged in a form that other subVIs would use like the error cluster. If the cluster is not passed on, the main program will need to bundle and unbundle the items. This can obscure the intention of the code and complicate the code diagram. Additionally, the complex data type will have an effect on performance.

5.4 ERROR HANDLING

Error handling is one of the most important considerations when a programming task is begun. For this reason there is an entire chapter in this book dedicated to error handling. This section will just highlight some of the driver-specific error-handling issues.

The main error handling that should be performed in the driver is the detection of errors that are passed in. If an error is passed into a driver, the driver should not execute any tasks. The driver should consist of a case statement controlled by the status field of the error cluster. The driver code would then execute only if no error passed in. When an error is passed into a driver, the instrument communication VIs will not execute if an error cluster is passed to them. Error processing should only occur in the upper levels of the program, as prescribed by the three-tiered design architecture. The benefit of not processing errors in the driver is the ability of the driver to be reused. If error processing is performed in the driver, the results of the processing may not be applicable to a new program using this driver. Doing error

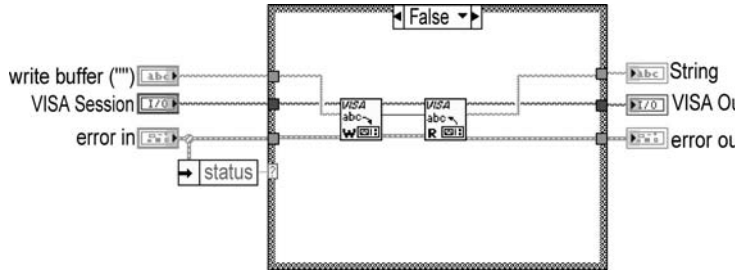

FIGURE 5.21

processing in the driver would cut down on code reuse. An example of the use of this "bypass" is shown in Figure 5.21.

The next issue with error processing in drivers is the implementation of timeouts. A driver should have a way out. If a driver is written to read the status of a register through the use of a While loop to read the data from the device, there should be a way to exit after a specified time if the desired response does not occur. This can result in setting an error if the program will not function without the desired value.

In writing applications that read data from a device, you should add code to ensure that any errors that occur during the data acquisition are handled in an appropriate manner. For example, assume you are reading data from a serial instrument. In this example you are reading the information from the serial port until the desired data is read. To perform this task, the read operation is in a While loop that is executing until the desired input is read. When the desired input is received, a "false" Boolean is wired to the conditional terminal of the While loop. If an error would occur, the desired input would never be received, resulting in the While loop continuing to execute until you stop the application. You should check the Boolean value of the error cluster in each iteration of the While loop to check for an error. The result of this error check can be combined with the result of the data check to determine whether to execute another iteration of the While loop. The Boolean from the error cluster and the data check can be combined through the Boolean logic functions to control the conditional terminal of the While loop. An example showing all three of the above-mentioned techniques is shown in Figure 5.22.

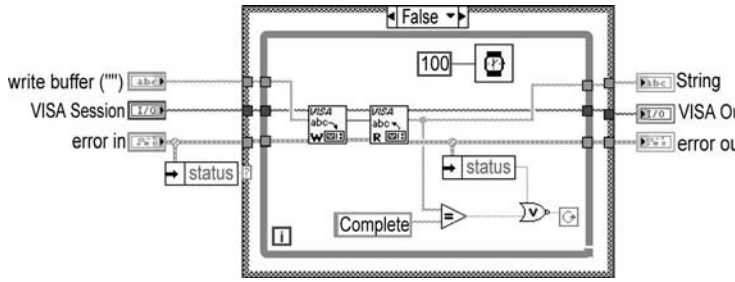

FIGURE 5.22

One type of error detection that should be mentioned is the ability to set error traps in the driver code for debugging purposes. During the development stages of a driver, "traps" can be put in place to trap and isolate errors. This can lead to faster error detection for the purpose of debugging the driver being developed. These traps can be either disabled or removed when the driver development has been completed. Some instances of error traps can be simply collecting the data being read in from a serial port, and saving the data to be reviewed by the developer. As some errors will only occur when running at full speed, recording the data for later analysis could be of great benefit. The recording of this same data would be considered unnecessary in the final driver version, hence the need for an error trap. Once the driver has been fully debugged, the trap can be eliminated. Data logging, discussed in the error-handling chapter, is a similar tool that allows you to save and view data after the VI has been executed.

When measurements are being made in a loop, or setup is being performed in a state machine, care needs to be taken with error handling. There should always be a shift register passing the error cluster to each iteration. When this is forgotten, errors become difficult to track because the error cluster gets cleared with the next iteration of the While or For loop.

5.5 NI SPY

It is difficult at times to debug drivers. Commands are sent to the instrument by the program, but are the parameters correct, how long do the calls take, is there a problem with the instrument, etc.? The developer performing the application debugging needs a way to monitor and verify that the program is doing what was intended. One tool provided by National Instruments can aid in code verification. The NI Spy utility is an application that monitors, records, and displays API calls made by National Instruments applications. The NI Spy can be used to locate and analyze any erroneous API calls that your application makes, and to verify that the instrument communication is correct.

5.5.1 NI Spy Introduction

The NI Spy program is similar to a GPIB analyzer. The NI Spy displays function call names, parameters, and GPIB status as the developer's program executes calls. The NI Spy allows access to information like the contents of data buffers, process and thread IDs, and time stamps for the start and finish times of the function calls. The spy program can also create a log of the information, although this can produce a significant performance loss.

5.5.2 Configuring NI Spy

The first step is to open the NI Spy program. If you go to the Start menu of your computer and then to the Programs folder, there should be a folder labeled National Instruments and there should be an icon for the NI Spy. When this icon is selected, the window shown in Figure 5.23 comes up. In the title bar, the name "NI Spy"

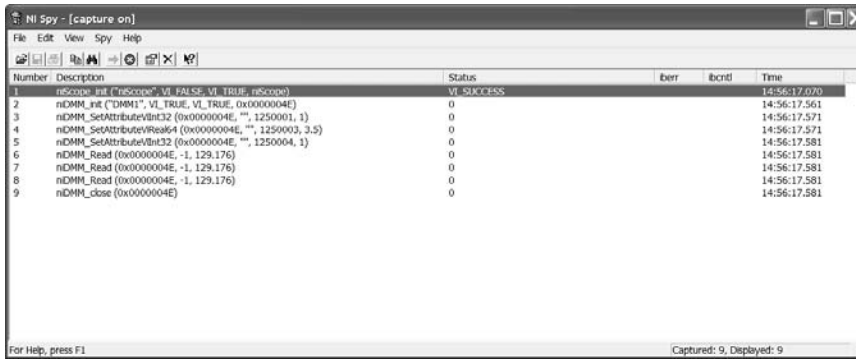

FIGURE 5.23

should appear followed by the program's status. In parentheses, the title bar will indicate whether capture is on or off. By default, Capture is off when you open the NI Spy application. Figure 5.23 shows the NI Spy window with Capture on.

Before starting the NI Spy program, the first step should be to configure the options for the application. By selecting the Spy menu, the following options are available to you: Start Capture, Options and Calculate Duration.

To modify the NI Spy capture options, select Options from the Spy menu. The NI Spy options can be modified only when Capture is off. NI Spy, by default, displays 1000 calls in the Capture window, displays Small Buffers and does not enable file logging. The Call History Depth option identifies how many API calls the NI Spy will display. If more than the selected number of API calls are made, the Capture window will show the most recent calls, discarding the calls at the beginning. If the NI Spy program is unable to display all of the API calls due to low system memory, a message box will appear giving the user the option to stop the capture or free up system resources before continuing.

The Buffer Limit per Parameter selection allows you to choose between Small or Large Buffer mode. The Small Buffer mode displays up to 64 bytes of data, whereas the Large Buffer mode displays up to 64K bytes of data. For either of these modes, if there is more data than the allowed buffer, the middle data will be removed. For example, in the Full Buffer mode, the first 32K bytes and the last 32K bytes of data will be displayed. A row of dashes between the two halves of the buffer is inserted to indicate that part of the data has been omitted.

The File Logging selection in the NI Spy options allows the program to record all calls to a log file. File logging is useful when debugging an application that causes the system to crash. If file logging in the Fail-Safe Logging mode, you can view the API calls that were captured prior to the system crash by opening the saved log file. In order to use this function, a file name must be provided to store the logged API calls. There are two modes of file logging available. The first is Fail-Safe Logging. Fail-Safe Logging is a method of guaranteeing that the log file will not be corrupted if the system crashes. The logging is accomplished by opening the log file, writing the data, and closing the log file after each API call. It should be

obvious that this method of logging the data is slow. If performance and time are an issue, Fast Logging is available. This method of logging opens the file at the start. The data from each call is written to the log file when the call is captured. The file is not closed until the capture is stopped or logging is disabled. The Fast Logging method of file logging is much faster than Fail-Safe Logging, but if your system crashes, data will be lost.

If you have more than one National Instruments driver installed on your computer, you can specify which APIs you want to spy on at any time. The View Selections tab shows the API choices that are available to monitor. You also have a choice of what columns to display. Types of National Instruments drivers are GPIB-488.2, VISA, and IVI-type drivers. By default, all installed APIs are enabled. There will be a check next to the API types selected for capture. You can omit any driver on the list by clicking on the name; the check will be removed.

Finally, there is an Error tab for specifying what to do if an error occurs. You have the option of ignoring errors, stopping if any error occurs or stopping only on specific error parameters. The ability to pinpoint the specific breakpoint is a big plus when trying to isolate an elusive fault.

5.5.3 Running NI Spy

There are three ways to start capturing API calls. The first is to select Start Capture from the Spy menu. The second method is to click on the arrow button on the toolbar. Finally, the user can press F8 to turn Capture on. Once you turn Capture on, you can run your application. When you want to view the captured information you can return to NI Spy to view the captured calls. To turn Capture off, click on the red "X" button on the toolbar.

You can view the API calls in the main NI Spy window as NI Spy captures them. The captured API calls are displayed in the order in which they are received. There is one line of information displayed for each captured call. The information includes the number of the call, a C-style function prototype, and the start time for the call.

By using the Properties dialog box you can see detailed call information for every captured API call. To see the properties of a specific call, double-click on the call in the Capture window, right-click on the call and select properties, or select Properties from the View menu. The Properties dialog box includes one to five pages of detailed information on the captured call. All API captured calls have a General tab, most captured calls have Input and Output tabs, some captured calls have a buffer page, and some IVI captures can have an Interchange Warning tab. The General section displays the process and threads IDs, the Windows handles, and the start and stop time statistics. The Input page displays the API call's input parameter types and values. The Output section displays the parameters that were returned after the call completion. The buffer page is present only for calls that involve the transfer of a buffer of data; this page displays the contents of the data buffer. Finally, the Interchange Warning section displays warnings about the specific call with respect to instrument interchangeability. This option is available for IVI drivers.

To search through the list of captured calls to find a specific string in the API function names, parameter values, or any other string, select Find from the Edit menu. Enter the text that you want to search for in the Find What box. Click the Find Next button to find the next captured call containing the specified string. The Match Errors Only selection can be used to limit the search to captured calls that have an error. If no search string is specified, the search locates the next captured call that failed. The Match Case selection specifies whether the search is case sensitive.

5.6 DRIVER GUIDELINES

Aside from the general driver information, there are a number of implementations that can add robustness and reusability to a driver. This section will give an overview of some of the functionality that should be added to a driver to accomplish the desired results.

One guideline that should be followed is the method of making only one measurement per driver. Since the programmer will want different measurements at different times, the programmer should keep one measurement to a driver. This allows the code to be reused easily. The user of the driver will not have to take a number of measurements in order to receive one desired value. Making multiple measurements when only one measurement is desired limits performance.

When developing a driver, the programmer should try to combine configuration settings into logical groups. If configuring an RF generator requires setting four different parameters every time, the configuration of those parameters should be in a common driver. This would allow the user to set the generator with the appropriate settings through the access of one driver.

When you are linking the controls and indicators to the connector panel of the icon, you should choose a connector configuration that will provide extra connectors. When all of the inputs and outputs have been wired, extra connectors allow for expansion without disconnecting all existing connections. When a driver is already called in a program, and if the programmer adds a new input or output, the user will not have to rewire all of the existing connections. When there are extra connectors, the existing connections do not change, allowing the current wiring to remain unchanged.

5.7 REUSE AND DEVELOPMENT REDUCTION

The biggest benefit of developing quality drivers is the ability to reuse the drivers. Even when the programmer does not expect to use a specific driver again in the future, things change quickly. There is no better feeling in software development than, when developing an application, you realize that the underlying code has already been written. If a driver has been properly written, applications that are completely different could still use the same driver. The ability to reuse code is the biggest factor in cycle-time reduction. By not having to rewrite drivers, which includes time to learn the equipment, coding, and debugging, the user can dramatically reduce the time required to develop an application. Making drivers generic

enough to reuse can require more time and effort up front, but the benefits that can be realized are substantial.

There are many drivers for numerous instruments and manufacturers that have already been written. The first place you can look for an instrument driver is on the installation CD that came with your LabVIEW application. The second disk is a disk of instrument drivers. In addition to these drivers, many of the drivers are available on the National Instruments Web page. Not only is this resource a comprehensive list of drivers, but also they are the most recent versions. The National Instruments ftp site is ftp.ni.com. Your login is "anonymous" and your password is your Internet address.

Many drivers available on the National Instruments Web page have been submitted to NI and accepted for distribution. There are standards to which NI requires all drivers submitted to adhere. Many of the standards have already been discussed, and these standards can be found in the application note, AN106. As the drivers have already been designed to the required standards, they should be easily inserted into your application with no modification. This allows the programmer to concentrate on developing the application without concern about the underlying communications. This can lead to significant development time reduction.

For unusual or difficult-to-find instrument drivers, there are some other resources available. The LabVIEW Info Group is a place you can try. The Info Group is a large knowledge base that you can utilize. For subscription requests you can send an e-mail to info-labview-on@labview.nhmfl.gov. To post a message to the Info Group, send an e-mail to info-labview@labview.nhmfl.gov. There are also some other user groups such as LAVA (LabVIEW Advanced Virtual Architects). There are discussion groups as well as code examples available at the LAVA Website. The LAVA address is lavausergroup.org.

5.8 DRIVER EXAMPLE

To tie together some of the driver techniques and guidelines, we will present an example set of drivers. This set of drivers will communicate with Microsoft Word using ActiveX. This example will create only a couple of relevant drivers for illustration purposes. If you want more information on ActiveX, Chapters 7 and 8 will give a detailed description and numerous examples.

The fist step is to define the task we want to accomplish. We will want to open Word, create a new file, set the margins, set the page size, set the page orientation, write text to the file, save the file, and close Word. The first step is to identify the driver types needed. You will need configuration drivers and measurement drivers. As configuration drivers perform instrument communication and configuration, the VIs needed to open Word, close Word, and configure the settings will be contained in these drivers. The action of reading or writing data to an instrument or application requires measurement VIs. The write text to file will fall into this classification.

A driver to open an automation reference to Word will need to be created. This action will be combined with the creation of a new file. This allows the user to open Word with a new document in the initial step. The next driver to be created will configure the page setup parameters. Most times when you are modifying a one-

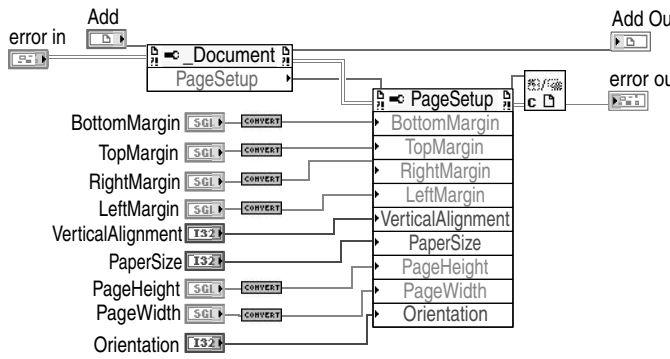

FIGURE 5.24

page setup parameter, you will want to modify additional page setup parameters. This is a good place to combine the configuration settings into one subVI to facilitate ease of programming. Not only will the programmer be able to see all of the input parameters that can be changed in one location, but the driver can also ensure order of execution. Some of the page setup parameters need to be modified after other parameters have been set. For example, you need to modify the page style prior to setting the orientation. The orientation setting will be reset after modifying the page style. If you are placing individual VIs to set these parameters, you could forget or be unaware of certain data dependencies, causing parameters to not be set in the desired manor. The code diagram for the Page Setup Configuration VI is shown in Figure 5.24. In addition to the data dependencies there are issues with data conversions. For example, when writing a value to a margin input, you would attempt to write data in inches. However, to get a margin value of one inch, a 72 needs to be wired to the input of the property node. Inside the driver, there is a function to convert an inch input to the required automation input. This allows you to abstract this information from the person using the driver.

The Write Text VI takes a string input and inserts it into the file at the specified index. If making multiple write statements, you could wire the end value from the previous write to the start value of the current Write VI. This allows you to do incremental data storage in the file. You would only want to have this VI write the text to the file. Any additional functions added to this VI would limit your ability to reuse the VI. For example, if you wanted to perform a spell check on the document, you would have to perform this spell check each time text is written to the file. You may want to check the spelling only after all of the text has been written to the file. If the spell check function is in its own VI, you can invoke this function when you need it. There is also the possibility you do not want to perform a spell check at all. Measurement VIs should be in their own VIs unless you are sure you will always want to do the multiple tasks together. An example using these VIs is shown in Figure 5.25. In the example, Word is opened; a new file is created (testfile); some of the page setup parameters are modified; two strings are written to the file, separated by a time delay; and the file is closed. More information on controlling Microsoft Word using ActiveX is included in Chapter 8.

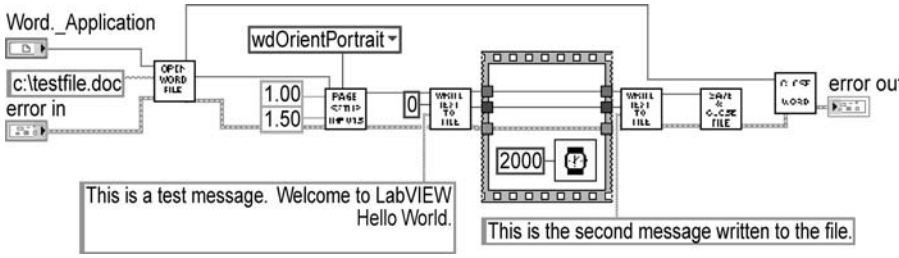

FIGURE 5.25

5.9 INSTRUMENT I/O ASSISTANT

The instrument IO Assistant is a tool that can be used to set up communications with an instrument or device. This tool is available in the Instrument IO palette. When the tool is selected it places an Express VI on the code diagram. The configuration window will come up automatically (depending on your LabVIEW settings) . Here you can select what connected device you want to communicate with and configure any needed settings.

Now that you have configured your instrument you can add steps to the code. If you click on add step you will have the choice of Query and Parse, Write, and Read and Parse. In this example we are communicating with a switch. The code will have the choice of opening or closing the switch and which channel to operate on. For this example, Write has been selected. When you select Write, you can configure several settings. Here two variables are added, a string to input open or close and a number for channel. This input is shown in Figure 5.26.

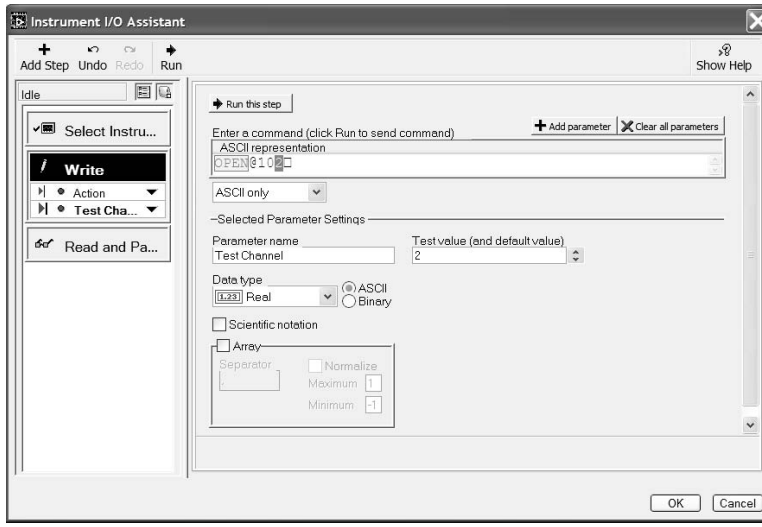

FIGURE 5.26

You can continue to add read and write actions until you have completed the needed actions. You can run the code directly from this interface or you can close the window, which will update the express VI on the code diagram. You can connect the inputs and outputs as needed and run the code from here.

5.10 IVI DRIVERS

IVI drivers were developed to allow hardware-independent test programs. In 1997, a number of manufacturing companies approached National Instruments to develop generic drivers that would be interchangeable. The IVI Foundation was a direct result of this effort. The organization, made up of representatives from National Instruments and a number of the instrument manufacturing companies including Hewlett Packard, Tektronix, Rohde & Schwarz, and Anritsu, has developed a set of standards and requirements for "generic" drivers. The IVI Foundation is an evolving group that is open to end users and interested parties. Anyone who is interested in joining can find more information on the IVI Foundation Web site (www.ivifoundation.org).

The goal of the IVI Foundation was to build upon the standards set by the VXI Plug&Play Systems Alliance. The VXI Plug&Play standards promote multi-vendor interoperability by standardizing the instrument commands for all VXI instruments. IVI instruments go one step further by trying to standardize an instrument type regardless of format. A power supply would have the same API regardless of the standard (GPIB, Serial, VXI, other) or the manufacturer.

IVI drivers are not language specific. By using DLLs to convert the commands from a uniform API to the required instrument code, there is a wide range of programming languages that can be used. LabVIEW and LabWindows/CVI are both capable of using IVI drivers; however, the DLLs can be written using only LabWindows. Due to the use of DLLs, IVI drivers are not platform independent. If you do not want to write your own drivers, or are not using LabWindows/CVI, a library of IVI drivers is available from National Instruments.

5.10.1 Classes of IVI Drivers

The initial rollout of the IVI standards encompassed five classes of IVI drivers. The current IVI standard includes eight classes of IVI drivers. The eight classes are the DC Power Supply, Oscilloscope, Digital Multi-Meter (DMM), Arbitrary Waveform/Function Generator, Power Meter, RF Signal Generator, Spectrum Analyzer and Switch. New classes may be defined as the technology advances.

Let's look at the DMM class as an example. The IVI driver for the DMM class (IviDmm) is designed to operate a typical DMM, as well as support advanced functions found in the more complex instruments. The IVI class description divides the DMM into two sections: fundamental capabilities and extensions. The fundamental capabilities cover functions like reading a measurement or setting a range. An extended capability would be like setting auto-range, making multiple measurements, or other advanced features not available on all DMMs. For the DMM, there are fourteen groups defined (13 extension groups). Groups refer to the defined classification of commands. Examples of extension groups are IviDmmMultiPoint

and IviDmmDeviceInfo. The IviDmmMultiPoint extension group supports the base DMM functions and also the ability to accept multiple triggers and acquire multiple samples per trigger. The IviDmmDeviceInfo extension group supports the base DMM functions and also the ability to return additional information concerning the instrument's state, such as accuracy and aperture time. Documentation on all the IVI classes and their groups is available on the IVI Foundation website.

5.10.2 Interchangeability

This section will discuss how IVI drivers allow for instrument interchangeability. One problem that has been seen in production testing for a long time is the lack of instrument interchangeability. This problem can arise for a number of reasons. An instrument that needs to be taken out for calibration or maintenance is one example. Other possible scenarios are when an instrument needs to be replaced and is no longer available; if the test system developer wants to use an instrument from another manufacturer; if the test software is going to be used by a group in another area with their own set of instruments. These issues are problems because the test software would have to be altered to replace an instrument with one from another manufacturer, or a newer model with new functions and commands. These problems force test system developers to stay with the same system instead of improving or cost reducing. The ability to change instruments would allow greater flexibility.

The first benefit of IVI drivers is the ability to interchange instruments. A power supply from a different manufacturer can replace the existing power supply without changing the test software. This will allow the development of a generic test station; users would be able to change instruments based on availability and cost.

5.10.3 Simulation

This section will discuss how an IVI driver can be used in simulation mode to allow debugging and input checking without the instrument being connected to the computer. When a programmer is developing software, the ability to incrementally debug the code is a technique that helps reduce development time. This would be an implementation of the spiral software development model. There is a full discussion of software development models (spiral and waterfall) in Chapter 4. By using IVI drivers in simulation mode, the test code can be debugged without the instrument being connected to the computer. The driver will return an instrument handle to allow a program using VISA to run without the instrument physically present. The user can also use the driver in simulation mode to choose the measurement that will be returned to the test program. This will allow the designer to test the program's response to common and unusual measurements returned by the instrument. The measurement returned can be set to random number generation within a range.

When using instrument-specific drivers, another feature is realized. The developer can perform range and status checking while developing the software. The driver will verify that the inputs sent to the instrument are within the specifications of the instrument. These are options that can be turned on or off. Turning on the range-checking feature helps the developer debug the test software. Turning off

range-checking allows for faster execution time when the program is run in the final environment.

5.10.4 State Management

An IVI driver can speed up application execution when state caching is used. One problem encountered when programming a test application, particularly when utilizing state machine architecture, is the lack of knowledge of the instrument's current state. The user will not know what state the instrument is in at a given time, requiring the programmer to set all necessary configurations, even if the instrument is already configured properly. This can add substantial time to a test application.

The solution is to use state caching. This can be performed when using LabVIEW or LabWindows/CVI. When using state caching, the last setting for each function on an instrument is stored. When the driver goes to change the setting of a function, the driver checks to see what the last known state of that function was. If the setting is the same, the driver will not execute the command. The driver also tracks changes in settings when different screens are displayed.

5.10.5 IVI Driver Installation

When the IVI driver CD is inserted into the drive, the IVI Driver Library Installation interface starts. In the interface you have the options of viewing the release notes, installing the IVI driver library, installing instrument drivers, and browsing the CD. To install the IVI software you will need to click the IVI Driver Library Installation selection. This will begin the standard installation interface. After making the typical selections, a selection screen will appear. The installer will prompt you to select the instrument drivers to install. This is the initial place to obtain and install the IVI instrument drivers. There are a number of items on this installer screen. On the left of the screen is a selection for the IVI class. On the right side is a listbox containing the specific drivers. In order to install the drivers you need to use in your development, you must first select the desired IVI class. This will list the available IVI drivers in the specific driver input. In the specific driver input is the list of available drivers with a checkbox selection on the left of the individual drivers. To select the needed driver, you need to select the appropriate checkbox.

In addition to the IVI class input and the specific drivers input, there are three additional options on the IVI driver installation screen. There is a button to select all instrument drivers, a button to deselect all instrument drivers, and a control to replace the existing drivers. This control can be set to either replace the instrument drivers currently installed with the IVI drivers, or to leave the existing instrument drivers. This is an important selection if you have made modifications to the current standard drivers; it will prevent the IVI installation from overwriting your changes.

The IVI installation will set up three categories of software. The installation categories are instrument drivers, utilities, and driver software. The instrument driver installation includes the IVI class drivers, the IVI class simulation drivers, and the IVI-specific drivers. The utility installation includes NI Spy, the Virtual Bench software, and the Measurement and Automation Explorer. The driver software

includes the IVI engine, NI VISA, NI DAQ, and the CVI run-time engine. When the installation is complete, the computer will need to be restarted.

5.10.6 IVI Configuration

The first step, after installing the IVI software, is to run the IVI Configuration Utility. The IVI Configuration Utility can be started by double-clicking the Measurement & Automation Explorer (MAX) icon on the Windows desktop, or by selecting the utility from the National Instruments folder in the Programs folder in the Start menu. The IVI settings are available under the IVI Drivers folder on the left window. Figure 5.27 shows the IVI configuration parameters in a MAX window.

There are three categories of IVI configuration items in the IVI folder. The main sections are Logical Names, Driver Sessions and Advanced. The logical name is what is used by LabVIEW to select the appropriate IVI driver similar to calling COM 1 to connect to the first serial port on a computer. To add a new logical name you can simply right click on the logical names folder and select add. Here you can set the logical name and what driver session it is linked to.

The Driver Sessions folder contains the loaded IVI drivers. If you have the equipment connected, and the software was installed properly the name should show up in the list. Don't worry if it is not there; you can download the drivers for the instrument you are looking for from NI's website. When you install the IVI drivers the instrument name will show up here.

When you click on the IVI driver name a configuration window will show up in the right MAX window. There are five tabs to configure: General, Hardware,

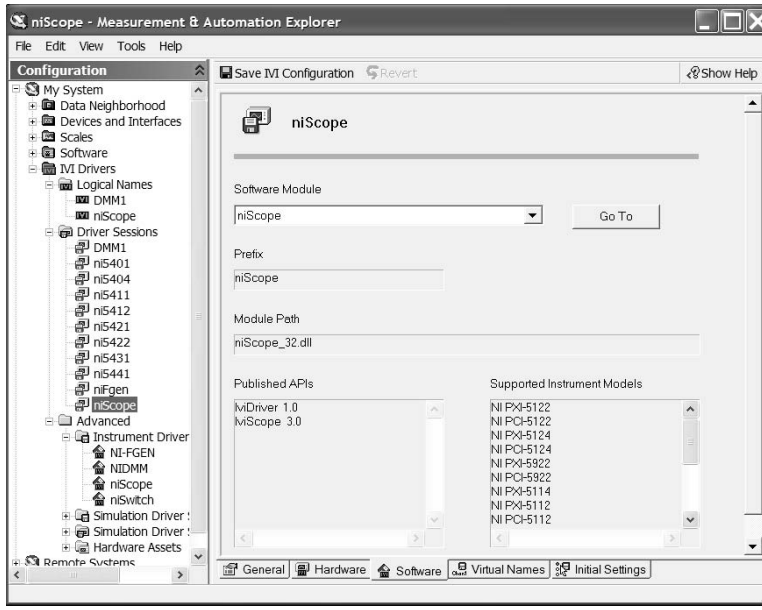

FIGURE 5.27

Software, Virtual Names and Initial Settings. Here you have the ability to select whether to simulate with a specific driver, choose the software module to use (shows what DLL contains the source code and the supported instrument models), virtual names for hardware options such as CHANNEL1, hardware links to actual installed equipment if available and initial instrument settings.

The Advanced folder gives you access to the driver software modules and hardware. In the Hardware Assets sub-menu you can define equipment in your test system and link to the resource descriptor. Here you will have a name that can be used in your code that is not tied to a specific instrument. You can define a DMM that is connected to a GPIB address. If you replace the DMM with one from another manufacturer, you would not have to change your code. You would still reference the DMM name. Be aware that the names used in the IVI configurations are case sensitive.

5.10.7 How to Use IVI Drivers

IVI class drivers are used in the same manner as standard instrument drivers. The IVI class drivers can be found in the Instrument I/O subpalette of the Functions palette. Each type of IVI class driver has its own subpalette. Each subpalette contains an Initialize and Close VI. There are also groups of VIs to perform instrument configuration, instrument functions, and utility functions that are necessary for the specific class driver. The developer can use these class drivers like typical VISA drivers. The programmer would put an Initialize VI on the diagram first. The main input to the Initialize VI is the logical name. The logical name is what tells the LabVIEW program what instrument and drivers to reference. As you will recall, in the setup of the IVI configuration items, the logical name references a particular virtual instrument. These logical names can be altered as needed using the Configuration utility. It is recommended that you set the name initially after installation and do not change it often. Applications that have been developed use this name, and may not work once the logical name has been altered. The virtual instrument refers to a specific driver in the Instrument Drivers folder, and a device. The specific driver then specifies the DLL containing the code module used to communicate with the device. The VIs associated with the instrument driver are placed in the Instrument Drivers palette during the installation.

The Initialize VI also has inputs to do an ID query and reset the instrument. The outputs of the VI are the Instrument Handle and the Error Out. The Instrument Handle can be passed throughout the VI and subVIs, just like a standard VISA instrument handle. Once the instrument is initialized, the functions required to perform the necessary programming task can be accomplished in two ways: the user can utilize the function VIs from the class driver subpalettes or make use of the LabVIEW Property node. When doing IVI driver configurations, the LabVIEW Property node is used in the same manner as ActiveX controls. As with all applications using communications, the final step is calling the Close IVI Class Driver VI. The following diagram shows an IVI example written with standard VIs and with the Property node. The VIs perform exactly the same function. Figure 5.28 illustrates the IVI example with and without the Property node.

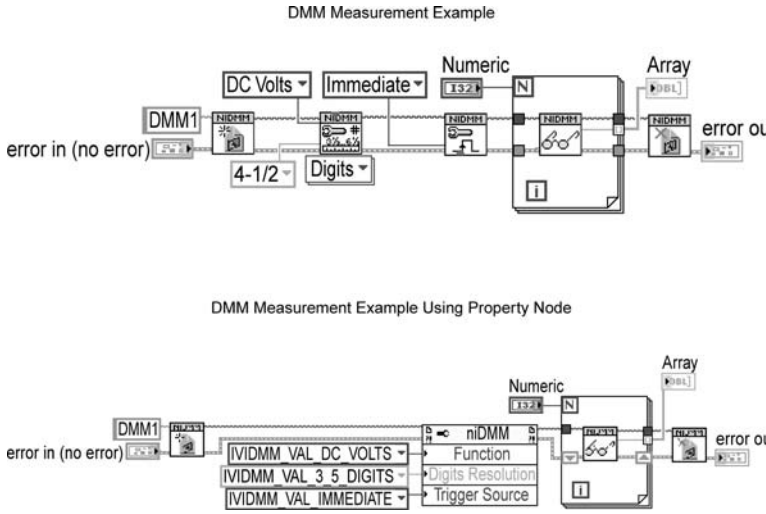

FIGURE 5.28

5.10.8 Soft Panels

The Soft Panels (IVI Virtual Bench) VIs were designed to simulate the front panel of an instrument. The main use for the soft panels is manual instrument control. The user can use the Front Panel VI to manually control the instrument. The similarity of the soft panel VI to the actual instrument interface allows the user to be familiar with some of the function immediately. The key is when the specific type of instrument changes. If a Hewlett Packard oscilloscope is replaced with a Tektronix oscilloscope, the user should still be able to control the instrument with no noticeable change. As the interface is the same, there are no new knobs or menus to learn. The IVI configuration files do all the work. An example of the niScope soft panel is shown in Figure 5.29.

5.10.9 IVI Driver Example

The information above can become confusing. Every name seems to include either virtual, instrument, or driver. In addition, each type of file references one or more of the other IVI file types. In order to alleviate some of the confusion, an example will be provided to help clarify things. It should be noted that there are a few examples that come with the IVI library. The examples are contained in the following path: labview\examples\instr\iviClass.llb.

For this example we will be simulating an oscilloscope using the IVI drivers. The first step is to create a logical name. In the IVI Configuration we need to go to the Logical Name folder and select Create New. You can enter a name and description. For this example we will call the instrument "niScope." You will be able to select a driver to associate with the name. We will use an existing driver. The selections available in the pull down menu depend on the drivers you have loaded. The driver we selected was the standard niScope driver.

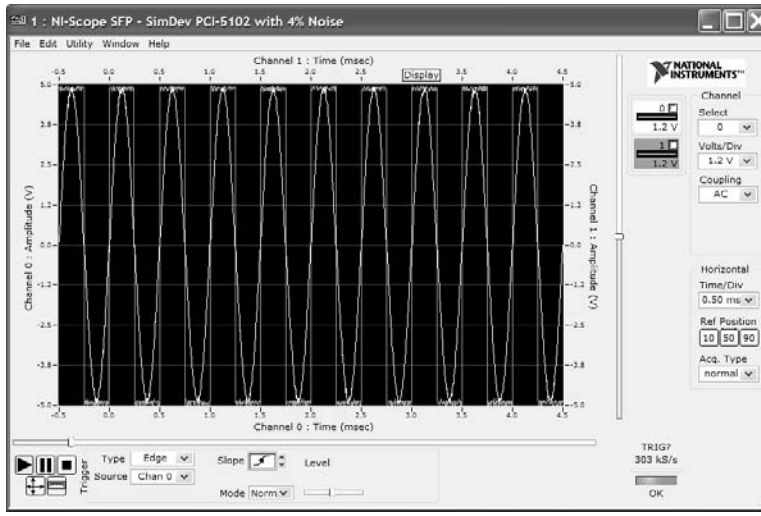

FIGURE 5.29

If you go to the Logical Name folder, you will see the summary page. Here you can modify the driver associated with the name. This is where you would make the modification if an instrument were exchanged for another in your equipment rack. If the program were written properly, with only class drivers being used in the code, no modifications would be necessary to the application. If we were going to use a real oscilloscope, the device could be changed here.

In the Driver Sessions folder you can see the driver properties. The General tab allows the programmer to select options such as Range Checking and Query Instrument Status. In the software tab, the programmer has the choice of using the class driver or the specific driver for the output data simulation. We will be using the class driver for this example. The programmer also has the opportunity either to modify or change the simulation virtual instrument and its settings. If you click on the initial settings tab you can configure the starting state of the instrument. You can go through the list of properties and change default values to match your instrument defaults. We will leave the current settings. The Virtual Names tab allows the programmer to associate a virtual channel name with the specific channel string.

Now that the IVI configurations are set up, the application can be written. As has been mentioned before, only IVI class drivers should be used in the application to reduce the amount of modifications if a new instrument is used. This example is based on the niScope example that comes with the IVI library (Acq Wfm Edge Triggered). The first coding step is to put the Initialize VI for the IVI oscilloscope class on the code diagram (niScope Initialize.vi). For the logical name input, a string constant or control with the text "niScope" should be wired to the first terminal. An alternate method is to create a resource name control Here you can select the appropriate name from a pull down list. The inputs for ID query and reset device are both defaulted as "true." And, as always, the Error In cluster should be wired to the final terminal of the input connector.

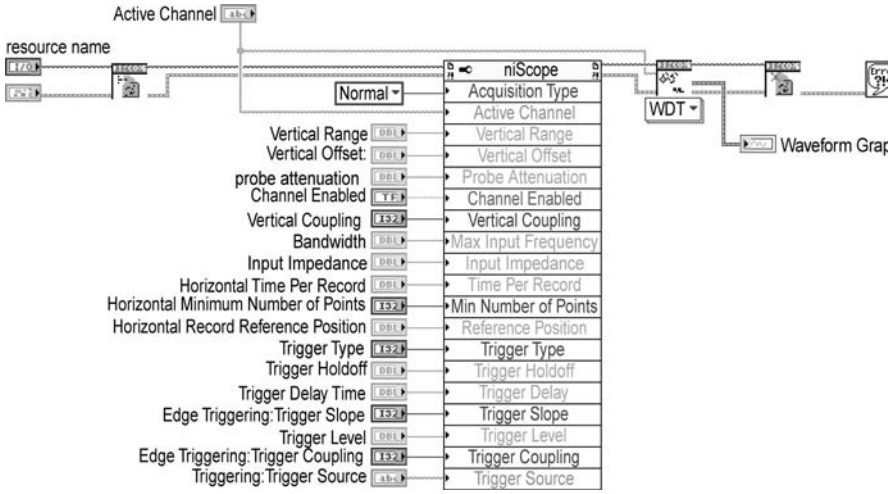

FIGURE 5.30

Once the instrument is initialized, the inputs for the vertical and horizontal parameters need to be configured. In addition to the scope parameters, the triggering inputs need to be set. For this example, we will be using the Property node to configure the necessary parameters instead of the class drivers. The first code diagram is shown in the following figure. The first trial of this VI used one Property node to set all of the vertical, horizontal, and triggering parameters. Selecting the item from either the ActiveX subpalette or the Application Control subpalette of the Function palette created the Property node. By using the positioning tool, you are able to increase the number of inputs by pulling the bottom corner down. The same task could also be accomplished by right-clicking on an input and selecting Add Element. A control was created for each input that was necessary with the appropriate default values being set. Figure 5.30 displays the Scope Example Code Diagram.

After setting the chart and triggering parameters, set the class VI to determine the actual number of data points to acquire. The output of this VI is then wired to the NIScope Read WDT VI. The data from the Read Waveform VI is then bundled together and wired to the waveform graph on the front panel. Finally, the Instrument Close VI is added to the code diagram.

Now it is time to debug our driver. The best method for debugging an application like this is to use the NI Spy utility to monitor the instrument communications. The NI Spy was discussed earlier in this chapter. There are a couple of IVI-specific items that need to be mentioned. When you go to the Spy menu of the NI Spy utility you will notice the installed IVI drivers available in the monitor list. For this example we will want to turn off the NI-VISA and the NI-488.2 monitoring options. They are being turned off to aid in interchangeability checking. If those items are turned off, any items with conflicts will be listed in blue. This will aid in spotting conflicts without having to go through all of the items captured. Once Capture has been turned on, we are ready to test our application.

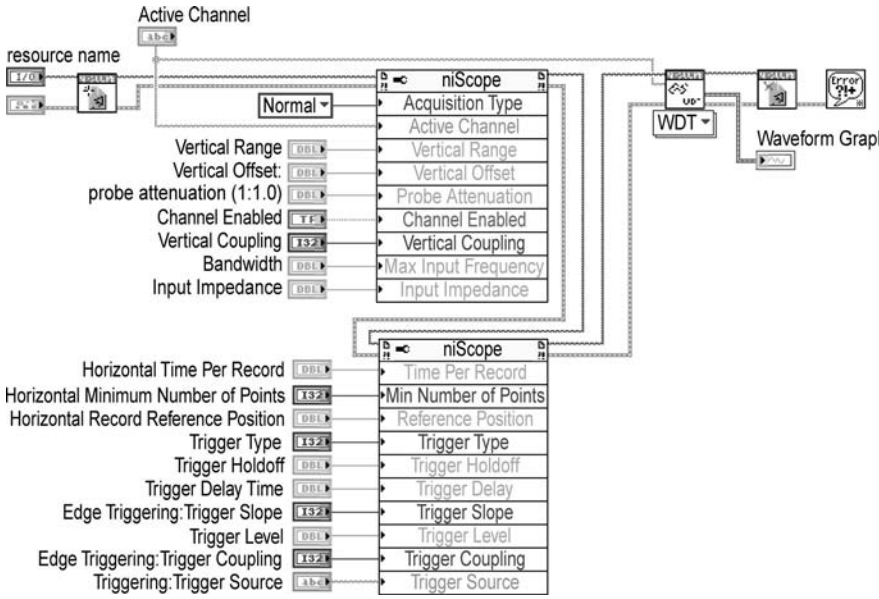

FIGURE 5.31

When we press the Start button, the program starts executing. As you should now be aware, there is an error in the application. A message box comes up stating that an error occurred at the tenth argument of the Property node. The listed possible reasons are that a Null is required for the channel name when setting an attribute that is not channel-based. The message box also lists the bad attribute. Clicking Continue will close the message box and complete the execution. As the horizontal parameters and the triggering parameters are not channel-based, they cannot be on the same property node as the vertical parameters. Figure 5.31 shows the modified code diagram that corrects this programming error.

Before we move on with the testing, let's take a look at the API captures from the NI Spy utility. The NI Spy display is shown in Figure 5.32. The tenth entry in

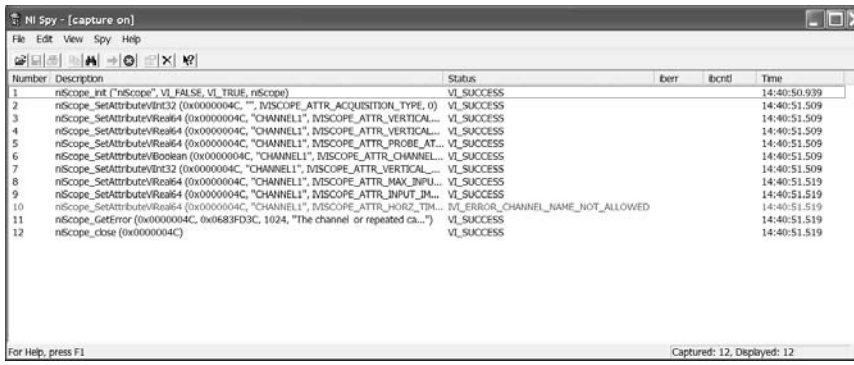

FIGURE 5.32

the list is the attempt to write the horizontal time per record parameter to the scope. Because an error occurred in this step, the line is in a red font. Double-clicking on the line opens up the Properties box. In the Properties box you can see what the inputs and outputs of the communications were. If you click on the Output tab, you will see the instrument returned the following error statement: (IVI_ERROR_CHANNEL_NAME_NOT_ALLOWED). This is followed by a description in the text box below the error statement. Now that we have modified our VI, we can attempt to run the application again. This time the application runs without error.

BIBLIOGRAPHY

LabVIEW Graphical Programming — Practical Applications in Instrumentation and Control. Gary W. Johnson, McGraw-Hill, New York, 1997.

G Programming Reference, National Instruments, Austin, TX, 1999.

Data Acquisition Basics Manual, National Instruments, Austin, TX, 1999.

Code Interface Reference Manual, National Instruments, Austin, TX, 1999.

IVI-1: Charter Document, IVI Foundation, San Diego, CA, November 1999.

IVI-4.2: IviDmm Class Specification, IVI Foundation, San Diego, CA, January 2004.

Developing COM/ActiveX Components with Visual Basic 6, Dan Appleman, SAMS, 1998.

Using TCP/IP. John Ray, QUE Corporation, Indianapolis, 1999.

Application Note No. AN006, Developing a LabVIEW Instrument Driver, National Instruments, Austin, TX.

Application Note No. AN087, Writing Win32 Dynamic Link Libraries (DLLs) and Calling Them from LabVIEW, National Instruments, Austin, TX.

Application Note No. AN111, LabVIEW Instrument Driver Standards, National Instruments, Austin, TX.

Application Note No. AN120, Using IVI Drivers to Simulate Your Instrumentation Hardware in LabVIEW and LabWindows/CVI, National Instruments, Austin, TX.

Application Note No. AN121, Using IVI Drivers to Build Hardware-Independent Test Systems with LabVIEW and LabWindows/CVI, National Instruments, Austin, TX.

Application Note No. AN122, Improving Test Performance through Instrument Driver State Management, National Instruments, Austin, TX.

6 Exception Handling

Code is often written without considering the potential that an error might occur. When events occur that an application is not expecting, problems arise. Then, during the debugging phase, an attempt is made to go back to the code and implement some error traps and correction. However, this is usually not sufficient. Exception handling must be taken into account during the early stages of application development. The implementation of an error handler leads to more robust code.

This chapter discusses errors and the topic of exception handling in LabVIEW. First, exception handling will be defined along with its role in applications. This explanation will also clarify the importance of exception handling. Next, the different types of errors that can occur will be discussed. This will be followed by a description of the available LabVIEW tools for exception handling, as well as some of the debugging tools. Finally, several different ways to deal with errors in applications will be demonstrated.

6.1 EXCEPTION HANDLING DEFINED

Exceptions are unintended or undesired events that occur during program execution. An exception can be any event that normally should not take place. This does not mean that the occurrence of the exception is unexpected, but simply should not happen under normal circumstances. An error results when something you did not want to happen, does. Therefore, it makes sense to make alternate paths of execution when exceptions take place. When exceptions or errors occur, they must be dealt with in an appropriate manner.

Suppose that you have written a program in which you divide two variables, Integer x by Integer y. The resulting quotient is then used for some other purpose. On some occasion, y may be set to zero. Some programs do not trap errors such as dividing by zero and allow the CPU to throw an exception. When the CPU throws an exception for a program, the program will be terminated by the operating system — in most cases this is an undesirable result. In LabVIEW, the result of this division is undefined. LabVIEW returns the result Inf, or infinity, on a floating point division by zero. If you were to use the integer quotient and remainder, a division by zero results in a quotient of zero. In both cases, the application does not throw an exception and close. However, this is an example of an unexpected and unintended outcome. Infinity can be converted successfully into a word integer in LabVIEW. If the value is converted for other uses, several other errors can result. This is an example of a simple error that has to be managed using exception handling.

Exception handling is needed to manage the problems or errors that occur. It is a mechanism that allows a program to detect and possibly recover from errors during execution. Exception handling leads to more robust code by planning ahead for potential problems. Depending on the purpose of an application, the ability of an application to respond to unexpected events can be critical. Typical programs that are used by one person, at their desk may not need a lot in terms of robust performance. An application that is automating an assembly line that is producing hundreds of thousands of production units for revenue would benefit significantly from robust and stable code. The implementation of an error handler increases the reliability of the code. It is difficult to prepare for all the possible errors that might occur, but preparing for the most probable errors can be done without much effort.

You can write your code to try to catch as many errors as possible, but that requires more code to implement. After a certain point you will have more code involved in catching errors than you do for performing the task that you originally set out to do. The exception handling code itself may sometimes contain errors. You also create a problem of what to do when the error is caught.

Error detection and error correction are two different activities, but are both part of exception handling. Error detection consists of writing code for the purpose of finding errors. Error correction is the process of managing and dealing with the occurrence of specific errors. First you have to catch the error when it occurs; then you have to determine what action to take.

Performing error detection is useful for debugging code during the testing or integration phase. Placing error checks in the code will help find where the faults lie during the testing phase. The same detection mechanisms can play a dual role. The detection mechanism can transfer control to the error handler once the handler has been developed. This will be beneficial if you are using an iterative development model, where specific features can be added in each cycle.

Exception handling is performed a little differently in each programming language. Java uses classes of exceptions for which the handler code can be written. For example, an exception is represented by an instance of the class "Throwable" or one of its subclasses. This object is used to carry information from the point at which an exception occurs to the handler that catches it. Programmers can also define their own exception classes for their applications.

C++ uses defined keywords for exception handling: Try, Catch, and Throw. The Try and Catch keywords identify blocks of code. Try statements force the application to remember their current location in the call stack and perform a test to detect an error. When an exception occurs, execution will branch directly to the catch block. After the catch block has executed, the call stack will be "rolled back" to the point where the program entered the Try block.

LabVIEW provides some tools for error detection. But just like other programming languages, implementation of exception handling code is left to the programmer. The following sections will guide you in creating error handling code for your application. Chapter 10 covers topics relating to Object-Oriented Programming, including definitions for object, class, and subclass, but exception handling in Java and C++ are beyond the scope of this book.

6.2 TYPES OF ERRORS

Errors that occur in LabVIEW programs can be categorized into either I/O-related or logic-related. I/O errors are those that result when a program is trying to perform operations with external instruments, files, or other applications. A logical error is the result of a bug in the code of the program. The previous example of dividing an integer value by zero is a logical error. These types of errors can be very tricky to find and correct. Both I/O- and logic-related errors are discussed in the following sections.

6.2.1 I/O Errors

Input/Output encompasses a wide range of activities and VIs within LabVIEW. Whether you are using communication VIs (TCP, UDP, .NET, USB, Bluetooth, etc.), data acquisition, instrument I/O, or file I/O, there is a probability that you will encounter related errors at some point.

I/O errors can be the consequence of several things. The first circumstance that can cause this type of error is improper initialization or configuration of a device or communication channel. For example, when performing serial communication, the baud rate must match between the external device and the controller. If this initialization is performed incorrectly an error will result. For some devices a command must be sent to put them into remote mode, which will allow communication with the controller. When reading or writing to a file, the file must be opened first. Similarly, when writing to a database, a connection has to be established before records can be inserted. Initialization can also include putting an instrument or device into a known state. Sometimes this can be done by simply sending a reset command, after which the device will enter a default state.

A second cause of I/O errors is simply sending the wrong commands or data to the instrument or application. When invalid data is sent, a write error will result. Some devices simply ignore the data whereas others return an acknowledgment. This can play a role in what type of correction and handling you perform. When data is being communicated to an external device, you have to ensure both the correct data and the correct format are being sent. You must adjust the information you are sending to suit what the device is expecting to receive. Typographical errors can also be classified in this section.

Another I/O-related error takes place when there is a problem with the instrument or application being used. When dealing with applications or files, this can occur for several different reasons. The file may not be in the specified path or directory. Alternatively, you may not have the needed file permissions to read or write to the file. Instrument I/O errors of this nature usually occur if the instrument is not powered-on or not functioning properly. A similar problem happens when the instrument locks up or freezes. Power cycling may return it to a known state and make it operational again. These types of errors can also be a result of incorrectly configuring the external device. Instruments can return unusual results when they are not configured appropriately.

Missing hardware or software options can be a source of I/O errors. You may also need to check if you have the correct interface drivers installed. Interface incompatibility and component incompatibility should be investigated.

The last and most common issue is a network or communication bus is interrupted. Internet Protocol (IP) based communication will periodically find a message has not gone through. IP itself does not guarantee message delivery. IP also does not guarantee delivery in order of transmission — its possible for IP packets to arrive in an order other than what they were transmitted in. Applications using any Ethernet communications needs to have the ability to sort packets when they arrive out of order, or simply do not arrive.

6.2.2 Logical Errors

Logical errors happen when there are faults in the code itself. The code diagram in Figure 6.1 illustrates an innocent mistake that can occur. In the While loop, the programmer intends the loop to stop executing when the temperature reaches 75.0 degrees or higher. However, the loop, as it stands currently, will stop when the temperature is lower than 75.0. This is an example of an easy mistake that can cause an error in applications. These types of problems can be difficult to find and are also very time consuming. Debugging tools are invaluable when looking for the source of faults.

Errors can sometimes occur when the inputs specified by the user are not validated. If the user does not provide reasonable inputs expected by the program, an error can occur. The application must validate the data to ensure it is within the acceptable range. For example, the user may have to specify which unit number, between one and ten, to perform a sequence of tests on. The program has to verify that only the acceptable range is entered before beginning execution. Unit zero may not exist for test purposes, therefore the code must check for the appropriate inputs. Be aware of numeric precision errors and conversion errors that can also be difficult to track down.

LabVIEW allows the programmer to set acceptable ranges for Numeric, Boolean, and List & Ring controls. This can be done by popping up on the control and selecting Data Range from the menu. The programmer also has the option of coercing the input value so that it is within the valid range on front panel controls only. The two options when setting a data range are to ignore or to coerce the value when a range is specified. This option is available in the drop-down box. The coercion option

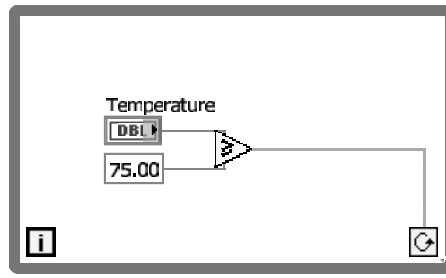

FIGURE 6.1

reduces the need to write code for performing the same task, but is a practice we generally do not recommend. If a value is entered out of range from the user interface it is preferable to notify the user the value is not valid and attempt to get the user to correct the input.

6.3 BUILT-IN ERROR HANDLING

LabVIEW notifies the user of some run-time errors for instrument and file I/O operations through dialog boxes. LabVIEW does not deal with the errors and, in general, leaves exception handling to the programmer. However, LabVIEW does provide some tools to aid the programmer in exception handling. The first tool that will be discussed is the error cluster. The error cluster is used in transporting information from the detection mechanism to the handler. After the error cluster, a brief description of VISA error handling will be presented. Next, the error-handling VIs will be considered. There are three error-handling VIs in particular: the Simple Error Handler VI, the General Error Handler VI, and the Find First Error VI. Section 6.4 will then discuss the implementation of exception handling code.

6.3.1 Error Cluster

The error cluster is a detection mechanism provided for programmers. The cluster consists of a status, code and source. Each of these provides information about the occurrence of an error. The status is a Boolean that returns "true" if an error condition is present. The code is a signed 32-bit signed integer that distinguishes the error. The source is simply a string that gives information on where the error originated. The error cluster as a whole provides basic details about the error that can be used for exception handling purposes. In LabVIEW 7 and 8, the coerce function of a control does not work when the VI is used as a subVI.

Figure 6.2 shows the Error In and Error Out clusters as they appear on the front panel. The Error In and Error Out clusters can be accessed through the Array & Cluster subpalette in the Controls palette. The error clusters are based on National Instruments' concept of error I/O. VIs that utilize this concept have both an Error In control and Error Out indicators, which are usually located on the bottom of the front panel. The cluster information is passed successively through VIs in an application, consistent with data flow programming.

FIGURE 6.2

The error clusters can serve a dual purpose in your application. By using error I/O, the order of execution of VIs can be forced. This eliminates the need for sequence structures to control the order of execution. Simply pass error clusters through VIs for detection and order.

When the cluster is passed in to a VI, the VI should check if an error condition is present. By default, a new, blank VI doesn't have an error cluster input or code to check the status. If there is no existing error, execution will continue. The cluster picks up information on whether an error has occurred during the VI's execution and passes this information to the next VI, which performs the same check. In the simplest case, when an error does occur in any VI, the VIs that follow and use the cluster should not execute. When the program completes, the error is displayed on the front panel.

The error I/O concept and the error clusters are easy to use and incorporate in applications. Many of the LabVIEW VIs that are available in the Functions palette are based on this concept: the Data Communications palette and contained subpallet, most of the Instrument I/O VIs (VISA, GPIB, GPIB 488.2), and some Data Acquisition and File I/O VIs use error I/O. The variable introduced in LabVIEW 8 also uses the error cluster. By using these VIs and wiring in the error clusters, much of the error detection work is already done for the programmer. These built-in VIs provide the detection needed in the lower-level operations. When wiring these VIs on the code diagram, you will notice that the Error In terminal is on the lower left side of the VI, whereas the Error Out terminal is on the lower right side. This is a convention followed by all VIs developed by National Instruments, and is also recommended when creating drivers.

Figure 6.3 is an example of how the error clusters can be used. The VI uses GPIB Write and GPIB Read from the Instrument I/O palette. It is a simple instrument driver that can be used to write data to and read data from an instrument. To perform error detection, the programmer only has to use the Error In and Error Out clusters and wire them accordingly in the code diagram. The error detection work is left to the Instrument I/O VIs. When this driver is needed as part of a larger application, the error I/O concept is used. Figure 6.4 uses two drivers with the Error In and Error Out wired. The second VI in the diagram will not execute if an error occurs during the execution of the first VI. Execution order is forced, causing the second driver to wait for the error cluster data from the first one. This approach can be applied successfully to larger applications.

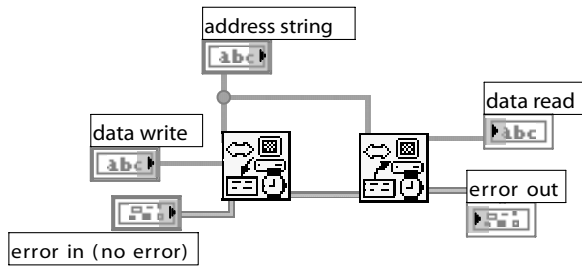

FIGURE 6.3

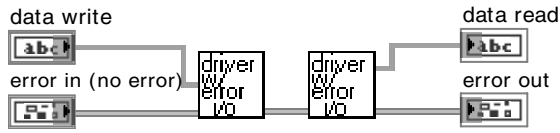

FIGURE 6.4

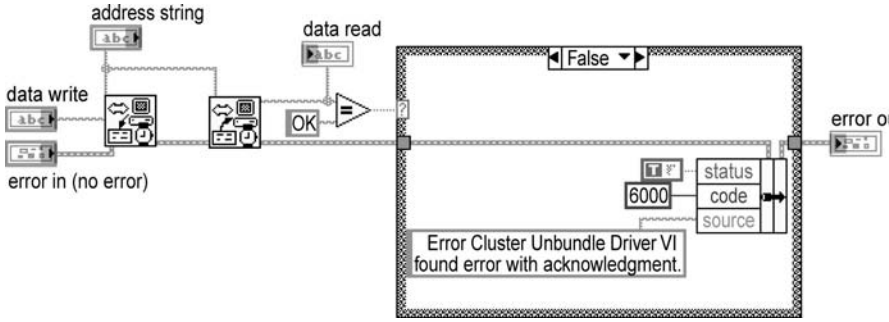

FIGURE 6.5

The error clusters can also be used to perform error checks other than those done by the available LabVIEW VIs. Suppose you are communicating to a device or application that returns acknowledgments when sending commands and data. An "OK" value is returned when the data is accepted and valid, and an "NOK" is returned if the data is invalid or the command is unknown. The LabVIEW VIs do not perform any check on instrument- or application-specific acknowledgments, only on general communication errors. Returning to the VI in the previous example, we can implement our own error check. Figure 6.5 shows how this is done.

The Bundle by Name was used from the Cluster palette to accomplish this. If the acknowledgment returned does not match "OK," then the error cluster information is altered. The Boolean is made true, the code assigned is 6000, and the source description is also wired in. LabVIEW reserves error codes 5000 to 9999 for user defined errors. If the acknowledgment returned matches the expected value, we wire the error cluster through the "true" case directly to Error Out without any alterations. The error detection for the correct acknowledgment will now be performed every time this driver is called.

Figure 6.6, Extra Source Info.vi, shows an example of how to get more information out of the error cluster for debugging and error-handling purposes. This VI adds extra information to the source string of the error cluster. First, the error cluster is unbundled using Unbundle by Name. The extra pieces of information that will be added include the time the error was generated and the call chain. Call Chain, available on the Application Control palette, returns the VI's call chain all the way to the top level in string format. The call chain information is useful for user-defined errors to indicate where the error was generated. These two pieces of data will then be bundled together with the original source information generated by the error cluster. You can put any other type of information you would like returned with the

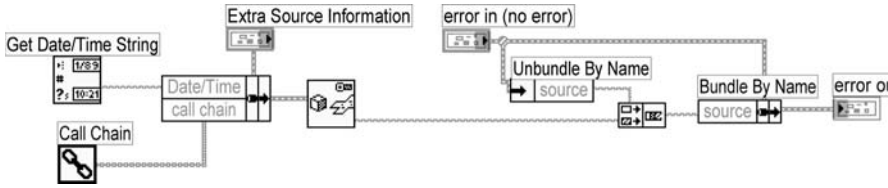

FIGURE 6.6

error cluster in a similar manner. It can be used to give the programmer more facts on the error that may be helpful for debugging. The errors can then be logged in a text file or database for reference. Error logging is demonstrated in Section 6.4.6 through a basic example.

6.3.2 Error Codes

A list of possible LabVIEW-generated errors is accessible through the Online Reference in the Help menu. The errors are listed by the error code ranges and the types of possible errors. Error codes can be either positive or negative values, depending on the type of error that is generated. When a zero error code is returned, it indicates that no error has occurred. Warnings are indicated with a code that is nonzero, whereas the status returned is "false." Table 6.1 contains a list of the error code ranges.

A handy tool for looking up error codes is also available through the Help menu in LabVIEW Version 5.0 and later. When Explain Error is selected, a new window appears with the error cluster on the left side and a text box on the right side. The error code can be input either in hexadecimal or decimal format. An explanation of the error will be provided for the error code in the text box. This tool provides a quick way to get additional information on an error for debugging purposes.

6.3.3 VISA Error Handling

VISA is a standard for developing instrument drivers and is not LabVIEW-specific. It is an Application Programming Interface (API) that is used to communicate with different types of instruments. VISA translates calls to the lower-level drivers, allowing you to program nonsimilar interfaces with one API. See Chapter 5 on instrument drivers for more information on VISA.

VISA is based on the error I/O concept, thus VISA VIs have both Error In and an Error Out clusters. When an error occurs, the VIs will not execute. There is a set of VISA-specific error codes that can be found in LabVIEW Help. The VISA Status Description VI can be used in error-handling situations. This VI is available in the VISA subpalette of the Instrument I/O palette.

When you are using instrument drivers that utilize VISA, there are some additional errors you may encounter. The first may be the result of VISA not being correctly installed on your computer. If you choose the typical install, NI-VISA is selected for installation by default. If you have performed the custom install, you must make sure the selection has been checked. You will not be able to use any

TABLE 6.1
Error Codes

Error Type	Code Range
Networking	–2147467263 through –1967390460
Instrument driver	1074003967 through –1074003950
VISA	–1073807360 to –1073741825
Report generation	–4105 to 41000
Formula parsing	–23096 to –23081
Mathematics	–23096 to –23000
Signal processing	–20999 and –20337 to –20301; –20115 to –20001
Point by point	–20207 to –20201
Regular expression	–4644 to –4600
Waveform	1820; –1811 to –1800
Apple Event	–1719 to –1700
Instrument driver	–1300 to –1210; 1073479937 to 107347994
Timed loop	–823 to –800
Windows registry access	–620 to –600
Signal processing, GPIB, instrument driver, formula parsing, VISA	0
GPIB	1 to 20; 30 to 32; 40–41
General	1 to 52; 67 to 91; 97 to 100; 116-118; 1000 to 1045; 1051 to 1086; 1088 to 1157; 1174 to 1188; 1190 to 1194; 1196 to 1198; 1307 to 1320; 1362
Networking	53 to 66; 108 to 121;1087; 1191
Serial	61 to 65
Windows connectivity	92 to 96; 1172; 1173; 1189; 1195; 1199; 14050 to 14053
Instrument driver	102, 103
MATLAB and Xmath	1046 to 1050; 1053
Run-time menu	1158 to 1169
Waveform	1800 to 1809
SMTP	16211 to 16554
Signal processing	20001 to 20353

VISA VIs unless your system has this option installed. Another error can be related to the lower-level serial, GPIB, or VXI drivers that VISA calls to perform the instrument communication. For example, if you have a GPIB card installed on your computer for controlling instruments, make sure the software for the card has also been installed correctly to allow the use of VISA VIs. You can use NI-Spy to monitor calls to the installed National Instrument drivers on your system. NI-Spy is briefly explained in Section 5.5.

When using VISA in your application, remember to close all VISA sessions or references you may have opened during I/O operations. Leaving open sessions can degrade the performance of your system. You can use Open VISA Session Monitor.vi to find out the sessions that you have open, and to close the ones that are not being used. This VI is available in the following directory: \LabVIEW\Vi.lib\Utility\visa.llb. This VI can be helpful while you are debugging an application.

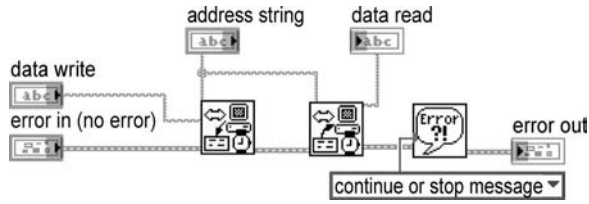

FIGURE 6.7

6.3.4 Simple Error Handler

The Simple Error Handler can be found in the Time & Dialog palette in the Functions menu. This VI is used for error reporting. It is used with LabVIEW VIs that utilize error I/O and the error cluster. The purpose of the Simple Error Handler is to notify the operator that an error has occurred, but it can be customized for added functionality. It takes the error cluster as input and determines if an error was generated. If an error has been generated, the VI displays a dialog box with the error code, a brief description of the error, and the location of the error. The Simple Error Handler utilizes a look-up table to display the description of the error based on the error code.

As mentioned, one of the uses of the Simple Error Handler is for error notification purposes. The programmer can select the type of dialog box to display by wiring the corresponding integer or enumerated constant. A value of 1 displays the dialog box with only the OK button for acknowledgment. A value of 2 displays a button dialog box with Continue and Stop buttons. This allows the operator to stop execution of the program. A value of 0 gives no notification to the operator, even when an error has been generated. This might be used when exception handling is to be performed elsewhere by using the error?, code out, or source out, outputs from the Simple Error Handler.

You must keep in mind that this VI will halt execution until the operator responds to the dialog box. If your intention is to start the program and walk away, the program will not continue if an error is generated. Dialog boxes should be used only when the program is being monitored. Consider using e-mail for notification using the SMTP package. Chapter 8 also shows you how to incorporate the e-mail feature using .NET, or there are SNMP utilities as part of the Data Communications pallete.

Figure 6.7 shows how the Simple Error Handler can be used. This is the same VI shown in Figure 6.3. Notice that the Simple Error Handler has been merely added as the last VI in the flow. The value of 2, which corresponds to the two-button dialog box (Continue and Stop), is being passed to the VI. If an error is detected in either GPIB Read or GPIB Write, the dialog box will appear displaying the error code, description, and the source of the error.

6.3.5 General Error Handler

The General Error Handler essentially performs the same task as the Simple Error Handler. The Simple Error Handler offers fewer choices when used in an application. The Simple Error Handler is a wrapper for the General Error Handler. The General

Error Handler can be used in the same situations as the Simple Error Handler, but as the General Error Handler has a few more options, it can be used for other purposes where more control is desired.

The General Error Handler allows the addition of programmer-defined error codes and corresponding error descriptions. When these arrays are passed in, they are added to the look-up table used for displaying error codes and descriptions. When an error occurs, the possible LabVIEW-defined errors are searched first, followed by the programmer-defined errors. The dialog box will then show the error code description and specify where it occurred.

The General Error Handler also offers limited exception handling options. The programmer can set the error status or cancel an error using this VI. An error can be canceled by specifying the error code, source, and the exception action. Set the exception action to Cancel Error on Match. The look-up tables are searched when an error occurs. When a match is found, the error status is set to "false." In addition, the source descriptor is also cleared and the error code is set to zero in the output cluster. Similarly, when the status of the Error In is "false," it can be set to "true" by passing the exception action for the error code and source.

6.3.6 Find First Error

The Find First Error VI is a part of the Error Utility Package located in \LabVIEW\vi.lib\Utility directory as part of the error.llb package. The purpose of this VI is to create an Error Out cluster. It takes the following inputs: Error Code Array, Multiline Error Source, and Error In Cluster. When the Error In status is "false" or is not wired in, the VI tests to see if the elements of the error code array are nonzero. The VI bundles the first nonzero element, the source, and a status value of "true" to create the Error Out cluster for passing back out. As the source is a multiline string, the index from the array of error codes is used to pick the appropriate error source for bundling. If an Error In cluster is passed in, then a check is first performed on the cluster's status. When the status is "true," the Error In cluster will be passed back out and the array check will not be performed.

Find First Error is practical for use with LabVIEW VIs that do not utilize error I/O but pass only the error code value out. Some VIs that output only the error code are the original serial I/O VIs and some Analysis VIs. The Find First Error VI can be used to convert the error code from these VIs to a cluster. The error cluster can then be used in conjunction with other VIs that utilize error I/O.

Figure 6.8 is an example of how the Find First Error can be used. Both the Bytes at Serial Port.vi and the Serial Port Read.vi pass an error code out. An array is built with the two error codes that are passed out. A multiline string for the source is also created in the example. The source will give information on the origin of the error. The Find First Error.vi assembles the error cluster and passes it to Error Out. If an error has occurred, the first error that occurred will be sent to the Error Out cluster. If no error was generated, the Error Out cluster will contain a "false" status Boolean, no error code, and an empty source string. The error cluster can then be passed to the General Error Handler or the Simple Error Handler to display a dialog box if needed.

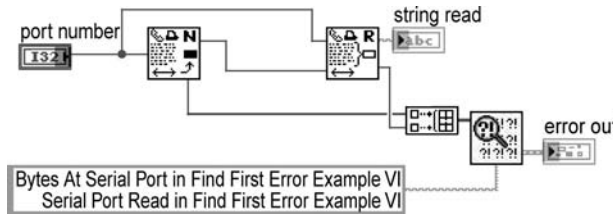

FIGURE 6.8

6.3.7 Clear Error

This VI is located in the dialog palette and as the name implies, it clears the error cluster of all values. Essentially, it takes an error cluster in and outputs a new error cluster with no information. This VI is useful when an exception has been successfully handled.

If the application is expected to log errors seen or if the error cluster is being used to carry information in the error string, then this VI should not be used — it will clear the error string and code.

6.4 PERFORMING EXCEPTION HANDLING

Exception handling encompasses both the detection of errors and the treatment of the errors once they have been found. The previous sections presented several types of errors that can occur, as well as the built-in LabVIEW functions that are available for exception handling. This section will illustrate different approaches that are effective for managing errors. The effectiveness of an error handler can be improved by building it into your application during the early stages of development. It will support the readability and maintainability of your code, as well as code reuse. When error handling is not considered while you are architecting the application, the handling code will consist of patches for each exception.

You may have some questions about the implementation of exception handling code in order to make the handler both efficient and effective. When should error detection, reporting, and handling be performed? What should the application do when an exception is detected? Where and how should it be implemented? The following subsections will address the where, how, and what on exception handling approaches for your application.

6.4.1 When?

The question of when to implement is a little bit trickier and depends on the specific situation or application being developed. This may vary depending on the objective of the application, the amount of time available, the programmers' intent, and several other factors. Some areas of an application that need handling may be easier to identify than others. You may be able to identify areas where errors cannot be tolerated, or where errors are prone to occur, through past experience. These are definite targets for error detection, reporting, and handling.

To answer this question as completely as possible, you must also look at specific instances in an application to determine what alternative scenarios are foreseeable as well as their possible consequences. To illustrate this point, consider an example in which you must open and read or write to a file using an I/O operation. To answer if exception handling code is needed, and maybe even what is needed, think about the following scenarios and consequences. What will need to happen if the file that is being written to cannot be opened? What happens if the read or write operation fails because the drive is full? What happens if the file cannot be closed? Answering these questions will help put the need for handling into perspective for the application. It will also help you look at the application and determine where the exception handling activities are needed by asking similar questions. Error handling will definitely need to be implemented if the file I/O operation is crucial to the application, and if other parts of the program are dependent on this activity's being successful.

6.4.2 Exception-Handling at Main Level

To answer the "where" question, exception handling should be managed at the Main Level or Test Executive level. The Main Level controls and dictates program flow. By performing exception handling at the Main Level, the program execution and control can be maintained by the Top Level. This is important because the exception handler code may alter the normal flow of the program if an error is detected. You may want the code to perform several different actions when an error has occurred. When exception handling is performed at lower levels, program control must also be passed to the lower levels. This is a good reason why the implementation of an exception handler should be considered when architecting the application. Application structure and processes for application development are discussed in Chapter 4. Reading Chapter 4 will help you get a better perspective on how to approach the development of an application and other topics that must be considered before you begin.

Performing exception handling at the Main Level also eliminates the need for duplicating code in several subVIs. This permits the error handler code to be located in one place. The separation of error handler code from the rest of the code reduces confusion and increases readability and maintainability. Logical flow of the program will be lost in the clutter when error handling is performed with the rest of the code. This is explained further in Section 6.4.5 on exception handling with state machines.

The suggested style is similar to other programming languages where Error Information is sent to a separate piece of code for handling purposes. As mentioned earlier, both Java and C++ have a separate section that performs the error handling after the evaluation of an error is completed. There is no such mechanism inherent in LabVIEW, but this approach resembles it.

6.4.3 Programmer-Defined Errors

Defining errors was briefly discussed in Section 6.3.1 along with the error cluster. The ability to define errors is significant because LabVIEW leaves application-specific error handling to the programmer. As mentioned earlier, error codes 5000-

9999 are dedicated for use by the programmer. The programmer must perform error checking in circumstances where faults cannot be tolerated, as was shown in Figure 6.5. An error code must then be assigned to the error check as well as a source string to indicate the origination.

When implementing a programmer-defined error in a subVI or driver, you must make sure that an error was not passed in. Simply unbundle the error cluster and check the value of the status Boolean. If an error was passed in, but you fail to check the status, you may overwrite the error cluster with the new Error Information that you implemented. This will make it nearly impossible to find the root of the problem during the debugging phase. You must also make use of shift registers when using error clusters within loop structures to pass data from one iteration to the next. If shift registers are not used, error data will be lost on each iteration.

Records must be kept of the error codes that have been assigned by the user. A look-up table can be created that contains all of the error codes and sources assigned. This can then be used with the General Error Handler or with other exception handling procedures. It may be a good practice to maintain a database or spreadsheet of user-defined error codes. A database facilitates the management as the number of codes grows in size.

When you are assigning error codes, you can group similar errors into specified ranges. This is helpful when deciding the course of action when errors occur. For instance, you can set aside error codes 6000-6999 for incorrect acknowledgments from instrument I/O operations. When an error in this range occurs, you can identify it and decide how to deal with it easily. LabVIEW-generated errors are grouped in a similar manner to facilitate their identification and management.

User-defined warnings can also be assigned codes to indicate that an undesired event has occurred. You can use these to signal that the data taken may not be entirely valid due to the occurrence of some event during application execution. The user can investigate the source of the warning further to determine the validity of the data. Multiple errors can be reported and handled by unbundling the error cluster and appending the new information.

6.4.4 Managing Errors

Once you have a list of the errors that you want to deal with that can be detected, you have to decide what to do with them if they occur. When an error occurs it should be passed to the exception handling code. The exception handling code can deal with the errors in different ways. Expanding on the idea of grouping similar errors, the code can check to see what range the error has fallen in to determine the course of action. Figure 6.9, Error Range Example.vi, is an example of grouping ranges of error codes for handling purposes. When a set of exceptions is considered to be logically related, it is often best to organize them into a family of exceptions.

The easiest way to deal with an error is to simply display a dialog box to notify the user that an error has occurred. This dialog box can be as simple as the one displayed by the General Error Handler. You can create your own VI to display a dialog box to include more information, including what the user can do to trouble-shoot the error. This usually results in halting execution of the program.

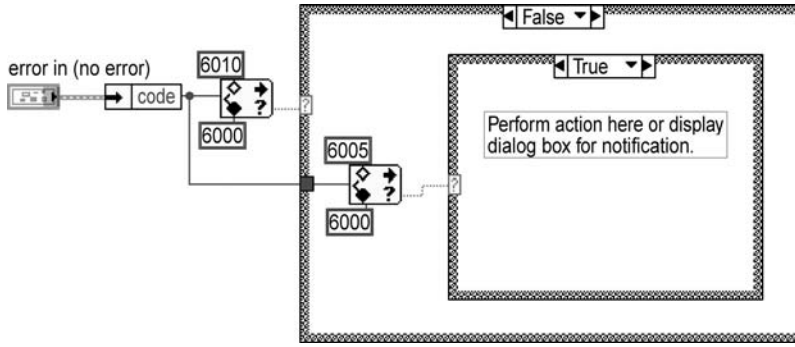

FIGURE 6.9

You can get more involved by attempting to correct or recover from an error in the exception handling code. In this case, the more general range checking technique will not suffice because the exact error code will be used to determine how to correct it. It also requires detailed knowledge of the error and exactly how it can be corrected. Suppose, for example, that you get a specific error telling you that the device under test did not respond to the commands sent to it. You also know that this happens when the device is not powered-on or has not been initialized properly. You can then attempt to correct this error by power cycling the device and initializing it. Then you can retry the communications and continue with the program if successful.

Figure 6.10 illustrates a technique for dealing with specific error codes as an alternative to the general range-checking method. This method needed to be used in LabVIEW 4.1 or older because the default state was not defined in these versions. If the error code did not exist in the array of error codes, the search 1-D array function would return -1. There is no '-1' case in the case statement and you would have had a problem pre LabVIEW 5. Current versions of LabVIEW have a default case permitting you to wire the code directly to the selector terminal of the case structure. For case statements using integers, such as this error array, set the default case to 0. This case will then execute for error codes for which no case has been defined.

The method displayed is similar to a look-up table described earlier. An array that contains all of the error codes is used with the Search 1D Array VI. The error

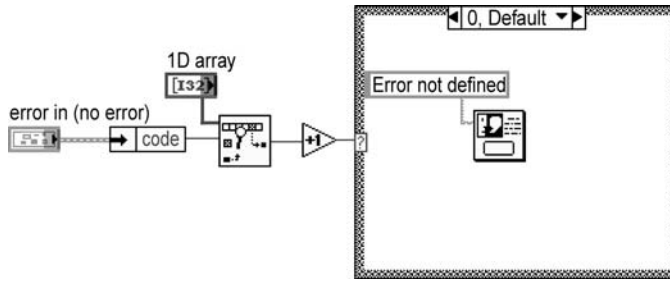

FIGURE 6.10

code is passed to it and the index of the error code is searched for. The index drives the case statement, which takes the right course of action for the error code. If there is no match for the error code, the Search 1D Array returns a value of –1. By adding 1 to the result, Case 0 is selected from the structure. This case will serve as the default case when no match is found. In the example shown, a dialog box is displayed indicating that the error code was not defined.

Another alternative is the use of strings to drive case structures. You can implement the previous example by unbundling the cluster to retrieve the source information. This string can then be used to determine the course of action by wiring it to the case selector terminal.

6.4.5 State Machine Exception Handling

The use of a state machine offers several advantages for exception handling code. One advantage is that the exception handling code can be located in one place. This is done through the use of an Error state. The Error state is responsible for all exception handling in the application. This eliminates the need for exception handling code in several places. Maintaining the code becomes easier when the code resides in one location. Using a state machine also facilitates exception handling management at the Main or Test Executive Level. The Error state is part of the Main Level, so control is maintained at the upper level.

Another advantage is that duplication of error handling code is reduced when the code is placed in one location. Similar errors may be generated in different parts of your code. If you do not perform exception handling in one place, you may have to write code in several places for the same type of error.

Conditional execution of code can be implemented without creating a complex error handler through the use of a state machine. Exception handling code determines program execution based on the severity of the error that was generated. You may want your code to skip execution of the parts of the code that are affected by the error, and continue execution of the rest of the program. For example, suppose you have a series of ten different tests you want to perform on a device under analysis. If an error occurs in Test 1 and that same error will affect Tests 5, 6, and 7, you may still want to execute Tests 2, 3, and 4. In this case, using a queued state machine will simplify the procedure for performing this task. The Error state can parse out the states that correspond to Tests 5, 6, and 7 from the list of states to execute. In cases where the error can be corrected, the program needs to remember where execution was halted so it can return to the same location and continue. The use of state machines facilitates implementation of this feature into exception handling code. Proper logic for diagnosing state information must be kept to make this possible. In addition, proper logging and saving routines should be incorporated to ensure that data is not lost.

The conditional execution can also be applied to tests that fail. You can design the application to execute a test depending on the outcome of another test. If Test 1 fails, you may want to skip Tests 2 and 3 but continue with the remaining tests. Again, you can parse the tests that should not be executed. Chapter 3 discusses the various state machines in depth. The example in Section 6.4.9 will help demonstrate the implementation of exception handling in a state machine context.

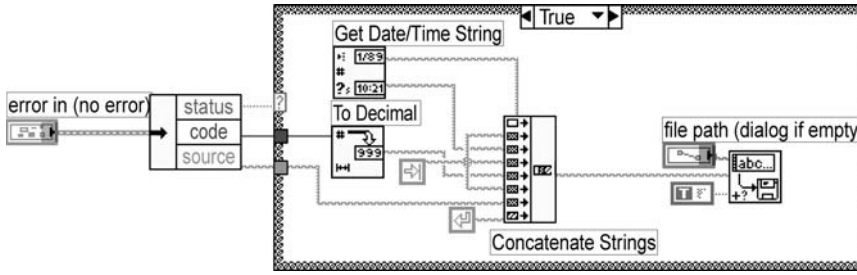

FIGURE 6.11

6.4.6 Logging Errors

Error logging is useful for keeping records of faults that have occurred during the execution of a program. The error log should report the code, the origin, a brief description, and when the error occurred. Upon the occurrence of an error, the log file is opened, written to, and closed. If further exception handling code exists, the error can be dealt with in the appropriate manner.

Error logging is beneficial in cases where the exception handling code has already been implemented and when there is no exception handler in the application. When exception handling has been implemented, error logging gives the programmer insight into the types of errors that are being generated and whether the code is handling them properly. The log can be used as a feedback mechanism to determine areas of the exception handling code that are unsatisfactory. These areas can then be enhanced to build a more robust application.

In instances when the exception handling code has not yet been developed, the error log can be used in a similar manner. The log can serve as a basis for developing the error handling code. The errors that occur more frequently can be addressed first. This method attempts to strike a balance in the amount of effort spent in developing an exception handler. The concept here is to gain the maximum benefit by attacking the most common errors.

Figure 6.11 is an example of a VI that logs errors. First, the status in the error cluster is checked to determine whether an error has occurred. If an error has been generated, the date, time, error code, and source are written out to a file that serves as the error log. The Write Characters to File VI is used to perform the logging. This VI can be used in multiple places where logging is desired, or in a central location along with other exception handling code. As the error information has been converted into a tab-delimited set of strings, it can be imported into Excel for use as a small database.

6.4.7 External Error Handler

An exception handler that is external to the application can be written to manage the errors that are generated during program execution. The application must then make a call to the external error handler. This can be beneficial when using the NI Test Executive. The error handler VI will be loaded when it is referenced in the

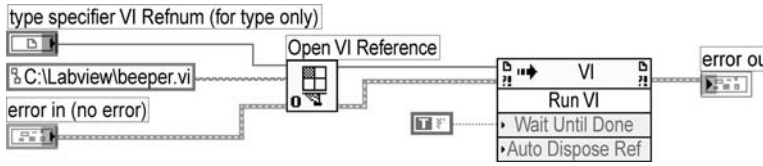

FIGURE 6.12

application. The error handler VI can be written to perform all the relevant tasks, similar to carrying out exception handling within an application.

If the error handler is written to accommodate general exceptions, it can be called in as many applications as needed. Figure 6.12, Load External Handler.vi, shows how a VI can be loaded and run from an application. First, a reference to the VI must be opened using Open VI Reference. This VI can be accessed through the Application Control palette. You must specify the path or directory in which the error handler resides. Set the VI Server Class to "Virtual Instrument" by popping up on the VI Refnum. The Invoke node is used to run the external VI. The Invoke node is also available in the Application Control palette. When the VI reference is passed to the Invoke node, the VI Server Class will automatically change to Virtual Instrument. Then, by popping up on "Methods," you can select the Run VI method from the menu. Data can be passed to the error handler VI using the Invoke node and selecting the Set Control Value method. The functions available on the Application Control palette are described in Chapter 2.

EXAMPLE:

An example of how an external exception handler is implemented is shown in Figure 6.13. This code diagram demonstrates the steps involved in using an external handler: opening a VI reference, passing the input values, running the external VI, and closing the reference. Opening a VI reference and running an external VI has already been described. In this example, the error cluster is passed to the external exception handler which determines the course of action.

First, a VI reference is opened to External Handler.vi as shown in the VI path. Then, the error cluster information is passed to External Handler.vi using the Set Control Value method on the Invoke Node. This method requires the programmer to specify the Control Name, the Type Descriptor, and the Flattened Data. The error cluster is passed to this method by flattening it using Flatten to String from the Data Manipulation subpalette in the Advanced palette. The flattened data string and the type descriptor are then wired directly from Flatten to String to the Set Control Value method. The Control Name is a string that must match identically the control name on the front panel of the VI to which the data is being passed. The name specified on the code diagram is Error In (No Error), as it appears on the front panel of the External Handler.vi. The VI is run using the Run VI method, and, finally the reference is closed.

Figure 6.14 illustrates the code diagram of External Handler.vi. This VI is similar to an exception handler shown previously. It takes the error cluster information and

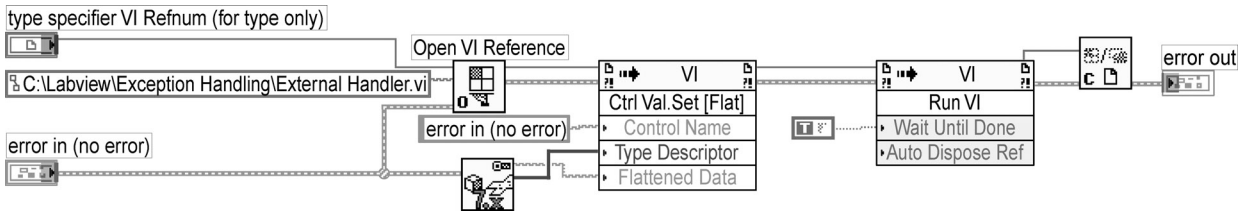

FIGURE 6.13

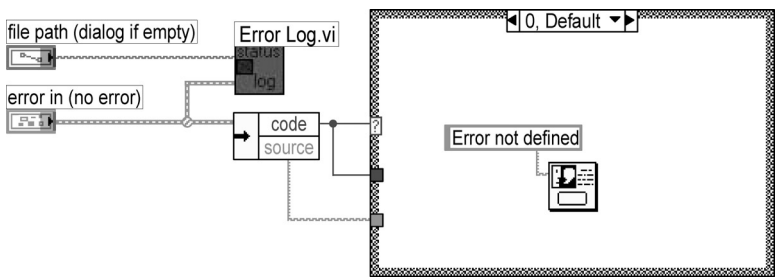

FIGURE 6.14

decides the course of action based on the error code. The Error Information is logged using Error Log.vi, and the case structure is driven by the error code. Case 0 is used as the default for error codes the handler is not prepared for.

In this example, the error cluster data was passed to the external VI. Similarly, data can be retrieved from the controls or indicators from the VI if it is desired. The Get All Control Values method can be used to perform this action. This method will retrieve all control or all indicator values from the external VI. The data is returned in an array of clusters, one element for each front panel control or indicator. The cluster contains the name of the control or indicator, the type descriptor, and the flattened data, similar to the way the values were passed to the External Handler VI in the example.

6.4.8 Proper Exit Procedure

In situations where fatal or unrecoverable errors occur, the best course of action may be to terminate execution of the program. This is also true when it is not reasonable to continue execution of the program when specific errors are generated. However, abnormal termination of the program can cause problems. When you do decide that the program should stop due to an error, you must also ensure that the program exits in a suitable manner.

All instrument I/O handles, files, and communication channels must be closed before the application terminates. Performing this task before exiting the program minimizes related problems. Consider, for example, a file that is left open when a program terminates. This may cause problems when other users or applications are attempting to write to the file because write privileges will be denied.

Upon the occurrence of an error, control is passed to the error handler. Therefore, it is the responsibility of the error handler to guarantee that all handles, files, and communication channels are closed if the error cannot be recovered from. The easiest way to implement this is to have the error handler first identify the error. If the error that was generated requires termination of the program, code within the handler can perform this task. Figure 6.15, Close Handles.vi, is an example of a VI that is used solely to close open communication channels. A VISA session, file refnum, TCP connection ID, and an Automation Refnum are passed to this VI, which then proceeds to close the references.

A program should be written to have only one exit point, where all necessary tasks are executed. The best way to implement this is to utilize a state machine. By using a state machine, only one exit point is needed and will serve as the Close state. Correspondingly, there is only one place where all exception handling is

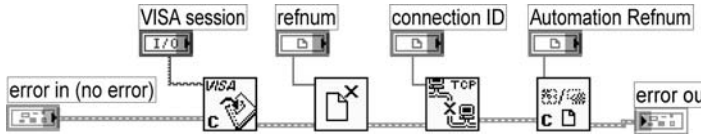

FIGURE 6.15

performed: the Error state. When an error is identified as fatal, the Error state will force the state machine to the Close state. The Close state will be responsible for terminating the program in the appropriate manner. All handles, files, and communication channels will be closed in this state. As only one Close state is needed, it will also be the last state executed during normal execution of the program when no error exists. This style makes the code easier to read and maintain.

6.4.9 Exception Handling Example

Several methods of performing exception handling were provided in this section. A closing example that utilizes some of the topics that were discussed is presented in Figure 6.16. The example utilizes the state machine structure with an Error state for error handling.

The purpose of Next State.vi is simply to determine which state will be executed next. The Next State VI is also responsible for checking if an error has occurred after the completion of each state. When an error has occurred, the next state that will be executed is the Error state. The Error state first logs the error using the Error Log VI. The error code is checked to determine if it falls in a certain range that corresponds to instrument driver errors. If the error code is within that range, it is considered as unrecoverable or fatal in this example. When a fatal error is detected, the Close state is wired out to the Next State VI to execute the proper exit procedure.

If the error code does not fall in the range specified, the code is again compared to an array of user-defined error codes. This drives the case structure, which will take the action that is appropriate depending on the error that was generated. When no match results from this comparison, Case 0 is executed as illustrated in Figure 6.17.

When a match results for Case 1, the Remove States VI will remove the states that cannot be executed due to the error that was generated. Then, the program will continue with the states that can be executed according to the elements in the states array. This is shown in Figure 6.18.

Figure 6.19 shows the Close state of the state machine. This state is executed during normal termination of the program, and also when a determination is made that a fatal error has occurred. As shown in Figure 6.16, the Error state will force the Close state to execute when an unrecoverable error has been found. The only task of the Close Handles VI is to close any references and communication channels that have been opened. This will minimize problems when the application is run again.

This example demonstrates the ideas presented in this section. First, exception handling was performed at the Main Level so that program control did not have to be passed to lower levels. Second, the error handler code was separated from the rest of the code to increase readability. Not only does this reduce confusion, it also reduces the need for duplicating code in several places. Next, the use of a state machine allowed the placement of exception handling code in one location to increase maintainability and conditional parsing of tests. Error logging was performed to keep a record of exceptions that occurred. Finally, a proper exit procedure for the application was implemented. Following good practices in the creation of an exception handler will lead to sound and reliable code.

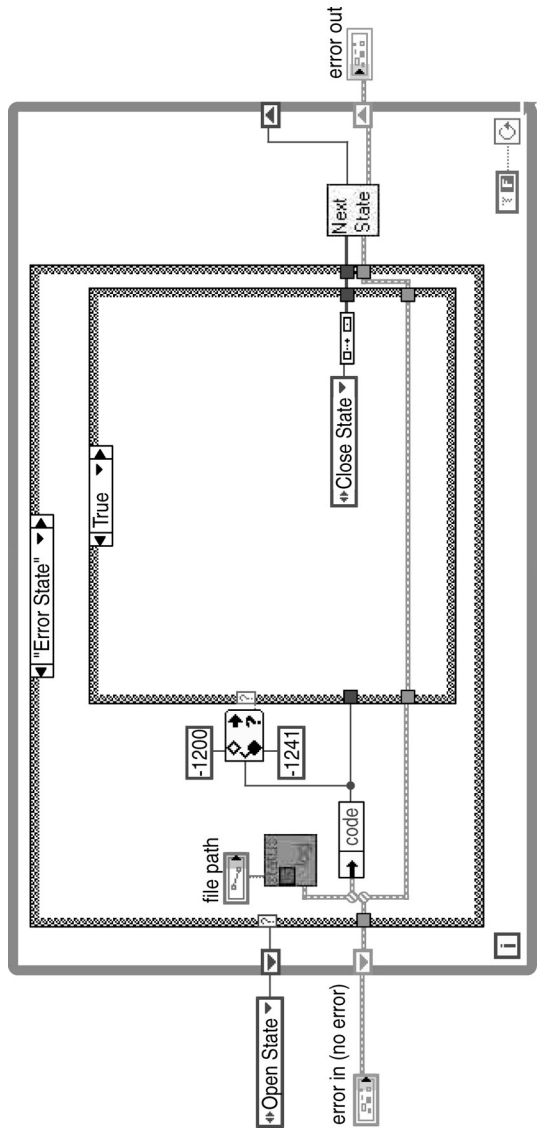
error out
Next State
Close State
True
"Error State"
-1200
-1241
code
file path
Open State
error in (no error)

FIGURE 6.16

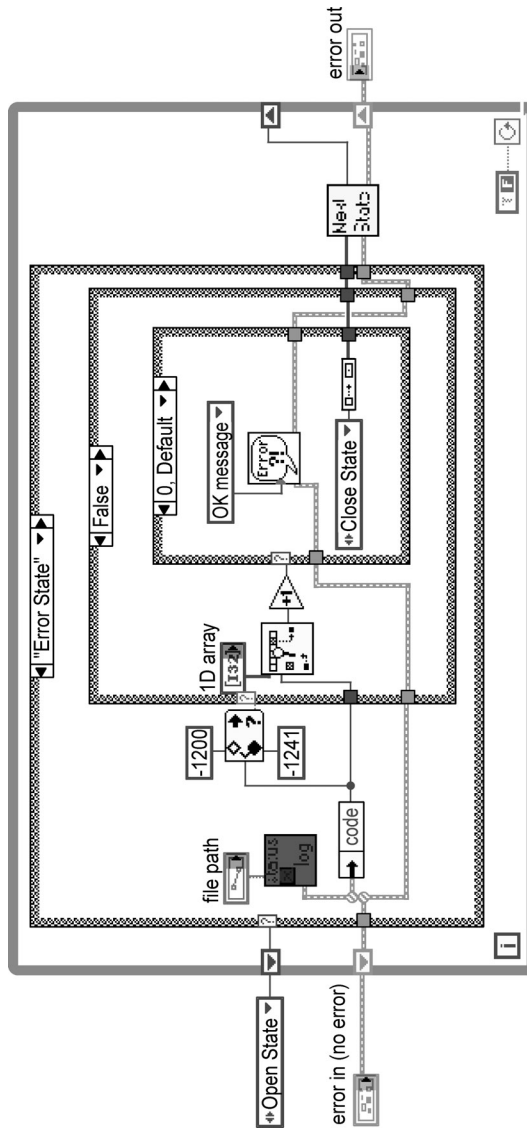
error out
Next State
"Error State"
False
0, Default
OK message
Close State
1D array
-1200
-1241
file path
code
Open State
error in (no error)

FIGURE 6.17

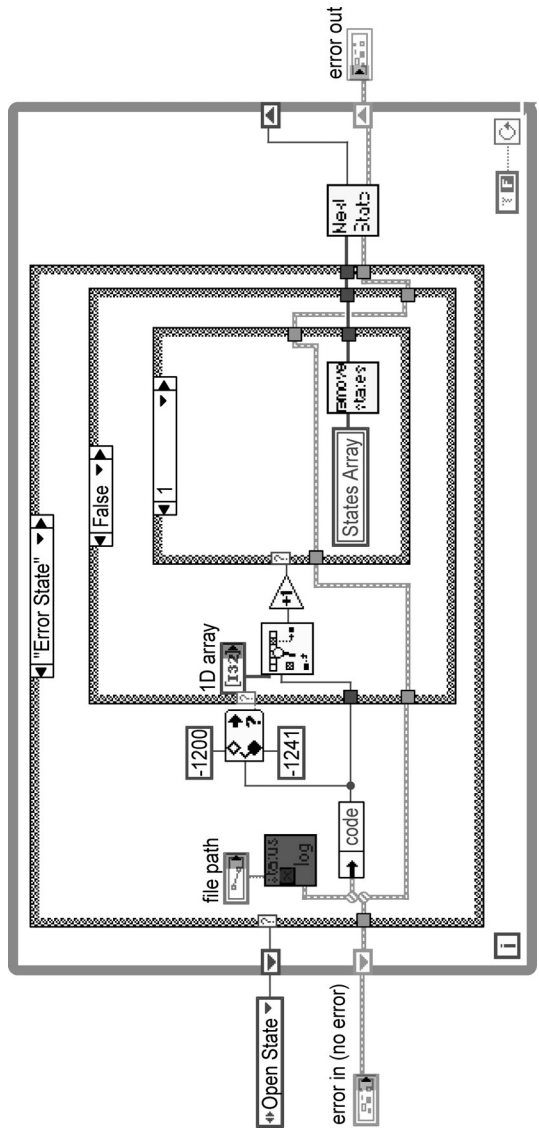
error out
Open State
error in (no error)
"Error State"
False
1
1D array
-1200
-1241
file path
code
log
States Array
Next State

FIGURE 6.18

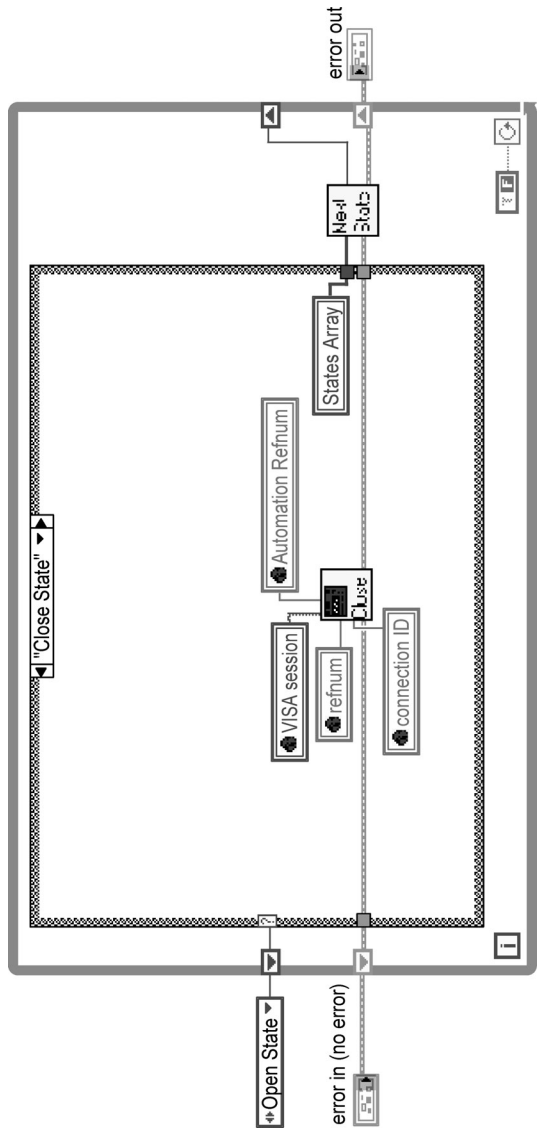
error out
Next State
States Array
Automation Refnum
"Close State"
Close
VISA session
refnum
connection ID
Open State
error in (no error)

FIGURE 6.19

6.5 DEBUGGING CODE

The techniques described in the previous sections for exception handling can be utilized for debugging LabVIEW code. Error detection is very valuable during the testing phase of code. Detection assists in finding where and why errors occurred. Bugs are faults in the code that have to be eliminated. The earlier bugs are found, the easier they are to fix. This section covers some LabVIEW tools that facilitate the process of debugging VIs. First, broken VIs and the error list will be discussed. A description on how to utilize execution highlighting along with the step buttons will follow. Then, the probe tool, the use of breakpoints, and suspending execution will be described. Data logging and NI Spy will then be presented. Finally, tips on utilizing these tools to debug programs will be provided.

6.5.1 Error List

A broken Run button indicates that a VI cannot be executed. A VI cannot be run when one or more errors exist in the code. Errors can be the result of various events such as bad wires or unwired terminals in the code diagram. You may also see a broken Run button when you are editing the code diagram. However, when you are finished coding, the Run button should no longer be broken. If the Run button is broken, you can find out more information on the errors that are preventing the VI from executing by pressing the Run button. Figure 6.20 shows the Error List window that appears.

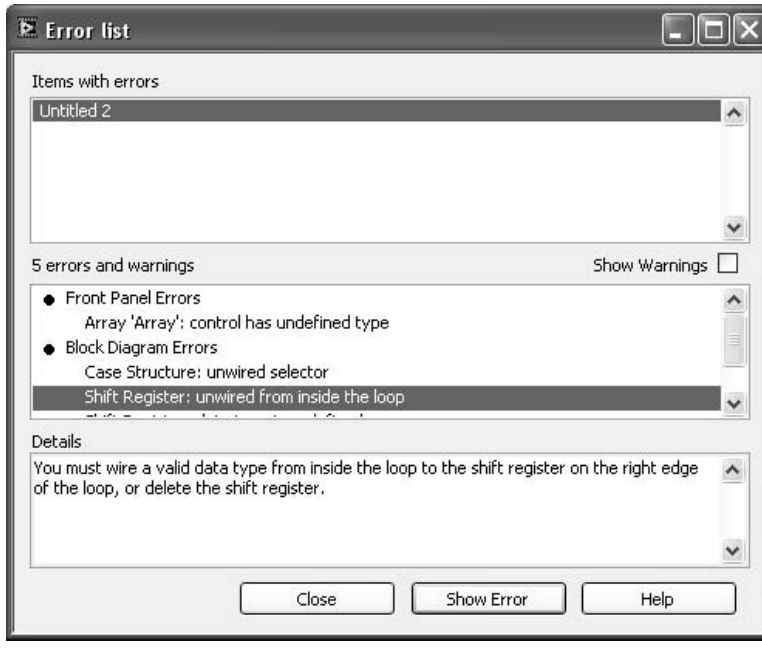

FIGURE 6.20

At the top of the Error List window is a list box that lists all of the VIs that contain errors and warnings. A box that lists all of the errors in each VI can be found just under this. Both front panel and block diagram errors will be listed. The list describes the nature of the errors. When an item in the list is selected, a text box below the list gives more information on the error and how it can be fixed. The Show Error button will find and highlight the cause of the error that is selected. There is also a checkbox, Display Warnings, which will list the warnings for the VI. The warnings do not prevent the VI from executing, but are recommendations for programming. You can set it to display warnings by default by selecting the corresponding checkbox in your Preference settings in the Edit menu.

Using the Error List, you can effectively resolve all of the errors that prevent the VI from running. Once all of the errors have been dealt with, the Run button will no longer be broken. The Error List provides an easy way to identify the errors in your code and determine the course of action to eliminate them.

6.5.2 Execution Highlighting

The Error List described above helps you to resolve the errors that are preventing a VI from running. But it does not assist in identifying bugs that are causing the program to produce unintended results. Execution Highlighting is a tool that can be used to track down bugs in a program. Execution Highlighting allows you to visually see the data flow from one object to the next as the VI runs. The data, represented by bubbles moving along the wires, can be seen moving through nodes in slow motion. The G Reference Manual calls this "animation." This is a very effective tool that National Instruments has incorporated into LabVIEW for debugging VIs. As LabVIEW is a visual programming language, it makes sense to incorporate visual debugging tools to aid programmers.

If you do not see data bubbles, perhaps your Preference settings have not enabled this option. By default, this option is activated. Select Preferences from the Edit pull-down menu, and choose Debugging from the drop-down menu. Make sure the box is checked to show data bubbles during Execution Highlighting.

Pressing the button with the light bulb symbol, located on the code diagram toolbar, will turn on Execution Highlighting. When the VI is run, the animation begins. Execution Highlighting can be turned on or off while the VI is running. Highlighting becomes more valuable when it is used in single-stepping mode. The speed of execution of the program is greatly reduced so you can see the animation and use other debugging tools while it is running.

6.5.3 Single-Stepping

Single-Stepping mode can be enabled by pressing the Pause button. This mode allows you to utilize the step buttons to execute one node at a time from the code diagram. Additionally, when Execution Highlighting is activated, you can see the dataflow and animation of the code while executing one node at a time. The Pause button can be pressed or released at any time while the VI is running, or even before it starts running. You can also press one of the step buttons located next to the

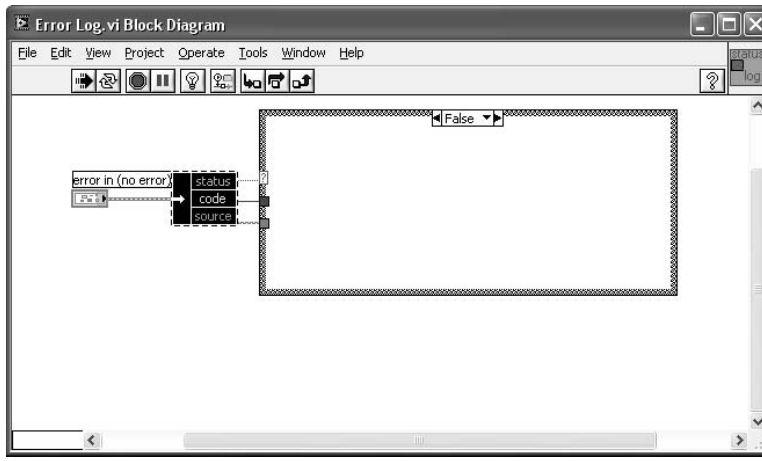

FIGURE 6.21

Execution Highlight button to enter Single-Stepping mode. The Pause button will become active automatically when these are used.

When the VI is in Single-Stepping mode, the three step buttons on the code diagram toolbar are used to control execution of the program. Depending on the code diagram, the step buttons will perform different actions. Use the Simple Help to determine what each button will do at a specific node on the code diagram. Simple Help can be accessed through the Help menu. When the cursor is placed over the step buttons, a description of their function will pop up. Figure 6.21 shows the Error Log VI in single-stepping mode with Execution Highlighting activated. The three step buttons can also be seen in this diagram.

The first step button on the toolbar is used for stepping into a particular structure or subVI. The structure or subVI will also be in Single-Stepping mode. You must then use the step buttons to complete the structure or subVI. The second button is used for stepping over objects, structures, and subVIs. If this button is pressed, the structure or subVI will execute and allow you to begin stepping again after its completion. The third button is used to complete execution of the complete code diagram. Once pressed, the remaining code will execute and not allow you to step through single objects unless Pause is pressed again.

6.5.4 Probe Tool

The Probe Tool can be accessed through the Tools palette or through the menu by popping up on a wire. The Probe Tool is used to examine data values from the wires on the code diagram. When a wire is probed, the data will be displayed in a new window that appears with the name of the value as the title. The probes and wires are numbered to help keep track of them when more than one is being used. You can probe any data type or format to view the value that is being passed along the wire. For example, if a cluster wire is being probed, a window with the cluster name appears displaying the cluster values. The values will be displayed once the

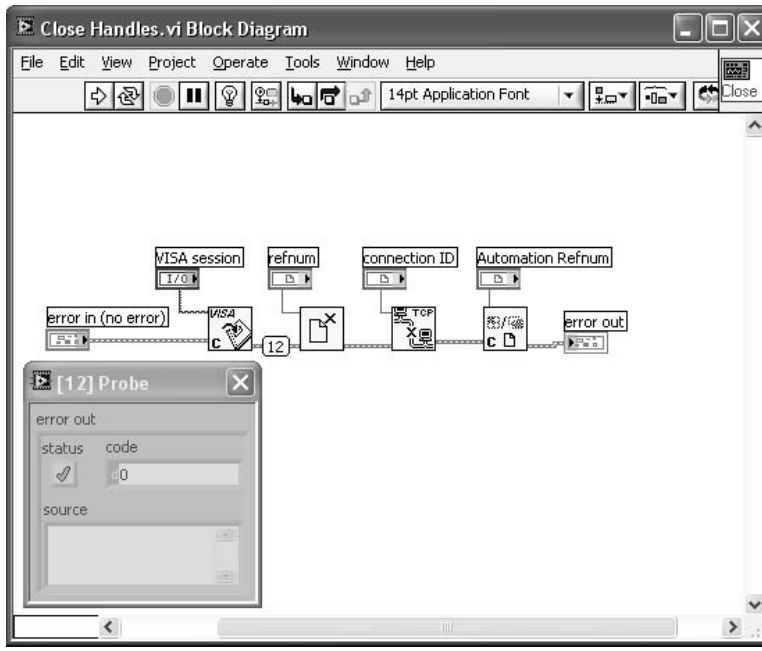

FIGURE 6.22

data has passed the point on the wire where the probe was placed when the VI was running.

The Probe Tool is very valuable when debugging VIs because it allows you to examine the actual data values that are being passed along the wires. If you are getting unexpected results or errors, you can audit values to ensure that they are correct. This tool helps you find the root of the problem. Figure 6.22 illustrates a probe on the error cluster between the VISA Close VI and the File Close VI. The wire is marked with a number, as is the window displaying the cluster values.

By default, auto probing is active in Execution Highlighting mode. This causes LabVIEW to display data values at nodes while Execution Highlighting is on. However, the complete data cannot always be viewed in this manner and is only useful for simple verification purposes. The Probe Tool will still be needed for data types such as clusters and arrays. Auto probing can be enabled or disabled from the same Preferences window as the data bubbles discussed earlier.

The conditional probe is one of the best debugging tools that have been added to LabVIEW. A conditional probe for a Numeric double precision value is shown in Figure 6.23. The data tab contains the typical information a probe has always contained — the value on the wire. The condition tab, however, is the new feature that adds a tremendous amount of value to the use of the probe in debugging.

Programmers that have used Visual Studio for C++ programming have had one very useful tool, called ASSERT. This function is a macro in C++ and part of the debugging object in Visual Basic, and this function is essentially a sanity check. The ASSERT function uses a Boolean expression as an argument. If the argument

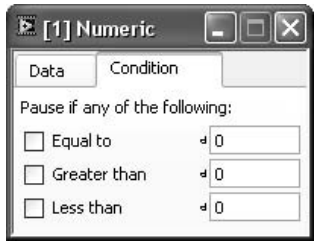

FIGURE 6.23

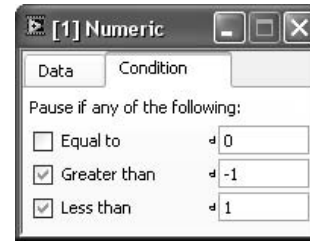

FIGURE 6.24

provided evaluates to true, then nothing happens. If the argument evaluates to false, the macro stops the program and provides a popup box telling where the assertion failure occurred so a developer can examine what issues could lead to the sanity check failure. This type of tool has become available to LabVIEW with the addition of the conditional probe.

The conditions available on conditional probes vary with the type of probe; not all wire types have conditional probes. For example, wires related to the communications pallet or the matrix control do not have conditional probes. Figure 6.24 shows the conditionals of a numeric conditional probe set to perform range checking. Each condition can be enabled with a different value. Figure 6.24 shows the probe configured to verify the range is between –1.0 and 1.0. If the greater or less than condition resolves to true, the program will pause and the developer has the ability to poke around the code diagram. This would allow a developer to set the conditional breakpoint to verify exception handling code catches issues and processes them as expected.

Conditional probes also function as a conditional breakpoint which makes it arguably the most flexible debugging tool in the LabVIEW toolbox.

6.5.5 Breakpoint Tool

The Breakpoint Tool is another debugging device accessible through the Tools palette. As the name suggests, the Breakpoint Tool allows you to set a breakpoint on the code diagram. Breakpoints can be set on objects, VIs, structures, or wires. A red frame around an object or structure indicates a breakpoint has been set, whereas a red dot represents a breakpoint on a wire. Breakpoints cause execution of the code to pause at the location where it has been set. If it is a wire, the data will pass the breakpoint before execution is paused. A breakpoint can be cleared using the same tool that is used to set it.

Breakpoints are valuable because they let the user pause the program at specific locations in the code. The program will execute in its normal manner and speed until it reaches the breakpoint, at which point it will pause. The code that is suspect can then be debugged using Single-Stepping mode, Execution Highlighting, and the Probe Tool.

Once a breakpoint has been set, the program will pause at the break location every time it is executed. You must remember to clear the breakpoint if you do not want the program to pause during the next iteration or execution. If you save the VI

while a breakpoint has been set, the breakpoint will be saved with the VI. The next time you open the VI and run it, execution will pause at the break location. You can use the Find function to locate any breakpoints that have been set.

Breakpoints are non-conditional, meaning whenever a breakpoint is encountered, the program is stopping. If conditional breakpoints are desired, the conditional probe is preferable to a breakpoint.

6.5.6 Suspending Execution

You can force a subVI to suspend execution, for debugging purposes, when it is called. This can be done using one of the following three methods. The first method is to select Suspend when Called from the Operate menu. The second method is to pop up on the subVI from the code diagram of the caller and select SubVI Node Setup. Then, check the box Suspend when Called. Alternatively, you can pop up on the icon while the subVI is open and select VI Setup. Then check the box Suspend When Called.

When you cause a subVI to suspend execution, its front panel will be displayed when it is called. The subVI also enters a special execution mode when it is suspended. The Run button begins execution of the subVI. When a subVI is suspended, it can be executed repeatedly by using the Run button. To the right of the Run button is the Return to Caller button. Once suspended, you can use Execution Highlighting, Single-Stepping, and the Probe Tool to debug the subVI. When you use Single-Stepping while a subVI is suspended, you can skip to the beginning and execute the VI as many times as needed.

6.5.7 Data Logging

Data Logging is another LabVIEW built-in tool that can be used for debugging purposes. Front panel data can be logged automatically by enabling Log at Completion from the Operate menu. When the VI is run the first time, a dialog box will appear, prompting the user to enter a filename for storage. Alternatively, a log file can be selected before running the VI, by selecting Log from the Data Logging submenu in the Operate menu. When the filename is selected prior to running the VI, the dialog box will not appear. The front panel data is entered into that log file after the VI executes.

The Data Logging feature is a method for saving data from tests, similar to a database. LabVIEW enters a date and time stamp, along with the data for the indicators and controls from the front panel. The data can then be viewed by selecting Retrieve from the Data Logging submenu. Figure 6.25 illustrates how the data appears when data is logged and retrieved using this feature. This is a simple front panel with two controls and two indicators. The multiplication and addition results of the two integer controls are displayed in the indicators. This is how the data will be displayed when it is retrieved from the log file. The time and date stamp appears at the top, along with controls for scrolling through the records and deleting records.

Data Logging is useful for saving data values from tests and for debugging VIs. It serves as a mechanism for quickly saving data from specific VIs that are being

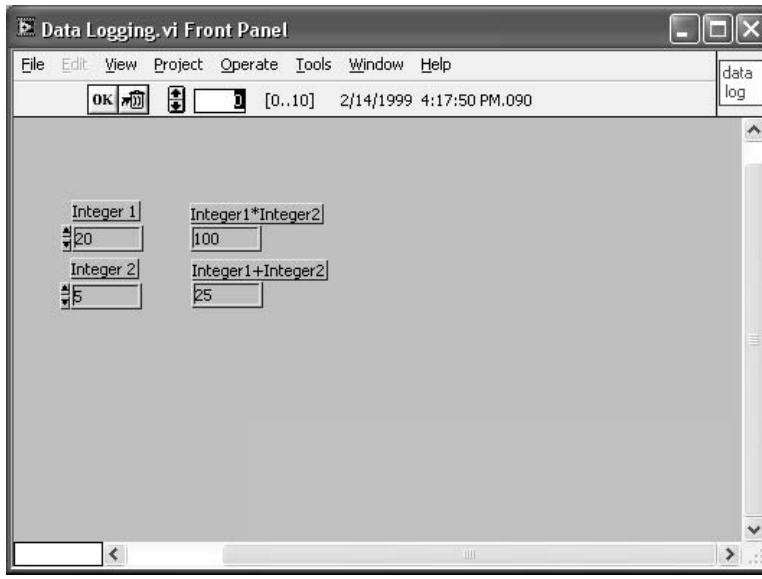

FIGURE 6.25

debugged. The saved data log can then be reviewed for suspect values. The data log is also useful for monitoring intermittent problems with VIs. The front panel data can be saved, retrieved, and purged as needed.

6.5.8 NI Spy/GPIB Spy

These two utilities are very similar and are both used as debugging tools on Windows operating systems. NI Spy monitors the calls that are made by applications to NI-488.2, NI-VISA, IVI, and NI-VXI drivers. Similarly, GPIB Spy tracks any calls that are made to GPIB drivers. They are useful for determining the source of communication errors, whether they are related to general communication problems or are application specific. They help you verify that communications with an instrument are correct. However, when either of these applications are running, they will degrade the speed of your application. Use them only when you are debugging your program to free up system resources, especially if execution time is a consideration for the application.

NI Spy displays the index number assigned to the call, a description of the operation and parameters, and the time that it occurred. The tool displays the calls as they are made during the execution of your application. Errors are immediately highlighted to indicate failures. NI Spy also allows you to log the activity for review at a later time.

GPIB Spy monitors calls to the Windows GPIB driver by Win32, and displays them while your application is executing. All errors or failures are highlighted for quick identification. You can view each call as it is made and see the results, including any timeouts. This utility can be used to verify that your application is sending the

right calls to the Windows GPIB driver. GPIB Spy lists an index number of the call, the names of the GPIB calls, output of the status word ibsta after the call, output of the error word iberr, output of the count variable ibcntl, and the time of each call. All of these contain useful information on the performance of the application. You can view detailed information by using the Properties button on the toolbar.

Familiarization with GPIB, NI-488.2, and the ANSI/IEEE 488.2 communication protocol may be necessary to fully utilize and understand the debugging features on both GPIB Spy and NI Spy. A discussion of IEEE 488.2 is beyond the scope of this book.

6.5.9 Utilization of Debugging Tools

The Error List, Execution Highlighting, Single-Stepping mode, Probe Tool, Breakpoint Tool, and suspending execution were described in the previous sections. These built-in LabVIEW features are very effective for debugging code when they are used in conjunction with on-line Help. Each one is a weapon the programmer can use for tracking down and resolving problems. These tools are summarized in Table 6.2 which lists the tool, its application or use, and how to access or enable it.

The software process model being followed determines when the debugging or testing phase for the code begins. In an iterative model, debugging is involved in each cycle of the process. In the Waterfall model, debugging is done only during one phase of the development cycle. In either case, the first action is to eliminate the errors that prevent the VI from running. As already described, the Error List will assist in removing these errors to allow the VI to run. This part of debugging should

TABLE 6.2
Debugging Tools

Tool	Application	Accessing
Error List	Used to list, locate, and resolve errors that prevent a VI from running.	Press broken Run button.
Execution Highlighting	Used to animate, visualize data flow along wires on code diagram.	Press highlight button with bulb.
Single-Stepping Mode	Allows execution of one node at a time.	Use Pause button.
Probe Tool	Displays data values passed along wires.	Available from Tools palette.
Breakpoint Tool	Halts execution of program at specific location.	Available from Tools palette.
Suspending Execution	Suspends subVI for repeated execution during debugging.	Use Operate menu, SubVI node setup by popping-up on icon or VI setup while VI is open.
Data Logging	Enables front panel data logging to file.	Use Operate menu and Data Logging submenu.
GPIB Spy/NI Spy	Monitor calls to Windows drivers.	Run application.

be performed as the application is being developed, regardless of the model being used. Getting rid of errors that prevent VI execution should be considered part of the coding phase in LabVIEW. This is analogous to syntax errors in traditional languages that are pointed out to the programmer during coding. The Error List makes this easy for even the novice programmer. It guides the programmer in resolving errors quickly.

If it is possible, try to test one VI at a time. Test the individual drivers and subVIs separately before attempting to run the main or executive. You may be overwhelmed when you try to debug a large program with many subVIs. Not only is it easier to concentrate on smaller parts of the program, but you reduce the errors that may be caused through the interaction of the subVIs with each other. A modular design approach with VIs that are specific and self-contained simplifies testing. This interaction through data flow may make it appear that more errors exist. You may also be able to create a simulator for I/O or other portions of the code that have not yet been prepared. Again, this will help in isolating problems related to the specific code at hand without having to deal with I/O errors.

Once the VI can be executed, the next step is to run it with Execution Highlighting enabled. The animation helps you see the data flow on the code diagram. Execution Highlighting will help you find bugs caused by incorrectly wired objects. While the VI is running, make sure that the code executes in the order that was intended, which can be identified with highlighting.

You may also want to probe certain wires with Execution Highlighting and make sure that the values are correct by using the Probe Tool. For instance, probing the error cluster between two objects or VIs will help narrow down where the error is generated. You will see the value of the Probe Tool for debugging once you begin to use it. The Probe Tool and Execution Highlighting can be used in Single-Stepping mode. Single-stepping mode lets you look at a section of code in even more detail to find the problems that exist.

If problems persist, a few suggestions are offered here for you to consider. These might seem basic, but they are the ones that are easy to overlook. First, make sure that the input values provided by the user controls are valid. The Probe Tool can be used to perform this check from the code diagram. When these input values are out of the acceptable range, the code will not execute as intended.

If you are performing communications with an external device, file, or application, check the commands or data sent. The device may not respond to unexpected commands. During this process, also check for correct file names, handles, and addresses. Examine the external device to see if it is functioning properly, and manually perform the actions you are trying to take through automation. Consider using delays in the program if the external device is not responding quickly. Investigate the execution order of your code to ensure that the correct sequence of events is occurring. Race conditions can result if the code is not executing as intended.

Inspect arrays for correct usage of indices. Arrays, lists, rings, and enumerated types all start off at zero and can cause potential problems if not accounted for. During this inspection, check case structures that are driven by these values to see if they correspond. Also make sure that you have a default case set up to ensure the

correct code is executing. You should also examine loop structures to make proper use of shift registers so data is not lost. This includes proper initialization of the shift registers.

Set personal time limits for how long you will attempt to determine where an error exists in code. It becomes very frustrating to attempt to debug a section of code for hours. When your time limit expires, a second opinion should be brought in. This second perspective will see the programming problem differently and may well propose a solution or at least ask questions that may lead you to a solution.

6.5.10 Evaluating Race Conditions

LabVIEW has always had inherent multitasking capabilities. In LabVIEW 5.0, multithreading became available. As the combination of hardware, operating systems, and software advances capabilities, new and exciting programming problems evolve in step with the newest technologies. This section will outline steps that can be taken to understand race conditions that can occur in LabVIEW code that leverages LabVIEW's parallel execution. Chapter 9 discusses multithreading in detail; this section will cover a more generic set of exceptions which are race conditions.

A race condition is simply when two branches of code are attempting to use the same set of data. Generally, a race condition involves one branch of code getting access to data in an order that was not intended. As an example, one branch of code might be coordinating instrument communications and is in the process of reading a trace from an instrument. A second branch of code is attempting to access the trace for analysis. The intention of the developer is to have the trace be read, stored, and then signaled to the analysis branch. A race condition exists if it is possible for the analysis branch to read the array before the communications branch has had the ability to write the array in memory.

Race conditions create special issues that may not be identified by typical debugging tools, and the reason is race conditions are timing related; most debugging tools do not give the developer a view of what is going on without altering the timing of the application. Single-stepping or execution highlighting most certainly changes execution timing in very profound ways. It will be almost impossible to identify a race condition in an application when the code is being stepped through.

Application logging can help identify some issues. In very high performance applications, it may disguise some issues though.

One way to view race conditions is they are not entirely deterministic. There are some probabilities that become involved. Essentially every VI, or VI subsystem, has a probability that it will be executing on a processor core at a given period of time. When application logging is used such as writing debugging information to a file, the amount of code running in a VI has been changed relative to the rest of the application. The code which is used to write the log file is the "new" code which is impacting application timing. This timing change can make some race conditions less likely to appear.

One way that we have successfully used to troubleshoot multithreaded C++ code that is applicable to LabVIEW is to draw out trees of the call stacks. Examine the LabVIEW application. Are the branches of code running in different subsystems?

If so, draw out the hierarchy of each subsystem. Record which VIs are called, and what data the VI is accessing. As the programmer, you will know which VIs have fairly small execution time, and which VIs have longer execution times. The smaller execution time of a VI, the lower the probability it is running concurrently with another specific VI.

Once you have the hierarchies drawn out for each branch or subsystems, examine the data accessed. Any data that is accessed by multiple branches is suspect of causing a race condition. Some common data may only be read by all branches. For example, at application start up, initial configuration information may be read in and stored. This type of data is unlikely to cause issues after application startup. Any data that is commonly accessed that can be written to by any of the branches may be a race condition.

In general, this level of analysis is not required for strictly LabVIEW code. If an application using external code such as .NET objects external code segments is exhibiting behavior such as crashing, examine which branches of code are accessing the external code. It is entirely possible, especially if LabVIEW's subsystems are being used, that there is a thread-related race condition. A semaphore can be used to restrict access to the code or object. Every branch or subsystem will need to use the semaphore to access the suspect code object.

In general, semaphores should be used to protect very specific areas — not entire sets of VIs. If entire sets of VIs are blocked off with a semaphore, then there is a very strong chance that application performance is going to be degraded. For example, code surrounding direct read or writes to a suspect object should be blocked off. Processing data that has been read or preparing data that is about to be written should be done outside the confines of the semaphore.

In many cases, access restrictions such as semaphores or notifications will resolve these types of problems. On occasion, a set of troublesome code will show up that access restriction does not fix. A last resort would be to set the suspect code to run in the User Interface subsystem of the application. This will force all calls to the code to be done by specific threads of the LabVIEW runtime engine. Forcing particular threads to be used is discussed in Chapter 9, multithreading.

6.6 SUMMARY

When you are developing an application, it may be easier to just omit code to perform error detection and handling because it requires extra work. However, exception handling is needed to manage the problems that may arise during execution. An exception is something that might occur during the execution of a program. These unintended events, or exceptions, must be dealt with in the appropriate manner. If exceptions are left unattended, you can lose control over the program, which may result in more problems.

An exception handler allows programmers to deal with situations that might arise when an application is running. It is a mechanism to detect and possibly correct errors. LabVIEW provides some built-in tools to aid the programmer in error detection and handling, but it is the responsibility of the programmer to implement the exception handling code. Several methods for dealing with errors were described in

this chapter. The topics discussed will assist the programmer in writing more robust code through the implementation of exception handlers.

Exception handling should be considered at an early phase of application development. It is appropriate to take exception handling into account when the structure or architecture of the application is being decided upon. Better applications can be developed when exception handling is a forethought, not an afterthought. Exception handling, when built into an application, will lead to sound and reliable code.

BIBLIOGRAPHY

G Programming Reference, National Instruments, Austin, TX, 1999.
Professional G Developers Tools Reference Manual, National Instruments, Austin, TX.
LabVIEW Function and VI Reference Manual, National Instruments, Austin, TX.
LabVIEW On-line Reference, National Instruments, Austin, TX.

7 Shared Variable

This chapter discusses the shared variable concept released in LabVIEW 8. The shared variable is a standard feature for the language that provides a service for distributed data throughout LabVIEW. The Data logging and Supervisory Control module extends the level of functionality. Shared variables are available for use on any operating system and real time (RT) targets. The shared variable engine itself requires a Windows installation or an RT target.

This chapter is devoted to a discussion of the shared variable and includes a discussion of the shared variable engine and communications employed to distribute data. On the surface, it may seem questionable to devote an entire chapter to a distributed data system. As this chapter progresses, it will become clear that the shared variable is not a simple service.

7.1 OVERVIEW OF SHARED VARIABLES

Shared variables are primarily a mechanism for distributed networking support. In general, distributed applications have processes that perform specific tasks with information that needs to be shared with other elements of the application. The shared variable provides a mechanism to the LabVIEW programmer that abstracts away the complexities of distributed programming. Distributed applications do not make life easier on the developer as they add-in a number of exception cases that need to be considered. For example, how does the application know if messages were lost, what if messages are late, the data is old, or what if it is necessary for IP networks to check that multiple copies of the same message have not been received?

The shared variable engine absolves LabVIEW developers from the vast majority of these issues. From a development perspective, what we primarily need to be concerned about is dragging and dropping the correct shared variable. Networking issues are handled by the shared variable engine and the client. It is possible from a shared variable to determine if the information is "fresh" or if the variable has not been updated in awhile. Network security issues are also addressed in configurable fashions through the shared variable engine and supporting NI security services. Distributed application development is roughly as complex as working with global variables.

Adding shared variables to projects and basic information are provided in Chapter 1 and Chapter 2. This section expands on shared variable mechanics and the benefits and penalties for using them.

Shared variables support essentially every data type that can be found in LabVIEW. Pull-down selection menus in the shared variable configuration screen provide what might be considered an incomplete list until you arrive at the last entry: custom control. Complex clusters can be configured as custom controls which, as stated, make the shared variable capable of supporting any data type that would be needed for distribution.

7.1.1 Single-Process Variables

When creating a shared variable, one of the configuration items is to determine if the variable is network-published or single process. Single-process shared variables do not sound so intuitive. What is the benefit of using a shared variable that is not shared? It would seem not of much benefit at all. Essentially, a single-process shared variable functions much like a global variable. In fact, there is a considerable amount of common code used between the shared single-process variable and the standard LabVIEW global variable. Any place a global variable is used, a single-process shared variable will also work — such as sharing data between two parallel loops.

Using a single process shared variable does have two advantages: The first is the ability to network-publish the variable with a simple configuration change; going from a global variable to a network-published variable requires code rework. The second advantage is the ability to use the time-stamp functionality of the shared variable, which does not exist on the global variable.

There is one significant disadvantage to using the shared variable in a single process. Performance benchmarking performed by National Instruments shows that the reading of a shared variable and global variable are roughly equivalent. The time required to write data to a shared variable, however, is more than for a global variable.

The buffering feature of the shared variable can also be used to emulate the performance of a queue. Similar to replacement for a global variable, this is useful for RT targets and should be used in any queues that are expected to be distributed at a later time.

One key difference between the single process variable and network-published variable is that historical data logging is not available to single-process variables.

7.1.2 Network-Published Variable

There are two supported formats for network-published variables. The shared variable configuration screen only allows the selection of a network-published variable. Data socket bindings are still supported and usable; however, new development should focus on the use of shared variables.

Unlike single-process variables, network-published variables can be buffered. The nomenclature of buffering is somewhat misleading; the variable's buffering functions as a FIFO queue. When using the queue, time-stamping data corresponds to when the particular copy was added into the queue. The use of the buffering feature needs to be carefully designed. All designs involving a queue have one exception case that must be considered: queue overflow. In the case of the shared variable, overflowing the queue is not allowed. Any writes to the queue when it is

full get silently discarded. LabVIEW does not receive an error notification that overflow errors are occurring.

One disadvantage of a network-published variable, compared to its single-process counterpart, is the setting of initial values. Default values for published shared variables are not defined unless the DSC module is installed. When designing an application without the DSC module, a VI needs to step up and write data to that variable at system startup. Otherwise, the nodes subscribed to the variable will be receiving undefined data.

7.2 SHARED VARIABLE ENGINE

Shared variables are published through the shared variable engine. The engine itself is available only on Windows-based machines or RT targets. On Windows, the engine runs as a service on the machine. As a service, there are a few items to the engine that are worth noting. Services do not require a user to be logged in. The service is running as a part of the operating system and is not tagged to a user in the task manager. This allows for the engine to run regardless of who is or is not actively logged into the machine. Unfortunately, services are not allowed to have their own user interfaces; external applications need to run to communicate with the service.

7.2.1 Accessing the Shared Variable Engine

As the shared variable engine is a service, it does not present a direct user interface. There are two ways for an administrator to access the engine: The first interface is through the shared variable manager. The manager is a NI provided application that permits configuration of the engine. The second is the Windows Event Viewer. As a service, the engine logs events to the operating system. The event viewer does not permit configuration of the engine but does allow an administrator to evaluate events the service is generating.

7.2.1.1 Shared Variable Manager

The shared variable manager is shown in Figure 7.1. The manager will be the primary tool for accessing the engine as it gives the administrator the most control over the engine. The manager window consists of three frames which are (1) a list of items, (2) the watch list, and (3) the alarms window. The items list provides a listing of all shared variable libraries available on the system, including RT targets. Each library is listed as an independent process and can be stopped and started independently. This is a core feature on large distribution points which may have libraries for multiple applications running concurrently.

From the variable manager you can add or remove libraries, assuming you have appropriate permissions. By default, LabVIEW installation sets the shared variable services to operate without security restrictions. In Section 7.7, it will be pointed out that leaving this configuration without security is, in general, not advisable.