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. In LabVIEW the result of this division is undefined. LabVIEW returns the result \( \text{Inf} \), or infinity, in this case. 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 sound code by planning ahead for potential problems. The ability of an application to respond to unexpected events is critical. 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, DDE, ActiveX, OLE, PPC, AppleEvent), data acquisition, instrument I/O, or file I/O, there is a possibility that you will encounter related errors.

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 while 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 give odd measurements 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.

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 which 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. This option is available in the drop-down box. The coercion option reduces the need to write code for performing the same task.

### 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 has
occurred. The code is a signed 32-bit 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.

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 indicator, which are usually located on the bottom of the
front panel. The cluster information is passed successively through VIs in an appli-
cation, consistent with data flow programming.

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 checks if an error has occurred. 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 informa-
tion 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 Communication palette VIs (TCP, UDP, DDE, ActiveX,
HiQ), 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. 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, while the Error Out terminal is on the lower right side. This is a convention followed by most 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.

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 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, while the status returned is “false.” Table 6.1 contains a list of the error code ranges. Note that if you are using LabVIEW 5.1, there is an additional list of error codes for VISA provided in the LabVIEW Version 5.1 Addendum Manual that is shipped with the CD.
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.

### Table 6.1 Error Codes

<table>
<thead>
<tr>
<th>Error Type</th>
<th>Code Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>G Function Error Codes</td>
<td>0 to 85</td>
</tr>
<tr>
<td>Data Acquisition VI Error Codes</td>
<td>-10001 to -10920</td>
</tr>
<tr>
<td>Analysis Error Codes</td>
<td>-20001 to -20065</td>
</tr>
<tr>
<td>TCP and UDP Error Codes</td>
<td>53 to 66</td>
</tr>
<tr>
<td>DDE Error Codes</td>
<td>14001 to 14020</td>
</tr>
<tr>
<td>PPC Error Codes</td>
<td>-900 to -932</td>
</tr>
<tr>
<td>LabVIEW Specific PPC Error Codes</td>
<td>1 to 5</td>
</tr>
<tr>
<td>AppleEvent Error Codes</td>
<td>-1700 to -1719</td>
</tr>
<tr>
<td>LabVIEW Specific Error Codes for Apple Events</td>
<td>1000 to 1004</td>
</tr>
<tr>
<td>GPIB Error Codes</td>
<td>0 to 32</td>
</tr>
<tr>
<td>Instrument Driver Error Codes</td>
<td>-1200 to -13xx</td>
</tr>
<tr>
<td>Serial Port Error Codes</td>
<td>61 to 65</td>
</tr>
<tr>
<td>VISA Error Codes</td>
<td>-1073807360 to -1073807202</td>
</tr>
<tr>
<td></td>
<td>1073676290 to 107367443</td>
</tr>
</tbody>
</table>

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. The VISA Status Description VI takes the VISA session and error cluster as inputs and returns the status description of the error that was generated.

When you are using instrument drivers that utilize VISA, there are some additional errors you may encounter. The first cause 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
VISA VIs unless your system has this option installed. Another cause 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.

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 only be used when the program is being monitored. Consider using e-mail for notification using the SMTP add-on package as an alternative. Chapter 8 also shows you how to incorporate the e-mail feature using ActiveX.

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 since 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 also found in the Time & Dialog palette in the Functions menu. 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. Since 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. The following are some VIs that output
only the error code: serial I/O VIs, PPC VIs, AppleEvent 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.

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 happen if the file that is being written to cannot be opened? What happens if the read or write operation fails? 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 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 archit ecting 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 6.4.6 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 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 troubleshoot the error. This usually results in halting execution of the program.

You can get more involved by attempting to correct 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 can be used in LabVIEW 4.1 or older. However, LabVIEW 5.0 has a default case permitting you to wire the code directly to the selector terminal of the case structure. You can use the pop-up menu to select the default case, which is normally Case 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 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 available in LabVIEW 5.0 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.

**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. Since 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.
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 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 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 others 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. 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 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. Since 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 being 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 per-
formed 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.

### 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.

At the top of the Error List window is a drop-down box that lists all of the VIs that contain errors. 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 Find 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.

FIGURE 6.20
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. Since 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 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 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
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 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 being 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.

### 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, while a red dot represents a breakpoint on a wire. Breakpoints cause execution of the code to pause at the location 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.
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 6.23 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 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 95/98/NT 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 \texttt{ibsta} after the call, output of the error word \texttt{iberr}, output of the count variable \texttt{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 the table.

Table 6.2 lists the tool, its application or use, and how to access or enable it.

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

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 the table.

Table 6.2 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 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 being 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 being 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.
6.6 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
LabVIEW Function and VI Reference Manual, National Instruments
LabVIEW On-line Reference, National Instruments