LabVIEW™ Core 3 Course Manual

Course Software Version 2014
November 2014 Edition
Part Number 375510D-01

Trademarks

Refer to the *NI Trademarks and Logo Guidelines* at `ni.com/trademarks` for more information on National Instruments trademarks.

ARM, Keil, and µVision are trademarks or registered of ARM Ltd or its subsidiaries.

LEGO, the LEGO logo, WEDO, and MINDSTORMS are trademarks of the LEGO Group.

TETRIX by Pitsco is a trademark of Pitsco, Inc.

FIELDBUS FOUNDATION™ and FOUNDATION™ are trademarks of the Fieldbus Foundation.

EtherCAT® is a registered trademark of and licensed by Beckhoff Automation GmbH.

CANopen® is a registered Community Trademark of CAN in Automation e.V.

DeviceNet™ and EtherNet/IP™ are trademarks of ODVA.

Go!, SensorDAQ, and Vernier are registered trademarks of Vernier Software & Technology. Vernier Software & Technology and `vernier.com` are trademarks or trade dress.

Xilinx is the registered trademark of Xilinx, Inc.

Taptite and Trilobular are registered trademarks of Research Engineering & Manufacturing Inc.

FireWire® is the registered trademark of Apple Inc.

Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

Handle Graphics®, MATLAB®, Real-Time Workshop®, Simulink®, Stateflow®, and xPC TargetBox® are registered trademarks, and TargetBox™ and Target Language Compiler™ are trademarks of The MathWorks, Inc.

Tektronix®, Tek, and Tektronix, Enabling Technology are registered trademarks of Tektronix, Inc.

The Bluetooth® word mark is a registered trademark owned by the Bluetooth SIG, Inc.

The ExpressCard™ word mark and logos are owned by PCMCIA and any use of such marks by National Instruments is under license.

The mark LabWindows is used under a license from Microsoft Corporation. Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

Other product and company names mentioned herein are trademarks or trade names of their respective companies.

Members of the National Instruments Alliance Partner Program are business entities independent from National Instruments and have no agency, partnership, or joint-venture relationship with National Instruments.

Patents

For patents covering National Instruments products/technology, refer to the appropriate location: **Help»Patents** in your software, the `patents.txt` file on your media, or the *National Instruments Patent Notice* at `ni.com/patents`.

Worldwide Technical Support and Product Information

ni.com

Worldwide Offices

Visit ni.com/niglobal to access the branch office websites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

National Instruments Corporate Headquarters

11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 683 0100

For further support information, refer to the *Additional Information and Resources* appendix. To comment on National Instruments documentation, refer to the National Instruments website at ni.com/info and enter the Info Code feedback.

Contents

Student Guide

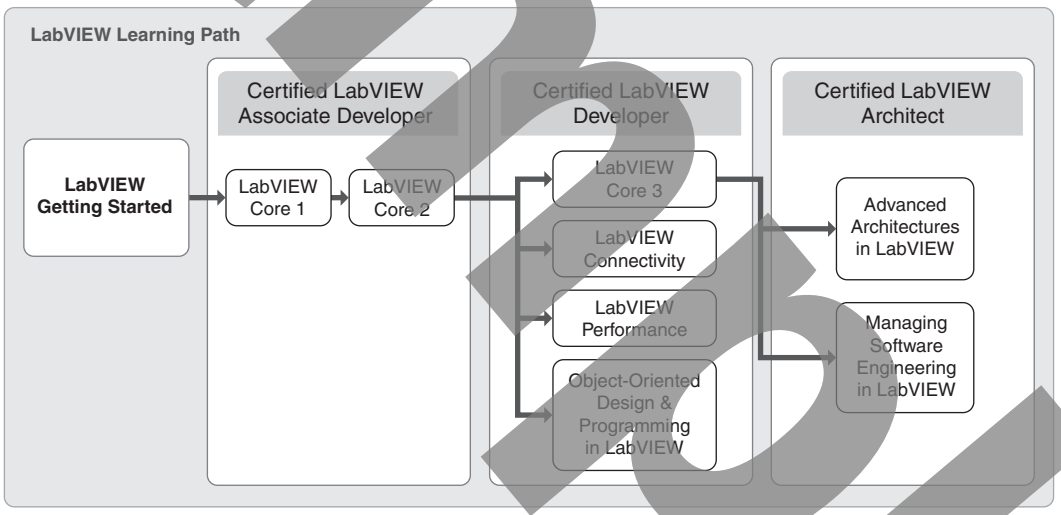
Thank you for purchasing the *LabVIEW Core 3* course kit. This course manual and the accompanying software are used in the three-day, hands-on *LabVIEW Core 3* course.

You can apply the full purchase of this course kit toward the corresponding course registration fee if you register within 90 days of purchasing the kit. Visit `ni.com/training` for online course schedules, syllabi, training centers, and class registration.

Note For course and exercise manual updates and corrections, refer to `ni.com/info` and enter the Info Code `core3`.

A. NI Certification

The *LabVIEW Core 3* course is part of a series of courses designed to build your proficiency with LabVIEW and help you prepare for NI LabVIEW certification exams. The following illustration shows the courses that are part of the LabVIEW training series. Refer to `ni.com/training` for more information about NI Certification.



B. Course Description

LabVIEW Core 3 introduces you to structured practices to design, implement, document, and test LabVIEW applications. This course focuses on developing hierarchical applications that are scalable, readable, and maintainable. The processes and techniques covered in this course help reduce development time and improve application stability. By incorporating these design practices early in your development, you avoid unnecessary application redesign, increase VI reuse, and minimize maintenance costs.

This course assumes that you have taken the *LabVIEW Core 1* and *LabVIEW Core 2* courses or have equivalent experience.

This course kit is designed to be completed in sequence. The course and exercise manuals are divided into lessons, described as follows.

In the course manual, each lesson consists of the following:

- An introduction that describes the purpose of the lesson and what you will learn
- A discussion of the topics in the lesson
- A summary quiz that tests and reinforces important concepts and skills taught in the lesson

In the exercise manual, each lesson consists of the following:

- A set of exercises to reinforce the topics in the lesson
- Some lessons include optional and challenge exercise sections or additional exercises to complete if time permits

Note The exercises in this course are cumulative and lead toward developing a final application at the end of the course. If you skip an exercise, use the solution VI for that exercise, available in the `<Solutions>\LabVIEW Core 3` directory, in later exercises.

C. What You Need to Get Started

Before you use this course manual, make sure you have the following items:

☐ Windows XP or later installed on your computer

☐ LabVIEW Professional Development System 2014 or later

☐ *LabVIEW Core 3* course CD, containing the following folders:

Directory	Description
`Exercises`	Folder containing VIs and other files used in the course
`Solutions`	Folder containing completed course exercises

D. Installing the Course Software

Complete the following steps to install the course software.

1. Insert the course CD in your computer.
2. Follow the prompts to install the course material.

The installer places the `Exercises` and `Solutions` folders at the top level of the root directory. Exercise files are located in the `<Exercises>\LabVIEW Core 3` directory.

Tip Folder names in angle brackets, such as `<Exercises>`, refer to folders in the root directory of your computer.

E. Course Goal

Given a requirements document for a LabVIEW development project, you will follow a software development process to design, implement, document and test the key application features in a manner that satisfies requirements for readability, scalability, and maintainability.

5

Managing and Logging Errors

No matter how confident you are in the VI you create, you cannot predict every problem a user can encounter.

This lesson describes several approaches to developing software that gracefully responds to different types of errors. You will learn how to determine whether an error should be handled locally or globally and when you should log error data to disk for later analysis.

Topics

A. Error Testing

Without a mechanism to check for errors, you know only that the VI does not work properly. It is important to consider how your system will respond to different types of errors when they occur.

Mission-critical applications require more complete error management than simpler LabVIEW applications. Comprehensive error management may increase development time, but it will save troubleshooting time in the long run.

When you develop your error test strategy, the first step is to identify the different errors that are likely to occur. Then determine how the system should respond to each type of error. Finally, develop tests to ensure that the system behaves as expected for each error.

While LabVIEW provides basic tools for error and exception handling, implementing a comprehensive error handling strategy is challenging and requires significant programming effort. A comprehensive error handling strategy requires both the ability to respond to specific error codes, for example: ignoring an error that does not affect the system, and the ability to take general actions based upon the type of error that occurs, for example: log all warnings to a file.

When creating software, you can develop separate error handling code for two areas of the error handling strategy—development and deployment. During development, the error handling code should be noticeable and should clearly indicate where errors occur. This helps you determine where any bugs might exist in a VI. During deployment, however, you want the error handling system to be unobtrusive to the user. The error handling system should allow a clean exit and provide very clear prompts to the user.

Force Errors

Error tests help you determine if the error handling strategy you designed works correctly and make sure the error handling code performs as you expect. Test the error handling code by creating test plans that force errors.

One way to force errors is to modify the calling code to pass invalid values to the subVI that you are testing to see how that subVI handles the error. For example, you could modify your code to:

- Send an invalid message to your message handling loop
- Attempt to write to a read-only file

You might also simulate user errors to see how your application handles those mistakes. For example, you could:

- Cancel a dialog that the user must complete
- Specify an invalid file for input/output

Verify System Behavior

The error test should verify that the proper errors are reported and recovered. The error test plan should also verify that the error handling code handles errors gracefully.

The behavior of the system will be different for different error classifications. Critical errors occur when the system cannot proceed safely. These errors should result in a safe system shutdown. Non-critical errors should be handled and the system should continue execution. A warning may only require user notification.

B. Local Error Handling

By default, as a VI runs, LabVIEW tests for errors at each execution node. If no error occurs, the node executes normally.

If LabVIEW detects an error, the node does not execute. If the error out terminal of that node is unwired, LabVIEW suspends execution of the calling VI, highlights the node and displays an error dialog. If the error out terminal of that node is wired, then LabVIEW passes the error to the next node. The next node will check the status of the error in terminal, skip execution, and either suspend execution of the calling VI or pass the error to the next node, depending on whether or not its error out terminal is wired. At the end of execution, if the error passes all the way through the application, LabVIEW displays an error dialog.

This method is not always the best way to handle errors. For example, you may have some code that must execute regardless of an error (hardware shutdown, for example).

To disable automatic error handling, follow these steps:

1. Navigate to **Tools»Options»Block Diagram**.
2. In the Error Handling section, disable **Enable automatic error handling in new VIs** and **Enable automatic error handling dialogs**.
3. Click **OK**.

You can choose other error handling methods. For example, if an I/O VI on the block diagram times out, you might not want the entire application to stop. You also might want the VI to retry for a certain period of time. In LabVIEW, you can make these error handling decisions on the block diagram of the VI.

You can design corrective error processing that tries to determine and fix the error. This allows each module to implement error correcting code. For example you could develop a Read File I/O VI that can fix errors which might have occurred with the Open/Create/Replace File VI.

Handling errors locally is the primary approach for minor errors that can be prevented or controlled and that should not result in system shutdown. Whenever possible, correct errors in the VI where the errors occur. If you cannot correct an error within the module, return the error to the calling VI so that the calling VI can correct, ignore, or display the error to the user. When you implement a module, make sure it handles any errors that it can. Include error clusters to pass error information into and out of the module.

Preventing Errors

You can design a system that either ensures an error cannot occur or proactively determines if an error will occur.

For example:

- It might be necessary to catch any problems that are in a configuration that is being passed to a data acquisition driver. The system can prevent the data acquisition driver from even executing if there are errors.
- If a control value will be used as the denominator of a divide operation, develop code that checks that the control value is non-zero before the divide operation.
- If a control value must remain between -10 and 10, set control limits on the front panel.

The goal of this technique is to prevent most errors from occurring in the first place.

Handling Specific Errors

Handling specific errors involves performing some sort of action because a specific error occurs. The first step in determining how to handle errors for a VI is to answer the following questions:

- What could go wrong?
- Can I do anything about it?
- Does responding to this error require higher-level code?
- How will the error affect subsequent code?
- How should the error affect subsequent code?

Only handle errors that you can do something about. Any errors that should result in system shutdown should be passed to the top-level application. Once you identify the errors that you can address, there are three main approaches to handling that error locally:

- Ignore the error—Some libraries or functions may generate errors that you expect. For example, the File Dialog VI returns error 43 if the user cancels the dialog. If you want to allow the user to cancel that window, then you should clear and ignore this error.
- Retry the operation that caused the error—If the error is caused by an intermittent communication or connectivity problem, retrying the operation may result in success. In general, it is best to limit the number of retry attempts. If you reach the retry limit, try another method for handling the error.
- Correct the error.

Document any assumptions that you make and errors that you decide to ignore.

Place error handling code as close to the error-generating code as you can to minimize the propagation of the error. This also avoids confusion about which piece of code generates the error. Additionally, some error handling methods, like retrying the operation that caused the error, are only effective when the error is detected and handled immediately.

Clearing Errors

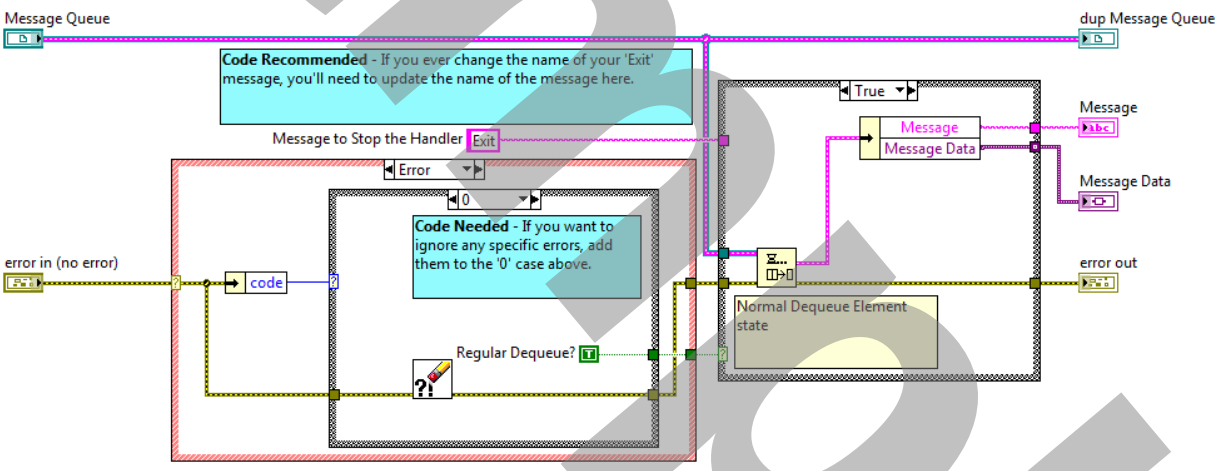
Clear errors only after handling the error locally or passing the information to a global error handler. Methods for clearing errors include using the Clear Errors VI or an empty error cluster constant.

Use the Clear Errors VI if there is additional code that executes after the error is handled. If the error handling is the last piece of code to execute, you can use an empty error cluster constant instead.

Queued Message Handler Local Error Handling

The Dequeue Message VI, shown in Figure 5-1, can ignore errors when it reads data from the message queue. The Error case reads the error code from the **error in** cluster. The case structure reads the error code and clears specific errors. By default, the only error that it clears is error 0 (no error). Modify this case to include any other errors that you want to ignore/clear when this VI executes. The errors to ignore depend on your application. For example, if you are reading the queue over a network, you may want to ignore timeout errors.

Figure 5-1. Dequeue Message VI



C. Global Error Handling

Some errors require action from a higher-level VI. For example, critical errors, which occur when the application cannot continue operating, require a safe system shutdown. You can design a system that passes error data out of the VI to the calling VI or a global handler. If you choose to pass the error out to the calling VI, the caller will have to either handle the error or pass it along to its caller. Errors can come from one of the following two sources:

- LabVIEW functions
- Custom errors defined by the developer

Custom Error Codes

VIs and functions in LabVIEW can return numeric error codes. Each product or group of VIs defines a range of error codes. Refer to the *Ranges of LabVIEW Error Codes* topic of the *LabVIEW Help* for error code tables listing the numeric error codes and descriptions for each product and VI grouping.

In addition to defining error code ranges, LabVIEW reserves some error code ranges for you to use in your application. You can define custom error codes in the range of -8999 through -8000, 5000 through 9999, or 500,000 through 599,999.

Determine where possible errors can occur in the VI and define error codes and messages for those errors. You can create custom error messages that are meaningful for your VIs.

National Instruments recommends that you define custom error codes in the range of 5000 to 9999.

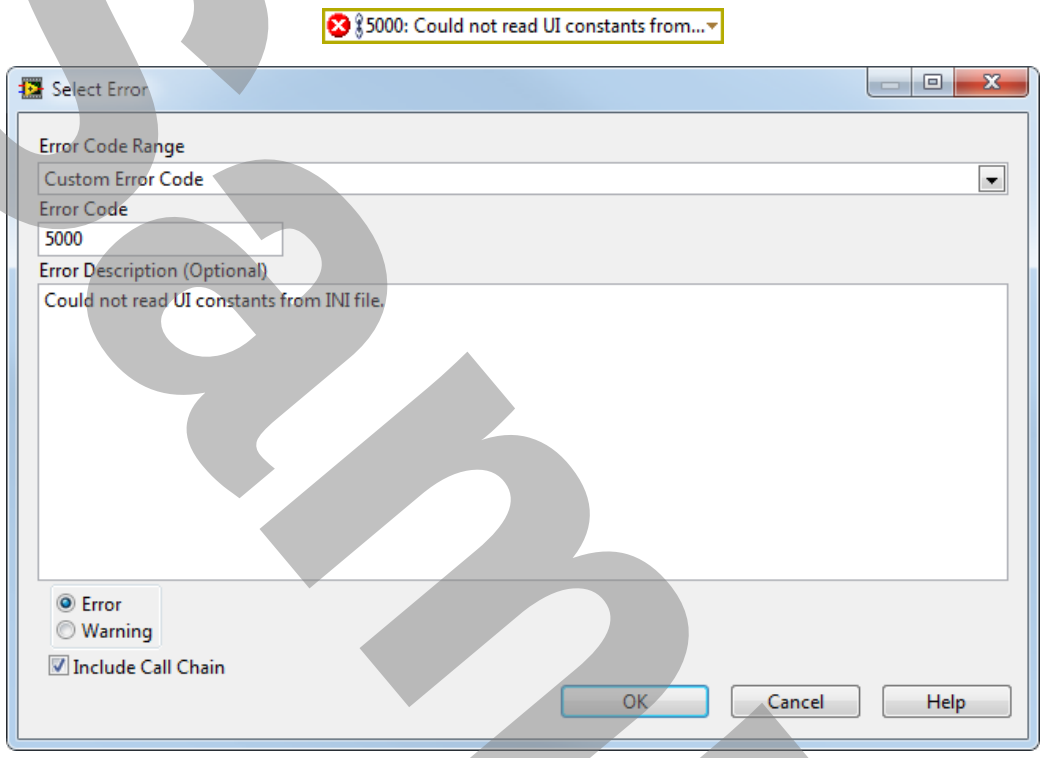
Define Custom Error Codes

The approach that you choose for defining custom error codes depends on how those custom codes need to be distributed.

Error Ring

An Error Ring constant defines a single custom error code for one particular instance in your application. The Error Ring constant allows you to choose an existing LabVIEW error cluster (code and description) or create your own custom error code, as shown in Figure 5-2.

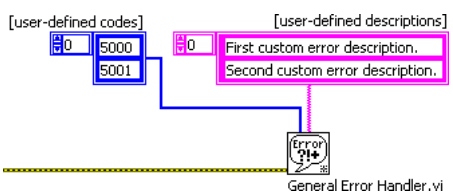
Figure 5-2. Define Errors with the Error Ring Constant



General Error Handler

Use the General Error Handler VI to define multiple custom error codes that your application does not widely use and you will not distribute in built applications or shared libraries. For this VI, create arrays of custom error codes and descriptions for your custom error codes. If an incoming error matches one in user-defined codes, the General Error Handler VI returns the corresponding description from the user-defined description input, as shown in Figure 5-3.

Figure 5-3. General Error Handler VI

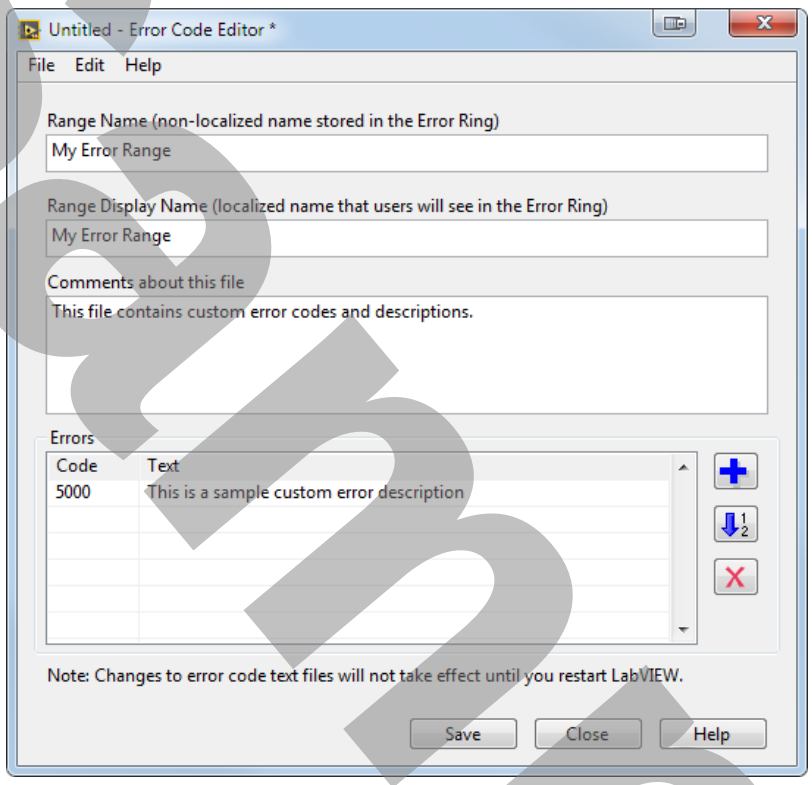


Refer to the *Defining Custom Error Codes Using the General Error Handler VI* topic of the *LabVIEW Help* for more information on using this VI.

Error Code Editor

Use the Error Code Editor to define multiple error codes that you use throughout your application and will distribute in built applications or shared libraries. Error Code File Editor creates an XML-based text file in the `<labview>\user.lib\errors` directory. To launch this dialog, shown in Figure 5-4, select **Tools»Advanced»Edit Error Codes** to launch the Error Code File.

Figure 5-4. Error Code File Editor



When this window launches, LabVIEW prompts you to either create an error code file or edit an existing file.

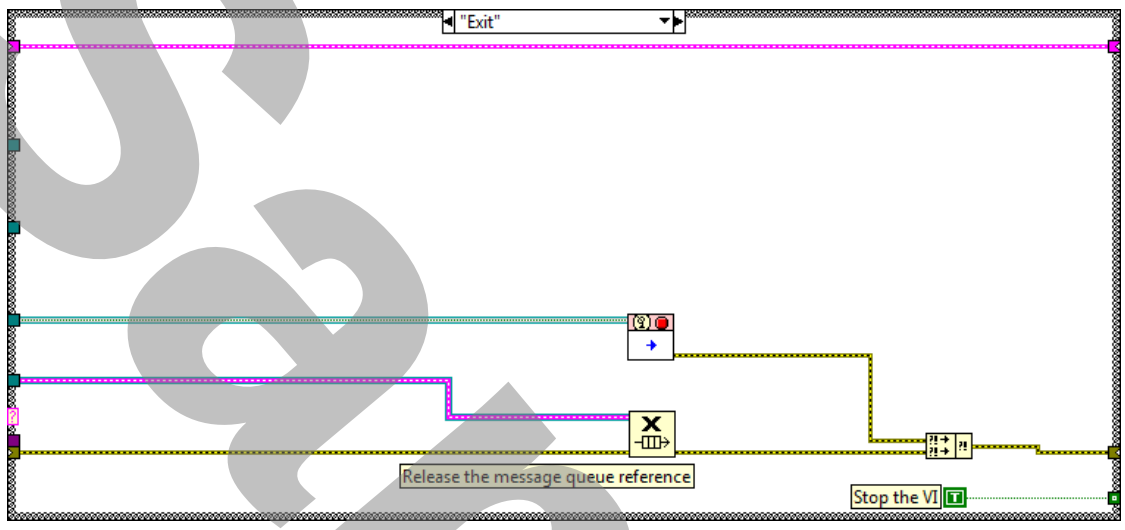
Refer to the *Defining Custom Error Codes to Distribute Throughout Your Application* topic of the *LabVIEW Help* for more information on using this tool.

After you create an error code file, refer to the *LabVIEW Error Code File Locations* Knowledgebase at `ni.com` for more information on how to deploy this file or distribute it to other development systems.

Queued Message Handler Critical Error Handling

If an error is detected in the event handling loop or `Dequeue Message.vi` in the message handling loop, the Exit case of the message handling loop, shown in Figure 5-5, executes.

Figure 5-5. QMH Critical Error Handling



The Exit message is passed out of `Dequeue Message.vi` unless you specify that the error code is a specific one that should be ignored. If you want your MHLs to shut down on a message other than Exit, you must make that change in this VI.

The Exit case of the message handling loop fires the Stop user event to stop the EHL and releases the MHL queue.

Global Error Handler VI

A global error handler consists of high-level code that checks for errors in an entire system. The global error handler should use error classification information (critical, non-critical, warning) to determine which actions to take. The global error handler should also handle the case of an unclassified error or an error with an unrecognized classification and react accordingly. Examples of actions a global error handler might take to respond to error classification (or the lack thereof) include logging the error, displaying error information in an error dialog, placing the system outputs in a safe state, and/or initiating a system shutdown/reboot.

The goal of the global error handler is to handle errors that could come from multiple locations. If an error can come only from one specific subVI, handle that error locally.

Functional global variables are commonly used to implement global error handlers that share the error response across parallel loops. The functional global variable should include at least three cases:

- Initialize—Initialize references and error data. Call this case when the application begins execution.

- Handle Errors—Handle any errors that have been passed to the functional global variable. Pass a clear error cluster out.
- Report Errors—Pass all errors that have occurred to the error out terminal for the functional global variable.

For the global error handler to be effective, you must select a method for communicating errors to that VI. If your global error handler is a functional global variable, you can call the Handle Errors case from any location in your application that generates errors that must be handled globally. For other approaches, you may use a queue to pass error data.

To practice the concepts in this section, complete Exercise 5-1.

D. Error Logging

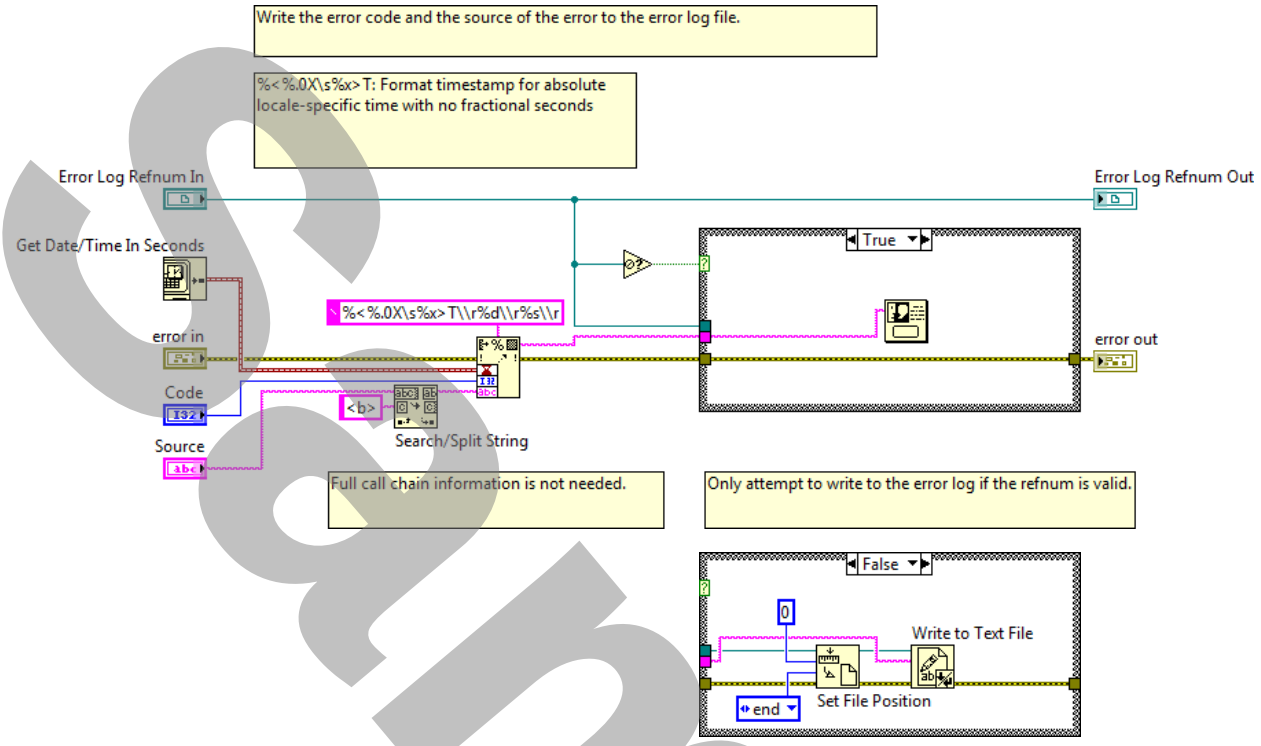
You should log information about critical errors to disk.

Error logging is a benefit for a user and a developer. Users have a historical record of what happened in case of an error and a developer can hopefully discern more about what caused an error. It is logical for the log to be readable by humans (ASCII characters). Because errors may be logged several times during an application's execution or in a quick amount of time (for example, before a operating system crashes), you want your error logging code to be very efficient and high performing. A lot of embedded systems don't have a lot of system resources to hold a large error log so you will want to either clean out the error log every once in a while or just log the most important things.

At minimum, you want to log the time of the error, the error code, and the error source. Depending on your application, you may want to log additional information.

The VI shown in Figure 5-6 shows an example of error logging. The input string input of the Format Into String function describes how each piece of information is written to the string. If the Error Log Refnum is valid, write to the end of the file. If not, then the string is displayed to the user. The reference is only invalid if the application is running from a location where it does not have write access, such as a CD.

Figure 5-6. Error Logging Example



Regardless of whether the error information is logged or displayed in a dialog, the information should be understandable, localizable, and indicate any actions that the user should take.

To practice the concepts in this section, complete Exercise 5-2.

Self-Review: Quiz

1. Which of the following methods are valid ways to generate a custom error code? (multiple answer)
   a. Manually create an errors.txt file
   b. Error Ring constant
   c. Simple Error Handler
   d. General Error Handler

2. What information should be logged for a critical error?
   a. Time of error
   b. Error status
   c. Error code
   d. Error source

Self-Review: Quiz Answers

1. Which of the following methods are valid ways to generate a custom error code? (multiple answer)
   - **a. Manually create an errors.txt file**
   - **b. Error Ring constant**
   - c. Simple Error Handler
   - **d. General Error Handler**

2. What information should be logged for a critical error?
   - **a. Time of error**
   - b. Error status
   - **c. Error code**
   - **d. Error source**

Notes