TestStand™

Using LabWindows™/CVI™ with TestStand

July 2003 Edition
Part Number 323201A-01

Worldwide Technical Support and Product Information

ni.com

National Instruments Corporate Headquarters

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

Worldwide Offices

Australia 1800 300 800, Austria 43 0 662 45 79 90 0, Belgium 32 0 2 757 00 20, Brazil 55 11 3262 3599, Canada (Calgary) 403 274 9391, Canada (Montreal) 514 288 5722, Canada (Ottawa) 613 233 5949, Canada (Québec) 514 694 8521, Canada (Toronto) 905 785 0085, Canada (Vancouver) 514 685 7530, China 86 21 6555 7838, Czech Republic 420 2 2423 5774, Denmark 45 45 76 26 00, Finland 385 0 9 725 725 11, France 33 0 1 48 14 24 24, Germany 49 0 89 741 31 30, Greece 30 2 10 42 96 427, India 91 80 51190000, Israel 972 0 3 6393737, Italy 39 02 413091, Japan 81 3 5472 2970, Korea 82 02 3451 3400, Malaysia 603 9131 0918, Mexico 001 800 010 0793, Netherlands 31 0 348 433 466, New Zealand 1800 300 800, Norway 47 0 66 90 76 60, Poland 48 0 22 3390 150, Portugal 351 210 311 210, Russia 7 095 238 7139, Singapore 65 6226 5886, Slovenia 386 3 425 4200, South Africa 27 0 11 805 8197, Spain 34 91 640 0085, Sweden 46 0 8 587 895 00, Switzerland 41 56 200 51 51, Taiwan 886 2 2528 7227, Thailand 662 992 7519, United Kingdom 44 0 1635 523545

For further support information, refer to the *Technical Support and Professional Services* appendix. To comment on the documentation, send email to techpubs@ni.com.

Important Information

Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this document is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

Trademarks

Patents

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

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

(1) NATIONAL INSTRUMENTS PRODUCTS ARE NOT DESIGNED WITH COMPONENTS AND TESTING FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH SURGICAL IMPLANTS OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY TO A HUMAN.

(2) IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS, SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR APPLICATIONS DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREAFTER COLLECTIVELY TERMED "SYSTEM FAILURES"). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONIC SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE, INJURY, OR DEATH, THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES, INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM NATIONAL INSTRUMENTS' TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

Conventions

The following conventions are used in this manual:

<>	Angle brackets that contain numbers separated by an ellipsis represent a range of values associated with a bit or signal name—for example, DIO<3..0>.
»	The » symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options** directs you to pull down the **File** menu, select the **Page Setup** item, and select **Options** from the last dialog box.
	This icon denotes a note, which alerts you to important information.
	This icon denotes a tip, which alerts you to advisory information.
bold	Bold text denotes items that you must select or click in the software, such as menu items and dialog box options. Bold text also denotes parameter names.
italic	Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text that is a placeholder for a word or value that you must supply.
`monospace`	Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames and extensions, and code excerpts.

Contents

Chapter 5
Configuring the LabWindows/CVI Adapter

Chapter 6
Creating Custom User Interfaces in LabWindows/CVI

Appendix A
Adding Type Libraries to LabWindows/CVI DLLs

Appendix B
Using the TestStand ActiveX APIs in LabWindows/CVI

Appendix C
Calling Legacy Code Modules

1 Introduction

This chapter discusses how National Instruments TestStand and National Instruments LabWindows™/CVI™ work together in a test system.

The Role of LabWindows/CVI in a TestStand-Based System

TestStand is a test management environment that you use to organize and execute code modules written in a variety of languages and application development environments (ADEs), including LabWindows/CVI. It handles core test management functionality such as the definition and execution of the overall testing process, user management, report generation, database logging, and more. TestStand can work in a variety of different testing scenarios and environments because it allows extensive customization of components like the process model, step types, and operator interfaces. You can use LabWindows/CVI to accomplish much of this customization in the following ways:

- Create code modules, such as tests and actions, that TestStand can call using the LabWindows/CVI Adapter
- Create custom user interfaces for your test system
- Create custom step types

Code Modules

TestStand can call LabWindows/CVI code modules with a variety of function prototypes. TestStand can also pass data to the code modules it calls and store the data that the code modules return. Additionally, the code modules that TestStand calls can access the complete TestStand application programming interface (API) for advanced applications.

Operator Interfaces

You can use LabWindows/CVI to build custom user interfaces for your test systems. Typically, these custom user interfaces are operator interfaces designed for use in production test systems. The full power of the LabWindows/CVI development environment allows you to customize these interfaces to meet your exact requirements. Refer to Chapter 9, *Creating Custom Operator Interfaces*, of the *TestStand Reference Manual*, for general information about creating custom operator interfaces.

Custom Step Types

You can use LabWindows/CVI to create code modules that you call from custom step types. These code modules can implement editable dialog boxes and other features of custom step types. Refer to Chapter 13, *Creating Custom Step Types*, of the *TestStand Reference Manual* for more information about custom step types.

LabWindows/CVI Adapter

The LabWindows/CVI Adapter offers advanced functionality for calling code modules from TestStand. Many of the features in the LabWindows/CVI Adapter were previously only available when you used the TestStand 2.0 DLL Flexible Prototype Adapter.

Use the LabWindows/CVI Adapter to perform the following tasks in TestStand:

- Call code modules with arbitrary function prototypes
- Create and edit code modules from TestStand
- Debug code modules (step in/step out) from TestStand
- Run code modules in-process or out-of-process using the LabWindows/CVI development system
- Call code modules from the LabWindows/CVI Test Executive Toolkit

2

Calling LabWindows/CVI Code Modules from TestStand

This chapter discusses how to call LabWindows/CVI code modules from TestStand using the LabWindows/CVI Adapter.

Note All of the tutorials in this manual require that you have LabWindows/CVI and TestStand installed on the same computer. In addition, you must configure the LabWindows/CVI Adapter to execute steps in an external instance of the LabWindows/CVI development system and to allow only new templates. Refer to Chapter 5, *Configuring the LabWindows/CVI Adapter*, for more information about configuring these settings for the adapter.

Introduction to the Edit LabWindows/CVI Module Call Dialog Box

Use the Edit LabWindows/CVI Module Call dialog box to configure calls to LabWindows/CVI code modules. To launch this dialog box, which is shown in Figure 2-1, select **Specify Module** from the context menu of any step that uses the LabWindows/CVI Adapter.

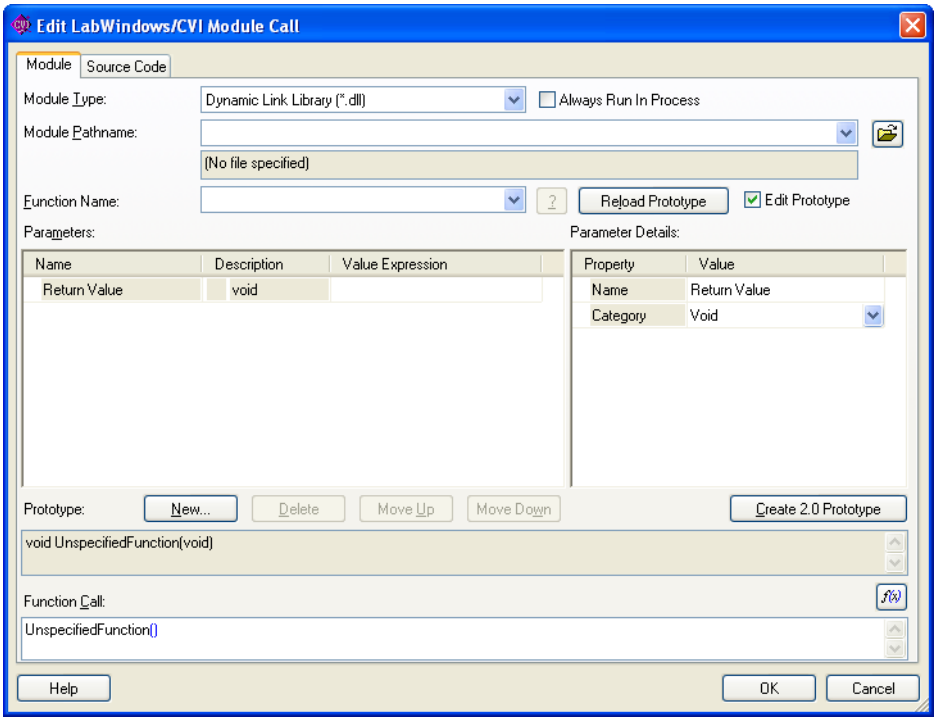


Figure 2-1. Edit LabWindows/CVI Module Call Dialog Box

The Edit LabWindows/CVI Module Call dialog box contains a Module and Source Code tab. The Module tab specifies the code module that the LabWindows/CVI Adapter executes for the step. The Source Code tab contains additional information that TestStand requires for creating and editing a code module in LabWindows/CVI.

Module Tab

Use the Module tab to specify the type of module, the module path, and the function name for the code module that the step executes. The LabWindows/CVI Adapter supports calling functions in C source files, object files, static library files, and dynamic link library (DLL) files.

Note National Instruments recommends using DLL files when you develop code modules using the LabWindows/CVI Adapter. The tutorials in this manual only demonstrate creating and debugging DLL code modules. Refer to Chapter 5, *Configuring the LabWindows/CVI Adapter*, for additional requirements for calling functions in C source files, object files, and static library files.

You can also use the Module tab in the Edit LabWindows/CVI Module Call dialog box to specify the function prototype, which includes the data type of each parameter and the values to pass for each parameter.

The Parameters Table control shows all of the available parameters for the function call and an entry for the return value. You can insert, remove, or rearrange the order of the parameters. The Parameters Table control contains the following columns:

- **Name**—Displays a symbolic name for the parameter.
- **Description**—Displays the short description of the parameter type using C syntax.
- **Value Expression**—Displays the argument expression to pass.

When you select a parameter in the Parameters Table control, the specific details about the parameter are displayed in the Parameter Details Table control. The information required for a parameter varies depending on whether the data type is a Numeric, String, Object, C Struct, or Array.

The Prototype control displays the function and the parameter information using C syntax. As an alternative to specifying the function name and the parameter values, you can use the Function Call control to directly edit the function name and all of the function arguments at once.

Source Code Tab

Use the Source Code tab to generate or edit the source code for the function and to resolve differences between the parameter list in the source code and the parameter information on the Module tab. You do not have to use the Source Code tab in order for TestStand to call the code module.

The Source Code tab contains pathname controls for specifying the source file that contains the function that the step calls and for specifying the project to use when editing the source code. If the code module is a DLL or static library, you must enter the name of the LabWindows/CVI project used to create the DLL or static library file. If the code module is an object file, you can also specify a project.

When you click Create Code or Edit Code, the LabWindows/CVI Adapter launches a copy of LabWindows/CVI and opens the source file. If you specify a project file in the Source Code tab, the LabWindows/CVI Adapter also opens the project in LabWindows/CVI. Clicking Create Code for a function that already exists in the file launches the Choose Code Template dialog box, in which you can specify to either replace the current function or add the new function above the current function.

Click **Verify Prototype** to check for conflicts between the parameter list in the source code and the parameter information on the Module tab.

Note Click the **Help** button located at the bottom of the Edit LabWindows/CVI Module Call dialog box to access the *TestStand Help*, which provides additional information about the dialog box.

Creating and Configuring a New Step Using the LabWindows/CVI Adapter

In this tutorial, you will learn how to insert a new step that uses the LabWindows/CVI Adapter and then configure that step to call a code module.

1. Launch the TestStand Sequence Editor.
2. Select **LabWindows/CVI Adapter** in the Adapter ring control.
3. Open a new Sequence File window, if one is not already open.
4. Select **File»Save As** and save the sequence file as `<TestStand>\Tutorial\CallCVICodeModule.seq`.
5. Right-click inside the new Sequence File window and insert a new `Pass/Fail Test` step.
6. By default, the new step is named Pass/Fail Test. After you insert the step, the name is highlighted. Type `CVI Pass/Fail Test` and press **<Enter>** to rename the step.

Tip To rename a step at a later time, select the step and press **<F2>**, or right-click the step name and select **Rename** from the context menu.

7. Right-click the `CVI Pass/Fail Test` step and select **Specify Module** from the context menu to launch the Edit LabWindows/CVI Module Call dialog box.
8. Select **Dynamic Link Library (*.dll)** in the Module Type ring control.
9. Click the **File Browse** button and select the following file:

   `<TestStand>\Tutorial\CallCVICodeModule.dll`.
10. On the Edit LabWindows/CVI Module Call dialog box, select the `PassFailTest` function in the Function Name ring control. This launches the Use Code Template dialog box.

Note When you select a function, the LabWindows/CVI Adapter attempts to read the function parameter information from the type library in the code module, if one exists. If the function parameter information is not defined, the adapter prompts you to use a code template to specify the function prototype.

11. Click **Yes** in the Use Code Template dialog box.

Note If the LabWindows/CVI Adapter is configured to use both new and legacy code templates, the adapter launches the Choose Code Template dialog box. Select **PassFail template for LabWindows/CVI**, then click **OK**.

Notice that the following function prototype is displayed in the Prototype control of the Edit LabWindows/CVI Module Call dialog box after you select the code template:

```
void PassFailTest(long *result, char
   ReportText[1024], short *errorOccurred, long
   *errorCode, char errorMsg[1024])
```

The Parameters Table control contains the default value expressions specified by the code template. When TestStand calls the code module, the LabWindows/CVI Adapter stores the returned values for the result and the error details in the specified properties of the step.

12. Click **OK** to save your settings and exit the Edit LabWindows/CVI Module Call dialog box.
13. Select **File»Save** to save the sequence file.
14. Select **Execute»Single Pass** to run the sequence file using the Single Pass Execution entry point.

    Because the LabWindows/CVI Adapter is configured to use an external instance of LabWindows/CVI to execute code modules, TestStand launches the LabWindows/CVI development environment to execute the function that the step calls.

    When the execution is complete, the resulting report shows that the step passed. The code module always returns `True` as its Pass/Fail output parameter.
15. Select **File»Unload All Modules** to instruct TestStand to unload the DLL that the step calls so that you can rebuild the DLL in the next chapter.

You have completed this tutorial. In the next chapter, you will learn how to create, edit, and debug code modules from TestStand.

3

Creating, Editing, and Debugging LabWindows/CVI Code Modules from TestStand

This chapter discusses how to use the LabWindows/CVI Adapter to create new code modules that you can call from TestStand, as well as how to edit and debug existing code modules.

Creating a New Code Module from TestStand

In this tutorial, you will learn how to create a new code module from TestStand.

1. Launch the TestStand Sequence Editor.
2. Open the following sequence file, which you created in the *Creating and Configuring a New Step Using the LabWindows/CVI Adapter* section of Chapter 2, *Calling LabWindows/CVI Code Modules from TestStand*:

   `<TestStand>\Tutorial\CallCVICodeModule.seq`.
3. Select **LabWindows/CVI Adapter** in the Adapter ring control.
4. Insert a Numeric Limit Test step after the `CVI Pass/Fail Test` step and rename it `CVI Numeric Limit Test`.
5. Right-click the `CVI Numeric Limit Test` step and select **Specify Module** from the context menu. This launches the Edit LabWindows/CVI Module Call dialog box.
6. Use the **Module** tab to complete the following steps:
   a. Select **Dynamic Link Library (*.dll)** from the Module Type ring control.
   b. For the **Module Pathname** control, click the **File Browse** button and select the following file:

      `<TestStand>\Tutorial\CallCVICodeModule.dll`.
   c. Type `NumericLimitTest` in the Function Name ring control.

7. Select the **Source Code** tab and type `CVINumericLimitTest.c` in the **Source File Containing Function** control.
8. For the **CVI Project File to Open** control, click the **File Browse** button and select the following file:

   `<TestStand>\Tutorial\CallCVICodeModule.prj`.
9. Click **Create Code** to create a code module.

   When you click Create Code, TestStand launches the Select a Pathname for the Source File dialog box. Browse to the `<TestStand>\Tutorial` subdirectory and click **OK**.

Note If TestStand launches the Choose Code Template dialog box, select the **NumericLimit template for LabWindows/CVI** and click **OK**. Do not select the Legacy NumericLimit template.

   TestStand creates a new code module based on the source code template for the TestStand Numeric Limit Test and opens that code module in LabWindows/CVI.

10. In LabWindows/CVI, uncomment the following code in the source file:

```
double testMeasurement = 5.0;
double lowLimit;
*measurement = testMeasurement;
```

11. Save and close the source file. Leave LabWindows/CVI open.
12. In the LabWindows/CVI project window, select **Build»Create Debuggable Dynamic Link Library** to rebuild the DLL.
13. Return to the sequence editor and select the **Module** tab on the Edit LabWindows/CVI Call dialog box.

    Notice that TestStand automatically updates the function prototype and parameter values according to the code template.
14. Click **OK** to save your settings and exit the Edit LabWindows/CVI Module Call dialog box.
15. Save the sequence file as `<TestStand>\Tutorial\CallCVICodeModule2.seq`.
16. Start a new execution of the sequence file using the Single Pass Execution entry point.

    When the execution is complete, the resulting report shows that the step passed with a numeric measurement of `5.0`.
17. Select **File»Unload All Modules** to unload the DLL.

Note For more information about step types and code templates, refer to Chapter 13, *Creating Custom Step Types*, of the *TestStand Reference Manual*. For more information about creating code modules from TestStand, refer to Chapter 5, *Configuring the LabWindows/CVI Adapter*, of this manual.

Editing an Existing Code Module from TestStand

In this tutorial, you will learn how to edit an existing code module from TestStand.

1. Open the following sequence file:

   `<TestStand>\Tutorial\CallCVICodeModule2.seq`.

2. Right-click the `CVI Numeric Limit Test` step and select **Edit Code**.

   LabWindows/CVI becomes the active application in which the `CVINumericLimitTest.c` source file is open.

3. Change the initial value in the declaration for the `testMeasurement` variable to `-5.0`.

4. Save and close the source file.

5. Rebuild the DLL.

6. In the TestStand Sequence Editor, start a new execution of the sequence file using the Single Pass Execution entry point.

   When the execution is complete, the resulting report shows that the step has failed. The code module now returns `-5` in the Measurement field.

Debugging a Code Module in TestStand

In this tutorial, you will learn how to debug a code module that you call from TestStand using the LabWindows/CVI Adapter.

1. Open the following sequence file:

   `<TestStand>\Tutorial\CallCVICodeModule.seq`.

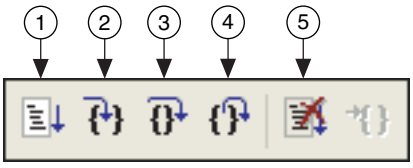
2. Place a breakpoint on the `CVI Pass/Fail Test` step by clicking to the left of the step.

   You will see the **Stop** icon to the left of the step when the breakpoint is set.

3. Select **Execute»Run MainSequence** to start an execution of the `MainSequence`.

   The execution starts and then pauses before executing the CVI Pass/Fail step.

4. When the execution pauses, click **Step Into** on the Sequence Editor toolbar. Figure 3-1 illustrates the Debug section of the Sequence Editor toolbar.



1	Run/Resume	4	Step Out
2	Step Into	5	Terminate Execution
3	Step Over		

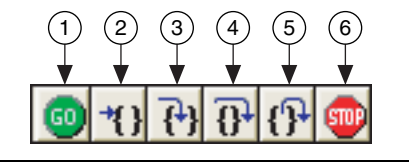
Figure 3-1. Debug Section of the Sequence Editor Toolbar

LabWindows/CVI becomes the active application, in which the LabWindows/CVI Pass-Fail Test code module is open and in a suspended state.

5. Click **Step Over** on the LabWindows/CVI toolbar to begin single-stepping through the code module.

6. When you have finished single-stepping through the code module, click **Finish Function** to return to TestStand. The execution then pauses at the next step in the sequence.

   Figure 3-2 shows the Debug section of the LabWindows/CVI toolbar.



1	Continue	4	Step Over
2	Goto Cursor	5	Finish Function
3	Step Into	6	Terminate Execution

Figure 3-2. Debug Section of the LabWindows/CVI Toolbar

7. Click **Resume** in TestStand to complete the execution.
8. Select **File»Unload All Modules** to unload the DLL.

You have completed this tutorial. In the next chapter, you will learn how TestStand passes different types of data to and from LabWindows/CVI code modules.

4

Using LabWindows/CVI Data Types with TestStand

This chapter describes how TestStand converts LabWindows/CVI data to and from its own data types.

Data Type Conversion

TestStand provides four basic built-in data types: number, string, Boolean, and object reference. TestStand also provides several standard named data types including Path and Error. You can create container data types that hold any number of other data types.

Note TestStand containers are analogous to C structures in LabWindows/CVI.

LabWindows/CVI has a greater variety of built-in data types than TestStand. For this reason, TestStand converts LabWindows/CVI data in certain ways when calling code modules. Table 4-1 describes how TestStand handles the various LabWindows/CVI data types.

Table 4-1. TestStand Equivalents for LabWindows/CVI Data Types

LabWindows/CVI C Data Type	TestStand Data Type
char, unsigned char, short, unsigned short, long, unsigned long, float, or double	Number TestStand stores all numeric C data types as double precision floating point numbers. TestStand does not set the format for a number property when assigning a value.
const char*, char[], const wchar_t*, const unsigned short*, wchar_t[], or unsigned short[]	Path, String, or Expression Refer to the *Calling Code Modules with String Parameters* section of this chapter for more information about using the String data type.
enum	Number

Table 4-1. TestStand Equivalents for LabWindows/CVI Data Types (Continued)

LabWindows/CVI C Data Type	TestStand Data Type
IDispatch *pDispatch, IUnknown *pUnknown, or CAObjHandle objHandle	Object reference Refer to the *Calling Code Modules with Object Parameters* section of this chapter for more information about using the Container data type in TestStand.
Array of *x*	Array of TestStand (*x*)
struct	Container Refer to the *Calling Code Modules with Struct Parameters* section of this chapter for more information about using the Container data type in TestStand.

Calling Code Modules with String Parameters

When you configure calls to code modules that have strings as parameters, you can specify whether to pass the string as a constant or as a buffer, as well as whether to pass the string as a C string or a unicode string.

If you pass the string as a constant, the LabWindows/CVI Adapter passes the address of the actual string directly to the function without copying it to a buffer. The code module must not change the contents of the string.

If you pass a string as a buffer, the LabWindows/CVI Adapter copies the contents of the string argument and a trailing zero element into a temporary buffer before calling the function. You specify the minimum size of the temporary buffer. If the string value is longer than the buffer size you specify, the LabWindows/CVI Adapter resizes the temporary buffer so that it is large enough to hold the contents of the string argument and the trailing zero element. After the function returns, the LabWindows/CVI Adapter copies the value that the function writes into the temporary buffer back to the string argument. The LabWindows/CVI Adapter only copies data from the beginning of the temporary buffer up to and including the first NULL character.

You can pass NULL to a string pointer parameter by passing an empty object reference or the constant `Nothing`.

Calling Code Modules with Object Parameters

You can configure calls to code modules that use an ActiveX Automation IDispatch Pointer (IDispatch *), ActiveX Automation IUnknown Pointer (IUnknown *), or a LabWindows/CVI ActiveX Automation Handle (CAObjHandle) as a parameter.

You can use these types to pass a reference to a built-in or custom TestStand object in a code module function. You can also use these types to pass the value of an object reference property to a code module function.

If you specify an object reference property as the value of an object parameter, TestStand passes the value of the property. Otherwise, TestStand passes a reference to the property object you specify.

The function that the step calls can invoke methods and access the properties on the object. You can pass the object parameter by value or by reference. If the function stores the value of the object for later use after the function returns, the function must properly add an additional reference to the ActiveX Automation IDispatch Pointer or ActiveX Automation IUnknown Pointer or duplicate the LabWindows/CVI ActiveX Automation Handle. If you pass the object by reference and the function alters the value of the reference, the function must release the original reference.

Calling Code Modules with Struct Parameters

When you configure calls to code modules that use structs as parameters, you specify that a TestStand data type maps to the entire C struct. TestStand can help you create a new custom data type that matches a C struct.

Use the Struct Passing tab of the Type Properties dialog box for a custom data type to specify how TestStand maps subproperties to members in a C struct. When you specify the data to pass for the struct parameter on the Edit LabWindows/CVI Call Module dialog box, you only need to specify an expression that evaluates to data with the data type.

Refer to the *TestStand Help* for more information about the Type Properties dialog box. Refer to Chapter 11, *Type Concepts*, of the *TestStand Reference Manual* for more information about where TestStand stores custom data types.

Creating TestStand Data Types from LabWindows/CVI Structs

In this tutorial, you will learn how to create a TestStand data type that is equivalent to a LabWindows/CVI struct and how to call a function in a DLL that has the struct as a parameter.

Building a New Custom Data Type

In this section, you will create a new container data type that contains both numeric and string subproperties.

1. Open the following sequence file:

   `<TestStand>\Tutorial\CallCVICodeModule2.seq`.
2. Select **Sequence File Types** in the View ring control in the Sequence File window.
3. Select the **Custom Data Types** tab in the Sequence File Types view.
4. Right-click inside the list view on the **Custom Data Types** tab and select **Insert Custom Data Type»Container** to insert a new data type.
5. Rename the new container data type `CVITutorialStruct`.
6. Click on the `CVITutorialStruct` node in the tree view of the **Custom Data Types** tab.
7. Right-click inside the list view on the **Custom Data Types** tab and select **Insert Field»Number** to insert a new field in the data type.
8. Rename the new field `Measurement`.
9. Right-click inside the list view and select **Insert Field»String** to insert a new field.
10. Rename the new field `Buffer`. You have completed the `CVITutorialStruct` container data type.
11. Leave the sequence file open, and continue to the next tutorial.

Specifying Structure Passing Settings

In this section, you will specify the structure passing properties for the CVITutorialStruct container data type.

1. Right-click the `CVITutorialStruct` node in the tree view and select **Properties** to launch the Type Properties dialog box.

Note The name of the Type Properties dialog box is specific to the name of the property you have selected.

2. Select the **C/C++ DLL Struct Passing** tab in the Type Properties dialog box.
3. Enable the **Allow Objects of This Type to be Passed as Structs** option on the **C/C++ DLL Struct Passing** tab.

   The Property control lists the two fields in the CVITutorialStruct container data type. Notice that the type of the Measurement property defaults to 64-bit Real Number.
4. Select the `Buffer` property.
5. Make sure that the **String Type** control setting is set to **C String Buffer**. This setting instructs TestStand to allow the C function to alter the value of the structure field.
6. Select **OK** to close the Type Properties dialog box.
7. Leave the sequence file open, and continue to the next tutorial.

Calling a Function With a Struct Parameter

In this tutorial, you will use the CVITutorialStruct container data type as a parameter to a function that a step calls.

1. Select **MainSequence** from the View ring control in the Sequence File window.
2. Select the **Locals** tab.
3. Right-click inside the list view of the Locals tab and select **Insert Local»Types»CVITutorialStruct** to insert an instance of the container data type.
4. Rename the new variable `CVIStruct`.
5. Select the **Main** tab and then select **LabWindows/CVI Adapter** in the Adapter ring control.
6. Insert a new Action step into the Main step group of `MainSequence` after the `CVI Numeric Limit Test` step.

7. Rename the step `Pass Struct Test`.
8. Right-click the new step and select **Specify Module** from the context menu to launch the Edit LabWindows/CVI Module Call dialog box.
9. Select the **Module** tab and click the **File Browse** button. Select the following file:

   `<TestStand>\Tutorial\CallCVICodeModule.dll`.
10. Type `PassStructTest` in the **Function Name** control.
11. Click **New** to insert a new parameter and enter the following information in the Parameter Details Table control:
    a. In the **Name** field, rename the parameter `cviStruct`.
    b. In the **Category** field, select `C Struct`.
    c. In the **Type** field, select `CVITutorialStruct`.
12. Enter `Locals.CVIStruct` in the Value Expression field for the parameter in the **Parameters Table** control.
13. Select the **Source Code** tab.
14. In the **Source File Containing Function** control, type `CVIStructPassingTest.c`.
15. Click the **File Browse** button and select the following file:

    `<TestStand>\Tutorial\CallCVICodeModule.prj`.
16. Enable the **Use Prototype From Module Tab** option.
17. Click **Create Code** to create a code module.
18. In the **Select a Pathname for the Source File** control, browse to the `<TestStand>\Tutorial` subdirectory and click **OK**.

    TestStand creates a new source file with an empty function.
19. In LabWindows/CVI, add the following type definition before the first function:

```
struct CVITutorialStructType {
    double measurement;
    char buffer[256];
};
```

    Add the following code to the `PassStructText` function:

```
if (cviStruct)
{
    cviStruct->measurement = 5.0;
    strcpy(cviStruct->buffer, "Average Voltage");
}
```

20. Save and close the source file.
21. In the LabWindows/CVI project window, select **Build»Create Debuggable Dynamic Link Library** to rebuild the DLL.
22. Return to the TestStand Sequence Editor and click **OK** on the Edit LabWindows/CVI Module Call dialog box to save the settings and close the dialog box.
23. Place a breakpoint on the new `Pass Struct Test` step.
24. Select **Run»MainSequence** to start a new execution of `MainSequence`.

    As you single-step through the sequence, review the values in the Locals.CVIStruct variable before and after executing the new step.
25. Select **File»Unload All Modules** to unload the DLL.

You have completed this tutorial. In the next chapter, you will learn how to configure the LabWindows/CVI Adapter.

5

Configuring the LabWindows/CVI Adapter

In this chapter, you will learn how to configure the various settings of the LabWindows/CVI Adapter.

To access the LabWindows/CVI Adapter Configuration dialog box, launch the general Adapter Configuration dialog box by selecting **Configure» Adapters**. Then, enable the **LabWindows/CVI** radio button in the Adapter column and click **Configure**.

Figure 5-1 shows the LabWindows/CVI Adapter Configuration dialog box.

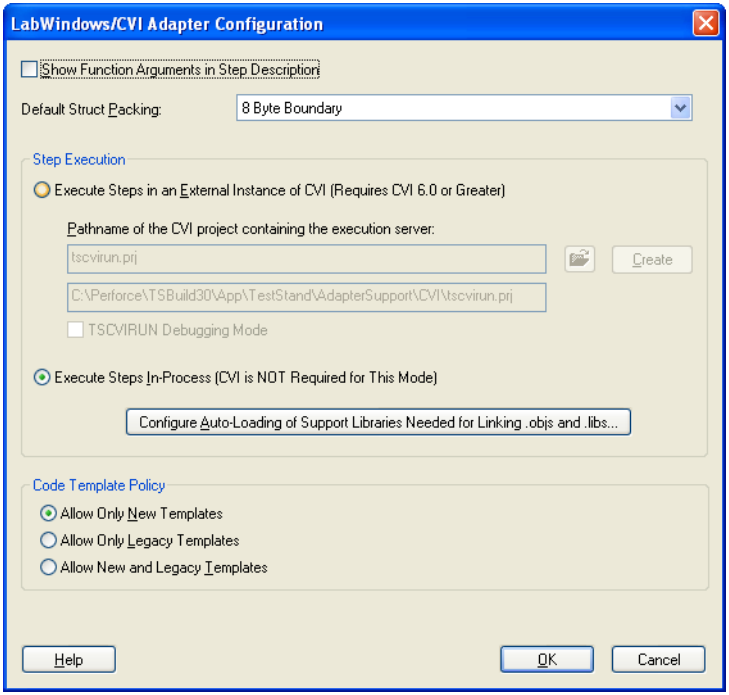


Figure 5-1. LabWindows/CVI Adapter Configuration Dialog Box

Showing Function Arguments in Step Descriptions

Use the Show Function Arguments in Step Description control to specify whether the description for a step in the sequence editor and operator interfaces include the parameters with the function. If you disable this option, the description only displays the function and module name.

Setting the Default Structure Packing Size

The LabWindows/CVI Adapter can call functions in code modules that have structure parameters. Use the Default Struct Packing control on the LabWindows/CVI Adapter Configuration dialog box to specify the default setting for how the LabWindows/CVI Adapter packs structure parameters it passes. The following options are available: 1-, 2-, 4-, 8-, and 16-byte boundaries.

The compatibility mode of the LabWindows/CVI development environment that you use to create your DLLs determines your choice for the structure packing value. For LabWindows/CVI, the default structure packing can be either 1- or 8-byte. For example, in Visual C++ compatibility mode, LabWindows/CVI has a default of 8-byte packing. Refer to the *Calling a Function With a Struct Parameter* section of Chapter 4, *Using LabWindows/CVI Data Types with TestStand*, for more information about calling code modules with struct parameters.

Selecting Where Steps Execute

The LabWindows/CVI Adapter can run code modules out-of-process using an external instance of the LabWindows/CVI development environment or run code modules in the same process as the sequence editor or operator interface you are running, without using the LabWindows/CVI development environment.

Use the Step Execution section of the LabWindows/CVI Adapter Configuration dialog box to select where steps execute.

Executing Code Modules in an External Instance of LabWindows/CVI

To execute tests in an external instance of LabWindows/CVI, the LabWindows/CVI Adapter launches a copy of the LabWindows/CVI development environment and loads an execution server project. You can specify the execution server project to load in the LabWindows/CVI

Adapter Configuration dialog box. The default project is `<TestStand>\AdapterSupport\CVI\tscvirun.prj`.

When a TestStand step calls a function in an object, static library, or DLL file, the execution server project automatically loads the code module and executes the function in an external instance of LabWindows/CVI. If you want a TestStand step to call a function in a C source file, you must include the C source file in the execution server project before you run the project. You must also include any support libraries other than LabWindows/CVI libraries that the object, static library, or C source file requires.

Debugging Code Modules

You can debug C source and DLL code modules when the LabWindows/CVI Adapter executes tests in an external instance of LabWindows/CVI. To debug DLL code modules, you must create a debuggable DLL in LabWindows/CVI.

Note LabWindows/CVI honors all breakpoints that you set in the source files for the DLL project.

When you execute tests in an external instance of LabWindows/CVI, you do not need to launch the sequence editor or operator interface application from LabWindows/CVI to debug DLL code modules that you call with the LabWindows/CVI Adapter.

If you click Step Into in TestStand while the execution is suspended on a step that calls into the DLL code module, LabWindows/CVI suspends on the first statement in the called function.

Executing Code Modules In-Process

When executing code modules in the same process as the sequence editor or operator interface, the LabWindows/CVI Adapter loads and runs code modules directly without using the LabWindows/CVI development environment.

Object and Library Code Modules

When the LabWindows/CVI Adapter loads an object or static library file, the LabWindows/CVI Run-Time Engine resolves all external references in the file. When running code modules in-process, the adapter must load the support libraries that the object file or static library file depends on before loading the code module file.

To configure a list of support libraries for the LabWindows/CVI Adapter to load, manually copy the support libraries to the `<TestStand>\Adapter Support\CVI\AutoLoadLibs` directory. You can also click the **Configure Auto-Loading of Support Libraries Needed for Linking .objs and .libs** button on the LabWindows/CVI Adapter Configuration dialog box to launch the Auto-Load Library Configuration dialog box, which is shown in Figure 5-2.

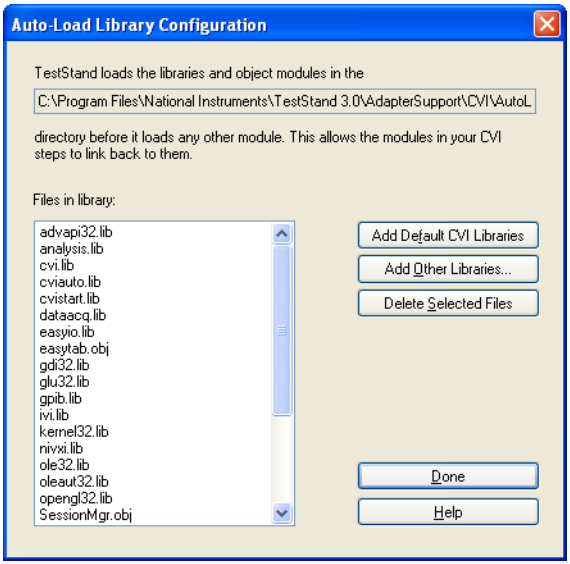


Figure 5-2. Auto-Load Library Configuration Dialog Box

You can configure the support libraries by performing one of the following actions in the Auto-Load Library Configuration dialog box:

- Click **Add Default CVI Libraries** to search for an installation of the LabWindows/CVI development environment and copy the LabWindows/CVI static library files to the auto-load library directory.
- Click **Add Other Libraries** to search for files to copy to the auto-load library directory.
- Click **Delete Selected Files** to remove the selected files from the auto-load library directory.

Source Code Modules

When TestStand executes code modules in-process, the LabWindows/CVI Adapter cannot directly execute code modules that exist in C source files. Instead, the adapter attempts to find an object file that has the same name. If the adapter finds the object file, it executes the code in the object file. If the adapter cannot find the object file, it prompts you to create the object file in an external instance of LabWindows/CVI. If you decline to create the object file, the adapter reports a run-time error.

Debugging DLL Code Modules

In order to debug code modules that are in-process, the code modules must exist in DLLs that were enabled for debugging in LabWindows/CVI at the time they were built. To debug a DLL in-process, launch the sequence editor or operator interface from LabWindows/CVI. Then select **Run» Select External Process** in the LabWindows/CVI project window to identify the executable you want to launch. Select **Run»Debug Project** to launch the executable and begin debugging.

If you click Step Into in TestStand while the execution is currently suspended on a step that calls into a LabWindows/CVI DLL that you are debugging, LabWindows/CVI suspends on the first statement in the DLL function.

Refer to your LabWindows/CVI documentation for more information about debugging DLLs.

Loading Subordinate DLLs

TestStand directly loads and runs the DLLs that you specify in the Specify Module dialog box for the LabWindows/CVI Adapter. Since your code modules most likely call subsidiary DLLs, such as instrument drivers, you must ensure that the operating system can find and load any DLLs. The operating system searches for the DLLs using the following search directory precedence:

1. The directory in which the application resides.
2. The current working directory.
3. In Windows 98, the `Windows\System` directory. In Windows 2000/NT/XP, the `Windows\System32` and `Windows\System` directories.
4. The `Windows` directory.
5. The directories listed in the PATH environment variable.

Note When the LabWindows/CVI Adapter attempts to directly load a DLL, the adapter temporarily sets the current working directory to the directory where the DLL code module resides.

Note Refer to Chapter 14, *Deploying TestStand Systems*, of the *TestStand Reference Manual* for more information about deploying your code modules and subsidiary DLLs for use with TestStand.

Per-Step Configuration of the LabWindows/CVI Adapter

You can direct TestStand to always run steps that use the LabWindows/CVI Adapter in-process. Make this selection by enabling the **Always Run In Process** option on the **Module** tab of the Edit LabWindows/CVI Module Call dialog box. This setting overrides the global setting in the LabWindows/CVI Adapter Configuration dialog box. Use this option when you create tools and step types for use with the LabWindows/CVI Adapter that you do not want to be affected by the global setting for the adapter.

Code Template Policy

The Code Template Policy section of the LabWindows/CVI Adapter Configuration dialog box allows you to specify whether TestStand allows you to create new test code modules using old, or *legacy*, code module templates. These legacy code module templates are files that are callable from previous versions of TestStand and the LabWindows/CVI Test Executive Toolkit. Refer to Appendix C, *Calling Legacy Code Modules*, for more information about legacy code module templates.

If you have configured the LabWindows/CVI Adapter using the Allow Only New Templates option and then create a new code module from the Edit LabWindows/CVI Module Call dialog box, TestStand either immediately creates a new code module based on the template for the specified step type or, if the step type has multiple templates available, launches the Choose Code Template dialog box. Use this dialog box, which is illustrated in Figure 5-3, to select the template to use for the new code module.

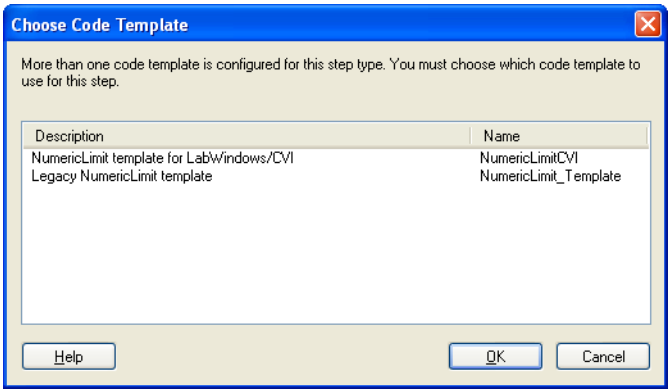


Figure 5-3. Choose Code Template Dialog Box

If you have configured the LabWindows/CVI Adapter using the Allow Only Legacy Templates option, TestStand immediately creates a new code module based on the legacy code module template for the specified step type.

If you have configured the LabWindows/CVI Adapter using the Allow New and Legacy Templates option, TestStand launches the Choose Code Template dialog box, in which you can select the template to use for the new code module.

6

Creating Custom User Interfaces in LabWindows/CVI

This chapter discusses the tools that TestStand provides for creating custom operator interfaces and for creating user interfaces for other components, such as custom step types.

Tip National Instruments recommends that you read Chapter 9, *Creating Custom Operator Interfaces*, of the *TestStand Reference Manual*, to obtain a general understanding of the TestStand User Interface (UI) Controls before proceeding with this chapter.

Note The TestStand UI Controls are not supported in Windows 98.

TestStand User Interface Controls

This section describes how to use the TestStand UI Controls in the LabWindows/CVI development environment to develop a custom operator interface application.

Creating and Configuring ActiveX Controls

To add a TestStand UI Control to a panel in the User Interface Editor, select **Create»ActiveX** and select a UI control whose name begins with `TestStand UI`. You can configure a UI control using the standard LabWindows/CVI Edit Control dialog box, which you launch by double-clicking the control. To open property pages that are supported by a UI control, right-click the control and select **Properties** from the context menu.

Programming with ActiveX Controls

In order to access the methods, properties, and events specific to an ActiveX control, you need to use the ActiveX driver for the control. The TestStand UI Controls driver and additional support instrument drivers are located in the `<TestStand>\API\CVI` directory.

Add the following function panel files to the LabWindows/CVI project for your TestStand application:

- **TestStand UI Controls** (`tsui.fp`)—Contains functions for dynamically creating controls, calling methods and accessing properties on controls, and handling events from the controls.
- **TestStand UI Support Library** (`tsuisupp.fp`)—Contains functions for various collections that the TestStand UI Controls driver uses.
- **TestStand Utility Functions** (`tsutil.fp`)—Contains utility functions for managing menu items that correspond to TestStand commands, localizing strings on your user interface, making dialog boxes associated with LabWindows/CVI code modules modal in respect to TestStand applications, and checking whether an execution that calls a code module has stopped.
- **TestStand API** (`tsapicvi.fp`)—Provides low-level access to TestStand objects.

For each interface that the ActiveX control supports, the driver contains a function that you can use to programmatically create an instance of the ActiveX control. The ActiveX driver also includes functions that you can use to register callback functions for receiving events defined by the control.

When you store ActiveX controls in `.uir` files, you do not need to use the creation functions included in the driver. The control is created when you load the panel from the file using the LoadPanel function. You identify the control in subsequent calls to User Interface Library functions with the constant name that you assigned to the control in the User Interface Editor.

When you use other functions in the driver, you must identify the control with a unique object handle which LabWindows/CVI then associates with the control. You obtain this handle when you call `GetObjHandleFromActiveXCtrl` using the constant name for the control. This handle is cached in the control, and you do not need to discard the handle explicitly.

LabWindows/CVI requires that a thread be initialized as apartment threaded before you can use ActiveX controls in a program. If you do not initialize the thread before creating an ActiveX control or before loading a panel containing an ActiveX control from a `.uir` file, LabWindows/CVI automatically initializes your thread to apartment threaded. If you use CA_InitActiveXThreadStyleForCurrentThread to initialize the thread

yourself, you must use COINIT_APARTMENTTHREADED as the threading model.

For general information about programming the TestStand API from LabWindows/CVI, refer to Appendix B, *Using the TestStand ActiveX APIs in LabWindows/CVI*.

Creating Custom Operator Interfaces

Operator interfaces that use the TestStand UI Controls typically perform the following basic operations:

- Configure connections, commands, and other control settings
- Register to handle events sent by the controls
- Start TestStand
- Wait in a main event loop until you close the application
- Shut down TestStand

Operator interfaces may also have a menu bar containing items that invoke TestStand commands, as well as non-TestStand items.

For additional information about creating a TestStand Operator Interface using the TestStand UI Controls in LabWindows/CVI, refer to the example operator interfaces included in TestStand. Begin with the simple operator interface example, `<TestStand>\OperatorInterfaces\NI\Simple\CVI\TestExec.prj`. For a more advanced example that includes menus and localization options, refer to the full featured example, `<TestStand>\OperatorInterfaces\NI\Full-Featured\CVI\TestExec.prj`.

To customize these example operator interfaces, copy the `OperatorInterfaces` directory and its contents from the `NI` subdirectory to the `<TestStand>\OperatorInterfaces\User` subdirectory before beginning your customizations. This ensures that newer installations of TestStand will not overwrite your custom operator interfaces.

Note Example operator interfaces that use the TestStand API instead of the TestStand UI Controls are available in the `<TestStand>\OperatorInterfaces\NI\TestStand 2.0.1 Operator Interfaces (Old)` directory. While these examples contain a large amount of complex source code, they provide less functionality than the simpler examples that use the ActiveX controls. Therefore, National Instruments does not recommend these older examples as a basis for new development.

Configuring the TestStand UI Controls

Refer to Table 6-1 for information about which functions in the example operator interface files demonstrate configuring connections, commands, and other settings for the TestStand UI Controls.

Table 6-1. Functions in Examples for Configuring the TestStand UI Controls

Source File	Functions
<TestStand>\OperatorInterfaces\NI\Simple\CVI\TestExec.c	SetupActiveXControls
<TestStand>\OperatorInterfaces\NI\Full-Featured\CVI\TestExec.c	GetActiveXControlHandles RegisterActiveXEventCallbacks ConnectTestStandControls ConnectStatusBarPanes RebuildMenuBar

Handling Events

TestStand UI Controls send events to notify your application of user input and application events, such as the completion of an execution. To handle an event in LabWindows/CVI, you register a callback function, which is automatically called when the control sends the event. Use the Event Callback Registration functions in the TestStand UI Controls driver to perform event registration.

For example, the following statement registers a callback function for the OnExitApplication event sent from the Application Manager control:

```
TSUI__ApplicationMgrEventsRegOnExitApplication (
    gAppMgrHandle, AppMgr_OnExitApp, NULL, 1, NULL);
```

The callback function can contain the following code, which verifies whether the TestStand Engine is in a state where it can shut down:

```
HRESULT CVICALLBACK AppMgr_OnExitApp(CAObjHandle
caServerObjHandle, void *caCallbackData)
{
    VBOOL canExitNow;

    if (!TSUI_ApplicationMgrShutdown(gAppMgrHandle,
        &errorInfo, &canExitNow) && (canExitNow))
        QuitUserInterface(0);
    return S_OK;
}
```

Starting and Shutting Down TestStand

When you initialize your operator interface application, use the `TSUI_ApplicationMgrStart` driver function to invoke the Start method on the Application Manager control, which starts the TestStand Engine and logs in a user.

LabWindows/CVI applications typically wait for user input by calling the RunUserInterface() function after loading and displaying the main user interface panel. The RunUserInterface() function handles all events, such as menu selections, control value changes, and ActiveX control events.

Typically, you stop an operator interface application by clicking the **Close** box or by executing the **Exit** command through either a TestStand menu or a Button control. For user interface events that request the operator interface to close, the operator interface must call the TSUI_ApplicationMgrShutdown function to unload sequence files, log out, and trigger an OnApplicationCanExit event. If the function determines that the TestStand Engine can shutdown, the canExitNow output parameter returns `True`. The operator interface application should then call the QuitUserInterface() function, which causes the preceding RunUserInterface() call to return. After the application exits the function call to RunUserInterface(), the operator interface application must call TSUI_ApplicationMgrShutdown a second time to complete the cleanup process and shutdown the TestStand Engine.

Menu Bars

The TestStand Utility Functions provide a set of functions for creating and handling menu items that execute the following TestStand UI Control functions:

- `TS_InsertCommandsInMenu`
- `TS_RemoveMenuCommands`
- `TS_CleanupMenu`

Use the `TS_InsertCommandsInMenu` function to create new menu items that execute commands you specify without requiring any additional code. To create menu items, you specify an array of command types. Each command type specifies a menu item or group of menu items to insert. You must also specify a handle to the Application Manager control, ExecutionView Manager control, or SequenceFileView Manager control to which the new menu items apply. TestStand uses a manager control to determine whether the menu item is visible or dimmed. TestStand installs

a callback for each menu item that automatically invokes the associated command when the user selects that menu item.

Call the `TS_InsertCommandsInMenu` function when your application rebuilds the menu bar in a MenuDimmerCallback in order to populate the menu bar with commands that apply to the current state of the application. Before you call this function, you can call `TS_RemoveMenuCommands` to remove any menu items you previously inserted.

Refer to the `RebuildMenuBar` function in the `<TestStand>\OperatorInterfaces\NI\Full-Featured\CVI\TestExec.c` source file for an example of rebuilding the menu bar.

Localization

The TestStand UI Controls and TestStand Utility Functions driver provide tools that localize your operator interfaces based on the TestStand language setting. Use the following functions to localize your operator interface:

- `TS_LoadPanelResourceStrings`
- `TS_LoadMenuBarResourceStrings`
- `TSUI_ApplicationMgrLocalizeAllControls`

Refer to the `<TestStand>\OperatorInterfaces\NI\Full-Featured\CVI\TestExec.c` source file for an example of localizing operator interface panels.

Other User Interface Utilities

This section outlines some of the functions available in the TestStand Utility Functions driver.

Making a Dialog Code Module Modal to TestStand

Code modules that TestStand calls may launch dialog boxes that are modal to TestStand application windows such as the TestStand Sequence Editor or Operator Interface.

The TestStand Utility Functions driver provides the following functions that make a dialog box modal to TestStand application windows:

- `TS_StartModalDialogEx`
- `TS_EndModalDialog`
- `TS_EndModalDialogAndDiscard`

For a demonstration of how to use these functions, refer to the `<TestStand>\Components\NI\StepTypes\MsgBox\msgbox.c` source file.

Checking For Stopped Execution

Code modules that TestStand calls may launch dialog boxes or perform other time-consuming operations. Therefore, it can be useful to have those code modules periodically check whether their parent execution has been terminated or aborted. This allows the code modules to stop gracefully and allow their parent execution to terminate or abort.

The TestStand Utility Functions driver provides the following functions that enable code modules called by TestStand to verify whether the execution that called it has been stopped:

- `TS_CancelDialogIfExecutionStops`
- `TS_CancelDialogIfExternalExecutionStops`

You can also refer to the dialog box code in the following example source files for a demonstration of how to use these functions:

- `<TestStand>\Examples\Demo\C\computer.c`
- `<TestStand>\Examples\Demo\C\auto.c`

A

Adding Type Libraries to LabWindows/CVI DLLs

If a LabWindows/CVI DLL file contains a type library, the LabWindows/CVI Adapter automatically populates the Function Name control on the Edit LabWindows/CVI Module Call dialog box with all of the function names in the type library. When you select a function in the DLL, the adapter queries the type library for the parameter list information and displays it in the Parameters Table control of the Edit LabWindows/CVI Module Call dialog box. If a DLL created with LabWindows/CVI does not have type library information, you must enter parameter information manually.

LabWindows/CVI can use the information specified in a function panel file to generate type library information to include in a DLL. Complete the following steps to instruct LabWindows/CVI to generate a type library resource from a function panel and add the type library resource to a DLL:

1. Open a new function panel file and create a function panel for each exported function that you want to include in the type library.
2. Add the function panel file to your LabWindows/CVI project.
3. In the LabWindows/CVI project window, select **Build»Target Settings** to launch the Target Settings dialog box.
4. In the Target Settings dialog box, click **Type Library** to launch the Type Library dialog box.
5. In the Type Library dialog box, enable the **Add Type Library Resource to DLL** option and enter the path to the file in the **Function Panel File** control.

   You can also choose to include links in the type library resource to a Windows help file, or generate a Windows help file from the function panel file by selecting **Options»Generate Windows Help** in the Function Tree Editor window.
6. In the Project window, select **Build»Create Debuggable Dynamic Link Library** to build the DLL.

Note If an exported function in a DLL uses the `__cdecl` calling convention instead of `__stdcall`, and you specify to add a type library resource to the DLL, LabWindows/CVI displays a warning when you build the DLL. This warning applies to any DLLs that you intend to use with Microsoft Visual Basic. Because the LabWindows/CVI Adapter can call functions with either calling convention, you can ignore the warning.

LabWindows/CVI imposes certain requirements on the declaration of the DLL API in a type library. Use the following guidelines to ensure that TestStand can use your DLL:

- Use typedefs for structure parameters and union parameters.
- Do not use enum parameters.
- Do not use structures that require forward references or that contain pointers.
- Do not use pointer types except when passing parameters by reference.

Refer to your LabWindows/CVI documentation for more information about adding type libraries to DLLs.

B

Using the TestStand ActiveX APIs in LabWindows/CVI

In some cases you may need to program the TestStand API or TestStand UI Controls from your LabWindows/CVI code modules and user interface source code. This chapter contains information about programming with the TestStand Engine and TestStand UI Controls APIs from LabWindows/CVI.

The *ActiveX Library* topic of the *LabWindows/CVI Online Help* contains fundamental information about ActiveX concepts and how to access ActiveX servers from LabWindows/CVI. National Instruments recommends that you become familiar with this material before proceeding with this appendix.

Using ActiveX Drivers in LabWindows/CVI

LabWindows/CVI creates and accesses ActiveX objects using functions in a LabWindows/CVI-generated driver. This driver uses function panels to define C functions for all the methods and properties available for each object. For servers that define events, the driver contains functions for registering callbacks for events.

The driver functions you use to invoke methods and properties have a special naming convention in which function names start with a prefix, such as `TS_`. Methods are followed by the class name and the method name. Properties are followed by either Get or Set and the property name. In some cases, the class, method, and property names are abbreviated to keep the function name within the constraints of the `.fp` file format.

The LabWindows/CVI ActiveX Automation Library uses the CAObjHandle data type for handles to ActiveX objects. The TestStand ActiveX drivers also follow this convention. Therefore, you can use the CAObjHandle data type for all handles to TestStand objects. However, one drawback of using the same data type for all TestStand objects is that the compiler cannot flag calls to methods in which you pass a handle for the wrong kind of object.

Objects can support more than one interface. For example, a SequenceContext object has a SequenceContext interface and a PropertyObject interface. When using handles in LabWindows/CVI to invoke methods or access properties of an object, you do not have to convert a specific reference for one interface to a specific reference for another interface. The ActiveX driver always queries the handle for the proper interface before invoking the method or accessing the property.

If you receive an object handle as the result of calling a method or getting the handle from a property, you must release the handle when you are finished with it. For more information about the CA_DiscardObjHandle function, refer to the *Adding and Releasing References* section of this appendix.

Some TestStand ActiveX API methods have output parameters that return strings. You must free these strings when you are done with them using the `CA_FreeMemory` function in the LabWindows/CVI ActiveX Automation Library.

Invoking Methods

TestStand objects have methods that you invoke to perform an operation or function on them. In LabWindows/CVI, you invoke methods on TestStand objects using the functions defined in the ActiveX driver for those objects.

The following function illustrates how to access the number of steps in a sequence:

```
int GetNumSteps(CAObjHandle sequence)
{
    int error = 0;
    ErrMsg errMsg = "";
    ERRORINFO errorInfo;
    CAObjHandle engine = 0;
    long *numSteps = 0;

    tsErrChk(TS_SequenceGetNumSteps (sequence,
        &errorInfo, TS_StepGroup_Main, &numSteps);

Error:

    return error;
}
```

The errorInfo variable is a structure that the LabWindows/CVI ActiveX Automation Library defines to hold information about errors that can occur in the operation of the function. The tsErrChk macro determines whether the function's return value or the errorInfo variable indicates that an error occurred and continues execution at the Error label when `True`.

Note The functions, constants, and enumerations in the `tsapicvi.fp` driver begin with the unique prefix `TS_`. This prefix is not included in the function, constant, and enumeration names in the *TestStand Help*.

Accessing Built-In Properties

TestStand defines a number of built-in properties that are always present for objects such as steps and sequences. Nearly every kind of object in TestStand has built-in properties, which are static with respect to the TestStand API. This means that the TestStand API has knowledge about each of these properties, which it uses to allow you to access these properties in the programming language you specify. Examples of built-in properties are the Name property of the Sequence object and the Sequence property of the SequenceContext object.

In LabWindows/CVI, you access built-in properties using a property function in the ActiveX driver. The following code obtains the value of the Name property from a Sequence object:

```
int GetSequenceName(CAObjHandle sequence)
{
    int error = 0;
    ErrMsg errMsg = "";
    ERRORINFO errorInfo;
    char *sequenceName = 0;

    tsErrChk(TS_SequenceGetName (sequence, &errorInfo,
        &sequenceName));

Error:
    // Free Resources
    if (sequenceName)

        CA_FreeMemory(sequenceName);

    return error;
}
```

The following function obtains a reference to a step from a Sequence object:

```
int GetStepInSequence(CAObjHandle sequence)
{
    int error = 0;
    ErrMsg errMsg = "";
    ERRORINFO errorInfo;
    CAObjHandle step = 0;

    tsErrChk(TS_SequenceGetStepByName (sequence,
        &errorInfo, &step));

Error:
    // Free Resources
    if (step)

        CA_DiscardObjHandle(step);

    return error;
}
```

Accessing Dynamic Properties

TestStand allows you to define your own custom step properties, sequence local variables, sequence file global variables, and station global variables. Because the TestStand API has no knowledge of the variables and custom step properties that you define, these variables and properties are dynamic with respect to the TestStand API. The TestStand API provides the PropertyObject class so that you can access dynamic properties and variables, while using lookup strings to identify specific properties by name.

The following example illustrates setting a local variable by calling a method of the PropertyObject class on a handle to a Sequence Context object:

```
int SetLocalVariable(CAObjHandle seqContextCVI)
{
    int error = 0;
    ErrMsg errMsg = "";
    ERRORINFO errorInfo;
    VBOOL propertyExists;

    // Set local variable NumericValue to a random number

    tsErrChk(TS_PropertyExists(seqContextCVI,
        &errorInfo, "Locals.NumericValue", 0,
        &propertyExists));

    if (propertyExists)
        tsErrChk(TS_PropertySetValNumber(seqContextCVI,
        &errorInfo, "Locals.NumericValue", 0, rand()));

Error:

    return error;
}
```

Adding and Releasing References

LabWindows/CVI automatically maintains an object reference for each handle you obtain for an object. If you assign the handle to another variable, LabWindows/CVI does not add a reference to the object. Use the `CA_DuplicateObjHandle` function in the LabWindows/CVI ActiveX Automation Library to obtain a new handle to an existing object, thus adding a reference to the object.

LabWindows/CVI automatically releases the object reference for each handle you obtain when you call the CA_DiscardObjHandle function from the LabWindows/CVI ActiveX Automation Library. The following example illustrates obtaining a handle to the TestStand Engine from the SequenceContext object, calling a method on the engine to acquire a version string, and releasing the handle to the engine and the string:

```
int GetEngineVersion(CAObjHandle seqContextCVI)
{
    int error = 0;
    ErrMsg errMsg = "";
    ERRORINFO errorInfo;
```

```
    CAObjHandle engine = 0;
    char *versionString = 0;

    tsErrChk(TS_SeqContextGetEngine(seqContextCVI,
        &errorInfo, &engine));
    tsErrChk(TS_EngineGetVersionString (engine,
        &errorInfo, &versionString));

Error:
    // Free Resources
    if (engine)
        CA_DiscardObjHandle(engine);
    if (versionString)
        CA_FreeMemory(versionString);
    return error;
}
```

Note If you fail to release the handle, LabWindows/CVI will not release the object. Repeatedly opening references to objects without closing them can cause your system to run out of memory.

Using TestStand API Constants and Enumerations

Some TestStand API methods require string and numeric constant input arguments. The acceptable values of these arguments are organized into groups that correspond to different properties and methods. For example, the SetValNumber method on the PropertyObject class has an options input argument that accepts many different numeric constants.

The header file for the ActiveX driver defines all constants and enumerations that the methods and properties require. The constant and enumeration names start with a prefix, such as `TS_`, followed by the constant or enumeration name.

Note The functions, constants, and enumerations in the `tsapicvi.fp` driver begin with the unique prefix `TS_`. This prefix is not included in the function, constant, and enumeration names in the *TestStand Help*.

For example, the ActiveX driver defines the RunModes constant as follows:

```
#define TS_RunMode_Normal    "Normal"
#define TS_RunMode_Skip      "Skip"
#define TS_RunMode_ForceFail "Fail"
#define TS_RunMode_ForcePass "Pass"
```

The ActiveX driver defines the StepGroups enumeration as follows:

```
enum TSEnum_StepGroups
{
    TS_StepGroup_Setup = 0,
    TS_StepGroup_Main = 1,
    TS_StepGroup_Cleanup = 2,
    TS_StepGroupsForceSizeToFourBytes = 0xFFFFFFFF
};
```

For parameters of functions of type enumeration, the LabWindows/CVI function panel displays the list of enumerations in a ring control.

For parameters of functions that specify a numeric constant, use the bitwise-OR operator to specify multiple options. For example, the following code only sets a local variable if the variable does not already exist:

```
int options = PropOption_DoNothingIfExists ||
    PropOption_InsertIfMissing;

tsErrChk (TS_PropertySetValNumber(seqContext, NULL,
    "Locals.NumericValue", options, rand()));
```

C

Calling Legacy Code Modules

Prior to TestStand 3.0, you had to use the DLL Flexible Prototype Adapter to call functions in LabWindows/CVI DLLs that did not use a specific prototype. Using TestStand 3.0 and later, you can call functions with a wide variety of parameter data types, including code modules with legacy function prototypes.

Prototypes of Legacy Code Modules

TestStand supports two legacy prototypes—standard and extended. In earlier versions of TestStand, National Instruments recommended that you use the standard prototype. The extended prototype provides backward compatibility with the LabWindows/CVI Test Executive Toolkit version 2.0 and earlier and offers an additional string parameter.

The following is the standard prototype:

```
void TX_TEST StandardFunc(tTestData *data, tTestError
    *error)
```

The following is the extended prototype:

```
int TX_TEST ExtendedFunc(const char *params, tTestData
    *data, tTestError *error)
```

While you would usually create new code modules from the Edit LabWindows/CVI Module Call dialog box for steps that use the LabWindows/CVI Adapter, TestStand can also create legacy-style code modules. Chapter 5, *Configuring the LabWindows/CVI Adapter*, details how to configure the LabWindows/CVI Adapter for creating new legacy-style code modules.

The legacy prototypes contain two structure parameters, **tTestData** and **tTestError**, which the LabWindows/CVI Adapter uses to pass values into and out of the code module.

tTestData Structure

The **tTestData** structure contains input and output data. Table C-1 lists the fields in the **tTestData** structure.

Table C-1. tTestData Structure Member Fields

Field Name	Data Type	In/ Out	Description
result	int	Out	Set by test function to indicate whether the test passed. Valid values are PASS or FAIL. The LabWindows/CVI Adapter copies this value into the Step.Result.PassFail property if the property exists.
measurement	double	Out	Numeric measurement that the test function returns. The LabWindows/CVI Adapter copies this value into the Step.Result.Numeric property if the property exists.
inBuffer	char *	In	For passing a string parameter to a test function. The LabWindows/CVI Adapter copies the Step.InBuf property value into this field if the property exists.
outBuffer	char *	Out	Output message to display in the report. The LabWindows/CVI Adapter copies the message value into the Step.Result.ReportText property if the property exists.
modPath	char * const	In	Directory path of the module that contains the test function. The LabWindows/CVI Adapter sets this value before executing the code module.
modFile	char * const	In	Filename of the module that contains the test function. The LabWindows/CVI Adapter sets this value before executing the code module.
hook	void *	In	Reserved (no longer used).
hookSize	int	In	Reserved (no longer used).
mallocFuncPtr	tMallocPtr const	In	Contains a function pointer to malloc, which a code module must use to allocate memory for any buffer that it assigns to the inBuffer, outBuffer, and errorMessage fields.

Table C-1. tTestData Structure Member Fields (Continued)

Field Name	Data Type	In/ Out	Description
freeFuncPtr	tFreeptr	In	Contains a function pointer to free, which a code module must use to free any buffers that the inBuffer, outBuffer, and errorMessage fields point to.
seqContextDisp	struct IDispatch *	In	Dispatch pointer to the sequence context. This value is NULL if you choose not to pass the sequence context.
seqContextCVI	CAObjHandle	In	LabWindows/CVI ActiveX Automation handle for the sequence context. This value is 0 if you choose not to pass the sequence context.
stringMeasurement	char *	Out	String value that the test function returns. The LabWindows/CVI Adapter copies this string into the Step.Result.String property if the property exists.
replaceStringParameter	tReplaceString Ptr const	In	Contains a function pointer to ReplaceString, which a code module can use to reassign a value to the inBuffer, outBuffer, and errorMessage fields. The ReplaceString prototype is as follows: int ReplaceString(char **destString, char *srcString); The function return value is non-zero if successful.
structVersion	int	In	Structure version number. A test module can use this value to detect new versions of the structure.

Note Use the sequence context to access all the objects, variables, and properties in the execution. Refer to the *TestStand Help* for more information about using the sequence context from a LabWindows/CVI code module.

tTestError Structure

The **tTestError** structure only contains output error information. Table C-2 lists the fields in the **tTestError** structure.

Table C-2. tTestError Structure Member Fields

Field Name	Data Type	In/ Out	Description
errorFlag	Boolean (int)	Out	The test function must set this value to `True` if an error occurs. The LabWindows/CVI Adapter copies this output value into the Step.Result.Error.Occurred property if the property exists.
errorLocation	tErrLoc (int)	Out	Reserved (no longer used).
errorCode	int	Out	The test function can set this value to a non-zero value if an error occurs.
errorMessage	char *	Out	The test function can set this field to a descriptive string if an error occurs.

Automatically Updated Step Properties

Before calling a code module, the LabWindows/CVI Adapter assigns values from TestStand to input fields of the **tTestData** structure. After calling the code module, the LabWindows/CVI Adapter copies the values of the output fields of the structures to properties of the step. The LabWindows/CVI Adapter copies a value into a property when the following conditions are true:

- The property exists.
- The code module does not change the value of the property directly through the TestStand API.

In some cases, the LabWindows/CVI Adapter translates the value of a structure field to a different value in the corresponding property.

Table C-3 lists all the properties that the LabWindows/CVI Adapter updates and the value translation, if any, that the adapter makes.

Table C-3. Step Properties Updated by LabWindows/CVI Adapter

Structure Member	Valid Values that Tests Can Return	Step.Result Property	Step Property Value
result	PASS or FAIL	PassFail	`True`/`False`
outBuffer	string value	ReportText	string value
measurement	floating-point value	Numeric	numeric value
stringMeasurement	string value	String	string value
errorFlag	True or False	Error.Occurred	`True`/`False`
errorCode	integer value	Error.Code	numeric value
errorMessage	string value	Error.Msg	string value

Example Code Module

When you create a legacy code module for the LabWindows/CVI Adapter, you must add the `stdtst.h` header file located in the `<TestStand>\Bin` directory to your source file. The `stdtst.h` file includes the type definitions for the **tTestData** and **tTestError** structures. The following is an example code module that uses the LabWindows/CVI standard prototype:

```
// Simple test example
#include "stdtst.h"
void TX_TEST __declspec(dllexport) FunctionName
(tTestData *testData, tTestError *testError)
{

    int error = 0;
    double measurement = 5.0;
    char *lastUserName = NULL;

    testData->measurement = measurement;
    if ((error = TS_PropertyGetValString(
        testData->seqContextCVI, NULL,
        "StationGlobals.TS.LastUserName",
        0, lastUserName)) < 0)
        goto Error;

Error:
    // FREE RESOURCES
    CA_FreeMemory(lastUserName);
```

```
    // Set the error flag to cause a run-time error
    if (error < 0)
    {
        testError->errorFlag = TRUE;
        testError->errorCode = error;
        testData->replaceStringFuncPtr(&testError->
        errorMessage, "ErrorText");
    }
}
```

D

Technical Support and Professional Services

Visit the following sections of the National Instruments Web site at ni.com for technical support and professional services:

- **Support**—Online technical support resources include the following:
  - **Self-Help Resources**—For immediate answers and solutions, visit our extensive library of technical support resources available in English, Japanese, and Spanish at ni.com/support. These resources are available for most products at no cost to registered users and include software drivers and updates, a KnowledgeBase, product manuals, step-by-step troubleshooting wizards, conformity documentation, example code, tutorials and application notes, instrument drivers, discussion forums, a measurement glossary, and so on.
  - **Assisted Support Options**—Contact NI engineers and other measurement and automation professionals by visiting ni.com/support. Our online system helps you define your question and connects you to the experts by phone, discussion forum, or email.
- **Training**—Visit ni.com/training for self-paced tutorials, videos, and interactive CDs. You also can register for instructor-led, hands-on courses at locations around the world.
- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, NI Alliance Program members can help. To learn more, call your local NI office or visit ni.com/alliance.

If you searched ni.com and could not find the answers you need, contact your local office or NI corporate headquarters. Phone numbers for our worldwide offices are listed at the front of this manual. You also can visit the Worldwide Offices section of ni.com/niglobal to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

Glossary

A

abort	To stop an execution without running any of the steps in the Cleanup step groups in the sequences on the call stack run. When you abort an execution, no report generation occurs.
ActiveX (Microsoft ActiveX)	Set of Microsoft technologies for reusable software components. Formerly called *OLE*.
ActiveX control	A reusable software component that adds functionality to any compatible ActiveX control container.
ActiveX server	Any executable code that makes itself available to other applications according to the ActiveX standard. ActiveX implies a client/server relationship in which the client requests objects from the server and asks the server to perform actions on the objects.
adapter	If an adapter is specific to an application development environment (ADE), the adapter knows how to open the ADE, how to create source code for a new code module in the ADE, and how to display the source for an existing code module in the ADE. Some adapters support stepping into the source code of the ADE while executing the step from the TestStand Sequence Editor.
Application Development Environment (ADE)	A programming environment such as LabVIEW, LabWindows/CVI, or Microsoft Visual Basic, in which you can create test modules and operator interfaces.
Application Programming Interface (API)	A set of classes, methods, and properties that you use to control a specific service, such as the TestStand Engine.

B

breakpoint	An interruption in the execution of a program.
button	A dialog box item that, when selected, executes a command associated with the dialog box.

C

class	Defines a list of methods and properties that you can use with respect to the objects that you create as instances of that class. A class is like a data type definition except that it applies to objects rather than variables.
code module	A program module, such as a Windows Dynamic Link Library (`.dll`) or LabVIEW code module (`.vi`), that contains one or more functions that perform a specific test or other action.
code template	A source file that contains skeleton code. The skeleton code serves as a starting point for the development of code modules for steps that use a particular step type.
context menu	Access context menus by right-clicking on an object. Menu options in a context menu pertain specifically to the object you have selected.
control	An input and output device in a panel or window, in which you can enter data or make a setting.

D

dialog box	A user interface in which you specify additional information for the completion of a command.
DLL	Dynamic Link Library.

E

engine	A module or set of modules that provide an API for creating, editing, executing, and debugging sequences. A sequence editor or operator interface uses the services of a test executive engine.
entry point	A sequence in the process model file that TestStand displays as a menu item, such as Test UUTs, Single Pass, and Report Options.
Execution entry point	A sequence in a process model that runs tests against a UUT. Execution entry points call the MainSequence callback in the client sequence file. The default process model contains two Execution entry points: Test UUTs and Single Pass. By default, Execution entry points are visible in the Execute menu. Execution entry points are only visible in the menu when the active window contains a sequence file that has a MainSequence callback.

expression	A formula that calculates a new value from the values of multiple variable or properties. In expressions, you can access all variables and properties in the sequence context that is active when TestStand evaluates the expression.

G

global variable	TestStand defines two types of global variables: sequence file globals and station globals. Sequence file globals are accessible by any sequence or step in the sequence file. Station globals are accessible by any sequence file loaded on the station. The values of station global variables are persistent across different executions and even across different invocations of TestStand.

L

LabVIEW	Laboratory Virtual Instrument Engineering Workbench. A program development application based on the G programming language and used commonly for test and measurement purposes.
LabWindows/CVI Adapter	*See* adapter.
local variable	A property of a sequence that holds a value or additional subproperties. Only a step within the sequence can directly access the property value.
lookup string	A string that defines a complete path from the object on which you call the method to the specific property you want to access.

O

object	A service that an ActiveX server makes available to clients.
operator interface	A program that provides a graphical user interface (GUI) for executing sequences on a production station.

P

parameters	A value passed to a program or subroutine.
process model	A sequence file you designate that performs a standard series of operations before and after a test executive executes the sequence that performs the tests. Common operations include identifying the UUT, notifying the operator of pass/fail status, generating a test report, and logging results.
property	A container of information, which stores and maintains a setting or attribute of an object. A property can be of type number, string, Boolean, container, ActiveX reference, a user-defined data type, or an array of these types. A property can contain a single value, an array of values of the same type, or no value at all. A property can also contain any number of subproperties. Only a container property has the ability to contain any number of subproperties. Each property has a name and a comment.

R

run-time error	An error condition that forces an execution to terminate. When the error occurs while running a sequence, TestStand jumps to the Cleanup step group, and the error propagates to any calling sequence up through to the top-level sequence.

S

sequence	Located within a sequence file, a sequence contains a series of steps that you specify to execute in a particular order. When and if a step is executed can depend on the results of previous steps.
sequence context	A TestStand object that contains references to all global variables and all local variables and step properties in active sequences. The contents of the sequence context changes depending on the currently executing sequence and step.
sequence editor	A program that provides a graphical user interface (GUI) for creating, editing, and debugging sequences.
sequence file	A file that contains the definition of one or more sequences.
Sequence File window	A separate window within the sequence editor in which a sequence file is displayed.

source code template	A set of source files that contain skeleton code, which serves as a starting point for the development of code modules for steps. TestStand uses the source code template when you click Create Code on the Source Code tab in the Specify Module dialog box for a step.
standard named data type	A data type that TestStand defines and names. You can add subproperties to the standard data types, but you cannot delete any of their built-in subproperties. The standard named data types are `Path`, `Error`, and `CommonResults`.
station global variables	Variables that are persistent across different executions and even across different invocations of the sequence editor or operator interfaces. The TestStand Engine maintains the value of station global variables in a file on the run-time computer.
step	An element that you can insert into a sequence that performs an action, such as calling a step module to perform a specific test. Typically, a sequence contains a series of steps that define your test and execution flow.
step group	A set of steps in a sequence. A sequence contains the following groups of steps: Setup, Main, and Cleanup. When TestStand executes a sequence, the steps in the Setup group execute first, the steps in the Main group execute next, and the steps in the Cleanup group execute last.
step property	A property of a step.

T

template	*See* code template.
test executive engine	*See* engine.

V

variables	A property that you can freely create in a certain context. You can have variables that are global to a sequence file or local to a particular sequence. You can also have station global variables.

W

window	A working area that supports specific tasks related to developing and executing programs.

Index

A

C

D

E

H

I

K

L

M

N

O

P

S

T

U

W