NI TestStand™

Using LabWindows™/CVI™ with TestStand

April 2007
373201B-01

NATIONAL INSTRUMENTS™

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 43 662 457990-0, Belgium 32 (0) 2 757 0020, Brazil 55 11 3262 3599, Canada 800 433 3488, China 86 21 5050 9800, Czech Republic 420 224 235 774, Denmark 45 45 76 26 00, Finland 385 (0) 9 725 72511, France 33 (0) 1 48 14 24 24, Germany 49 89 7413130, India 91 80 41190000, Israel 972 3 6393737, Italy 39 02 413091, Japan 81 3 5472 2970, Korea 82 02 3451 3400, Lebanon 961 (0) 1 33 28 28, Malaysia 1800 887710, Mexico 01 800 010 0793, Netherlands 31 (0) 348 433 466, New Zealand 0800 553 322, Norway 47 (0) 66 90 76 60, Poland 48 22 3390150, Portugal 351 210 311 210, Russia 7 495 783 6851, Singapore 1800 226 5886, Slovenia 386 3 425 42 00, South Africa 27 0 11 805 8197, Spain 34 91 640 0085, Sweden 46 (0) 8 587 895 00, Switzerland 41 56 2005151, Taiwan 886 02 2377 2222, Thailand 662 278 6777, Turkey 90 212 279 3031, United Kingdom 44 (0) 1635 523545

For further support information, refer to the *Technical Support and Professional Services* appendix. To comment on National Instruments documentation, refer to the National Instruments Web site at ni.com/info and enter the info code feedback.

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, AO <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 tip, which alerts you to advisory information.
	This icon denotes a note, which alerts you to important 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. Italic text 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.

Contents

Chapter 5
Configuring the LabWindows/CVI Adapter

Chapter 6
Creating Custom User Interfaces in LabWindows/CVI

Appendix A
Using the TestStand ActiveX APIs in LabWindows/CVI

Appendix B
Adding Type Libraries to LabWindows/CVI DLLs

Appendix C
Calling Legacy Code Modules

1

Introduction

This chapter discusses how NI TestStand and NI 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. TestStand 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 user 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.

User Interfaces

You can use LabWindows/CVI to build custom user interfaces for your test systems and for creating custom sequence editors. Typically, these custom user interfaces are 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. You can also create user interfaces using the TestStand UI Controls and the TestStand API. Refer to Chapter 9, *Creating Custom User Interfaces*, of the *NI TestStand Reference Manual*, for general information about creating custom user 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 *NI 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. You can 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

2

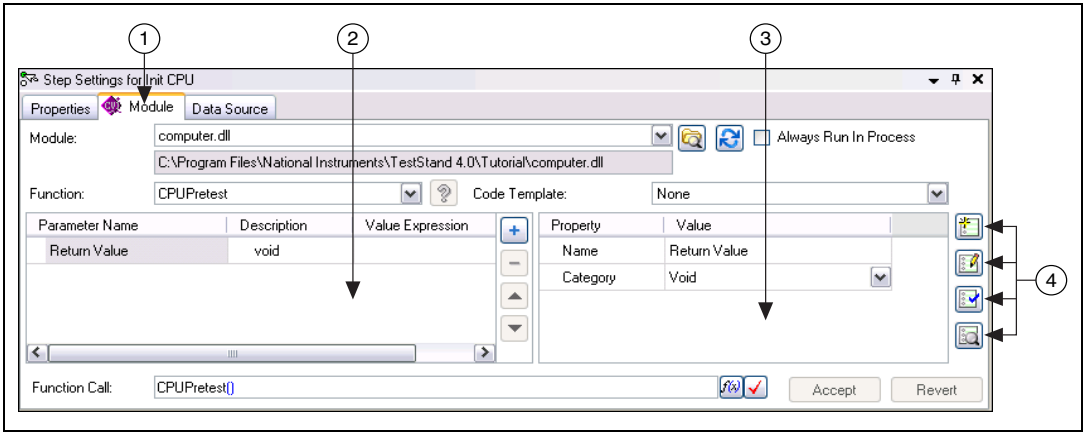
Calling LabWindows/CVI Code Modules from TestStand

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 code templates. Refer to Chapter 5, *Configuring the LabWindows/CVI Adapter*, for more information about configuring these settings for the adapter.

Introduction to the LabWindows/CVI Module Tab

Use the LabWindows/CVI Module tab in the TestStand Sequence Editor to configure calls to LabWindows/CVI code modules. To view the LabWindows/CVI Module tab, which is shown in Figure 2-1, select **Specify Module** from the context menu of any step that uses the LabWindows/CVI Adapter.



1 LabWindows/CVI Module Tab	3 Parameter Details Table Control
2 Parameters Table Control	4 Source Code Buttons

Figure 2-1. LabWindows/CVI Module Tab

LabWindows/CVI Module Tab

Use the LabWindows/CVI 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. The Module tab also includes Source Code buttons for creating and editing a code module in LabWindows/CVI.

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 also use the LabWindows/CVI Module tab to specify the function prototype, which includes the data type of each parameter and the values to pass for each parameter.

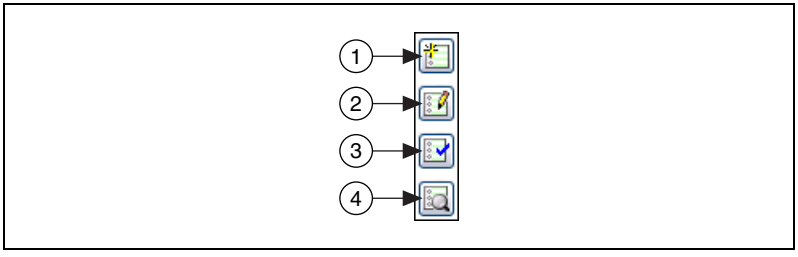
The Parameters Table control shows all of the available parameters for the function call and an entry for the return value. You can insert, remove, or rearrange the order of the parameters. The Parameters Table control contains the following columns:

- **Parameter 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. 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 Buttons

Use the Source Code buttons, shown in Figure 2-2, 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 LabWindows/CVI Module tab. You do not have to use the Source Code buttons in order for TestStand to call the code module.



1	Create Code	3	Verify Prototype
2	Edit Code	4	Source Code Files

Figure 2-2. LabWindows/CVI Module Tab Source Code Buttons

The Source Code Files button opens the CVI Source Code Files dialog box, in which you can specify the source file that contains the function that the step calls and to specify 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 optionally specify a project.

When you click the **Create Code** or **Edit Code** buttons, the LabWindows/CVI Adapter launches a copy of LabWindows/CVI and opens the source file. If you specify a project file using the Source Code Files button, the LabWindows/CVI Adapter opens the project in LabWindows/CVI when you click the Create Code button. If you click the Create Code button for a function that already exists in the file, the function you specified in the Code Template ring control is used and LabWindows/CVI displays the Generate Code dialog box, in which you can specify to either replace the current function or add the new function above or below the current function.

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

Note Click the **Help** or **Help Topic** button located on the Help toolbar to access the *NI TestStand Help*, which provides additional information about the LabWindows/CVI Module tab.

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 Insertion Palette.
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. Insert a Pass/Fail step in the Main step group of the Sequence File window. Rename the new step `CVI Pass/Fail Test`.
6. On the LabWindows/CVI Module tab of the Step Settings pane, click the **File Browse** button and select the following file:

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

7. On the LabWindows/CVI Module tab, select the `PassFailTest` function in the Function ring control.

Note When you select a function, the LabWindows/CVI Adapter attempts to read the export information that LabWindows/CVI includes in the DLL, or the function parameter information from the type library in the code module, if one exists. If the function parameter information is not defined, you can either select a code template from the Code Template ring control to specify the function prototype or specify the function prototype by adding parameters to the Parameters Table control.

Note If the LabWindows/CVI Adapter is configured to use both new and legacy code templates, nothing is selected in the Code Template ring control.

8. Select **PassFail template for LabWindows/CVI** in the Code Template ring control.

   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.

9. Select **File»Save** to save the sequence file.
10. 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.

11. 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 `<TestStand>\Tutorial\CallCVICodeModule.seq`., 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*.
3. Select **LabWindows/CVI Adapter** in the Insertion Palette.
4. Insert a Numeric Limit Test step after the `CVI Pass/Fail Test` step and rename it `CVI Numeric Limit Test`.
5. Select the `CVI Numeric Limit Test` step and click the LabWindows/CVI Module tab of the Step Settings pane.
6. Use the LabWindows/CVI **Module** tab to complete the following steps:
   a. For the **Module** control, click the **File Browse** button and select the following file:

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

   b. Type `NumericLimitTest` in the Function ring control.
   c. Select **NumericLimit template for LabWindows/CVI** in the Code Template ring control.

7. Click the **Source Code Files** button to launch the CVI Source Code Files dialog box. Complete the following steps:
   a. Type `CVINumericLimitTest.c` in the **Source File Containing Function** control.
   b. For the **CVI Project File to Open** control, click the **File Browse** button and select the following file:

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

   c. Click **Close**.

8. Click **Create Code** to create a code module.

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

   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.

Note The TestStand Numeric Limit Test step type requires code modules to store a measurement value in the Step.Result.Numeric property, and the step type performs a comparison operation to determine whether the step passes or fails. Code modules can update step properties by either passing step properties as parameters to and from the code module or by using the TestStand API in the code module. If you use a default code template from NI to create a code module, the parameters needed to access the step properties are created for you.

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

```
double testMeasurement = 10.0;
double lowLimit;
*measurement = testMeasurement;
```

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

    Notice that TestStand automatically updates the function prototype and parameter values according to the code template.

13. Save the sequence file as `<TestStand>\Tutorial\CallCVICodeModule2.seq`.

14. 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 `10.0`.

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

Note If LabWindows/CVI does not launch, the LabWindows/CVI Adapter is not configured to execute steps in an external instance. To configure the LabWindows/CVI Adapter, select **Configure»Adapters**. Then select **LabWindows/CVI** and click **Configure**. In the Step Execution section, select **Execute Steps in an External Instance of CVI**. Click **OK** to close the LabWindows/CVI Adapter Configuration dialog box. Click **OK** when the adapter displays a warning confirming that all modules will be unloaded. Begin step 4 again.

For more information about configuring the LabWindows/CVI Adapter refer to Chapter 5, *Configuring the LabWindows/CVI Adapter*.

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

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 container data types 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 object reference 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.

Note The LabWindows/CVI Adapter supports return values of type void and numeric, which includes 32-bit doubles and 8-, 16-, and 32-bit integers.

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 data type 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 Module tab of the Step Settings pane, you only need to specify an expression that evaluates to data with the data type.

Refer to the NI TestStand Help for more information about the Type Properties dialog box. Refer to Chapter 11, *Type Concepts*, of the *NI 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. In the Sequence File window, right-click and select **View»Types** from the context menu. Make sure that the `CallCVICodeModule2.seq` sequence file is selected in the View Types For pane.
3. Select the **Custom Data Types** node in the Types view.
4. Right-click the **Custom Data Types** node 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 under the **Custom Data Types** node.
7. Click the plus sign to expand the `CVITutorialStruct` node.
8. Right-click inside the tree view under the `CVITutorialStruct` node and select **Insert Field»Number** to insert a new field in the data type.
9. Rename the new field `Measurement`.
10. Right-click inside the tree view and select **Insert Field»String** to insert another new field in the CVITutorialStruct container data type.
11. Rename the new field `Buffer`. You have completed the `CVITutorialStruct` container data type.
12. 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 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 Struct Passing** tab.

   The Property ring control lists the two fields in the CVITutorialStruct container data type. Notice that the Numeric Type control for the Measurement property defaults to 64-bit Real Number (double).
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. Click the **CallCVIModule2.seq** tab in the Sequence File window.
2. Select the **Variables** tab at the bottom of the window.
3. Right-click the **Locals** node in the tree view and select **Insert Local»Type»CVITutorialStruct** to insert an instance of the container data type.
4. Rename the new variable `CVIStruct`.
5. Select the **Steps: MainSequence** tab and then select **LabWindows/CVI Adapter** in the Insertion Palette.
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. Click the LabWindows/CVI Module tab on the Step Settings pane.

9. Click the **File Browse** button next to the Module control. Select the following file:

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

10. Type `PassStructTest` in the **Function** control.

11. Click the **Add Parameter** button to insert a new parameter and enter the following information in the Parameter Details Table control:
    a. In the **Parameter 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. Click the **Source Code Files** button to launch the CVI Source Code Files dialog box. Complete the following steps:
    a. In the **Source File Containing Function** control, type `CVIStructPassingTest.c`.
    b. Click the **File Browse** button next to the CVI Project File to Open option and select the following file:

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

    c. Click **Close**.
14. Click the **Create Code** button to create a code module.
15. In the Select a Source File dialog box, browse to the `<TestStand>\Tutorial` subdirectory and click **OK**.

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

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

    Add the following code to the `PassStructTest` function:

```
if (cviStruct)
{
    cviStruct->measurement = 10.0;
    strcpy(cviStruct->buffer, "Average Voltage");
}
```

17. Save and close the source file.

18. In the LabWindows/CVI project window, select **Build»Create Debuggable Dynamic Link Library** to rebuild the DLL.
19. Return to the TestStand Sequence Editor.
20. Place a breakpoint on the new `Pass Struct Test` step.
21. Select **Execute»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.
22. 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**. Select **LabWindows/CVI** in the Adapter column and then click **Configure**.

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

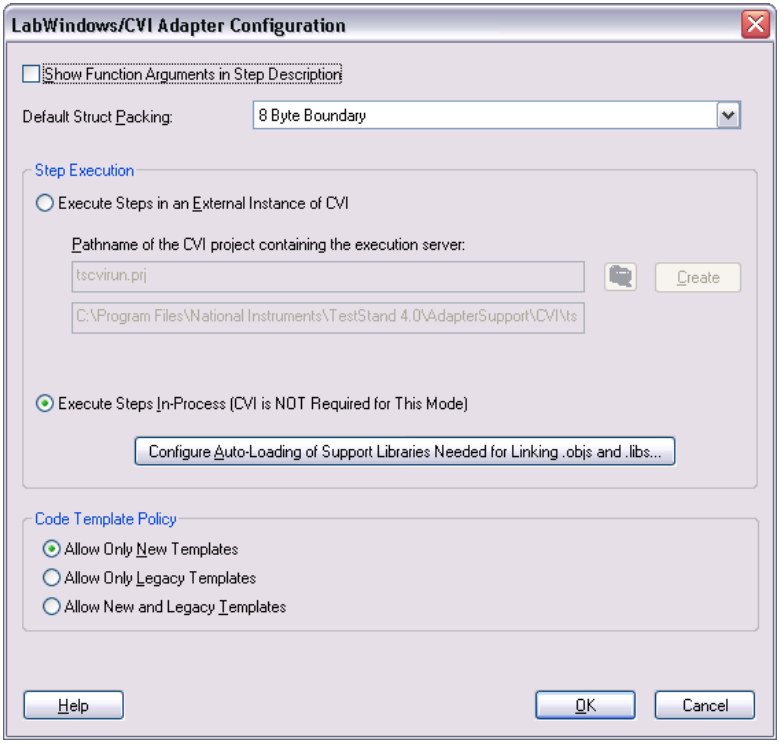


Figure 5-1. LabWindows/CVI Adapter Configuration Dialog Box

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 user 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 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 user 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. 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 user 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 user 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.

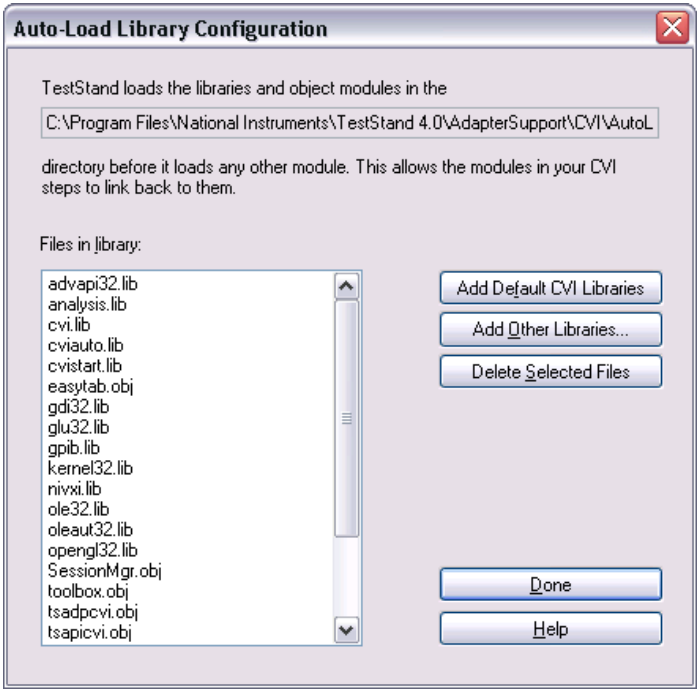


Figure 5-2. Auto-Load Library Configuration Dialog Box

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, you must launch the sequence editor or user interface from LabWindows/CVI. Select **Run» Specify 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 on the LabWindows/CVI Module tab 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 LabWindows/CVI Adapter first attempts to load subordinate DLLs using the following alternate search directory precedence, which includes the directory of the DLL.

1. The directory containing the DLL that the adapter is calling directly.
2. The current working directory of the application. (Windows 2000 and Windows XP SP1 and earlier only)
3. The `Windows\System32` and `Windows\System` directories.
4. The `Windows` directory.

5. The current working directory of the application. (Windows XP SP2 and later only)
6. The directories listed in the PATH environment variable.

If the LabWindows/CVI Adapter fails to load a DLL, the adapter temporarily sets the current working directory to be the directory of the DLL and attempts to load subordinate DLLs using the following search directory precedence:

1. The directory containing the application that loaded the adapter.
2. The current working directory of the application. (Windows 2000 and Windows XP SP1 and earlier only)
3. The `Windows\System32` and `Windows\System` directories.
4. The `Windows` directory.
5. The current working directory of the application. (Windows XP SP2 and later only)
6. The directories listed in the PATH environment variable

Note National Instruments does not recommend placing subordinate DLLs in the same directory containing the application that loaded the adapter, and may not support loading DLLs from this location in future versions.

Note Refer to Chapter 14, *Deploying TestStand Systems*, of the *NI 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 LabWindows/CVI Module tab. 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 you can call from previous versions of TestStand. 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 LabWindows/CVI Module tab, TestStand either immediately creates a new code module based on the code template for the specified step type or, if the step type has multiple code templates available, launches the Choose Code Template dialog box. Use the Choose Code Template dialog box, which is illustrated in Figure 5-3, to select the code template to use for the new code module.

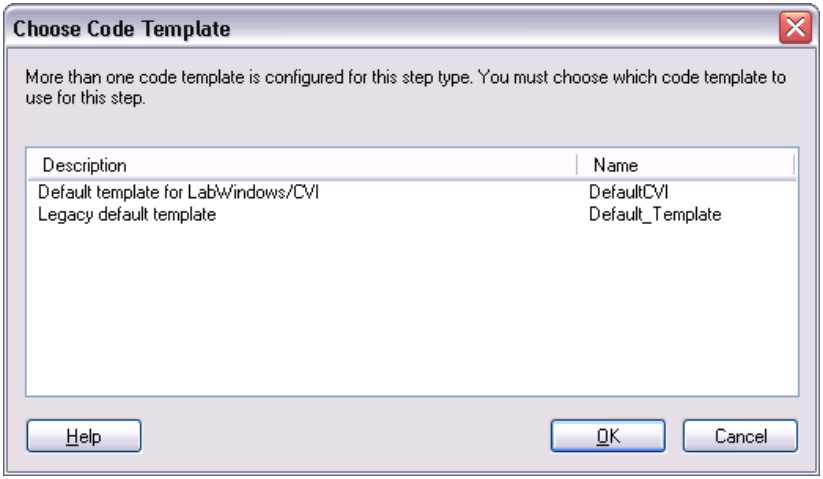


Figure 5-3. Choose Code Template Dialog Box

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 user 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 User Interfaces*, of the *NI TestStand Reference Manual*, to obtain a general understanding of the TestStand User Interface (UI) Controls before proceeding with this chapter.

TestStand User Interface Controls

This section describes how to use the TestStand UI Controls in the LabWindows/CVI development environment to develop a custom user interface application, including custom sequence editors.

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 A, *Using the TestStand ActiveX APIs in LabWindows/CVI*.

Creating Custom User Interfaces

User 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

User 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 User Interface using the TestStand UI Controls in LabWindows/CVI, refer to the example user interfaces included in TestStand. Begin with the simple user interface example, `<TestStand>\UserInterfaces\NI\Simple\CVI\TestExec.prj`. For a more advanced sequence editor example that includes menus and localization options, refer to the full-featured example, `<TestStand>\UserInterfaces\NI\Full-Featured\CVI\TestExec.prj`.

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

Note TestStand no longer includes example user interfaces that use the TestStand API instead of the TestStand UI Controls. These examples contained a large amount of complex source code and they provide less functionality than the simpler examples that use the TestStand UI Controls. Therefore, National Instruments recommends that you use the new examples that use the TestStand UI Controls as a basis for new development.

Configuring the TestStand UI Controls

Refer to Table 6-1 for information about which functions in the example user 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>\UserInterfaces\NI\Simple\CVI\TestExec.c	SetupActiveXControls
<TestStand>\UserInterfaces\NI\Full-Featured\CVI\TestExec.c	GetActiveXControlHandles RegisterActiveXEventCallbacks ConnectTestStandControls ConnectStatusBarPanes RebuildMenuBar

Enabling Sequence Editing

The TestStand UI Controls support both operate mode and sequence editing mode. You can instruct the Application Manager control to allow the user to create and edit sequence files by setting the Application.IsEditor property to `True`.You can also set the property by specifying the `/editor` command-line flag.

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 user 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 a user 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 user interface to close, the user 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 user 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 user 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 the following set of functions for creating and handling TestStand-specific menu items without requiring any additional code:

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

Use the `TS_InsertCommandsInMenu` function to create new menu items that execute commands you specify. To create menu items, you specify an array of command types and the menu bar and menu IDs determine where

to insert the commands. 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>\UserInterfaces\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 user interfaces based on the TestStand language setting. Use the following functions to localize your user interface:

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

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

Other User Interface Utilities

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

Making a Dialog Box 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 custom user interfaces.

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

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 *NI 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.

While many of the functions specified in the tsapicvi library are simple wrappers to API methods that require no storage of information, there are several functions, especially those containing Get or New, where TestStand is actively allocating new memory to hold the information. In any instance where you are using a function of this type, you must release the allocated memory at the end of your code using calls to CA_FreeMemory, CA_DiscardObjHandle, or a similar function.

If you are concerned about whether a function returns a piece of data that needs to be manually released, refer to the *LabWindows/CVI Help* or *TestStand Help* for that function. Both of these resources explicitly state if the function is allocating memory and often contain additional code fragments explaining how to use the function.

The following are examples of functions that allocate memory:

```
TS_PropertyGetValString()
TS_PropertyGetValIDispatch()
TS_PropertyGetPropertyObject()
TS_NewEngine()
TS_SeqFileNewEditContext()
TS_EngineNewSequence()
```

The following example uses one of the previous functions and then releases the memory:

```
char *stringVal = NULL;
TS_PropertyGetValString (propObj, &errorInfo,
    "Step.Limits.String", 0, &stringVal);
...
CA_FreeMemory (stringVal);
```

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 *NI 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()));
```

Handling Events

ActiveX controls can generate events to notify your application of user input and application events. To handle events in LabWindows/CVI, you must register a callback function using the event callback registration functions in the instrument driver for an ActiveX control. Use the `CA_UnregisterEventCallback` function if you need to close the callback before closing your application.

Refer to Chapter 6, *Creating Custom User Interfaces in LabWindows/CVI*, for more information about handling events that TestStand UI Controls generate.

B

Adding Type Libraries to LabWindows/CVI DLLs

If a DLL contains export information or if a LabWindows/CVI DLL file contains a type library, the LabWindows/CVI Adapter automatically populates the Function control on the LabWindows/CVI Module tab with all of the function names exported from the DLL. In addition, when you select a function in the DLL, the adapter queries the export information or the type library for the parameter list information and displays it in the Parameters Table control on the LabWindows/CVI Module tab. If a DLL was not created with LabWindows/CVI 7.0 or later, or if the DLL does not have type library information, you must enter the parameter information manually in the Parameters Table control.

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.

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 using the LabWindows/CVI Module tab 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.
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.

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

Field Name	Data Type	In/ Out	Description
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	tReplaceStringPtr 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 *NI 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.

Updating Step Properties

You can use the following two methods to pass data between your code module and TestStand.

- Using **tTestData** structure.
- Using the sequence context ActiveX reference. This method allows you to call the TestStand ActiveX API functions to set the variables used to store the results of your test, such as Step.Result.PassFail.

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

Note The values set using the sequence context ActiveX reference take precedence over the values set using the **tTestData** structure. In other words, if you use both methods to set the value of the same variable, the values you set using the sequence context ActiveX reference are recognized. The values you set using the **tTestData** structure are ignored.

You can use both the sequence context ActiveX reference and the **tTestData** structure together in your code module provided that you do not try to set the same variable twice. For example, if you use the sequence context ActiveX reference to set the value of Step.Result.PassFail and then use the **tTestData** structure to set the value of Step.Result.ReportText, both values are set correctly.

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>\AdapterSupport\CVI` 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 at ni.com/support include the following:
  - **Self-Help Resources**—For answers and solutions, visit the award-winning National Instruments Web site for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on.
  - **Free Technical Support**—All registered users receive free Basic Service, which includes access to hundreds of Application Engineers worldwide in the NI Discussion Forums at ni.com/forums. National Instruments Application Engineers make sure every question receives an answer.

    For information about other technical support options in your area, visit ni.com/services or contact your local office at ni.com/contact.
- **Training and Certification**—Visit ni.com/training for self-paced training, eLearning virtual classrooms, interactive CDs, and Certification program information. 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, National Instruments Alliance Partner 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.

Index

A

C

D

E

H

I

K

L

M

N

P

R

S

T

U

W