NI TestStand™

Using LabWindows™/CVI™ with TestStand

May 2008
373201C-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 358 (0) 9 725 72511, France 01 57 66 24 24, Germany 49 89 7413130, India 91 80 41190000, Israel 972 3 6393737, Italy 39 02 41309277, Japan 0120-527196, 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 media, 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:

»	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.
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.
`monospace italic`	Italic text in this font denotes text that is a placeholder for a word or value that you must supply.
Platform	Text in this font denotes a specific platform and indicates that the text following it applies only to that platform.

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 LabWindows/CVI Code Modules

1

Role of LabWindows/CVI in a TestStand-Based System

NI TestStand is a test management environment 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 such as the process model, step types, and user interfaces. You can use LabWindows/CVI in the following ways to accomplish much of this customization:

- Create code modules, such as tests and actions, TestStand can call using the LabWindows/CVI Adapter
- Create custom user interfaces for test systems
- 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 the code modules return. Additionally, the code modules TestStand calls can access the complete TestStand application programming interface (API) for advanced applications.

Custom User Interfaces

You can use the LabWindows/CVI development environment to build custom user interfaces for test systems and for creating custom sequence editors. Typically, custom user interfaces are designed for use in production test systems. You can also create user interfaces using the TestStand User Interface (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 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, *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:

- 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

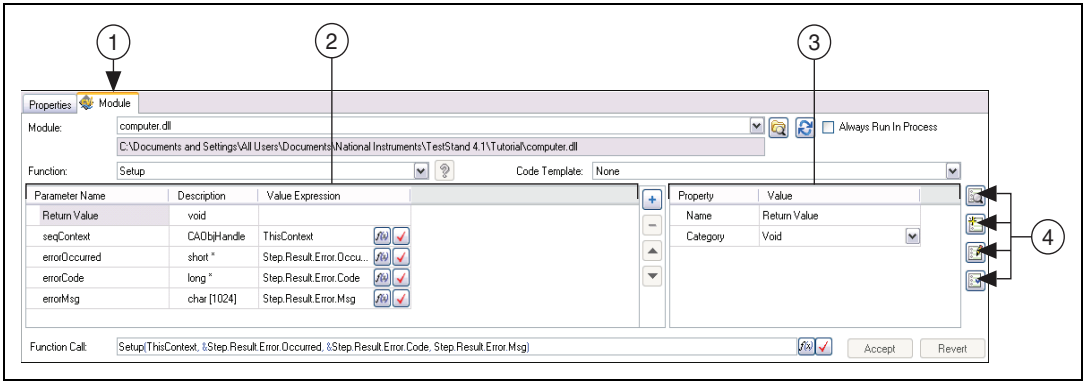
You can call LabWindows/CVI code modules from TestStand using the LabWindows/CVI Adapter.

Required LabWindows/CVI Settings

All 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.

LabWindows/CVI Module Tab

Use the LabWindows/CVI Module tab in the TestStand Sequence Editor to configure calls to LabWindows/CVI code modules. Select a step that uses the LabWindows/CVI Adapter to view the LabWindows/CVI Module tab in the Step Settings pane, as shown in Figure 2-1.



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

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 includes Source Code buttons for creating and editing code modules 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 demonstrate creating and debugging only 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:

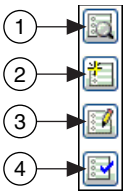
- **Parameter Name**—Displays a symbolic name for the parameter.
- **Description**—Displays the short description of the parameter type using C syntax.
- **Value Expression**—Displays the argument expression to pass.

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

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

The Source Code Files button launches the CVI Source Code Files dialog box, in which you can specify the source file that contains the function 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 or Edit Code buttons. 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 launches 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.

Note If LabWindows/CVI returns a warning that some TestStand API files were not found when you open a project, remove the files from the project and re-add them from the `<National Instruments>\Shared\CVI\instr\TestStand\API` directory for LabWindows/CVI 8.5 and later and from the `<CVI>\instr\TestStand\API\CVI` directory for LabWindows/CVI 8.1.1 or earlier.

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.

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

Complete the following steps to insert a new step that uses the LabWindows/CVI Adapter and then configure the step to call a code module.

1. Launch the TestStand Sequence Editor and select **LabWindows/CVI Adapter** on the Insertion Palette.
2. Open a new Sequence File window if one is not already open.
3. Select **File»Save As** and save the sequence file as `<TestStand Public>\Tutorial\CallCVICodeModule.seq`. The `<TestStand Public>` directory is located at `C:\Documents and Settings\All Users\Documents\National Instruments\TestStand x.x` on Windows 2000/XP and at `C:\Users\Public\Documents\National Instruments\TestStand x.x` on Windows Vista.
4. Insert a Pass/Fail step in the Main step group of the Sequence File window and rename the new step `CVI Pass/Fail Test`.
5. On the LabWindows/CVI Module tab of the Step Settings pane, click the **File Browse** button, select `<TestStand Public>\Tutorial\CallCVICodeModule.dll`, and click **Open**.

6. 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 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.

7. 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.

8. Select **File»Save** to save the sequence file.
9. 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 the step calls.

   When the execution completes, the resulting report indicates the step passed. The code module always returns `True` as its Pass/Fail output parameter.

10. Select **File»Unload All Modules** to instruct TestStand to unload the DLL the step calls so you can rebuild the DLL in the next chapter. Close the Execution window.

3

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

You can use the LabWindows/CVI Adapter to create new code modules to call from TestStand, and to edit and debug existing code modules.

Creating a New Code Module

Complete the following steps to create a new code module from TestStand.

1. Launch the TestStand Sequence Editor and select the **LabWindows/CVI Adapter** in the Insertion Palette.
2. Open `<TestStand Public>\Tutorial\CallCVICodeModule.seq`. if it is not already open. You created this sequence file 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. Insert a Numeric Limit Test step after the `CVI Pass/Fail Test` step and rename it `CVI Numeric Limit Test`.
4. Select the `CVI Numeric Limit Test` step and use the LabWindows/CVI Module tab of the Step Settings pane to complete the following steps.

   a. For the **Module** control, click the **File Browse** button, shown at left, and select `<TestStand Public>\Tutorial\CallCVICodeModule.dll`.
   b. Type `NumericLimitTest` in the Function ring control.
   c. Select **NumericLimit template for LabWindows/CVI** in the Code Template ring control.

5. Click the **Source Code Files** button, shown at left, 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 `<TestStand Public>\Tutorial\CallCVICodeModule.prj`.

c. Click **Close**.

6. Click the **Create Code** button, shown at left, to create a code module.

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

   TestStand creates a new code module based on the available source code templates for the TestStand Numeric Limit Test and opens the 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 if the step passes or fails. Code modules can update step properties by passing step properties as parameters to and from the module or by using the TestStand API in the module. If you use a default code template from National Instruments to create a module, TestStand creates the parameters needed to access the step properties for you.

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

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

8. Save and close the source file. Leave LabWindows/CVI open.
9. In the LabWindows/CVI project window, select **Build»Create Debuggable Dynamic Link Library** to rebuild the DLL.
10. Return to the TestStand Sequence Editor and click the **LabWindows/CVI Module** tab. Notice that TestStand automatically updates the function prototype and parameter values based on the information stored in the code template for the Numeric Limit Test step type.
11. Save the sequence file as `<TestStand Public>\Tutorial\CallCVICodeModule2.seq`.
12. Select **Execute»Single Pass** to run the sequence file using the Single Pass Execution entry point. When the execution completes, the resulting report indicates the step passed with a numeric measurement of `10.0`.

13. Select **File»Unload All Modules** to unload the DLL.
14. Leave this sequence file open so you can use it in the next tutorial.

Refer to the *Code Templates Tab* section of Chapter 13, *Custom Step Types*, of the *NI TestStand Reference Manual* for more information about creating code templates for step types.

Editing an Existing Code Module

Complete the following steps to edit an existing code module.

1. Open `<TestStand Public>\Tutorial\CallCVICodeModule2.seq`, if it is not already open.
2. Right-click the `CVI Numeric Limit Test` step and select **Edit Code** from the context menu. 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, select **Execute»Single Pass** to run the sequence file using the Single Pass Execution entry point.When the execution completes, the resulting report indicates the step failed and the code module returns `5` in the Measurement field.

Debugging a Code Module

Complete the following steps to debug a code module you call from TestStand using the LabWindows/CVI Adapter.

1. Open `<TestStand Public>\Tutorial\CallCVICodeModule.seq`.
2. Place a breakpoint on the `CVI Pass/Fail Test` step.
3. Save the sequence file and select **Execute»Run MainSequence** to start an execution of `MainSequence`.
4. When the execution pauses, click the **Step Into** button 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, complete all executions and select **Configure»Adapters**. Then select **LabWindows/CVI** and click the **Configure** button. 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 that confirms all modules will be unloaded. Begin step 3 again.

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

5. Click the **Step Into** or **Step Over** button on the LabWindows/CVI toolbar to begin single-stepping through the code module. You can click the **Continue** button at any time to finish single-stepping through the code module.
6. When you finish single-stepping through the code module, click the **Finish Function** button on the LabWindows/CVI toolbar to return to TestStand. The execution pauses at the next step in the sequence.
7. Select **Debug»Resume** in TestStand to complete the execution.
8. Select **File»Unload All Modules** to unload the DLL.
9. Close the Execution window.

4

Using LabWindows/CVI Data Types with TestStand

TestStand provides number, string, Boolean, and object reference built-in data types. 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. 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, so TestStand converts LabWindows/CVI data types 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
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.

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

LabWindows/CVI C Data Type	TestStand Data Type
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.

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 use 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 NUL character 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 it is large enough to hold the contents of the string argument and the trailing NUL character. After the function returns, the LabWindows/CVI Adapter copies the value 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 NUL 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 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 in 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 Chapter 11, *Type Concepts*, of the *NI TestStand Reference Manual* for more information about where TestStand stores custom data types. Refer to the *NI TestStand Help* for more information about the Type Properties dialog box.

Creating TestStand Data Types from LabWindows/CVI Structs

Complete the following steps to create a TestStand data type that matches a LabWindows/CVI struct and 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 `<TestStand Public>\Tutorial\CallCVICodeModule2.seq`, if it is not already open.
2. In the Sequence File window, right-click and select **View»Types** from the context menu. Make sure the `CallCVICodeModule2.seq` sequence file is selected in the View Types For pane.
3. Select the **Custom Data Types** item in the Types window.
4. Right-click the **Custom Data Types** item and select **Insert Custom Data Type»Container** to insert a new data type. Rename the new container data type `CVITutorialStruct`.
5. Click the `CVITutorialStruct` item. Click the plus sign to expand the `CVITutorialStruct` item.
6. Right-click inside the Types window under the `CVITutorialStruct` item and select **Insert Field»Number** to insert a new field in the data type. Rename the new field `Measurement`.
7. Right-click inside the Types window and select **Insert Field»String** to insert another new field in the CVITutorialStruct container data type. Rename the new field `Buffer`.

   You have completed the `CVITutorialStruct` container data type. Leave the sequence file open, and continue to the next tutorial.

   Refer to the *NI TestStand Help* and to Chapter 12, *Standard and Custom Data Types*, of the *NI TestStand Reference Manual* for more information about custom data types and the Types window.

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` item in the Types window 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. Click 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 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 a step calls.

1. Click the **CallCVIModule2.seq** tab in the Sequence File window.
2. Click the Variables pane.
3. Right-click the **Locals** item on the Variables pane 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** pane 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 and select the `<TestStand Public>\Tutorial\CallCVICodeModule.dll` file.
10. Type `PassStructTest` in the **Function** control.
11. Click the **Add Parameter** button, shown at left, 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. Click the **Source Code Files** button to launch the CVI Source Code Files window. Complete the following steps to select the source file and project file to use when inserting thefunction.
    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 `<TestStand Public>\Tutorial\CallCVICodeModule.prj` file.
    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 Public>\Tutorial` directory 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");
}
```

Add the following statement to the top of the source file to include the declaration of the strcpy function:

```
#include <ansi_c.h>
```

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`.

    Single-step through the sequence and 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. Close the Execution window.

5

Configuring the LabWindows/CVI Adapter

You can configure the TestStand LabWindows/CVI Adapter to select a LabVIEW server, reserve loaded functions for execution, establish a code template policy, and change legacy settings.

Select **Configure»Adapters** to launch the Adapter Configuration dialog box, select **LabWindows/CVI** in the Adapter column, and click the **Configure** button to launch the LabWindows/CVI Adapter Configuration dialog box, as shown in Figure 5-1.

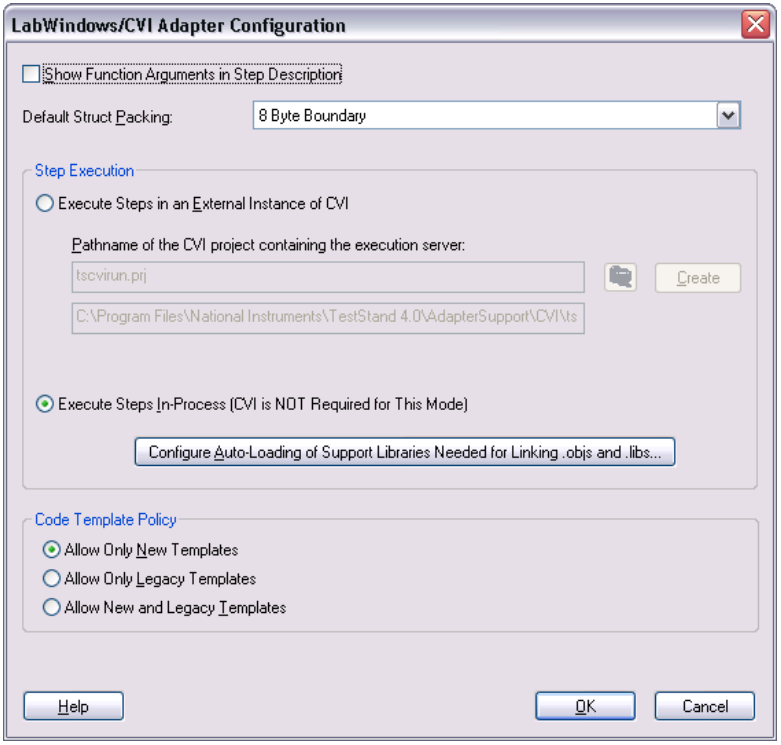


Figure 5-1. LabWindows/CVI Adapter Configuration Dialog Box

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 you use to create DLLs determines the structure packing value. For LabWindows/CVI, the default structure packing can be either 1- or 8-byte. For example, in Microsoft 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 in 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 Public>\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 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 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 you call with the LabWindows/CVI Adapter.

If you click Step Into in the TestStand Sequence Editor 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 on which the object file or static library file depends 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 Public>\AdapterSupport\CVI\AutoLoadLibs` directory. You can also click the **Configure Auto-Loading of Support Libraries Needed for Linking .objs and .libs** button in the LabWindows/CVI Adapter

Configuration dialog box to launch the Auto-Load Library Configuration dialog box, as shown in Figure 5-2.

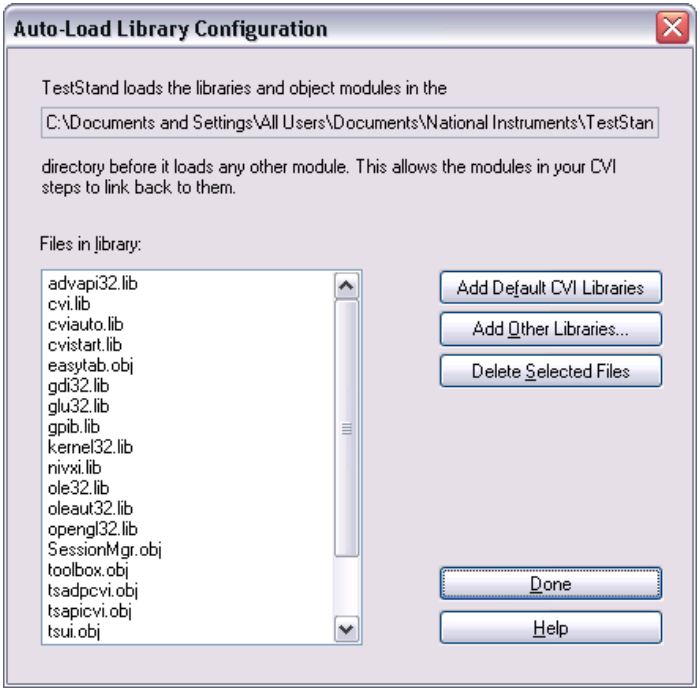


Figure 5-2. Auto-Load Library Configuration Dialog Box

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

- Click the **Add Default CVI Libraries** button 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 the **Add Other Libraries** button to search for files to copy to the auto-load library directory.
- Click the **Delete Selected Files** button to remove the selected files from the auto-load library directory.

Note Data Execution Prevention (DEP) is a Windows security feature that prevents an application from executing dynamically loaded code. Calling LabWindows/CVI object and static library files from an executable with DEP enabled, such as a custom user interface, is not supported and results in an error.

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 with 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 in-process code modules, the code modules must exist in DLLs 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 the TestStand Sequence Editor while the execution is currently suspended on a step that calls into a LabWindows/CVI DLL you are debugging, LabWindows/CVI suspends on the first statement in the DLL function.

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

Loading Subordinate DLLs

TestStand directly loads and runs the DLLs you specify on the LabWindows/CVI Module tab for the LabWindows/CVI Adapter. Because code modules most likely call subsidiary DLLs, such as instrument drivers, you must ensure that the operating system can find and load any DLL you specify.

The LabWindows/CVI Adapter first attempts to load subordinate DLLs using the following search directory precedence:

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

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

For backward compatibility, when the LabWindows/CVI Adapter fails to load a DLL, the adapter temporarily sets the current working directory to the directory of the DLL and attempts to load subordinate DLLs using the following deprecated search directory precedence:

1. The directory that contains the application that loaded the adapter
2. **(Windows 2000 and Windows XP SP1 and earlier)** The current working directory of the application, which the adapter sets to the directory that contains the DLL it calls directly
3. The `Windows\System32` and `Windows\System` directories
4. The `Windows` directory
5. **(Windows XP SP2 and later)** The current working directory of the application, which the adapter sets to the directory that contains the DLL it calls directly
6. The directories listed in the PATH environment variable

Note National Instruments does not recommend placing subordinate DLLs in the directory that contains the application that loaded the adapter because TestStand might 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 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 do not want the global settings for the adapter to affect the tools and step types you create for use with the LabWindows/CVI Adapter.

Code Template Policy

Use the Code Template Policy section in the LabWindows/CVI Adapter Configuration dialog box to specify if TestStand allows you to create new test code modules using old, or *legacy*, code module templates. These legacy code module templates are files you can call from previous versions of TestStand. Refer to Appendix C, *Calling Legacy LabWindows/CVI Code Modules*, for more information about legacy code module templates.

If you enable the Allow Only New Templates option and create a new code module from the LabWindows/CVI Module tab, TestStand creates a new code module based on the code template for the specified step type. If the step type has multiple code templates available, TestStand launches the Choose Code Template dialog box, as shown in Figure 5-3, in which you can select the code template to use for the new code module.

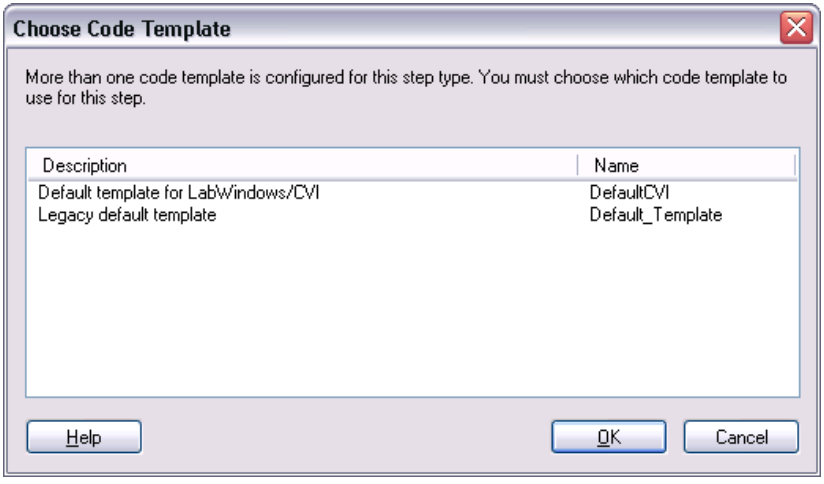


Figure 5-3. Choose Code Template Dialog Box

If you enable 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 enable 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.

Refer to the *NI TestStand Help* for more information about the Choose Code Template dialog box.

6

Creating Custom User Interfaces in LabWindows/CVI

You can create custom user interfaces and create user interfaces for other components, such as custom step types.

National Instruments recommends reading Chapter 9, *Creating Custom User Interfaces*, of the *NI TestStand Reference Manual* to obtain a general understanding of the TestStand UI Controls before you proceed with this chapter.

TestStand User Interface Controls

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

Select **Create»ActiveX** and select a UI control whose name begins with `TestStand UI` to add a TestStand UI Control to a panel in the User Interface Editor. Double-click the control to launch the standard LabWindows/CVI Edit Control dialog box, in which you can configure the control. Right-click the control and select **Properties** from the context menu to open property pages a UI control supports.

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. TestStand copies these files to the `<National Instruments Shared>\CVI\TestStand\API` directory for LabWindows/CVI 8.5 and later and to the `CVI>\instr\TestStand\API\CVI` directory for LabWindows/CVI 8.1.1 or earlier.

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 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 user interfaces, 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 the ActiveX control supports, the driver contains a function you can use to programmatically create an instance of the ActiveX control. The ActiveX driver also includes functions 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 the driver includes. 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 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 the `GetObjHandleFromActiveXCtrl` function 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 the thread to apartment threaded. If you use the `CA_InitActiveXThreadStyleForCurrentThread` function to initialize the thread yourself, you must use `COINIT_APARTMENTTHREADED` as the threading model.

Refer to Appendix A, *Using the TestStand ActiveX APIs in LabWindows/CVI* for general information about programming the TestStand API from 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 the controls generate
- Start TestStand
- Wait in a main event loop until you close the application
- Shut down TestStand

User interfaces can also include a menu bar that contains non-TestStand items and items that invoke TestStand commands.

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

TestStand installs the source code files for the default user interfaces in the `<TestStand>\UserInterfaces` and `<TestStand Public>\UserInterfaces` directories. To modify the installed user interfaces or to create new user interfaces, modify the files in the `<TestStand Public>\UserInterfaces` directory. You can use the read-only source files for the default user interfaces in the `<TestStand>\UserInterfaces` directory as a reference. When you modify installed files, rename the files after you modify them if you want to create a separate custom component. You do not have to rename the files after you modify them if you only want to modify the behavior of an existing component. If you do not rename the files and you use the files in a future version of TestStand, changes National Instruments makes to the component might not be compatible with the modified version of the component. Storing new and customized files in the `<TestStand Public>` directory ensures that new installations of the same version of TestStand do not overwrite the customizations and ensures that uninstalling TestStand does not remove the files you customize.

TestStand no longer includes example user interfaces that use the TestStand API. These examples contained a large amount of complex source code, and they provided less functionality than the simpler examples that use the TestStand UI Controls. National Instruments recommends using the 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 Public>\UserInterfaces\Simple\CVI\TestExec.c`	`SetupActiveXControls`

`<TestStand Public>\UserInterfaces\Full-Featured\CVI\TestExec.c`	`GetActiveXControlHandles`
	`RegisterActiveXEventCallbacks`
	`ConnectTestStandControls`
	`ConnectStatusBarPanes`
	`RebuildMenuBar`

Enabling Sequence Editing

The TestStand UI Controls support both Operator Mode and Editor Mode. To allow the user to create and edit sequence files, set the `ApplicationMgr.IsEditor` property to `True` for the Application Manager control. You can also specify the `/editor` command-line flag to set the property.

Handling Events

TestStand UI Controls generate events to notify the 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 LabWindows/CVI automatically calls when the control generates 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 the user interface application, use the `TSUI_ApplicationMgrStart` driver function to invoke the `ApplicationMgr.Start` method, 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, 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 the menu item.

Call the `TS_InsertCommandsInMenu` function when the application rebuilds the menu bar in a `MenuDimmerCallback` function 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 Public>\UserInterfaces\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 user interfaces based on the TestStand language setting. Use the following functions to localize the user interface:

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

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

Other User Interface Utilities

You can also launch dialog boxes modal to TestStand application windows and enable functions to check for stopped executions.

Making Dialog Boxes Modal to TestStand

Code modules that TestStand calls can launch dialog boxes modal to TestStand application windows, such as the TestStand Sequence Editor or custom user interfaces.

Use the following functions the TestStand Utility Functions driver provides to make a dialog box modal to TestStand application windows.

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

Refer to the `<TestStand>\Components\StepTypes\MsgBox\msgbox.c` source file to see how to use these functions.

Checking for Stopped Execution

Code modules TestStand calls can launch dialog boxes or perform other time-consuming operations. In these cases, it can be useful for code modules to periodically check if TestStand terminated or aborted their parent execution so the code modules can stop gracefully so the parent execution can terminate or abort.

Use the following functions the TestStand Utility Functions driver provides to enable code modules that TestStand calls to verify if the execution that called the function has stopped.

- `TS_CancelDialogIfExecutionStops`
- `TS_CancelDialogIfExternalExecutionStops`

Refer to the dialog box code in the following example source files to see how to use these functions:

- `<TestStand Public>\Examples\Demo\C\computer.c`
- `<TestStand Public>\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 User Interface (UI) Controls from LabWindows/CVI code modules and user interface source code.

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 becoming 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, so 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 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. Refer to the *Adding and Releasing References* section of this appendix for more information about the CA_DiscardObjHandle function.

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

Invoking Methods

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

The following function shows 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 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 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 has built-in properties, which are static with respect to the TestStand API. The TestStand API recognizes each of these properties, allowing you to access them in the programming language you specify. Examples of built-in properties are the `Sequence.Name` property and the `SequenceContext.Sequence` property.

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

```
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 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 you define, these variables and properties are dynamic with respect to the TestStand API. The TestStand API provides the `PropertyObject` class so you can access dynamic properties and variables from within code modules, where you use 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 `SequenceContext` 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 shows how to obtain a handle to the TestStand Engine from the `SequenceContext` object, how to call a method on the engine to acquire a version string, and how to release 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 does not release the object. Repeatedly opening references to objects without closing them can cause the system to run out of memory.

While many of the functions specified in the `tsapicvi.fp` 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 the code using calls to `CA_FreeMemory`, `CA_DiscardObjHandle`, or similar functions.

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 NI 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 `PropertyObject.SetValNumber` method has an options input argument that accepts many different numeric constants.

The header file for the ActiveX driver defines all constants and enumerations 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

TestStand controls can generate events to notify the application of user input and application events, such as the completion of an execution. 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, and use the `CA_UnregisterEventCallback` function if you need to close the callback before closing the application.

Refer to Chapter 6, *Creating Custom User Interfaces in LabWindows/CVI*, for more information about handling events the 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 you want to include in the type library.
2. Add the function panel file to the 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 the **Type Library** button 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 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 the 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 the LabWindows/CVI documentation for more information about adding type libraries to DLLs.

C

Calling Legacy LabWindows/CVI 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 standard and extended legacy prototypes. In earlier versions of TestStand, National Instruments recommended using 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. Refer to Chapter 5, *Configuring the LabWindows/CVI Adapter*, for more information about configuring the LabWindows/CVI Adapter for creating new legacy-style code modules.

The legacy prototypes contain the **tTestData** and **tTestError** structure parameters**,** 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 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 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 to which the inBuffer, outBuffer, and errorMessage fields point.
seqContextDisp	struct IDispatch *	In	Dispatch pointer to the sequence context. This value is `NULL` if you choose not to pass the sequence context.

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

Field Name	Data Type	In/ Out	Description
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 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 the code module and TestStand.

- Using the **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 the 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 the LabWindows/CVI Adapter updates and the value translation, if any, the adapter makes.

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

Structure Member	Valid Values 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 the code module if 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 Public>\AdapterSupport\CVI` directory to the 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 award-winning National Instruments Web site at ni.com for technical support and professional services:

- **Support**—Technical support resources at ni.com/support include the following:
  - **Self-Help Technical Resources**—For answers and solutions, visit ni.com/support 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. Registered users also receive access to the NI Discussion Forums at ni.com/forums. NI Applications Engineers make sure every question submitted online receives an answer.
  - **Standard Service Program Membership**—This program entitles members to direct access to NI Applications Engineers via phone and email for one-to-one technical support as well as exclusive access to on demand training modules via the Services Resource Center. NI offers complementary membership for a full year after purchase, after which you may renew to continue your benefits.

    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

F

H

I

K

L

M

N

P

R

S

T

U

W