

NI TestStand™

Using LabVIEW™ and LabWindows™/CVI™ with
TestStand

Worldwide Technical Support and Product Information

ni.com

Worldwide Offices

Visit 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.

National Instruments Corporate Headquarters

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

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 THEREFORE 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

Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

National Instruments respects the intellectual property of others, and we ask our users to do the same. NI software is protected by copyright and other intellectual property laws. Where NI software may be used to reproduce software or other materials belonging to others, you may use NI software only to reproduce materials that you may reproduce in accordance with the terms of any applicable license or other legal restriction.

End-User License Agreements and Third-Party Legal Notices

You can find end-user license agreements (EULAs) and third-party legal notices in the following locations:

- Notices are located in the <National Instruments>_Legal Information and <National Instruments> directories.
- EULAs are located in the <National Instruments>\Shared\MDF\Legal\license directory.
- Review <National Instruments>_Legal Information.txt for more information on including legal information in installers built with NI products.

Trademarks

CVI, LabVIEW, National Instruments, NI, ni.com, NI TestStand, the National Instruments corporate logo, and the Eagle logo are trademarks of National Instruments Corporation. Refer to the *Trademark Information* at ni.com/trademarks for other National Instruments trademarks.

The mark LabWindows is used under a license from Microsoft Corporation. Windows is a registered trademark of Microsoft Corporation in the United States and other countries. Other product and company names mentioned herein are trademarks or trade names of their respective companies.

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

Patents

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

Export Compliance Information

Refer to the *Export Compliance Information* at ni.com/legal/export-compliance for the National Instruments global trade compliance policy and how to obtain relevant HTS codes, ECCNs, and other import/export data.

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 **Options»Settings»General** directs you to pull down the **Options** menu, select the **Settings** item, and select **General** 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 bold

Bold text in this font denotes the messages and responses that the computer automatically prints to the screen. This font also emphasizes lines of code that are different from the other examples.

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 1

Role of LabVIEW and LabWindows/CVI in a TestStand-Based System

Code Modules.....	1-1
Custom User Interfaces	1-2
Custom Step Types.....	1-2
LabVIEW Adapter.....	1-2
LabWindows/CVI Adapter.....	1-3

PART I

Using LabVIEW with TestStand

Chapter 2

Calling LabVIEW VIs from TestStand

Required LabVIEW Settings	2-1
LabVIEW Module Tab.....	2-1
Creating and Configuring a New Step Using the LabVIEW Adapter.....	2-4
Creating a New Step That Calls a VI in the Context of a LabVIEW Project.....	2-5
Creating New Steps That Call LabVIEW Class Member VIs.....	2-6
Executing the Sequence.....	2-7

Chapter 3

Creating, Editing, and Debugging LabVIEW VIs from TestStand

Creating a New VI from TestStand.....	3-1
Creating a New LabVIEW Project and Adding a New VI from TestStand.....	3-2
Editing an Existing VI from TestStand	3-4
Debugging VIs.....	3-5

Chapter 4

Using LabVIEW Data Types with TestStand

Calling VIs with Enumeration Parameters	4-5
Updating Enumeration Parameter Values	4-5
Normalizing Numeric Values You Pass to Enumeration Parameter Values.....	4-7
Calling VIs with Ring Control Parameters.....	4-7
Calling VIs with String Parameters.....	4-8
Calling VIs with Cluster Parameters	4-9
Specifying Each Cluster Element Individually	4-10
Passing Existing TestStand Container Variables to LabVIEW.....	4-10
Creating a New Custom Data Type.....	4-11
Updating Cluster Element Mapping.....	4-12
Creating TestStand Data Types from LabVIEW Clusters.....	4-13

Chapter 5

Configuring the LabVIEW Adapter

Selecting a LabVIEW Server.....	5-1
Using a LabVIEW Run-Time Engine or Other Executable Server	5-3
Per-Step Configuration of the LabVIEW Adapter	5-4
Code Template Policy.....	5-4
Legacy VI Settings.....	5-5
LabVIEW Project Settings.....	5-6

Chapter 6

Creating Custom User Interfaces in LabVIEW

TestStand User Interface Controls.....	6-1
TestStand VIs and Functions.....	6-1
Creating Custom User Interfaces.....	6-2
Example User Interfaces.....	6-2
Caveats for Using the Example User Interfaces	6-2
Configuring the TestStand UI Controls.....	6-4
Enabling Sequence Editing.....	6-4
Handling Events.....	6-4
Starting and Shutting Down TestStand.....	6-6
Menu Bars and Menu Event Handling	6-6
Localization	6-7
Other User Interface Utilities.....	6-8
Making Dialog Boxes Modal to TestStand.....	6-8
Checking for Suspended or Stopped Executions within Code Modules	6-8
Running User Interfaces	6-9

Chapter 7

Effectively Using LabVIEW with TestStand

Organizing LabVIEW-Based Systems	7-1
LabVIEW Projects.....	7-2
Editing VIs in LabVIEW Projects	7-2
Updating and Patching VIs in LabVIEW Projects	7-3
Deploying VIs in LabVIEW Projects	7-3
LabVIEW Packed Project Libraries	7-4
Editing VIs in LabVIEW Packed Project Libraries.....	7-4
Updating and Patching VIs in LabVIEW Packed Project Libraries.....	7-5
Deploying VIs in LabVIEW Packed Project Libraries.....	7-5
Stand-alone VIs.....	7-6
Editing Stand-alone VIs.....	7-6
Updating and Patching Stand-alone VIs.....	7-6
Deploying Stand-alone VIs.....	7-7
LabVIEW Libraries	7-8
LabVIEW Project Libraries	7-8

LabVIEW-Built Shared Libraries	7-8
Editing VIs in LabVIEW-Built Shared Libraries	7-9
Special Considerations When Using LabVIEW with TestStand Systems	7-9
Using Relative Paths	7-9
Memory Management for Large Data Sets	7-9
Reusing VIs	7-9
Reserving Loaded VIs for Execution	7-10
Partially Specifying LabVIEW Clusters	7-10
Executing VIs with Separate Compiled Code	7-10
Conditional Disable Structures and Symbols	7-11
XControls	7-11
Building a TestStand Deployment with LabVIEW	7-11
Creating Custom Step Types that Call VIs for Substeps	7-12
Using LabVIEW Classes with TestStand	7-13
Network-Published Shared Variables	7-14
Deploying Variables	7-14
Using an Aliases File	7-15
LabVIEW Utility Step Types	7-16
Symmetric Multiprocessing in VIs Executed from TestStand	7-16
LabVIEW Run-Time Engine Execution Threads	7-18
CPU Affinity for LabVIEW Execution Threads	7-18
General Strategies for Optimizing LabVIEW Code Module Performance on SMP Systems	7-20
LabVIEW INI File Tokens Associated with Execution Systems	7-21

Chapter 8

Calling LabVIEW VIs on Remote Computers

Configuring a Step to Run Remotely	8-1
Configuring the LabVIEW VI Server to Run VIs Remotely	8-2
User Access to VI Server	8-3

Chapter 9

Using the TestStand ActiveX APIs in LabVIEW

Invoking Methods	9-1
Accessing Built-In Properties	9-1
Accessing Dynamic Properties	9-2
Releasing ActiveX References	9-3
Using TestStand API Constants and Enumerations	9-3
Obtaining a Different Interface for TestStand Objects	9-4
Acquiring a Derived Class from the PropertyObject Class	9-5
Duplicating COM References in LabVIEW Code Modules	9-5
Setting the Preferred Execution System for LabVIEW VIs	9-6
Handling Events	9-7

Chapter 10

Calling Legacy LabVIEW VIs

Format of Legacy VIs	10-1
Test Data Cluster	10-2
Error Out Cluster	10-3
Input Buffer String Control.....	10-4
Invocation Info Cluster	10-4
Sequence Context Control	10-4

PART II

Using LabWindows/CVI with TestStand

Chapter 11

Calling LabWindows/CVI Code Modules from TestStand

Required LabWindows/CVI Settings	11-1
LabWindows/CVI Module Tab	11-1
Source Code Buttons	11-2
Creating and Configuring a New Step Using the LabWindows/CVI Adapter	11-3

Chapter 12

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

Creating a New Code Module from TestStand.....	12-1
Editing an Existing Code Module from TestStand.....	12-3
Debugging a Code Module	12-3

Chapter 13

Using LabWindows/CVI Data Types with TestStand

Calling Code Modules with String Parameters.....	13-2
Calling Code Modules with Object Parameters.....	13-3
Calling Code Modules with Struct Parameters.....	13-3
Calling Code Modules with Pointer Parameters.....	13-4
Creating TestStand Data Types from LabWindows/CVI Structs.....	13-4
Creating a New Custom Data Type	13-4
Specifying Structure Passing Settings	13-5
Calling a Function with a Struct Parameter	13-5

Chapter 14

Configuring the LabWindows/CVI Adapter

Showing Function Arguments in Step Descriptions.....	14-1
Setting the Default Structure Packing Size.....	14-2
Selecting Where Steps Execute.....	14-2
Executing Code Modules in an External Instance of LabWindows/CVI.....	14-2
Debugging Code Modules.....	14-2
Executing Code Modules In-Process.....	14-3
Object and Library Code Modules.....	14-3
Source Code Modules.....	14-4
Debugging DLL Code Modules.....	14-4
Loading Subordinate DLLs.....	14-4
Per-Step Configuration of the LabWindows/CVI Adapter.....	14-5
Code Template Policy.....	14-5

Chapter 15

Creating Custom User Interfaces in LabWindows/CVI

TestStand User Interface Controls.....	15-1
Creating and Configuring ActiveX Controls.....	15-1
Programming with ActiveX Controls.....	15-1
Creating Custom User Interfaces.....	15-2
Example User Interfaces.....	15-2
Configuring the TestStand UI Controls.....	15-3
Enabling Sequence Editing.....	15-3
Handling Events.....	15-4
Handling Variants.....	15-4
Starting and Shutting Down TestStand.....	15-5
Menu Bars.....	15-6
Localization.....	15-6
Other User Interface Utilities.....	15-7
Making Dialog Boxes Modal to TestStand.....	15-7
Checking for Suspended or Stopped Execution within Code Modules.....	15-7

Chapter 16

Using the TestStand ActiveX APIs in LabWindows/CVI

Using ActiveX Drivers in LabWindows/CVI.....	16-1
Invoking Methods.....	16-2
Accessing Built-In Properties.....	16-2
Accessing Dynamic Properties.....	16-3
Adding and Releasing References.....	16-4
Using TestStand API Constants and Enumerations.....	16-6
Handling Events.....	16-7

Chapter 17

Adding Type Libraries to LabWindows/CVI DLLs

Generating Type Library Information	17-1
Specifying String Buffer Size in a Type Library	17-2

Chapter 18

Calling Legacy LabWindows/CVI Code Modules

Prototypes of Legacy Code Modules	18-1
tTestData Structure	18-2
tTestError Structure	18-4
Updating Step Properties	18-4
Example Code Module	18-6

Appendix A

Technical Support and Professional Services

Index

Role of LabVIEW and LabWindows/CVI in a TestStand-Based System

You can use the NI TestStand test management environment to organize and execute code modules written in a variety of languages and application development environments (ADEs), including LabVIEW and LabWindows™/CVI™. TestStand handles core test management functionality, including the definition and execution of the overall testing process, user management, report generation, database logging, and more.



Note If you are a new user, National Instruments recommends that you read Chapter 1, *Introduction to TestStand*, of the *Getting Started with TestStand* manual and complete the tutorials in the manual to familiarize yourself with TestStand concepts and features.

TestStand can work in a variety of different testing scenarios and environments because it allows extensive customization of components, such as process models, step types, and user interfaces. You can use LabVIEW and LabWindows/CVI in the following ways to accomplish much of this customization:

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

Code Modules

TestStand can call LabVIEW virtual instruments (VIs) with a variety of connector pane configurations. TestStand can call VIs that reside on the same computer as TestStand or on other network computers, including computers running the LabVIEW Real-Time (RT) module. TestStand can call LabWindows/CVI code modules with a variety of function prototypes.

TestStand can also pass data to the VIs and code modules it calls and store the data the VI or code module returns. Additionally, the VIs and code modules TestStand calls can access the complete TestStand application programming interface (API) for advanced applications.

Custom User Interfaces

You can use the LabVIEW or 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. With the LabVIEW Full or Professional Development System and with LabWindows/CVI, you can also create user interfaces using the TestStand User Interface (UI) Controls and the TestStand API. Refer to the *NI TestStand Help* for more information about creating custom user interfaces.

Refer to the *NI TestStand User Interface Controls Reference Poster* for an overall picture of a TestStand User Interface and the TestStand UI Controls API. Refer to the *NI TestStand API Reference Poster* and the *NI TestStand Help* for detailed information about the TestStand API and TestStand UI Controls.

Custom Step Types

You can use LabVIEW to create VIs and LabWindows/CVI to create code modules you call from custom step types. These VIs and code modules can implement editable dialog boxes and other features of custom step types. Refer to the *NI TestStand Help* for more information about custom step types.

LabVIEW Adapter

The LabVIEW Adapter offers advanced functionality for calling VIs from TestStand. You can use the LabVIEW Adapter to complete the following tasks:

- Call VIs with arbitrary connector panes.
- Call VIs that exist on disk, in LLBs, or in LabVIEW packed project libraries. You can also call VIs in the context of a LabVIEW project or LabVIEW class member VIs.



Note You must have LabVIEW 2009 SP1 f1 or later to use LabVIEW projects in TestStand.

- Call member VIs from LabVIEW classes using LabVIEW dynamic dispatching, when required.



Note You must have LabVIEW 2012 or later to use LabVIEW dynamic dispatching when calling LabVIEW classes in TestStand.

- Call VIs on remote computers.
- Run VIs using the LabVIEW development system or a LabVIEW executable.
- Run VIs using the LabVIEW Run-Time Engine.
- Create and edit VIs from TestStand.
- Debug VIs (step in/step out) from TestStand.

Refer to the [Using LabVIEW with TestStand](#) part of this manual for information about using LabVIEW with TestStand.

LabWindows/CVI Adapter

The LabWindows/CVI Adapter offers advanced functionality for calling code modules from TestStand. You can use the LabWindows/CVI Adapter to complete the following tasks:

- Call code modules with arbitrary prototypes in `.dll`, `.lib`, `.obj`, and `.c` code modules files
- 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

Refer to the [Using LabWindows/CVI with TestStand](#) part of this manual for information about using LabWindows/CVI with TestStand.

Using LabVIEW with TestStand

Use this section of this manual to learn how to use LabVIEW with TestStand.

- Chapter 2, *Calling LabVIEW VIs from TestStand*—Use the LabVIEW Adapter to call VIs from TestStand.
- Chapter 3, *Creating, Editing, and Debugging LabVIEW VIs from TestStand*—Use the LabVIEW Adapter to create new VIs to call from TestStand and to edit and debug existing VIs.
- Chapter 4, *Using LabVIEW 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, Error, LabVIEWAnalogWaveform, and others. You can create container data types to hold any number of other data types.
- Chapter 5, *Configuring the LabVIEW Adapter*—Configure the LabVIEW Adapter to select a LabVIEW server, reserve loaded VIs for execution, establish a code template policy, configure auto deployment of libraries with shared variables from LabVIEW projects, and change legacy VI settings.
- Chapter 6, *Creating Custom User Interfaces in LabVIEW*—You can create custom user interfaces and create user interfaces for other components, such as custom step types.
- Chapter 7, *Effectively Using LabVIEW with TestStand*—You can use the advanced features of LabVIEW to organize and deploy TestStand VIs.
- Chapter 8, *Calling LabVIEW VIs on Remote Computers*—You can directly call VIs on remote computers, including computers that run the LabVIEW development system or a LabVIEW executable and PXI controllers that run the LabVIEW Real-Time (RT) module.
- Chapter 9, *Using the TestStand ActiveX APIs in LabVIEW*—You can use the TestStand API or TestStand User Interface (UI) Controls from LabVIEW test and user interface VIs.
- Chapter 10, *Calling Legacy LabVIEW VIs*—In versions of TestStand earlier than 3.0, you could call VIs only with a specific set of controls and indicators. Using TestStand 3.0 or later, you can call VIs with a wide variety of connector panes, including VIs with legacy configurations.

Calling LabVIEW VIs from TestStand

Use the LabVIEW Adapter to call VIs from TestStand.

Required LabVIEW Settings

All the tutorials in Part I of this manual require you to have the LabVIEW development system and TestStand installed on the same computer. In addition, you must configure the LabVIEW Adapter to run VIs using the LabVIEW development system. Refer to Chapter 5, [Configuring the LabVIEW Adapter](#), for more information about configuring these settings for the adapter.

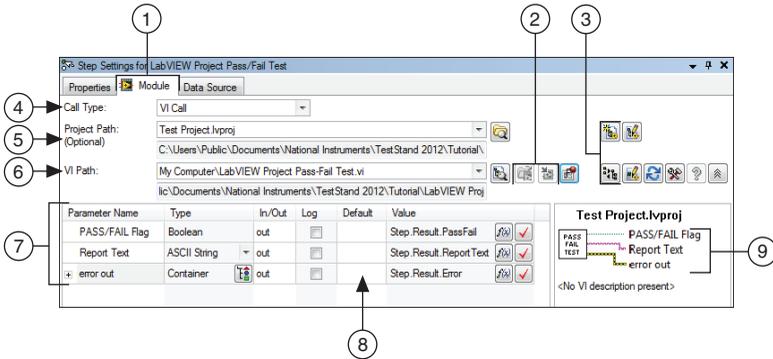
Confirm the following settings in LabVIEW:

- To edit or run a VI from TestStand, you must include the VI in the VI Server: Exported VIs list in LabVIEW. By default, LabVIEW allows access to all VIs. Select **Tools»Options** to launch the Options dialog box. Select the **VI Server** category and browse to the Exported VIs section.
- Confirm that "*" is included in the VI Server: Exported VIs list and that the **Allow Access** option is enabled.

LabVIEW Module Tab

Use the LabVIEW Module tab in the TestStand Sequence Editor to configure calls to VIs. Select a step that uses the LabVIEW Adapter to view the LabVIEW Module tab on the Step Settings pane, as shown in Figure 2-1.

Figure 2-1. LabVIEW Module Tab



- | | |
|--|---|
| 1 Module Tab | 6 LabVIEW VI Path Control |
| 2 Express VI Buttons | 7 VI Parameter Table |
| 3 LabVIEW Project, Source Code, and Help Buttons | 8 Log Column |
| 4 Call Type Ring Control | 9 VI Context Help Image and Description |
| 5 LabVIEW Project Path Control | |

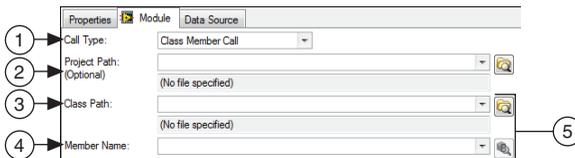
Use the LabVIEW Module tab in the sequence editor to complete the following tasks:

- Specify whether the step calls a member VI from a LabVIEW class or calls a VI that is a not a member of a LabVIEW class
- Select the LabVIEW project that refers to the VI (optional)
- Select the VI the step executes
- Specify whether LabVIEW shows the front panel of the VI when TestStand calls the VI

The Module tab includes source code buttons for creating and editing a VI in LabVIEW and Express VI buttons for selecting an Express VI and converting an Express VI to a standard VI.

When you configure a call to a member VI from a LabVIEW class, the LabVIEW Module tab displays the Class Path and Member Name controls instead of the VI Path control, as shown in Figure 2-2.

Figure 2-2. LabVIEW Class and Class Member Controls on the LabVIEW Module Tab



- | | |
|--------------------------------|--|
| 1 Call Type Ring Control | 4 LabVIEW Class Member Name Control |
| 2 LabVIEW Project Path Control | 5 LabVIEW Class and Class Member Buttons |
| 3 LabVIEW Class Path Control | |

The LabVIEW Module tab also contains the following specific information about the VI to call:

- **VI Parameter Table**—Contains information about each control or indicator wired to the connector pane of the VI. These are the parameters of the VI.
 - **Parameter Name**—Caption text or label text, if no caption exists, of the control or indicator.
 - **Type**—TestStand equivalent data type for the control or indicator. Refer to Chapter 4, *Using LabVIEW Data Types with TestStand*, for more information about how LabVIEW data types map to TestStand data types.
 - **In/Out**—Specifies whether the parameter is an input (control on the VI front panel wired to the connector pane) or an output (indicator on the VI front panel wired to the connector pane).
 - **Log**—When you enable this option, the step logs the parameter as an additional result. Enabling this option is equivalent to enabling the Value to Log option on the Additional Results panel of the Properties tab of the Step Settings pane. Refer to the *NI TestStand Help* for more information about the Additional Results panel.
 - **Default**—When you enable this option, TestStand uses the default value of the control for the parameter, cluster element, or array element. This option is not available when the terminal of the VI is marked as Required.
 - **Value**—A TestStand expression. For input parameters, TestStand passes the result of this expression to the VI unless you place a checkmark in the option in the Default column. For output parameters, TestStand stores the data the VI returns in the location this expression specifies.



Note The VI Parameter Table displays parameters according to the order in the VI context help image.

- **LabVIEW Project, Source Code, and Help Buttons**—Use these buttons for the following tasks:
 - Edit a LabVIEW project
 - Create or edit a VI in LabVIEW
 - Refresh the parameter information for the VI
 - Launch the LabVIEW Advanced Settings window
 - Launch the help associated with the VI, if available
 - Undock the *LabVIEW Help*

Refer to the *NI TestStand Help* for more information about the LabVIEW Project, Source Code, and Help buttons on the LabVIEW Module tab.

- **VI Context Help Image and Description**—Displays the context help image of the VI as shown in the LabVIEW Context Help window and displays the description of the VI from the Documentation tab of the LabVIEW VI Properties dialog box. When you click a label or terminal of the VI icon, TestStand highlights the parameter in the VI Parameter Table.

Click the **NI TestStand Help** or **Help Topic** button on the Help toolbar to access the *NI TestStand Help*, which provides more information about the options on the LabVIEW Module tab.

Creating and Configuring a New Step Using the LabVIEW Adapter

Complete the following steps to insert a new step that uses the LabVIEW Adapter and configure the step to call a stand-alone VI.

1. Launch the sequence editor and click the LabVIEW icon, as shown in the following figure, at the top of the Insertion Palette to specify the module adapter the step uses. You can also select adapters from the Adapter ring control on the Environment toolbar. The adapter you select applies only to the step types that can use the module adapter. Refer to the *NI TestStand Help* for more information about the Insertion Palette and the TestStand toolbars.



2. Open a new Sequence File window if one is not already open.
3. Select **File»Save <filename> As** and save the sequence file as `<TestStand Public>\Tutorial\Call LabVIEW VI.seq`. The `<TestStand Public>` directory is located by default at `C:\Users\Public\Documents\National Instruments\TestStand on Windows 7/Vista` and at `C:\Documents and Settings\All Users\Documents\National Instruments\TestStand on Windows XP`.
4. Insert a Pass/Fail Test step in the Main step group in the Sequence File window and rename the new step `LabVIEW Pass/Fail Test`.
5. On the LabVIEW Module tab of the Step Settings pane, click the **Browse for VI** button, as shown in the following figure and located to the right of the VI Path control, select `<TestStand Public>\Tutorial\LabVIEW Pass-Fail Test.vi`, and click **Open**. TestStand reads the description and connector pane information from the VI and updates the LabVIEW Module tab so you can configure the data to pass to and from the VI.



6. In the VI Parameter Table, enter `Step.Result.PassFail` in the Value column of the **PASS/FAIL Flag** output parameter and enter `Step.Result.ReportText` in the Value column of the **Report Text** output parameter.



Note All expression fields support typeahead and autocomplete with drop-down lists and context-sensitive highlighting. At any point while editing an expression, you can press `<Ctrl-Space>` to show a drop-down list of valid expression elements.

When TestStand calls the VI, it places the value the VI returns in the **PASS/FAIL Flag** and **Report Text** indicators into the `Result.PassFail` and `Result.ReportText` properties of the step, respectively.

Notice that TestStand automatically fills in the Value column of the **error out** output parameter with the `Step.Result.Error` property.



Note By default, when a VI uses the standard LabVIEW **error out** cluster as an output parameter, TestStand automatically passes that value into the `Step.Result.Error` property for the step. You can also update the value manually. If an error occurs during execution of the VI and the **error out** cluster is passed to `Step.Result.Error`, TestStand launches the Run-Time Error dialog box by default.

7. Save the sequence file.

Creating a New Step That Calls a VI in the Context of a LabVIEW Project

Complete the following steps to insert a new LabVIEW step and configure the step to call a VI in the context of a LabVIEW project.



Note You must have LabVIEW 2009 SP1 f1 or later to use LabVIEW projects in TestStand.

1. Insert another Pass/Fail Test step in the Main step group and rename the new step `LabVIEW Project Pass/Fail Test`.
2. Click the **Browse for LabVIEW Project** button, as shown in the following figure and located to the right of the Project Path control, on the LabVIEW Module tab and select `<TestStand Public>\Tutorial\Test Project.lvproj` and click **Open**. Notice that some of the TestStand button icons and tool tips change to indicate support for LabVIEW projects.



3. Click the **Browse for VI in LabVIEW Project** button, as shown in the following figure and located to the right of the VI Path control, to launch the Select VI from LabVIEW Project dialog box.



4. Select `LabVIEW Project Pass-Fail Test.vi` and click **OK**. TestStand reads the description and connector pane information from the VI and updates the LabVIEW Module tab so you can configure the data to pass to and from the VI. Notice that the path defined in the project is selected as the VI path.
5. In the VI Parameter Table, enter `Step.Result.PassFail` in the Value column of the **PASS/FAIL Flag** output parameter and enter `Step.Result.ReportText` in the Value column of the **Report Text** output parameter.
6. Save the sequence file.

Refer to Chapter 7, *Effectively Using LabVIEW with TestStand*, for more information about using LabVIEW projects in TestStand.

Creating New Steps That Call LabVIEW Class Member VIs

Complete the following steps to insert two new LabVIEW steps, configure the first step to call a static member VI from a LabVIEW class to create a LabVIEW class object, and configure the second step to call a dynamically dispatched member method VI on the created object.



Note You must have LabVIEW 2012 or later to call member VIs from a LabVIEW class in TestStand and to use LabVIEW dynamic dispatching when calling LabVIEW classes in TestStand.

1. Insert an Action step in the Main step group and rename the new step `Create LabVIEW Class Object` and complete the following steps to configure the step.
 - a. Select **Class Member Call** from the Call Type ring control on the Module tab. Notice that some of the TestStand button icons and tool tips change to indicate support for LabVIEW class member calls.
 - b. Click the **Browse for LabVIEW Project** button on the LabVIEW Module tab and select `<TestStand Public>\Tutorial\Test Project.lvproj` and click **Open**.
 - c. Click the **Browse for LabVIEW Class in LabVIEW Project** button, as shown in the following figure and located to the right of the Class Path control, on the LabVIEW Module tab to launch the Select a Class from LabVIEW Project dialog box.



- d. Select `LabVIEW Child Class.lvclass` and click **OK**.
- e. Select **LabVIEW Child Static Factory Member.vi** from the Member Name ring control.

- f. In the VI Parameter Table, enter `Locals.myChildObject` in the Value column of the **LabVIEW Child Class out** output parameter.
 - g. Right-click the `Locals.myChildObject` value and select **Create Locals.myChildObject»Object Reference** from the context menu to create `Locals.myChildObject` as an object reference local variable.
2. Insert another Pass/Fail Test step in the Main step group and rename the new step `LabVIEW Class Member Pass/Fail Test` and complete the following steps to configure the step.
 - a. Select **Class Member Call** from the Call Type ring control on the Module tab.
 - b. Click the **Browse for LabVIEW Project** button on the LabVIEW Module tab and select `<TestStand Public>\Tutorial\Test Project.lvproj` and click **Open**.
 - c. Click the **Browse for LabVIEW Class in LabVIEW Project** button on the LabVIEW Module tab to launch the Select a Class from LabVIEW Project dialog box.
 - d. Select `LabVIEW Parent Class.lvclass` and click **OK**.
 - e. Select **LabVIEW Dynamic Pass-Fail Test Member.vi** from the Member Name ring control.
 - f. In the VI Parameter table, enter `Locals.myChildObject` in the Value column of the **LabVIEW Parent Class in** input parameter and the **LabVIEW Child Class out** output parameter.
 - g. In the VI Parameter Table, enter `Step.Result.PassFail` in the Value column of the **PASS/FAIL Flag** output parameter and enter `Step.Result.ReportText` in the Value column of the **Report Text** output parameter.
 3. Save the sequence file.

Refer to the [Using LabVIEW Classes with TestStand](#) section of Chapter 7, *Effectively Using LabVIEW with TestStand*, for more information about using LabVIEW classes in TestStand.

Executing the Sequence

Complete the following steps to execute the sequence of steps.

1. 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 steps passed. The VI always returns `True` as the **Pass/Fail** output parameter.

When you initiate an execution, by default, the TestStand Sequence Analyzer uses the rules and settings in the current sequence analyzer project to analyze the active sequence file and detect the most common situations that can cause run-time failures. The sequence analyzer prompts you to resolve the reported errors before you execute the sequence file. If no errors exist or if you select to continue execution with errors, the sequence editor opens an Execution window.

Click the **Toggle Analyze File Before Executing** button, as shown in the following figure, on the Sequence Analyzer toolbar or select **Debug»Sequence Analyzer»Toggle Analyze**

File Before Executing to enable or disable this option. Click the **Analyze <filename>** button on the Sequence Analyzer toolbar to analyze a sequence file at any time during development. Refer to the *NI TestStand Help* for more information about using the sequence analyzer.



2. Close the Execution window.

Creating, Editing, and Debugging LabVIEW VIs from TestStand

Use the LabVIEW Adapter to create new VIs to call from TestStand and to edit and debug existing VIs.

Creating a New VI from TestStand

Complete the following steps to create a new VI from TestStand.

1. Launch the TestStand Sequence Editor and select the **LabVIEW Adapter** on the Insertion Palette.
2. Open `<TestStand Public>\Tutorial\Call LabVIEW VI.seq`. You created this sequence file in the *Creating and Configuring a New Step Using the LabVIEW Adapter* section of Chapter 2, *Calling LabVIEW VIs from TestStand*.



Note If you did not save `Call LabVIEW VI.seq`, you can open the `Call LabVIEW VI.seq` solution file from the `<TestStand Public>\Tutorial\Solution` directory. If you use the solution file, save any changes you make to the file in the `<TestStand Public>\Tutorial` directory so you do not overwrite the installed solution file.

3. Insert a **Numeric Limit Test** step after the **LabVIEW Class Member Pass/Fail Test** step and rename it `LabVIEW Numeric Limit Test`.
4. Select the **LabVIEW Numeric Limit Test** step and use the **LabVIEW Module** tab to complete the following steps.
 - a. Click the **Create VI** button, as shown in the following figure, to create a new VI.



- b. In the File dialog box, browse to the `<TestStand Public>\Tutorial` directory, enter `LabVIEW Numeric Limit Test.vi` in the **File Name** control, and click **OK**. TestStand creates a new VI based on the available code templates for the **TestStand Numeric Limit Test** and opens the VI in LabVIEW.



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 pass step properties as parameters to and from the module or use the TestStand API in the module to update step properties. When 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.

- c. In LabVIEW, select **Window»Show Block Diagram** to open the block diagram.
 - d. Right-click the **Numeric Measurement** indicator terminal, select **Create»Constant** from the context menu, and enter `10.0`.
 - e. Save and close the VI.
5. Return to the sequence editor and click the **LabVIEW Module** tab. Notice that TestStand automatically updates the output parameters for the VI based on the information stored in the code template for the Numeric Limit Test step type.
 6. Save the sequence file as `<TestStand Public>\Tutorial\Call LabVIEW VI 2.seq`.
 7. 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`.



Note You must have Internet Explorer 7.0 or later to view TestStand reports.

8. Leave the sequence file open so you can use it in the next tutorial.

Refer to the *NI TestStand Help* for more information about creating code templates for step types.

Creating a New LabVIEW Project and Adding a New VI from TestStand

Complete the following steps to create a new LabVIEW project and add a new VI to that project from TestStand.

1. Open `<TestStand Public>\Tutorial\Call LabVIEW VI 2.seq`, if it is not already open.
2. Insert a Numeric Limit Test step after the LabVIEW Numeric Limit Test step and rename it `LabVIEW Project Numeric Limit Test`.

3. Select the **LabVIEW Project Numeric Limit Test** step and use the LabVIEW Module tab to complete the following steps.
 - a. Click the **Create LabVIEW Project** button, as shown in the following figure, to create a new LabVIEW Project.



- b. In the File dialog box, browse to the `<TestStand Public>\Tutorial` directory, enter `LabVIEW Project Test.lvproj` in the File Name control, and click **OK**. TestStand creates a new LabVIEW Project and opens the project in LabVIEW.
4. In TestStand, select the **LabVIEW Project Numeric Limit Test** step and use the LabVIEW Module tab to complete the following steps.
 - a. Click the **Add or Remove VIs from LabVIEW Project** button, as shown in the following figure, to launch the Add or Remove Items from LabVIEW Project dialog box.



- b. Right-click the **My Computer** item in the LabVIEW Project control and select **New VI** from the context menu to launch the Select the LabVIEW VI to Create dialog box.
 - c. In the Select LabVIEW VI to Create dialog box, browse to the `<TestStand Public>\Tutorial` directory, enter `LabVIEW Project Numeric Limit Test.vi` in the File Name control, and click **OK**. TestStand creates a new VI based on the available code templates for the TestStand Numeric Limit Test step type and adds the VI to the LabVIEW project.
 - d. Right-click the new VI under the My Computer item and select **Edit VI** from the context menu. TestStand opens the VI in LabVIEW under the project context.



Note The LabVIEW project context is indicated at the bottom left corner of the LabVIEW VI front panel.

- e. In LabVIEW, open the block diagram for the VI.
 - f. Right-click the **Numeric Measurement** indicator terminal, select **Create»Constant** from the context menu, and enter `10.0`.
 - g. Save and close the VI.
5. Return to TestStand, select `LabVIEW Project Numeric Limit Test.vi` in the Add or Remove Items from the LabVIEW Project dialog box, and click **Select** to close the Add or Remove Items from LabVIEW Project dialog box.

On the LabVIEW Module tab, notice that TestStand automatically updates the output parameters for the VI based on the information stored in the code template for the Numeric Limit step type.

6. Save the sequence file.
7. 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 steps passed with a numeric measurement of 10.0.
8. Leave the sequence file open so you can use it for the next tutorial.

Editing an Existing VI from TestStand

Complete the following steps to edit an existing VI from TestStand.

1. Open `<TestStand Public>\Tutorial\Call LabVIEW VI 2.seq` if it is not already open. You created this sequence file in the [Creating a New LabVIEW Project and Adding a New VI from TestStand](#) section of this chapter.
 2. Select the **LabVIEW Pass/Fail Test** step and use the LabVIEW Module tab to complete the following steps.
 - a. Click the **Edit VI** button, as shown in the following figure, on the LabVIEW Module tab or right-click the **LabVIEW Pass/Fail Test** step and select **Edit Code** from the context menu. LabVIEW becomes the active application in which the LabVIEW Pass-Fail Test VI is open.
-
- b. Open the block diagram for the VI and change the **PASS/FAIL Flag** Boolean constant to `False`.
 - c. Save and close the VI.
3. Return to TestStand, select the **LabVIEW Project Pass/Fail Test** step, and use the LabVIEW Module tab to complete the following steps.
 - a. Click the **Edit VI** button on the LabVIEW Module tab or right-click the **LabVIEW Project Pass/Fail Test** step and select **Edit Code** from the context menu. LabVIEW becomes the active application and the LabVIEW Project Pass-Fail Test VI is open in the LabVIEW project context.
 - b. Open the block diagram for the VI and change the **PASS/FAIL Flag** Boolean constant to `False`.
 - c. Save and close the VI.
 4. Return to TestStand and 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 steps failed, and the VI returns `False` in the **PASS/FAIL Flag** indicator.
 5. Close the Execution window.

Debugging VIs

Complete the following steps to debug VIs you call from TestStand using the LabVIEW Adapter.

1. Open `<TestStand Public>\Tutorial\Call LabVIEW VI.seq`.
2. Place a breakpoint on the LabVIEW Pass/Fail Test step. Refer to Chapter 4, *Debugging Sequences*, of the *Getting Started with TestStand* manual for more information about using breakpoints in sequences.
3. Save the sequence file and select **Execute»Run MainSequence** to start an execution of `MainSequence`. The execution pauses at the LabVIEW Pass/Fail Test step.
4. Complete the following steps to debug the LabVIEW Pass-Fail Test VI the LabVIEW Pass/Fail Test step calls.
 - a. Click the **Step Into** button on the Debug toolbar in TestStand. LabVIEW becomes the active application, in which the LabVIEW Pass-Fail Test VI is open and in a suspended state.
 - b. Open the block diagram of the suspended VI.
 - c. Click the **Step Into** or **Step Over** button on the LabVIEW toolbar to begin stepping through the VI. You can click the **Finish VI** button at any time to finish stepping through the VI.
 - d. When you finish stepping through the VI, click the **Return to Caller** button on the LabVIEW toolbar to return to TestStand. The execution pauses at the LabVIEW Project Pass/Fail Test step.
5. Complete the following steps to debug the LabVIEW Project Pass-Fail Test VI the LabVIEW Project Pass/Fail Test step calls from a LabVIEW project.
 - a. Click the **Step Into** button on the Debug toolbar in TestStand. LabVIEW becomes the active application, in which the LabVIEW Project Pass-Fail Test VI is open and in a suspended state in the LabVIEW project context.
 - b. Open the block diagram of the suspended VI.
 - c. Click the **Step Into** or **Step Over** button on the LabVIEW toolbar to begin stepping through the VI. You can click the **Finish VI** button at any time to finish stepping through the VI.
 - d. When you finish stepping through the VI, click the **Return to Caller** button on the LabVIEW toolbar to return to TestStand. The execution pauses at the Create LabVIEW Class Object step.
6. Click the **Step Over** button on the Debug toolbar in TestStand to execute the Create LabVIEW Class Object step. The execution pauses at the LabVIEW Class Member Pass/Fail Test step in the sequence.

7. Complete the following steps to step into the LabVIEW Dynamic Pass-Fail Test Member VI the LabVIEW Class Member Pass/Fail Test step calls from a LabVIEW class.
 - a. Click the **Step Into** button on the Debug toolbar in TestStand. LabVIEW becomes the active application, in which the LabVIEW Dynamic Pass-Fail Test Member VI is open and in a suspended state in the LabVIEW project context.
 - b. Open the block diagram of the suspended VI.
 - c. Click the **Step Into** or **Step Over** button on the LabVIEW toolbar to begin stepping through the VI. You can click the **Finish VI** button at any time to finish stepping through the VI.
 - d. When you finish stepping through the VI, click the **Return to Caller** button on the LabVIEW toolbar to return to TestStand. The execution pauses at the end of the Main step group.
8. Select **Debug»Resume** or click the **Resume** button on the Debug toolbar in TestStand to complete the execution.
9. Close the Execution window.

You can run a VI multiple times before returning to TestStand. However, LabVIEW passes the results from only the most recent run to TestStand when you finish debugging.

Using LabVIEW 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, Error, LabVIEWAnalogWaveform, and others. You can create container data types to hold any number of other data types. TestStand container data types are analogous to LabVIEW clusters. You can use references to external objects, such as ActiveX (Microsoft ActiveX) objects or VISA sessions, between different types of code modules.

LabVIEW includes a greater variety of built-in data types than TestStand does, so TestStand converts LabVIEW data types in certain ways when calling VIs. Table 4-1 shows how TestStand converts LabVIEW numeric representations, and Table 4-2 shows how TestStand converts LabVIEW controls or indicators.

Table 4-1. TestStand Equivalents for LabVIEW Numeric Representations

LabVIEW Numeric Representation	TestStand Data Type
Real number (U8, U16, U32, I8, I16, I32, SGL, DBL, or EXT)	Number TestStand does not support extended-precision, (EXT) floating-point numbers. TestStand converts any EXT numbers from LabVIEW into double-precision (DBL) numbers.
64-bit integer numeric (I64)	Number {Signed 64-bit integer}
Unsigned 64-bit integer numeric (UI64)	Number {Unsigned 64-bit integer}
Fixed-point numeric	TestStand does not support calling VIs with fixed-point numeric indicators or controls.
Complex number (CSG, CDB, or CXT)	Number TestStand maps each part of the complex number to separate TestStand Number properties. Refer to the previous information about how TestStand converts EXT numbers.

Table 4-2. TestStand Equivalents for LabVIEW Controls and Indicators

LabVIEW Control or Indicator	TestStand Data Type
Enum (U32, U16, or U8)	Number Refer to the <i>Calling VIs with Enumeration Parameters</i> section of this chapter for more information about using the enum data type.
Ring control (Text Ring, Menu Ring, Text and Picture Ring, or Picture Ring)	Number Refer to the <i>Calling VIs with Ring Control Parameters</i> section of this chapter for more information about using the ring control data type.
String	String Refer to the <i>Calling VIs with String Parameters</i> section of this chapter for more information about using the string data type.
Path	Path or string
ActiveX Control or Automation Refnum	Object reference
.NET Refnum	Object reference You cannot pass references to .NET objects you create outside of LabVIEW, such as with the TestStand .NET Adapter, to VIs. You can store references to .NET objects you create in LabVIEW within TestStand properties and then pass them to other VIs.
Waveform	LabVIEW AnalogWaveform To create variables for I64 or UI64 Waveforms, you must create a new custom data type, because the representation of the Y element that corresponds to the standard LabVIEWAnalogWaveform data type is set to double-precision 64-bit floating-point.
Digital waveform	LabVIEWDigitalWaveform
Digital data	LabVIEWDigitalData

Table 4-2. TestStand Equivalents for LabVIEW Controls and Indicators (Continued)

LabVIEW Control or Indicator	TestStand Data Type
Picture	<p>String</p> <p>You must select Binary String from the ring control in the Type column of the VI Parameter Table on the LabVIEW Module tab. Refer to the Calling VIs with String Parameters section of this chapter for information about using the string data type.</p>
Refnum (File I/O, VI, Menu, Queue, TCP connection, and so on)	<p>Number</p> <p>You cannot use references to internal LabVIEW objects inside TestStand or in other types of code modules. You can store only references to LabVIEW objects in TestStand properties and then pass the properties to other VIs.</p>
Timestamp	<p>String</p> <p>TestStand rounds fractions of a second to the nearest millisecond.</p> <p>Refer to the Calling VIs with String Parameters section of this chapter for information about using the string data type.</p>
Error I/O	<p>Error</p> <p>By default, when a VI uses the standard LabVIEW error out cluster as an output parameter, TestStand automatically passes that value into the <code>Step.Result.Error</code> property for the step. You can also update the value manually. If an error occurs during execution of the VI and the error out cluster is passed to <code>Step.Result.Error</code>, TestStand launches the Run-Time Error dialog box by default.</p>
Array of x	Array of TestStand (x)
Variant	Any TestStand data type

Table 4-2. TestStand Equivalents for LabVIEW Controls and Indicators (Continued)

LabVIEW Control or Indicator	TestStand Data Type
LabVIEW Object	Object reference. Refer to the <i>Using LabVIEW Classes with TestStand</i> section of this chapter for more information about LabVIEW Class objects.
Cluster	Container Refer to the <i>Calling VIs with Cluster Parameters</i> section of this chapter for more information about using the container data type.
I/O controls (DAQmx Task Name, DAQmx Channel Name, VISA Resource Name, IVI Logical Name, FieldPoint IO Point, or Motion Resource)	LabVIEWIOControl Refer to the <i>NI TestStand Help</i> for more information about using the DeviceName and SessionNumber properties of the LabVIEWIOControl data type.
IMAQ Session	Number
Other I/O controls (DAQmx Physical Channel Name, Terminal Name, Analog Trigger Source, Scale Name, Device Name, or Switch Name)	String Refer to the <i>Calling VIs with String Parameters</i> section of this chapter for more information about using the string data type.



Note You must have LabVIEW 8.6.1 or later to call VIs with 64-bit integer numeric indicators or controls from TestStand.



Note LabVIEW does not support calling the `PropertyObject.SetValInteger64`, `PropertyObject.GetValInteger64`, `PropertyObject.SetValInteger64ByOffset`, `PropertyObject.GetValInteger64ByOffset`, `PropertyObject.SetValUnsignedInteger64`, `PropertyObject.GetValUnsignedInteger64`, `PropertyObject.SetValUnsignedInteger64ByOffset`, or `PropertyObject.GetValUnsignedInteger64ByOffset` methods. Use the `PropertyObject.GetValVariant` or `PropertyObject.SetValVariant` methods instead.

Calling VIs with Enumeration Parameters

When you configure calls to VIs that use enumerated values as parameters, the Value column of the VI Parameter Table on the LabVIEW Module tab shows a ring control for enumeration input parameters. The ring control contains the items in the LabVIEW enumeration.

At edit time, TestStand stores the numeric value and the corresponding string label value of the enumeration you select.

When you pass an enumeration value as a parameter to TestStand at run time, TestStand stores only the numeric value. In addition, for input parameters at run time, TestStand passes the numeric value to LabVIEW without checking whether the numeric value is within range. Refer to the [Normalizing Numeric Values You Pass to Enumeration Parameter Values](#) section of this chapter for information about when TestStand normalizes enumeration input parameter values to be within range.

Updating Enumeration Parameter Values

If you change the number of items in the enumeration in LabVIEW, you must reload the VI prototype in TestStand before you can execute the sequence.

When you reload a VI prototype and you specify an expression for the enumeration parameter, TestStand always retains the expression

When you reload a VI prototype and you do not specify an expression for the enumeration parameter, TestStand updates enumeration parameter values in the following ways depending on the version of LabVIEW you are using:

- **(LabVIEW 2011 SP1 or earlier)** TestStand retains the numeric value and updates the corresponding string label to match the numeric value. If you delete the numeric value, TestStand updates the string label to use the last enumeration value in the list.
- **(LabVIEW 2012 or later)** TestStand updates enumeration parameter values in a way similar to how LabVIEW updates enumeration controls. TestStand updates enumeration parameter values differently depending on whether the enumeration is an instance of a LabVIEW type definition, as described in Tables 4-3 and 4-4.

Table 4-3 describes how TestStand updates enumeration parameter values when you reload the VI prototype, you do not specify an expression for the enumeration parameter, and the enumeration is an instance of a LabVIEW type definition.

Table 4-3. Updating Enumeration Parameter Values after Reloading a VI Prototype when the Enumeration is an Instance of a LabVIEW Type Definition

Change You Make to LabVIEW Type Definition	Effect You See in TestStand VI Parameter Table	Effect You See in TestStand Step Properties
You modify or delete the string label, and the numeric value currently stored in the step property matches another string label.	TestStand retains the numeric value and updates the corresponding string label to match the numeric value.	TestStand retains the numeric value and updates the corresponding string label.
You move the string label to a different position in the list of enumeration values.	Teststand retains the string label.	TestStand retains the string label and updates the corresponding numeric value.
You delete the string label and the numeric value.	TestStand updates the string label to use the last enumeration value in the list.	TestStand updates the numeric value and updates the string label to use the last enumeration value in the list.

Table 4-4 describes how TestStand updates enumeration parameter values when you reload the VI prototype, you do not specify an expression for the enumeration parameter, and the enumeration is not an instance of a LabVIEW type definition.

Table 4-4. Updating Enumeration Parameter Values after Reloading a VI Prototype when the Enumeration is Not an Instance of a LabVIEW Type Definition

Change You Make in VI	Effect You See in TestStand VI Parameter Table	Effect You See in TestStand Step Properties
You delete the numeric value currently stored in the step property.	TestStand updates the string label to use the last enumeration value in the list.	TestStand updates the numeric value and updates the string label to use the last enumeration value in the list.
You update the string label but retain the numeric value currently stored in the step property.	TestStand updates the string label.	TestStand retains the numeric value and updates the corresponding string label.

Normalizing Numeric Values You Pass to Enumeration Parameter Values

When you use TestStand to specify a VI for a LabVIEW step code module and the VI contains enumeration parameters but you pass a numeric value that is outside the range of valid enumeration values to an enumeration parameter, LabVIEW modifies the value to be within the valid range before executing the VI, as described in Table 4-5.

Table 4-5. LabVIEW Behavior for Normalizing Out-of-Range Numeric Values Passed to Enumerated VI Parameters

TestStand Version	LabVIEW Version That Executes VI	Way in Which Numeric Value is Out of Range	LabVIEW Behavior
2012	2012 or later	Numeric value is negative.	LabVIEW uses the first enumerated value in the list (0).
		Numeric value is greater than the last enumerated value in the list.	LabVIEW uses the last enumerated value in the list.
	Earlier than 2012	Any numeric value.	LabVIEW does not modify any numeric value. If a numeric value is out of range, the behavior is undefined.
Earlier than 2012	Any	Any numeric value.	LabVIEW does not modify any numeric value. If a numeric value is out of range, the behavior is undefined.

Calling VIs with Ring Control Parameters

When you configure calls to VIs that use ring controls as parameters and you are using LabVIEW 2012 or later, the Value column of the VI Parameter Table on the LabVIEW Module tab shows a ring control for LabVIEW ring control input parameters. The ring control contains the items in the LabVIEW ring control.



Note The VI Parameter Table does not show a ring control for a picture ring control.

At edit time, TestStand stores the numeric value and the string label value of the ring control parameter value you select.

When you pass a ring control value as a parameter to TestStand at run time, TestStand stores only the numeric value. In addition, for input parameters at run time, TestStand passes the numeric value to LabVIEW without checking whether the numeric value is within range. This behavior is similar to LabVIEW.

TestStand supports all numeric representations of a ring control parameter. Table 4-1 describes which TestStand data type to use for each representation.

When you reload a VI prototype and you specify an expression for the ring control parameter, TestStand always retains the expression.

When you reload a VI prototype and you do not specify an expression for the ring control parameter, TestStand always retains the numeric value and updates the corresponding string label to match the numeric value. If you modify or delete the numeric value, TestStand displays the numeric value because it cannot find a corresponding string label. This behavior is similar to LabVIEW.

Calling VIs with String Parameters

When you configure calls to VIs that use strings as parameters, you can specify to store the string data as a binary string when reading the data from the VI or when passing the data to the VI. This option is necessary because LabVIEW strings can contain binary data, including NUL characters, but TestStand strings cannot contain NUL characters.

For string parameters, use the ring control in the Type column of the VI Parameter Table on the LabVIEW Module tab to select ASCII String or Binary String. The default value is ASCII String. TestStand does not modify the values of ASCII strings it passes to or from VIs.

Select **Binary String** in the Type column to store a LabVIEW string that contains binary data in a TestStand property. TestStand handles the string data in different ways depending on the version of LabVIEW you are using, as Table 4-6 describes.

Table 4-6. TestStand Behavior for Handling Binary Data in a LabVIEW String

LabVIEW Version	TestStand Behavior
<ul style="list-style-type: none"> • LabVIEW 2012 or later Development System on a non-multibyte operating system • LabVIEW 2009 SP1 or later Development System on a multibyte operating system • LabVIEW 2009 SP1 or later RTE on any operating system 	TestStand compresses the binary data, encodes the compressed data, and stores the compressed data using only printable ASCII characters. To pass a compressed string to a VI, select Binary String in the Type column. TestStand unencodes and decompresses the binary data the string stores before passing it to the VI.

Table 4-6. TestStand Behavior for Handling Binary Data in a LabVIEW String (Continued)

LabVIEW Version	TestStand Behavior
<ul style="list-style-type: none"> • LabVIEW 2011 SP1 or earlier Development System on a non-multibyte operating system • LabVIEW 2009 or earlier Development System on a multibyte system • LabVIEW 2009 or earlier RTE on any operating system 	<p>TestStand escapes the string before storing it and substitutes hexadecimal codes for the unprintable characters, such as the NUL character, in the string. To pass an escaped string to a VI, select Binary String in the Type column. TestStand unescapes the string before passing it to the VI and substitutes the correct character values for the hexadecimal values in the escaped string.</p>



Note To retrieve the raw data in TestStand when the binary data is compressed, call the `PropertyObject.GetValBinary` method of the TestStand API on the variable that stores the binary string.

Calling VIs with Cluster Parameters

When you configure calls to VIs that use clusters as parameters, you can specify that each cluster element maps to a different TestStand expression or that the entire LabVIEW cluster maps to a TestStand data type.

You can create a custom data type that matches a LabVIEW cluster. For input parameters, you can pass the default value for the entire cluster or the default values for specific elements of the cluster control on the front panel of the VI.

For output parameters, you can optionally specify where TestStand stores the cluster value or specific elements of the cluster value.

When you use several VIs with a complex array of clusters that is not used in TestStand, you can use the `LabVIEWClusterArray` data type, which adapts to the array of clusters as needed. First, use the `LabVIEWClusterArray` data type as an output expression that TestStand uses to adapt the data type to fit the array of clusters. Then, you can pass the data type to the remaining VIs as an input.

Specifying Each Cluster Element Individually

To configure each cluster element individually, specify a different TestStand expression for each element of the cluster. Figure 7 shows a VI Parameter Table in which the data source for the Number element of the **Input Cluster** parameter is a local variable. TestStand passes the default value for the String element of the **Input Cluster**.

Figure 7. Input Cluster Data Sources

Parameter Name	Type	In/Out	Log	Default	Value
- Input Cluster	Container	in	<input type="checkbox"/>	<input type="checkbox"/>	
Number	Number (DBL)	in	<input type="checkbox"/>	<input type="checkbox"/>	Locals.Numeric
String	ASCII String	in	<input type="checkbox"/>	<input checked="" type="checkbox"/>	""
PASS/FAIL Flag	Boolean	out	<input type="checkbox"/>		Step.Result.PassFail
Report Text	ASCII String	out	<input type="checkbox"/>		Step.Result.ReportText
+ error out	Container	out	<input type="checkbox"/>		Step.Result.Error

Passing Existing TestStand Container Variables to LabVIEW

Instead of passing each cluster element individually, you can create a TestStand custom data type that maps to the entire LabVIEW cluster.

Use the LabVIEW Cluster Passing tab of the Type Properties dialog box for the new custom data type to specify how TestStand maps subproperties to elements in a LabVIEW cluster. Then, when you specify the data to pass for a cluster parameter, use a variable that is an instance of the new custom data type. Refer to the *NI TestStand Help* for more information about the Type Properties dialog box.

Figure 8 shows how the data passed to the **Input Cluster** parameter is a local variable called ContainerData with a type of InputData. Figure 9 shows the custom InputData data type on the Types pane of the TestStand Sequence Editor.

Figure 8. ContainerData Local Variable

Parameter Name	Type	In/Out	Log	Default	Value
- Input Cluster	Container	in	<input type="checkbox"/>	<input type="checkbox"/>	Locals.ContainerData
Number	Number (DBL)	in	<input type="checkbox"/>	<input type="checkbox"/>	Locals.ContainerData.Number
String	ASCII String	in	<input type="checkbox"/>	<input type="checkbox"/>	Locals.ContainerData.String
PASS/FAIL Flag	Boolean	out	<input type="checkbox"/>		Step.Result.PassFail
Report Text	ASCII String	out	<input type="checkbox"/>		Step.Result.ReportText
+ error out	Container	out	<input type="checkbox"/>		Step.Result.Error

Figure 9. InputData Custom Data Type

Name	Value	Type	Version	Usage	Comment
Step Types					
Edit Substep		Step Type, Type Definition	5.0.0.113	NI_DatabaseTyp...	
Pass/Fail Test		Step Type, Type Definition	5.0.0.0	NI_Types.ini; Call...	
Numeric Limit Test		Step Type, Type Definition	5.0.0.0	NI_Types.ini; Call...	
... <Right click to insert Type>					
Custom Data Types					
InputData		Container, Type Definition	0.0.0.3	Call LabVIEW VI ...	
Number	23	Number			
String	"My String Data"	String			
... <Right click to insert Field>					
... <Right click to insert Type>					
Standard Data Types					

Creating a New Custom Data Type

Use the Create Custom Data Type From Cluster dialog box to create a TestStand custom data type, such as a container, that matches an existing LabVIEW cluster. Click the **Create Custom Data Type** button, as shown in the following figure and located in the Type column of the VI Parameter Table on the LabVIEW Module tab, to launch the Create Custom Data Type From Cluster dialog box, as shown in Figure 10.

**Figure 10.** Create Custom Data Type From Cluster Dialog Box

Cluster Element Label	Item Type	Property Type	Property Name
Number	Number (DBL)	Number	Number
String	ASCII String	String	String

Create Custom Data Type In File:
Call LabVIEW VI 3.seq -- <Selected Sequence File>

Buttons: Help, Create, Cancel

The **Create Custom Data Type From Cluster** dialog box contains the following options:

- **Type Name**—The name of the new custom data type.
- **List Box**—The cluster element labels, associated types, and the property names to which they map in the new custom data type. If the type of cluster element is variant, a combo box becomes visible in the Types column which you can use to select the property for the TestStand type. The property name is customizable.
- **Create Custom Data Type In File**—The location where TestStand creates the new data type. By default, TestStand creates the new data type in the current sequence file.

When you update a strict type definition in LabVIEW, you must also update all instances of the custom data type you created in TestStand from that LabVIEW strict type definition. TestStand does not automatically update custom data types with changes you make in LabVIEW.

Refer to the *NI TestStand Help* for more information about custom data types and the Create Custom Data Type From Cluster dialog box.

Updating Cluster Element Mapping

If the Value column of the VI Parameter Table contains a '???' <Unknown Value>' error for a subproperty of a parameter, TestStand cannot map the subproperty to a cluster element because the cluster item names the TestStand type defines do not match the names in the LabVIEW cluster definition.

Complete the following steps to update the subproperty.

1. Select **View»Types** to launch the Types window.
2. Browse to the definition of the type for the parameter that contains the subproperty with the unknown value. Right-click the type and select **Properties** to launch the Type Properties dialog box.
3. Click the **LabVIEW Cluster Passing** tab.
4. In the Property ring control, select the subproperty with the unknown value. If the Cluster Item Label control is empty or if the subproperty name is misspelled, TestStand cannot map the subproperty to a cluster element.
5. Enter the correct name for the subproperty in the Cluster Item Label control and click **OK**.
6. On the LabVIEW Module tab, click the **Apply Cluster Passing Changes** button, as shown in the following figure and located in the Type column of the VI Parameter Table for the parameter that contains the subproperty.



7. Click **Yes** in the Update Cluster Mapping dialog box to confirm that you want to apply the changes.

Creating TestStand Data Types from LabVIEW Clusters

Complete the following steps to create a TestStand data type that matches a LabVIEW cluster.

1. Open `<TestStand Public>\Tutorial\Call LabVIEW VI 2.seq` if it is not already open. You created this sequence file in the [Creating a New VI from TestStand](#) section of Chapter 3, [Creating, Editing, and Debugging LabVIEW VIs from TestStand](#).
2. Save the sequence file as `Call LabVIEW VI 3.seq`.
3. Select the **LabVIEW Adapter**.
4. Insert a new Pass/Fail Test step in the Main step group after the LabVIEW Project Numeric Limit Test step and rename the step `Pass Container` to `VI`.
5. On the LabVIEW Module tab of the Step Settings pane, click the **Browse for VI** button, located to the right of the VI Path control, and select `<TestStand Public>\Tutorial\VI with Cluster Input.vi`. The **Report Text** output parameter of the VI returns a string that contains the number and string elements of the **Input Cluster** parameter.
6. Click the **Create Custom Data Type** button in the Type column of the **Input Cluster** parameter to launch the Create Custom Data Type From Cluster dialog box. TestStand maps the cluster elements to subproperties in a container called `Input_Cluster`, which is a new TestStand custom data type. You can rename the data type and subproperties as necessary and specify where TestStand stores the new data type.
7. In the Create Custom Data Type From Cluster dialog box, change the type name to `InputData` and click the **Create** button to accept the automatically assigned values and to create the data type in the current sequence file.
8. On the LabVIEW Module tab, remove the checkmark from the Default column for the **Input Cluster** input parameter, click the **Expression Browse** button in the Value column to launch the Expression Browser dialog box, and complete the following steps.
 - a. On the Variables/Properties tab, right-click the **Locals** item and select **Insert Types» InputData** to create a local variable of type `InputData`. Rename the local variable `ContainerData`.
 - b. Right-click the `Number` subproperty of `ContainerData` and select **Properties** from the context menu to launch the Number Properties dialog box.
 - c. Enter `23` in the **Value** control and click **OK**.
 - d. Right-click the `String` subproperty of `ContainerData` and select **Properties** from the context menu to launch the String Properties dialog box.
 - e. Enter `My String Data` in the **Value** control and click **OK**.
 - f. Enter `Locals.ContainerData` in the **Expression** control on the Variables/Properties tab and click **OK**. The Value column for the **Input Cluster** parameter now contains `Locals.ContainerData`.

- g. Click the **Check Expression for Errors** button, as shown in the following figure, to verify that the expression contains valid syntax.



9. Enter `Step.Result.ReportText` in the **Value** column for the **ReportText** output parameter. When TestStand calls the VI, it passes the values in the `ContainerData` local variable to the **Input Cluster** control on the VI and returns the **Number** and **String** elements of the **Input Cluster** parameter to the `ReportText` property of the step.
10. Save the changes.
11. Select **Execute»Single Pass** to run the sequence file using the Single Pass Execution entry point. When the execution completes, the resulting report shows the text the VI with Cluster Input VI returns.
12. Close the Execution window.

Configuring the LabVIEW Adapter

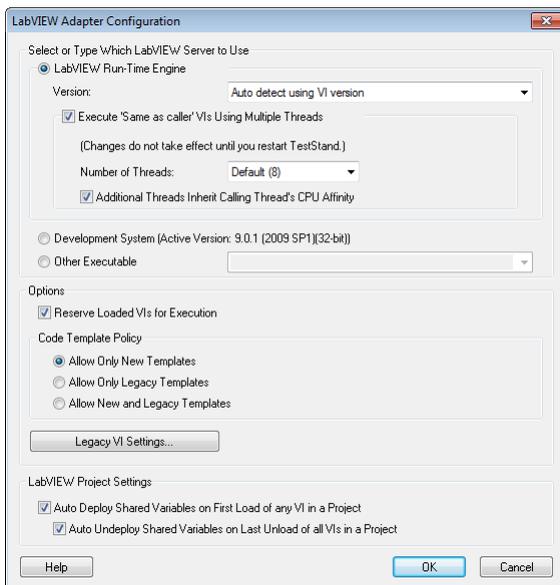
Configure the LabVIEW Adapter to select a LabVIEW server, reserve loaded VIs for execution, establish a code template policy, change legacy VI settings, and specify behavior for deploying LabVIEW shared variables.

Selecting a LabVIEW Server

The LabVIEW Adapter can run VIs using the LabVIEW development system, the LabVIEW Run-Time Engine (RTE), or a LabVIEW executable built with an ActiveX Automation server enabled.

Select **Configure»Adapters** to launch the Adapter Configuration dialog box, select **LabVIEW** in the Adapter column, and click the **Configure** button to launch the LabVIEW Adapter Configuration dialog box, as shown in Figure 5-1. Use this dialog box to select the server you want TestStand to use.

Figure 5-1. LabVIEW Adapter Configuration Dialog Box



- **LabVIEW Run-Time Engine**—Runs VIs in the same process as TestStand to provide optimal performance when calling VIs. When you select this option, you cannot create or edit VIs from TestStand or debug VIs TestStand calls. You must install and select the LabVIEW RTE version that matches the version of the VIs that TestStand executes.

Select the **Auto detect using VI version** option in the Version control to let TestStand automatically detect the version of the LabVIEW RTE to use when executing a LabVIEW code module VI. Use this option when you have VIs saved in different versions of LabVIEW and you want them to always run in the LabVIEW RTE without mass compiling to a specific LabVIEW version.

You can specify whether VIs execute using multiple threads, the number of such threads, and whether the additional LabVIEW execution threads adopt the thread affinity of the TestStand execution thread. These settings apply only to VIs with the preferred execution system set to **Same as caller** and only when the VIs execute using the LabVIEW RTE.

Refer to the *Symmetric Multiprocessing in VIs Executed from TestStand* section of Chapter 7, *Effectively Using LabVIEW with TestStand*, for more information about using symmetric multiprocessing (SMP) in VIs executed from TestStand.



Note This option performs the same behavior as the **Always Run VI in LabVIEW Run-Time Engine** option in the Advanced Settings window, which you access from the LabVIEW Module tab. However, the **Auto detect using VI version** option applies to all LabVIEW steps in a sequence. Refer to the *Per-Step Configuration of the LabVIEW Adapter* section of this chapter for more information about the **Always Run VI in LabVIEW Run-Time Engine** option.

- **Development System (Active Version)**—Creates or edits VIs from TestStand and debugs VIs TestStand calls in LabVIEW. The VIs execute in the LabVIEW development system process. You must have the LabVIEW development system installed on the same computer as TestStand to use this option.
- **Other Executable**—Uses a LabVIEW executable you build with the Build Executable functionality in LabVIEW. When you select this option, you cannot create or edit VIs from TestStand or debug VIs TestStand calls. The version of LabVIEW that built the executable must match the version of the VIs that TestStand executes.

To use this option, enter the ActiveX Automation server name associated with the LabVIEW executable. You must install and register the executable on the same computer as TestStand. Launch the executable once to register it as an ActiveX Automation server. Refer to the <TestStand Public>\Components\RuntimeServers\LabVIEW directory for an example server VI and application build script for LabVIEW.



Note (Windows Vista) Microsoft Windows Vista requires you to log in as a user with administrator privileges to register a server. LabVIEW cannot register ActiveX Automation servers on Windows Vista when building executables without elevation.

Using a LabVIEW Run-Time Engine or Other Executable Server

When you select LabVIEW Run-Time Engine or Other Executable as a server in the LabVIEW Adapter Configuration dialog box, you must ensure that the server can locate the complete hierarchy of VIs, including any subVIs, that TestStand executes. The version of the VIs that TestStand executes must match the version of the LabVIEW RTE or the version of LabVIEW that built the executable. Refer to the *NI TestStand Help* for more information about using the TestStand Deployment Utility. Refer to Chapter 7, *Effectively Using LabVIEW with TestStand*, for more information about calling and deploying LabVIEW VIs.



Note TestStand can use newer versions of the LabVIEW RTE when you install LabVIEW on your development system. You can include newer versions of the LabVIEW RTE in deployments using the Drivers and Components dialog box of the TestStand Deployment Utility. Refer to the *NI TestStand Help* for more information about the Drivers and Components dialog box and the deployment utility.

When you use the TestStand Version Selector to activate a different version of TestStand, TestStand copies new versions of the VIs to the `<LabVIEW>\vi.lib\addons\TestStand` directory. If you select LabVIEW Run-Time Engine as the server on a development system, TestStand might report that it cannot load some VIs in the LabVIEW RTE, even though the VIs run successfully using the LabVIEW development system as the server, because TestStand cannot load a subVI that was saved in a different version of LabVIEW than the LabVIEW RTE specified in the LabVIEW Adapter Configuration dialog box. This discrepancy occurs most often when a top-level VI uses a subVI or controls located in the `<LabVIEW>\vi.lib\addons\TestStand` directory and that subVI was saved in a different version of LabVIEW than the LabVIEW RTE. Mass compiling the `<LabVIEW>\vi.lib\addons\TestStand` directory usually resolves the discrepancy. You can also mass compile top-level VIs, which in turn compiles any subVIs. Mass compile the VIs located in this directory after you activate the new version of TestStand.

This behavior can also occur in the following situations:

- When you try to run a VI you save in the `<TestStand>\API` directory and the `<TestStand Public>\Examples` directory
- When the directories contain VIs with the same names as the subVIs the VI uses
- When the VIs were saved with a different version of LabVIEW

Save VIs in a working directory instead of using the TestStand directories to work around this issue.

Normally, the LabVIEW RTE can execute top-level VIs only when you save the VI with the entire VI hierarchy or when the LabVIEW RTE can locate the subVIs using the LabVIEW search directories. By default, the LabVIEW RTE does not search for required subVIs within the LabVIEW development system, such as files in `vi.lib`. However, when TestStand loads a LabVIEW RTE on a computer where you also installed the LabVIEW development system,

TestStand automatically adds directories from the development system to the search paths for the LabVIEW RTE, but only for directories where the version of the development system matches the version of the LabVIEW RTE. Therefore, when TestStand executes a top-level VI saved without a hierarchy, the LabVIEW RTE searches the directory of the top-level VI and then searches the directories TestStand added to find subVIs. When the LabVIEW RTE finds a subVI saved in a different version of LabVIEW first, the LabVIEW RTE cannot load the subVI and does not load the top-level VI.

Per-Step Configuration of the LabVIEW Adapter

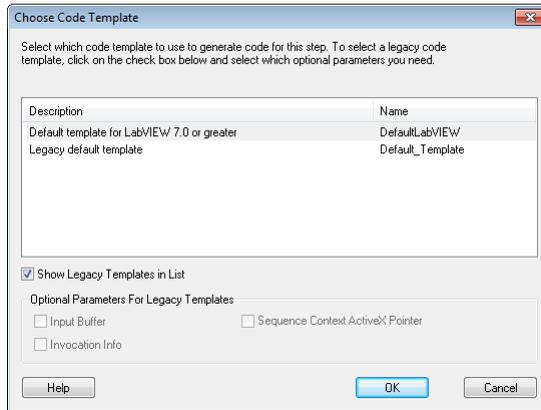
You can direct TestStand to always use the LabVIEW RTE to execute a step. Click the **Advanced Settings** button on the LabVIEW Module tab to launch the LabVIEW Advanced Settings window and enable the **Always Run VI in LabVIEW Run-Time Engine** option in the Advanced Settings window.

When you enable the Always Run VI in LabVIEW Run-Time Engine option, TestStand selects the appropriate version of the LabVIEW RTE according to the version of LabVIEW in which you last compiled the VI. This setting overrides the global setting in the LabVIEW 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.

Code Template Policy

Use the Code Template Policy section in the LabVIEW Adapter Configuration dialog box to allow the use of old, or legacy, VI templates when you create new test VIs. Legacy VI templates are VIs you can call from previous versions of TestStand. Refer to Chapter 10, *Calling Legacy LabVIEW VIs*, for more information about legacy TestStand VIs.

When you select the Allow New and Legacy Templates option in the LabVIEW Adapter Configuration dialog box and create a new VI using the LabVIEW Module tab, TestStand launches the Choose Code Template dialog box, in which you can select the template to use. Enable the **Show Legacy Templates in List** option to show the legacy templates. When you select a legacy template, you enable the Optional Parameters for Legacy Templates section in the Choose Code Template dialog box, in which you can select the optional parameters you want to include as input parameters for the VI, as shown in Figure 5-2.

Figure 5-2. Choose Code Template Dialog Box

When you select the Allow Only Legacy Templates option in the LabVIEW Adapter Configuration dialog box and create a new VI using the LabVIEW Module tab, TestStand launches the Optional Parameters dialog box, in which you select the optional parameters, such as **Input Buffer**, **Invocation Info**, or **Sequence Context ActiveX Pointer**, you want to include as input parameters for the VI.

When you select the Allow Only New Templates option in the LabVIEW Adapter Configuration dialog box and create a new VI using the LabVIEW Module tab, TestStand creates a new VI based on the code template for the specified step type. When the step type has multiple code templates available, TestStand launches the Choose Code Template dialog box, as shown in Figure 5-2, in which you can select the code template to use for the new VI.

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

Legacy VI Settings

Click the **Legacy VI Settings** button in the LabVIEW Adapter Configuration dialog box to launch the Legacy VI Settings dialog box, in which you can configure settings relevant to calling legacy test VIs. The Legacy VI Settings dialog box contains expressions the LabVIEW Adapter evaluates to generate values to pass to the VI in the various **Invocation Info** cluster fields. Legacy VIs can use the **Invocation Info** cluster as an optional input. Refer to the *Invocation Info Cluster* section of Chapter 10, *Calling Legacy LabVIEW VIs*, for more information about the **Invocation Info** cluster.

LabVIEW Project Settings

Enable the **Auto Deploy Shared Variables on First Load of any VI in a Project** option to direct TestStand to deploy to the local computer the shared variables defined in LabVIEW libraries within the LabVIEW projects you use in TestStand. The deployment of shared variables occurs the first time TestStand loads a LabVIEW step that references the project during execution. The LabVIEW library to deploy should contain only shared variables.



Note Enabling this option does not deploy shared variables defined in LabVIEW packed project libraries specified in LabVIEW projects you use in TestStand. Use the Deploy Library step type to deploy or undeploy to the local computer the shared variables defined in a LabVIEW packed project library file.



Note You must have LabVIEW 2009 SP1 f1 or later to use LabVIEW projects in TestStand.



Note You must have LabVIEW 2010 or later to use LabVIEW packed project libraries in TestStand.

When you enable the **Auto Undeploy Shared Variables on Last Unload of all VIs in a Project**, TestStand undeploys the shared variables defined in LabVIEW libraries within the LabVIEW project after TestStand unloads the last LabVIEW step that references the LabVIEW project. TestStand automatically enables this option when you enable the **Auto Deploy Shared Variables on First Load of any VI in a Project** option.



Note If you do not want TestStand to automatically deploy shared variables, leave the Auto Deploy Shared Variables on First Load of any VI in a Project option disabled. Refer to the *NI TestStand Help* and the *Deploying Variables* section of Chapter 7, *Effectively Using LabVIEW with TestStand*, for more information about deploying LabVIEW shared variables.

Creating Custom User Interfaces in LabVIEW

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

TestStand User Interface Controls

Use the TestStand User Interface (UI) Controls on the **Controls»TestStand** palette in LabVIEW to develop a custom user interface application, including custom sequence editors.

When you place TestStand UI Controls on the front panel of a VI, you can use the LabVIEW ActiveX functionality to program the controls. You can also configure the controls interactively using the LabVIEW ActiveX Property Browser or control property pages if available. Right-click the control and select **Property Browser** from the context menu to open the LabVIEW ActiveX Property Browser. Right-click the control and select **Properties** from the context menu to launch the property page.

Refer to Chapter 9, *Using the TestStand ActiveX APIs in LabVIEW*, for more information about programming the TestStand API from LabVIEW. Refer to the *NI TestStand Help* and to the *TestStand User Interface Controls Reference Poster* for more information about the TestStand UI Controls.

TestStand VIs and Functions

The TestStand VIs and functions on the **Functions»TestStand** palette in LabVIEW are the LabVIEW versions of the functions in the TestStand Utility (TSUtil) Functions Library.

Use the TestStand VIs and functions for the following tasks:

- Inserting menu items that automatically execute commands the TestStand UI Controls provide
- Localizing the strings in a user interface
- Making dialog boxes VIs launch modal to TestStand applications
- Determining whether an execution that calls a VI code module is terminating or aborting
- Allowing the calling execution thread to suspend when a VI code module performs a lengthy operation
- Setting and obtaining the values of TestStand properties and variables

Right-click the VI on the **Functions** palette or on the block diagram and select **Help** from the context menu to access the help for the VI.

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.

Example User Interfaces

Refer to the example user interfaces included with TestStand for more information about creating a user interface using the TestStand UI Controls in LabVIEW. Begin with the simple user interface example, `<TestStand Public>\UserInterfaces\Simple\LabVIEW\TestExec.llb\Simple OI - Top-Level VI.vi`. Refer to the full-featured example, `<TestStand Public>\UserInterfaces\Full-Featured\LabVIEW\TestExec.llb\Full UI - Top-Level VI.vi`, 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 Public>\UserInterfaces` and `<TestStand>\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.

Caveats for Using the Example User Interfaces

Consider the following issues when using the example LabVIEW user interfaces:

- The `Load Top-Level VI.vi`, located in the `<TestStand Public>\UserInterfaces\Full-Featured\LabVIEW` directory, launches the LabVIEW user interface in the context of a LabVIEW project.
- If you use the files located in the `<TestStand Public>\UserInterfaces\Full-Featured\LabVIEW` or `<TestStand Public>\UserInterfaces\`

Simple\LabVIEW directories as a starting point for creating a user interface, National Instruments recommends that you always run the user interface in the context of its LabVIEW project to avoid name collisions. If you expect the user interface to run outside the context of the LabVIEW project, you must modify the names of the VIs in the user interface to ensure the names will not collide with the names of code module VIs or its subVIs.

- A LabVIEW project library provides a namespace for the LabVIEW simple and full-featured user interface examples. If you load any VI from the user interface by name, you must update the code that loads the VI to include the namespace.
- When you run the `TestExec.exe` build specification to generate an executable, LabVIEW prefixes VIs listed under the Dependencies node in the LabVIEW project with `Simple UI -` or `Full UI -` for the simple and full-featured user interfaces, respectively, to avoid name collisions. If you load any VI from the user interface by name, you must update the code that loads the VI to include VI prefixes.

Complete one of the following options for all dynamic calls listed under the Dependencies node, including VIs that you call from top-level files but that you have not explicitly added to the My Computer target in the LabVIEW project, to resolve this issue:

- Add the `Simple UI -` or `Full UI -` prefix to the path you pass to the Open VI Reference function when the caller VI runs in the LabVIEW RTE. You can use the `Application:Kind` property in LabVIEW to determine whether the VI is running in the development system or the LabVIEW RTE.
- Add the VIs listed under the Dependencies node to the My Computer target to prevent the name change when you build the executable. LabVIEW prefixes only the files listed under the Dependencies node. National Instruments recommends that you prefix these VIs manually to avoid potential name collisions that might prevent the user interface from running VIs that have conflicting names.
- 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.



Note National Instruments recommends that when a TestStand User Interface performs ActiveX operations that can process messages or performs TestStand operations that can call back into LabVIEW, the application must perform these operations in a LabVIEW execution system other than **user interface**, such as **standard** or **other 2**. Performing these operations in the user interface execution system can result in hang conditions.

Configuring the TestStand UI Controls

Refer to the following example user interface VIs for examples of configuring connections, commands, and other settings for the TestStand UI Controls:

- Simple OI - Configure Application Manager
- Simple OI - Configure SequenceFileView Manager
- Simple OI - Configure ExecutionView Manager
- Full UI - Configure StatusBar
- Full UI - Configure SequenceFileView Manager
- Full UI - Configure ListBar
- Full UI - Configure ExecutionView Manager

Enabling Sequence Editing

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

Refer to the *NI TestStand Help* for more information about using Operator Mode and Editor Mode with the TestStand UI Controls.

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 LabVIEW, you register a callback VI, which LabVIEW automatically calls when the control generates the event.

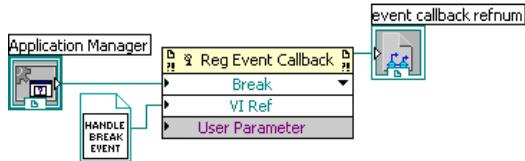
Complete the following steps to use the Register Event Callback function, available in the LabVIEW Full or Professional Development System.

1. Wire the reference of the control that sends the event you want to handle to the **Event** input of the Register Event Callback function.
2. Use the **Event** input terminal drop-down list to select the specific event you want to handle.
3. When you want to pass custom data to the callback VI, wire the custom data to the **User Parameter** input of the Register Event Callback function. The **User Parameter** input can be any data type.
4. Right-click the **VI Ref** input of the Register Event Callback function and select **Create Callback VI** from the context menu. LabVIEW creates an empty callback VI with the correct input parameters for the particular event, including an input parameter for any custom data you wired to the **User Parameter** input in step 3.
5. Save the new callback VI. The block diagram that contains the Register Event Callback function now shows a Static VI Reference node wired to the **VI Ref** input of the function. This node returns a strictly typed reference to the new callback VI.

6. Complete the block diagram of the callback VI to perform the operation you specify when the control generates the event.
7. When the application finishes handling events for the control, use the Unregister for Events function to close the **event callback refnum** output of the Register Event Callback function.

Figure 6-1 shows how to register a callback VI to handle the Break event for the Application Manager control.

Figure 6-1. Registering a Callback VI for the Break Event



You can resize the Register Event Callback function node to show multiple sets of terminals to handle multiple events. Refer to the Simple OI - Configure Event Callbacks and to the Full UI - Configure Event Callbacks example user interface VIs for examples of registering and handling events from the TestStand UI Controls.

You must limit the tasks you perform in a callback VI to ensure that LabVIEW handles the event in a timely manner to allow the front panel to quickly respond to user input and prevent possible hang conditions. When a callback VI performs ActiveX operations that can process messages or performs TestStand operations that can call back into LabVIEW, the application must perform these operations outside of the callback VI. You can define a user event the callback VI generates to defer these types of operations.

The ReDraw user event in the full example user interface shows how callback VIs can defer operations to perform outside of the callback VI. The example user interface performs the following tasks:

- Calls the Full UI - Create LabVIEW Application Events VI to create the ReDraw user event.
- Callback VIs, such as the Full UI - Resized Event Callback VI, generate the ReDraw user event when the user interface must resize and reposition controls on the front panel.
- The ReDraw User Event case in the main event loop of the Full UI - Top-Level VI sets a global variable while processing the current event to prevent callback VIs from generating new ReDraw events. The ReDraw User Event case calls the Full UI - Disable Panel Updates VI to prevent the front panel from updating, calls the Full UI - ArrangeControls VI to update the position and size of controls on the front panel, and calls the Full UI - Re-enable Panel Updates VI to update the front panel.

Starting and Shutting Down TestStand

Invoke the `ApplicationMgr.Start` method to start TestStand, as shown in the Simple OI - Top-Level VI and in the Full UI - Top-Level VI example user interfaces.

User interface applications wait in a main event loop after starting TestStand. The main event loop must handle the events that stop the user interface application. The main event loop can also handle other events, such as menu selections and LabVIEW control changes.

Typically, you click the **Close** box or execute the **Exit** command through a TestStand menu or a Button control to stop a user interface application.

When you click the **Close** box, the Event structure in the main event loop handles the Panel Close? event. The block diagram that handles the event invokes the `ApplicationMgr.Shutdown` method and discards the event. When the `ApplicationMgr.Shutdown` method returns `True` to indicate TestStand is ready to shut down, the main event loop stops. When the `ApplicationMgr.Shutdown` method returns `False`, the main event loop waits because TestStand cannot shut down until the executions complete or you unload sequence files. When TestStand is ready, the Application Manager control generates the `ApplicationMgr.ExitApplication` event.

The callback VI for the `ApplicationMgr.ExitApplication` event generates a LabVIEW Quit Application user event, which the example user interface VIs handle, to inform the main event loop to stop.

Refer to the following example user interface VIs for examples of the main event loop and how to shut down TestStand. These VIs also provide examples of creating, generating, and handling the Quit Application event.

- Simple OI - Top-Level VI
- Simple OI - ExitApplication Event Callback
- Full UI - Top-Level VI
- Full UI - Create LabVIEW Application Events
- Full UI - ExitApplication Event Callback

Menu Bars and Menu Event Handling

The TestStand VIs and functions palette contains the following VIs for creating and handling menu items that execute TestStand UI Control commands:

- TestStand - Insert Commands in Menu
- TestStand - Cleanup Menus
- TestStand - Remove Commands From Menus
- TestStand - Execute Menu Command

Because maintaining the current state of the menu bar can be difficult, National Instruments recommends that you handle the menu bar only when required. The Event structure in a main

event loop can include a case to handle the Menu Activation? event to determine when you open a menu or select a shortcut key that might be linked to a menu item. The block diagram that handles this event can then rebuild the menu bar.

The Full UI - Top-Level VI in the example user interface shows how to rebuild the menu bar. The Menu Activation? case in the main event loop of the VI determines which control has focus, if the control is a TestStand UI Control, and calls the Full UI - Rebuild Menu Bar VI to rebuild the menu bar. When you click the menu bar, LabVIEW does not automatically return focus to the control after handling a user menu event. The Menu Activation? case in the Full UI - Top-Level VI passes a reference for the control with focus to the Menu Selection (User) case so the application can later restore focus to the control.

You can add a Menu Selection (User) case to the Event structure in a main event loop to handle user menu selections but limit the tasks you perform in the Menu Selection (User) case to ensure that LabVIEW handles the menu selection in a timely manner. When the case performs ActiveX operations that can process messages or performs TestStand operations that can call back into LabVIEW, the application must perform these operations in a LabVIEW execution system other than **user interface**, such as **standard** or **other 2**.

The Full UI - Top-Level VI in the example user interface shows how to process user menu events. The VI uses the standard LabVIEW execution system. The Menu Selection (User) case in the main event loop calls the Full UI - Add To Menu Queue VI to queue the operation for processing the menu outside of the main event loop. The Full UI - Process Menu Queue VI waits for and processes queued operations. For TestStand menu items, the Full UI - Process Menu Queue VI calls the TestStand - Execute Menu Command VI to execute the appropriate TestStand command. For non-TestStand menu items, the VI calls the Full UI - Process User Menus VI, which you can customize to handle user menu selections.

The LabVIEW application menu items for copy, cut, and paste operate on LabVIEW controls only and do not operate on TestStand UI Controls. In addition, the TestStand menu commands operate on TestStand UI Controls only and not on LabVIEW controls. When you rebuild a LabVIEW menu in the Menu Activation? event case and you call the TestStand - Insert Commands in Menu VI to insert `CommandKind_Edit_Copy`, `CommandKind_Edit_Cut`, or `CommandKind_Edit_Paste`, pass `False` to the **TestStand UI Control Has Focus** control and "Edit" to the **Top-Level Menu to Insert Into** control to insert the corresponding LabVIEW application menu items instead of the TestStand menu command.

Localization

The TestStand UI Controls and TestStand VIs and functions provide tools that localize user interfaces based on the TestStand language setting. Use the following VIs to localize the user interface:

- TestStand - Get Resource String
- TestStand - Localize Menu
- TestStand - Localize Front Panel

Refer to the Full UI - Localize Operator Interface and to the Full UI - About Box example user interface VIs for examples of localizing user interfaces.

Refer to the *NI TestStand Help* for more information about localizing all the user-visible TestStand UI Controls strings.

Other User Interface Utilities

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

Making Dialog Boxes Modal to TestStand

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

Use the TestStand - Start Modal Dialog and the TestStand - End Modal Dialog VIs, located on the TestStand palette, to make a dialog box modal to TestStand application windows. Refer to the `<TestStand Public>\Examples\ModalDialogs\LabVIEW` directory for examples of how to use these VIs.

Checking for Suspended or Stopped Executions within Code Modules

The VIs that TestStand calls can launch dialog boxes or perform other time-consuming operations. In these cases, it can be useful for those VIs to periodically check whether TestStand is terminating or aborting their parent execution so the VIs can stop gracefully and allow the parent execution to terminate or abort.

Code modules can also allow TestStand to suspend the execution thread at a breakpoint without requiring the code modules to first return to TestStand.

Use the following VIs to enable VIs that TestStand calls to determine whether the execution receives a request to terminate or abort the execution:

- TestStand - Initialize Termination Monitor
- TestStand - Get Termination Monitor Status
- TestStand - Close Termination Monitor

The Termination Monitor only recognizes requests to terminate or abort while monitoring, so a code module that executes in a cleanup step group of an already terminating execution monitors for a subsequent request to terminate the step or abort the execution.

Refer to the VIs in the following directories for examples of how to use these VIs:

- `<TestStand Public>\Examples\Demo\LabVIEW\Computer Motherboard Test`
- `<TestStand Public>\Examples\Demo\LabVIEW\Auto`

Use the TestStand – Set Thread Externally Suspended VI to allow TestStand to suspend the execution at a breakpoint while still executing code in the code module.

Use the `Execution.GetStates` method to determine whether the execution is suspended.

Running User Interfaces

Consider the following guidelines when running user interfaces:

- You must close all running LabVIEW user interfaces before you exit Microsoft Windows. When you shut down, restart, or log off of Windows while a user interface is running, the user interface cancels the operation and might exit with an error.
- When you run a LabVIEW user interface in the LabVIEW development system, you can call remote VIs only when the VI is the same or earlier version as the LabVIEW development system. TestStand does not support calling VIs on remote computers when you save a VI without the block diagram and the version of the VI is different than the active version of LabVIEW installed on the computer.
- By default, you can run only one copy of a LabVIEW-built executable at a time, which prevents the TestStand `/useexisting` command-line option from working. Add the `"allowmultipleinstances = TRUE"` option to the INI options file in the same directory as the LabVIEW built executable to allow more than one copy to execute at a time.
- A LabVIEW user interface reports object leaks on shutdown when you pass a TestStand object as the **ActiveXData** parameter of a `UIMessage` in the `UIMessage` callback VI. LabVIEW does not release the object reference until LabVIEW unloads the callback VI. Refer to the National Instruments website at ni.com/info and enter the Info Code 4H6DULT3 to access the National Instruments KnowledgeBase article *PropertyObjects Were Not Released Warning When Using LabVIEW* for more information about using a dynamically called VI to ensure that LabVIEW releases the object.
- To prevent a LabVIEW user interface from hanging while running in LabVIEW, do not call subroutine VIs from inside a VI used as an ActiveX callback VI. Instead, change the execution priority of the VIs to a value other than subroutine.

Effectively Using LabVIEW with TestStand

Use the advanced features of LabVIEW to organize and deploy TestStand VIs. These LabVIEW features can impact the performance, robustness, and deployment capabilities of a test system. Consider the requirements and organization of the test system when using these features.

Organizing LabVIEW-Based Systems

A LabVIEW-based TestStand system can utilize TestStand workspaces, sequence files, and sequences to organize and represent the architecture of the application in LabVIEW. TestStand sequences contain steps, which can call LabVIEW VI code modules. Depending on the architecture of the test system, these code module VIs can call utility VIs, which can in turn utilize VIs stored in `vi.lib` or other LabVIEW libraries.

LabVIEW provides several organizational options for the LabVIEW VIs you create and use in a test system. TestStand supports the following options:

- LabVIEW projects (`.lvproj`)
- LabVIEW packed project libraries (`.lvlibp`)
- Stand-alone LabVIEW VIs (`.vi`)
- LabVIEW libraries (`.lib`)
- LabVIEW project libraries (`.lvlib`)
- LabVIEW-built shared libraries (`.dll`)



Note National Instruments recommends using a single version of LabVIEW to compile all the VIs for a test system.



Note TestStand support for LabVIEW packed project libraries and auto-populating folders in LabVIEW projects is available only in LabVIEW 2010 or later.



Note You must have LabVIEW 2009 SP1 f1 or later to use LabVIEW projects in TestStand.

Consider the test system requirements when choosing an organizational approach for LabVIEW code module VIs and utility VIs. The approaches vary in impact on the maintenance, performance, robustness, and deployment of the test system.

The following sections discuss how to organize a TestStand system using LabVIEW projects, LabVIEW packed project libraries, stand-alone LabVIEW VIs, LabVIEW libraries, LabVIEW project libraries, and LabVIEW-built shared libraries.



Note National Instruments does not recommend using LabVIEW-generated .NET assemblies to develop a TestStand system.

LabVIEW Projects

LabVIEW projects are files that reference VIs, files necessary for those VIs to run properly, and supplemental files such as documentation or related links. You can use folders and libraries to group items in a hierarchy.

LabVIEW projects provide a system-level view of the files an application requires. The LabVIEW Project Explorer helps developers easily find and organize files from within the development environment and provides functionality that addresses the challenges of managing and developing large LabVIEW applications.

Use LabVIEW projects to group together LabVIEW files and other files and to integrate with source code control providers. You can also use LabVIEW projects to prevent cross-linking by qualifying common VI names using project libraries for name spacing and to help you with large application development.

Refer to the *LabVIEW Help* for more information about LabVIEW projects.

Editing VIs in LabVIEW Projects

Consider the following recommendations and requirements for editing VIs in LabVIEW projects when deciding whether to use LabVIEW projects in a TestStand system:

- TestStand supports VIs organized in folders, LabVIEW project libraries, LabVIEW libraries, or LabVIEW packed project libraries defined in the LabVIEW project under the *My Computer* target. LabVIEW creates an application instance for each target in a LabVIEW project, including *My Computer*.
- VIs you call from TestStand in the context of a LabVIEW project cannot use global variables to share data with VIs outside of that project because when you open a VI specified in a LabVIEW project, the VI opens in the application instance for the project. LabVIEW also creates a main application instance that contains open VIs not included in a project and VIs you did not open from a project. Additionally, LabVIEW loads shared libraries in a unique application instance, which prevents naming conflicts with the VIs in the shared library or with VIs outside of the shared library.
- National Instruments recommends using shared variables to transfer data between VIs executing in more than one project.



Note TestStand supports automatically deploying and undeploying shared variables defined in LabVIEW projects. Refer to the *LabVIEW Project Settings* section of Chapter 5, *Configuring the LabVIEW Adapter*, and to the

NI TestStand Help for more information about using shared variables with LabVIEW projects.

Configure TestStand steps to execute code modules specified in a LabVIEW project if the test system meets any of the following conditions:

- The TestStand system includes multiple VIs with the same name
- Code module VIs use LabVIEW conditional disable structures that depend on custom symbols defined in a LabVIEW project
- Code module VIs use NI-DAQmx tasks, channels, and scales defined in a LabVIEW project
- LabVIEW FPGA Host VIs that use LabVIEW FPGA targets are defined under the `My Computer` target in a LabVIEW project

Updating and Patching VIs in LabVIEW Projects

Consider the following upgrade and patching recommendation when deciding whether to use LabVIEW projects in a TestStand system:

- National Instruments recommends using LabVIEW packed project libraries to modularize a TestStand system. If you do not use LabVIEW packed project libraries and you want to change VIs in a TestStand system, you must re-deploy all VIs in the system. Failing to do so can cause run-time errors when LabVIEW recompiles VIs to account for inplaceness changes.

Deploying VIs in LabVIEW Projects

Consider the following deployment recommendations and requirements when deciding whether to use LabVIEW projects in a TestStand system:

- The size of a deployment affects its size on disk and the time required to install. If the test system contains multiple LabVIEW projects, the size of the deployed image is larger and takes longer to build. To reduce the time required to create a deployable build featuring multiple LabVIEW projects, enable the **Consolidate Files Shared by Projects** option in LabVIEW VI Options dialog box of the TestStand Deployment Utility to create a merged project for use in a deployable image.
- TestStand does not support deploying project libraries with the same name and different file paths. You receive an error when you attempt to include two project libraries with the same name.

Refer to the *NI TestStand Help* for more information about using the TestStand Deployment Utility.

Refer to Chapter 2, *Calling LabVIEW VIs from TestStand*, and Chapter 3, *Creating, Editing, and Debugging LabVIEW VIs from TestStand*, for more information about calling VIs in LabVIEW Projects.

LabVIEW Packed Project Libraries

LabVIEW packed project libraries are LabVIEW project libraries that package multiple LabVIEW-related files and dependencies into a single non-editable file (.lvlibp) that you can call from other VIs. Refer to the *LabVIEW Help* for more information about LabVIEW packed project libraries.

In TestStand, you can directly open and call VIs exported from a LabVIEW packed project library, or you can access the VIs through a LabVIEW project.



Note You must have LabVIEW 2010 or later to use LabVIEW packed project libraries with TestStand.

Editing VIs in LabVIEW Packed Project Libraries

Consider the following recommendations and requirements for editing VIs when deciding whether to use LabVIEW packed project libraries in a TestStand system:

- National Instruments recommends enabling the **Callers adapt at run time to Exported VI connector pane state** option on the Connector Pane State page of the Packed Library Properties dialog box in the packed project library build specification in LabVIEW to allow other LabVIEW VIs to call exported VIs stored in the LabVIEW packed project library in order to adapt to inplaceness changes in the exported VI without recompiling.
- National Instruments recommends creating an architecture with multiple packed project libraries that group similar sets of VIs together so that you can update code more easily.
- You cannot edit VIs inside packed project libraries. You must have the source VIs and the project that contains the build specification for the packed project library to make a change to the packed project library. To modify a VI stored in a packed project library, you must modify the source VI and rebuild the packed project library, and replace the older packed project library with the newer version.
- National Instruments recommends that you wait until you have finished making major changes to VIs before building the packed project library. If VIs that use the packed project library are in the same project, you can replace all instances of VIs in the project library with a call to a built packed project library in the LabVIEW Project Explorer window.
- You cannot load a VI stored in a packed project library independently of other items in the same packed project library. LabVIEW loads every VI in the packed project library when you call the first VI, resulting in a longer initial load time and higher memory usage. LabVIEW loads subsequent VIs immediately.
- When you call the first VI in a packed project library, LabVIEW consolidates types in all VIs in the packed project library to reduce the time required to load every VI inside the packed project library. This behavior is faster than loading the same VIs if they were stored outside of the packed project library. Packed project libraries are also faster to load because they consist of a single file, so the test system does not have to search the disk for additional files.
- Packed project libraries have an MSI-recognized version resource to help you more easily make partial updates to the test system.

- Packed project libraries support debugging when built with debugging information. However, including the debugging information makes the packed project library larger and slightly slower to execute. Consider benchmarking the differences in size and execution speed to determine the impact of including the debugging information in the packed project library.

Updating and Patching VIs in LabVIEW Packed Project Libraries

Consider the following upgrade and patching recommendation when deciding whether to use LabVIEW packed project libraries in a TestStand system:

- Packed project libraries are designed to make patching easier. You can replace a packed project library with a newer version without changing the VI or TestStand steps that call into the packed project library.

Deploying VIs in LabVIEW Packed Project Libraries

Consider the following deployment recommendations and requirements when deciding whether to use LabVIEW packed project libraries in a TestStand system:

- Packed project libraries reduce the number of files required to create a deployment using the TestStand Deployment Utility. Because packed project libraries are precompiled files, the total build time is also reduced.
- Packed project libraries are compressed files. If you are deploying a prebuilt packed project library, the deployment utility copies only the packed project library. If you configure the deployment utility to create a packed project library, it must create the packed project library on every build. You must determine whether the test system will benefit more from the faster build times the prebuilt packed project libraries offer or the convenience of using the deployment utility to build the packed project library. The deployment utility also supports building the packed project library with debugging information to support debugging of the packed project library on the deployed system.

Refer to the *NI TestStand Help* for information about using the Packed Project Library Options dialog box to configure options for how the deployment utility builds LabVIEW packed project libraries.



Note Use the Deploy Library step type to deploy or undeploy to the local computer the shared variables defined in a LabVIEW packed project library file or to deploy or undeploy shared variables defined in LabVIEW packed project libraries within a selected LabVIEW project.



Note TestStand can deploy only shared variables defined in a LabVIEW packed project library file using the Deploy Library step type if the packed project library file defines only shared variables and does not contain any VI files. TestStand cannot auto-deploy shared variables defined in a LabVIEW packed project library specified in a LabVIEW project or in a LabVIEW project library saved in a LLB specified in a LabVIEW project.

Refer to Chapter 2, *Calling LabVIEW VIs from TestStand*, and Chapter 3, *Creating, Editing, and Debugging LabVIEW VIs from TestStand*, for more examples of calling VIs from packed project libraries. Refer to the *NI TestStand Help* for more information about the deployment utility and the Deploy Library step type.

Stand-alone VIs

Code module VIs use functions that manipulate input from the user interface or other sources and display that information or move it to other files or computers. A stand-alone VI is a file on disk that is not part of a LabVIEW library and is not called in the context of a project.

Editing Stand-alone VIs

Consider the following editing recommendations and requirements when deciding whether to use stand-alone VIs in a TestStand system:

- Loading multiple stand-alone VIs can negatively impact performance, especially if the test sequences load VIs dynamically.
- Using multiple stand-alone VIs can result in name conflicts. LabVIEW does not support loading multiple VIs with the same name from different locations. Use LabVIEW project libraries and packed project libraries to add a namespace to the VIs and avoid name conflicts.
- National Instruments does not recommend editing packaged VIs on a deployed system because unexpected errors, such as broken VIs or run-time errors, can occur. National Instruments recommends rebuilding and redeploying the deployment image in this situation. Analysis and instrument drivers are examples of VIs that use project libraries.

Updating and Patching Stand-alone VIs

Consider the following upgrade and patching recommendations and requirements when deciding whether to use stand-alone VIs in a TestStand system:

- Inplaceness allows LabVIEW to reuse memory a caller VI uses in a subVI, thus reducing the memory requirements of the test system. Due to inplaceness changes, a change to a subVI can impact all VIs calling the subVI even if the connector pane of the subVI remains constant, therefore forcing all calling VIs to recompile.



Note National Instruments recommends using LabVIEW packed project libraries to modularize a TestStand system. If you do not use LabVIEW packed project libraries, and you want to change VIs in a TestStand system, you must re-deploy all VIs in the system. Failing to do so can cause run-time errors when LabVIEW recompiles VIs to account for inplaceness changes.

- Stand-alone LabVIEW VIs do not have version resources. If a file does not have a version resource, installers use a heuristic process to determine whether to upgrade the file. These heuristics might cause confusion when VIs are not upgraded as you expected, such as a test system in which multiple installers update a single set of utility VIs.

- National Instruments recommends that you save all code modules in one version of LabVIEW. VIs saved in different versions of LabVIEW cannot share data when executed using the LabVIEW Run-Time Engine (RTE).

Deploying Stand-alone VIs

Consider the following deployment recommendations and requirements when deciding whether to use stand-alone VIs in a TestStand system:

- National Instruments does not recommend distributing two VIs with the same name to different locations on a target computer. When a VI with the same name from a different location is in memory, LabVIEW uses the VI in memory when attempting to load a sub VI. However, LabVIEW reports an error when you attempt to load a top-level VI with the same name as a VI in memory even when the similarly named VI in memory is an exact copy of the one you want to load.
- TestStand does not support including two VIs with the same name in a deployment unless the VIs are in different project libraries.
- When you build a TestStand deployment that calls VIs that use shared variables, you must complete the following tasks:
 - Add project library files that contain shared variables to the workspace you use to create the deployment. While processing sequence files, the deployment utility automatically includes project library files in the deployment when a LabVIEW Utility Deploy Library step references the project library.
 - Add an aliases file to the deployment and configure the deployment to install the aliases file to the proper location on the destination system so LabVIEW can resolve network paths. Refer to the *Using an Aliases File* section of this chapter for more information about the correct location for the files.
 - Include the NI Variable Engine component in the installer to ensure that the destination system can use the shared variables.
- You must execute a LabVIEW Utility Deploy Library step in a TestStand sequence to deploy shared variables to the local computer. You can also manually deploy shared variables to the local computer using a project in the LabVIEW development environment. Refer to the *NI TestStand Help* for more information about the Deploy Library step type.

Refer to Chapter 2, *Calling LabVIEW VIs from TestStand*, and Chapter 3, *Creating, Editing, and Debugging LabVIEW VIs from TestStand*, for more information about calling stand-alone VIs. Refer to the *NI TestStand Help* for more information about the deployment utility.

LabVIEW Libraries

LabVIEW libraries (LLBs) are files that group several VIs together in a `.llb` file. LLBs reduce the number of files on disk, making it easier to deploy a test system.

Consider the following when working with LLBs:

- An installer cannot install a single VI to an LLB. You must overwrite the entire LLB to install additional files.
- When you make changes to an LLB, it is difficult to determine which files have changed.
- LLBs do not have hierarchy information, so all VIs are stored at the same level. However, for display purposes, you can mark VIs called from TestStand as top-level VIs in the LLB.
- LLBs do not have namespaces and cannot prevent name collisions.
- All VIs in an LLB are visible, so you cannot hide the names of private VIs and subVIs.
- LLBs do not have version resources, so multiple installers upgrading a single LLB might have unintended consequences.

If you want to build LLBs of test VIs when creating a deployment of a test system, you must build the LLB and update the TestStand sequences that call test VIs to reference the LLB because the deployment utility does not create LLBs of top-level VIs. However, if the test VIs are already organized in LLBs, the deployment utility maintains the LLB structure.

National Instruments recommends that you save VIs as individual files organized in directories instead of using LLBs, especially if multiple developers are working on the same project.

LabVIEW Project Libraries

LabVIEW project libraries are collections of VIs, type definitions, shared variables, palette menu files, and other files, including other project libraries. In TestStand, you can call public VIs in a project library, but you cannot call private VIs. To call a VI in a project library, you must specify the path to the VI file. TestStand does not use qualified paths that include the project library path and name.

LabVIEW-Built Shared Libraries

You can use the LabVIEW Application Builder to build a shared library, or DLL. TestStand can execute exported VIs in LabVIEW DLLs built using different versions of LabVIEW. Use the C/C++ DLL Adapter to configure a step to call one of the exported functions in the DLL. Refer to the *NI TestStand Help* for more information about the C/C++ DLL Adapter.

To create a DLL in LabVIEW, you must first create a LabVIEW project that refers to the VIs you want to export and then create a build specification. You must configure the type of each parameter for all exported VIs. Refer to LabVIEW documentation for more information about debugging LabVIEW DLLs.



Note LabVIEW does not support exporting every LabVIEW data type. You might have difficulty passing data to certain LabVIEW types from TestStand. You must also manually add parameter information when configuring the LabVIEW-generated DLL in TestStand.



Note You must install the version of the LabVIEW RTE that corresponds to the version of LabVIEW used to create the DLL on any computer that uses the DLL. You can include the appropriate RTE installer in a deployment utility-built installer.



Note TestStand supports—but National Instruments does not recommend—distributing DLLs with the same name that steps in a sequence file call.

Editing VIs in LabVIEW-Built Shared Libraries

To ensure that windows messages are handled during the execution of a LabVIEW DLL you build for use with TestStand, National Instruments recommends that you disable the **Delay operating system messages in shared library** option on the Advanced Page of the Shared Library Properties dialog box in the LabVIEW Professional Development System.

Special Considerations When Using LabVIEW with TestStand Systems

Consider the following aspects of a TestStand system when using LabVIEW.

Using Relative Paths

Use relative paths to specify code modules because you typically organize code module VIs in a folder relative to the TestStand sequence file and TestStand steps. Absolute paths can cause errors in deployment and multiple-developer scenarios because directory structures frequently vary among systems. Absolute paths also increase the maintenance burden if you want to move files within the TestStand system.

Memory Management for Large Data Sets

LabVIEW automatically handles memory allocation. Because this process is automatic, LabVIEW copies data frequently. If the test system involves large sets of data, large and frequent data copies might result in an out-of-memory error. Refer to the *NI LabVIEW Help* for more information about handling large data sets.

Reusing VIs

If a user interface calls a VI that will also be called as a code module, you must ensure that the name of the VI called from the user interface does not match the name of the VI when it is called as a code module. Apply a prefix to all dependent VIs when building the user interface into a stand-alone application to avoid a naming conflict among the dependencies of the user interface executable and the code module VIs called from test sequences.



Note When you apply a prefix to a user interface-dependent VI, the user interface does not share the same instance of the VI as the VI when it is called as a code module. Use shared variables or LabVIEW queues to share information between user interface and code module VIs.

Reserving Loaded VIs for Execution

Enable the **Reserve Loaded VIs for Execution** option in the LabVIEW Adapter Configuration dialog box to reserve any VIs TestStand loads for calling with the LabVIEW Adapter. Use this option to avoid reloading VIs when they are used and to make references you create in a VI you call from TestStand—such as I/O, ActiveX, and synchronization references—persist across calls to other VIs. You can store these references in a TestStand property and pass them to subsequent VIs you call from TestStand.

When you open a reserved VI in LabVIEW, the Run arrow, as shown in the following figure, indicates the VI is reserved, and you cannot edit the VI. To edit a VI TestStand has reserved, click the **Edit VI** button on the LabVIEW Module tab or right-click the step and select **Edit Code** from the context menu to open the VI in TestStand. You can also select **File»Unload All Modules** in the TestStand Sequence Editor before you open the VI in LabVIEW.



You must close any references you create to VIs. When TestStand loads and reserves VIs, LabVIEW does not automatically close the references until TestStand unloads the VIs that created the references. Failing to close the references might result in a memory leak in the test system.

Partially Specifying LabVIEW Clusters

If you have a cluster with default values and you need to modify certain values on the cluster depending on an operation you want to perform, you can use partial clusters. In TestStand, you can specify whether to pass default values for individual elements of a cluster control that are recommended or optional. This behavior is referred to as partially specifying a cluster, or partial cluster passing. To specify that you want to apply the default value to an element of a partial cluster, enable the **Default** option on the Module tab of the LabVIEW Adapter Configuration dialog box or in the Specify Module dialog box.

Executing VIs with Separate Compiled Code

In LabVIEW 2010, you can separate compiled code from a VI. When you open a VI with separate compiled code, LabVIEW generates and saves a VI object file (.viobj) with the compiled code. When you separate compiled code from a VI, recompiling the code of the VI does not create an unsaved change. As a result, if you use a source code control system, you do not have to check out all the files in a hierarchy when you edit a VI.

TestStand supports calling these VIs from LabVIEW. During deployment, the TestStand Deployment Utility converts these VIs with separated compiled code to regular VIs, allowing TestStand to execute these VIs using the LabVIEW RTE on the deployed machine.

Conditional Disable Structures and Symbols

The Conditional Disable Structure contains one or more subdiagrams, or cases. You can specify conditions for the structure based on custom symbols defined in a project. For all VIs in the main application instance, TestStand evaluates any conditions that use custom symbols when the VI is recompiled. These expressions are not evaluated again during execution.

XControls

TestStand supports calling VIs that use XControls on Microsoft Windows systems.

Building a TestStand Deployment with LabVIEW

TestStand supports deploying VIs from project libraries stored in LabVIEW 8.5.1 or later. However, the following restrictions exist that do not apply to earlier versions of LabVIEW:

- National Instruments does not recommend deploying VIs that were previously deployed using the TestStand Deployment Utility. When the deployment utility packages VIs, the utility includes all subVIs and creates partial project libraries that contain only the required VIs from the project libraries.

When you attempt to redeploy VIs, the build in the deployment utility can fail when you attempt to include a partial project library and the original complete project library or when you attempt to include two copies of the same VI on the system. When you deploy custom LabVIEW-based step types to a development system and then attempt to include the step types in a new deployment, the build might fail.

To work around this limitation, you must remove the duplicate VIs and partial project libraries from the test system and relink the VIs to the original VIs and complete project libraries on the system before you build a new deployment.

Complete the following steps to resolve any duplicate VIs from previously deployed files.

1. Build the deployment and review the VIs the utility reports as duplicates in the status log.
2. Review the SupportVIs LLB or directory in the previously deployed files to determine which VIs and project libraries conflict with VIs located on the development system. Typically, a SupportVIs LLB or directory contains duplicate VIs and project libraries from `vi.lib` and `user.lib`.
3. Remove any duplicate VIs and partial project libraries the deployment utility reports and duplicate VIs in a SupportVIs LLB or directory.
4. Load all top-level VIs included in the previously deployed files and resave the VIs. LabVIEW prompts you to browse for any missing subVIs.

5. When the previously deployed files contain sequence files with steps that call Express VIs or any duplicate VI, you must reconfigure the Express VI steps and update the VI pathname for steps that call the VIs.

Select **Tools»Update VI Calls** to run the Update VI Calls tool to update the Express VIs a LabVIEW step instance calls and to check or update a standard VI call prototype. Use this tool when you want to run the Express VIs a LabVIEW step instance calls in a later version of the LabVIEW Run-Time Engine. Also, use the Update VI Calls tool to update existing Express VI instances with any changes made to the Express VI. Refer to the *NI TestStand Help* for more information about the Update VI Calls tool.

6. Repeat step 1 to determine if duplicate VIs still exist.

Creating Custom Step Types that Call VIs for Substeps

When you create a custom step type that defines a Pre-Step, Post-Step, or Edit substep that calls a VI created using LabVIEW 2009 or earlier, you must create a source distribution to ensure that TestStand can resolve the file paths to all the dependencies of the VI when TestStand executes the custom step type.

If you created the VI using LabVIEW 2010 or later, you must create a LabVIEW packed project library to add all the dependencies of the VI to a single location.

When you create a source distribution or packed project library, National Instruments recommends that you enable the Always Run VI in LabVIEW Run-Time option in the LabVIEW Advanced Settings window to ensure that the VIs execute in the correct version of the LabVIEW RTE.

If you do not create a source distribution or packed project library, LabVIEW browses its search directories to resolve the file paths to all the dependencies of the VI, which might break the VI.

Refer to the *NI TestStand Help* for more information about substeps and TestStand search directories.

Using LabVIEW Classes with TestStand

Select the **Class Member Call** option from the Call Type ring control on the LabVIEW module tab in the sequence editor or on the Module tab of the Edit LabVIEW VI Call dialog box in a TestStand User Interface to call a member VI defined in a LabVIEW class, as shown in Table 7-1.

Table 7-1. Calling Member VIs Defined in a LabVIEW Class

LabVIEW Version	TestStand Behavior
LabVIEW 2009 SP1 or later	TestStand supports calling static member VIs defined in a LabVIEW class. TestStand statically calls member VIs for which dynamic dispatching is enabled.
LabVIEW 2012 or later	TestStand supports dynamically dispatching a member VI from a LabVIEW class.

You can call only public members, methods, and properties from LabVIEW classes, but cannot execute a member call remotely.

Refer to the [Creating New Steps That Call LabVIEW Class Member VIs](#) and the [Save the sequence file](#) sections of Chapter 2, [Calling LabVIEW VIs from TestStand](#), for a tutorial on how to configure LabVIEW steps to call member VIs from LabVIEW classes.



Note TestStand does not support dynamic dispatching when you select the **VI Call** option from the Call Type ring control.



Note You must have LabVIEW 2012 or later to use LabVIEW dynamic dispatching when calling LabVIEW classes in TestStand.

TestStand supports passing an instance of a LabVIEW Class data type, also called a LabVIEW class object, wired to and from the connector pane of a member VI when you use LabVIEW 2009 SP1 or later. You can store LabVIEW class objects in TestStand object reference variables.

The lifetime of the LabVIEW class object is the lifetime of the TestStand object reference variable that holds the LabVIEW class object. If the VI that creates the LabVIEW class object is running in the context of a LabVIEW project, the lifetime of the LabVIEW class object is the lifetime of the LabVIEW project. When the object reference is released either by going out-of-scope or by obtaining a new value, the LabVIEW class object is also released. This behavior is different than other LabVIEW reference data types, such as an I/O reference.

The VI that creates the reference determines the lifetime of LabVIEW reference data types stored in TestStand. As long as the VI that created the LabVIEW reference is reserved, the LabVIEW reference is valid. This is not the case for LabVIEW class objects. A LabVIEW class

object is still valid even after the VI that originally created it is unreserved or closed. If LabVIEW is restarted, trying to use a LabVIEW class object a previous LabVIEW instance obtained results in an error because the LabVIEW class object is not valid in the new instance of LabVIEW. In addition, if you try to use the LabVIEW class object after you close the LabVIEW project context in which the LabVIEW class object exists, an error occurs because closing a LabVIEW project context releases all LabVIEW class objects created in the project context.

When you select the **VI Call** option from the Call Type ring control, you are responsible for calling the appropriate member VI for the LabVIEW class object stored in TestStand.

Because you can pass any instance of a LabVIEW class object to any member VI from TestStand, no guarantee exists that the LabVIEW class object you passed to the member VI is compatible with the class object the member VI expects. To help LabVIEW check that the class object you passed from TestStand to the member VI is compatible with the class object the member VI expects, add the Preserve Run-Time Class function to the member VI. The Preserve Run-Time Class function returns an error if two class objects are incompatible. You can work around this limitation by creating top-level wrapper VIs that call dynamic dispatch member VIs of LabVIEW class objects when you select the **VI Call** option from the Call Type ring control to configure LabVIEW steps.



Note LabVIEW 2012 or later returns an error if you call a dynamically dispatched member VI that passes a derived class on the dynamic dispatch input terminals. For the dynamic dispatch to work properly, you must select the **Class Member Call** option from the Call Type ring control when you configure the step.

Network-Published Shared Variables

Use network-published shared variables so VIs on distributed systems can share data across the network. LabVIEW identifies shared variables through a network path that includes the computer name (target name), the project library name(s), and the shared variable name.

Deploying Variables

You must deploy a project library that contains the shared variables to which you want to connect from within a VI. LabVIEW automatically deploys shared variables from a library when you execute a VI that reads or writes shared variables and the project library that contains the shared variables is open in the current LabVIEW project.

When executing VIs without a project, TestStand cannot automatically deploy shared variables. Therefore, you must deploy shared variables to the local computer in one of the following ways:

- Manually deploy the shared variables in the LabVIEW development environment using a project.
- Use the Deploy Library step type to deploy or undeploy to the local computer the shared variables defined in a LabVIEW packed project library file or to deploy or undeploy shared

variables defined in LabVIEW packed project libraries within a selected LabVIEW project. The project library can contain only shared variables and cannot contain any VI files. Refer to the *NI TestStand Help* for more information about the LabVIEW Utility Deploy Library step type.

When executing VIs specified in a LabVIEW project, TestStand supports automatically deploying and undeploying shared variables defined in the LabVIEW project when the project is first loaded for execution.



Note TestStand does not support calling a VI or DLL that programmatically deploys shared variables using the LabVIEW Deploy Library method on a LabVIEW Application reference or using the LabVIEW Deploy Items method on a LabVIEW Project reference. When you programmatically deploy shared variables, the TestStand or LabVIEW application might return an error and terminate. You can use the Deploy Library method from within a stand-alone executable without adverse effects on TestStand.



Note TestStand can deploy a shared variable bound to another shared variable only when you use an NI Publish-Subscribe Protocol (NI-PSP) URL to bind the shared variable to deploy to another shared variable. If you attempt to use a Deploy Library step to deploy a shared variable to a variable in a LabVIEW project, the deployment fails. Refer to the *LabVIEW Help* for more information about deploying shared variables and NI-PSP URLs.

Using an Aliases File

LabVIEW VIs can use shared variables deployed on remote targets. National Instruments recommends specifying these VIs using a LabVIEW project, which also has the remote target defined, and executing these VIs using a LabVIEW project from TestStand.

When you deploy a project library or execute these VIs in TestStand on a remote target, LabVIEW must determine how to resolve the target name stored in the network path for the variable. When you configure a target in a LabVIEW project, LabVIEW stores the IP address for the target in an aliases file located in the same directory as the project. The aliases file uses the same base name as the project and a `.aliases` file extension.

When executing VIs in the main application instance, the server you configured the LabVIEW Adapter to use in the LabVIEW Adapter Configuration dialog box determines the aliases file LabVIEW uses in the following ways:

- **LabVIEW Run-Time Engine**—LabVIEW looks for an aliases file with the same name and location as the TestStand application, such as `SeqEdit.aliases` for the sequence editor and `TestExec.aliases` for a user interface. You must quit and restart the TestStand application after you copy the aliases file from the project directory to the application directory.

- **Development System**—LabVIEW looks for an aliases file, `LabVIEW.aliases`, located in the same directory as the LabVIEW development system executable. You must quit and restart LabVIEW after you copy the aliases file from the project directory to the LabVIEW directory.
- **Other Executable**—LabVIEW looks for an aliases file with the same name and location as the LabVIEW executable server. For example, `TestStandLVRTS.aliases` for `TestStandLVRTS.exe` and `TestExec.aliases` for a user interface. You must quit and restart the executable after you copy the aliases file from the project directory to the executable directory.

When executing VIs specified in a LabVIEW project, LabVIEW looks for an alias file with the same name and location as the specified LabVIEW project.



Note Ensure the alias file is not read-only to allow LabVIEW to update the file, if necessary, on the deployed machine.

LabVIEW Utility Step Types

In TestStand, you can use the following LabVIEW Utility step types to simplify running a VI on a LabVIEW remote system and to deploy or undeploy shared variables:

- **Check Remote System Status**—Determines whether LabVIEW is running on a remote computer and whether TestStand can connect to the remote computer.
- **Run VI Asynchronously**—Runs a VI in a new thread in the TestStand execution.
- **Deploy Library**—Deploys or undeploys to the local computer the shared variables defined in a LabVIEW project library file or packed project library file, or deploys or undeploys shared variables defined in LabVIEW project libraries within a selected LabVIEW project.

Refer to the *NI TestStand Help* for more information about the LabVIEW Utility step types.

Symmetric Multiprocessing in VIs Executed from TestStand

LabVIEW can use multiple execution threads to execute VIs called from TestStand. Executing VIs with multiple threads can improve performance on multi-core and multi-processor systems, particularly when the VIs contain complex data flows, such as parallel loops, branched wires, or asynchronous communication with hardware devices.



Note By default, in scenarios that do not include TestStand, LabVIEW uses multiple threads to execute VIs if the preferred execution system of the VIs is not set to **user interface**.

When TestStand calls a VI, the number of execution threads LabVIEW uses to run the VI depends on the following factors:

- Whether you enable the Execute ‘Same as caller’ VIs Using Multiple Threads option in the LabVIEW Adapter Configuration dialog box.
- The setting of the Preferred Execution System option on the Execution page of the VI Properties dialog box in LabVIEW
- The version of LabVIEW that executes the VI

Table 7-2 shows how these factors apply when you execute VIs using the LabVIEW 2009 or later development system or RTE. Table 7-3 shows how these factors apply when you execute VIs using the LabVIEW 8.6.1 or earlier development system or RTE.



Note LabVIEW always uses a single thread to execute VIs for which the preferred execution system is **user interface**. National Instruments strongly discourages using the **user interface** preferred execution system for VIs you call from TestStand. The remainder of the information in this section excludes consideration of the **user interface** preferred execution system.

Table 7-2. Number of Execution Threads when Using LabVIEW 2009 or Later

LabVIEW Setting: Preferred Execution System	TestStand Option: Execute ‘Same as caller’ VIs Using Multiple Threads	
	Enabled	Disabled
Same as caller	Multiple threads	Single thread
Not Same as caller	Multiple threads	Multiple threads

Table 7-3. Number of Execution Threads when Using LabVIEW 8.6.1 or Earlier

LabVIEW Setting: Preferred Execution System	TestStand Option: Execute ‘Same as caller’ VIs Using Multiple Threads	
	Enabled	Disabled
Same as caller	Single thread	Single thread
Not Same as caller	Multiple threads	Multiple threads

Using the LabVIEW Adapter, you can also configure the CPU affinity and the number of execution threads the LabVIEW RTE uses to execute VIs for which the preferred execution system is **Same as caller**. You can adjust these settings to optimize performance for a particular system. Refer to the *Using TestStand on SMP Systems* topic of the *NI TestStand Help* for more information about optimizing TestStand performance on SMP systems and configuring CPU affinity.

LabVIEW Run-Time Engine Execution Threads

If the preferred execution system of the VI is **Same as caller**, the LabVIEW RTE uses the TestStand execution thread to execute the VI. With LabVIEW 2009 or later, if you enable the Execute ‘Same as caller’ VIs Using Multiple Threads option in the LabVIEW Adapter Configuration dialog box, the LabVIEW Adapter configures the LabVIEW RTE to use additional LabVIEW threads, in addition to the TestStand execution thread, to execute the VI.

If the preferred execution system of the VI is set to something other than **Same as caller**, LabVIEW uses only LabVIEW threads to execute the VI. The TestStand execution thread processes window messages while waiting for the VI execution to complete.

CPU Affinity for LabVIEW Execution Threads

When TestStand executes a VI using the LabVIEW development system, the CPU affinity for all LabVIEW execution threads is set to allow all CPUs.

When TestStand executes a VI using the LabVIEW RTE, the CPU affinity for LabVIEW execution threads depends on the following factors:

- The CPU affinity of the TestStand execution thread
- Whether you enable the Execute ‘Same as caller’ VIs Using Multiple Threads option in the LabVIEW Adapter Configuration dialog box
- Whether you enable the Additional Threads Inherit Calling Thread’s CPU Affinity option in the LabVIEW Adapter Configuration dialog box
- The setting of the Preferred Execution System option on the Execution page of the VI Properties dialog box in LabVIEW
- The version of LabVIEW that executes the VI

Table 7-4 shows how these factors apply when TestStand executes VIs using the LabVIEW 2009 or later RTE and the preferred execution system of the VI is **Same as caller**.

Table 7-4. CPU Affinity Behavior when Using LabVIEW 2009 or Later RTE

TestStand Option: Execute 'Same as caller' VIs Using Multiple Threads	TestStand Option: Additional Threads Inherit Calling Thread's CPU Affinity	CPU Affinity Behavior
Disabled	—	The VI executes using the TestStand execution thread and its CPU affinity.
Enabled	Disabled	The CPU affinity for all execution threads, including the TestStand execution thread, is set to allow all CPUs. The CPU affinity of the TestStand execution thread is restored to its original affinity when the VI execution completes and the TestStand execution thread returns to TestStand.
Enabled	Enabled	The CPU affinity for all additional LabVIEW execution threads is set to match the CPU affinity of the TestStand thread.

When TestStand executes VIs using the LabVIEW 2009 or later RTE and the preferred execution system of the VIs is set to something to other than **Same as caller**, the CPU affinity for all LabVIEW execution threads is set to allow all CPUs.

Table 7-5 describes the CPU affinity for LabVIEW execution threads when TestStand executes VIs using the LabVIEW 8.6.1 or earlier RTE.

Table 7-5. CPU Affinity Behavior when Using LabVIEW 8.6.1 or Earlier RTE

LabVIEW Setting: Preferred Execution System	CPU Affinity Behavior
Same as caller	The VI executes using the TestStand execution thread and its CPU affinity.
Not Same as caller	The CPU affinity for all LabVIEW execution threads is set to allow all CPUs

Refer to the *Multitasking in LabVIEW* topic of the *LabVIEW Help* for more information about multithreading in LabVIEW.

General Strategies for Optimizing LabVIEW Code Module Performance on SMP Systems

TestStand-based test systems can have many different kinds of VIs. For the purpose of evaluating the performance of VIs on SMP systems, it is helpful to categorize VIs in the following ways:

- **VIs that have singular linear data flow**—Executing these VIs with multiple threads provides no performance benefit.
- **VIs that have complex data flows**—Executing these VIs with multiple threads can provide better performance than a single thread because different data flow branches can execute concurrently on different CPUs.

Use the following recommended LabVIEW Adapter settings to achieve optimal LabVIEW code module performance, depending on the LabVIEW data flow complexity of the majority of the VIs on the test system and the amount of tuning you intend to complete.

- If most of the VIs in the test system have singular linear data flow, National Instruments recommends that you disable the Execute ‘Same as caller’ VIs Using Multiple Threads option in the LabVIEW Adapter Configuration dialog box to avoid time spent in the LabVIEW Adapter creating superfluous threads.
- If most of the VIs in the test system have complex data flow or if the distribution of simple and complex data flow VIs in the test system is even, National Instruments recommends that you enable the Execute ‘Same as caller’ VIs Using Multiple Threads option. Depending on the complexity of the VIs, you can obtain even better performance by using the Number of Threads option in the LabVIEW Adapter Configuration dialog box to modify the number of LabVIEW execution threads TestStand configures the LabVIEW RTE to use to execute VIs.

Furthermore, if you have manually tuned the CPU affinity of TestStand execution threads, in accordance with the information in the *Using TestStand on SMP Systems* topic in the

NI TestStand Help, National Instruments recommends that you run performance tests to determine whether enabling the Additional Threads Inherit Calling Thread's CPU Affinity option in the LabVIEW Adapter Configuration dialog box improves or degrades performance.

- Regardless of the distribution of simple and complex data flow VIs in the test system, you can fine-tune the SMP performance of the system by completing the following steps.
 1. Disable the Execute 'Same as caller' VIs Using Multiple Threads option.
 2. Run performance tests to identify specific VIs that perform better when executing using multiple threads.
 3. Set the preferred execution system on the Execution page of the VI Properties dialog box in LabVIEW for the VIs you identify to something other than **Same as caller** or **user interface**.
 4. Fine-tuning performance might also require that you configure the default number of threads available in a LabVIEW execution system. Refer to the *LabVIEW INI File Tokens Associated with Execution Systems* section of this chapter for more information about configuring the default number of threads available in a LabVIEW execution system.

LabVIEW INI File Tokens Associated with Execution Systems

You can add tokens to a `.ini` file to configure the number of threads assigned to a LabVIEW execution system. The full name and location of the `.ini` file depends on which of the following servers you configure in the LabVIEW Adapter Configuration dialog box to use to execute VIs:

- **LabVIEW RTE**—LabVIEW looks for a `.ini` file with the same name and location as the TestStand application, such as `SeqEdit.ini` for the sequence editor or `TestExec.ini` for a TestStand User Interface.
- **Development System**—LabVIEW looks for a `.ini` file named `LabVIEW.ini`, located in the same directory as the `LabVIEW.exe` development system executable.

For the LabVIEW development system, add the following tokens below the `[LabVIEW]` tag in the `.ini` file. For the LabVIEW RTE, add the following tokens below the `[LVRT]` tag in the `.ini` file.

```
ESys.StdNParallel = -1
ESys<Built-in Execution System>.Normal = n
```

where

- `n` is the number of threads to assign to the execution system. Specify `-1` to use the default number of threads in LabVIEW, which is the greater of four or the number of CPUs in the system.
- `<Built-in Execution System>` is the execution system for which you want to set the number of threads. Table 7-6 lists the valid values to specify the execution system.

Table 7-6. Valid Execution System Values

Execution System Name	Value
Standard	<EmptyString>
Instrument	.Instrument
DAQ	.DAQ
Other1	.other1
Other2	.other2



Note You must restart LabVIEW for the new settings to take effect.



Note The settings you specify apply to all versions of the LabVIEW RTE that TestStand loads.

Calling LabVIEW VIs on Remote Computers

You can directly call VIs on remote computers, including computers that run the LabVIEW development system or a LabVIEW executable and PXI controllers that run the LabVIEW Real-Time (RT) module. TestStand does not support automatically downloading VIs to systems that run the LabVIEW RT module. To use the LabVIEW RT module, you must call a VI on the local system and the local VI must then call the VI on the LabVIEW RT module, or you must manually download the VI to the LabVIEW RT module and call the VI by remote path or by name if the VI is already in memory.

Because TestStand uses the LabVIEW VI Server to run VIs remotely, the remote computers can use any operating system LabVIEW supports, including Linux and Mac OS X.

To call a VI remotely, you must configure the TestStand step to specify that the call occurs on a remote computer. In addition, you must configure the remote computer to allow TestStand to call VIs located on the computer. You must also configure the computer running TestStand to have network access to the remote computer running the LabVIEW VI Server.

Configuring a Step to Run Remotely

Complete the following steps to configure a step to run remotely. The VI must be present on the local computer so TestStand can configure and run the VI.

1. Click the **Advanced Settings** button on the LabVIEW Module tab to launch the LabVIEW Advanced Settings window, in which you can specify the name or an expression that evaluates to the name of the remote computer on which you want to run the VI.
2. If the remote computer is running the LabVIEW development system or a LabVIEW executable and you are calling a VI that is not part of a LabVIEW project, use the Remote VI Path control in the LabVIEW Advanced Settings window to specify the path to the VI on the remote computer. If you are using a VI that is part of a LabVIEW project, use the Remote Project Path control to specify the path to the LabVIEW project on the remote computer and use the Remote VI Path control to specify the path within the LabVIEW project of the VI you want to call. Refer to the *NI TestStand Help* for more information about the Remote VI Path control.



Note TestStand does not support remotely calling VIs that use projects on PXI controllers that run the LabVIEW RT Module because LabVIEW RT does not support LabVIEW projects.

You can use the LabVIEW development system to download VIs to the PXI controller. You can also use the TestStand FTP Files step type to download files from and upload files to a remote computer. Refer to the *NI TestStand Help* for more information about the FTP Files step type.

You can also use FTP to download VIs to the PXI controller. Use the LabVIEW development system to create a Source Distribution with all the VIs to ensure that you include all the dependencies of the VIs to transfer to the hard drive of the LabVIEW RT target. In the distribution, disable the options to exclude VIs from the `<LabVIEW>\vi.lib`, `<LabVIEW>\instr.lib`, and `<LabVIEW>\user.lib` directories. Then, use FTP to transfer the source distribution output to the hard drive of the LabVIEW RT target.

After you download the VIs to the PXI controller running the LabVIEW RT module, you can use the Remote VI Path option in the LabVIEW Advanced Settings window to call the VIs.



Note Refer to the *Getting Started with the LabVIEW Real-Time Module* manual in the `<LabVIEW>\manuals` directory for more information about downloading VIs to a PXI controller.

Configuring the LabVIEW VI Server to Run VIs Remotely

The LabVIEW development system or built executable must be running on the remote computer, and you must configure the development system or built executable to allow VI calls through the TCP/IP protocol of the VI Server.

You can configure the VI Server settings for a main application instance; a project application instance; a target, such as a LabVIEW RT Server; or a LabVIEW built application.

- To configure the VI Server settings for the main application instance, select **Tools»Options** in LabVIEW to launch the Options dialog box and select the **VI Server** category.
- To configure the VI Server settings for a project application instance, right-click **My Computer** in the LabVIEW Project Explorer window, select **Properties** from the context menu to launch the My Computer Properties dialog box, and select the **VI Server** category.
- To configure the VI Server settings for a target, right-click the appropriate target in the LabVIEW Project Explorer window, select **Properties** from the context menu to launch the Target Properties dialog box, and select the **VI Server** category.
- To configure the VI Server settings for a LabVIEW built executable, add VI Server-specific settings to the LabVIEW configuration INI file associated with the executable, or use the Server properties of the Application class to programmatically enable and configure the VI Server for a running application.

Use the VI Server category in the Options dialog box to enable the TCP/IP protocol and the VI call options. You can also specify the TCP/IP port the server uses. The port you specify in LabVIEW must be the same port you specify in TestStand in the LabVIEW Advanced Settings window.

Use the Machine Access section of the VI Server category in the Options dialog box to allow specific computers access to the LabVIEW VI Server. You can also specify certain computers or entire domains that can call VIs on the server computer.

Use the Exported VIs section of the VI Server category in the Options dialog box to configure the VIs you want to call through the LabVIEW VI Server. You must export all VIs you want to call remotely from TestStand. The default setting in LabVIEW is to export all VIs. The asterisk (*) in the Exported VIs list control indicates this setting.

Refer to the *LabVIEW Help* for more information about configuring VI Server options.

User Access to VI Server

TestStand does not support user-based VI security for executing VIs on remote computers. You must use the Machine access list in the Machine Access section of the VI Server category in the Options dialog box to protect a VI server on a remote computer.

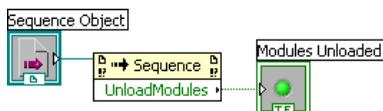
Using the TestStand ActiveX APIs in LabVIEW

You can use the TestStand API or TestStand User Interface (UI) Controls from LabVIEW test and user interface VIs. Refer to the LabVIEW documentation for more information about ActiveX concepts and how to use LabVIEW as an ActiveX client.

Invoking Methods

TestStand objects have methods you invoke to perform an operation or function. In LabVIEW, use the Invoke Node to invoke methods. The block diagram in Figure 9-1 shows how to invoke the `Sequence.UnloadModules` method to unload all code modules in the sequence.

Figure 9-1. Invoking the UnloadModules Method

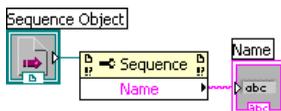


Accessing Built-In Properties

TestStand defines a number of built-in properties that are always present for objects, such as steps and sequences. Nearly every kind of TestStand object has built-in properties that are static with respect to the TestStand API, which you can use to access the properties in the programming language you specify. Examples of built-in properties are the `Sequence.Name` property and the `SequenceContext.Sequence` property.

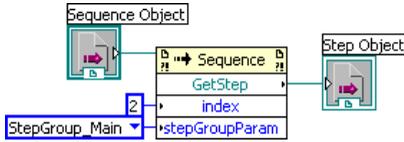
In LabVIEW, use the Property Node to access built-in properties. The block diagram in Figure 9-2 shows how to obtain the value of the `Sequence.Name` property.

Figure 9-2. Obtaining the Value of the Name Property from a Sequence Object



The block diagram in Figure 9-3 shows how to obtain a reference to a step of a sequence that a Sequence object references.

Figure 9-3. Obtaining a Reference to a Step of a Sequence That a Sequence Object References



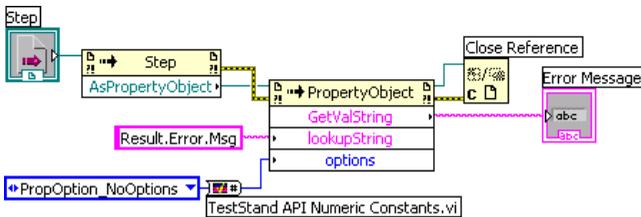
Accessing Dynamic Properties

In TestStand, you can define custom step properties, sequence local variables, sequence file global variables, and station global variables. Because the TestStand API is independent 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.

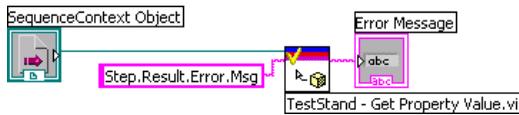
To access dynamic properties of an object, you must first use the `AsPropertyObject` method of the object to convert the specific object reference to a `PropertyObject` reference. Then, use the `PropertyObject` interface to access custom properties of the object. The `PropertyObject` interface uses a lookup string to specify the specific custom property.

The block diagram in Figure 9-4 shows how to use the `GetValString` method on the `PropertyObject` interface of a Step object to obtain the error message value for the current step.

Figure 9-4. Using the `GetValString` Method to Obtain the Error Message Value for the Current Step



You can use the TestStand - Get Property Value VI or the TestStand - Set Property Value VI to access dynamic properties of a `SequenceContext` object. The block diagram in Figure 9-5 shows how to use the TestStand - Get Property Value VI to obtain the error message value for the current step.

Figure 9-5. Using VIs to Obtain the Error Message Value for the Current Step

Releasing ActiveX References

When a method or property returns an ActiveX reference, you must use the Automation Close function in LabVIEW to release the reference.



Note If you do not release the ActiveX reference, LabVIEW does not release it for you until the VI hierarchy finishes executing. Repeatedly opening references to objects without closing them can cause the computer to run out of memory.

Using TestStand API Constants and Enumerations

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

To facilitate programming with the TestStand API within VIs and to help you manage all the available string and numeric constants for the TestStand API properties and methods, TestStand provides two enumerated constant VIs—the TestStand API String Constants VI and the TestStand API Numeric Constants VI.

Use the TestStand API String Constants VI to locate and select the string constant arguments you can use with TestStand API properties and methods. Use the TestStand API Numeric Constants VI to locate and select the various numeric constant arguments you can use with TestStand API properties and methods. Use both of these VIs in conjunction with the constants associated with the TestStand API methods and properties. Refer to the *NI TestStand Help* for more information about using constants with the TestStand methods and properties.

Use the OR function in LabVIEW to combine more than one numeric constant. When you need to combine more than two constants, use the Compound Arithmetic function and set the mode to OR.

The block diagram in Figure 9-4 shows how to use the TestStand API Numeric Constants VI to obtain the value of the `PropOptions_NoOptions` constant.

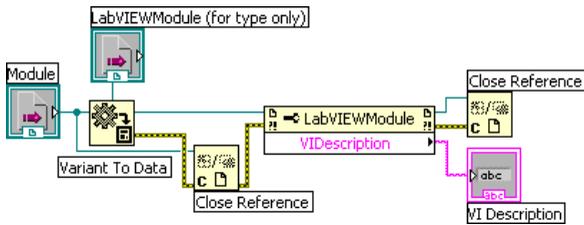
Some TestStand API methods require enumeration input arguments. For these methods, right-click the input parameter on the Invoke Node in LabVIEW, select **Create Constant** from the context menu to create a LabVIEW ring constant, and select the value you want in the resulting constant.

Obtaining a Different Interface for TestStand Objects

In some cases, you might need to obtain a different interface for a TestStand object than the current interface. In ActiveX/COM terminology, this action is called a QueryInterface. For example, when you have a Module reference to a LabVIEWModule object and need to access the LabVIEWModule interface instead, perform a QueryInterface on the Module object to obtain the LabVIEWModule interface. In LabVIEW, use the Variant To Data function with the reference to accomplish this task.

The block diagram in Figure 9-6 shows how to obtain the LabVIEWModule interface of a Module object to obtain the VIDescription property of the object. Notice that you must release the reference the Variant To Data function returns when you are finished with it.

Figure 9-6. Converting a Module Reference to a LabVIEWModule Reference



Complete the following steps to select an ActiveX Server class (type) for an Automation refnum control or constant

1. Right-click the **Automation** control or constant and select **Select ActiveX Class»Browse** to launch the Select Object From Type Library dialog box.
2. Select the appropriate ActiveX Server from the Type Library list control.
3. Confirm that the **Show Creatable Objects Only** option is not selected.
4. Select the required ActiveX Server Class in the **Object** control.
5. Click **OK** to close the Select Object From Type Library dialog box.

For example, the LabVIEW Module class is defined in the NI TestStand <version> Adapter API <version> Version 1.0 Server. To select the LabVIEW Module class, select **NI TestStand <version> Adapter API <version> Version 1.0** as the ActiveX server, and then select **LabVIEW Module** as the required ActiveX Server Class in the **Object** control.

Acquiring a Derived Class from the PropertyObject Class

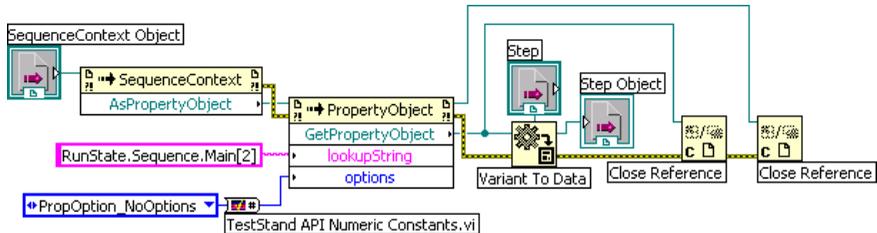
In some cases, you might need to use the `PropertyObject` class methods to obtain a reference to a TestStand object. You might then want to access one of the static properties of the TestStand object, such as the run mode for the third step in the Main step group of the currently executing sequence. For methods in the `PropertyObject` class that can return objects derived from `PropertyObject`, you must acquire the derived interface for the object to access the built-in properties and methods of the derived class. Use the method described in the [Obtaining a Different Interface for TestStand Objects](#) section of this chapter to acquire the derived interface for an object.

The block diagram in Figure 9-7 shows how to use a lookup string to obtain a reference to a Step object from a `SequenceContext` object.



Note You must close the Step object reference when you no longer need it by using a Close Reference node.

Figure 9-7. Using a Lookup String to Obtain a Reference to a Step Object from a SequenceContext Object

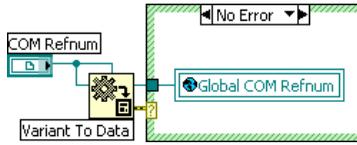


Duplicating COM References in LabVIEW Code Modules

When TestStand passes an ActiveX reference to a LabVIEW code module, the reference automatically closes when the called VI completes executing and returns to TestStand. Wiring the reference to a global variable or to a shift register in another VI does not prevent the reference from closing. Consequently, any subsequent use of the original reference within LabVIEW fails.

Use the LabVIEW Variant to Data function to create a duplicate reference for use after calling a VI. Wire the original reference to the **type** and **variant** inputs of the Variant to Data function and wire the **data** output to a global variable, as shown in Figure 9-8.

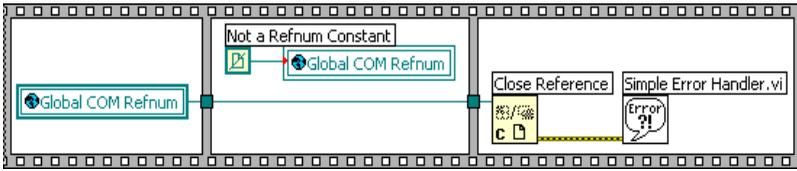
Figure 9-8. Creating Duplicate COM Reference



The reference in the global variable prevents TestStand from releasing the referenced object until you explicitly close the reference in LabVIEW or until TestStand unloads the code module into which it passed the reference.

Once LabVIEW no longer needs the duplicate reference, read the global variable and wire the reference to the LabVIEW Close Reference function to close the reference. Wire the global variable to the LabVIEW Not A Refnum constant to close the global variable, as shown in Figure 9-9.

Figure 9-9. Closing Duplicate COM Reference



Setting the Preferred Execution System for LabVIEW VIs

When a VI calls synchronous methods of the TestStand API, you must correctly set the Preferred Execution System on the Execution page of the VI Properties dialog box for the VIs. When you call synchronous methods that do not return until the LabVIEW server executes a VI on behalf of TestStand, the VI that calls these methods and the VI that TestStand attempts to run using the LabVIEW VI Server cannot run in the same LabVIEW execution system. When the VIs attempt to run in the same execution system, the execution of the synchronous TestStand method consumes the execution system in which the VI needs to run, causing deadlock.

Because LabVIEW handles ActiveX communication through the user interface execution system, you cannot set either of the VIs in this scenario to run in the user interface execution system. For example, you can have a LabVIEW code module that calls the `Engine.NewExecution` method followed by the `Engine.WaitForEnd` method and a new execution that calls LabVIEW code modules. Deadlock can occur when either VI in this scenario uses **Same as caller** or **user interface** as the preferred execution system.

To avoid deadlock, use the Execution page of the VI Properties dialog box to select a different preferred execution system, such as **other 1** or **other 2**, for each VI.

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 LabVIEW, use the Register Event Callback function to register a callback VI and then use the Unregister for Events function to close the callback before you close the application.

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

Calling Legacy LabVIEW VIs

In versions of TestStand earlier than 3.0, you could call VIs only with a specific set of controls and indicators. Using TestStand 3.0 or later, you can call VIs with a variety of connector panes, including VIs with legacy configurations.

Format of Legacy VIs

All legacy-style VIs must include the **Test Data** cluster and **error out** cluster indicators. The **Input Buffer**, **Invocation Info**, and **Sequence Context** controls are optional inputs to legacy VIs, which can contain any combination of these controls.

You must assign each control and indicator of the test VI to a terminal on the connector pane of the test VI. If these assignments do not exist, TestStand returns an error when it attempts to call the test VI. TestStand does not require that you use a particular connector pane pattern, and it does not require that you assign the controls and indicators to specific terminals.

Although you usually create new VIs using the LabVIEW Module tab for steps that use the LabVIEW Adapter, TestStand can also create legacy-style VIs. Refer to Chapter 5, [Configuring the LabVIEW Adapter](#), for more information about configuring the LabVIEW Adapter to create new legacy-style VIs.

You can use the following methods to pass data between the code module and TestStand:

- Use the **Test Data** cluster
- Use the sequence context ActiveX reference to call the TestStand ActiveX API functions to set the variables used to store the results of the test, such as `Step.Result.PassFail`



Note The values the sequence context ActiveX reference sets take precedence over the values the **Test Data** cluster sets. When you use both methods to set the value of the same variable, TestStand recognizes the values the sequence context ActiveX reference sets and ignores the values the **Test Data** cluster sets. You can use the sequence context ActiveX reference and the **Test Data** cluster together in the code module if you do not try to set the same variable twice. For example, when you use the sequence context ActiveX reference to set the value of `Step.Result.PassFail` and then use the **Test Data** cluster to set the value of `Step.Result.ReportText`, TestStand sets both values correctly.



Note The specific control and indicator labels described in this chapter are required. Do not modify them in any way.

Test Data Cluster

The LabVIEW Adapter uses the **Test Data** cluster to return result data from the VI to TestStand, which then uses the data to make a PASS/FAIL determination.

Table 10-1 lists the elements of the **Test Data** cluster, the data type of the cluster element, and descriptions of how the LabVIEW Adapter uses each cluster element.

Table 10-1. Test Data Cluster Elements

Cluster Element	Data Type	Description
PASS/FAIL Flag		The test VI sets this element to indicate whether the test passed. Valid values are <code>True</code> (PASS) or <code>False</code> (FAIL). The adapter copies the value into the <code>Step.Result.PassFail</code> property when the property exists.
Numeric Measurement		Numeric measurement the test VI returns. The adapter copies this value into the <code>Step.Result.Numeric</code> property when the property exists.
String Measurement		String value the test VI returns. The adapter copies this string into the <code>Step.Result.String</code> property when the property exists.
Report Text		Output message to include in the report. The adapter copies this message value into the <code>Step.Result.ReportText</code> property when the property exists.

The LabVIEW Adapter also supports an legacy version of the **Test Data** cluster from the LabVIEW Test Executive. The LabVIEW Test Executive **Test Data** cluster does not contain a **Report Text** element but instead contains the **Comment** and **User Output string elements**.

Table 10-2 lists the elements of the legacy **Test Data** cluster, the data type of the cluster element, and descriptions of how the LabVIEW Adapter uses each cluster element.

Table 10-2. Legacy Test Data Cluster Elements from LabVIEW Test Executive

Cluster Element	Data Type	Description
Comment		Output message to display in the report. The adapter copies this message value into the <code>Step.Result.ReportText</code> property when the property exists.
User Output		String value the test VI returns. The adapter dynamically creates the step property <code>Step.Result.UserOutput</code> and copies the string value to the step property.

Error Out Cluster

TestStand uses the content of the **error out** cluster to determine when a run-time error occurs and when to take appropriate action if necessary. When you create a VI, use the standard LabVIEW **error out** cluster.

Table 10-3 lists the elements of the **error out** cluster, the data type of the cluster element, and descriptions of how the LabVIEW Adapter uses each cluster element.

Table 10-3. Error Out Cluster Elements

Cluster Element	Data Type	Description
status		The test VI must set this element to <code>True</code> when an error occurs. The adapter copies the output value into the <code>Step.Result.Error.Occurred</code> property when the property exists.
code		The test VI can set this element to a non-zero value when an error occurs. The adapter copies the output value into the <code>Step.Result.Error.Code</code> property when the property exists.
source		The test VI can set this element to a descriptive string when an error occurs. The adapter copies the output value into the <code>Step.Result.Error.Msg</code> property when the property exists.

Input Buffer String Control

Use the **Input Buffer** string control to pass input data directly to the VI. The LabVIEW Adapter automatically copies the `Step.InBuf` property value into the **Input Buffer** string control when the property exists.

Invocation Info Cluster

Use the **Invocation Info** cluster to pass additional information to the VI.

Table 10-4 lists the elements of the **Invocation Info** cluster, the data type of the cluster element, and descriptions of how the LabVIEW Adapter uses each cluster element.

Table 10-4. Invocation Info Cluster Elements

Cluster Element	Data Type	Description
Test Name		The adapter uses the name of the step that invokes the test VI.
loop #		The adapter uses the loop count when the step that invokes the test VI loops on the step.
Sequence Path		The adapter uses the name and absolute path of the sequence file that runs the test VI.
UUT Info		The adapter uses the value from the <code>RunState.Root.Locals.UUT.SerialNumber</code> property when the property exists. Otherwise, the adapter copies an empty string. Refer to the <i>NI TestStand Help</i> for more information about how to configure the Serial Number String option in the Legacy VI Settings dialog box.
UUT #		The adapter uses the value from the <code>RunState.Root.Locals.UUT.UUTLoopIndex</code> property when the property exists. Otherwise, the adapter copies an empty string. Refer to the <i>NI TestStand Help</i> for more information about how to configure the UUT Iteration Number option in the Legacy VI Settings dialog box.

Sequence Context Control

Use the **Sequence Context** control to obtain a reference to the TestStand `SequenceContext` object. You can 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 VI.

Using LabWindows/CVI with TestStand

Use this section of this manual to learn how to use LabWindows/CVI with TestStand.

- Chapter 11, *Calling LabWindows/CVI Code Modules from TestStand*—Use the LabWindows/CVI Adapter to call LabWindows/CVI code modules from TestStand.
- Chapter 12, *Creating, Editing, and Debugging LabWindows/CVI Code Modules from TestStand*—Use the LabWindows/CVI Adapter to create new code modules to call from TestStand and to edit and debug existing code modules.
- Chapter 13, *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 to hold any number of other data types.
- Chapter 14, *Configuring the LabWindows/CVI Adapter*—Configure the LabWindows/CVI Adapter to show function arguments in step descriptions, set the default structure packing size, select where steps execute, and establish a code template policy.
- Chapter 15, *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.
- Chapter 16, *Using the TestStand ActiveX APIs in LabWindows/CVI*—You can use the TestStand API or TestStand User Interface (UI) Controls from LabWindows/CVI code modules and user interface source code.
- Chapter 17, *Adding Type Libraries to LabWindows/CVI DLLs*—When a DLL contains export information or when 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.
- Chapter 18, *Calling Legacy LabWindows/CVI Code Modules*—In versions of TestStand earlier than 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 or later, you can call functions with a variety of parameter data types, including code modules with legacy function prototypes.

Calling LabWindows/CVI Code Modules from TestStand

Use the LabWindows/CVI Adapter to call LabWindows/CVI code modules from TestStand.

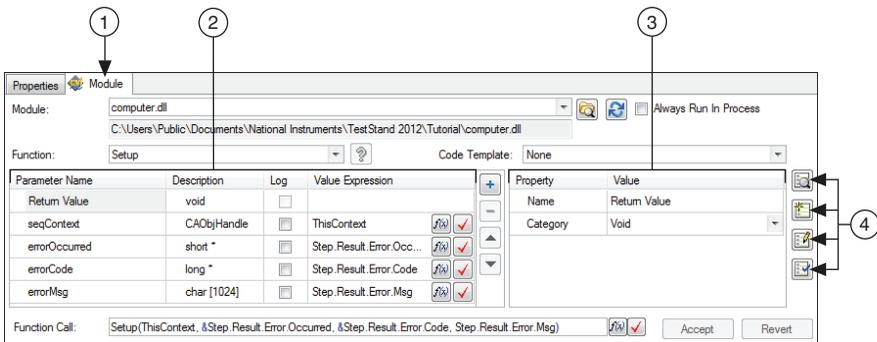
Required LabWindows/CVI Settings

All the tutorials in Part II of this manual require you to have the LabWindows/CVI development system 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 14, [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 on the Step Settings pane, as shown in Figure 11-1.

Figure 11-1. LabWindows/CVI Module Tab



- 1 LabWindows/CVI Module Tab
- 2 Parameters Table Control

- 3 Parameter Details Table Control
- 4 Source Code Buttons

The LabWindows/CVI Adapter supports calling functions in C source files, object files, static library files, and dynamic link library (DLL) files. The LabWindows/CVI 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 14, [Configuring the LabWindows/CVI Adapter](#), for additional requirements for calling functions in C source files, object files, and static library files.

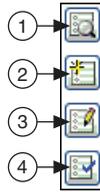
You also use the LabWindows/CVI Module tab to specify the function prototype, which includes the data type of each parameter and the values to pass for each parameter. The Parameters Table control shows all the available parameters for the function call and an entry for the return value. You can insert, remove, or rearrange the order of the parameters. The Parameters Table control contains the following columns:

- **Parameter Name**—A symbolic name for the parameter.
- **Description**—The short description of the parameter type using ANSI C syntax.
- **Log**—When enabled, the step logs the parameter as an additional result. Enabling this option is equivalent to enabling the Value to Log option on the Additional Results panel of the Properties tab of the Step Settings pane. Refer to the *NI TestStand Help* for more information about the Additional Results panel.
- **Value Expression**—The argument expression to pass.

When you select a parameter in the Parameters Table control, the Parameter Details Table control contains the specific details about the parameter. The data type (Numeric, String, Object, C Struct, or Array) determines the information required for a parameter. 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 the function arguments at once.

Source Code Buttons

Use the source code buttons, shown in Figure 11-2, to create 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 for TestStand to call the code module.

Figure 11-2. LabWindows/CVI Module Tab Source Code Buttons

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



Note If LabWindows/CVI returns a warning when you open a project that some TestStand API files were not found, remove the files from the project and then add them again from the <National Instruments>\Shared\CVI\instr\TestStand\API directory for LabWindows/CVI 8.5 or later or from the <CVI>\instr\TestStand\API\CVI directory for LabWindows/CVI 8.1.1 or earlier. The <National Instruments> directory is located at C:\Program Files. The <CVI> directory is located at C:\Program Files\National Instruments.

Refer to the *NI TestStand Help* for more information about the LabWindows/CVI Module tab and the source code buttons.

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 configure the step to call a code module.

1. Launch the sequence editor and click the LabWindows/CVI icon, as shown in the following figure, at the top of the Insertion Palette to specify the module adapter for the step. You can also select adapters from the Adapter ring control on the Environment toolbar. The adapter you select applies only to the step types that can use the module adapter. Refer to the *NI TestStand Help* for more information about the Insertion Palette and the TestStand toolbars.



2. Open a new Sequence File window if one is not already open.
3. Select **File»Save <filename> As** and save the sequence file as <TestStand Public>\Tutorial\CallCVIcodeModule.seq. The <TestStand Public> directory is located by default at C:\Users\Public\Documents\National Instruments\TestStand on Windows 7/Vista and at C:\Documents and Settings\All Users\Documents\National Instruments\TestStand on Windows XP.

4. Insert a Pass/Fail Test step in the Main step group in 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, as shown in the following figure and located to the right of the Module control, select `<TestStand Public>\Tutorial\CallCVIcodeModule.dll`, and click **Open**.



6. On the LabWindows/CVI Module tab, select the `PassFailTest` function from 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. When the function parameter information is not defined, you can select a code template from the Code Template ring control to specify the function prototype or add parameters to the Parameters Table control to specify the function prototype.

7. Select **PassFail template for LabWindows/CVI** from the Code Template ring control. The Parameters Table control contains the default value expressions the code template specifies. 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. 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 the **Pass/Fail** output parameter.



Note You must have Internet Explorer 7.0 or later to view TestStand reports.

When you initiate an execution, by default, the TestStand Sequence Analyzer uses the rules and settings in the current sequence analyzer project to analyze the active sequence file and detect the most common situations that can cause run-time failures. The sequence analyzer prompts you to resolve the reported errors before you execute the sequence file. If no errors exist or if you select to continue execution with errors, the sequence editor opens an Execution window.

Click the **Toggle Analyze File Before Executing** button, as shown in the following figure, on the Sequence Analyzer toolbar or select **Debug»Sequence Analyzer»Toggle Analyze File Before Executing** to enable or disable this option. Click the **Analyze <filename>** button on the Sequence Analyzer toolbar to analyze a sequence file at any time during

development. Refer to the *NI TestStand Help* for more information about using the sequence analyzer.



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

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

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 from TestStand

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

1. Launch the TestStand Sequence Editor and select the **LabWindows/CVI Adapter** on the Insertion Palette.
2. Open `<TestStand Public>\Tutorial\CallCVIcodeModule.seq`. You created this sequence file in the *Creating and Configuring a New Step Using the LabWindows/CVI Adapter* section of Chapter 11, *Calling LabWindows/CVI Code Modules from TestStand*.



Note If you did not save `CallCVIcodeModule.seq`, you can open the `CallCVIcodeModule.seq` solution file from the `<TestStand Public>\Tutorial\Solution` directory. If you use the solution file, save any changes you make to the file in the `<TestStand Public>\Tutorial` directory so you do not overwrite the installed solution file.

3. Save the sequence file as `<TestStand Public>\Tutorial\CallCVIcodeModule2.seq`.
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 use the LabWindows/CVI Module tab to complete the following steps.
 - a. For the Module control, click the **File Browse** button and select `<TestStand Public>\Tutorial\CallCVIcodeModule.dll`.
 - b. Enter `NumericLimitTest` in the Function ring control.
 - c. Select **NumericLimit template for LabWindows/CVI** from the Code Template ring control. 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.

6. Click the **Source Code Files** button to launch the CVI Source Code Files window and complete the following steps.
 - a. Enter `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**.
7. Click the **Create Code** button to create a code module. When you click the **Create Code** button, TestStand launches the Select a Source File dialog box, in which you can specify the source code to use for the code module.
8. Browse to the `<TestStand Public>\Tutorial` directory and click **OK**. TestStand creates a new code module named `CVINumericLimitTest.c` based on the available code templates for the TestStand Numeric Limit Test step type 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 whether the step passes or fails. Code modules can pass step properties as parameters to and from the module or use the TestStand API in the module to update step properties. When 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.

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. Click **OK** when LabWindows/CVI prompts you that the files are created.
12. Return to the sequence editor and save the sequence file.
13. 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`.
14. Select **File»Unload All Modules** to unload the DLL.
15. Close the Execution window, but leave the sequence file open so you can use it in the next tutorial.

Refer to the *NI TestStand Help* for more information about creating code templates for step types.

Editing an Existing Code Module from TestStand

Complete the following steps to edit an existing code module from TestStand.

1. Open `<TestStand Public>\Tutorial\CallCVIcodeModule2.seq` if it is not already open. You created this sequence file in the [Creating a New Code Module from TestStand](#) section of this chapter.
2. Right-click the **CVI Numeric Limit Test** step and select **Edit Code** from the context menu or click the **Edit Code** button on the LabWindows/CVI Module tab. 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. Leave LabWindows/CVI open.
5. Rebuild the DLL. Click **OK** when LabWindows/CVI prompts you that the files are created.
6. In the 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.
7. Close the Execution and Sequence File windows.

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. Select **Execute»Run MainSequence** to start an execution of `MainSequence`.
4. When the execution pauses, click the **Step Into** button on the Debug toolbar in TestStand. 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. 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. 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** again when the adapter displays a warning that confirms TestStand will unload all modules. Click **Done** in the Adapter Configuration dialog box. Begin step 3 again. Refer to Chapter 14, [Configuring the LabWindows/CVI Adapter](#), for more information about configuring the LabWindows/CVI Adapter.

5. Click the **Step Into** button or the **Step Over** button on the LabWindows/CVI toolbar to begin stepping through the code module. You can click **Continue** at any time to finish stepping through the code module and to continue execution in TestStand.
6. When you finish 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 and the Sequence File window.

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 to hold any number of other data types. TestStand container data types are analogous to C structures in LabWindows/CVI.

LabWindows/CVI includes a greater variety of built-in data types than TestStand does, so TestStand converts LabWindows/CVI data types in certain ways when calling code modules, as shown in Table 13-1.

Table 13-1. TestStand Equivalents for LabWindows/CVI C 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.
__int64, long long	Number {Signed 64-bit integer}
unsigned __int64, unsigned long long	Number {Unsigned 64-bit integer}
const char*, char[], const wchar_t*, const unsigned short*, wchar_t[], or unsigned short[]	Path, String, or Expression Refer to the <i>Calling Code Modules with String Parameters</i> 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 <i>Calling Code Modules with Object Parameters</i> section of this chapter for more information about using the object reference data type.
Array of <i>x</i>	Array of TestStand (<i>x</i>)

Table 13-1. TestStand Equivalents for LabWindows/CVI C Data Types (Continued)

LabWindows/CVI C Data Type	TestStand Data Type
Pointer to <i>x</i>	Object reference You can store a pointer a LabWindows/CVI code module returns in a TestStand object reference variable. You can then pass a pointer to a subsequent call to a LabWindows/CVI code module.
struct	Container Refer to the <i>Calling Code Modules with Struct Parameters</i> section of this chapter for more information about using the container data type.



Note The LabWindows /CVI Adapter supports return values of type void, void*, and numeric, which include doubles, and 8-, 16-, 32- and 64-bit integers.

Calling Code Modules with String Parameters

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

When 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 content of the string.

When you pass a string as a buffer, the LabWindows/CVI Adapter copies the content 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. When you pass a string value that is longer than the buffer size you specify, the LabWindows/CVI Adapter resizes the temporary buffer so it is large enough to hold the content 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 copies only data from the beginning of the temporary buffer up to and including the first NUL character.

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

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.

When 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. When 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. When you pass the object by reference and the function alters the value of the reference, the function must release the original reference.

When you pass an array of object references that use the CAObjHandle data type and the function alters the value of any of the array elements, do not use the function to discard the original handles the array elements contain. TestStand discards the original handles after the call completes.

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.

When you pass a C struct by reference, a member of the C struct uses the CAObjHandle data type, and the function alters the value of the struct member, do not use the function to discard the original handle. TestStand discards the original handle after the call completes.

Refer to the *NI TestStand Help* for more information about where TestStand stores custom data types and about the Type Properties dialog box.

Calling Code Modules with Pointer Parameters

When you configure calls to code modules that return values of type `void*`, you can store the memory address for the pointer in TestStand as an Object Reference. You can also pass pointers previously stored in Object References to code modules that use pointers as parameters. This allows you to allocate a pointer in a code module and store it in TestStand and then pass that pointer to a different code module during the sequence execution.

You must release the pointer from a code module when it is no longer needed.

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.

Creating a New Custom Data Type

Complete the following steps to create a new container data type that contains numeric and string subproperties.

1. Open `<TestStand Public>\Tutorial\CallCVIcodeModule2.seq` if it is not already open. You created this sequence file in the [Creating a New Code Module from TestStand](#) section of Chapter 12, [Creating, Editing, and Debugging LabWindows/CVI Code Modules from TestStand](#).
2. Select the **LabWindows/CVI Adapter** on the Insertion Palette.
3. Select **View»Types** or right-click the Sequence File window and select **View»Types** from the context menu. Ensure the `CallCVIcodeModule2.seq` sequence file is selected on the View Types For pane.
4. Select the **Custom Data Types** item in the Types window.
5. Right-click the **Custom Data Types** item and select **Insert Custom Data Type»Container** from the context menu to insert a new data type. Rename the new container data type `CVITutorialStruct`.
6. Expand the `CVITutorialStruct` item.
7. Right-click the `CVITutorialStruct` item and select **Insert Field»Number** from the context menu to insert a new field in the data type. Rename the new field `Measurement`.
8. Right-click the `CVITutorialStruct` item and select **Insert Field»String** from the context menu to insert another new field in the `CVITutorialStruct` container data type. Rename the new field `Buffer`.
9. Save the sequence file but leave the file open, and continue to the next tutorial.

Refer to the *NI TestStand Help* for more information about custom data types and the Types window.

Specifying Structure Passing Settings

Complete the following steps to specify the structure passing properties for the `CVITutorialStruct` container data type.

1. Right-click the `CVITutorialStruct` item in the Types window and select **Properties** from the context menu to launch the Type Properties dialog box. The name of the Type Properties dialog box is specific to the name of the property you select.
2. Click the **C Struct Passing** tab of 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. Ensure the String Type control is set to **C String Buffer** to allow the C function to alter the value of the structure field.
6. Click the **Version** tab of the Type Properties dialog box and disable the **Modified** option.
7. Select **OK** to close the Type Properties dialog box.
8. Save the sequence file, leave the file open, and continue to the next tutorial.

Calling a Function with a Struct Parameter

Complete the following steps to 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. Save the sequence file as `CallCVIModule3.seq`.
2. Click the **Variables** pane.
3. Right-click the **Locals** item on the Variables pane and select **Insert Local»Type»CVITutorialStruct** from the context menu to insert an instance of the container data type. Rename the new variable `CVIstruct`.
4. Select the **Steps** pane.
5. Select the **LabWindows/CVI Adapter** on the Insertion Palette, if it is not already selected.
6. Insert a new Action step into the Main step group of `MainSequence` after the CVI Numeric Limit Test step. Rename the step `Pass Struct Test`.
7. On the LabWindows/CVI Module tab of the Step Settings pane, click the **File Browse** button next to the Module control and select the `<TestStand Public>\Tutorial\CallCVIcodeModule.dll` file.
8. Enter `PassStructTest` in the Function control.

9. Click the **Add Parameter** button, as shown in the following figure, to insert a new parameter in the Parameters Table control 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`.
10. Enter `Locals.CVIStruct` in the Value Expression column for the parameter in the Parameters Table control.
 11. 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 the function.
 - a. Enter `CVIStructPassingTest.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**.
 12. Click the **Create Code** button to create a code module. TestStand launches the Select a Source File dialog box.
 13. Browse to the `<TestStand Public>\Tutorial` directory and click **OK**. TestStand creates a new source file named `CVIStructPassingTest.c` with an empty function.
 14. In LabWindows/CVI, complete the following steps.
 - a. Add the following type definition before the `PassStructTest` function:


```
struct CVITutorialStruct {
    double measurement;
    char buffer[256];
};
```
 - b. Add the following code to the `PassStructTest` function:


```
if (cviStruct)
{
    cviStruct->measurement = 10.0;
    strcpy(cviStruct->buffer, "Average Voltage");
}
```
 - c. Add the following statement to the top of the source file to include the declaration of the `strcpy` function:


```
#include <ansi_c.h>
```
 15. Save and close the source file.

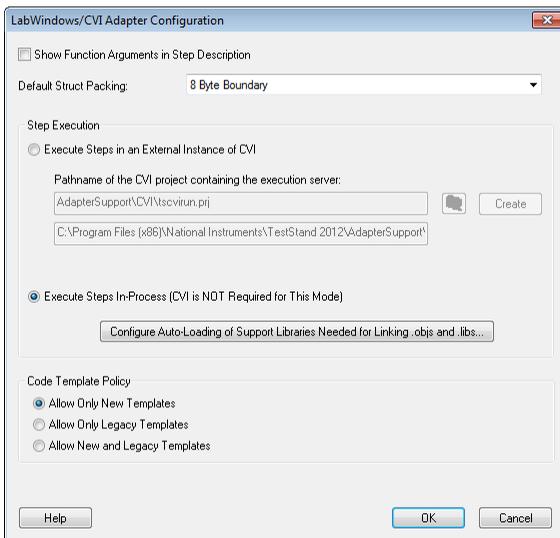
16. In the LabWindows/CVI project window, select **Build»Create Debuggable Dynamic Link Library** to rebuild the DLL. Click **OK** when LabWindows/CVI notifies you that the files are created.
17. Return to the sequence editor and place a breakpoint on the new Pass Struct Test step.
18. Save the sequence file.
19. Select **Execute»Run MainSequence** to start a new execution of `MainSequence`.
20. Step through the sequence and review the values in the `Locals.CVIStruct` variable on the Variables pane before and after executing the new step. Refer to the *Using the Execution Window Variables Pane* section of Chapter 5, *Using Variables and Properties*, in the *Getting Started with TestStand* manual for more information about the Variables pane.
21. Select **File»Unload All Modules** to unload the DLL.
22. Close the Execution window.

Configuring the LabWindows/CVI Adapter

Configure the LabWindows/CVI Adapter to show function arguments in step descriptions, set the default structure packing size, select where steps execute, and establish a code template policy.

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 14-1.

Figure 14-1. LabWindows/CVI Adapter Configuration Dialog Box



Showing Function Arguments in Step Descriptions

Enable the **Show Function Arguments in Step Description** option to include the parameters with the function in the description for a step in the TestStand Sequence Editor and TestStand User Interfaces. When you disable this option, the description displays only 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 ring control to specify the default setting for how the LabWindows/CVI Adapter packs structure parameters it passes. You can select 1-, 2-, 4-, 8-, or 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 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 Code Modules with Struct Parameters](#) section of Chapter 13, [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 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. When you want a 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.

Refer to the *NI TestStand Help* for more information about using the TestStand Deployment Utility.

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.

When you click **Step Into** in the 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

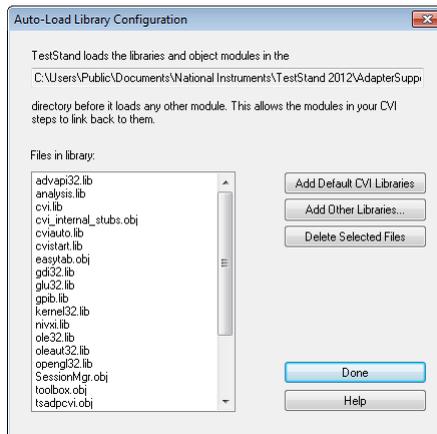
When executing code modules in the same process as the sequence editor or user interface, the LabWindows/CVI Adapter loads and runs code modules directly without using the LabWindows/CVI development environment.

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 enable the **Execute Steps In-Process (CVI is NOT Required for This Mode)** option and 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 14-2.

Figure 14-2. Auto-Load Library Configuration Dialog Box



Select one of the following actions in the Auto-Load Library Configuration dialog box to configure the support libraries:

- 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 Microsoft Windows security feature that prevents an application from executing dynamically loaded code. TestStand does not support calling LabWindows/CVI object and static library files from an executable with DEP enabled, such as a custom user interface, and returns 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. When the adapter finds the object file, it executes the code in the object file. When the adapter cannot find the object file, it prompts you to create the object file in an external instance of LabWindows/CVI. When you decline to create the object file, the adapter reports a run-time error.

Debugging DLL Code Modules

You can debug in-process code modules, but 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.

When you click **Step Into** in the sequence editor while the execution is 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 XP SP1 or earlier)** The current working directory of the application
3. The `Windows\System32` and `Windows\System` directories
4. The `Windows` directory
5. **(Windows XP SP2 or 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 XP SP1 or 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 or 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.

Per-Step Configuration of the LabWindows/CVI Adapter

You can direct TestStand to always run steps that use the LabWindows/CVI Adapter in-process. Enable the **Always Run In Process** option on the LabWindows/CVI Module tab to override 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.

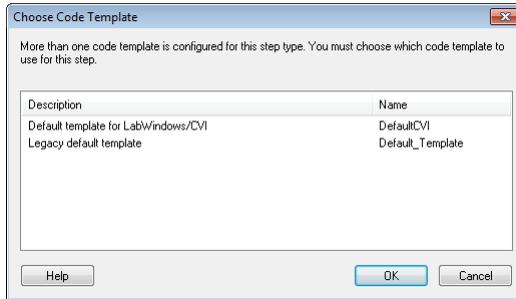
Code Template Policy

Use the Code Template Policy section in the LabWindows/CVI Adapter Configuration dialog box to allow the use of old, or legacy, code module templates when you create new test code modules. Legacy code module templates are files you can call from previous versions of TestStand. Refer to Chapter 18, [Calling Legacy LabWindows/CVI Code Modules](#), for more information about legacy code module templates.

The Code Template ring control on the LabWindows/CVI Module tab contains different values based on the code template policy setting. When you select the Allow Only New Templates option in the LabWindows/CVI Adapter Configuration dialog box, the ring control contains only the new code templates associated with the step type. When you select the Allow Only Legacy Templates option in the LabWindows/CVI Adapter Configuration dialog box, the ring control contains only the legacy code templates associated with the step type. When you select the Allow New and Legacy Templates option in the LabWindows/CVI Adapter Configuration dialog box, the ring control contains all the code templates associated with the step type.

When you use the Edit LabWindows/CVI Module Call dialog box to create code, such as in a custom sequence editor, and multiple code templates are available based on the step type and the code template policy settings, TestStand launches the Choose Code Template dialog box, in which you can select the code template to use for the new code module, as shown in Figure 14-3.

Figure 14-3. Choose Code Template Dialog Box



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

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.

TestStand User Interface Controls

Use the TestStand User Interface (UI) Controls in the LabWindows/CVI development environment to develop a custom user interface application, including custom sequence editors.

Creating and Configuring ActiveX Controls

In the LabWindows/CVI User Interface Editor, select **Create»ActiveX** and select a user interface control beginning with `TestStand UI` to add a TestStand UI Control to a panel. 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 the property pages the UI control supports.

Programming with ActiveX Controls

To access the methods, properties, and events specific to an ActiveX control, you must 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\instr\TestStand\API` directory.

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

- **TestStand UI Controls** (`tsui.fp`)—Functions for dynamically creating controls, calling methods and accessing properties on controls, and handling events from the controls.
- **TestStand UI Support Library** (`tsuisupp.fp`)—Functions for various collections the TestStand UI Controls driver uses.
- **TestStand Utility Functions** (`tsutil.fp`)—Utility functions for managing menu items that correspond to TestStand commands, localizing strings in user interfaces, making dialog boxes associated with LabWindows/CVI code modules modal to TestStand applications, and checking if 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 the control defines.

When you store ActiveX controls in `.uir` files, you do not need to use the creation functions the driver includes because 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 that 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 you to initialize a thread as apartment-threaded before you can use ActiveX controls in a program. When 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. When you use the `CA_InitActiveXThreadStyleForCurrentThread` function to initialize the thread yourself, you must use `COINIT_APARTMENTTHREADED` as the threading model.

Refer to Chapter 16, *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.

Example User Interfaces

Refer to the example user interfaces included with TestStand for more information about creating a 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 Public>\UserInterfaces` and `<TestStand>\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 do not use the TestStand UI Controls. 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

Table 15-1 lists the functions in the example user interface files that demonstrate configuring connections, commands, and other settings for the TestStand UI Controls.

Table 15-1. Functions in Examples for Configuring the TestStand UI Controls

Source File	Functions
<code><TestStand Public>\UserInterfaces\Simple\CVI\TestExec.c</code>	<code>SetupActiveXControls</code>
<code><TestStand Public>\UserInterfaces\Full-Featured\CVI\TestExec.c</code>	<code>GetActiveXControlHandles</code> <code>RegisterActiveXEventCallbacks</code> <code>ConnectTestStandControls</code> <code>ConnectStatusBarPanels</code> <code>RebuildMenuBar</code>

Enabling Sequence Editing

The TestStand UI Controls support Operator Mode and Editor Mode. Set the `ApplicationMgr.IsEditor` property to `True` for the Application Manager control to allow users to create and edit sequence files. You can also use 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;
}
```

Handling Variants

Several functions in the TestStand API return variants. If the return value of these functions is a reference to an ActiveX/COM object, you must cast these variants to a valid handle type. To do this properly in LabWindows/CVI, you must create a temporary variable to hold the value and then convert the value to the correct type of handle.

Use the following function call to convert an `IUnknown` reference to a `CAObjHandle`:

```
CA_CreateObjHandleFromInterface (IUnknownReference,
    &IID_IUnknown, 0, LOCALE_NEUTRAL, 0, 0, &CAObjToCreate);
```

Use the following function call to convert a variant to a `CAObjHandle`:

```
CA_VariantConvertToType(&VariantReference,
    CAVT_OBJHANDLE, &CAObjToCreate);
```

A common situation where you need to use this technique is when you handle UIMessages. The following example shows how you can extract the ActiveXData parameter of the custom user message as a property object to handle UIMessages:

```
HRESULT CVICALLBACK ApplicationMgr_OnUserMessage (
    CAObjHandle caServerObjHandle, void *caCallbackData,
    TSUIObj_UIMessage uiMsg)
{
    int error = 0;
    ErrMsg errMsg = "";
    ERRORINFO errorInfo;
    LPUNKNOWN tempPropObject = 0;
    CAObjHandle PropObject = 0;

    tsErrChk(TS_UIMessageGetActiveXData(uiMsg, NULL,
        &tempPropObject));
    CA_CreateObjHandleFromInterface (tempPropObject,
        &IID_IUnknown, 0, LOCALE_NEUTRAL, 0, 0, &PropObject);

    //TODO: Implement your code here

Error:
    //free resources
    if (PropObject)
        CA_DiscardObjHandle(PropObject);
    return error;
}
```

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 click the **Close** box or execute the Exit command through a TestStand menu or a Button control to stop a user interface application. 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. When the function determines that the TestStand Engine can shut down, the **canExitNow** output parameter returns `True`. The user interface application then calls 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 shut down the TestStand Engine.

Refer to the *NI TestStand Help* and to the *NI TestStand User Interface Controls Reference Poster* for more information about the TestStand UI Controls.

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 sequence 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 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.

Refer to the *NI TestStand Help* for more information about using manager controls.

Localization

The TestStand UI Controls and TestStand Utility Functions drivers 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.

Refer to the *NI TestStand Help* for more information about localizing all the user-visible TestStand UI Controls strings.

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`

Refer to the `<TestStand>\Components\StepTypes\MsgBox\msgbox.c` source file for an example of to use these functions.

Checking for Suspended or Stopped Execution within Code Modules

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

Code modules can also allow TestStand to suspend the parent execution without requiring the code modules to first return to TestStand.

Use the `TS_CancelDialogIfExecutionStops` function the TestStand Utility Functions driver provides to enable code modules that display dialog boxes to verify whether the execution that called the function has stopped.

Refer to the dialog box code in the following example source files for examples of how to use this function:

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

Use the `Execution.InitTerminationMonitor` and `Execution.GetTerminationMonitorStatus` methods to determine whether the execution receives a request to terminate or abort the execution. The Termination Monitor only recognizes requests to terminate or abort while monitoring, so a code module that executes in a

Cleanup step group of an already terminating execution monitors for a subsequent request to terminate the step or abort the execution.

Use the `Thread.ExternallySuspended` method to allow `TestStand` to suspend the execution thread at a breakpoint while still executing code in the code module.

Use the `Execution.GetStates` method to determine whether the execution is suspended.

Using the TestStand ActiveX APIs in LabWindows/CVI

You can use 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.

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 callback sequences 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 `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 `.fcp` 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 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.

When 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 chapter for more information about the `CA_DiscardObjHandle` function.

Some TestStand ActiveX API methods include 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. In LabWindows/CVI, you invoke methods on TestStand objects using the functions the ActiveX driver for those objects defines.

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 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 TestStand object has built-in properties that are static with respect to the TestStand API, which you can use to access the properties in the programming language you specify. The `Sequence.Name` and `SequenceContext.Sequence` properties are examples of built-in properties.

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 = NULL;
    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 named CVI Pass/Fail Test from a Sequence object:

```
int GetStepInSequence(CAObjHandle sequence)
{
    int error = 0;
    ErrMsg errMsg = "";
    ERRORINFO errorInfo;
    CAObjHandle step = 0;
    tsErrChk(TS_SequenceGetStepByName (sequence,
        &errorInfo, "CVI Pass/Fail Test", StepGroup_Main,
        &step));
Error:
    // Free Resources
    if (step)
        CA_DiscardObjHandle(step);
    return error;
}
```

Accessing Dynamic Properties

In TestStand, you can define custom step properties, sequence local variables, sequence file global variables, and station global variables. Because the TestStand API is independent 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 calling a method of the `PropertyObject` class on a handle to a `SequenceContext` object to set a local variable:

```
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. When 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, which adds 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 = NULL;
```

```

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 do not release the handle, LabWindows/CVI does not release the object for you. Repeatedly opening references to objects without closing them can cause the computer 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.

When you are concerned about a function returning a piece of data that must be manually released, refer to the *LabWindows/CVI Help* or to the *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 includes 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 sets a local variable only when the variable does not already exist:

```
int options = PropOption_DoNothingIfExists |
    PropOption_InsertIfMissing;

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

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 must register a callback function using the event callback registration functions in the instrument driver for an ActiveX control and then use the `CA_UnregisterEventCallback` function to close the callback before you close the application.

Refer to Chapter 15, *Creating Custom User Interfaces in LabWindows/CVI*, for more information about handling events that TestStand UI Controls generate.

Adding Type Libraries to LabWindows/CVI DLLs

When a DLL contains export information or when 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 the function names the DLL exports.

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 adds it in the Parameters Table control on the LabWindows/CVI Module tab.

You must enter the parameter information manually in the Parameters Table control for DLLs created with LabWindows/CVI 6.0 or earlier and for DLLs that do not have type library information.

Generating Type Library Information

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 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.
6. Click **OK** to close the Type Library dialog box and to close the Target Settings dialog box.
7. In the Project window, select **Build»Create Debuggable Dynamic Link Library** to build the DLL.

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, union, and enumeration 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.

Specifying String Buffer Size in a Type Library

Complete the following steps to embed string buffer size information in a type library in LabWindows/CVI so that TestStand automatically specifies the correct string buffer size for a string parameter.

1. In LabWindows/CVI, open a function panel (.fcp) file.
2. In the Function Tree Editor, select the function panel window that corresponds to the function you want to edit and select **Edit»Edit Function Panel Window** to launch the Function Panel Editor.
3. Complete the following steps to create a data type for the parameter. You must complete these steps only once for each .fcp file.
 - a. Select **Options»Data Types** to launch the Edit Data Type List dialog box.
 - b. Enter `char [1024]` in the Type control and click **Add**.
 - c. Click **Done** to close the Edit Data Type List dialog box.
4. On the function panel, select the control for the string parameter and select **Edit»Edit Control** to launch the Edit Input Control dialog box.
5. Select `char [1024]` from the Data type control and click **OK** to close the Edit Input Control dialog box.
6. Save the changes to the function panel file.
7. Select **Build»Create Debuggable Dynamic Link Library** to rebuild the DLL. Click **OK** when LabWindows/CVI prompts you that the files are created.

Calling Legacy LabWindows/CVI Code Modules

Versions of TestStand earlier than 3.0 required the DLL Flexible Prototype Adapter to call functions in LabWindows/CVI DLLs that did not use a specific prototype. Using TestStand 3.0 or later, you can call functions with a 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 compatibility with the LabWindows/CVI Test Executive Toolkit version 2.0 or 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)
```

Although you 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 14, *Configuring the LabWindows/CVI Adapter*, for more information about configuring the LabWindows/CVI Adapter to create 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, as shown in Table 18-1.

Table 18-1. 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 adapter copies this value into the <code>Step.Result.PassFail</code> property when the property exists.
measurement	double	Out	Numeric measurement the test function returns. The adapter copies this value into the <code>Step.Result.Numeric</code> property when the property exists.
inBuffer	char *	In	For passing a string parameter to a test function. The adapter copies the <code>Step.InBuf</code> property value into this field when the property exists.
outBuffer	char *	Out	Output message to include in the report. The adapter copies this message value into the <code>Step.Result.ReportText</code> property when the property exists.
modPath	char * const	In	Directory path of the module that contains the test function. The adapter sets this value before executing the code module.
modFile	char * const	In	Filename of the module that contains the test function. The 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 const	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.

Table 18-1. tTestData Structure Member Fields (Continued)

Field Name	Data Type	In/Out	Description
seqContextDisp	struct IDispatch *	In	Dispatch pointer to the sequence context. This value is <code>NULL</code> when you choose not to pass the sequence context.
seqContextCVI	CAObjHandle	In	LabWindows/CVI ActiveX Automation handle for the sequence context. This value is 0 when you choose not to pass the sequence context.
stringMeasurement	char *	Out	String value the test function returns. The adapter copies this string into the <code>Step.Result.String</code> property when the property exists.
replaceStringFuncPtr	tReplaceStringPtr r const	In	Contains a function pointer to <code>ReplaceString</code> , which a code module can use to reassign a value to the <code>inBuffer</code> , <code>outBuffer</code> , and <code>errorMessage</code> fields. The following is the <code>ReplaceString</code> prototype: <pre>int ReplaceString(char **destString, char *srcString);</pre> The function return value is non-zero when 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 contains only output error information, as shown in Table 18-2.

Table 18-2. tTestError Structure Member Fields

Field Name	Data Type	In/Out	Description
errorFlag	Boolean (int)	Out	The test function must set this value to <code>True</code> when an error occurs. The adapter copies this output value into the <code>Step.Result.Error.Occurred</code> property when 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 when an error occurs. The adapter copies the output value into the <code>Step.Result.Error.Code</code> property when the property exists and the errorFlag is set.
errorMessage	char *	Out	The test function can set this field to a descriptive string when an error occurs. The adapter copies the output value into the <code>Step.Result.Error.Msg</code> property when the property exists and the errorFlag is set.

Updating Step Properties

You can use the following methods to pass data between the code module and TestStand:

- Use the **tTestData** structure
- Use the sequence context ActiveX reference 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 18-3 lists all the properties the LabWindows/CVI Adapter updates and the value translation, if any, the adapter makes.

Table 18-3. Step Properties the LabWindows/CVI Adapter Updates

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 the sequence context ActiveX reference sets take precedence over the values the **tTestData** structure sets. When you use both methods to set the value of the same variable, TestStand recognizes the values the sequence context ActiveX reference sets and ignores the values the **tTestData** structure sets. You can use 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, when 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`, TestStand sets both values 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");
    }
}
```

Technical Support and Professional Services

Log in to your National Instruments ni.com User Profile to get personalized access to your services. Visit the following sections of ni.com for technical support and professional services:

- **Support**—Technical support at ni.com/support includes the following resources:
 - **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 eLearning training modules at ni.com/elearning. All customers automatically receive a one-year membership in the Standard Service Program (SSP) with the purchase of most software products and bundles including NI Developer Suite. NI also offers flexible extended contract options that guarantee your SSP benefits are available without interruption for as long as you need them. Visit ni.com/ssp for more information.

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 training and certification program information. You can also 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.

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

ActiveX

- adding references
 - (LabWindows/CVI), 16-4
- API, TestStand
 - LabVIEW, 9-1
 - LabWindows/CVI, 16-1
- controls (LabWindows/CVI)
 - configuring, 15-1
 - creating, 15-1
 - programming, 15-1
- LabWindows/CVI drivers, using, 16-1
- releasing references
 - LabVIEW, 9-3
 - LabWindows/CVI, 16-4

adapters. *See* module adapters

aliases file, 7-15

C

Check Remote System Status step type, 7-16

classes. *See* LabVIEW classes

cluster parameters (LabVIEW), 4-9

- cluster elements
 - mapping, updating, 4-12
 - specifying individually, 4-10
- container variable, passing as, 4-10
- data types, creating
 - custom, 4-11
 - TestStand (tutorial), 4-13

code modules (LabVIEW)

See also VIs

- COM references, duplicating in, 9-5
- LabVIEW projects, executing in, 7-3
- LabVIEW-based system, role in, 7-1
- relative paths, 7-9
- saving, 7-7
- suspended or stopped executions,
 - checking for, 6-8
- symmetric multiprocessing (SMP)
 - systems, optimizing performance for, 7-20

- TestStand API, accessing from, 9-2
- unique names for, 7-9

code modules (LabWindows/CVI)

- C source files, 14-4

debugging

- DLLs in-process, 14-4
- from TestStand (tutorial), 12-3
- in external instances, 14-2

executing

- in external instance, 14-2
- in-process, 14-3
- selecting where to execute, 14-2

legacy, 18-1

loading subordinate DLLs
(tutorial), 14-4

object files, 14-3

parameters

- object, 13-3
- pointer, 13-4
- string, 13-2
- struct, 13-3

static library files, 14-3

stopped, 15-7

suspended, 15-7

TestStand

- calling from, 11-1
- creating from (tutorial), 12-1
- debugging from (tutorial), 12-3
- editing from (tutorial), 12-3

code template policy

- LabVIEW, 5-4
- LabWindows/CVI, 14-5

COM references, duplicating
(LabVIEW), 9-5

Conditional Disable Structure, 7-11

constants

- LabVIEW, 9-3
- LabWindows/CVI, 16-6

container variables, passing to
LabVIEW, 4-10

Controls palette (LabVIEW), 6-1

Index

CPU affinity for LabVIEW execution
 threads, 7-18
 LabVIEW 2009 or later (table), 7-19
 LabVIEW 8.6.1 or earlier (table), 7-20
custom step types
 LabVIEW, 1-2
 step types that call VIs for substeps,
 creating, 7-12
 LabWindows/CVI, 1-2

D

data types (LabVIEW), 4-1
 creating
 custom for clusters, 4-11
 TestStand data types from clusters
 (tutorial), 4-13
 LabVIEW Class, 7-13
 TestStand equivalents
 controls and indicators (table), 4-2
 numeric representations (table), 4-1
data types (LabWindows/CVI), 13-1
 creating custom (tutorial), 13-4
 for struct parameters (tutorial), 13-4
 function with a struct parameter, calling
 (tutorial), 13-5
 structure passing settings, specifying
 (tutorial), 13-5
 TestStand equivalents (table), 13-1
debugging VIs from TestStand (tutorial), 3-5
Deploy Library step type, 7-5, 7-7, 7-14, 7-16
deploying shared variables, 7-14
 auto deploy and auto undeploy, 5-6
deploying VIs
 in LabVIEW packed project
 libraries, 7-5
 in LabVIEW projects, 7-3
 stand-alone, 7-7
deployment, building with LabVIEW, 7-11
diagnostic tools (NI resources), A-1
directory, 2-4, 11-3
DLLs (LabVIEW). *See* LabVIEW-built
 shared libraries (DLLs)

DLLs (LabWindows/CVI)
 adding type libraries, 17-1
 requirements for declaring in a type
 library, 17-2
documentation
 NI resources, A-1
drivers (NI resources), A-1

E

editing VIs
 in LabVIEW packed project
 libraries, 7-4
 in LabVIEW projects, 7-2
 in LabVIEW-built shared libraries
 (DLLs), 7-9
 stand-alone VIs, 7-6
Editor Mode, 6-4, 15-3
enumeration parameters in VIs, 4-5
 normalizing numeric values passed to
 enumeration parameter values, 4-7
 table, 4-7
updating, 4-5
 when enumeration is an instance of
 LabVIEW type definition
 (table), 4-6
 when enumeration is not an
 instance of LabVIEW type
 definition (table), 4-6
enumerations
 LabVIEW, 9-3
 LabWindows/CVI, 16-6
Error Out cluster, legacy, 10-3
events
 handling in user interfaces
 LabVIEW, 6-4
 LabWindows/CVI, 15-4, 16-7
 handling menu events in user interfaces
 (LabVIEW), 6-6
examples (NI resources), A-1

executions

- preferred execution system, setting (LabVIEW), 9-6
- suspended or stopped, checking for LabVIEW, 6-8
- LabWindows/CVI, 15-7
- threads. *See* symmetric multiprocessing (SMP) in VIs executed from TestStand

F

- function arguments, showing in step descriptions, 14-1

H

- help, technical support, A-1

I

- INI file tokens, configuring the number of LabVIEW execution threads with, 7-21
- valid execution system values (table), 7-22
- Input Buffer string control, legacy, 10-4
- instrument drivers (NI resources), A-1
- interfaces, obtaining for TestStand objects (LabVIEW), 9-4
- Invocation Info cluster, legacy, 10-4

K

- KnowledgeBase, A-1

L

- LabVIEW
 - classes, 7-13
 - member VI, calling (tutorial), 2-6
 - dynamically dispatched member method VI, 2-7
 - static member VI, 2-6
 - clusters, partially specifying, 7-10
 - Conditional Disable Structure, 7-11
 - custom user interfaces, creating, 6-1
 - data types. *See* data types (LabVIEW)

- execution threads. *See* symmetric multiprocessing (SMP) in VIs executed from TestStand

legacy VIs, 10-1

- Error Out cluster, 10-3
- format, 10-1
- Input Buffer string control, 10-4
- Invocation Info cluster, 10-4
- Sequence Context control, 10-4
- settings, 5-5
- Test Data cluster, 10-2

libraries, 7-8

localization, 6-7

Module tab, 2-1

- class and class member controls (figure), 2-2
- figure, 2-2

packed project libraries, 7-4

- custom step types that call VIs for substeps, using for, 7-12

VIs

- deploying, 7-5
- editing, 7-4
- updating, 7-5

preferred execution system, setting, 9-6

project libraries, 7-8

projects, 7-2

- creating a step that calls a VI in the context of a LabVIEW project (tutorial), 2-5
- deploying VIs, 7-3
- editing VIs, 7-2
- TestStand, creating from (tutorial), 3-2
- updating and patching VIs, 7-3

required settings, 2-1

servers

- LabVIEW Run-Time Engine, 5-3
- other executable, 5-3
- selecting, 5-1
- VI Server, configuring (tutorial), 8-2

shared libraries, LabVIEW-built

- (DLLs). *See* LabVIEW-built shared libraries (DLLs)

- shared variables, 7-14
 - aliases file, 7-15
 - deploying, 7-14
- stand-alone VIs, 7-6
 - deploying, 7-7
 - editing, 7-6
 - updating, 7-6
- TestStand
 - ActiveX APIs, 9-1
 - deployment, building, 7-11
 - special considerations for, 7-9
 - using effectively with, 7-1
- VIs. *See* VIs
- XControls, 7-11
- LabVIEW Adapter, 1-2
 - configuring, 5-1
 - member VI, calling (tutorial), 2-6
 - dynamically dispatched member method VI, 2-7
 - static member VI, 2-6
 - per-step configuration, 5-4
 - preferred execution system, setting, 9-6
 - sequence, executing (tutorial), 2-7
 - step, creating and configuring new (tutorial), 2-4
 - VI, calling in the context of a LabVIEW project (tutorial), 2-5
- LabVIEW Class data type, using with
 - TestStand, 7-13
- LabVIEW classes, 7-13
 - controls on LabVIEW Module tab (figure), 2-2
 - member VI, calling (tutorial), 2-6
 - dynamically dispatched member method VI, 2-7
 - static member VI, 2-6
- LabVIEW clusters, partially specifying, 7-10
- LabVIEW libraries (LLBs), 7-8
- LabVIEW packed project libraries, 7-4
 - VIs
 - deploying, 7-5
 - editing, 7-4
 - updating, 7-5
- LabVIEW project libraries, 7-8
- LabVIEW projects
 - creating a step that calls a VI in the context of a LabVIEW project (tutorial), 2-5
 - deploying VIs, 7-3
 - editing VIs, 7-2
 - TestStand, creating from (tutorial), 3-2
 - updating and patching VIs, 7-3
- LabVIEW Run-Time Engine execution
 - threads, 7-18
- LabVIEW systems, organizing, 7-1
- LabVIEW Utility step types, 7-16
 - Check Remote System Status, 7-16
 - Deploy Library, 7-5, 7-7, 7-14, 7-16
 - Run VI Asynchronously, 7-16
- LabVIEW-built shared libraries (DLLs), 7-8
 - creating, 7-8
 - editing VIs, 7-9
- LabWindows/CVI
 - custom user interfaces, creating, 15-2
 - data types. *See* data types (LabWindows/CVI)
 - legacy code modules, 18-1
 - example, 18-6
 - prototypes, 18-1
 - tTestData structure, 18-2
 - tTestError structure, 18-4
 - updating step properties, 18-4
 - localization, 15-6
 - Module tab, 11-1
 - required settings, 11-1
 - TestStand ActiveX APIs, 16-1
- LabWindows/CVI Adapter, 1-3
 - configuring, 14-1
 - per-step configuration, 14-5
 - step
 - creating and configuring new (tutorial), 11-3
 - selecting where to execute, 14-2
- legacy
 - code modules (LabWindows/CVI), 18-1
 - VIs. *See* LabVIEW legacy VIs
- localization
 - LabVIEW, 6-7
 - LabWindows/CVI, 15-6

M

- memory management, 7-9
- menu bars in user interfaces
 - LabVIEW, 6-6
 - LabWindows/CVI, 15-6
- modal dialog boxes in user interfaces
 - LabVIEW, 6-8
 - LabWindows/CVI, 15-7
- module adapters
 - LabVIEW, 1-2
 - configuring, 5-1
 - LabWindows/CVI, 1-3
 - configuring, 14-1
- Module tab
 - LabVIEW, 2-1
 - class and class member controls (figure), 2-2
 - figure, 2-2
 - LabWindows/CVI, 11-1

N

- National Instruments support and services, A-1
- NI-PSP URL, deploying shared variables with, 7-15

O

- object parameters (LabWindows/CVI), 13-3

P

- parameters
 - cluster (LabVIEW). *See* cluster parameters (LabVIEW)
 - object (LabWindows/CVI), 13-3
 - pointer (LabWindows/CVI), 13-4
 - ring control parameters (LabVIEW), 4-7
 - string. *See* string parameters
 - struct. *See* struct parameters (LabWindows/CVI)
- performance of LabVIEW code modules, optimizing on symmetric multiprocessing (SMP) systems, 7-20
- pointer parameters (LabWindows/CVI), 13-4

- preferred execution system
 - impact when executing VIs in TestStand, 7-17
 - options other than Same as caller, 7-18
 - Same as caller, 7-17
 - setting, 9-6
 - user interface, using to call VIs from TestStand (note), 7-17
- programming examples (NI resources), A-1
- properties, accessing
 - built-in TestStand properties
 - LabVIEW, 9-1
 - LabWindows/CVI, 16-2
 - dynamic TestStand properties
 - LabVIEW, 9-2
 - LabWindows/CVI, 16-3
- PropertyObject class, acquiring derived class in LabVIEW, 9-5
- prototypes, legacy (LabWindows/CVI), 18-1

R

- relative paths, 7-9
- remote computers, calling VIs on, 8-1
 - configuring a step to run remotely (tutorial), 8-1
- reusing VIs, 7-9
- ring control parameters (LabVIEW), 4-7
- Run VI Asynchronously step type, 7-16

S

- Sequence Context control, legacy, 10-4
- sequences, executing with LabVIEW steps (tutorial), 2-7
- shared variables, 7-14
 - aliases file, 7-15
 - deploying, 7-14
 - bound shared variables using NI-PSP URL, 7-15
- software (NI resources), A-1
- step properties, updating for legacy LabWindows/CVI code modules, 18-4

Index

step types

custom

LabVIEW, 1-2

LabWindows/CVI, 1-2

LabVIEW Utility, 7-16

Check Remote System Status, 7-16

Deploy Library, 7-5, 7-7, 7-14, 7-16

Run VI Asynchronously, 7-16

steps

creating and configuring (tutorial)

LabVIEW, 2-4

LabWindows/CVI, 11-3

function arguments, showing

(LabWindows/CVI), 14-1

updating properties for legacy

LabWindows/CVI code

modules, 18-4

stopped executions

LabVIEW, 6-8

LabWindows/CVI, 15-7

string buffer size, specifying in type libraries

(tutorial), 17-2

string parameters

LabWindows/CVI code modules, 13-2

VI, 4-8

struct parameters (LabWindows/CVI), 13-3

custom data types, creating

(tutorial), 13-4

default packing size, setting, 14-2

functions, calling (tutorial), 13-5

support, technical, A-1

suspended executions

LabVIEW, 6-8

LabWindows/CVI, 15-7

symmetric multiprocessing (SMP) in VIs

executed from TestStand, 7-16

code module performance,

optimizing, 7-20

execution threads, 7-16

configuring the number of, 7-17

INI file tokens, 7-21

LabVIEW 2009 or later

(table), 7-17

LabVIEW 8.6.1 or earlier

(table), 7-17

valid execution system values

(table), 7-22

CPU affinity for, 7-18

LabVIEW 2009 or later

(table), 7-19

LabVIEW 8.6.1 or earlier

(table), 7-20

LabVIEW Run-Time Engine, 7-18

preferred execution system

impact when executing VIs in

TestStand, 7-17

options other than Same as

caller, 7-18

Same as caller, 7-17

user interface, using to call VIs

from TestStand (note), 7-17

T

technical support, A-1

Termination Monitor, 6-8, 15-7

Test Data cluster, legacy, 10-2

TestStand

ActiveX API

LabVIEW, 9-1

LabWindows/CVI, 16-1

container variables, passing to

LabVIEW, 4-10

<TestStand Public> directory, 2-4, 11-3

user interfaces

shutting down in

LabVIEW, 6-6

LabWindows/CVI, 15-5

starting in

LabVIEW, 6-6

LabWindows/CVI, 15-5

TestStand API

accessing

built-in properties

LabVIEW, 9-1

LabWindows/CVI, 16-2

dynamic properties

LabVIEW, 9-2

LabWindows/CVI, 16-3

ActiveX drivers for

LabWindows/CVI, 16-1

- COM references, duplicating (LabVIEW), 9-5
 - constants
 - LabVIEW, 9-3
 - LabWindows/CVI, 16-6
 - derived class, acquiring from PropertyObject class in LabVIEW, 9-5
 - enumerations
 - LabVIEW, 9-3
 - LabWindows/CVI, 16-6
 - LabVIEW, 9-1
 - LabWindows/CVI, 16-1
 - methods, invoking
 - LabVIEW, 9-1
 - LabWindows/CVI, 16-2
 - TestStand objects, obtaining interfaces for (LabVIEW), 9-4
 - TestStand User Interface (UI) Controls
 - configuring
 - LabVIEW, 6-4
 - LabWindows/CVI, 15-3
 - LabVIEW, 6-1
 - LabWindows/CVI, 15-1
 - training and certification (NI resources), A-1
 - troubleshooting (NI resources), A-1
 - tTestData structure, legacy, 18-2
 - tTestError structure, legacy, 18-4
 - type libraries
 - LabWindows/CVI, generating in (tutorial), 17-1
 - string buffer size, specifying (tutorial), 17-2
- U**
- updating VIs
 - in LabVIEW packed project libraries, 7-5
 - in LabVIEW projects, 7-3
 - stand-alone VIs, 7-6
 - user interfaces
 - creating
 - LabVIEW, 6-2
 - LabWindows/CVI, 15-2
 - custom
 - LabVIEW, 1-2, 6-1
 - LabWindows/CVI, 1-2, 15-1
 - Editor Mode, 6-4, 15-3
 - events, handling
 - LabVIEW, 6-4
 - LabWindows/CVI, 15-4, 16-7
 - example
 - LabVIEW, 6-2
 - caveats, 6-2
 - LabWindows/CVI, 15-2
 - LabVIEW, running in, 6-9
 - localization
 - LabVIEW, 6-7
 - LabWindows/CVI, 15-6
 - menu bars
 - LabVIEW, 6-6
 - LabWindows/CVI, 15-6
 - menu events, handling (LabVIEW), 6-6
 - modal dialog boxes
 - LabVIEW, 6-8
 - LabWindows/CVI, 15-7
 - sequence editing, enabling
 - LabVIEW, 6-4
 - LabWindows/CVI, 15-3
 - shutting down TestStand in
 - LabVIEW, 6-6
 - LabWindows/CVI, 15-5
 - starting TestStand in
 - LabVIEW, 6-6
 - LabWindows/CVI, 15-5
 - suspended or stopped executions, checking for
 - LabVIEW, 6-8
 - LabWindows/CVI, 15-7
 - variants, handling (LabWindows/CVI), 15-4

V

variables, deploying (LabVIEW), 7-14

variants, handling in user interfaces
(LabWindows/CVI), 15-4

VI Server, configuring (tutorial), 8-2

VIs

calling

from TestStand, 2-1

on remote computers, 8-1

cluster parameters, 4-9

code modules. *See* code modules

compiled code, executing with, 7-10

custom step types that call VIs for
substeps, creating, 7-12

debugging from TestStand (tutorial), 3-5

editing from TestStand (tutorial), 3-4

enumeration parameters, 4-5

normalizing numeric values passed
to enumeration parameter
values, 4-7

table, 4-7

updating, 4-5

when enumeration is an
instance of LabVIEW type
definition (table), 4-6

when enumeration is not an
instance of LabVIEW type
definition (table), 4-6

LabVIEW Adapter, configuring to
run, 2-1

LabVIEW Real-Time module, 8-1

legacy, 10-1

settings, 5-5

preferred execution system, setting, 9-6

remote access, configuring, 8-3

remotely running VIs with VI

Server, 8-2

reserving, 7-10

reusing, 7-9

ring control parameters, 4-7

stand-alone

deploying, 7-7

editing, 7-6

updating, 7-6

stopped, 6-8

string parameters, 4-8

suspended, 6-8

symmetric multiprocessing (SMP). *See*
symmetric multiprocessing (SMP) in
VIs executed from TestStand

TestStand, creating from (tutorial), 3-1

VIs and Functions palette (LabVIEW), 6-1

W

Web resources, A-1

X

XControls, 7-11