# NI TestStand™

## Using LabVIEW™ and LabWindows™/CVI™ with TestStand

May 2009
375070A-01

**Worldwide Technical Support and Product Information**

ni.com

**National Instruments Corporate Headquarters**

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

**Worldwide Offices**

Australia 1800 300 800, Austria 43 662 457990-0, Belgium 32 (0) 2 757 0020, Brazil 55 11 3262 3599, Canada 800 433 3488, China 86 21 5050 9800, Czech Republic 420 224 235 774, Denmark 45 45 76 26 00, Finland 358 (0) 9 725 72511, France 01 57 66 24 24, Germany 49 89 7413130, India 91 80 41190000, Israel 972 3 6393737, Italy 39 02 41309277, Japan 0120-527196, Korea 82 02 3451 3400, Lebanon 961 (0) 1 33 28 28, Malaysia 1800 887710, Mexico 01 800 010 0793, Netherlands 31 (0) 348 433 466, New Zealand 0800 553 322, Norway 47 (0) 66 90 76 60, Poland 48 22 328 90 10, Portugal 351 210 311 210, Russia 7 495 783 6851, Singapore 1800 226 5886, Slovenia 386 3 425 42 00, South Africa 27 0 11 805 8197, Spain 34 91 640 0085, Sweden 46 (0) 8 587 895 00, Switzerland 41 56 2005151, Taiwan 886 02 2377 2222, Thailand 662 278 6777, Turkey 90 212 279 3031, United Kingdom 44 (0) 1635 523545

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

© 2009 National Instruments Corporation. All rights reserved.

# Important Information

## Warranty

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

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

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

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

## Copyright

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.

## Trademarks

National Instruments, NI, ni.com, NI TestStand, and LabVIEW are trademarks of National Instruments Corporation. Refer to the *Terms of Use* section on `ni.com/legal` for more information about National Instruments trademarks.

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`.

## WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

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

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

# Conventions

The following conventions are used in this manual:

» — The » symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options** directs you to pull down the **File** menu, select the **Page Setup** item, and select **Options** from the last dialog box.

(Note icon) — This icon denotes a note, which alerts you to important information.

**bold** — Bold text denotes items that you must select or click in the software, such as menu items and dialog box options. Bold text also denotes parameter names.

*italic* — Italic text denotes variables, emphasis, a cross-reference, or an introduction to a key concept. Italic text also denotes text that is a placeholder for a word or value that you must supply.

`monospace` — Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames, and extensions.

*`monospace italic`* — Italic text in this font denotes text that is a placeholder for a word or value that you must supply.

**Platform** — Text in this font denotes a specific platform and indicates that the text following it applies only to that platform.

# Contents

## Chapter 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 ........................................ 2-2
Creating and Configuring a New Step Using the LabVIEW Adapter ........................................ 2-3


## Chapter 3
## Creating, Editing, and Debugging LabVIEW VIs from TestStand


Creating a New VI from TestStand ........................................ 3-1
Editing an Existing VI from TestStand ........................................ 3-2
Debugging a VI ........................................ 3-3


## Chapter 4
## Using LabVIEW Data Types with TestStand


Calling VIs with String Parameters ........................................ 4-4
Calling VIs with Cluster Parameters ........................................ 4-4
    Specifying Each Cluster Element Individually ........................................ 4-5
    Passing Existing TestStand Container Variables to LabVIEW ........................................ 4-5
    Creating a New Custom Data Type ........................................ 4-6
Creating TestStand Data Types from LabVIEW Clusters ........................................ 4-8

## Chapter 5
## Configuring the LabVIEW Adapter

Selecting a LabVIEW Server ........ 5-1
Using a LabVIEW Run-Time Engine or Other Executable Server ........ 5-2
Using a LabVIEW 8.6.*x* or Later Development System ........ 5-4
Per-Step Configuration of the LabVIEW Adapter ........ 5-4
Reserving Loaded VIs for Execution ........ 5-4
Code Template Policy ........ 5-5
Legacy VI Settings ........ 5-7

## 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
Configuring the TestStand UI Controls ........ 6-3
Enabling Sequence Editing ........ 6-4
Handling Events ........ 6-4
Starting and Shutting Down TestStand ........ 6-6
Menu Bars and Menu Event Handling ........ 6-7
Localization ........ 6-8
Other User Interface Utilities ........ 6-8
Making Dialog Boxes Modal to TestStand ........ 6-8
Checking for Suspended or Stopped Executions within Code Modules ........ 6-9
Running User Interfaces ........ 6-9

## Chapter 7
## Using LabVIEW 8.*x* with TestStand

Using LabVIEW 8.0 ........ 7-1
LabVIEW 8.0 Real-Time Module Incompatibility ........ 7-1
Projects ........ 7-1
Project Libraries ........ 7-2
Network-Published Shared Variables ........ 7-2
Deploying Variables ........ 7-2
Using an Aliases File ........ 7-3
NI-DAQmx Tasks, Channels, and Scales in LabVIEW Projects ........ 7-3
Conditional Disable Structures and Symbols ........ 7-4
64-Bit Integer Data Type ........ 7-4
XControls ........ 7-4
Remote Execution ........ 7-4
Building a TestStand Deployment with LabVIEW 8.0 ........ 7-5
LabVIEW Object-Oriented Programming ........ 7-7

## 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
Configuring a LabVIEW RT Server to Run VIs....8-3
User Access to VI Server....8-4

## 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....9-4
Obtaining a Different Interface for TestStand Objects....9-5
Acquiring a Derived Class from the PropertyObject Class....9-5
Duplicating COM References in LabVIEW Code Modules....9-6
Setting the Preferred Execution System for LabVIEW VIs....9-7
Handling Events....9-8

## 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....10-5

# 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....11-3
Creating and Configuring a New Step Using the LabWindows/CVI Adapter....11-4

## 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
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 .................................................... 14-2
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 .............................................................. 14-3
Executing Code Modules In-Process ............................................................ 14-3
Object and Library Code Modules ................................................. 14-3
Source Code Modules...................................................................... 14-5
Debugging DLL Code Modules ..................................................... 14-5
Loading Subordinate DLLs............................................................................ 14-5
Per-Step Configuration of the LabWindows/CVI Adapter............................ 14-6
Code Template Policy ................................................................................................. 14-7

## 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.................................................................................. 15-3
Configuring the TestStand UI Controls ......................................................... 15-4
Enabling Sequence Editing ........................................................................... 15-4

Handling Events ........ 15-4
Handling Variants ........ 15-5
Starting and Shutting Down TestStand ........ 15-6
Menu Bars ........ 15-7
Localization ........ 15-8
Other User Interface Utilities ........ 15-8
Making Dialog Boxes Modal to TestStand ........ 15-8
Checking for Suspended or Stopped Execution within Code Modules ........ 15-8

## 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 ........ 16-3
Accessing Dynamic Properties ........ 16-4
Adding and Releasing References ........ 16-5
Using TestStand API Constants and Enumerations ........ 16-7
Handling Events ........ 16-8

## Chapter 17
## Adding Type Libraries To LabWindows/CVI DLLs

Generating Type Library Information ........ 17-1

## Chapter 18
## Calling Legacy LabWindows/CVI Code Modules

Prototypes of Legacy Code Modules ........ 18-1
tTestData Structure ........ 18-2
tTestError Structure ........ 18-3
Updating Step Properties ........ 18-4
Example Code Module ........ 18-5

## Appendix A
## Technical Support and Professional Services

## Index

# 1 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. 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 Chapter 9, *Creating Custom User Interfaces*, of the *NI TestStand Reference Manual* for more information about creating custom user interfaces.

# 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 Chapter 13, *Custom Step Types*, of the *NI TestStand Reference Manual* 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 perform the following tasks:

- Call VIs with arbitrary connector panes
- Call VIs on remote computers
- Run VIs in the LabVIEW Run-Time Engine
- Call VIs from versions of TestStand earlier than 3.0 and LabVIEW Test Executive VIs
- Create and edit VIs from TestStand
- Debug VIs (step in/step out) from TestStand
- Run VIs using the LabVIEW development system or a LabVIEW executable

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 perform the following tasks:

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

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, 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, *Using LabVIEW 8.x with TestStand*—TestStand supports LabVIEW 8.*x* features, with some limitations. Additional requirements exist for building a TestStand deployment system that includes LabVIEW 8.*x* 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.

# 2

# Calling LabVIEW VIs from TestStand

Use the LabVIEW Adapter to call VIs from TestStand.

## Required LabVIEW Settings

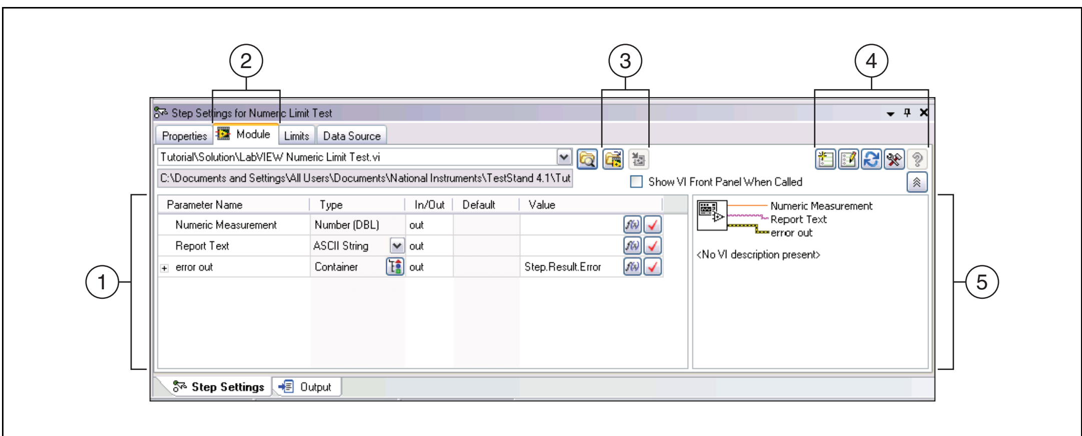
All the tutorials in this manual require you to have the LabVIEW development system and TestStand installed on the same computer. In addition, you must configure the LabVIEW Adapter to run VIs using the LabVIEW development system. Refer to Chapter 5, *Configuring the LabVIEW Adapter*, for more information 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. By default LabVIEW allows access to all VIs. To view the VI Server: Exported VIs list, select **Tools» Options** and select the **VI Server: Exported VIs** category in the Options dialog box.
- If you use LabVIEW 8.0.1 or earlier, select **Tools»Options» Performance and Disk** to confirm the **Run with multiple threads** checkbox includes a checkmark to avoid errors when running VIs.

# 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.




| | | | |
|---|---|---|---|
| 1 | VI Parameter Table | 4 | Source Code and Help Buttons |
| 2 | Parameters Table Control | 5 | VI Context Help Image and Description |
| 3 | Express VI Buttons | | |

**Figure 2-1.** LabVIEW Module Tab

Use the LabVIEW Module tab in the TestStand Sequence Editor to specify the VI the step executes and to specify if LabVIEW shows the front panel of the VI when TestStand calls the VI. The Module tab includes Source Code buttons for selecting an Express VI and converting an Express VI to a standard VI.

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

- **VI Parameter Table**—Contains the following information about each control and indicator, also called the parameters of the VI, wired to the connector pane of the VI:
  - **Parameter Name**—The caption text for the control or indicator. When no caption exists, this field contains the label text.
  - **Type**—The LabVIEW 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) or an output (indicator).
- **Default**—Specifies if TestStand uses the default value for the parameter, cluster element, or array element. When the terminal on the VI is marked as Required, this option is not available.
- **Value**—A TestStand expression. For input parameters, TestStand passes the result of this expression to the VI unless you enable the checkbox in the Default column. For output parameters, TestStand stores the data the VI returns in the location this expression specifies.

**Note** Parameters are listed in the VI Parameter Table according to their order in the VI context help image.

- **Source Code and Help Buttons**—Use these buttons to create or edit a VI in LabVIEW, refresh the parameter information for the VI, open the LabVIEW Advanced Settings window, display the help associated with the VI if available, and undock the *LabVIEW Help*.
- **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 page in 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 **Help** or **Help Topic** button on the Help toolbar to access the *NI TestStand Help*, which provides additional information about 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 test VI.

1. Launch the TestStand Sequence Editor and select the **LabVIEW Adapter** on the Insertion Palette.
2. Open a new Sequence File window if one is not already open.
3. Select **File»Save As** and save the sequence file as `<TestStand Public>\Tutorial\Call LabVIEW VI.seq`. The `<TestStand Public>` directory is located at `C:\Documents and Settings\`

`All Users\Documents\National Instruments\TestStand` *`x.x`* on Windows 2000/XP and at `C:\Users\Public\Documents\National Instruments\TestStand` *`x.x`* on Windows Vista.

4. Insert a Pass/Fail step in the Main step group of the Sequence File window and rename the new step `LabVIEW Pass/Fail Test`.
5. On the LabVIEW Module tab of the Step Settings pane, click the **Browse for VI** button, 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.

7. 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.

8. Select **File»Save** to save the sequence file.
9. Select **Execute»Single Pass** to run the sequence file using the Single Pass Execution entry point. When the execution completes, the resulting report indicates the step passed. The VI always returns `True` as the **Pass/Fail** output parameter.
10. Close the Execution window.

# 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.

## 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*.
3. Insert a Numeric Limit Test step after the LabVIEW 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 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 update step properties by passing step properties as parameters to and from the module or by using the TestStand API in the module. 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 TestStand 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`.
8. Leave the sequence file open so you can use it in the next tutorial.

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

# Editing an Existing 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 VI from TestStand* section of this chapter.
2. 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.
3. Open the block diagram for the VI and change the **PASS/FAIL Flag** Boolean constant to `False`.
4. Save and close the VI.
5. In the TestStand Sequence Editor, select **Execute»Single Pass** to run the sequence file using the Single Pass Execution entry point. When the execution completes, the resulting report indicates the step failed, and the VI returns `False` in the **PASS/FAIL Flag** indicator.
6. Close the Execution window.

# Debugging a VI

Complete the following steps to debug a VI 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.
3. Save the sequence file and select **Execute»Run MainSequence** to start an execution of `MainSequence`.
4. When the execution pauses, click the **Step Into** button on the sequence editor toolbar. LabVIEW becomes the active application, in which the LabVIEW Pass-Fail Test VI is open and in a suspended state.
5. Open the block diagram of the suspended VI.
6. Click the **Step Into** button or the **Step Over** button on the LabVIEW toolbar to begin single-stepping through the VI. You can click the **Continue** button at any time to finish single-stepping through the VI.
7. When you finish single-stepping through the VI, click the **Return to Caller** button on the LabVIEW toolbar to return to TestStand. The execution pauses at the next step in the sequence.
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 the VI multiple times before returning to TestStand. However, LabVIEW passes the results only from the last run to TestStand when you finish debugging.

# 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. 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, as shown in Table 4-1.

**Table 4-1.** TestStand Equivalents for LabVIEW Data Types

| LabVIEW Data Type | TestStand Data Type |
|---|---|
| Real number (U8, U16, U32, I8, I16, I32, SGL, DBL, or EXT) | Number<br>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 | TestStand does not support calling VIs with 64-bit integer numeric indicators or controls. |
| Fixed-point numeric | TestStand does not support calling VIs with fixed-point numeric indicators or controls. |
| Complex number (CSG, CDB, or CXT) | Number<br>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 4-1.** TestStand Equivalents for LabVIEW Data Types (Continued)

| LabVIEW Data Type | TestStand Data Type |
|---|---|
| Enum (U32, U16, or U8) | Number<br><br>For input parameters, the Value column on the LabVIEW Module tab shows a ring control that contains the items in the LabVIEW enumeration.<br><br>At edit time, TestStand stores the numeric value and string label value of the enum you select. When you pass an enum value to TestStand at run time, TestStand stores only the numeric value. |
| String | String<br><br>Refer to the *Calling VIs with String Parameters* 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<br><br>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. If you use LabVIEW 7.1.1, the objects must be marshalable by reference. |
| Waveform | LabVIEWAnalogWaveform |
| Digital waveform | LabVIEWDigitalWaveform |
| Digital data | LabVIEWDigitalData |
| Picture | String<br><br>You must select Binary String on the LabVIEW Module tab. Refer to the *Calling VIs with String Parameters* section of this chapter for more information about using the string data type. |

**Table 4-1.** TestStand Equivalents for LabVIEW Data Types (Continued)

| LabVIEW Data Type | TestStand Data Type |
|---|---|
| Refnum (File I/O, VI, Menu, Queue, TCP connection, and so on) | Number<br><br>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. |
| Timestamp | String<br><br>Refer to the *Calling VIs with String Parameters* section of this chapter for more information about using the string data type. |
| Error I/O | Error<br><br>When a VI contains the standard **error out** cluster as an output parameter, TestStand automatically detects it and maps the output to `Step.Result.Error`. |
| Array of *x* | Array of TestStand (*x*) |
| Variant | Any TestStand data type |
| LabVIEW Object | TestStand does not support calling VIs with LabVIEW Object indicators or controls. |
| Cluster | Container<br><br>Refer to the *Calling VIs with Cluster Parameters* section of this chapter for more information about using the container data type. |
| I/O data types (DAQmx Task Name, DAQmx Channel Name, VISA Resource Name, IVI Logical Name, FieldPoint IO Point, or Motion Resource) | LabVIEWIOControl<br><br>Refer to the *NI TestStand Help* for more information about using the DeviceName and SessionNumber properties of the LabVIEWIOControl data type. |

**Table 4-1.** TestStand Equivalents for LabVIEW Data Types (Continued)

| LabVIEW Data Type | TestStand Data Type |
|---|---|
| IMAQ Session | Number |
| Other I/O data types (DAQmx Physical Channel Name, Terminal Name, Analog Trigger Source, Scale Name, Device Name, or Switch Name) | String<br>Refer to the *Calling VIs with String Parameters* section of this chapter for more information about using the string data type. |

# Calling VIs with String Parameters

When you configure calls to VIs that use strings as parameters, you can specify to escape the string data when reading the data from the VI or unescape the string data 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 escapes the string before storing it and substitutes hexadecimal codes for the unprintable characters, such as the NUL character, in the string.

To pass a string escaped 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.

# 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.

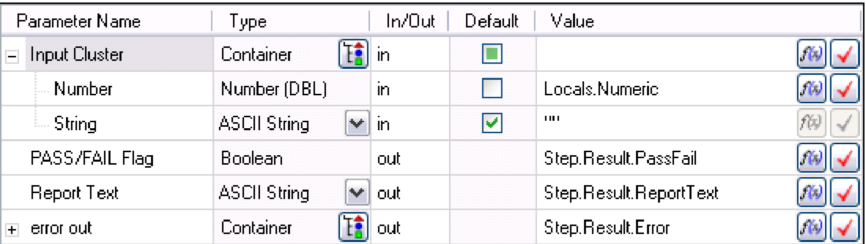
You can create a custom data type that matches a LabVIEW cluster. For input parameters, you can pass the default value for the entire cluster or the default values for specific elements of the cluster control on the front panel of the VI.

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 4-1 illustrates a VI Parameter Table in which the data source for the Number element of **Input Cluster** is a local variable. TestStand passes the default value for the String element of **Input Cluster**.



| Parameter Name | Type | In/Out | Default | Value |
|---|---|---|---|---|
| Input Cluster | Container | in | | |
| Number | Number (DBL) | in | | Locals.Numeric |
| String | ASCII String | in | | "" |
| PASS/FAIL Flag | Boolean | out | | Step.Result.PassFail |
| Report Text | ASCII String | out | | Step.Result.ReportText |
| error out | Container | out | | Step.Result.Error |

**Figure 4-1.** Input Cluster Data Sources

## 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 4-2 shows how the data passed to the **Input Cluster** parameter is a local variable called ContainerData with a type of InputData. Figure 4-3 shows the custom InputData data type.

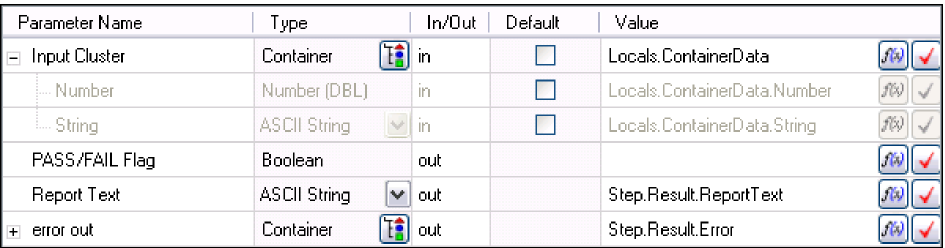



**Figure 4-2.** ContainerData Local Variable

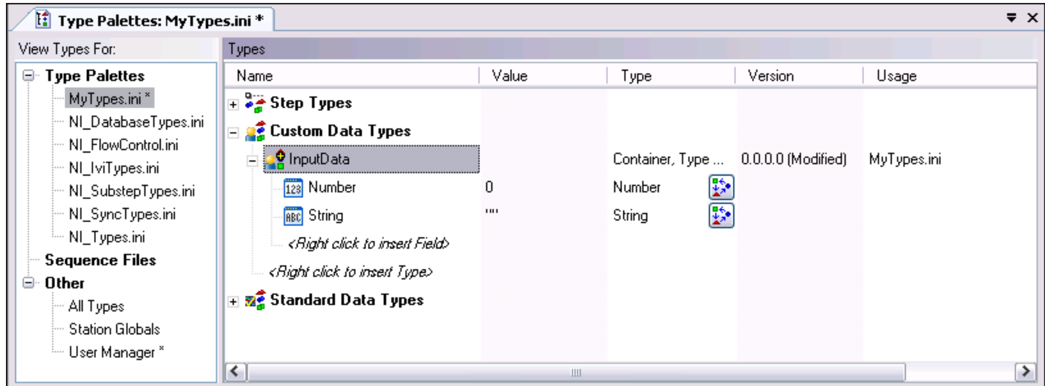



**Figure 4-3.** TestStand Custom InputData Data Type

## 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, shown at left, 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 4-4.

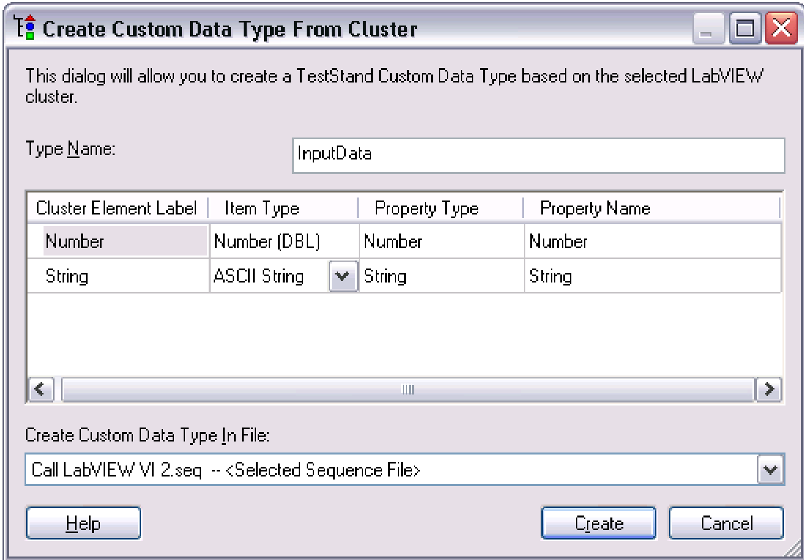



**Figure 4-4.** Create Custom Data Type From Cluster Dialog Box

- **Type Name**—Specify the name of the TestStand custom data type you want to create.
- **Property Type**—Specify the TestStand data type to use for cluster elements that are variants.
- **Property Name**—Specify the names of the subproperties that map to the elements of the cluster.
- **Create Custom Data Type In File**—Specify where TestStand creates the type.

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 Chapter 11, *Type Concepts*, of the *NI TestStand Reference Manual* for more information about where TestStand stores custom data types. Refer to Chapter 12, *Standard and Custom Data Types*, of the *NI TestStand Reference Manual* and to the *NI TestStand Help* for more information about custom data types. Refer to the *NI TestStand Help* for more information about the Create Custom Data Type From Cluster dialog box.

# 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. Select the **LabVIEW Adapter** on the Insertion Palette.
3. Insert a new Pass/Fail Test step in the Main step group of `MainSequence` after the LabVIEW Numeric Limit Test step and rename the step `Pass Container to VI`.
4. On the LabVIEW Module tab, click the **Browse for VI** button and select `<TestStand Public>\Tutorial\VI with Cluster Input.vi`. The **Report Text** output parameter of this VI returns a string that contains the number and string elements of the **Input Cluster** parameter.
5. 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.
6. 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.
7. 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. Right-click **Locals** in the Variables/Properties tab 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** field 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** field and click **OK**.

f. Enter `Locals.ContainerData` in the **Expression** field and click **OK**. The Value column for the **Input Cluster** parameter now contains `Locals.ContainerData.String`.

8. 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.
9. Save the changes.
10. 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.
11. Close the Execution window.

# 5

# Configuring the LabVIEW Adapter

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

## 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,

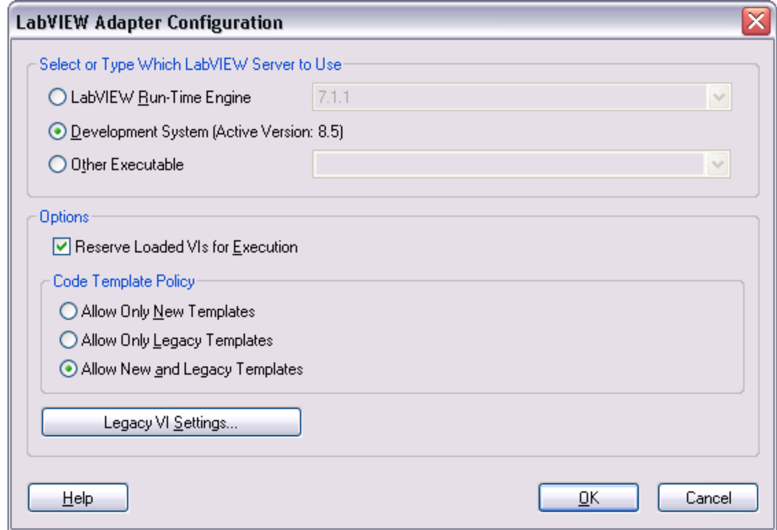



**Figure 5-1.** LabVIEW Adapter Configuration Dialog Box

- **LabVIEW Run-Time Engine**—Provides optimal performance when calling VIs by running VIs in the same process as TestStand. 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.
- **Development System (Active Version: *X.X*)**—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 8.0 or later or the Build Application or Shared Library (DLL) functionality in LabVIEW 7.1.1. 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 server name associated with the LabVIEW executable. The executable must be installed and registered on the same computer as TestStand. Launch the executable once to register it as an ActiveX server. Refer to the `<TestStand Public>\Components\RuntimeServers\LabVIEW` directory for an example server VI and application build script for LabVIEW 8.0 or later and for LabVIEW 7.1.1.

**Note (Windows Vista)** Windows Vista requires you to log in as a user with administrator privileges to register a server. LabVIEW cannot register ActiveX servers on Windows Vista when building executables without elevation. When you use `TestStandLVRTS.exe` with Windows Vista to run VIs and you receive an error because the server is not registered, run `TestStandLVRTS.exe` as an administrator to register the server.

## 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 Chapter 14, *Deploying TestStand Systems*, of the *NI TestStand Reference Manual* for more information about deploying VIs for use with TestStand and about

including a LabVIEW RTE in a deployment installer. Refer to Chapter 7, *Using LabVIEW 8.x with TestStand*, for more information about calling and deploying LabVIEW 8.*x* VIs.

When 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. This discrepancy occurs most often when a top-level VI uses a subVI or controls located in the `vi.lib\addons\TestStand` directory and that subVI was saved in a different version of LabVIEW than the LabVIEW RTE. Mass compiling the `vi.lib\addons\TestStand` directory usually resolves the discrepancy. You can also mass compile top-level VIs, which in turn compiles any subVIs. When you use the TestStand Version Selector to activate a different version of TestStand, TestStand copies new versions of the VIs to the `vi.lib\addons\TestStand` directory. 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. Therefore, when TestStand executes a top-level VI saved without its hierarchy, the LabVIEW RTE attempts to find subVIs by first searching the directory of the top-level VI and then searching the directories TestStand added. When the LabVIEW RTE finds a subVI saved by a different version of LabVIEW first, the LabVIEW RTE cannot load the subVI and does not load the top-level VI.

## Using a LabVIEW 8.6.*x* or Later Development System

When you install a version of LabVIEW later than LabVIEW 8.6.*x* and you want to use that version of LabVIEW with TestStand, you must complete the following steps to update `TestStand - Default Values 86.llb` to allow the LabVIEW Adapter to retrieve the default parameter values of the VIs.

1. Create a copy of `TestStand - Default Values 86.llb`.
2. Rename the copy `TestStand - Default Values xx.llb`, where `xx` indicates the version of LabVIEW, with out a revision number, you want to use with TestStand.
3. Mass compile `TestStand - Default Values xx.llb` using the version of LabVIEW you want to use with TestStand.

## 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 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.

# 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 reduce the amount of time required for TestStand to call the VIs and makes 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.

Although reserving VIs with this option reduces the amount of time required for TestStand to call the VIs, it also blocks other applications from using any VIs TestStand loads, including subVIs of the VIs TestStand calls directly.

When you open a reserved VI in LabVIEW, the Run arrow, shown at left, indicates the VI is reserved, and you cannot edit the VI. To edit a VI TestStand has reserved, click the **Edit Code** button, shown at left, 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 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 could result in a memory leak in the test system.

## 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. The 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 enable the Allow New and Legacy Templates option 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 enable the Allow Only Legacy Templates option 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 enable the Allow Only New Templates option 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.

# 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. Refer to Chapter 9, *Creating Custom User Interfaces*, of the *NI TestStand Reference Manual* for more information about the TestStand User Interface (UI) Controls.

## TestStand User Interface Controls

Use the TestStand UI Controls on the **Controls»TestStand** palette to develop a custom user interface application, including custom sequence editors.

When you place the 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 Property Browser or control property pages if available. Right-click the control and select **Property Browser** from the context menu to open the LabVIEW Property Browser. Right-click the control and select **Properties** from the context menu to open the property page.

Refer to Chapter 9, *Using the TestStand ActiveX APIs in LabVIEW*, for more information about programming the TestStand API from LabVIEW.

## TestStand VIs and Functions

The TestStand VIs and functions on the **Functions»TestStand** palette 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 code module VI has stopped
- Pausing the calling execution thread when a lengthy operation suspends a code module VI
- Setting and getting 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.

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.

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** When you place a TestStand UI Control on the front panel of a VI, you must set the Preferred Execution System to **user interface** for the VI. In addition, 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.

## 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 TestStand Application Manager control.

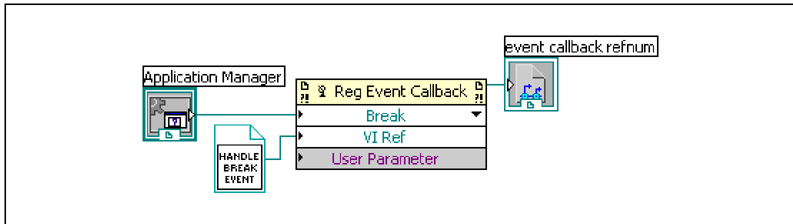



**Figure 6-1.** Registering a Callback VI for the Break Event

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

Start TestStand by invoking the `ApplicationMgr.Start` method, 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 stop a user interface application by clicking the **Close** box or by executing the **Exit** command through a TestStand menu or a Button control.

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 executes the appropriate TestStand command by calling the TestStand - Execute Menu Command VI. 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.

# 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 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 terminated or aborted 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 parent execution without requiring the code modules to first return to TestStand.

Use the following VIs to enable VIs that TestStand calls to verify whether the execution that called the VI has stopped:

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

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 parent execution thread 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 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 Web site at `ni.com/info` and enter the info code `4H6DULT3` to access the National Instruments KnowledgeBase article 4H6DULT3 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.

# 7

# Using LabVIEW 8.*x* with TestStand

Refer to the LabVIEW documentation for more information about the LabVIEW features included in this chapter.

## Using LabVIEW 8.0

LabVIEW 8.0 introduced support for many new features, such as projects, project libraries, and DAQmx tasks. As a result of some of these changes, TestStand 3.1 or earlier is not compatible with LabVIEW 8.0. Use TestStand 3.5 or later to use LabVIEW 8.0 with TestStand.

The following sections include the LabVIEW 8.0 features TestStand supports, limitations of the TestStand support, and additional requirements to build a TestStand deployment system that includes LabVIEW 8.0 VIs.

### LabVIEW 8.0 Real-Time Module Incompatibility

TestStand is not compatible with the LabVIEW 8.0 Real-Time (RT) module and cannot download VIs to the LabVIEW 8.0 RT module. Refer to Chapter 8, *Calling LabVIEW VIs on Remote Computers*, for more information about executing VIs in the LabVIEW 8.0 RT module.

### Projects

When you run a VI in LabVIEW 8.0, the VI runs inside the application instance for a target in a LabVIEW project or the VI runs without a project in the main application instance. In TestStand, the LabVIEW Adapter always runs a VI in the main application instance, and LabVIEW 8.0 project settings do not apply. TestStand does not support LabVIEW project features or settings unless otherwise noted in the following sections.

## Project Libraries

LabVIEW project libraries are collections of VIs, type definitions, 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.

## Network-Published Shared Variables

LabVIEW 8.0 supports 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.

Because TestStand executes VIs without a project, LabVIEW 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 LabVIEW Utility Deploy Library step type in TestStand to deploy a project library that contains shared variables. The project library can contain only shared variables and cannot contain any VI files. You must install Remote Execution Support for NI TestStand to use the LabVIEW Utility Deploy Library step type. Refer to the *NI TestStand Help* for more information about the LabVIEW Utility Deploy Library step type.

**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 standalone executable without adverse effects on TestStand.

## Using an Aliases File

When you deploy a project library or execute a VI in TestStand that accesses a shared variable, 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.

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. For example, `SeqEdit.aliases` for the TestStand 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.

# NI-DAQmx Tasks, Channels, and Scales in LabVIEW Projects

TestStand does not support VIs that use NI-DAQmx tasks, channels, and scales defined in a project. You must define NI-DAQmx tasks, channels, and scales in Measurement & Automation Explorer (MAX).

## Conditional Disable Structures and Symbols

LabVIEW 8.0 includes the Conditional Disable Structure, which contains one or more subdiagrams, or cases. Depending on the configuration, LabVIEW uses one of the subdiagrams for the duration of the execution. You can specify conditions for the structure based on custom symbols defined in a project. Because TestStand executes VIs in the main application instance, any conditions that use custom symbols always evaluate to `False`.

## 64-Bit Integer Data Type

TestStand does not support calling VIs that contain terminals connected to 64-bit integer numeric controls or indicators.

## XControls

TestStand supports calling VIs that use XControls on Windows systems.

## Remote Execution

To execute VIs on a remote LabVIEW computer, TestStand requires a version of the Remote Execution Support for NI TestStand component that matches the version of LabVIEW with which you saved the VIs on the local computer. LabVIEW installs and enables this component only when TestStand is present. When you install TestStand after you install LabVIEW, TestStand installs a version of the component that might not match the version of LabVIEW on the computer. In this case, you might need to rerun the LabVIEW installer.

Also, you must install a version of the Remote Execution Support for NI TestStand component that matches libraries that you deploy using the LabVIEW Utility Deploy Library step type.

## Building a TestStand Deployment with LabVIEW 8.0

TestStand 4.0 or later supports deploying LabVIEW 8.0 VIs and VIs from project libraries. However, the following restrictions exist that do not apply to earlier versions of LabVIEW:

- National Instruments does not recommend editing packaged VIs on a deployed system because unexpected errors can occur. TestStand includes only required VIs from a project library in a deployment. When you attempt to add new VIs from a project library to the deployment image, LabVIEW might not be able to find all the VIs it needs when you run the new VIs. National Instruments recommends rebuilding and redeploying the deployment image in this situation. Analysis and instrument drivers are examples of VIs that use project libraries.
- National Instruments does not recommend deploying VIs that were previously deployed using the TestStand Deployment Utility. When the TestStand 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 TestStand 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 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.

- 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 subVI. 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 no longer supports distributing duplicate DLLs that VIs call. TestStand 4.0 or later supports—but National Instruments does not recommend—distributing duplicate DLLs that steps in a sequence file call.
- TestStand does not support deploying duplicate project libraries. You receive an error when you attempt to include two project libraries with the same name.
- TestStand no longer supports including two VIs with the same name in a deployment unless the VIs are in different project 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.

- 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 TestStand 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.
  - Deploy shared variables to the local computer by executing a LabVIEW Utility Deploy Library step in a TestStand sequence. You can also manually deploy shared variables to the local computer using a project in the LabVIEW development environment.

# LabVIEW Object-Oriented Programming

LabVIEW 8.2 or later includes support for Object-Oriented Programming. TestStand cannot directly call a VI that wires a LabVIEW object to its connector pane. However, you can wire the LabVIEW object to the Flatten to String function, return the data string to TestStand as a binary string, and store the flattened LabVIEW object in a string property or variable in TestStand.

You cannot access the properties or invoke the methods on the flattened LabVIEW string object in TestStand. To access the properties or invoke the methods on the LabVIEW object, you must pass the flattened LabVIEW object as a binary string to a VI and wire the string to the Unflatten From String function along with the object constant of the correct type to create the LabVIEW object.

# 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. TestStand supports downloading VIs to systems that run the LabVIEW 7.1.1 RT module but does not support downloading VIs to systems that run the LabVIEW 8.0 or later RT module. To use the LabVIEW 8.0 or later 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, Solaris (LabVIEW 7.1.1 only), and Mac OS.

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, use the Remote VI Path text box to specify the path to the VI on the remote computer.

If the remote computer is a PXI controller running LabVIEW 7.1.1 RT, TestStand downloads the VI to the remote computer or loads the VI using the Remote VI Path you specified in the LabVIEW Advanced Settings window. TestStand skips this step when the VI is already present in memory on the controller at the time TestStand loads the code module for the step.

If the remote computer is a PXI controller running LabVIEW 8.0 RT or later, TestStand does not download the VI to the remote computer. You can use the LabVIEW 8.0 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.

**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.

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. Disable the options to exclude VIs from `vi.lib`, `instr.lib`, and `user.lib`. 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 8.0 or greater RT module, you can use the Remote VI Path option in the LabVIEW Advanced Settings window to call the VIs.

# 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.

In LabVIEW 7.1.1, select **Tools»Options** and navigate to the VI Server: Configuration settings to enable the TCP/IP protocol and the VI calls 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, which you can access on the LabVIEW Module tab.

Use the VI Server: TCP/IP or Machine Access settings 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 VI Server: Exported VIs settings 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, indicated by an asterisk (*).

In LabVIEW 8.0 or later, you can make changes to the LabVIEW VI Server settings using LabVIEW projects. You must enable the TCP/IP Protocol and specify the Machine Access, User Access, and Exported VIs settings.

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

# Configuring a LabVIEW RT Server to Run VIs

You can configure a LabVIEW RT Server to run VIs remotely.

For a LabVIEW 7.1.1 RT Server, launch LabVIEW on the host computer, select the appropriate RT target, and select **Tools»RT Target <*IP Address/Host Name*> Options**. Configure the VI Server settings to specify which VIs can run remotely and which computers can access the RT target remotely.

When you finish configuring the RT target, set the execution target back the to the local computer before you attempt to use TestStand to call VIs on the RT target.

In addition, you must also configure the RT target to allow access to the computer running TestStand when you want TestStand to download VIs to the RT target.

For a LabVIEW 8.0 or later RT Server, launch LabVIEW on the host computer, right-click the appropriate RT target in the Project Explorer window, and select **Properties** from the context menu to launch the target properties dialog box. Configure the VI Server settings to specify which VIs can run remotely and which computers can access the RT target remotely.

Refer to the *Configuring RT Target Properties* section of the *Getting Started with the LabVIEW Real-Time Module* manual in the `<LabVIEW>\manuals` directory or to the *LabVIEW Help* for more information about configuring VI Server settings.

## User Access to VI Server

TestStand does not support user-based VI security for executing VIs on remote computers. You must use the VI Server Machine access list to protect a VI server on a remote computer.

# 9

# Using the TestStand ActiveX APIs in LabVIEW

You can use the TestStand API or TestStand 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.

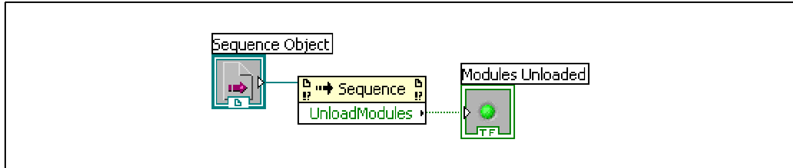



**Figure 9-1.** Invoking the UnloadModules Method

## 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.

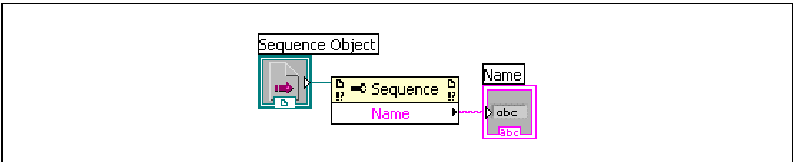



**Figure 9-2.** Obtaining the Value of the Name Property from a Sequence Object

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

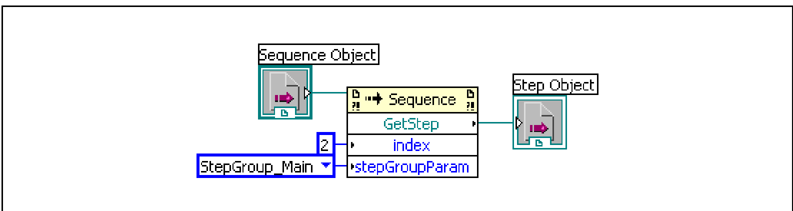



**Figure 9-3.** Obtaining a Reference to a Step of a Sequence that a Sequence Object References

# 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 by using 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.

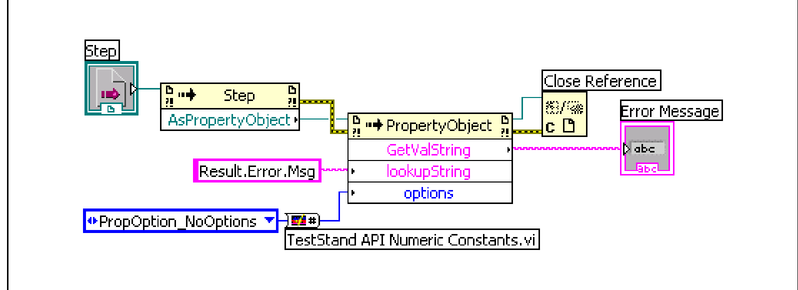



**Figure 9-4.** Using the GetValString Method to Obtain the Error Message Value for the Current Step

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.

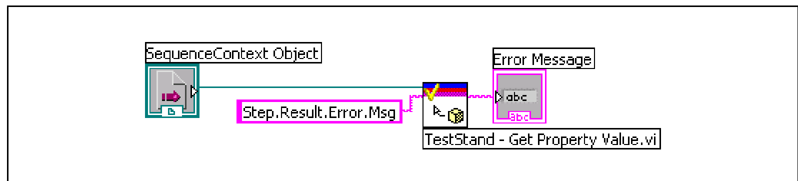



**Figure 9-5.** Using VIs to Obtain the Error Message Value for the Current Step

# 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.

It can be difficult to remember all the available string and numeric constants for the TestStand API properties and methods. To facilitate programming with the TestStand API within VIs, 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 methods in the TestStand API 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 get 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.

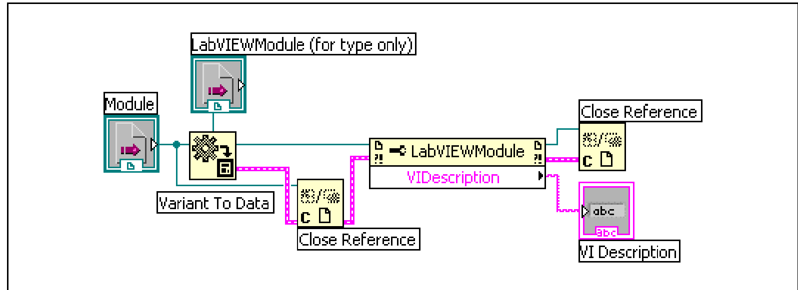



**Figure 9-6.** Converting a Module Reference to a LabVIEWModule Reference

# 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 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.

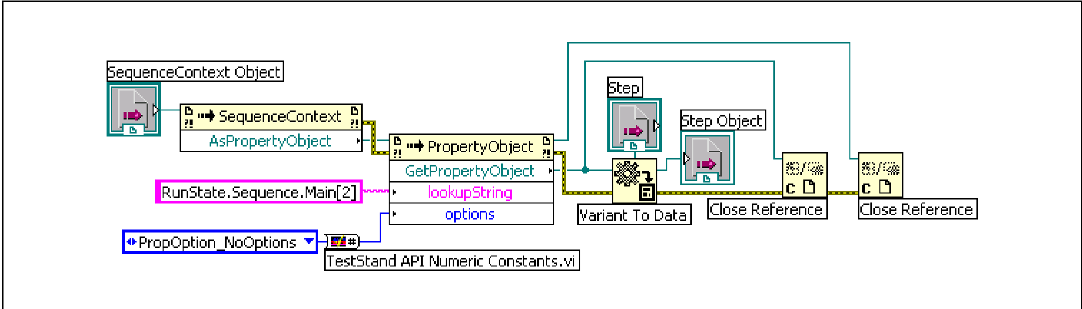



**Figure 9-7.** Using a Lookup String to Obtain a Reference to a Step Object from a SequenceContext Object

# 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.

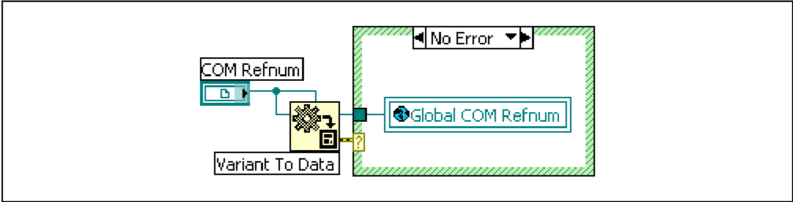



**Figure 9-8.** Creating Duplicate COM Reference

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, close the reference by reading the global variable and wiring the reference to the LabVIEW Close Reference function. Clear the global variable by wiring the global variable to the LabVIEW Not A Refnum constant, as shown in Figure 9-9.

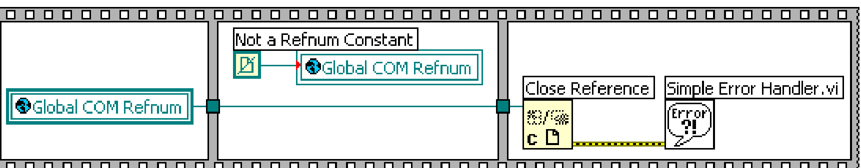



**Figure 9-9.** Closing Duplicate COM Reference

# Setting the Preferred Execution System for LabVIEW VIs

When a VI calls synchronous methods of the TestStand API, you must correctly set the LabVIEW Preferred Execution System 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 be set to run in the same LabVIEW execution system. When the VIs are set to run in the same execution system, a deadlock occurs because the execution of the synchronous TestStand method consumes the execution system in which the VI needs to run.

Because LabVIEW handles ActiveX communication through its 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 its preferred execution system. In addition, both VIs in this scenario must use different preferred execution system settings. Use the VI Properties dialog box for each VI to configure the LabVIEW execution system.

# Handling Events

TestStand User Interface (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.

# 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 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** | TF | The test VI sets this element to indicate whether the test passed. Valid values are `True` (PASS) or `False` (FAIL). The adapter copies the value into the `Step.Result.PassFail` property when the property exists. |
| **Numeric Measurement** | DBL | Numeric measurement the test VI returns. The adapter copies this value into the `Step.Result.Numeric` property when the property exists. |
| **String Measurement** | abc | String value the test VI returns. The adapter copies this string into the `Step.Result.String` property when the property exists. |
| **Report Text** | abc | Output message to include in the report. The adapter copies this message value into the `Step.Result.ReportText` property when the property exists. |

The LabVIEW Adapter also supports an older 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 two string elements, **Comment** and **User Output**.

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

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

| Cluster Element | Data Type | Description |
|---|---|---|
| **Comment** | abc | Output message to display in the report. The adapter copies this message value into the `Step.Result.ReportText` property when the property exists. |
| **User Output** | abc | String value the test VI returns. The adapter dynamically creates the step property `Step.Result.UserOutput` 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** | TF | The test VI must set this element to `True` when an error occurs. The adapter copies the output value into the `Step.Result.Error.Occurred` property when the property exists. |
| **code** | I32 | The test VI can set this element to a non-zero value when an error occurs. The adapter copies the output value into the `Step.Result.Error.Code` property when the property exists. |
| **source** | abc | The test VI can set this element to a descriptive string when an error occurs. The adapter copies the output value into the `Step.Result.Error.Msg` 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** | abc | The adapter uses the name of the step that invokes the test VI. |
| **loop #** | I32 | 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** | abc | The adapter uses the value from the `RunState.Root.Locals.UUT.SerialNumber` property when the property exists. Otherwise, the adapter copies an empty string. Refer to the *NI TestStand Help* for more information about how to configure the Serial Number String option in the Legacy VI Settings dialog box. |
| **UUT #** | I32 | The adapter uses the value from the `RunState.Root.Locals.UUT.UUTLoopIndex` property when the property exists. Otherwise, the adapter copies an empty string. Refer to the *NI TestStand Help* 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.

# Part II

# 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.

# 11

# Calling LabWindows/CVI Code Modules from TestStand

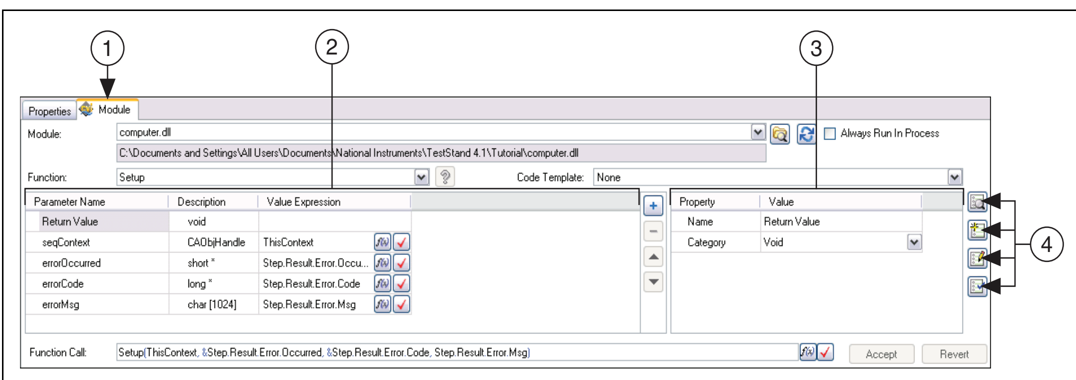
Use the LabWindows/CVI Adapter to call LabWindows/CVI code modules from TestStand.

## Required LabWindows/CVI Settings

All the tutorials in this manual require you to have LabWindows/CVI and TestStand installed on the same computer. In addition, you must configure the LabWindows/CVI Adapter to execute steps in an external instance of the LabWindows/CVI development system and to allow only new code templates. Refer to Chapter 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.




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

**Figure 11-1.** LabWindows/CVI Module Tab

The LabWindows/CVI Adapter supports calling functions in C source files, object files, static library files, and dynamic link library (DLL) files. The Module tab includes Source Code buttons for creating and editing code modules in LabWindows/CVI.

**Note** National Instruments recommends using DLL files when you develop code modules using the LabWindows/CVI Adapter. The tutorials in this manual demonstrate creating and debugging only DLL code modules. Refer to Chapter 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:

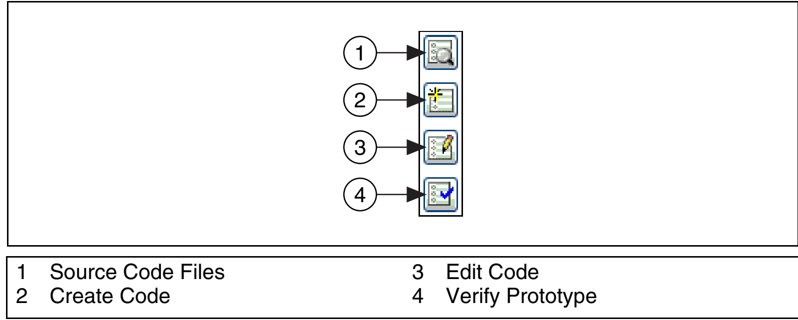
- **Parameter Name**—The symbolic name for the parameter.
- **Description**—The short description of the parameter type using C syntax.
- **Value Expression**—The argument expression to pass.

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.




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

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

The **Source Code Files** button launches the CVI Source Code Files dialog box, in which you can specify the source file that contains the function the step calls and to specify the project to use when editing the source code. When the code module is a DLL or static library, you must enter the name of the LabWindows/CVI project used to create the DLL or static library file. When the code module is an object file, you can optionally specify a project.

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

**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`.

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

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

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

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

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

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

**Note** When you select a function, the LabWindows/CVI Adapter attempts to read the export information LabWindows/CVI includes in the DLL or the function parameter information from the type library in the code module if one exists. 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 specify the function prototype by adding parameters to the Parameters Table control.

7. Select **PassFail template for LabWindows/CVI** in the Code Template ring control. The Parameters Table control contains the default value expressions 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. Select **File»Save** to save the sequence file.
9. Select **Execute»Single Pass** to run the sequence file using the Single Pass Execution entry point. Because the LabWindows/CVI Adapter is configured to use an external instance of LabWindows/CVI to execute code modules, TestStand launches the LabWindows/CVI development environment to execute the function the step calls. When the execution completes, the resulting report indicates the step passed. The code module always returns `True` as the **Pass/Fail** output parameter.
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.

# 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.

## 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*.
3. Insert a Numeric Limit Test step after the CVI Pass/Fail Test step and rename it `CVI Numeric Limit Test`.
4. Select the **CVI Numeric Limit Test** step and use the LabWindows/CVI Module tab 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** in 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.

5. Click the **Source Code Files** button to launch the CVI Source Code Files dialog box 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**.
6. 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.
7. 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 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 update step properties by passing step properties as parameters to and from the module or by using the TestStand API in the module. 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.

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

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

9. Save and close the source file. Leave LabWindows/CVI open.
10. In the LabWindows/CVI project window, select **Build»Create Debuggable Dynamic Link Library** to rebuild the DLL.
11. Return to the TestStand Sequence Editor and save the sequence file as `<TestStand Public>\Tutorial\CallCVICodeModule2.seq`.
12. Select **Execute»Single Pass** to run the sequence file using the Single Pass Execution entry point. When the execution completes, the resulting report indicates the step passed with a numeric measurement of `10.0`.

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

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

# Editing an Existing Code Module 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.
5. Rebuild the DLL.
6. In the TestStand Sequence Editor, select **Execute»Single Pass** to run the sequence file using the Single Pass Execution entry point. When the execution completes, the resulting report indicates the step failed, and the code module returns `5` in the Measurement field.
7. Close the Execution window.

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

**Note** If LabWindows/CVI does not launch, the LabWindows/CVI Adapter is not configured to execute steps in an external instance. To configure the LabWindows/CVI Adapter, complete all executions and select **Configure»Adapters**. Select **LabWindows/CVI** and click the **Configure** button. In the Step Execution section, select **Execute Steps in an External Instance of CVI**. Click **OK** to close the LabWindows/CVI Adapter Configuration dialog box. Click **OK** again when the adapter displays a warning that confirms all modules will be unloaded. 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 single-stepping through the code module. You can click the **Continue** button at any time to finish single-stepping through the code module.
6. When you finish single-stepping through the code module, click the **Finish Function** button on the LabWindows/CVI toolbar to return to TestStand. The execution pauses at the next step in the sequence.
7. Select **Debug»Resume** in TestStand to complete the execution.
8. Select **File»Unload All Modules** to unload the DLL.
9. Close the Execution window.

# 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. 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 Data Types

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

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

| LabWindows/CVI C Data Type | TestStand Data Type |
|---|---|
| Array of *x* | Array of TestStand (*x*) |
| struct | Container<br><br>Refer to the *Calling Code Modules with Struct Parameters* section of this chapter for more information about using the container data type. |

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

# Calling Code Modules with String Parameters

When you configure calls to code modules that use strings as parameters, you 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 `NULL` to a string pointer parameter by passing an empty object reference or the constant `Nothing`.

# Calling Code Modules with Object Parameters

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

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

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.

# Calling Code Modules with Struct Parameters

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

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

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

# Creating TestStand Data Types from LabWindows/CVI Structs

Complete the following steps to create a TestStand data type that matches a LabWindows/CVI struct and call a function in a DLL that has the struct as a parameter.

## 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. Right-click in the Sequence File window and select **View»Types** from the context menu. Make sure the `CallCVICodeModule2.seq` sequence file is selected in 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 inside the Types window under 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 inside the Types window under 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. Leave the sequence file open and continue to the next tutorial.

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

## Specifying Structure Passing Settings

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. Make sure 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. Leave the sequence 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.
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: MainSequence** pane.
5. Select the **LabWindows/CVI Adapter** on the Insertion Palette.
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. Click the **LabWindows/CVI Module** tab on the Step Settings pane.

8. Click the **File Browse** button next to the Module control and select the `<TestStand Public>\Tutorial\CallCVICodeModule.dll` file.
9. Enter `PassStructTest` in the Function control.

10. Click the **Add Parameter** button, shown at left, to insert a new parameter and enter the following information in the Parameter Details Table control:
    a. In the **Name** field, rename the parameter `cviStruct`.
    b. In the **Category** field, select `C Struct`.
    c. In the **Type** field, select `CVITutorialStruct`.
11. Enter `Locals.CVIStruct` in the Value Expression field for the parameter in the Parameters Table control.
12. 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. In the Source File Containing Function control, enter `CVIStructPassingTest.c`.
    b. Click the **File Browse** button next to the CVI Project File to Open option and select the `<TestStand Public>\Tutorial\CallCVICodeModule.prj` file.
    c. Click **Close**.
13. Click the **Create Code** button to create a code module. TestStand launches the Select a Source File dialog box.
14. Browse to the `<TestStand Public>\Tutorial` directory and click **OK**. TestStand creates a new source file named `CVIStructPassingTest.c` with an empty function.
15. In LabWindows/CVI, complete the following steps.
    a. Add the following type definition before the first 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>
```

16. Save and close the source file.
17. In the LabWindows/CVI project window, select **Build»Create Debuggable Dynamic Link Library** to rebuild the DLL.
18. Return to the TestStand Sequence Editor.
19. Place a breakpoint on the new Pass Struct Test step.
20. Save the sequence file.
21. Select **Execute»Run MainSequence** to start a new execution of `MainSequence`.
22. Single-step through the sequence and review the values in the `Locals.CVIStruct` variable before and after executing the new step.
23. Select **File»Unload All Modules** to unload the DLL.
24. Close the Execution window.

# 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.

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.

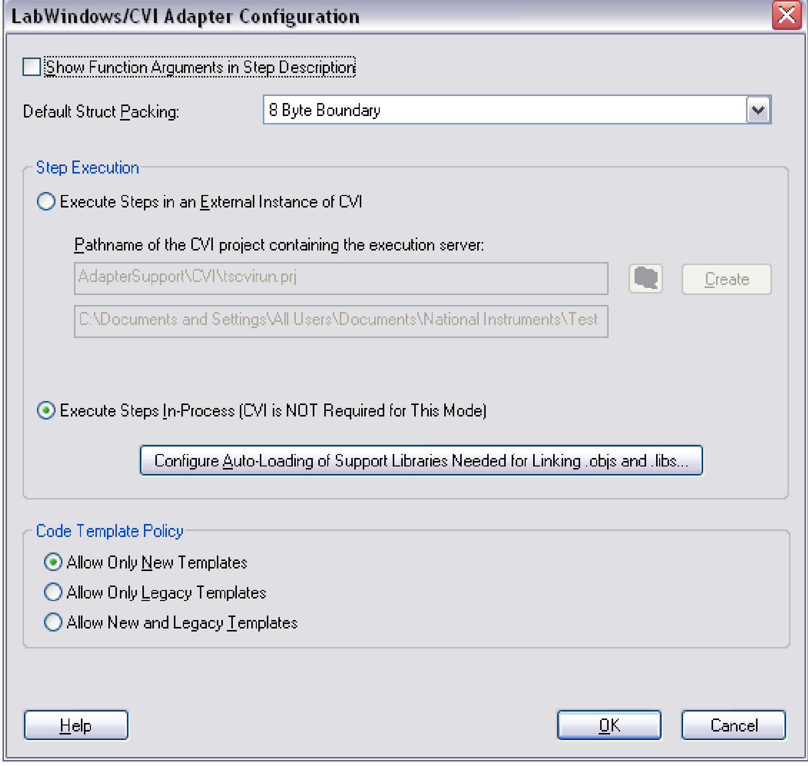



**Figure 14-1.** LabWindows/CVI Adapter Configuration Dialog Box

# 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 sequence editor and 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 control to specify the default setting for how the LabWindows/CVI Adapter packs structure parameters it passes. The following options are available: 1-, 2-, 4-, 8-, and 16-byte boundaries.

The compatibility mode of the LabWindows/CVI development environment you use to create DLLs determines the structure packing value. For LabWindows/CVI, the default structure packing can be 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 TestStand step calls a function in an object, static library, or DLL file, the execution server project automatically loads the code module and executes the function in an external instance of LabWindows/CVI. When you want a TestStand step to call a function in a C source file, you must include the C source file in the execution server project before you run the project. You must also include any support libraries other than LabWindows/CVI libraries the object, static library, or C source file requires.

### Debugging Code Modules

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

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

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

## Executing Code Modules In-Process

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

### Object and Library Code Modules

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

To configure a list of support libraries for the LabWindows/CVI Adapter to load, manually copy the support libraries to the `<TestStand Public>\AdapterSupport\CVI\AutoLoadLibs` directory. You can also click the **Configure Auto-Loading of Support Libraries Needed for**

**Linking .objs and .libs** button in the LabWindows/CVI Adapter Configuration dialog box to launch the Auto-Load Library Configuration dialog box, as shown in Figure 14-2.

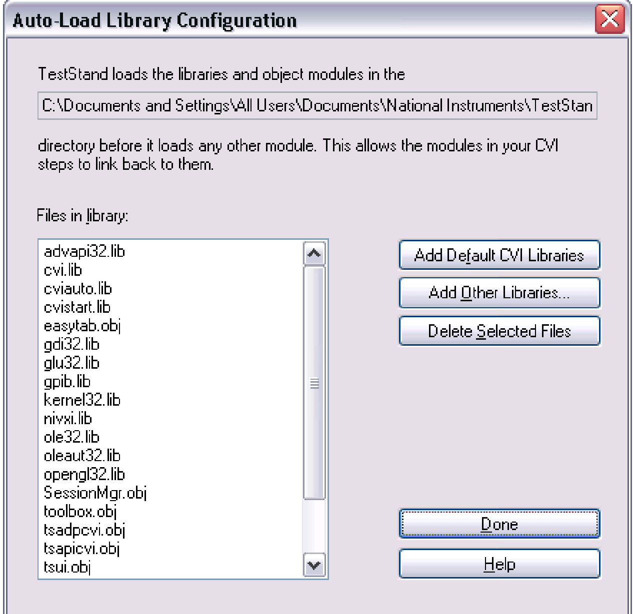



**Figure 14-2.** Auto-Load Library Configuration Dialog Box

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

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

**Note** Data Execution Prevention (DEP) is a Windows security feature that prevents an application from executing dynamically loaded code. 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 TestStand 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 2000 and 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 2000 and 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.

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

## Per-Step Configuration of the LabWindows/CVI Adapter

You can direct TestStand to always run steps that use the LabWindows/CVI Adapter in-process. 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. The 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 enable the Allow Only New Templates option, the ring control contains only the new code templates associated with the step type. When you enable the Allow Only Legacy Templates option, the ring control contains only the legacy code templates associated with the step type. When you enable the Allow New and Legacy Templates option, 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.

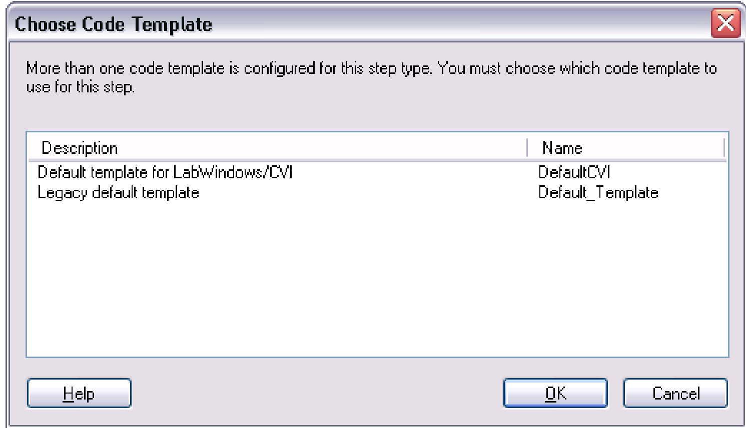



**Figure 14-3.** Choose Code Template Dialog Box

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

# 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. Refer to Chapter 9, *Creating Custom User Interfaces*, of the *NI TestStand Reference Manual* for more information about the TestStand User Interface (UI) Controls.

## TestStand User Interface Controls

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

### Creating and Configuring ActiveX Controls

In the LabWindows/CVI User Interface Editor, select **Create»ActiveX** and select a user interface control whose name begins 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 need to use the ActiveX driver for the control. The TestStand UI Controls driver and additional support instrument drivers are located in the `<TestStand>\API\CVI` directory. TestStand copies these files to the `<National Instruments>\Shared\CVI\instr\TestStand\API` directory for LabWindows/CVI 8.5 or later and to the `<CVI>\instr\TestStand\API\CVI` directory for LabWindows/CVI 8.1.1 or earlier.

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

- **TestStand UI Controls** (`tsui.fp`)—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.

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

## 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 accomplish this by extracting the ActiveXData parameter of the custom user message as a property object:

```
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 stop a user interface application by clicking the Close box or by executing the Exit command through a TestStand menu or a Button control. For user interface events that request the user interface to close, the user interface must call the `TSUI_ApplicationMgrShutdown` function to unload sequence files, log out, and trigger an `OnApplicationCanExit` event. 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.

## Menu Bars

The TestStand Utility Functions provide the following set of functions for creating and handling TestStand-specific menu items without requiring any additional code:

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

Use the `TS_InsertCommandsInMenu` function to create new menu items that execute commands you specify. To create menu items, you specify an array of command types, and the menu bar and menu IDs determine where to insert the commands. Each command type specifies a menu item or group of menu items to insert. You must also specify a handle to the Application Manager control, ExecutionView Manager control, or SequenceFileView Manager control to which the new menu items apply. TestStand uses a manager control to determine whether the menu item is visible or dimmed. TestStand installs a callback for each menu item that automatically invokes the associated command when the user selects 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.

## 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.

# 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 to see how 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 monitor whether an execution is terminating or aborting.

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

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

# 16

# Using the TestStand ActiveX APIs in LabWindows/CVI

You can use the TestStand API or TestStand 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 callbacks for events.

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

The LabWindows/CVI ActiveX Automation Library uses the CAObjHandle data type for handles to ActiveX objects. The TestStand ActiveX drivers also follow this convention, so you can use the CAObjHandle data type for all handles to TestStand objects. However, one drawback of using the same data type for all TestStand objects 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 defined in the ActiveX driver for those objects.

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

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

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

Error:
    return error;
}
```

The errorInfo variable is a structure the LabWindows/CVI ActiveX Automation Library defines to hold information about errors that can occur in the operation of the function. The `tsErrChk` macro determines whether the function 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 setting a local variable by calling a method of the `PropertyObject` class on a handle to a `SequenceContext` object:

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

    // Set local variable NumericValue to a random number

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

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

Error:
    return error;
}
```

# Adding and Releasing References

LabWindows/CVI automatically maintains an object reference for each handle you obtain for an object. 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 User Interface (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.

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 the function names exported from the DLL.

In addition, when you select a function in the DLL, the adapter queries the export information or the type library for the parameter list information and 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 enum parameters.
- Do not use structures that require forward references or that contain pointers.
- Do not use pointer types except when passing parameters by reference.

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

# 18 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. With 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 backward 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 `Step.Result.PassFail` property when the property exists. |
| measurement | double | Out | Numeric measurement the test function returns. The adapter copies this value into the `Step.Result.Numeric` property when the property exists. |
| inBuffer | char * | In | For passing a string parameter to a test function. The adapter copies the `Step.InBuf` 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 `Step.Result.ReportText` 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. |
| seqContextDisp | struct IDispatch * | In | Dispatch pointer to the sequence context. This value is `NULL` when you choose not to pass the sequence context. |

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

| Field Name | Data Type | In/ Out | Description |
|---|---|---|---|
| seqContextCVI | CAObjHandle | In | LabWindows/CVI ActiveX Automation handle for the sequence context. This value is `0` 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 `Step.Result.String` property when the property exists. |
| replaceStringFuncPtr | tReplaceStringPtr const | In | Contains a function pointer to `ReplaceString`, which a code module can use to reassign a value to the inBuffer, outBuffer, and errorMessage fields. The `ReplaceString` prototype is as follows:<br>`int ReplaceString(char **destString, char *srcString);`<br>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 `True` when an error occurs. The adapter copies this output value into the `Step.Result.Error.Occurred` property when the property exists. |
| errorLocation | tErrLoc (int) | Out | Reserved (no longer used). |

**Table 18-2.** tTestError Structure Member Fields (Continued)

| Field Name | Data Type | In/ Out | Description |
|---|---|---|---|
| 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 `Step.Result.Error.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 `Step.Result.Error.Msg` property when the property exists and the **errorFlag** is set. |

# Updating Step Properties

You can use the following two 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 Updated by LabWindows/CVI Adapter

| Structure Member | Valid Values Tests Can Return | Step.Result Property | Step Property Value |
|---|---|---|---|
| result | PASS or FAIL | PassFail | `True`/`False` |
| outBuffer | string value | ReportText | string value |

**Table 18-3.** Step Properties Updated by LabWindows/CVI Adapter (Continued)

| Structure Member | Valid Values Tests Can Return | Step.Result Property | Step Property 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:

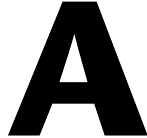
```
// Simple test example
#include "stdtst.h"
void TX_TEST __declspec(dllexport) FunctionName
    (tTestData *testData, tTestError *testError)
{
    int error = 0;
    double measurement = 5.0;
    char *lastUserName = NULL;

    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

Visit the following sections of the award-winning National Instruments Web site at `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 on demand training modules via the Services Resource Center. NI offers complementary membership for a full year after purchase, after which you may renew to continue your benefits.

    For information about other technical support options in your area, visit `ni.com/services`, or contact your local office at `ni.com/contact`.
- **Training and Certification**—Visit `ni.com/training` for self-paced training, eLearning virtual classrooms, interactive CDs, and Certification program information. You also can register for instructor-led, hands-on courses at locations around the world.
- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, National Instruments Alliance Partner members can help. To learn more, call your local NI office or visit `ni.com/alliance`.

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

# Index

## Numerics


64-bit integer data types in LabVIEW 8.0, 7-4


## A


ActiveX
- adding references (LabWindows/CVI), 16-5
- API, TestStand
  - LabVIEW, 9-1
  - LabWindows/CVI, 16-1
- releasing references
  - LabVIEW, 9-3
  - LabWindows/CVI, 16-5
- using LabWindows/CVI drivers, 16-1

ActiveX controls (LabWindows/CVI)
- configuring, 15-1
- creating, 15-1
- programming, 15-1

adapters. *See* module adapters

aliases file (LabVIEW), 7-3


## C


cluster parameters (LabVIEW), 4-4
- creating custom data types, 4-6
- creating TestStand data types, 4-8
- passing as container variable, 4-5
- specifying cluster elements individually, 4-5

code modules (LabWindows/CVI), 1-1
- C source files, 14-5
- calling from TestStand, 11-1
- creating from TestStand (tutorial), 12-1
- debugging
  - DLLs in-process, 14-5
  - from TestStand (tutorial), 12-3
  - in external instances, 14-3
- editing from TestStand (tutorial), 12-3
- executing
  - in external instance, 14-2
  - in-process, 14-3
- legacy, 18-1
- loading subordinate DLLs (tutorial), 14-5
- object files, 14-3
- object parameters, 13-3
- selecting where to execute, 14-2
- static library files, 14-3
- stopped, 15-8
- string parameters, 13-2
- structure parameters, 13-3
- suspended, 15-8

Code Template Policy
- LabVIEW, 5-5
- LabWindows/CVI, 14-7

COM references, duplicating (LabVIEW), 9-6

Conditional Disable Structures and symbols (LabVIEW), 7-4

constants
- LabVIEW, 9-4
- LabWindows/CVI, 16-7

container variables, passing to LabVIEW, 4-5

Controls palette (LabVIEW), 6-1

conventions used in the manual, *iv*


## D


data types (LabVIEW), 4-1
- creating custom for clusters, 4-6
- creating from clusters (tutorial), 4-8

data types (LabWindows/CVI), 13-1
- calling a function with a struct parameter, 13-5
- creating custom (tutorial), 13-4
- creating custom for struct parameters (tutorial), 13-4
- specifying structure passing settings (tutorial), 13-5

deployment, restrictions with LabVIEW 8.0, 7-5
diagnostic tools (NI resources), A-1
directory, 2-3
DLLs (LabWindows/CVI)
- adding type libraries, 17-1
- requirements for declaring in a type library, 17-2

documentation
- conventions used in the manual, *iv*
- NI resources, A-1

drivers (NI resources), A-1


## E


Editor Mode, 6-4, 15-4
enumerations
- LabVIEW, 9-4
- LabWindows/CVI, 16-7

Error Out cluster, legacy (LabVIEW), 10-3
events
- handling in user interfaces
  - LabVIEW, 6-4
  - LabWindows/CVI, 15-4, 16-8
- handling menu events in user interfaces (LabVIEW), 6-7

examples (NI resources), A-1
executions
- checking for suspended or stopped executions
  - LabVIEW, 6-9
  - LabWindows/CVI, 15-8
- remote execution in LabVIEW 8.0, 7-4
- setting preferred execution system (LabVIEW), 9-7


## F


function arguments, showing in step descriptions, 14-2


## I


Input Buffer string control, legacy (LabVIEW), 10-4
instrument drivers (NI resources), A-1
interfaces, obtaining for TestStand objects (LabVIEW), 9-5
Invocation Info cluster, legacy (LabVIEW), 10-4


## K


KnowledgeBase, A-1


## L


LabVIEW
- creating custom user interfaces, 6-1
- Module tab, 2-2
- object-oriented programming, 7-7
- required settings, 2-1
- RT Server, configuring, 8-3
- servers, selecting, 5-1
  - LabVIEW 8.6.x or later development system, 5-4
  - LabVIEW Run-Time Engine, 5-2
  - Other Executable, 5-2
- setting preferred execution system, 9-7
- TestStand ActiveX APIs, 9-1
- VI Server, configuring (tutorial), 8-2
- VIs. *See* VIs

LabVIEW 8.0, 7-1
64-bit integer data types, 7-4
aliases file, 7-3
Conditional Disable Structures and symbols, 7-4
deployments, building, 7-5
network-published shared variables, 7-2
aliases file, 7-3
deploying, 7-2
NI-DAQmx in projects, 7-3
project libraries, 7-2
projects, 7-1
real-time module incompatibility, 7-1
remote execution, 7-4
symbols, 7-4
XControls, 7-4
LabVIEW Adapter, 1-2
configuring, 5-1
creating and configuring new steps (tutorial), 2-3
per-step configuration, 5-4
setting preferred execution system, 9-7
LabWindows/CVI
creating custom user interfaces, 15-3
Module tab, 11-1
required settings, 11-1
TestStand ActiveX APIs, 16-1
LabWindows/CVI Adapter, 1-3
configuring, 14-2
creating and configuring new steps (tutorial), 11-4
per-step configuration, 14-6
selecting where steps execute, 14-2
legacy code modules (LabWindows/CVI), 18-1
example, 18-5
prototypes, 18-1
tTestData structure, 18-2
tTestError structure, 18-3
updating step properties, 18-4
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, 10-5
settings, 5-7
Test Data cluster, 10-2
localization
LabVIEW, 6-8
LabWindows/CVI, 15-8

## M

menu bars, user interfaces
LabVIEW, 6-7
LabWindows/CVI, 15-7
modal dialog boxes, user interfaces
LabVIEW, 6-8
LabWindows/CVI, 15-8
module adapters
LabVIEW, 1-2
configuring, 5-1
LabWindows/CVI, 1-3
configuring, 14-1
Module tab
LabVIEW, 2-2
LabWindows/CVI, 11-1

## N

National Instruments support and services, A-1
NI-DAQmx in projects in LabVIEW 8.0, 7-3

## O

object parameters (LabWindows/CVI), 13-3

## P

programming examples (NI resources), A-1
project libraries in LabVIEW 8.0, 7-2
projects in LabVIEW 8.0, 7-1
properties, accessing
  built-in TestStand properties
    LabVIEW, 9-1
    LabWindows/CVI, 16-3
  dynamic TestStand properties
    LabVIEW, 9-2
    LabWindows/CVI, 16-4
PropertyObject class, acquiring derived class in LabVIEW, 9-5
prototypes, legacy (LabWindows/CVI), 18-1

## R

real-time module incompatibility in LabVIEW 8.0, 7-1
remote computers (LabVIEW)
  calling VIs, 8-1
  configuring a LabVIEW RT Server, 8-3
  configuring and running steps (tutorial), 8-1
remote execution in LabVIEW 8.0, 7-4

## S

Sequence Context control, legacy (LabVIEW), 10-5
shared variables (LabVIEW), 7-2
  aliases file, 7-3
  deploying, 7-2
software (NI resources), A-1
step properties, updating (LabWindows/CVI), 18-4
step types, custom
  LabVIEW, 1-2
  LabWindows/CVI, 1-2
steps
  creating and configuring (tutorial)
    LabVIEW, 2-3
    LabWindows/CVI, 11-4
  showing function arguments (LabWindows/CVI), 14-2
  updating properties (LabWindows/CVI), 18-4
stopped executions
  LabVIEW, 6-9
  LabWindows/CVI, 15-8
string parameters
  LabWindows/CVI code modules, 13-2
  VIs, 4-4
struct parameters (LabWindows/CVI), 13-3
  calling functions (tutorial), 13-5
  creating custom data types (tutorial), 13-4
  setting default packing size, 14-2
suspended executions
  LabVIEW, 6-9
  LabWindows/CVI, 15-8
symbols in LabVIEW 8.0, 7-4

## T

technical support (NI Resources), A-1
Test Data cluster, legacy (LabVIEW), 10-2
TestStand
  ActiveX API
    LabVIEW, 9-1
    LabWindows/CVI, 16-1
  passing container variables to LabVIEW, 4-5
  shutting down in user interfaces
    LabVIEW, 6-6
    LabWindows/CVI, 15-6
  starting in user interfaces
    LabVIEW, 6-6
    LabWindows/CVI, 15-6
  <TestStand Public> directory, 2-3

TestStand API
accessing
built-in properties
LabVIEW, 9-1
LabWindows/CVI, 16-3
dynamic properties
LabVIEW, 9-2
LabWindows/CVI, 16-4
acquiring derived class from PropertyObject class in LabVIEW, 9-5
ActiveX drivers for LabWindows/CVI, 16-1
constants
LabVIEW, 9-4
LabWindows/CVI, 16-7
duplicating COM references (LabVIEW), 9-6
enumerations
LabVIEW, 9-4
LabWindows/CVI, 16-7
invoking methods
LabVIEW, 9-1
LabWindows/CVI, 16-2
LabVIEW, 9-1
LabWindows/CVI, 16-1
obtaining interfaces for TestStand objects (LabVIEW), 9-5
TestStand User Interface (UI) Controls
configuring
LabVIEW, 6-3
LabWindows/CVI, 15-4
LabVIEW, 6-1
LabWindows/CVI, 15-1
training and certification (NI resources), A-1
troubleshooting (NI resources), A-1
tTestData structure, legacy (LabWindows/CVI), 18-2
tTestError structure, legacy (LabWindows/CVI), 18-3
type libraries, generating in LabWindows/CVI (tutorial), 17-1

## U

user interfaces
checking for suspended or stopped executions
LabVIEW, 6-9
LabWindows/CVI, 15-8
creating
LabVIEW, 6-2
LabWindows/CVI, 15-3
custom
LabVIEW, 1-2, 6-1
LabWindows/CVI, 1-2, 15-1
Editor Mode, 6-4, 15-4
enabling sequence editing, 6-4, 15-4
handling events
LabVIEW, 6-4
LabWindows/CVI, 15-4, 16-8
handling menu events (LabVIEW), 6-7
handling variants (LabWindows/CVI), 15-5
localization
LabVIEW, 6-8
LabWindows/CVI, 15-8
menu bars
LabVIEW, 6-7
LabWindows/CVI, 15-7
modal dialog boxes
LabVIEW, 6-8
LabWindows/CVI, 15-8
running in LabVIEW, 6-9
shutting down TestStand
LabVIEW, 6-6
LabWindows/CVI, 15-6
starting TestStand
LabVIEW, 6-6
LabWindows/CVI, 15-6

## V

variables, deploying (LabVIEW), 7-2
variants, handling in user interfaces (LabWindows/CVI), 15-5
VI Server, configuring (tutorial), 8-2
VIs
- calling from TestStand, 2-1
- calling on remote computers, 8-1
- cluster parameters, 4-4
- code modules, 1-1
- configuring LabVIEW Adapter to run, 2-1
- creating from TestStand (tutorial), 3-1
- debugging from TestStand (tutorial), 3-3
- editing from TestStand (tutorial), 3-2
- LabVIEW Real-Time module, 8-1
- legacy, 10-1
- legacy settings, 5-7
- reserving, 5-4
- running VIs remotely with LabVIEW RT Server, 8-3
- running VIs remotely with VI Server, 8-2
- setting access list for remote access, 8-4
- setting preferred execution system, 9-7
- stopped, 6-9
- string parameters, 4-4
- suspended, 6-9

VIs and Functions palette (LabVIEW), 6-1

## W

Web resources (NI Resources), A-1

## X

XControls in LabVIEW 8.0, 7-4